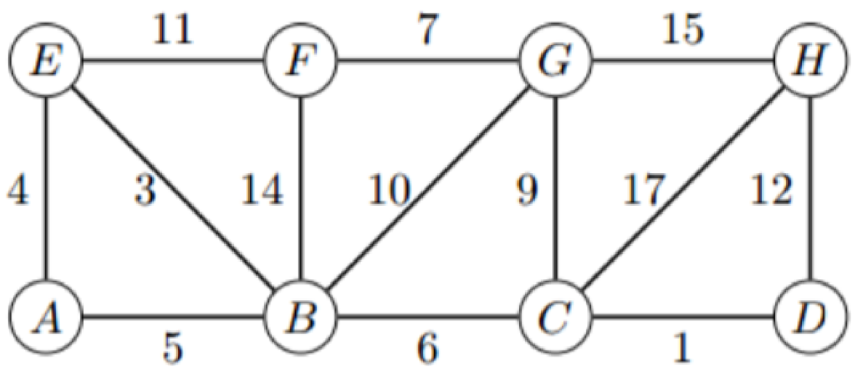
CS 325 Spring 2019, Homework 5

May 13, 2019

1. Consider the weighted graph below:



   1. Demonstrate Prim's algorithm starting from vertex A. Write the edges in the order they were added to the minimum spanning tree.

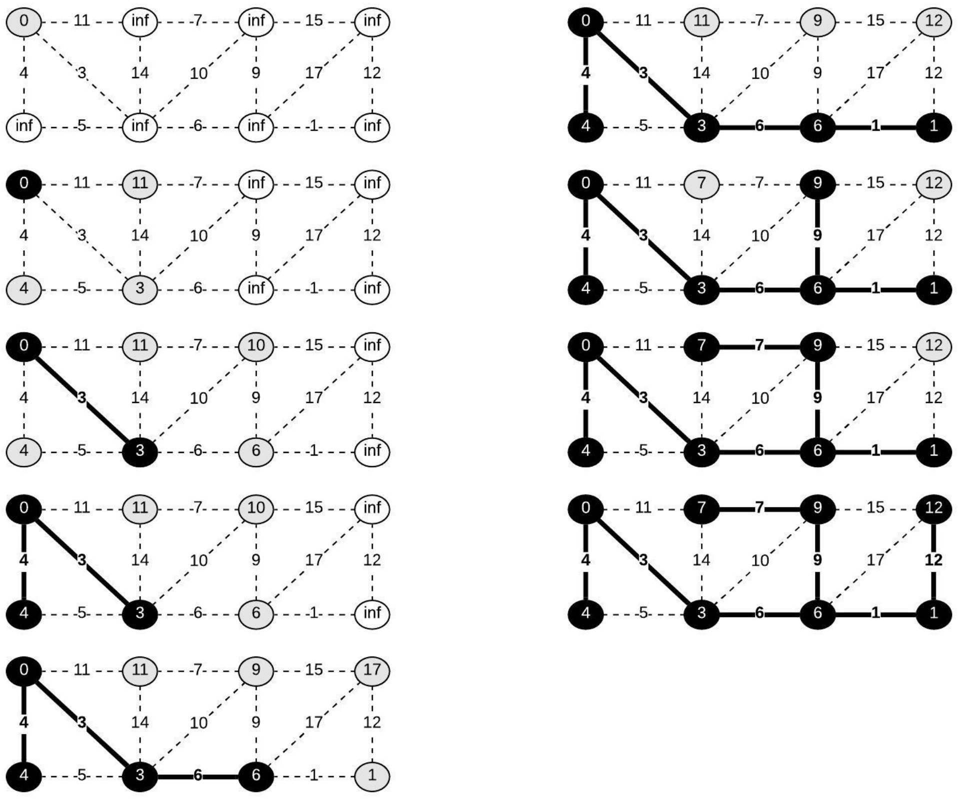
      The idea behind Prim's algorithm is simple, a spanning tree means all vertices must be connected. So two disjoint subsets of vertices must be connected to make a **Spanning** Tree. And they must be connected with the minimum weight edge to make it a **Minimum** Spanning Tree.

      Algorithm:

      1. Create a set *mstSet* that keeps track of vertices already included in MST.
      2. Assign a key value to all vertices in the input graph. Initialize all key values as INFINITE. Assign key value as 0 for the first vertex so that it is picked first.
      3. While *mstSet* doesn't include all vertices,
         1. Pick a vertex u which is not in *mstSet* and has minimum key value.
         2. Include u to *mstSet*.
         3. Update key value of all adjacent vertices of u. To update the key values, iterate through all adjacent vertices. For every adjacent vertex v, if weight of edge $u - v$ is less than the previous key value of v, update the key value as weight of $u - v$.

   2. If each edge weight is increased by 1 will this change the minimum spanning tree? Explain.

      Increasing each edge weight by 1 will not change the minimum spanning. Although the edge weights changes, for each edge in the graph, its relative weight compared to all the other edge weights will still be the same. Thus, the MST does not change.



Prim's MST.

```
In [1]: def PrimMST(G, V):
            key    = {}       # To store the distances of vertices to their nearest neighbors
            parent = {}       # To store the parents of vertices
            heap   = Heap()   # Heap to store unselected vertices

            # Initialize min heap with all vertices
            # key as all infinity
            # parent as all -1 (no parent)
            i = 0
            for v in G.keys():
                parent[v] = -1
                key[v] = float('inf')
                heap.array.append(heap.NewMinHeapNode(v, key[v]))
                heap.pos[v] = i
                i += 1

            # Set the key value of src node to 0
            key[heap.array[0][0]] = 0
            heap.DecreaseKey(heap.array[0][0], 0)

            # Initialize size of min heap to V, the number
            # of vertices in the graph
            heap.size = V

            # The loop continues until all the vertices
            # have been finalized
            while not heap.IsEmpty():
                # Extract the vertex with minimum key value
                u = heap.ExtractMin()[0]

                # Traverse through all adjacent vertices of
                # u (the extracted vertex) and update their
                # key values
                for neighbor in G[u]:
                    v = neighbor[0]

                    # If distance to v is not finalized yet,
                    # and distance to v is less than its
                    # previously calculated distance
                    if heap.IsInMinHeap(v) and neighbor[1] < key[v]:
                        key[v] = neighbor[1]
                        parent[v] = u
                        heap.DecreaseKey(v, key[v])

            return parent
```

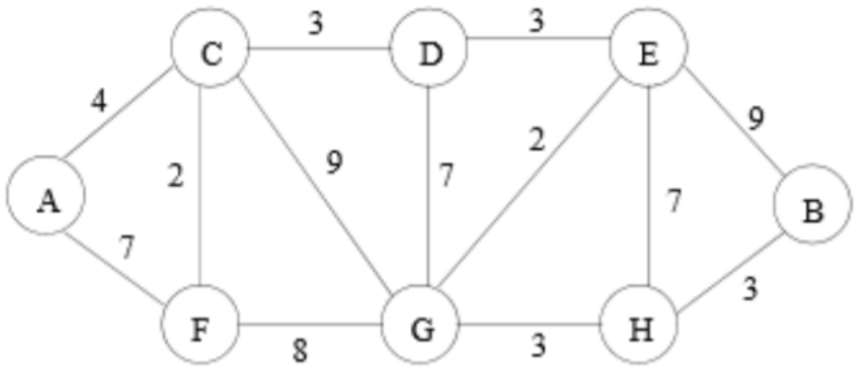
```
In [2]: import graph as g

        def PrintMST(parent):
            print('Prim\'s MST')
            print('Vertex\tParent')
            for v, p in parent.items():
                print('{0}\t{1}'.format(v, p))
            print()

        if __name__ == '__main__':
            graph = g.Graph(8)
            graph.AddEdge('A', 'E', 4)
            graph.AddEdge('A', 'B', 5)
            graph.AddEdge('E', 'B', 3)
            graph.AddEdge('E', 'F', 11)
            graph.AddEdge('B', 'F', 14)
            graph.AddEdge('B', 'G', 10)
            graph.AddEdge('F', 'G', 7)
            graph.AddEdge('B', 'C', 6)
            graph.AddEdge('C', 'G', 9)
            graph.AddEdge('C', 'H', 17)
            graph.AddEdge('G', 'H', 15)
            graph.AddEdge('C', 'D', 1)
            graph.AddEdge('D', 'H', 12)
            parent = graph.PrimMST()
            PrintMST(parent)
```

```
Prim's MST
Vertex        Parent
A        -1
E        A
B        E
F        G
G        C
C        B
H        D
D        C
```

