Ex 5
Implementation of Best first search and A* Algorithm for real-world problems

3/3/22

Experiment 5

Implementation of Best First search and A* Algorithm.

Best First Search

Aim → The idea of Best First search is to use an evaluation function to decide which adjacent is most promising then explore in BFS and DFS.

ALGORITHM

1) Create an empty Priority Queue Priority Queue pq;
2) Insert "start" in pq. pq.insert(start)
3) Until PriorityQue is empty

    u = PriorityQue.Delete min

    ~~if u is the goal~~

    ~~Exit~~

    Else

    Foreach neighbor

30)

A* Algorithm

Aim → To approximate the shortest path in real life situation, like - in maps, games where there can be many hindrances.

ALGORITHM

1. Initialize the open list
2. Initialize the closed list put the starting node on the open list (you can leave its ~~fise~~ f at zero).
3. While the open list is not empty
   - a) Find the node let's say q.
   - b) pop q off the open list
   - c) generate q's 8 successors and set their parents.
   - d) For each successor, if successor is the goal, stop search.
   - e) push q on the closed list

   end (while loop)

```python
# This class represent a graph
class Graph:
    # Initialize the class
    def __init__(self, graph_dict=None, directed=True):
        self.graph_dict = graph_dict or {}
        self.directed = directed
        if not directed:
            self.make_undirected()
    # Create an undirected graph by adding symmetric edges
    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
    # Add a link from A and B of given distance, and also add the inverse link if
the graph is undirected
    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
    # Get neighbors or a neighbor
    def get(self, a, b=None):
        links = self.graph_dict.setdefault(a, {})
        if b is None:
            return links
        else:
            return links.get(b)
    # Return a list of nodes in the graph
    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)
# This class represent a node
class Node:
    # Initialize the class
    def __init__(self, name:str, parent:str):
        self.name = name
        self.parent = parent
        self.g = 0 # Distance to start node
```

```python
        self.h = 0 # Distance to goal node
        self.f = 0 # Total cost
    # Compare nodes
    def __eq__(self, other):
        return self.name == other.name
    # Sort nodes
    def __lt__(self, other):
         return self.f < other.f
    # Print node
    def __repr__(self):
        return ('({0},{1})'.format(self.position, self.f))
# Best-first search
def best_first_search(graph, heuristics, start, end):

    # Create lists for open nodes and closed nodes
    open = []
    closed = []
    # Create a start node and an goal node
    start_node = Node(start, None)
    goal_node = Node(end, None)
    # Add the start node
    open.append(start_node)

    # Loop until the open list is empty
    while len(open) > 0:
        # Sort the open list to get the node with the lowest cost first
        open.sort()
        # Get the node with the lowest cost
        current_node = open.pop(0)
        # Add the current node to the closed list
        closed.append(current_node)

        # Check if we have reached the goal, return the path
        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 reversed path
            return path[::-1]
        # Get neighbours
        neighbors = graph.get(current_node.name)
        # Loop neighbors
        for key, value in neighbors.items():
            # Create a neighbor node
            neighbor = Node(key, current_node)
            # Check if the neighbor is in the closed list
            if(neighbor in closed):
                continue
            # Calculate cost to goal
            neighbor.g = current_node.g + graph.get(current_node.name,
neighbor.name)
            neighbor.h = heuristics.get(neighbor.name)
            neighbor.f = neighbor.h
            # Check if neighbor is in open list and if it has a lower f value
            if(add_to_open(open, neighbor) == True):
                # Everything is green, add neighbor to open list
                open.append(neighbor)
    # Return None, no path is found
    return None
# Check if a neighbor should be added to open list
def add_to_open(open, neighbor):
    for node in open:
        if (neighbor == node and neighbor.f >= node.f):
            return False
    return True

def main():

    graph = Graph()

    graph.connect('A', 'B', 111)
    graph.connect('A', 'C', 85)
    graph.connect('B', 'C', 104)
    graph.connect('B', 'D', 140)
    graph.connect('B', 'E', 183)
    graph.connect('C', 'F', 230)
```

```python
    graph.connect('C', 'G', 67)
    graph.connect('G', 'H', 191)
    graph.connect('G', 'D', 64)
    graph.connect('F', 'E', 171)


    graph.make_undirected()

    heuristics = {}

    heuristics['A'] = 215
    heuristics['G'] = 137

    heuristics['C'] = 164

    heuristics['F'] = 132

    heuristics['D'] = 75

    heuristics['B'] = 153

    heuristics['E'] = 0

    path = best_first_search(graph, heuristics, 'A', 'E')
    print(path)
    print()

if __name__ == "__main__": main()
```

```
Run    Command: RA1911033010060/exp5/bestfirst.py

['A: 0', 'B: 111', 'E: 294']


Process exited with code: 0
```

```python
from collections import deque
```

```python
class Graph:


    def __init__(self, adjacency_list):
        self.adjacency_list = adjacency_list

    def get_neighbors(self, v):
        return self.adjacency_list[v]

    def h(self, n):
        H = {
            'A': 1,
            'B': 1,
            'C': 1,
            'D': 1
        }

        return H[n]

    def a_star_algorithm(self, start_node, stop_node):

        open_list = set([start_node])
        closed_list = set([])


        g = {}

        g[start_node] = 0


        parents = {}
        parents[start_node] = start_node

        while len(open_list) > 0:
            n = None


            for v in open_list:
                if n == None or g[v] + self.h(v) < g[n] + self.h(n):
```

```
            n = v;

        if n == None:
            print('Path does not exist!')
            return None

        if n == stop_node:
            reconst_path = []

            while parents[n] != n:
                reconst_path.append(n)
                n = parents[n]

            reconst_path.append(start_node)

            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_list and m not in closed_list:
                open_list.add(m)
                parents[m] = n
                g[m] = g[n] + weight

            else:
                if g[m] > g[n] + weight:
                    g[m] = g[n] + weight
                    parents[m] = n

                    if m in closed_list:
                        closed_list.remove(m)
                        open_list.add(m)

        open_list.remove(n)
        closed_list.add(n)
```

```python
        print('Path does not exist!')
        return None

adjacency_list = {
    'A': [('B', 1), ('C', 3), ('D', 7)],
    'B': [('D', 5)],
    'C': [('D', 12)]
}
graph1 = Graph(adjacency_list)
graph1.a_star_algorithm('A', 'D')
```

Run Command: RA1911033010060/exp5/astar.py

```
Path found: ['A', 'B', 'D']


Process exited with code: 0
```