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:

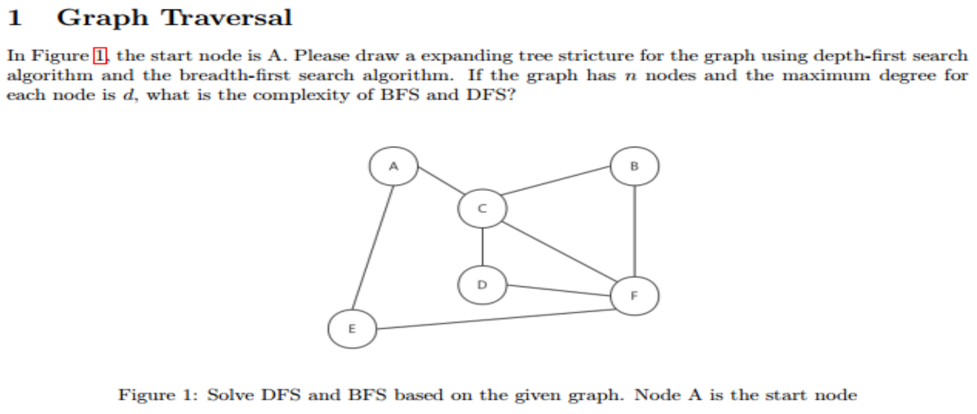
- breadth first search
- depth first search
- uniform cost search
- A star search

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

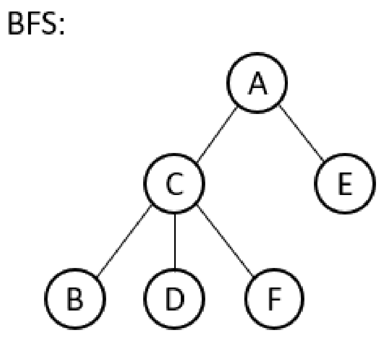


1 Graph Traversal

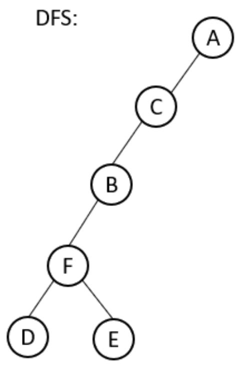
In Figure 1, the start node is A. Please draw a expanding tree stricture for the graph using depth-first search algorithm and the breadth-first search algorithm. If the graph has n nodes and the maximum degree for each node is d, what is the complexity of BFS and DFS?

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

(a) Assignment 1's description



(b) BFS's expanding tree



(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.

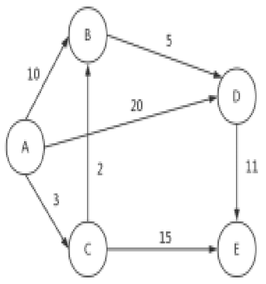
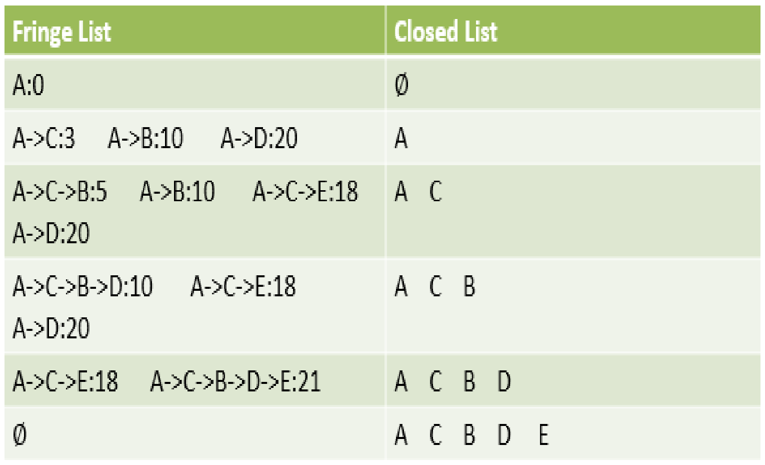


Figure 2: Uniform cost graph search problem

(a) Assignment 2's description



Fringe List	Closed List
A:0	Ø
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
Ø	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 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.

