Adam_Syed_assignment_01_arad_to_bucharest_bfs_dfs_ucs

January 31, 2024

0.1 CSE 5713: Assignment 01 - Uninformed Search - Arad to Bucharest - BFS, DFS, and UCS

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. [35] Breadth First Search (BFS)
2. [35] Depth First Search (DFS)
3. [30] Uniform Cost Search (UCS)

You will use the map from Lecture 01 - Uninformed Search which shows the major cities in Romania and the distances between them for those cities that are directly connected. A screenshot of the relevant slide is given below. A data structure stores this information, romania_map. This has been provided so you can access and apply this data in your algorithm implementations. Details of this data structure 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 Uniform Cost Search, your routine should return the *cheapest path*. However, that will not necessarily be the case for BFS, or DFS. (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.

[11]:
```
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 1. BFS Implementation

Provide your implementation of the BFS Search below.

[18]:
```
from collections import deque
def bfs(graph, start, goal):
    visited = set()
    queue = deque([(start, [start])])
    while queue:
        current_node, path = queue.popleft()
        if current_node == goal:
            return path, sum(graph[path[i]][path[i+1]] for i in␣
 ↪range(len(path)-1))
        if current_node not in visited:
            visited.add(current_node)
            for neighbor in graph[current_node]:
                if neighbor not in visited:
                    queue.append((neighbor, path + [neighbor]))
    return None, None
```

[19]:
```
bfs_path, bfs_cost = bfs(romania_map, 'Arad', 'Bucharest')
bfs_path, bfs_cost
```

[19]:
```
(['Arad', 'Sibiu', 'Fagaras', 'Bucharest'], 450)
```

[20]:
```
def calculate_individual_costs(path, graph):
    return {path[i]: graph[path[i-1]][path[i]] if i > 0 else 0 for i in␣
 ↪range(len(path))}

bfs_costs = calculate_individual_costs(bfs_path, romania_map)
bfs_costs
```

[20]:
```
{'Arad': 0, 'Sibiu': 140, 'Fagaras': 99, 'Bucharest': 211}
```

0.2.4 2. DFS Implementation

Provide your implementation of the DFS Search below.

[28]:
```
def dfs(graph, start, goal):
    visited = set()
    stack = [(start, [start])]

    while stack:
        current_node, path = stack.pop()

        if current_node == goal:
            return path, sum(graph[path[i]][path[i+1]] for i in␣
 ↪range(len(path)-1))

        if current_node not in visited:
            visited.add(current_node)
            for neighbor in graph[current_node]:
                if neighbor not in visited:
                    stack.append((neighbor, path + [neighbor]))

    return None, None
```

[29]:
```
dfs_path, dfs_cost = dfs(romania_map, 'Arad', 'Bucharest')
dfs_path, dfs_cost
```

[29]:
```
(['Arad',
  'Timisoara',
  'Lugoj',
  'Mehadia',
  'Dobreta',
  'Craiova',
  'Pitesti',
  'Bucharest'],
```

733)

[30]:
```
dfs_costs = calculate_individual_costs(dfs_path, romania_map)
dfs_costs
```

[30]:
```
{'Arad': 0,
 'Timisoara': 118,
 'Lugoj': 111,
 'Mehadia': 70,
 'Dobreta': 75,
 'Craiova': 120,
 'Pitesti': 138,
 'Bucharest': 101}
```

0.2.5 3. UCS Implementation

Provide your implementation of the UCS Search below.

[39]:
```
import heapq
def ucs(graph, start, goal):
    frontier = [(0, start, [start])]
    explored = set()

    while frontier:
        path_cost, current_node, path = heapq.heappop(frontier)

        if current_node == goal:
            return path, path_cost

        if current_node not in explored:
            explored.add(current_node)
            for neighbor, cost in graph[current_node].items():
                if neighbor not in explored:
                    total_cost = path_cost + cost
                    heapq.heappush(frontier, (total_cost, neighbor, path +␣
 ↪[neighbor]))

    return None, None
```

[40]:
```
ucs_path, ucs_cost = ucs(romania_map, 'Arad', 'Bucharest')
ucs_path, ucs_cost
```

[40]:
```
(['Arad', 'Sibiu', 'Rimnicu Vilcea', 'Pitesti', 'Bucharest'], 418)
```

[41]:
```
ucs_costs = calculate_individual_costs(ucs_path, romania_map)
ucs_costs
```

[41]:
```
{'Arad': 0,
 'Sibiu': 140,
 'Rimnicu Vilcea': 80,
 'Pitesti': 97,
 'Bucharest': 101}
```

[ ]: