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

Ajinkya Gadgil

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]= inifinity

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]}

1. For storing costs/distance for each node
2. For storing predecessor for each node

* Min-Heap

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. Prim’s algorithm uses same approach as Dijkstra’s.

Pseudo code:

Algorithm Prim’s:

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]= inifinity

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]}

1. For storing weights for each node
2. For storing predecessor for each node
3. To check if node is visited or not

* Min-Heap

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. **z**

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.

**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:

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]}

1. 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)