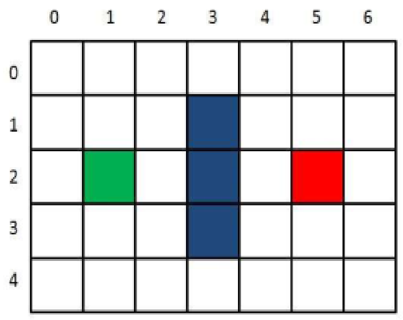
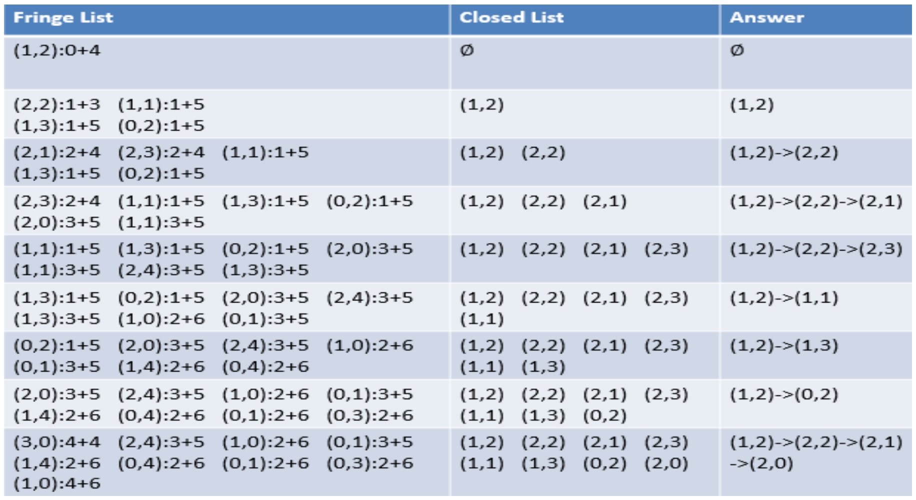


Figure 3: Use A* to solve for the maze

(a) Assignment 3's description

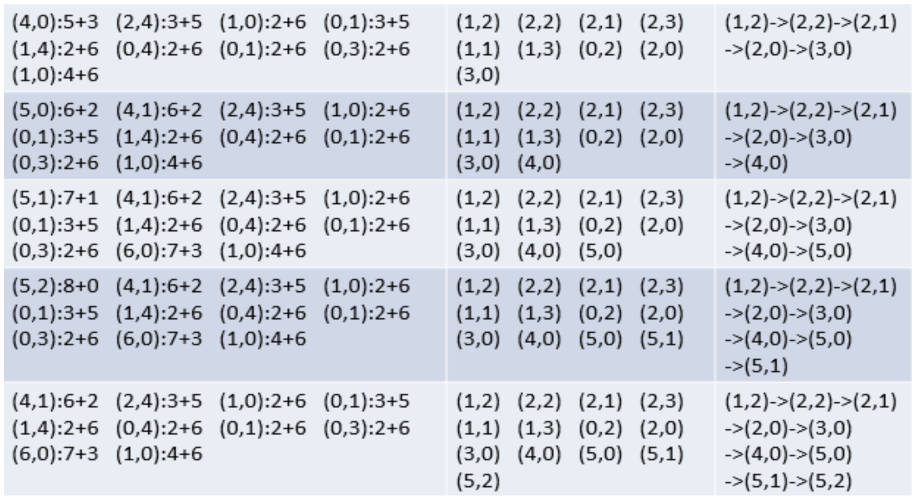
For this question, we can select the Manhattan Distance from the start to de goal as h heuristic function.

