Kruskal's Algorithm

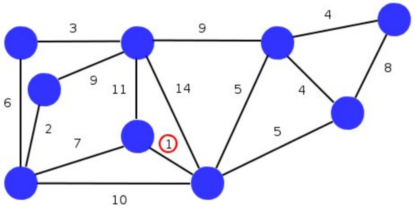
There are two MST algorithms based on the same greedy choice

Kruskal's Algorithm (Input: a connected, weighted graph $G = (V, E)$)

- Sort all edges in G by weight
- Put each vertex in G into a separate set
- For $(u, v) \in E$ (in order)
  - If u and v are in different sets
    - Add (u, v) to the MST
    - Combine u's set with v's set

- Gradually join $|V|$ components
- Add next lowest weight edge if it joins two components

Kruskal's Algorithm



- The set of edges is iterated over in weight order
- If the next edge connects two distinct components it is added

Implementing Kruskal's Algorithm

Kruskal's Algorithm (Input: a connected, weighted graph $G = (V, E)$)

- Sort all edges in G by weight
- Put each vertex in G into a separate set
- For $(u, v) \in E$ (in order)
    - If u and v are in different sets
        - Add (u, v) to the MST
        - Combine u's set with v's set

Question

How can the basic algorithm be implemented?

- What is returned?
- What data structures could be used?
- What would be the performance?

Kruskal's Algorithm: Implementation

Kruskal's Algorithm (Input: a connected, weighted graph $G = (V, E)$)

```
1     T = new Graph(V)
2     Add all edges in E to a queue Q prioritised by min weight
3     for v in V
4       Set Sv = {v}
5     while Q is not empty
6       {x,y} = Q.remove()
7       if x in Si and y in Sj and i != j
8         T.add_edge(x,y)
9         Si = Si + Sj
10        Sj = {}
11     return T
```

- T is a new graph, initialise with V (line 1), then add edges (line 8)
- Sorting or using priority queue are equivalent

Kruskal's Algorithm: Performance

Kruskal's Algorithm (Input: a connected, weighted graph $G = (V, E)$)

```
T = new Graph(V)
Add all edges in E to a queue Q prioritised by min weight
for v in V
  Set Sv = {v}
while Q is not empty
  {x,y} = Q.remove()
  if x in Si and y in Sj and i != j
    T.add_edge(x,y)
    Si = Si + Sj
    Sj = {}
 return T
```

Question

What is the time complexity?

Kruskal's Algorithm: Performance

Kruskal's Algorithm (Input: a connected, weighted graph $G = (V, E)$)

```
T = new Graph(V)
Add all edges in E to a queue Q prioritised by min weight
for v in V
  Set Sv = {v}          // use "disjoint sets" structure
while Q is not empty
  {x,y} = Q.remove()
  if x in Si and y in Sj and i != j
    T.add_edge(x,y)
    Si = Si + Sj
    Sj = {}
 return T
```

- The disjoint set data structure is $O(\log |V|)$ for all operations
- See books for details

Performance of Kruskal's Algorithm

For a graph with V vertices and E edges:

- Sorting the edges is $O(E \log_2 E)$
- Remainder depends on set operations

Operations on disjoint sets such as these possible in $O(\log V)$ time

- See disjoint set (Cormen) , union-find (Sedgewick) data structure

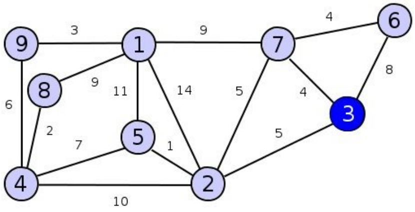
So, the loop to build the MST is $O(E \log_2 V)$

- $E < V^2$, so $\log_2 E < 2 \log_2 V$ and $E \log_2 E = O(E \log_2 V)$
- So, overall time is $O(E \log_2 V)$

Prim's Algorithm

Prim's Algorithm (Input: connected, weighted graph $G = (V, E)$, vertex r)

- Add r to MST
- While MST has fewer than $|V| - 1$ edges
  - Add least weight edge that connects MST to new vertex



- Focus on one component
- Only consider edges from that component

Prim's Algorithm: Implementation

Prim's Algorithm (Input: connected, weighted graph G, vertex r)

```
T = new Graph(G.num_vertices)
tree_vertex = new boolean[G.num_vertices]
tree_vertex[r] = true
Q = new MinPriorityQueue()                  // by weight
for v in G.adj[r] {  Q.add((r,v))  }
while T has fewer than |V| - 1 edges
  (x,y) = Q.remove()                 // tree_vertex[x] is true
  if not tree_vertex[y]
    tree_vertex[y] = true
    T.add_edge(x,y)
    for v in G.adj[y] { Q.add((y,v)) }
return T
```

- Just one set of vertices to track
- No new data structures needed

Prim's Algorithm

Discussion

What is the time complexity of Prim's algorithm?

Prim's Algorithm (Input: connected, weighted graph G, vertex r)

```
T = new Graph(G.num_vertices)
tree_vertex = new boolean[G.num_vertices]
tree_vertex[r] = true
Q = new MinPriorityQueue()                    // by weight
for v in G.adj[r] {  Q.add((r,v))  }
while T has fewer than |V| - 1 edges
  (x,y) = Q.remove()                // tree_vertex[x] is true
  if not tree_vertex[y]
    tree_vertex[y] = true
    T.add_edge(x,y)
    for v in G.adj[y] { Q.add((y,v)) }
return T
```

Performance of Prim's Algorithm

Prim's Algorithm also executes in $O(E \log_2 V)$ time assuming a queue implemented as a binary heap

- The queue operations determine the running time
- All edges are added to the queue
- Worst case: all edges removed from queue
- $E \log_2 E = O(E \log_2 V)$ as before