AU 332 Artificial Intelligence: Principles and Techniques

By: Chi Zhang (517021910658)

HW#: 1

September 23, 2019

I. INTRODUCTION

A. Purpose

- Question 1: Draw a expanding tree stricture for the graph using depth-first search algorithm and the breadth-first search algorithm
- Question 2: Calculate the shortest path from node A to E using UCS method. Write step by step update of the fringe list and closed list.
- Question 3: Write out the complete path finding process from the green grid to the red grid using A* algorithm

B. Equipment

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

- Python 3.7.0 (Anaconda)

C. Procedure

1. Problem 1

The expanding tree stricture for the graph is shown in FIG.1. The schematic form for depth-first search algorithm is shown in FIG.2 and the schematic form for breadth-first search algorithm is shown in FIG.3.

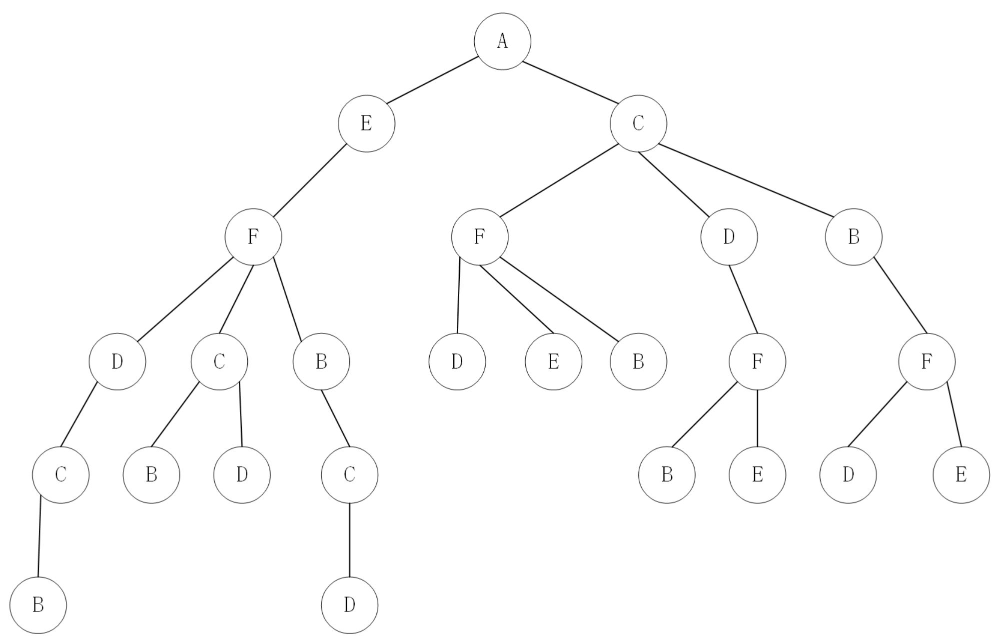

FIG. 1: expanding tree stricture

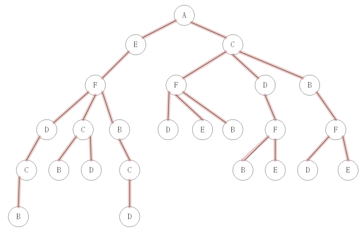

FIG. 2: depth-first search algorithm

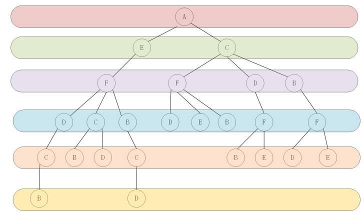

FIG. 3: breadth-first search algorithm

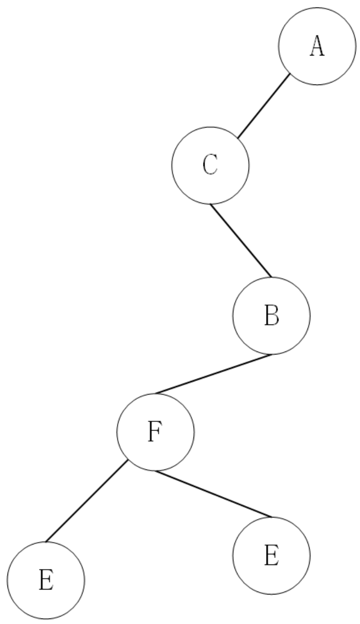

FIG. 4: depth-first search tree

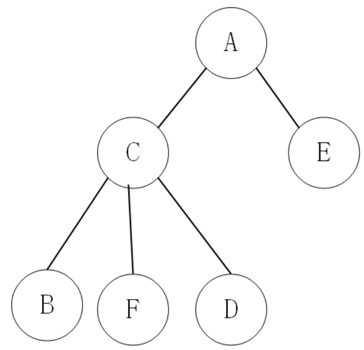

FIG. 5: breadth-first search tree

Question: If the graph has n nodes and the maximum degree for each node is d, what is the complexity of BFS and DFS?

If the graph has n nodes and the maximum degree for each node is d, the time complexity of BFS and DFS is both $O(nd)$. And the space complexity of BFS and DFS is both $O(n)$.

2. Problem 2

- Step 1:
  - Expand: A
  - Fringe List: A-C(3), A-B(10), A-D(20)
  - Closed Set: A
- Step 2:
  - Expand: A-C
  - Fringe List: A-C-B(5), A-B(10), A-C-E(18), A-D(20)
  - Closed Set: A, A-C
- Step 3:
  - Expand: A-C-B
  - Fringe List: A-C-B-D(10), A-B(10), A-C-E(18), A-D(20)
  - Closed Set: A, A-C, A-C-B
- Step 4:
  - Expand: A-C-B-D
  - Fringe List: A-C-B-D-E(21), A-B(10), A-C-E(18), A-D(20)
  - Closed Set: A, A-C, A-C-B, A-C-B-D
- and we can obtain the shortest path A-C-B-D-E, which lenth is 21 from A to E.

3. Problem 3

As the heuristic h is admissible if $0 \leq h(n) \leq h^*(n)$. The *Manhattan distance* is chosen as heuristic h. For the grid (x, y), h is given by:

$$h(n) = |x - 5| + |y - 2|$$

FIG. shows the value of $h(n)$ of each grid and the value of $g(n)$ of each grid. To determine the path finding process, I write the code *Q3* which uses A* algorithm. The code is included in its entirety in Appendix. With the help of A* algorithm, the route is chosen by:

- step 1 [1, 2]

  fringe list [(4, [[1, 2], [2, 2], 1]), (6, [[1, 2], [0, 2], 1]), (6, [[1, 2], [1, 3], 1]), (6, [[1, 2], [1, 1], 1])]

  closed set [[1, 2], [1, 2]]
- step 2 [2, 2]

  fringe list [(6, [[1, 2], [0, 2], 1]), (6, [[1, 2], [1, 1], 1]), (6, [[1, 2], [1, 3], 1]), (6, [[2, 2], [2, 1], 2]), (6, [[2, 2], [2, 3], 2])]

  closed set [[1, 2], [1, 2], [2, 2]]
- step 3 [0, 2]

  fringe list [(6, [[1, 2], [1, 1], 1]), (6, [[2, 2], [2, 1], 2]), (6, [[1, 2], [1, 3], 1]), (6, [[2, 2], [2, 3], 2]), (8, [[0, 2], [0, 1], 2]), (8, [[0, 2], [0, 3], 2])]

  closed set [[1, 2], [1, 2], [2, 2], [0, 2]]

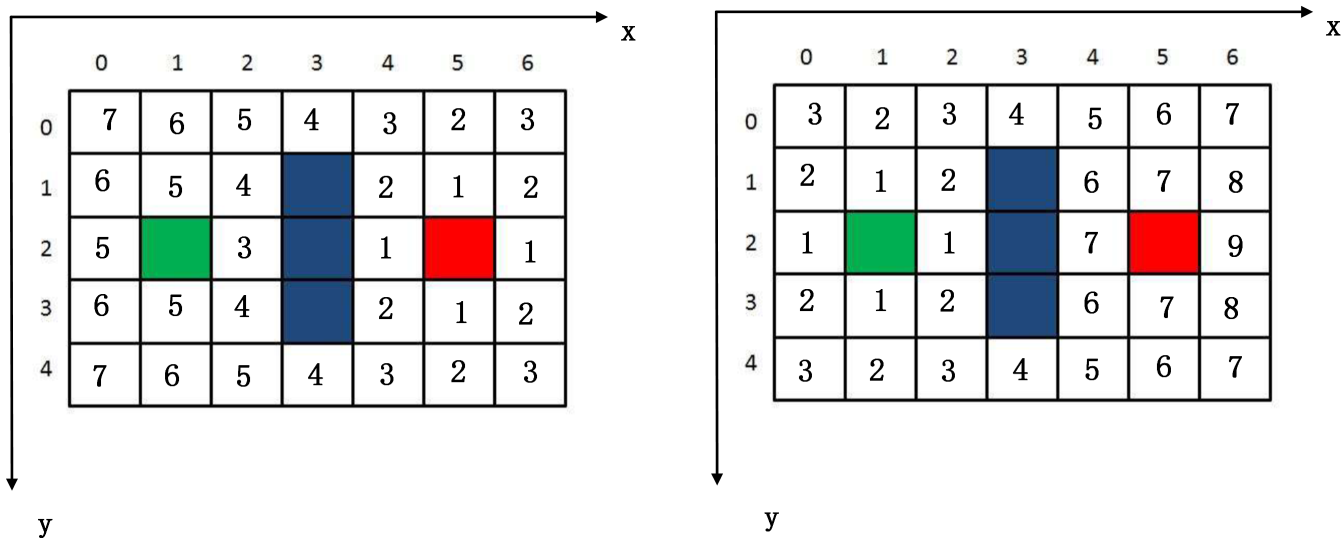

FIG. 6: the value of $h(n)$ and $g(n)$ of each grid

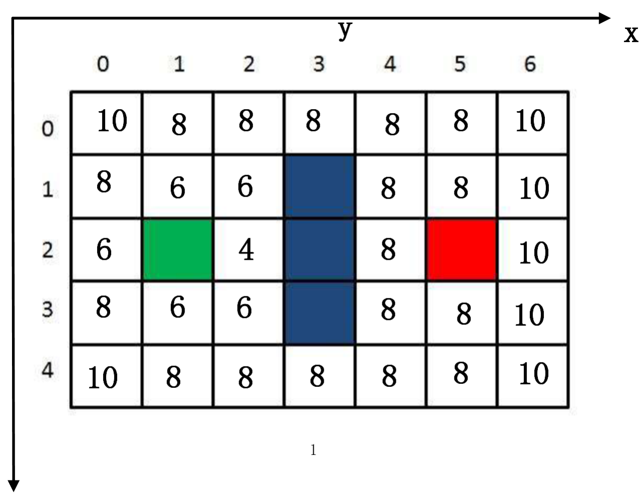

FIG. 7: the value of $f(n)$ of each grid

- step 4 [1, 1]

  fringe list [(6, [[1, 1], [2, 1], 2]), (6, [[1, 2], [1, 3], 1]), (8, [[0, 2], [0, 3], 2]), (6, [[2, 2], [2, 1], 2]), (8, [[0, 2], [0, 1], 2]), (8, [[1, 1], [0, 1], 2]), (8, [[1, 1], [1, 0], 2]), (6, [[2, 2], [2, 3], 2])]

  closed set [[1, 2], [1, 2], [2, 2], [0, 2], [1, 1]]

- step 5 [2, 1]

  fringe list [(6, [[1, 2], [1, 3], 1]), (6, [[2, 2], [2, 1], 2]), (8, [[0, 2], [0, 3], 2]), (6, [[2, 2], [2, 3], 2]), (8, [[0, 2], [0, 1], 2]), (8, [[1, 1], [0, 1], 2]), (8, [[1, 1], [1, 0], 2]), (8, [[2, 1], [2, 0], 3])]

  closed set [[1, 2], [1, 2], [2, 2], [0, 2], [1, 1], [2, 1]]

- step 6 [1, 3]

fringe list [(6, [[1, 3], [2, 3], 2]), (6, [[2, 2], [2, 1], 2]), (8, [[0, 2], [0, 3], 2]), (8, [[1, 3], [0, 3], 2]), (6, [[2, 2], [2, 3], 2]), (8, [[1, 1], [0, 1], 2]), (8, [[1, 1], [1, 0], 2]), (8, [[2, 1], [2, 0], 3]), (8, [[1, 3], [1, 4], 2]), (8, [[0, 2], [0, 1], 2])]

closed set [[1, 2], [1, 2], [2, 2], [0, 2], [1, 1], [2, 1], [1, 3]]

- step 7 [2, 3]

  fringe list [(6, [[2, 2], [2, 1], 2]), (6, [[2, 2], [2, 3], 2]), (8, [[0, 2], [0, 3], 2]), (8, [[1, 3], [0, 3], 2]), (8, [[0, 2], [0, 1], 2]), (8, [[1, 1], [0, 1], 2]), (8, [[1, 1], [1, 0], 2]), (8, [[2, 1], [2, 0], 3]), (8, [[1, 3], [1, 4], 2]), (8, [[2, 3], [2, 4], 3])]

  closed set [[1, 2], [1, 2], [2, 2], [0, 2], [1, 1], [2, 1], [1, 3], [2, 3]]

- step 8 [0, 1]

  fringe list [(8, [[0, 2], [0, 3], 2]), (8, [[1, 3], [0, 3], 2]), (8, [[1, 1], [0, 1], 2]), (8, [[1, 3], [1, 4], 2]), (8, [[2, 3], [2, 4], 3]), (8, [[2, 1], [2, 0], 3]), (8, [[1, 1], [1, 0], 2]), (10, [[0, 1], [0, 0], 3])]

  closed set [[1, 2], [1, 2], [2, 2], [0, 2], [1, 1], [2, 1], [1, 3], [2, 3], [0, 1]]

- step 9 [0, 3]

  fringe list [(8, [[1, 1], [0, 1], 2]), (8, [[1, 3], [0, 3], 2]), (8, [[1, 1], [1, 0], 2]), (8, [[1, 3], [1, 4], 2]), (8, [[2, 3], [2, 4], 3]), (8, [[2, 1], [2, 0], 3]), (10, [[0, 1], [0, 0], 3]), (10, [[0, 3], [0, 4], 3])]

  closed set [[1, 2], [1, 2], [2, 2], [0, 2], [1, 1], [2, 1], [1, 3], [2, 3], [0, 1], [0, 3]]

- step 10 [1, 0]

  fringe list [(8, [[1, 0], [2, 0], 3]), (8, [[1, 3], [0, 3], 2]), (8, [[2, 1], [2, 0], 3]), (8, [[1, 3], [1, 4], 2]), (8, [[2, 3], [2, 4], 3]), (10, [[0, 3], [0, 4], 3]), (10, [[1, 0], [0, 0], 3]), (10, [[0, 1], [0, 0], 3])]

  closed set [[1, 2], [1, 2], [2, 2], [0, 2], [1, 1], [2, 1], [1, 3], [2, 3], [0, 1], [0, 3], [1, 0]]

- step 11 [2, 0]

  fringe list [(8, [[1, 3], [0, 3], 2]), (8, [[1, 3], [1, 4], 2]), (8, [[2, 1], [2, 0], 3]), (8, [[2, 0], [3, 0], 4]), (8, [[2, 3], [2, 4], 3]), (10, [[0, 3], [0, 4], 3]), (10, [[1, 0], [0, 0], 3]), (10, [[0, 1], [0, 0], 3])]

  closed set [[1, 2], [1, 2], [2, 2], [0, 2], [1, 1], [2, 1], [1, 3], [2, 3], [0, 1], [0, 3], [1, 0], [2, 0]]

