18/02/22

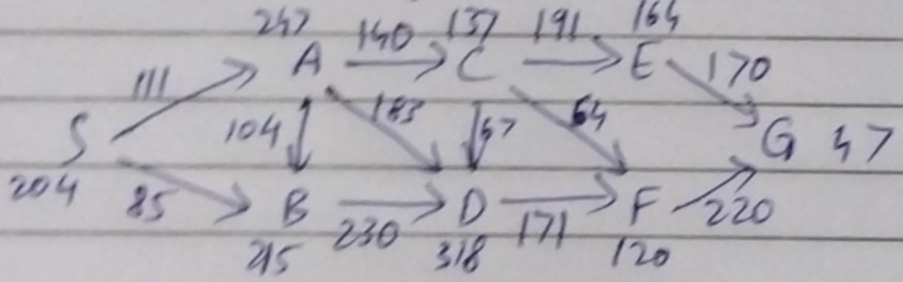
Experiment-5

Best First Search (BFS) and A* search

Aim: To execute ~~&~~ Best First Search and A* search.

Algorithm:
1) Create two empty lists open & closed.
2) Start from initial node and put it in open list
3) Repeat following steps till goal reached
   i) If open is empty, return false and exit list
   ii) Select first node from open and move it to closed.
   iii) Expand ~~no~~ first node to generate immediate node next to it and add them to open list.
   iv) Reorder nodes in open list in ascending order.

Graph:



Result: Best First Search and A* search have been successfully executed.

Experiment 5 - Best First Search and A* search

Dhawal Patil
RA1911003010575
CSE A2

Code (Best First Search)

```
class Graph:
    def __init__(self, graph_dict=None, directed=True):
        self.graph_dict = graph_dict or {}
        self.directed = directed
        if not directed:
            self.make_undirected()
    def make_undirected(self):
        for a in list(self.graph_dict.keys()):
            for (b, dist) in self.graph_dict[a].items():
                self.graph_dict.setdefault(b, {})[a] = dist
    def connect(self, A, B, distance=1):
        self.graph_dict.setdefault(A, {})[B] = distance
        if not self.directed:
            self.graph_dict.setdefault(B, {})[A] = distance
    def get(self, a, b=None):
        links = self.graph_dict.setdefault(a, {})
        if b is None:
            return links
        else:
            return links.get(b)
    def nodes(self):
        s1 = set([k for k in self.graph_dict.keys()])
        s2 = set([k2 for v in self.graph_dict.values() for k2, v2 in v.items()])
        nodes = s1.union(s2)
        return list(nodes)
class Node:
    def __init__(self, name:str, parent:str):
        self.name = name
        self.parent = parent
        self.g = 0
        self.h = 0
        self.f = 0
    def __eq__(self, other):
        return self.name == other.name
    def __lt__(self, other):
         return self.f < other.f
    def __repr__(self):
        return ('({0},{1})'.format(self.position, self.f))
def best_first_search(graph, heuristics, start, end):
    open = []
    closed = []
    start_node = Node(start, None)
    goal_node = Node(end, None)
    open.append(start_node)
    while len(open) > 0:
        open.sort()
        current_node = open.pop(0)
        closed.append(current_node)
        if current_node == goal_node:
            path = []
            while current_node != start_node:
```

```
            path.append(current_node.name + ': ' + str(current_node.g))
            current_node = current_node.parent
        path.append(start_node.name + ': ' + str(start_node.g))
        return path[::-1]
    neighbors = graph.get(current_node.name)
    for key, value in neighbors.items():
        neighbor = Node(key, current_node)
        if(neighbor in closed):
            continue
        neighbor.g = current_node.g + graph.get(current_node.name, neighbor.name)
        neighbor.h = heuristics.get(neighbor.name)
        neighbor.f = neighbor.h
        if(add_to_open(open, neighbor) == True):
            open.append(neighbor)
  return None
def add_to_open(open, neighbor):
  for node in open:
    if (neighbor == node and neighbor.f >= node.f):
      return False
  return True
graph = Graph()
graph.connect('S', 'A', 111)
graph.connect('S', 'B', 85)
graph.connect('A', 'B', 104)
graph.connect('A', 'C', 140)
graph.connect('A', 'D', 183)
graph.connect('B', 'D', 230)
graph.connect('C', 'D', 67)
graph.connect('C', 'E', 191)
graph.connect('C', 'F', 64)
graph.connect('D', 'F', 171)
graph.connect('E', 'G', 170)
graph.connect('F', 'G', 220)
graph.make_undirected()
heuristics = {}
heuristics['S'] = 204
heuristics['A'] = 247
heuristics['B'] = 215
heuristics['C'] = 137
heuristics['D'] = 318
heuristics['E'] = 164
heuristics['F'] = 120
heuristics['G'] = 47
path = best_first_search(graph, heuristics, 'S', 'G')
print(path)
print()
```

Code (A* Search)

```
from collections import deque
class Graph:
  def __init__(self, adjac_lis):
    self.adjac_lis = adjac_lis
  def get_neighbors(self, v):
    return self.adjac_lis[v]
  def h(self, n):
    H = {
      'S': 1,
```

```
            'A': 1,
            'B': 1,
            'C': 1,
            'D': 1,
            'E': 1,
            'F': 1,
            'G': 1,
        }
        return H[n]
    def a_star_algorithm(self, start, stop):
        open_lst = set([start])
        closed_lst = set([])
        poo = {}
        poo[start] = 0
        par = {}
        par[start] = start
        while len(open_lst) > 0:
            n = None
            for v in open_lst:
                if n == None or poo[v] + self.h(v) < poo[n] + self.h(n):
                    n = v;
            if n == None:
                print('Path does not exist!')
                return None
            if n == stop:
                reconst_path = []
                while par[n] != n:
                    reconst_path.append(n)
                    n = par[n]
                reconst_path.append(start)
                reconst_path.reverse()
                print('Path found: {}'.format(reconst_path))
                return reconst_path
            for (m, weight) in self.get_neighbors(n):
                if m not in open_lst and m not in closed_lst:
                    open_lst.add(m)
                    par[m] = n
                    poo[m] = poo[n] + weight
                else:
                    if poo[m] > poo[n] + weight:
                        poo[m] = poo[n] + weight
                        par[m] = n
                        if m in closed_lst:
                            closed_lst.remove(m)
                            open_lst.add(m)
            open_lst.remove(n)
            closed_lst.add(n)
        print('Path does not exist!')
        return None
adjac_lis = {
    'S': [('A', 111),('B', 85)],
    'A': [('B', 104), ('C', 140), ('D', 183)],
    'B': [('D', 230)],
    'C': [('D', 67),('E', 191),('F', 64)],
    'D': [('F', 171)],
    'E': [('G', 170)],
    'F': [('G', 220)]
```

```
}
graph1 = Graph(adjac_lis)
graph1.a_star_algorithm('S', 'G')
```

Screenshot

Output (Best First Search)

```
['S: 0', 'A: 111', 'C: 251', 'F: 315', 'G: 535']
```

Output (A* Search)

```
Path found: ['S', 'A', 'C', 'F', 'G']
```