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	∅	∅
(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

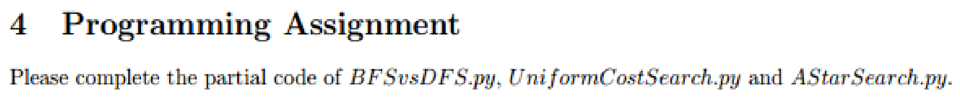


(4,0):5+3 (2,4):3+5 (1,0):2+6 (0,1):3+5 (1,4):2+6 (0,4):2+6 (0,1):2+6 (0,3):2+6 (1,0):4+6	(1,2) (2,2) (2,1) (2,3) (1,1) (1,3) (0,2) (2,0) (3,0)	(1,2)->(2,2)->(2,1) ->(2,0)->(3,0)
(5,0):6+2 (4,1):6+2 (2,4):3+5 (1,0):2+6 (0,1):3+5 (1,4):2+6 (0,4):2+6 (0,1):2+6 (0,3):2+6 (1,0):4+6	(1,2) (2,2) (2,1) (2,3)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

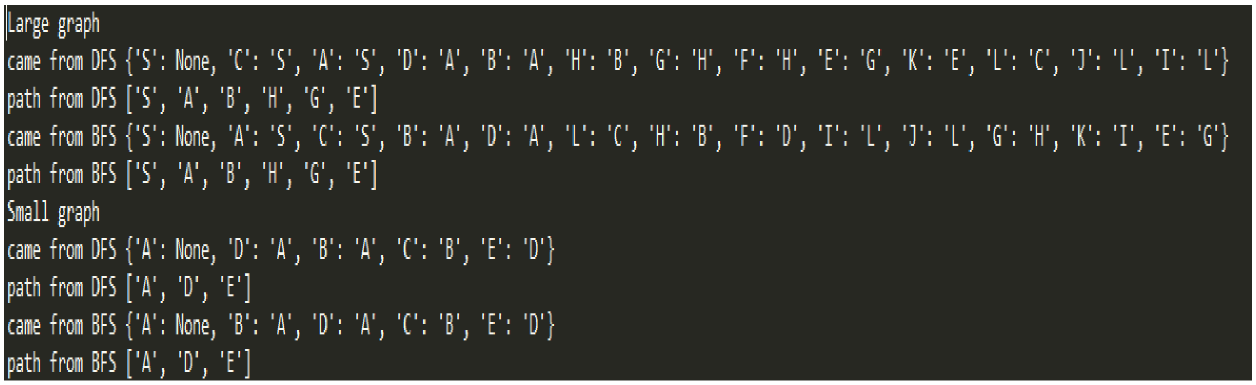
V. ASSIGNMENT 4



4 Programming Assignment

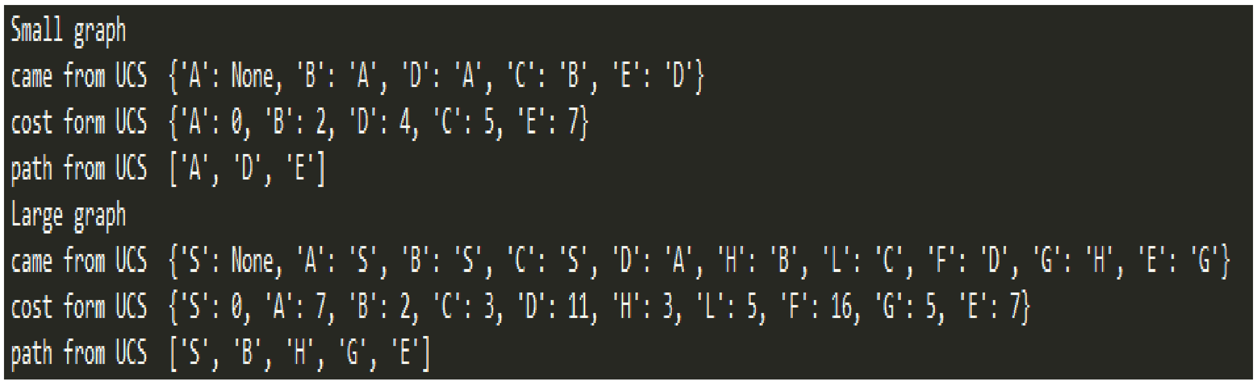
Please complete the partial code of *BFSvsDFS.py*, *UniformCostSearch.py* and *AStarSearch.py*.

