GRAPH ALGORITHMS AND RELATED DATA STRUCTURES

By Suhail Lala

Description:

- ## Single-source Shortest Path:

  Dijkstra's Algorithm is used to find the single source shortest path for each vertex starting from the source vertex. It is a greedy algorithm which grows a cloud of vertices, beginning with the start vertex and eventually covering all vertices. For each vertex, we measure the distance to it from the start vertex in the subgraph consisting of the cloud and its adjacent vertices. At each step we add to the cloud a vertex outside the cloud with the smallest distance from the source vertex and update the distance values of any adjacent vertices.

- ## Minimum Spanning Tree:

  Prim's Algorithm is used to find the minimum spanning tree. It is a greedy algorithm and operates like Dijkstra's algorithm. The tree starts with an arbitrary root vertex r and grows until it spans all vertices. Each vertex stores the smallest weight of an edge connecting it to a vertex in the cloud. At each step we add to the cloud a vertex outside the cloud with the smallest weight and update the weights of adjacent vertices.

Data Structures used:

- ### Single-source Shortest Path:

  Single source Shortest Path makes use of dictionaries for storing information about the graph, distance from the start vertex and lowest cost path for each vertex. It also uses lists for keeping track of the visited vertices. Lastly, it uses a min-heap to choose which vertex to explore.

- ### Minimum Spanning Tree:

  Minimum Spanning Tree uses the same data structures that are used for single-source shortest path – dictionaries, lists and a min-heap. The only difference being a list is used to keep track of the tree edges compared to a dictionary which was used to keep track of the paths for Dijkstra’s algorithm.

Complexity Analysis:

- ### Single-source Shortest Path:

```
Relax(u, v, w):
    If v.d > u.d + w(u, v) do
        v.d = u.d + w(u, v)
        v.path = u + u.path
        heap.push(v.d)

Dijkstra(G, s):
    For each vertex v in G.V:
        v.d = inf
        v.path = NIL
    s.d = 0
    heap.push(s.d)
    S = empty list
    While heap > 0 do
        u = Extract-min(heap)
        if u not in S do:
            S.append(u)
            For each vertex v in G.Adj(u) do
                Relax(u, v, v.cost)
```

Time to build the min-heap	= O(n log n)
Each extract-min operation	= O(log n)
Each relax operation	= O(log n)
Maximum relax operations	= O(m)

Hence, total running time	= O(n log n + m log n)

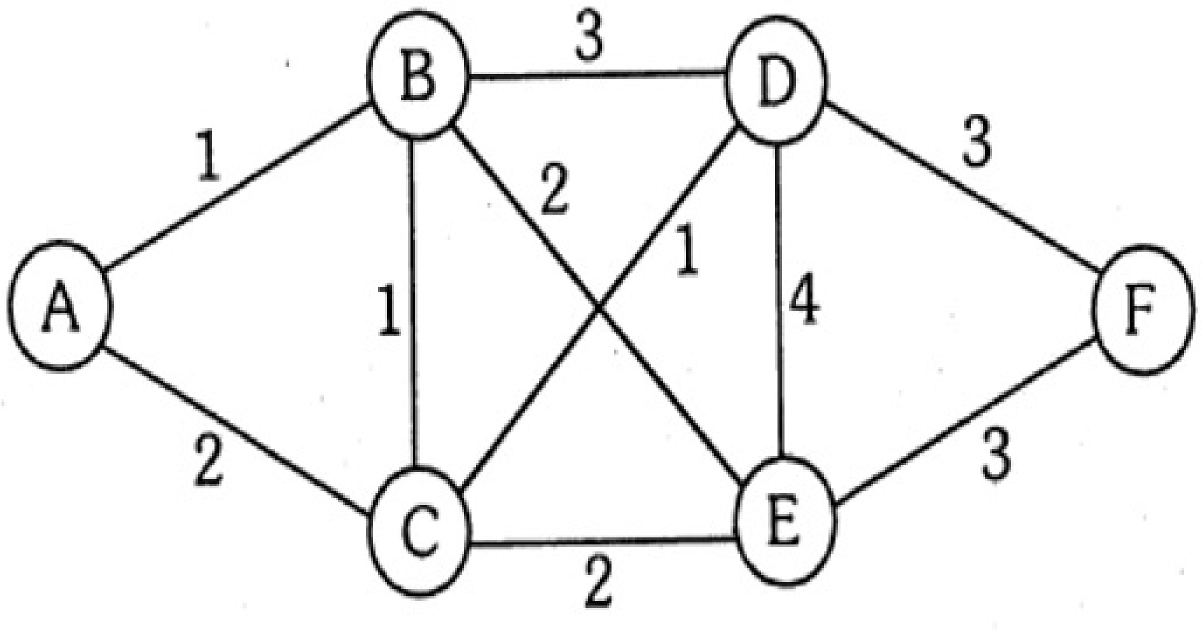
- ## Minimum Spanning Tree:

```
MST_Prim(G, s):
    For each vertex v in G.V:
        v.key = inf
        v.parent = NIL
    s.key = 0
    heap.push(s.key)
    S = empty list
    While heap > 0 do
        u = Extract-min(heap)
        if u not in S do:
            S.append(u)
            For each vertex v in G.Adj(u) do
                If w(u, v) < v.key and v not in S do:
                    v.key = w(u, v)
                    v.parent = u
                    heap.push(v.key)
```

Time to build the min-heap	= O(n log n)
Each extract-min operation	= O(log n)
Each inner loop operation	= O(log n)
Maximum inner loop operations	= O(m)
Hence, total running time	= O(n log n + m log n)

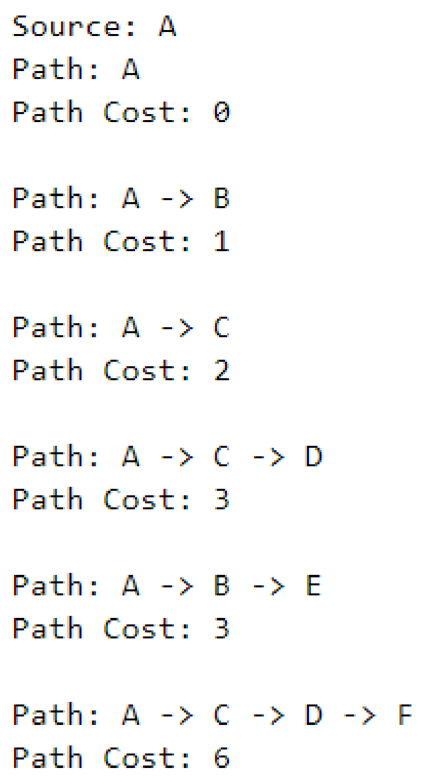
Sample input/output:

- Input:



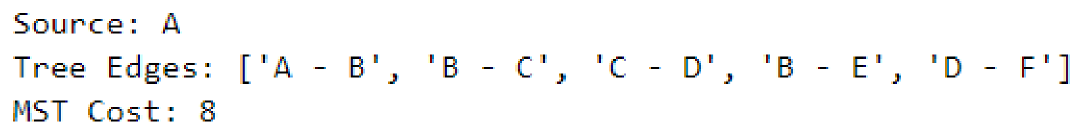
```
6 10 U
A B 1
A C 2
B C 1
B D 3
B E 2
C D 1
C E 2
D E 4
D F 3
E F 3
A
```

- Output:
    - Single-source Shortest Path:



```
Source: A
Path: A
Path Cost: 0

Path: A -> B
Path Cost: 1

Path: A -> C
Path Cost: 2

Path: A -> C -> D
Path Cost: 3

Path: A -> B -> E
Path Cost: 3

Path: A -> C -> D -> F
Path Cost: 6
```

    - Minimum Spanning Tree:



```
Source: A
Tree Edges: ['A - B', 'B - C', 'C - D', 'B - E', 'D - F']
MST Cost: 8
```

Instructions to run program:

The source code is a Jupyter notebook. So, any IDE compatible with Jupyter Notebook can be used to run it.