Assignment 11: Graphs and Graph Search

CS 301

April 8, 2024

Last week, we looked at graph representations in code, as well as looked at algorithms to search within such graphs. For this assignment, lets have you implement those algorithms and explore them. For this assignment, implement your code in a `GraphSearch.py` Python file

1. Implement a `AdjacencyMatrix` and `AdjacencyList` class, representing the two common means of implementing graphs in code, specifically undirected graphs. Your classes (along with the constructor), should have the following methods:
    a. `readGraph(filepath)` reads the text file given at `filepath` and initializes a graph based on the information in the file and returns `True` if it is able to successfully read and create a graph, `False` otherwise (you may choose to print. The file is composed in the following format.:

```
Num_vertices      Num_edges
list_of_vertices
edge_1
edge_2
...
edge_num_edges
```

The first line in the file contains two numbers; the first number is the number of vertices in the graph, and the second number is the number of edges in the graph.
The second line contains a list of vertices separated by commas.
Subsequent lines contain an edge (each on a separate line) in the graph as two vertices separated by whitespace. An example graph file `(graph.txt)` is included for your testing.

b. `addVertex(vertex)` adds a vertex to the graph indicated by the `vertex` string. If the vertex already exists, do nothing. This method returns `True` on success, `False` otherwise.
c. `addEdge(edge)` adds an edge represented by the 2-tuple `edge` to the graph. If the edge already exists, do nothing. The method should ensure that the vertices in the edge already exist in the vertex and should return `False` if either is not present. On successful addition, the method should return `True`.
d. `deleteVertex(vertex)` deletes a vertex from the graph indicated by `vertex`. Note that deletion on a vertex should include deletion of all edges associated with the vertex. On successful deletion, the method returns `True`, `False` otherwise.
e. `deleteEdge(edge)` deletes the edge indicated by the 2-tuple `edge`. If the edge does not exist, return `False`. If the edge is successfully deleted, return `True`.
f. `getNeighbors(vertex)` returns the neighbors (vertices associated with the given `vertex` with an edge) as a list. If the vertex has no neighbors, return an empty list. If the vertex does not exist, return `False`.

2. Implement a function `BFS(graph, start_vertex, end_vertex)` that implements Breadth-First Search on the given `graph(AdjacencyMatrix or AdjacencyList)`, starting at `start_vertex` and ending at `end_vertex`. The function should return a list of edge tuples indicating the path from `start_vertex` to `end_vertex`, should one exist. If a path does not exist, return an empty list. If either vertex does not exist in the graph, return False. In comments, describe the average running time for BFS on a graph $G = \{V, E\}$ where V and E are sets of vertices and edges.
3. Implement a function `DFS(graph, start_vertex, end_vertex)` that implements Depth-First Search on the given graph, starting at `start_vertex` and ending at `end_vertex`. The function should return a list of edge tuples indicating the path from `start_vertex` to `end_vertex`, should one exist. If a path does not exist, return an empty list. If either vertex does not exist in the graph, return False. In comments, describe the average running time for DFS on a graph $G = \{V, E\}$ where V and E are sets of vertices and edges.

4. Using the sample `graph.txt` file, run the following searches using your DFS and BFS code. In comments in your code, provide a list of vertices that your code visits while conducting the search for each method. If the found path differ between DFS and BFS, discuss why that occurred.
    a. Start Vertex: a, End Vertex: p
    b. Start Vertex: n, End Vertex: b
    c. Start Vertex: g, End Vertex: q
    d. Start Vertex: c, End Vertex: o
    e. Start Vertex: o, End Vertex: p