(a) Assignment 4's description



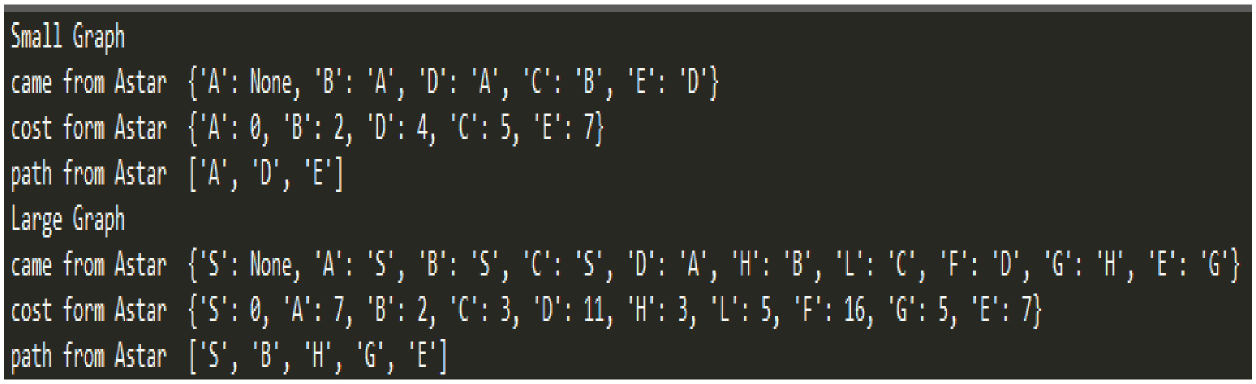
```
Large graph
came from DFS {'S': None, 'C': 'S', 'A': 'S', 'D': 'A', 'B': 'A', 'H': 'B', 'G': 'H', 'F': 'H', 'E': 'G', 'K': 'E', 'L': 'C', 'J': 'L', 'I': 'L'}
path from DFS ['S', 'A', 'B', 'H', 'G', 'E']
came from BFS {'S': None, 'A': 'S', 'C': 'S', 'B': 'A', 'D': 'A', 'L': 'C', 'H': 'B', 'F': 'D',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

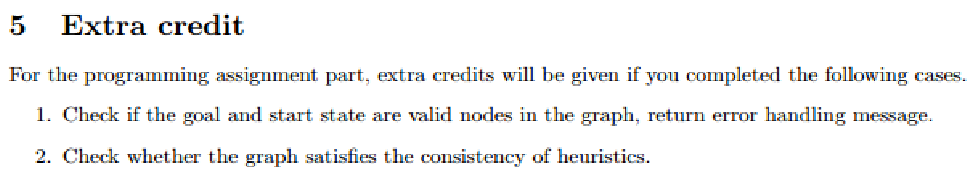


```
Small Graph
came from Astar  {'A': None, 'B': 'A', 'D': 'A', 'C': 'B', 'E': 'D'}
cost form Astar  {'A': 0, 'B': 2, 'D': 4, 'C': 5, 'E': 7}
path from Astar  ['A', 'D', 'E']
Large Graph
came from Astar  {'S': None, 'A': 'S', 'B': 'S', 'C': 'S', 'D': 'A', 'H': 'B', 'L': 'C', 'F': 'D', 'G': 'H', 'E':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

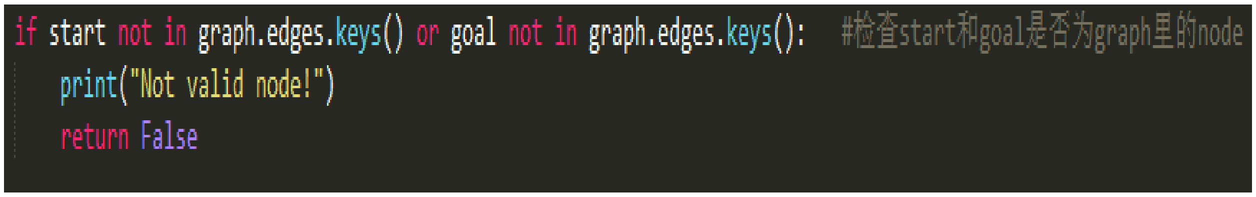


5 Extra credit

For the programming assignment part, extra credits will be given if you completed the following cases.

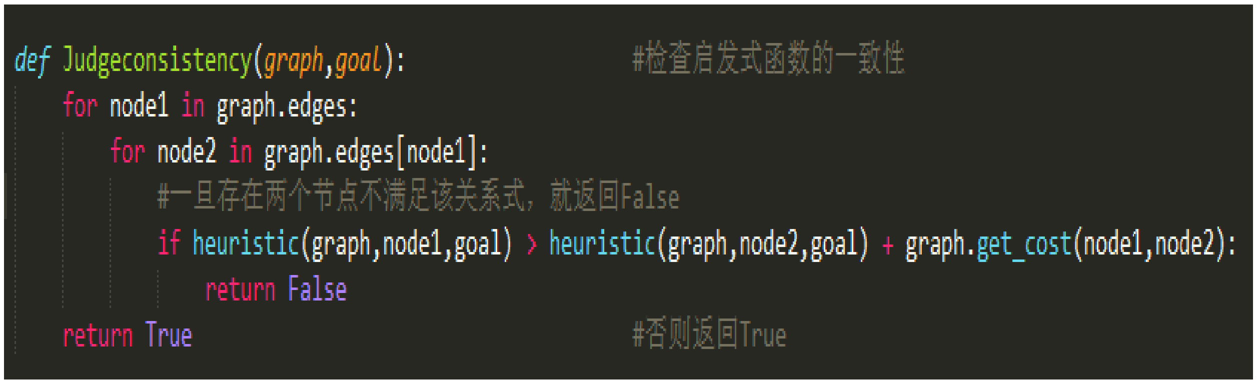
1. Check if the goal and start state are valid nodes in the graph, return error handling message.
2. Check whether the graph satisfies the consistency of heuristics.

(a) Assignment 5's description



```python
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.



```python
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

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

class Graph:
    """
␣␣␣␣Defines␣a␣graph␣with␣edges,␣each␣edge␣is␣treated␣as␣dictionary
␣␣␣␣look␣up.␣function␣neighbors␣pass␣in␣an␣id␣and␣returns␣a␣list␣of
␣␣␣␣neighboring␣node

␣␣␣␣"""
    def __init__(self):
        self.edges = {}

    def neighbors(self, id):
        # check if the edge is in the edge dictionary
        if id in self.edges:
            return self.edges[id]
        else:
            print("The␣node␣", id , "␣is␣not␣in␣the␣graph")
            return False


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 = []
    ### START CODE HERE ### (     6 line of code)
    path.append(goal)              #   goal
    while(goal != start):          #            goal   start
      goal = came_from[goal]
      path.append(goal)
    path.reverse()                 #
    ### END CODE HERE ###
    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
␣␣␣␣"""
    came_from = {}
    came_from[start] = None
    ### 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
        print("Not␣valid␣node!")
        return False

    fringe = Queue()                               #Fringe  List        FIFO
    closed = []                                    #Closed  List

    fringe.put(start)                              #    start
    closed.append(start)                           #   start
    while not fringe.empty():                      # Fringe
           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
     ### 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
␣␣␣␣"""
     came_from = {}
     came_from[start] = None
     ### 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
         print("Not␣valid␣node!")
         return False

