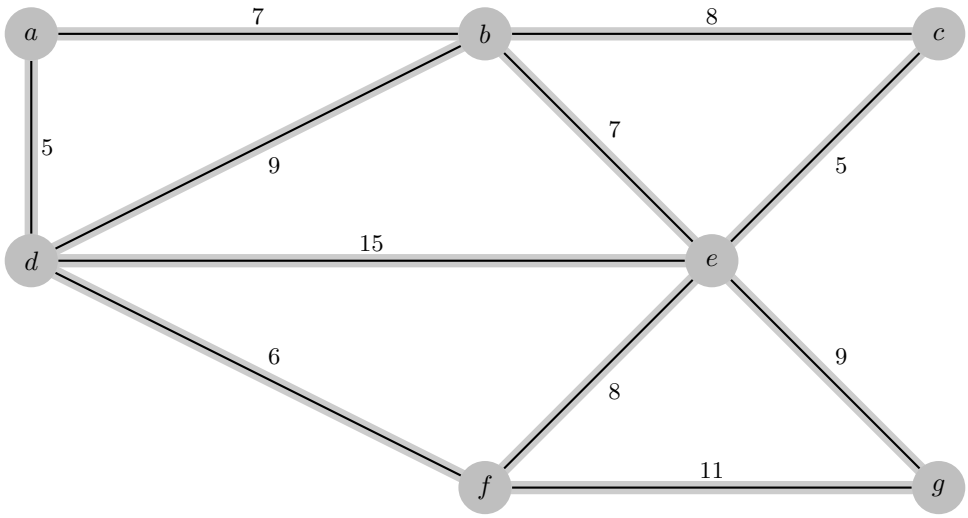
University of Illinois @ Urbana-Champaign

CI 487: Data Structures for CS Teachers

Implementation #7: Dijkstra's Shortest Path Algorithm and Prim's Minimum Spanning Tree



1 Step 0: The Code You are Given

To begin, you are given an adjacency list implementation of an undirected, weighted graph which had the following methods:

1. `addVertex(E vertex){ ... }`
2. `addEdgeWeighted(E source, E dest, int weight){ ... }`

You can use these methods to create graphs to test your implementations of Prim's MST and Dijkstra's SP. Much like the BST and DFS implementations we will be using extra data structures to keep track of attributes related to the state of a given vertex. For both Dijkstra's and Prim's you are given the same starter code since both algorithms are tracking the same attributes (i.e., visited, parent, distance).

```
Map<T, T> parents = new HashMap<T, T>();
Map<T, Integer> dists = new HashMap<>();
Queue<T> pq = new PriorityQueue<T>(Comparator.comparing(dists::get));
```

These have the following purposes:

- `parents`: This keeps track of our vertex's parent by having vertices of type `E` as both keys and values.
- `dists`: This keeps track of the distance attribute related to a given node by having vertices of type `E` as keys and `Integers` as values.
- `pq`: This is our min priority queue (lowest value in front) and how we keep track of and access vertices that we still have left to visit. Additionally, we are overriding it's comparison behavior by saying it should get each vertex's value from `dists` in order to determine it's position within the priority queue.

Additionally, you are given an implementation of `reprioritize` which removes and readds and entry to a priority queue in order to force said vertex to be reordered within a priority queue. This method will appear in the psudeocode for each of the algorithms and should be called after updating the distance attribute associated with a given vertex to ensure it is repositioned properly.

1.1 Prim's Minimum-Spanning Tree

Algorithm 1: Prim's Minimum Spanning Tree

```
Function Prim(G, Source)
    for u ∈ G.Vertecies do
        u.dist ← ∞
        u.parent ← null
    end
    Source.dist ← 0
    PQ ← ∅
    PQ.Enqueue(all vertecies)
    while PQ ≠ ∅ do
        u ← PQ.RemoveMin()
        for edge ∈ G.Adj[u] do
            v ← eedge.dest
            if v ∈ PQ AND edge.weight < v.dist then
                v.parent ← u
                v.dist ← e.weight
                PQ.Reprioritize(v)
            else
        end
    end
return
```