2. A region contains a number of towns connected by roads. Each road is labeled by the average number of minutes required for a fire engine to travel to it. Each intersection is labeled with a circle. Suppose that you work for a city that has decided to place a fire station at location G. (While this problem is small, you want to devise a method to solve much larger problems).



   1. What algorithm would you recommend be used to find the fastest route from the fire station to each of the intersections? Demonstrate how it would work on the example above if the fire station is placed at G. Show the resulting routes and times.

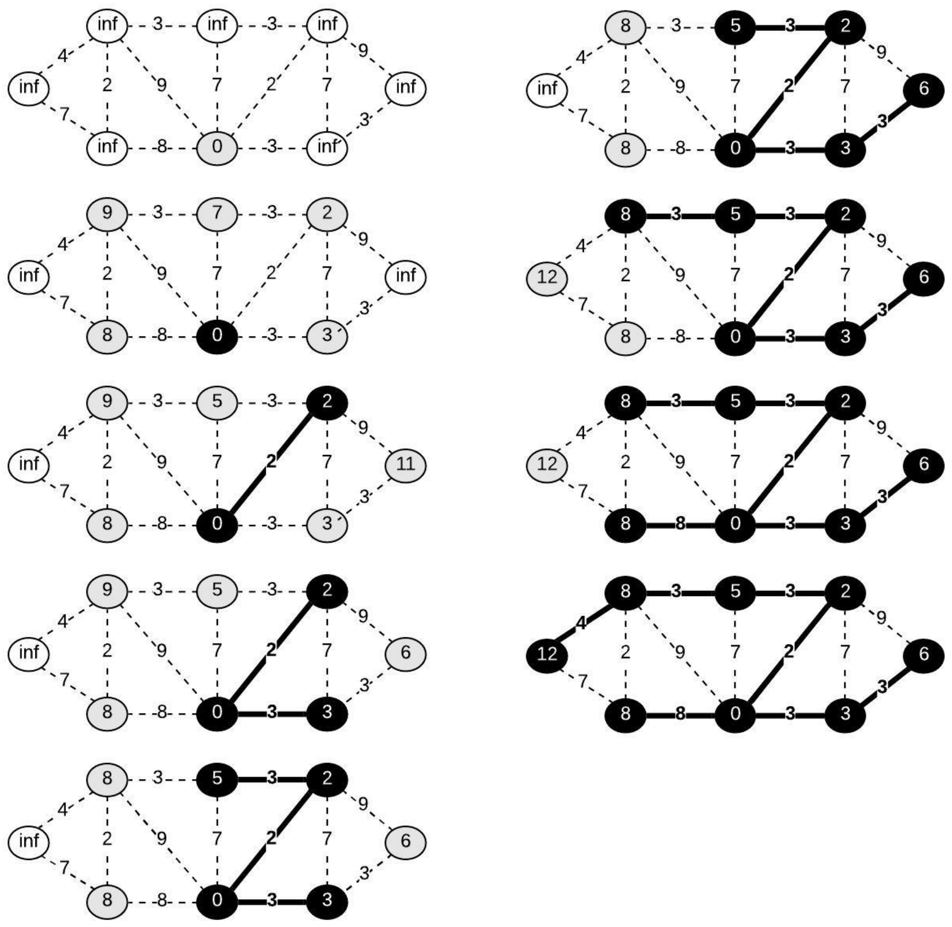
      To find the fastest route from the fire station to each of the intersection, we should use Dijkstra's algorithm. Dijkstra's algorithm is very similar to Prim's algorithm for minimum spanning tree. Like Prim's MST, we generate a SPT (shortest path tree) with given source as root. We maintain two sets, one set contains vertices included in shortest path tree, other set includes vertices not yet included in shortest path tree. At every step of the algorithm, we find a vertex which is in the other set (set of not yet included) and has a minimum distance from the source.