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

     fringe.put(start)                                   #     start
     closed.append(start)                                #   start
     while not fringe.empty():                           # Fringe
         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
     ### END CODE HERE ###

     return came_from



# 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.
if __name__=="__main__":
     small_graph = Graph()
     small_graph.edges = {
         'A': ['B','D'],
         'B': ['A', 'C', 'D'],
         'C': ['A'],
         'D': ['E', 'A'],
         'E': ['B']
     }
     large_graph = Graph()
     large_graph.edges = {
         'S': ['A','C'],
         'A': ['S','B','D'],
         'B': ['S', 'A', 'D','H'],
         'C': ['S','L'],
         'D': ['A', 'B','F'],
         'E': ['G','K'],
         'F': ['H','D'],
         'G': ['H','E'],
         'H': ['B','F','G'],
         'I': ['L','J','K'],
         'J': ['L','I','K'],
         'K': ['I','J','E'],
         'L': ['C','I','J']
     }
     print("Large␣graph")
     start = 'S'
     goal = 'E'
     came_fromDFS = depth_first_search(large_graph, start, goal)
     print("came␣from␣DFS" , came_fromDFS)
     pathDFS = reconstruct_path(came_fromDFS, start, goal)
     print("path␣from␣DFS", pathDFS)
     came_fromBFS = breadth_first_search(large_graph, start, goal)
     print("came␣from␣BFS", came_fromBFS)
     pathBFS = reconstruct_path(came_fromBFS, start, goal)
     print("path␣from␣BFS", pathBFS)
```

```
    print("Small graph")
    start = 'A'
    goal = 'E'
    came_fromDFS = depth_first_search(small_graph, start, goal)
    print("came from DFS" , came_fromDFS)
    pathDFS = reconstruct_path(came_fromDFS, start, goal)
    print("path from DFS", pathDFS)
    came_fromBFS = breadth_first_search(small_graph, start, goal)
    print("came from BFS", came_fromBFS)
    pathBFS = reconstruct_path(came_fromBFS, start, goal)
    print("path from BFS", pathBFS)
```

B. code 2:UniformCostSearch

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

class Graph:
    """
    Defines a graph with edges, each edge is treated as dictionary
    look up. function neighbors pass in an id and returns a list of
    neighboring node

    """
    def __init__(self):
        self.edges = {}
        self.edgeWeights = {}
        self.locations = {}

    def neighbors(self, id):
        if id in self.edges:
            return self.edges[id]
        else:
            print("The node ", id , " is not in the graph")
            return False

    def get_node_location(self, id):
        return self.nodeLocation[id]

    def get_cost(self,from_node, to_node):
        #print("get cost for ", from_node, to_node)
        nodeList = self.edges[from_node]
        #print(nodeList)
        try:
            edgeList = self.edgeWeights[from_node]
            return edgeList[nodeList.index(to_node)]
        except ValueError:
            print("From node ", from_node, " to ", to_node, " does not exist a direct connection")
            return False

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 = []
    ### START CODE HERE ### (     6 line of code)
    path.append(goal)              #   goal
    while(goal != start):          #           goal   start
      goal = came_from[goal]
      path.append(goal)
    path.reverse()                 #
    ### 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
    ### START CODE HERE ### (     15 line of code)
    if start not in graph.edges.keys() or goal not in graph.edges.keys():   #
             s t a r t   g o a l       g r a p h     n o d e
        print("Not␣valid␣node!")
        return False

    fringe = PriorityQueue()                            #Fringe  L i s t
    closed = []                                         #Closed  L i s t

    fringe.put(start,cost_so_far[start])                #      s t a r t
    closed.append(start)                                #    s t a r t
    while not fringe.empty():                           #  F r i n g e   L i s t          g o a l
      current = fringe.get()
      if current == goal:                        #                          g o a l      g o a l            U C S
         break
      for child in graph.neighbors(current):       #

         if child not in closed:
            cost_so_far[child] = cost_so_far[current] + graph.get_cost(current,child)
            fringe.put(child,cost_so_far[child])
            closed.append(child)
            came_from[child] = current
    ### END CODE HERE ###
    return came_from, cost_so_far




# The main function will first create the graph, then use uniform cost search
# which will return the came_from dictionary
# then use the reconstruct path function to rebuild the path.
if __name__=="__main__":
    small_graph = Graph()
    small_graph.edges = {
        'A': ['B','D'],
        'B': ['A', 'C', 'D'],
        'C': ['A'],
        'D': ['E', 'A'],
        'E': ['B']
    }
    small_graph.edgeWeights={
        'A': [2,4],
        'B': [2, 3, 4],
        'C': [2],
        'D': [3, 4],
        'E': [5]
    }

    large_graph = Graph()
    large_graph.edges = {
        'S': ['A','B','C'],
        'A': ['S','B','D'],
        'B': ['S', 'A', 'D','H'],
        'C': ['S','L'],
        'D': ['A', 'B','F'],
        'E': ['G','K'],
        'F': ['H','D'],
        'G': ['H','E'],
        'H': ['B','F','G'],
        'I': ['L','J','K'],
        'J': ['L','I','K'],
        'K': ['I','J','E'],
        'L': ['C','I','J']
    }
    large_graph.edgeWeights = {
        'S': [7, 2, 3],
        'A': [7, 3, 4],
        'B': [2, 3, 4, 1],
        'C': [3, 2],
        'D': [4, 4, 5],
        'E': [2, 5],
        'F': [3, 5],
        'G': [2, 2],
        'H': [1, 3, 2],
        'I': [4, 6, 4],
        'J': [4, 6, 4],
        'K': [4, 4, 5],
        'L': [2, 4, 4]
    }

    print("Small␣graph")
    start = 'A'
    goal = 'E'
    came_from_UCS, cost_so_far = uniform_cost_search(small_graph, start, goal)
    print("came␣from␣UCS␣" , came_from_UCS)
    print("cost␣form␣UCS␣", cost_so_far)
    pathUCS = reconstruct_path(came_from_UCS, start, goal)
    print("path␣from␣UCS␣", pathUCS)

    print("Large␣graph")
    start = 'S'
    goal = 'E'
    came_from_UCS, cost_so_far = uniform_cost_search(large_graph, start, goal)
    print("came_from_UCS ", came_from_UCS)
    print("cost_form_UCS ", cost_so_far)
    pathUCS = reconstruct_path(came_from_UCS, start, goal)
    print("path_from_UCS ", pathUCS)
```