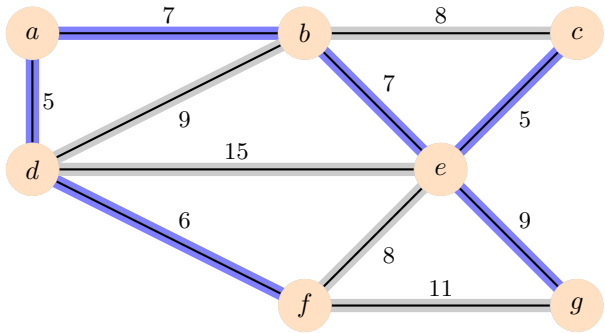


Figure 1: Prim's Minimum Spannning Tree (Source = D)

For the method `primsMST` you should implement the algorithm described by Algorithm 1. Additionally, after running the algorithm your method should print all of the edges that make up the minimum spanning tree. Additionally, the following methods may be useful when implementing this algorithm:

- `contains(E e)`: The `Queue<E>` interface contains a queue method that can be used to determine if the queue contains an element.
- `offer(E e)`: The `Queue<E>` interface uses the `offer` method to allow for the enqueueing of elements.
- `poll(E e)`: The `Queue<E>` interface uses the `offer` method to allow for the dequeueing of elements.
- `keySet()`: The `Map<E>` interface has this method which will let you get all the keys from a map. For our adjacency graph, the keys are our set of vertecies. This will be useful for when you initially add all vertecies to the queue.

Hint: After running this algorithm, the `parents` HashMap will contain these edges.

Example Output

```
Prims Edges (Source=D):
A, D
B, A
C, E
D, null
E, B
F, D
G, E
```

1.2 Dijkstras Shortest Path Algorithm

Algorithm 2: Dijkstra's Shortest Path

```
Function Dijkstra(G, Source, Dest)
    for u ∈ G.Vertex do
        u.distance ← ∞
        u.parent ← null
    end
    Source.distance ← 0
    PQ ← G.Vertex
    PQ.Enqueue(all vertecies)
    while PQ ≠ ∅ do
        u ← PQ.getMin()
        for edge ∈ G.Adj[u] do
            v ← edge.dest
            PathWeight ← u.distance + edge.weight)
            if v ∈ PQ AND PathWeight < v.distance
             then
                v.parent ← u
                v.distance ← PathWeight
                PQ.reprioitize(v)
            end
        end
    end
return
```

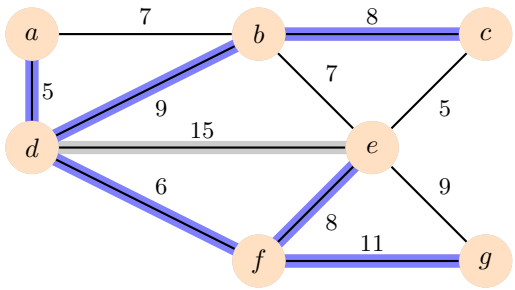


Figure 2: Dijkstra's Shortest Path (Source = D)

Though there are several variations to Dijkstra's shortest path algorithm the one we will be focusing on will be for weighted, undirected graphs. This is in preparation for the final project which deals with the creation and traversal of a graph of this type. Additionally, we will be focusing on the implementation on finding the shortest path from a given source node to a given destination. The first step in doing this will be to implement Algorithm 2 wich will find the shortest path from a given source node to *every* vertex in the graph.

Algorithm 3: Backtrack Traversal from Dijkstra's

```
Function PrintDijkstraSP(G, Source, Dest)
    Q ← ∅
    curr ← Dest
    while Source ∉ Q do
        Q.add(curr)
        curr = curr.parent
    end
return
```

Upon the completion of running the algorithm you should have two data structures, one storing all of the nodes and their parents and another storing the total distance from the source node to each vertex in the graph. We can use another algorithm to begin at the destination and backtrack to the source. Along the way we can collect nodes and recreast the shortest path traversal. The algorithm for doing so is given by Algorithm 3. Here, you will begin at the destination and work your way backwards through your traversal (i.e., the parents of each vertex) until you find the source. Once this is done, output that traversal in addition to the total distance. The methdod you will implement for doing this is called `printSP` and takes: 1) the source vertex, 2) the destination vertex, 3) the map of parent-child pairs, and 3) the map of distance-value pairs.

Example Output

```
Dijkstras Edges (Source=D, Dest=G):
[D, F, G]: 17
```