asignment_02_arad_to_bucharest_bfs_dfs_ucs_gbfs_a_star

September 27, 2023

0.1 CSE 4705: Assignment 02 - Arad to Bucharest - BFS, DFS, UCS, GBFS, A*

0.2 Problem 1

[100] Write a routine that solves the problem of finds a travel path of cities from from Arad to Bucharest in Romania, as discussed in class. Do this using each of the following approaches (points shown in brackets):

1. [15] Breadth First Search (BFS)
2. [10] Depth First Search (DFS)
3. [25] Uniform Cost Search (UCS)
4. [25] Greedy Best First Search (GBFS)
5. [25] A*

You will use the map from Lecture 03 - Informed Search which shows the major cities in Romania and the distances between them for those cities that are directly connected. Also, you will use the straight-line-distances shown in the adjacent table for your heuristic function, $h(n)$ for GBFS and A*. A screenshot of the relevant slide is given below. Data structures that store this information, romania_map and sld_to_bucharest, have been provided so you can access and apply this data in your algorithm implementations. Details of these data structures are given below.

0.2.1 Output for Each Routine

Each of your routines should return an output or set of outputs that clearly indicates the following:

1. The sequence of cities from Arad to Bucharest. (Make sure the cities, Arad and Bucharest are explicitly listed as the first and last cities in your output.) One suggestion is to return this output in the form of a list.
2. Cost to travel to each city from its predecessor.
3. Total cost for the path.

In the case of A* and Uniform Cost Search, your routines should return the *cheapest path*. However, that will not necessarily be the case for BFS, DFS, or GBFS. (Why not?)

0.2.2 Romania Graph

You will use the data structure stored in the romania_map, assigned below to implement the search across the various cities to find a path from Arad to Bucharest.

Some details about romania_map: - A dictionary of dictionaries - The outer dictionary is as follows: each key is a city and the value for that city is a nested dictionary of cities to which the said city is directly connected. - The nested dictionary contains the cities to which the parent key is directly connected (keys) and the corresponding distances from the parent city to those respective cities (values). - For example, for the city Oradea, we have a key in the outer dictionary (Oradea), and the associated value is a dictionary containing the Zerind and Sibiu as keys, where for each of these the values are the distances from Oradea to these respective cities.

[59]:
```
romania_map = {
    'Oradea':{'Zerind':71, 'Sibiu':151},
    'Zerind':{'Oradea':71, 'Arad':75},
    'Arad':{'Zerind':75, 'Sibiu':140, 'Timisoara':118},
    'Timisoara':{'Arad':118, 'Lugoj':111},
    'Lugoj':{'Timisoara':111, 'Mehadia':70},
    'Mehadia':{'Lugoj':70, 'Dobreta':75},
    'Dobreta':{'Mehadia':75, 'Craiova':120},
    'Sibiu':{'Oradea':151, 'Fagaras':99, 'Rimnicu Vilcea':80, 'Arad':140},
    'Rimnicu Vilcea':{'Sibiu':80, 'Pitesti':97, 'Craiova':146},
    'Craiova':{'Rimnicu Vilcea':146, 'Pitesti':138, 'Dobreta':120},
    'Fagaras':{'Sibiu':99, 'Bucharest':211},
    'Pitesti':{'Rimnicu Vilcea':97, 'Bucharest':101, 'Craiova':138},
    'Neamt':{'Iasi':87},
    'Giurgiu':{'Bucharest':90},
    'Bucharest':{'Pitesti':101, 'Fagaras':211, 'Urziceni':85, 'Giurgiu':90},
    'Iasi':{'Neamt':87, 'Vaslui':92},
    'Urziceni':{'Bucharest':85, 'Vaslui':142, 'Hirsova':98},
    'Vaslui':{'Iasi':92, 'Urziceni':142},
    'Hirsova':{'Urziceni':98, 'Eforie':86},
    'Eforie':{'Hirsova':86}
}
```

0.2.3 Heuristic Function Data - Straight-Line Distances to Bucharest

You will use the dictionary below as your resource for retrieving straight-line distance data for implementing the GBFS and A* algorithms.

[60]:
```
sld_to_Bucharest = {'Arad':366,
                    'Bucharest':0,
                    'Craiova':160,
                    'Dobreta':242,
                    'Eforie':161,
                    'Fagaras':176,
                    'Giurgiu':77,
                    'Hirsova':151,
                    'Iasi':226,
                    'Lugoj':244,
                    'Mehadia':241,
                    'Neamt':234,
```

```
                    'Oradea':380,
                    'Pitesti':100,
                    'Rimnicu Vilcea':193,
                    'Sibiu':253,
                    'Timisoara':329,
                    'Urziceni':80,
                    'Vaslui':199,
                    'Zerind':374
                   }
```

0.2.4 1. BFS Implementation

Provide your implementation of the BFS Search below.