C.   code 3:AStarSearch

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

class Graph:
    """
    Defines a graph with edges, each edge is treated as dictionary
    look up. function neighbors pass in an id and returns a list of
    neighboring node

    """
    def __init__(self):
        self.edges = {}
        self.edgeWeights = {}
        self.locations = {}

    def neighbors(self, id):
        if id in self.edges:
            return self.edges[id]
        else:
            print("The node ", id , " is not in the graph")
            return False

    # this function get the g(n) the cost of going from from_node to
    # the to_node
    def get_cost(self,from_node, to_node):
        #print("get_cost_for ", from_node, to_node)
        nodeList = self.edges[from_node]
        #print(nodeList)
        try:
            edgeList = self.edgeWeights[from_node]
            return edgeList[nodeList.index(to_node)]
        except ValueError:
            print("From node ", from_node, " to ", to_node, " does not exist a direct connection")
            return False


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 = []
    ### START CODE HERE ### (   6 line of code)
    path.append(goal)                    #  goal
    while(goal != start):                #           goal  start
        goal = came_from[goal]
        path.append(goal)
    path.reverse()                       #
    ### END CODE HERE ###
    return path

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
    """
    heuristic_value = 0
    ### START CODE HERE ### (   15 line of code)
    heuristic_value = abs(graph.locations[current_node][0] - graph.locations[goal_node][0])\
                      + abs(graph.locations[current_node][1] - graph.locations[goal_node][1])
                                                                                    #
    ### END CODE HERE ###
    return heuristic_value