- step 12 [1, 4]

  fringe list [(8, [[1, 4], [2, 4], 3]), (8, [[2, 0], [3, 0], 4]), (8, [[2, 1], [2, 0], 3]), (8, [[2, 3], [2, 4], 3]), (10, [[1, 0], [0, 0], 3]), (10, [[0, 3], [0, 4], 3]), (10, [[1, 4], [0, 4], 3]), (10, [[0, 1], [0, 0], 3])]

  closed set [[1, 2], [1, 2], [2, 2], [0, 2], [1, 1], [2, 1], [1, 3], [2, 3], [0, 1], [0, 3], [1, 0], [2, 0], [1, 4]]

- step 13 [2, 4]

  fringe list [(8, [[2, 0], [3, 0], 4]), (8, [[2, 3], [2, 4], 3]), (8, [[2, 1], [2, 0], 3]), (8, [[2, 4], [3, 4], 4]), (10, [[1, 0], [0, 0], 3]), (10, [[0, 3], [0, 4], 3]), (10, [[1, 4], [0, 4], 3]), (10, [[0, 1], [0, 0], 3])]

  closed set [[1, 2], [1, 2], [2, 2], [0, 2], [1, 1], [2, 1], [1, 3], [2, 3], [0, 1], [0, 3], [1, 0], [2, 0], [1, 4], [2, 4]]

- step 14 [3, 0]

  fringe list [(8, [[2, 1], [2, 0], 3]), (8, [[2, 3], [2, 4], 3]), (10, [[0, 1], [0, 0], 3]), (8, [[2, 4], [3, 4], 4]), (10, [[1, 0], [0, 0], 3]), (10, [[0, 3], [0, 4], 3]), (10, [[1, 4], [0, 4], 3]), (8, [[3, 0], [4, 0], 5])]

  closed set [[1, 2], [1, 2], [2, 2], [0, 2], [1, 1], [2, 1], [1, 3], [2, 3], [0, 1], [0, 3], [1, 0], [2, 0], [1, 4], [2, 4], [3, 0]]

- step 15 [3, 4]

  fringe list [(8, [[3, 0], [4, 0], 5]), (10, [[0, 3], [0, 4], 3]), (8, [[3, 4], [4, 4], 5]), (10, [[1, 4], [0, 4], 3]), (10, [[1, 0], [0, 0], 3]), (10, [[0, 1], [0, 0], 3])]

  closed set [[1, 2], [1, 2], [2, 2], [0, 2], [1, 1], [2, 1], [1, 3], [2, 3], [0, 1], [0, 3], [1, 0], [2, 0], [1, 4], [2, 4], [3, 0], [3, 4]]

- step 16 [4, 0]

  fringe list [(8, [[3, 4], [4, 4], 5]), (10, [[0, 3], [0, 4], 3]), (8, [[4, 0], [4, 1], 6]), (10, [[1, 4], [0, 4], 3]), (10, [[1, 0], [0, 0], 3]), (10, [[0, 1], [0, 0], 3]), (8, [[4, 0], [5, 0], 6])]

  closed set [[1, 2], [1, 2], [2, 2], [0, 2], [1, 1], [2, 1], [1, 3], [2, 3], [0, 1], [0, 3], [1, 0], [2, 0], [1, 4], [2, 4], [3, 0], [3, 4], [4, 0]]

- step 17 [4, 4]

  fringe list [(8, [[4, 0], [4, 1], 6]), (8, [[4, 4], [5, 4], 6]), (8, [[4, 0], [5, 0], 6]), (10, [[0, 3], [0, 4], 3]), (10, [[1, 0], [0, 0], 3]), (10, [[0, 1], [0, 0], 3]), (8, [[4, 4], [4, 3], 6]), (10, [[1, 4], [0, 4], 3])]

  closed set [[1, 2], [1, 2], [2, 2], [0, 2], [1, 1], [2, 1], [1, 3], [2, 3], [0, 1], [0, 3], [1, 0], [2, 0], [1, 4], [2, 4], [3, 0], [3, 4], [4, 0], [4, 4]]

- step 18 [4, 1]

  fringe list [(8, [[4, 0], [5, 0], 6]), (8, [[4, 1], [4, 2], 7]), (8, [[4, 4], [4, 3], 6]), (8, [[4, 1], [5, 1], 7]), (10, [[1, 0], [0, 0], 3]), (10, [[0, 1], [0, 0], 3]), (10, [[1, 4], [0, 4], 3]), (10, [[0, 3], [0, 4], 3]), (8, [[4, 4], [5, 4], 6])]

  closed set [[1, 2], [1, 2], [2, 2], [0, 2], [1, 1], [2, 1], [1, 3], [2, 3], [0, 1], [0, 3], [1, 0], [2, 0], [1, 4], [2, 4], [3, 0], [3, 4], [4, 0], [4, 4], [4, 1]]

- step 19 [5, 0]

  fringe list [(8, [[4, 1], [4, 2], 7]), (8, [[4, 1], [5, 1], 7]), (8, [[4, 4], [4, 3], 6]), (8, [[4, 4], [5, 4], 6]), (10, [[1, 0], [0, 0], 3]), (10, [[0, 1], [0, 0], 3]), (10, [[1, 4], [0, 4], 3]), (10, [[0, 3], [0, 4], 3]), (8, [[5, 0], [5, 1], 7]), (10, [[5, 0], [6, 0], 7])]

  closed set [[1, 2], [1, 2], [2, 2], [0, 2], [1, 1], [2, 1], [1, 3], [2, 3], [0, 1], [0, 3], [1, 0], [2, 0], [1, 4], [2, 4], [3, 0], [3, 4], [4, 0], [4, 4], [4, 1], [5, 0]]

- step 20 [4, 2]

  fringe list [(8, [[4, 1], [5, 1], 7]), (8, [[4, 2], [5, 2], 8]), (8, [[4, 4], [4, 3], 6]), (8, [[5, 0], [5, 1], 7]), (8, [[4, 4], [5, 4], 6]), (10, [[0, 1], [0, 0], 3]), (10, [[1, 4], [0, 4], 3]), (10, [[0, 3], [0, 4], 3]), (10, [[5, 0], [6, 0], 7]), (10, [[4, 2], [4, 3], 8]), (10, [[1, 0], [0, 0], 3])]

  closed set [[1, 2], [1, 2], [2, 2], [0, 2], [1, 1], [2, 1], [1, 3], [2, 3], [0, 1], [0, 3], [1, 0], [2, 0], [1, 4], [2, 4], [3, 0], [3, 4], [4, 0], [4, 4], [4, 1], [5, 0], [4, 2]]

- step 21 [5, 1]

  fringe list [(8, [[4, 2], [5, 2], 8]), (8, [[4, 4], [5, 4], 6]), (8, [[4, 4], [4, 3], 6]), (8, [[5, 0], [5, 1], 7]), (8, [[5, 1], [5, 2], 8]), (10, [[0, 1], [0, 0], 3]), (10, [[1, 4], [0, 4], 3]), (10, [[0, 3], [0, 4], 3]), (10, [[5, 0], [6, 0], 7]), (10, [[4, 2], [4, 3], 8]), (10, [[1, 0], [0, 0], 3]), (10, [[5, 1], [6, 1], 8])]

  closed set [[1, 2], [1, 2], [2, 2], [0, 2], [1, 1], [2, 1], [1, 3], [2, 3], [0, 1], [0, 3], [1, 0], [2, 0], [1, 4], [2, 4], [3, 0], [3, 4], [4, 0], [4, 4], [4, 1], [5, 0], [4, 2], [5, 1]]

- step 22 [5, 2]

  final path ['(1,2)', '(1,1)', '(1,0)', '(2,0)', '(3,0)', '(4,0)', '(4,1)', '(4,2)', '(5,2)']

4. Problem 4

- *BFSvsDFS.py*

  The file provides the class *Graph*, which includes a dictionary *edges* and function *neighbors*. *edges* is treated as dictionary to look up. Function *neighbors* pass in an id and returns a list of neighboring node

– function *reconstruct_path(came_from, start, goal)*

Given a dictionary named *came_from*. The key of the dictionary is the node character and its value is the parent node, the start node and the goal node, the function will compute the path from start to the end.

To accomplish this function, I start with *goal* because each node might have more than one children but one father node only. With the *came_from* provided, I can search for the father node of every node. Start with *goal* and append its father to the *path* list. Loop untill *start* appears as a node's father. At last, reverse the list and return it as result.

```
def reconstruct_path(came_from, start, goal):
    path = []
    ### START CODE HERE ###  ( 6 line of code)
    if came_from is None:
        print("Path reconstruction failed!!")
        return

    current_node = goal
    while(current_node is not start):
        path.append(current_node)
        current_node = came_from[current_node]
    path.append(current_node)
    path.reverse()

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

– function *breadth_first_search(graph, start, goal)*

Given a graph, a start node and a goal node, the function utilizes breadth first search algorithm to find the path from start node to the goal node. It will return a dictionary whose key is each node and corresponding value is its parent node

To accomplish this function, I use *queue* to expand the search region. The elements in the queue are stored as format *[father, child]*. Firstly put the *[None, start]* into the queue, While the queue is not empty, get the element from the head of queue as variable *Expand*. Add the father and child of *Expand* into the *came_from* dictionary. To realize **early stoping** mentioned in the code, I check if *goal* neighbors to the child of *Expand*. If so, add the goal to the dictionary as child of child in *Expand*, then return *came_from*.

It needs to be aware that for a gragh search, you should never expand a state twice. So I set the list *closed_set* to store the node that has been put into the queue. Before putting any node into the queue, check the node if it is in the *closed_set*.

```
def breadth_first_search(graph, start, goal):
    came_from = {}
    came_from[start] = None
    ### START CODE HERE ###  ( 10 line of code)

    ## check goal and start ============================
    if not ((goal in graph.edges) and (start in graph.edges) ):
        print("Error! Goal or Start is not in the graph")
        return None
    ##=================================================

    closed_set = [start]
    BFS_queue = Queue(maxsize=0)
    BFS_queue.put([None, start])
    while not BFS_queue.empty():
        Expand = BFS_queue.get()
        came_from[Expand[1]] = Expand[0]
        if goal in graph.neighbors(Expand[1]):
            came_from[goal] = Expand[1]
            return came_from
        for value in graph.neighbors(Expand[1]):
            if value not in closed_set:
                BFS_queue.put([Expand[1], value])
                closed_set.append(value)

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

– function *depth_first_search(graph, start, goal)*

Given a graph, a start node and a goal node, the function utilizes depth first search algorithm to find the path from start node to the goal node. It will return a dictionary whose key is each node and corresponding value is its parent node

To accomplish this function, I use *stack* to expand the search region. The elements in the stack are stored as format *[father, child]*. Firstly put the *[None, start]* into the stack, While the stack is not empty, get the element from the head of stack as variable *Expand*. Add the father and child of *Expand* into the *came_from* dictionary. To realize **early stoping** mentioned in the code, I check if *goal* neighbors to the child of *Expand*. If so, add the goal to the dictionary as child of child in *Expand*, then return *came_from*.