[61]:
```
class Node:
    def __init__(self, value):
        self.value = value
        self.adj_nodes = {} # key = adjacent node object : value = distance
        self.prev = None

class Graph:
    def __init__(self):
        self.nodes = []
        self.cities = {} # key = city name : value = node object

    def add_node(self, value):
        if isinstance(value, Node):
            self.nodes.append(value)
            return value

        new_node = Node(value)
        self.nodes.append(new_node)
        return new_node

    def add_edge(self, node1, node2, distance=None):
        node1.adj_nodes[node2] = distance
        node2.adj_nodes[node1] = distance


    def add_cities(self, locations): #add locations from a dictionary to graph
        for city in locations:
            if city not in self.cities:
                Source_Node = self.add_node(city)
                self.cities[city] = Source_Node
            else:
                Source_Node = self.cities[city]
```

```
            for neighbor in locations[city]:
                if neighbor not in self.cities:
                    Destination_Node = self.add_node(neighbor)
                    self.cities[neighbor] = Destination_Node
                else:
                    Destination_Node = self.cities[neighbor]

                self.add_edge(Source_Node, Destination_Node,␣
↪locations[city][neighbor])



    #Reset Node pointers to run next algo
    def reset_prev_pointers(self):
        for node in self.nodes:
            node.prev = None


    # traverse previous nodes, format, then return path as string
    def sol_found(self,start_node, node):
        goal_node = node
        path = []
        distance = 0
        path_str = ""

        while (node.prev != None):
            cur_node = node
            path.append(cur_node)

            distance += cur_node.adj_nodes[node.prev]

            node = node.prev

        path.append(start_node)

        for node in path[::-1]:
            path_str += str(node.value)
            if node != goal_node:
                path_str += " -> "


        path_str += f"  total distance= {distance}"
        return  path_str
```

[62]:
```
city_graph = Graph()
city_graph.add_cities(romania_map)
```

[63]:
```
# ************************************
# BFS CODE
# ************************************
def bfs(self, start_node, end_node):
    if (not isinstance(start_node, Node)):
        start_node = self.cities[start_node]

    if (not isinstance(end_node, Node)):
        end_node = self.cities[end_node]
    visited_nodes = set()
    frontier = []
    visited_nodes.add(start_node)
    frontier.append(start_node)
    while (frontier): # while frontier is not empty
        current_node = frontier.pop(0) #current_node = frontier.get() get first␣
 ↪city in frontier

        for adj_node in current_node.adj_nodes:
            if adj_node not in visited_nodes:
                adj_node.prev = current_node
            if adj_node == end_node: #if neighbor_node is goal state
                return self.sol_found(start_node,adj_node)

            #visited_nodes.add(neighbor_node)
            visited_nodes.add(adj_node)
            #frontier.put(neighbor_node)
            frontier.append(adj_node)

    return "no path"
```

[64]:
```
print(f"BFS path: {bfs(city_graph,'Arad', 'Bucharest')}")
city_graph.reset_prev_pointers()
```

```
BFS path: Arad -> Sibiu -> Fagaras -> Bucharest  total distance= 450
```

[ ]:
```

```

0.2.5 2. DFS Implementation

Provide your implementation of the DFS Search below.

[65]:
```
# ************************************
# DFS CODE
# ************************************

#picks node and explores as deep as possible until goal node is found

def dfs(self, start_node, end_node):
```

```
    if (not isinstance(start_node, Node)):
        start_node = self.cities[start_node]

    if (not isinstance(end_node, Node)):
        end_node = self.cities[end_node]

    visited_nodes = set()
    frontier = []
    visited_nodes.add(start_node)
    frontier.append(start_node)
    while (frontier): # while frontier is not empty
        current_node = frontier.pop() #current_node = frontier.get() get last␣
 ↪city in frontier (only difference between bfs & dfs)
        for adj_node in current_node.adj_nodes:
            if adj_node not in visited_nodes:
                adj_node.prev = current_node
            if adj_node == end_node: #if neighbor_node is goal state
                return self.sol_found(start_node,adj_node)
            #visited_nodes.add(neighbor_node)
            visited_nodes.add(adj_node)
            #frontier.put(neighbor_node)
            frontier.append(adj_node)

    return "no path"
```

[66]:
```
print(f"DFS path: {dfs(city_graph,'Arad', 'Bucharest')}")
city_graph.reset_prev_pointers()
```

```
DFS path: Arad -> Timisoara -> Lugoj -> Mehadia -> Dobreta -> Craiova -> Pitesti
-> Bucharest  total distance= 733
```

[ ]:

[ ]:

[ ]:

0.2.6 3. UCS Implementation

Provide your implementation of the UCS Search below.

