SP21-BCS-007

October 1, 2023

1 LAB 04: BFS and DFS for Graphs

1.1 Task 1: Implement Node Class

Write class Node that has the following attributes:

- state
- parent
- action
- path_cost

[ ]:
```
class Node:
    def __init__(self, state, parent, actions, totalcost):
        self.state = state
        self.parent = parent
        self.actions = actions
        self.totalcost = totalcost
```

1.2 Task 2: Implemnet Graphs using Node Class

Create different graphs using the Node class. Use dictionaries to store the graph nodes. The keys of the dictionaries are the names of the nodes and the values are the Node objects.

Graph 1:

[ ]:
```
graph1 = {
    "A": Node("A", None, ["B", "C", "C"], None),
    "B": Node("B", None, ["D", "A"], None),
    "C": Node("C", None, ["A", "F", "G"], None),
    "D": Node("D", None, ["B", "E"], None),
    "E": Node("E", None, ["D", "A"], None),
    "G": Node("G", None, ["C"], None),
    "F": Node("F", None, ["C"], None),
}
```

Graph 2:

[ ]:
```
graph2 = {
    "S" : Node("S", None, [], None),
    "p" : Node("p", None, [], None),
```

```
    "q" : Node("q", None, [], None),
    "r" : Node("r", None, [], None),
    "a" : Node("a", None, [], None),
    "b" : Node("b", None, [], None),
    "c" : Node("c", None, [], None),
    "d" : Node("d", None, [], None),
    "e" : Node("e", None, [], None),
    "f" : Node("f", None, [], None),
    "h" : Node("h", None, [], None),
    "G" : Node("G", None, [], None),
}
```

Graph 3:

```
[ ]: romania = {
    "Arad": Node("Arad", None, ["Zerind", "Sibiu", "Timisoara"], None),
    "Zerind": Node("Zerind", None, ["Arad", "Oradea"], None),
    "Oradea": Node("Oradea", None, ["Zerind", "Sibiu"], None),
    "Sibiu": Node("Sibiu", None, ["Arad", "Oradea", "Fagaras", "Rimnicu␣
 ↪Vilcea"], None),
    "Timisoara": Node("Timisoara", None, ["Arad", "Lugoj"], None),
    "Lugoj": Node("Lugoj", None, ["Timisoara", "Mehadia"], None),
    "Mehadia": Node("Mehadia", None, ["Lugoj", "Drobeta"], None),
    "Drobeta": Node("Drobeta", None, ["Mehadia", "Craiova"], None),
    "Craiova": Node("Craiova", None, ["Drobeta", "Rimnicu Vilcea", "Pitesti"],␣
 ↪None),
    "Rimnicu Vilcea": Node("Rimnicu Vilcea", None, ["Sibiu", "Craiova",␣
 ↪"Pitesti"], None),
    "Fagaras": Node("Fagaras", None, ["Sibiu", "Bucharest"], None),
    "Pitesti": Node("Pitesti", None, ["Rimnicu Vilcea", "Craiova",␣
 ↪"Bucharest"], None),
    "Bucharest": Node("Bucharest", None, ["Fagaras", "Pitesti", "Giurgiu",␣
 ↪"Urziceni"], None),
    "Giurgiu": Node("Giurgiu", None, ["Bucharest"], None),
    "Urziceni": Node("Urziceni", None, ["Bucharest", "Hirsova", "Vaslui"],␣
 ↪None),
    "Hirsova": Node("Hirsova", None, ["Urziceni", "Eforie"], None),
    "Eforie": Node("Eforie", None, ["Hirsova"], None),
    "Vaslui": Node("Vaslui", None, ["Urziceni", "Iasi"], None),
    "Iasi": Node("Iasi", None, ["Vaslui", "Neamt"], None),
    "Neamt": Node("Neamt", None, ["Iasi"], None),
}
```

1.3 Task 3: Implementing BFS

Implement BFS algorithm for the graphs created in Task 2. The algorithm should take the graph and the starting node as input and return the path from the starting node to the goal node. The path should be a list of node names.

[ ]:
```python
def BFS(graph, start, goal):
    queue = []
    visited = []
    queue.append(start)
    visited.append(start)
    path = []
    while queue:
        current = queue.pop(0)
        path.append(current)
        if current == goal:
            return path
        for neighbor in graph[current].actions:
            if neighbor not in visited:
                queue.append(neighbor)
                visited.append(neighbor)
    return path
```

Other Implementations of BFS:

[ ]:
```python
def BFS2(graph, start, goal):
    queue = []
    visited = []
    queue.append(start)
    visited.append(start)

    while queue:
        current = queue.pop(0)
        if current == goal:
            return actionSequence(graph, start, goal)  # Change to start here
        for neighbor in graph[current].actions:
            if neighbor not in visited:
                queue.append(neighbor)
                visited.append(neighbor)
                graph[neighbor].parent = current


def actionSequence(graph, start, goal):
    solution = [goal]
    current = goal
    while current != start:
        currentParent = graph[current].parent
        solution.append(currentParent)
        current = currentParent
    solution.reverse()
    return solution
```

1.4 Task 4: Testing BFS

Test BFS algorithm with the graphs created in Task 2. Print the path returned by the algorithm.

Graph 1:

```
[ ]: print(BFS(graph1, "D", "F"))
     print(BFS2(graph1, "D", "F"))
     print(actionSequence(graph1, "D", "F"))
```

```
['D', 'B', 'E', 'A', 'C', 'F']
['D', 'B', 'A', 'C', 'F']
['D', 'B', 'A', 'C', 'F']
```

Graph 2:

```
[ ]: print(BFS(graph2, "S", "G"))
     print(BFS2(graph2, "S", "G"))
     print(actionSequence(graph2, "S", "G"))
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
c:\Users\LENOVO\Repos\AI-Algorithms-Python\LAB_04\main.ipynb Cell 19 line 1
----> <a href='vscode-notebook-cell:/c%3A/Users/LENOVO/Repos/
 ↪AI-Algorithms-Python/LAB_04/main.ipynb#X31sZmlsZQ%3D%3D?line=0'>1</a>␣
 ↪print(BFS(graph2, "S", "G"))
      <a href='vscode-notebook-cell:/c%3A/Users/LENOVO/Repos/
 ↪AI-Algorithms-Python/LAB_04/main.ipynb#X31sZmlsZQ%3D%3D?line=1'>2</a>␣
 ↪print(BFS2(graph2, "S", "G"))
      <a href='vscode-notebook-cell:/c%3A/Users/LENOVO/Repos/
 ↪AI-Algorithms-Python/LAB_04/main.ipynb#X31sZmlsZQ%3D%3D?line=2'>3</a>␣
 ↪print(actionSequence(graph2, "S", "G"))

c:\Users\LENOVO\Repos\AI-Algorithms-Python\LAB_04\main.ipynb Cell 19 line 1

     <a href='vscode-notebook-cell:/c%3A/Users/LENOVO/Repos/AI-Algorithms-Pytho /
 ↪LAB_04/main.ipynb#X31sZmlsZQ%3D%3D?line=9'>10</a> if current == goal:
     <a href='vscode-notebook-cell:/c%3A/Users/LENOVO/Repos/AI-Algorithms-Pytho /
 ↪LAB_04/main.ipynb#X31sZmlsZQ%3D%3D?line=10'>11</a>     return path
---> <a href='vscode-notebook-cell:/c%3A/Users/LENOVO/Repos/AI-Algorithms-Pytho /
 ↪LAB_04/main.ipynb#X31sZmlsZQ%3D%3D?line=11'>12</a> for neighbor in␣
 ↪graph[current].actions:
     <a href='vscode-notebook-cell:/c%3A/Users/LENOVO/Repos/AI-Algorithms-Pytho /
 ↪LAB_04/main.ipynb#X31sZmlsZQ%3D%3D?line=12'>13</a>     if neighbor not in␣
 ↪visited:
     <a href='vscode-notebook-cell:/c%3A/Users/LENOVO/Repos/AI-Algorithms-Pytho /
 ↪LAB_04/main.ipynb#X31sZmlsZQ%3D%3D?line=13'>14</a>         queue.
 ↪append(neighbor)

KeyError: 'S'
```

Graph 3:

```
[ ]: print(BFS(romania, "Arad", "Bucharest"))
     print(BFS2(romania, "Arad", "Bucharest"))
     print(actionSequence(romania, "Arad", "Bucharest"))
```

```
['Arad', 'Zerind', 'Sibiu', 'Timisoara', 'Oradea', 'Fagaras', 'Rimnicu Vilcea',
'Lugoj', 'Bucharest']
['Arad', 'Sibiu', 'Fagaras', 'Bucharest']
['Arad', 'Sibiu', 'Fagaras', 'Bucharest']
```

1.5 Task 5: Implementing DFS

Implement DFS algorithm for the graphs created in Task 2. The algorithm should take the graph and the starting node as input and return the path from the starting node to the goal node. The path should be a list of node names.

```
[ ]: def DFS(graph, start, goal):
         stack = []
         visited = []
         stack.append(start)
         visited.append(start)
         path = []

         while stack:
             current = stack.pop()
             path.append(current)

             if current == goal:
                 return path

             for neighbor in graph[current].actions:
                 if neighbor not in visited:
                     stack.append(neighbor)
                     visited.append(neighbor)
                     graph[neighbor].parent = current

         return path
```

1.6 Task 6: Testing DFS

Test DFS algorithm with the graphs created in Task 2. Print the path returned by the algorithm.

```
[ ]: print(DFS(graph1, "D", "F"))
     # print(DFS(graph2, "S", "G"))
     print(DFS(romania, "Arad", "Bucharest"))
```

```
['D', 'E', 'A', 'C', 'G', 'F']
['Arad', 'Timisoara', 'Lugoj', 'Mehadia', 'Drobeta', 'Craiova', 'Pitesti',
```

```
'Bucharest']
```