It needs to be aware that for a gragh search, you should never expand a state twice. So I set the list *closed_set* to store the node that has been put into the stack. Before putting any node into the stack, check the node if it is in the *closed_set*.

```
def depth_first_search(graph, start, goal):
    came_from = {}
    came_from[start] = None
    ### START CODE HERE ###  ( 10 line of code)

    ## check goal and start ============================
    if not ((goal in graph.edges) and (start in graph.edges) ):
        print("Error! Goal or Start is not in the graph")
        return
    ##==================================================

    closed_set = [start]
    DFS_queue = LifoQueue(maxsize=0)
    DFS_queue.put([None, start])
    while not DFS_queue.empty():
        Expand = DFS_queue.get()
        came_from[Expand[1]] = Expand[0]
        if goal in graph.neighbors(Expand[1]):
            came_from[goal] = Expand[1]
            return came_from
        for value in graph.neighbors(Expand[1]):
            if value not in closed_set:
                DFS_queue.put([Expand[1], value])
                closed_set.append(value)

    ### END CODE HERE ###

    return came_from
```

The demonstrations for functions will be shown in part **II EXPERIMENT**

- *UniformCostSearch.py*

  The file provides the class *Graph*, which includes the dictionary *edges*, the dictionary *edgeWeights*, the function *neighbors* and the function *get_cost*. *edges* and *edgeWeights* are treated as dictionary to look up. Pass in an id to function *neighbors*, it will return a list of neighboring node, while *get_cost* accepts two adjacent nodes and returns the cost.

  – function *reconstruct_path(came_from, start, goal)*

    Given a dictionary named *came_from*. The key of dictionary is the node character and its value is the parent node, the start node and the goal node, the function will compute the path from start to the end.

    To accomplish this function, I start with *goal* because each node might have more than one children but one father node only. With the *came_from* provided, I can search for the father node of every node. Start with *goal* and append its father to the *path* list. Loop untill *start* appears as a node's father. At last, reverse the list and return it as result.

```
def reconstruct_path(came_from, start, goal):
    path = []
    ### START CODE HERE ###  ( 6 line of code)
    if came_from is None:
        print("Path reconstruction failed!!")
        return

    current_node = goal
    while(current_node is not start):
```

```
            path.append(current_node)
            current_node = came_from[current_node]
        path.append(current_node)
        path.reverse()

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

– function *uniform_cost_search*

Given a graph, a start node and a goal node, the function utilizes uniform cost search algorithm to find the path from start node to the goal node. It will return two dictionaries, whose key is each node and corresponding value is its parent node, and whose key is each node and corresponding value is the cost from start to the node.

To accomplish this function, I use *PriorityQueue* to expand the search region. The elements in the PriorityQueue are stored as format *[value,[father, child]]*. Firstly put the *[0,[None, start]]* into the PriorityQueue. While the PriorityQueue is not empty, get the element from the head of PriorityQueue as variable *Expand*. Add the father and child of *Expand* into the *came_from* dictionary, then add the child and value of *Expand* into the *cost_so_far*. To realize **early stoping** mentioned in the code, I check if *goal* neighbors to the child of *Expand*. If so, add the goal to the dictionary as child of child in *Expand*, then return *came_from* and *cost_so_far*.

It needs to be aware that for a gragh search, you should never expand a state twice. So I set the list *closed_set* to store the node that has been put into the PriorityQueue. Before putting any node into the PriorityQueue, check the node if it is in the *closed_set*.

```
    def uniform_cost_search(graph, start, goal):

        came_from = {}
        cost_so_far = {}
        came_from[start] = None
        cost_so_far[start] = 0
        ### START CODE HERE ### ( 15 line of code)

        ### q.put((num,value)), smaller the num is, higher the priority is
        ### the cost_so_far should store the value that the nodes cost from start
        closed_set = [start]
        UCS_queue = PriorityQueue(maxsize=0)
        UCS_queue.put((0, [None, start]))
        while not UCS_queue.empty():
            ### Expand[1][1] is current node, Expand[1][0] is its father
            ### Expand[0] is the cost that from start to the Expand[1][1]
            Expand = UCS_queue.get()
            came_from[Expand[1][1]] = Expand[1][0]
            cost_so_far[Expand[1][1]] = Expand[0]
            if goal in graph.neighbors(Expand[1][1]):
                came_from[goal] = Expand[1][1]
                cost_so_far[goal] = graph.get_cost(Expand[1][1], goal) + Expand[0]
                return came_from, cost_so_far
            for value in graph.neighbors(Expand[1][1]):
                if value not in closed_set:
                    UCS_queue.put((graph.get_cost(Expand[1][1], value)+Expand[0], [Expand[1][1], value]))
                    closed_set.append(value)

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

The demonstration for functions will be shown in part **II EXPERIMENT**

- *AStarSearch.py*

The file provides the class *Graph*, which includes the dictionary *edges*, the dictionary *edgeWeights*, the function *neighbors* and the function *get_cost*. *edges* and *edgeWeights* are treated as dictionary to look up. Pass in an id to function *neighbors*, it will return a list of neighboring node, while *get_cost* accepts two adjacent nodes and returns the cost.

