Introduction to Artificial Intelligence

18 May, 2023

Proje 1 Report

The aim of our project is to write a program that chooses the shortest distance to be traveled and the way to go between the two cities entered using the information in the CSV file given to us. For this, I first imported my CSV file and defined it as path in the main method.

```
  path=r'C:\Users\BÜŞRA\Desktop\proje 1\data\cities.csv'
```

First of all, I throw an exception in the ***CityNotFoundError(Exception)*** class in case the city name is not found in the CSV file and the user does not enter it.

Then I wrote the "build_graph" method to read the data in a suitable data structure and initialized an empty graph as a dict in the method. This graph will then proceed according to the inputs entered by the user.

The following code reads the CSV file and creates a CSV reader. At first, I thought there was no need for the encoding part, and I got many errors, then I realized that the problem was caused by encoding, so I added "udf-8".

```
with open(path, 'r',  encoding="utf-8") as file:
        reader = csv.DictReader(file)
```

The following for loop iterates over the rows of the CSV file. It also adds the nodes and edge to the graph.

```
    for row in reader:
            source = row['city1']
            target = row['city2']
            distance = float(row['distance'])

            if source not in graph:
                graph[source] = {}
            if target not in graph:
                graph[target] = {}

            graph[source][target] = distance
            graph[target][source] = distance
```

The "**uniform_cost_search**" method, which includes the algorithm part of our program, has three parameters: graph, start and end.

graph is a dictionary for representing the weight graph structure. The dict contains the name of the neighboring node and the cost of the distance between them.

start is the city the user is currently in, the starting node.

end is the city the user wants to arrive at, the destination node.

This method will also give us the message CityNotFoundError if the start and end variables do not match one of the cities in our CSV file.

```
priority_queue = [(0, start)]
```

In this code, we create a priority queue to store the nodes to be visited.

```
cost_so_far = {start: 0}
```

Here we create a dictionary to track the total cost from the start node to each node.

```
previous_node = {start: None}
```

Finally, we need to create a dictionary to track the shortest path node.

Let's look at the following nested loop, which is the most important algorithm of this method.

The following steps are followed until the priority queue is empty:

1. The lowest cost node is opened from the priority queue.
2. If the current node is the end node, the shortest path is found and the loop ends.
3. Neighbors of the current node are discovered.
4. Calculate the total cost from the starting node to the neighboring node
5. If the neighboring node has not been visited before or the new cost is lower.
6. The cost of the neighboring node and the previous node are updated.
7. The neighboring node is prioritized with the new cost.
8. The shortest path is recreated. The **'previous_node**' dictionary is used here. Briefly, we start from the destination node and each node of the path is added to the **'path'** list by following the previous node.
9. Finally, the total cost to the target node is taken from the **'cost_so_far'** dict and returned with '**path**'.

```
 while priority_queue:

        current_cost, current_node = heapq.heappop(priority_queue)

        if current_node == end:
            break

        for neighbor, edge_cost in graph[current_node].items():

            total_cost = current_cost + edge_cost

            if neighbor not in cost_so_far or total_cost <
cost_so_far[neighbor]:
                cost_so_far[neighbor] = total_cost
                previous_node[neighbor] = current_node
                heapq.heappush(priority_queue, (total_cost, neighbor))

    path = []
    current = end
    while current:
        path.append(current)
        current = previous_node[current]
    path.reverse()
    distance = cost_so_far[end]
    return path, distance
```

Finally, in the main method, we want the user to enter the current city and the city they want to go to. In addition, we want to suppress the shortest distance between two cities and this road by calling the methods we wrote above here. In this method, we need try-catch blocks because when the program gives an error about file and city, these blocks catch the error and give an error message to the user.

```
try:
        graph = build_graph(path)
        result = uniform_cost_search(graph, start, end)
        if result:
            path, distance = result

            print("Shortest path: ", " -> ".join(path))
            print("distance: ", distance)
        else:
            print("No path found.")
    except FileNotFoundError:
        print("File not found.")
    except CityNotFoundError    :
        pass
```

I had a hard time finding the distance. The program was only pressing the route path correctly, but not subtracting the distance of the path. Then I realized that I made a mistake in the while loop in the "**uniform_cost_search**" method and the problem was solved.

```
PS C:\Users\BÜŞRA> & C:/Users/Public/anaconda3/python.exe "c:/Users/BÜŞRA/Desktop/proje 1/submission.py"
Enter the start (current city): Ankara
Enter the end (target city): Kayseri
Shortest path:  Ankara -> Konya -> Kayseri
distance:  280.0
PS C:\Users\BÜŞRA>
```

Büşra ZENBİLCİ

20170808054