[67]:
```
#built in PQ doesn't support indexing
class PriorityQueue:
    def __init__(self):
        self.queue = [] # key = priority  , value = node object

    def sort_queue(self):
        self.queue = sorted(self.queue, key=lambda x: x[0])
```

```
    def put(self, node):
        self.queue.append(node)
        self.sort_queue()

    def get(self):
        return self.queue.pop(0)

    def empty(self):
        return len(self.queue) == 0

    def get_queue(self):
        return self.queue

    def count(self, node):
        return self.queue.count(node)

    def remove_occurrences_except_min(self, node_value):
        occurrences = [n for n in self.queue if n[1].value == node_value]
        if occurrences:
            min_occurrence = min(occurrences, key=lambda x: x[0])
            self.queue = [n for n in self.queue if n[1].value != node_value or␣
 ↪n == min_occurrence]
```

[68]:
```
# ************************************
# UCS CODE
# ************************************
def compute_cost(cur_node):
    cost = 0
    tmp = cur_node
    while tmp.prev is not None:
        cost += tmp.adj_nodes[tmp.prev]
        tmp = tmp.prev
    return cost

def ucs(self, start_node, end_node):
    if (not isinstance(start_node, Node)):
        start_node = self.cities[start_node]
    if (not isinstance(end_node, Node)):
        end_node = self.cities[end_node]
    visited_nodes = set()
    frontier = PriorityQueue()  # (priority, node)
    frontier.put((0, start_node))
    while (frontier):  # while frontier is not empty
        current_cost, current_node = frontier.get()
        visited_nodes.add(current_node)
```

```
        if current_node == end_node:
            return self.sol_found(start_node, current_node)
        for adj_node in current_node.adj_nodes:
            if adj_node not in visited_nodes:
                adj_node.prev = current_node
                # compute cost and put it in frontier (priority queue)
                current_cost = compute_cost(adj_node)
                # print(adj_node.value)
                #if these same node.value is in the frontier with a lower cost␣
 ↪then update the current cost to that cost
                frontier.put((current_cost, adj_node))
            if frontier.count(adj_node) > 1:
                print(frontier.count(adj_node))
                frontier.remove_occurrences_except_min(adj_node.value)
    return "no path"
```

[69]:
```
print(f"UCS path: {ucs(city_graph,'Arad', 'Bucharest')}")
city_graph.reset_prev_pointers()
```

```
UCS path: Arad -> Sibiu -> Rimnicu Vilcea -> Pitesti -> Bucharest  total
distance= 418
```

[ ]:

[ ]:

0.2.7 4. GBFS Implementation

Provide your implementation of the GBFS Search below.

[70]:
```
# ************************************
# gbfs CODE
# ************************************
def gbfs(self, start_node , end_node):
    if (not isinstance(start_node, Node)):
        start_node = self.cities[start_node]
    if (not isinstance(end_node, Node)):
        end_node = self.cities[end_node]

    visited_nodes = set()
    frontier = PriorityQueue()  # (priority, node)

    frontier.put((0, start_node))
    while (frontier):
        current_cost, current_node = frontier.get()
        visited_nodes.add(current_node)
        if current_node == end_node:
            return self.sol_found(start_node, current_node)
```

```
        for adj_node in current_node.adj_nodes:
            if adj_node not in visited_nodes:
                adj_node.prev = current_node
                # compute cost and put it in frontier (priority queue)
                current_cost = sld_to_Bucharest[adj_node.value] #shortest long␣
↪distance to Bucharest
                frontier.put((current_cost, adj_node))
            if frontier.count(adj_node) > 1:
                frontier.remove_occurrences_except_min(adj_node.value)
```

[71]:
```
print(f"GBFS path: {gbfs(city_graph,'Arad', 'Bucharest')}")
city_graph.reset_prev_pointers()
```

```
GBFS path: Arad -> Sibiu -> Fagaras -> Bucharest  total distance= 450
```

[ ]:

[ ]:

[ ]:

0.2.8 5. A* Implementation

Provide your implementation of the A* Algorithm below.

[72]:
```
def Astar(self, start_node , end_node):
    if (not isinstance(start_node, Node)):
        start_node = self.cities[start_node]
    if (not isinstance(end_node, Node)):
        end_node = self.cities[end_node]

    visited_nodes = set()
    frontier = PriorityQueue()  # (priority, node)

    frontier.put((0, start_node))

    while (frontier):
        current_cost, current_node = frontier.get()
        visited_nodes.add(current_node)
        if current_node == end_node:
            return self.sol_found(start_node, current_node)
        for adj_node in current_node.adj_nodes:
            if adj_node not in visited_nodes:
                adj_node.prev = current_node
                # compute cost and put it in frontier (priority queue)
                current_cost = sld_to_Bucharest[adj_node.value]
                current_cost += compute_cost(adj_node) #shortest long distance␣
↪to bucharest + cost to get to current node
```

```
                # print(adj_node.value)
                frontier.put((current_cost, adj_node))
            if frontier.count(adj_node) > 1:
                print(frontier.count(adj_node))
                frontier.remove_occurrences_except_min(adj_node.value)
```

[73]:
```
print(f"A* path: {Astar(city_graph,'Arad', 'Bucharest')}")
city_graph.reset_prev_pointers()
```

```
A* path: Arad -> Sibiu -> Rimnicu Vilcea -> Pitesti -> Bucharest  total
distance= 418
```

[ ]:

[ ]:

[ ]: