PROJECT 2

Project 2: Graph Algorithms and Related Data Structures

Singles-source shortest path algorithm, Minimum Spanning Tree (MST), and Strongly Connected Components (SSCs)

Saumitra Apte (**801202857**)
Ajinkya Gadgil (**801200445**)

Project Overview

This project includes Implementation and analysis of below algorithms

1) Single source shortest path Algorithm (Dijkstra's)
2) Minimum Spanning Tree Algorithm (Prim's)
3) Strongly Connected Components.

Along with implementation and analysis, Pseudocode, Data structures used and examples have also been included.

Dijkstra's Single source shortest path algorithm

This algorithm is used to compute shortest distance of all the vertices from the given vertex. At every step, we add the vertex in the set/cloud with minimum distance from the given node.

Pseudo code:

Algorithm Dijkstra's:

G← graph
S← source

Let PQ be the Min-heap
Let dist be the dictionary for tracking costs
Let pred be the dictionary for tracking predecessor

PQ.push(0,S)

For all v in G:
  dist[v]= infinity
  pred[v]=null

dist[S]=0

While PQ is not empty:
    node= heappop(pq) //return vertex having minimum cost
    for each neighbour v for node:
        if dist[v]>dist[u]+weight[u,v]
            dist[v]=dist[u]+weight[u,v]
            pred[v]=u
            heappush(pq,dist[v],v)

Data Structures used:

- **Dictionary**:
  1) For storing graph in adjacency list format, dictionary is used where key is node /sources and it will contain pairs of destination and edge weights as values.
  For example, If there is an edge from source A to destination B and has weight 2 and edge to destination C with weight 3 then in the dictionary it would be stored as {'A': ['B',2], ['C',3]}.

2) For storing costs/distance for each node
3) For storing predecessor for each node

- **Min-Heap**
  1) Min heap is used to store node and its cost. This is used to get the node having minimum distance from the source.

Detailed analysis of runtime:

1) For initialization of the graph the required complexity is O(V) where V is the number of vertices. Here we set the distance of each vertex to infinity and predecessor of each vertex to Null.
2) We have used min heap based priority queue and time complexity to build(add element to heap and perform heapify operation)the min heap is O(log V).
3) The heappop operation removes the minimum element from the heap. Each extraction takes log V and this would be performed until all the vertices are removed from the heap and hence its complexity would be O(V log V).
4) The relax operation (distance operation) is performed for all the adjacent nodes i.e. E times (number of edges). Each operation takes O(log V) and we do it for each adjacent edge hence its complexity is O(E log V)
5) Therefore, total time complexity is O (V log V) + O (E log V). Hence total time complexity is **O((V+E) log V)** where V is number of vertices and E is number of edges. **If all the vertices are reachable from source the time complexity would be O(E log V)**

Minimum Spanning Tree algorithm using Prim's

We have to find a spanning tree with minimized total weight from given weighted undirected graph.

Pseudo code:

Algorithm Prims:

```
G← graph
S← source

Let PQ be the Min-heap
Let keys be the dictionary for tracking weight
Let pred be the dictionary for tracking predecessor
Let visited be the dictionary for tracking if node is visited.

PQ.push(0,S)

For all v in G:
  keys[v]= infinity
  pred[v]=null
  visited[v]=false

keys[S]=0

While PQ is not empty:
        node= heappop(pq)              //return vertex having minimum cost
        visited[node]=true
        for each neighbour v for node:
                if visited[v]=false AND keys[v]>weight(u,v):
                        keys[v]=weight(u,v)
                        heappush(pq,(weight(u,v),v))
                        pred[v]=u
```

Data Structures used:

- **Dictionary**:
  1) For storing graph in adjacency list format, dictionary is used where key is node /sources and it will contain pairs of destination and edge weights as values. For example, If there is an edge from source A to destination B and has weight 2 and edge to destination C with weight 3 then in the dictionary it would be stored as {'A': ['B',2], ['C',3]}
  2) For storing weights for each node
  3) For storing predecessor for each node
  4) To check if node is visited or not
- **Min-Heap**
  1) Min heap is used to store node and its cost. This is used to get the node having minimum distance from the source.

Run time analysis:

1) For initialization of the graph the required complexity is O(V) where V is the number of vertices. Here we set the distance of each vertex to infinity and predecessor of each vertex to Null.
2) We have used min heap based priority queue and time complexity to build (add element to heap and perform heapify operation the min heap is O(log V).
3) The heappop operation removes the minimum element from the heap. Each extraction takes log V and this would be performed until all the vertices are removed from the heap and hence its complexity would be O(V log V).
4) The key/weight update operation is performed for all the adjacent nodes i.e. E times (number of edges). Each operation takes O(log V) and we do it for each adjacent edge hence its complexity is O(E log V)
5) Therefore, total time complexity is O (V log V) + O (E log V). Hence total time complexity is **O((V+E) log V)** where V is number of vertices and E is number of edges. Overall complexity will be **O (E log v)**

Strongly Connected Components

In Strongly connected components, we find maximal subgraphs such a way that each vertex reaches all other vertices. We use Depth First search to get the result. Then Transpose on the graph is taken and DFS is applied on Transposed graph in the order of finished times found in first DFS.

Pseudo Code

Algorithm Strongly_connected_components:

Let G← graph

Let visited be the dictionary to check if node is visited
Let S be the stack to arrange nodes according to finish time

Dfs_main(graph) //function call for DFS on G

Let G' be the graph with reversed edges
Let res be the nodes connected to given node
Let final be the list to store strongly connected components
Let reverse_visited be the dictionary to check if node in G' is visited

```
While S not empty:
        vertex=S.pop()
        If reverse_visited[vertex ]!=1:
                scc_dfs(vertex)          //function call for DFS on G'
                final.append(res)
                res=[]
   return final

Function scc_dfs(Vertex v):
reverse_visited[v]=1
res.append(v)
for all neighbours node in G'[v]:
        if(reverse_visited[node]!=1)
                scc_dfs(node)

Function DFS_main(Graph graph):
        For v in graph:
                If visited[v]!=1:
                        Dfs_visit(v)

Function DFS_visit(Vetex v):
        Visited[v]=1
        For each neighbour node in v:
                If(visited[node] ! =1):
                        DFS_visit(node)
```

S.push(node)

Data Structures used:

- Dictionary:
  a) For storing graph in adjacency list format, dictionary is used where key is node /sources and it will contain pairs of destination and edge weights as values.
  For example, If there is an edge from source A to destination B and has weight 2 and edge to destination C with weight 3 then in the dictionary it would be stored as {'A': ['B',2], ['C',3]}
  b) To check if nodes are visited
- Stack: Stack is used to store the nodes according to their finished times.
- List: List is used to store the strongly connected nodes.

Run-Time Analysis:

- Depth First search's time complexity is O(E+V) where V is number of vertices and E is number of Edges. Transpose of the graph will also take O(E+V). Stack's push and pop operation takes O (1). As Strongly connected Component involves DFS and transpose and again DFS, ultimately time complexity will be O(E+V) where E are the edges and V is vertices.

Steps to Execute the code:

1) Go to the folder where the files are located which have the code and graph inputs text file.
2) Run files (dijkstra.py, prims.py and strongly_connected_components.py) individually using python3 filename.py.
3) After running each of the file, it will display the menu to select which graph to run the algorithm on. The graphs are pre-defined in the files.
4) Files undirectedGraph1.txt, undirectedGraph2.txt, undirectedGraph3.txt, undirectedGraph4 has undirected graph inputs. Files directedGraph1, directedGraph2 has directed graph inputs. scc_1.txt, scc_2.txt, scc_3.txt and scc_2 has files for strongly connected components.
5) In graph files first line has the graph information where first number is number of vertices, second number is number of edges and third is ‘u‘or ‘d’ which specifies if the graph is directed or undirected. The last line has the source node. The other lines specify source, destination and weight between the nodes.
6) Menu’s have been given as per the requirements for example strongly connected components only directed graphs is used.

Sample Input and Outputs

1) Dijkstra's Single Source shortest path algorithm

Input:

```
10 20 d
1 2 5
2 5 6
5 8 2
8 10 4
10 9 7
9 6 8
6 3 1
3 1 4
1 4 3
2 3 15
4 6 6
4 5 7
4 7 1
7 6 8
7 9 7
7 10 9
7 8 12
7 5 11
2 8 9
6 10 5
1
```

Output

```
Path (1 —> 2): Minimum cost = 5, Route = ['1', '2']
Path (1 —> 5): Minimum cost = 10, Route = ['1', '4', '5']
Path (1 —> 8): Minimum cost = 12, Route = ['1', '4', '5', '8']
Path (1 —> 10): Minimum cost = 13, Route = ['1', '4', '7', '10']
Path (1 —> 9): Minimum cost = 11, Route = ['1', '4', '7', '9']
Path (1 —> 6): Minimum cost = 9, Route = ['1', '4', '6']
Path (1 —> 3): Minimum cost = 10, Route = ['1', '4', '6', '3']
Path (1 —> 4): Minimum cost = 3, Route = ['1', '4']
Path (1 —> 7): Minimum cost = 4, Route = ['1', '4', '7']
```

2) Minimum Spanning Tree Algorithm (Prim's)

Input

```
10 14 U
0 5 343
1 6 954
1 5 879
2 5 1054
2 4 1364
3 6 433
4 5 1106
8 0 464
0 7 1435
1 7 811
6 7 837
4 9 766
3 9 1053
1 9 524
1
```

Output

```
Edge : ( 5 , 0 ) Weight: 343
Edge : ( 1 , 5 ) Weight: 879
Edge : ( 7 , 6 ) Weight: 837
Edge : ( 5 , 2 ) Weight: 1054
Edge : ( 9 , 4 ) Weight: 766
Edge : ( 6 , 3 ) Weight: 433
Edge : ( 0 , 8 ) Weight: 464
Edge : ( 1 , 7 ) Weight: 811
Edge : ( 1 , 9 ) Weight: 524
Total Cost of minimum spanning tree is: 6111
```

3) **Strongly Connected Components**

Input:

12 17 D
A B 1
B C 2
B E 4
B D 3
C F 2
E F 6
E B 8
E G 3
F C 1
F H 7
G H 3
G J 2
H K 4
I G 3
J I 1
K L 3
L J 2
L

Output:
The strong connected components are: [['A'], ['B', 'E'], ['D'], ['C', 'F'], ['H', 'G', 'I', 'J', 'L', 'K']]

Output Screens

1) Sample output screens are available in output screens folder.