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

    ### START CODE HERE ### (     15 line of code)
    if start not in graph.edges.keys() or goal not in graph.edges.keys():   #
             s t a r t   g o a l       g r a p h     n o d e
        print("Not_valid_node!")
        return False

    fringe = PriorityQueue()                          #Fringe  L i s t
    closed = []                                       #Closed  L i s t

    fringe.put(start,cost_so_far[start] + heuristic(graph,start,goal))         #
              s t a r t                    g    n    + h    n
    closed.append(start)                              #   s t a r t
    while not fringe.empty():                         #  F r i n g e   L i s t          g o a l
        current = fringe.get()
        if current == goal:                           #                          g o a l      g o a l            A
             s t a r
            break
        for child in graph.neighbors(current):        #

            if child not in closed:
                cost_so_far[child] = cost_so_far[current] + graph.get_cost(current,child)
                fringe.put(child,cost_so_far[child] + heuristic(graph,child,goal))
                closed.append(child)
                came_from[child] = current
    ### END CODE HERE ###
    return came_from, cost_so_far

def Judgeconsistency(graph,goal):                     #
    for node1 in graph.edges:
        for node2 in graph.edges[node1]:
            #                                                        F a l s e
            if heuristic(graph,node1,goal) > heuristic(graph,node2,goal) + graph.get_cost(node1,node2):
                return False
    return True                                       #          T r u e

# The main function will first create the graph, then use A* search
# which will return the came_from dictionary
# then use the reconstruct path function to rebuild the path.
if __name__=="__main__":
    small_graph = Graph()
    small_graph.edges = {
        'A': ['B','D'],
        'B': ['A', 'C', 'D'],
        'C': ['A'],
        'D': ['E', 'A'],
        'E': ['B']
    }
    small_graph.edgeWeights={
        'A': [2,4],
        'B': [2, 3, 4],
        'C': [2],
        'D': [3, 4],
        'E': [5]
    }
    small_graph.locations={
        'A': [4,4],
        'B': [2,4],
        'C': [0,0],
        'D': [6,2],
        'E': [8,0]
    }

    large_graph = Graph()
    large_graph.edges = {
        'S': ['A','B','C'],
        'A': ['S','B','D'],
        'B': ['S', 'A', 'D','H'],
        'C': ['S','L'],
        'D': ['A', 'B','F'],
        'E': ['G','K'],
        'F': ['H','D'],
        'G': ['H','E'],
        'H': ['B','F','G'],
        'I': ['L','J','K'],
        'J': ['L','I','K'],
        'K': ['I','J','E'],
        'L': ['C','I','J']
    }
    large_graph.edgeWeights = {
        'S': [7, 2, 3],
        'A': [7, 3, 4],
        'B': [2, 3, 4, 1],
        'C': [3, 2],
        'D': [4, 4, 5],
        'E': [2, 5],
        'F': [3, 5],
        'G': [2, 2],
        'H': [1, 3, 2],
        'I': [4, 6, 4],
        'J': [4, 6, 4],
        'K': [4, 4, 5],
        'L': [2, 4, 4]
    }

    large_graph.locations = {
        'S': [0, 0],
        'A': [-2,-2],
        'B': [1,-2],
        'C': [6,0],
        'D': [0,-4],
        'E': [6,-8],
        'F': [1,-7],
        'G': [3,-7],
        'H': [2,-5],
        'I': [4,-4],
        'J': [8,-4],
        'K': [6,-7],
        'L': [7,-3]
    }
    print("Small Graph")
    start = 'A'
    goal = 'E'
    came_from_Astar, cost_so_far = A_star_search(small_graph, start, goal)
    print("came from Astar " , came_from_Astar)
    print("cost form Astar ", cost_so_far)
    pathAstar = reconstruct_path(came_from_Astar, start, goal)
    print("path from Astar ", pathAstar)


    print("Large Graph")
    start = 'S'
    goal = 'E'
    came_from_Astar, cost_so_far = A_star_search(large_graph, start, goal)
    print("came from Astar " , came_from_Astar)
    print("cost form Astar ", cost_so_far)
    pathAstar = reconstruct_path(came_from_Astar, start, goal)
    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!