– function *reconstruct_path(came_from, start, goal)*

The function is same as the previous *reconstruct_path(came_from, start, goal)*.

– function *def heuristic(graph, current_node, goal_node)*

I choose *Euclidean distance* between a node and the goal as the heuristic of the node. To check whether the graph satisfies the consistency of heuristics, in the function *heuristic*, I import the function *uniform_cost_search* to compare the heuristic and the least cost of the node. If the

```
def heuristic(graph, current__node, goal__node):

  heuristic__value = 0
  ### START CODE HERE ###  ( 15 line of code)

  from UniformCostSearch import uniform__cost__search
  import math

  came__from__UCS, _ = uniform__cost__search(graph, current__node, goal__node)
  path = reconstruct__path(came__from__UCS, current__node, goal)

  actural__distance = 0
  for i in range(len(path)-1):
    actural__distance += graph.get__cost(path[i],path[i+1])

  heuristic__value = math.sqrt( (graph.locations[current__node][0] - graph.locations[goal__node][0])**2 + (graph.locations[
        current__node][1] - graph.locations[goal__node][1])**2)

  if actural__distance < heuristic__value:
    print("graph doesn't satisfies the consistency of heuristics")

  ### END CODE HERE ###
  return heuristic__value
```

If heuristic value is more than actral distance, it means that graph doesn't satisfies the consistency of heuristics

– function *A_star_search*

Given a graph, a start node and a goal node, the function utilizes A* search algorithm to find the path from start node to the goal node. It will return two dictionaries, whose key is each node and corresponding value is its parent node, and whose key is each node and corresponding value is the cost from start to the node.

To accomplish this function, I use *PriorityQueue* to expand the search region. The elements in the PriorityQueue are stored as format *[value,[father, child, CostToChild]]*. The *value* computed by adding the cost from start to the current node and the heuristic of the current node. The *CostToChild* is the cost from start to the child node. Firstly put the *[0+heuristic(graph,start,goal), [None, start,0]]* into the PriorityQueue. While the PriorityQueue is not empty, get the element from the head of PriorityQueue as variable *Expand*. Add the father and child of *Expand* into the *came_from* dictionary, then add the child and CostToChild of *Expand* into the *cost_so_far*. To realize **early stoping** mentioned in the code, I check if *goal* neighbors to the child of *Expand*. If so, add the goal to the dictionary as child of child in *Expand*, then return *came_from* and *cost_so_far*.

```
def A__star__search(graph, start, goal):
  came__from = {}
  cost__so__far = {}
  came__from[start] = None
  cost__so__far[start] = 0

  ### START CODE HERE ###  ( 15 line of code)

  closed__set = set(start)
  Astar__queue = PriorityQueue(maxsize=0)
  Astar__queue.put((0 + heuristic(graph,start,goal), [None, start, 0]))
  while not Astar__queue.empty():
    # Expand[1][1] is current node, Expand[1][0] is its father
    # Expand[0] is the value of h()+g()
    # Expand[1][2] is the cost that from start to the Expand[1][1]
    Expand = Astar__queue.get()
    closed__set.add(Expand[1][1])
    came__from[Expand[1][1]] = Expand[1][0]
    cost__so__far[Expand[1][1]] = Expand[1][2]

    if goal in graph.neighbors(Expand[1][1]):
      came__from[goal] = Expand[1][1]
      cost__so__far[goal] = graph.get__cost(Expand[1][1], goal) + Expand[1][2]
      return came__from, cost__so__far
    for value in graph.neighbors(Expand[1][1]):
      if value not in closed__set:
        Astar__queue.put((graph.get__cost(Expand[1][1], value)+Expand[1][2] + heuristic(graph,value,goal),
                          [Expand[1][1], value, graph.get__cost(Expand[1][1], value)+Expand[1][2] ]))
  ### END CODE HERE ###
  return came__from, cost__so__far
```

The demonstration for functions will be shown in part **II EXPERIMENT**

II. EXPERIMENT

This section consists of screenshots taken during the laboratory procedure.

A. *BFSvsDFS.py*

For large graph, the search starts from S to E. For small graph, the search starts from A to E.

```
small_graph.edges = {
    'A': ['B','D'],
    'B': ['A', 'C', 'D'],
    'C': ['A'],
    'D': ['E', 'A'],
    'E': ['B']
}
```

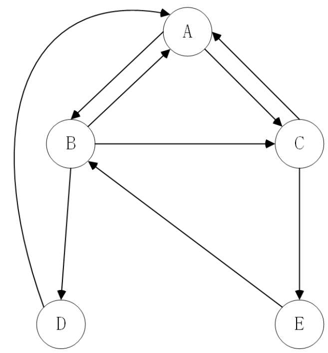

FIG. 8: small graph

```
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']
}
```

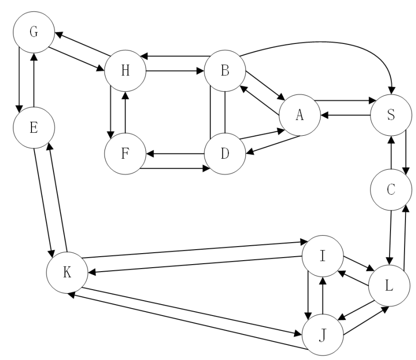

FIG. 9: large graph

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

FIG. 10: result for BFS and DFS

B. *UniformCostSearch.py*

For large graph, the search starts from S to H. For small graph, the search starts from A to E.

```
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]
}
```

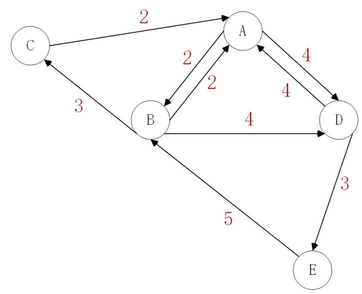

FIG. 11: small graph

```
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]
}
```

FIG. 12: large graph

C. *AStarSearch.py*

The graphs are same as the previous graphs.

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

FIG. 13: result for UCS

```
Small Graph
graph doesn't satisfies the consistency of heuristics
came from Astar  {'A': None, 'D': 'A', 'E': 'D'}
cost form Astar  {'A': 0, 'D': 4, 'E': 7}
path from Astar  ['A', 'D', 'E']
Large Graph
graph doesn't satisfies the consistency of heuristics
graph doesn't satisfies the consistency of heuristics
graph doesn't satisfies the consistency of heuristics
graph doesn't satisfies the consistency of heuristics
graph doesn't satisfies the consistency of heuristics
graph doesn't satisfies the consistency of heuristics
came from Astar  {'S': None, 'B': 'S', 'H': 'B', 'G': 'H', 'E': 'G'}
cost form Astar  {'S': 0, 'B': 2, 'H': 3, 'G': 5, 'E': 7}
path from Astar  ['S', 'B', 'H', 'G', 'E']
```

FIG. 14: result for A*

III. DISCUSSION & CONCLUSION

Homework1 is more difficult than HW0. I really spent lots of time on the homework. During the process of learning, I made many mistakes, which help me further understand the algorithms involved. The most inspiring thing is that I successfully solve the A* Search problem in Q3 by writing code. The process is pretty hard but I enjoyed it.

IV. APPENDIX

A. code of *Q3.py*

```
from queue import PriorityQueue
import math

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.locations = {}

def heuristic(graph, node, target = [5,2]):
    distance = abs(graph.locations['({},{})'.format(node[0],node[1])][0]-target[0]) + abs(graph.locations['({},{})'.format(node[0],node[1])][1]-target[1])
    return distance
```

```python
def reconstruct_path(came_from, start, goal):

    path = []
    current_node = goal
    while(came_from[current_node] is not None):
        path.append(current_node)
        # print(current_node)
        if came_from[current_node] is None :
            break
        current_node = '({},{})'.format( came_from[current_node][0],came_from[current_node][1])
    path.append(current_node)
    path.reverse()
    return path

def AstarSearch(graph, start = [1,2], target = [5,2]):
    came_from = {} # key is child and value is father
    cost_so_far = {}

    came_from['({},{})'.format(start[0],start[1])] = None
    cost_so_far['({},{})'.format(start[0],start[1])] = 0

    closed_set = [start]
    Astar_queue = PriorityQueue(maxsize=0)
    Astar_queue.put((0 + heuristic(graph,start), [None, start, 0]))

    step = 0

    while not Astar_queue.empty():
        Expand = Astar_queue.get()
        if Expand[1][1] in closed_set and step is not 0:
            continue

        step+=1
        print('\item '+"step "+ str(step))
        print(Expand[1][1])
        print('\n')

        closed_set.append(Expand[1][1])
        came_from['({},{})'.format(Expand[1][1][0],Expand[1][1][1])] = Expand[1][0]
        cost_so_far['({},{})'.format(Expand[1][1][0],Expand[1][1][1])] = Expand[1][2]

        if (Expand[1][1][0] is target[0]) and (Expand[1][1][1] is target[1]) :
            return came_from,cost_so_far

        for i in [-1,0,1]:
            for j in [-1,0,1]:
                if abs(i+j) is not 1 :
                    continue
                next_node = [Expand[1][1][0]+i, Expand[1][1][1]+j]
                if (next_node not in closed_set) and ( '({},{})'.format(next_node[0],next_node[1]) in graph.locations):
                    Astar_queue.put((1+Expand[1][2]+heuristic(graph,next_node), [Expand[1][1], next_node, 1+Expand[1][2]]))

        print('fringe list')
        print(Astar_queue.queue)
        print('\n')
        print('closed set')
        print(closed_set)
        print('\n')

    return came_from,cost_so_far


if __name__=="__main__":
    graph = Graph()
    for i in range (0,7):
        for j in range (0,5):
            graph.locations['({},{})'.format(i,j)] = [i, j]

    del graph.locations['(3,1)']
    del graph.locations['(3,2)']
    del graph.locations['(3,3)']

    came_from, cost_so_far = AstarSearch(graph)

    path = reconstruct_path(came_from,"(1,2)","(5,2)")

    print('final path ')
    print(path)
```