      Algorithm:

      1. Create a Min Heap of size V where V is the number of vertices in the given graph. Every node of min heap contains vertex number and distance value of the vertex.
      2. Initialize Min Heap with source vertex as root (the distance value assigned to source vertex is 0). The distance value assigned to all other vertices is INF (infinite).
      3. While Min Heap is not empty, do the following,
         1. Extract the vertex with minimum distance value node from Min Heap. Let the extracted vertex be u.
         2. For every adjacent vertex v of u, check if v is in Min Heap. If v is in Min Heap and distance value is more than weight of $u - v$ plus distance value of u, then update the distance value of v.

   2. Suppose one "optimal" location (maybe instead of G) must be selected for the fire station such that it minimizes the time to the farthest intersection. Devise an algorithm to solve this problem given an arbitrary road map. Analyze the running time complexity of your algorithm when there are f possible locations for the fire station (which must be at one of the intersections) and r possible roads.

      To find the optimal location that minimizes the time to the farthest intersection, we need to run Dijkstra's algorithm f number of times, trying out each vertex in the graph as


Dijkstra's Algorithm

the source vertex and finding which source location has the lowest maximum distance. Since we are using the adjacency list representation of graphs and a binary heap, the running time complexity of this algorithm would be $O(fr \log f)$.

$$\underset{v_i \in V}{\operatorname{argmin}} \max(dijkstra(G, v_i))$$

3. In the above graph what is the "optimal" location to place the fire station? Explain.

   The optimal location to place the fire station is location E. E has the lowest distance to the farthest intersection, which is 10.

```
In [3]: def dijkstra(G, V, src):
            dist = {}       # Array to store distance of vertices from source
            heap = Heap()   # Heap to store unselected vertices

            # Initialize min heap with all vertices
            # and distance array as all infinity
            i = 0
            for v in G.keys():
                dist[v] = float('inf')
                heap.array.append(heap.NewMinHeapNode(v, dist[v]))
                heap.pos[v] = i
                i += 1

            # Set the dist value of src node to 0
            dist[src] = 0
            heap.DecreaseKey(src, dist[src])

            # Initialize size of min heap to V, the number
            # of vertices in the graph
            heap.size = V

            # The loop continues until all the vertices
            # have been finalized
            while not heap.IsEmpty():
                # Extract the vertex with minimum distance value
                u = heap.ExtractMin()[0]

                # Traverse through all adjacent vertices of
                # u (the extracted vertex) and update their
                # distance values
                for neighbor in G[u]:
                    v = neighbor[0]

                    # If shortest distance to v is not finalized
                    # yet, and distance to v through u is less
                    # than its previously calculated distance
                    if heap.IsInMinHeap(v) and neighbor[1] + dist[u] < dist[v]:
                        dist[v] = neighbor[1] + dist[u]
                        heap.DecreaseKey(v, dist[v])

            return dist

        def find_optimal_location(G, V):
            max_dist, optimal_loc = min((max(dijkstra(G, V, v).values()), v) for v in G.keys())
            return max_dist, optimal_loc

In [4]: import graph as g
```

```python
def PrintDist(dist, src):
    print('Dijkstra\'s algorithm')
    print('Source: {0}'.format(src))
    print('Vertex\tDistance')
    for v, d in dist.items():
        print('{0}\t{1}'.format(v, d))
    print()

if __name__ == '__main__':
    graph = g.Graph(8)
    graph.AddEdge('A', 'F', 7)
    graph.AddEdge('A', 'C', 4)
    graph.AddEdge('C', 'F', 2)
    graph.AddEdge('C', 'G', 9)
    graph.AddEdge('C', 'D', 3)
    graph.AddEdge('F', 'G', 8)
    graph.AddEdge('G', 'D', 7)
    graph.AddEdge('G', 'E', 2)
    graph.AddEdge('G', 'H', 3)
    graph.AddEdge('D', 'E', 3)
    graph.AddEdge('E', 'H', 7)
    graph.AddEdge('E', 'B', 9)
    graph.AddEdge('H', 'B', 3)
    dist = graph.Dijkstra('G')
    PrintDist(dist, 'G')
    max_dist, optimal_loc = graph.OptimalLoc()
    print('Optimal location: {0}'.format(optimal_loc))
    print('Max distance: {0}'.format(max_dist))
```

```
Dijkstra's algorithm
Source: G
Vertex        Distance
A        12
F        8
C        8
G        0
D        5
E        2
H        3
B        6

Optimal location: E
Max distance: 10
```

3. Suppose there are two types of professional wrestlers: "Babyfaces" ("good guys") and "Heels" ("bad guys"). Between any pair of professional wrestlers, there may or may not be a rivalry. Suppose we have n wrestlers and we have a list of r pairs of rivalries.

    1. Give pseudocode for an efficient algorithm that determines whether it is possible to designate some of the wrestlers as Babyfaces and the remainder as Heels such that each rivalry is between a Babyface and a Heel. If it is possible to perform such a designation, your algorithm should produce it.

       This is a Graph-Bipartitioning problem. We want to split the vertices into two independent sets such that no 2 vertices from the same set are adjacent to one another.

       ```
       BIPARTITE(G, V, src)
           let color[1..V] be a new array
           color[1..V] = -1
           color[src] = 1
           q = queue()
           q.append(src)
           while q is not empty:
               u = q.pop()
               for v in adj(u):
                   if color[v] = -1:
                       color[v] = 1 - color[u]
                       q.append(v)
                   elif color[v] = color[u]:
                       return False
           return color
       ```

    2. What is the running time of your algorithm?

       The time complexity of the above approach is the same as a Breadth-First Search (BFS) which is $O(V + E)$ when using an adjacency list representation of graphs.

    3. **Implement**: Babyfaces vs Heels in C, C++ or Python. Name your program wrestler and include compile and executions instructions in the README file.

```
In [5]: def BipartiteUtil(G, color, src):
            # Assign first color to source
            color[src] = 1

            # Create a queue (FIFO) of vertices and enqueue
            # source vertex for BFS traversal
            queue = deque()
            queue.append(src)

            # Run while queue is not empty
            while queue:
                u = queue.popleft()

                # Return false if there is a self-loop
                if u in G[u]:
                    return False

                # Traverse through all adjacent vertices of
                # u (the extracted vertex) and update their
                # color assignment
                for neighbor in G[u]:
                    v = neighbor[0]

                    # If vertex v has not been visited yet
                    if color[v] == -1:
                        # Assign alternate color to v
                        color[v] = 1 - color[u]
                        queue.append(v)

                    # If v has been visited and it has the
                    # same color as u
                    elif color[v] == color[u]:
                        return False
            return True

        def Bipartite(G):

            # Create a map to store colors assigned to all
            # vertices. The value '-1' is used to indicate
            # that no color is assigned to the vertex. The
            # value 1 is used to indicate first color is
            # assigned and value 0 indicates second color
            # is assigned.
            color = {v:-1 for v in G.keys()}
            for v in color.keys():
                if color[v] == -1:
                    if not BipartiteUtil(color, v):
                        return False
```

```
    return color
```