CS 225
Data Structures

April 23 – MST II
Brad Solomon
Learning Objectives

• Formalize Minimum Spanning Tree (MST)

• Analyze Kruskal and Prims’ respective algorithms

• Compare runtimes and implementation strategies
Minimum Spanning Tree Algorithms

**Input:** Connected, undirected graph $G$ with edge weights (unconstrained, but must be additive)

**Output:** A graph $G'$ with the following properties:
- $G'$ is a spanning graph of $G$
- $G'$ is a tree (connected, acyclic)
- $G'$ has a minimal total weight among all spanning trees
Kruskal’s Algorithm

KruskalMST(G):
DisjointSets forest
foreach (Vertex v : G):
    forest.makeSet(v)
PriorityQueue Q    // min edge weight
foreach (Edge e : G):
    Q.insert(e)
Graph T = (V, {})
while |T.edges()| < n-1:
    Vertex (u, v) = Q.removeMin()
    if forest.find(u) != forest.find(v):
        T.addEdge(u, v)
        forest.union( forest.find(u),
                      forest.find(v) )
return T
Kruskal’s Algorithm

KruskalMST(G):
DisjointSets forest
foreach (Vertex v : G):
    forest.makeSet(v)
PriorityQueue Q    // min edge weight
foreach (Edge e : G):
    Q.insert(e)
Graph T = (V, {})
while |T.edges()| < n-1:
    Vertex (u, v) = Q.removeMin()
    if forest.find(u) != forest.find(v):
        T.addEdge(u, v)
        forest.union( forest.find(u),
                      forest.find(v) )
return T

Priority Queue:

<table>
<thead>
<tr>
<th>Building</th>
<th>Heap</th>
<th>Sorted Array</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Line 6-8)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| Each removeMin            |      |              |
| (Line 13)                 |      |              |
Kruskal’s Algorithm

KruskalMST(G):

DisjointSets forest

foreach (Vertex v : G):
    forest.makeSet(v)

PriorityQueue Q    // min edge weight

foreach (Edge e : G):
    Q.insert(e)

Graph T = (V, {})

while |T.edges()| < n-1:
    Vertex (u, v) = Q.removeMin()
    if forest.find(u) != forest.find(v):
        T.addEdge(u, v)
        forest.union( forest.find(u),
                      forest.find(v) )

return T
Which Priority Queue Implementation is better for running Kruskal’s Algorithm?

- Heap:
- Sorted Array:
"The Muddy City" by CS Unplugged, Creative Commons BY-NC-SA 4.0
Consider an arbitrary partition of the vertices on $G$ into two subsets $U$ and $V$. Let $e$ be an edge of minimum weight across the partition. Then $e$ is part of some minimum spanning tree.
Partition Property

The partition property suggests an algorithm:
Prim’s Algorithm

```plaintext
1  PrimMST(G, s):
2      Input: G, Graph;
3          s, vertex in G, starting vertex
4  Output: T, a minimum spanning tree (MST) of G
5
6  foreach (Vertex v : G):
7      d[v] = +inf
8          p[v] = NULL
9      d[s] = 0
10
11     PriorityQueue Q   // min distance, defined by d[v]
12     Q.buildHeap(G.vertices())
13     Graph T           // "labeled set"
14
15     repeat n times:
16         Vertex u = Q.removeMin()
17         T.add(u)
18         foreach (Vertex v : neighbors of u not in T):
19             if cost(v, u) < d[v]:
20                 d[v] = cost(v, u)
21                 p[v] = u
22
23     return T
```
Prim’s Algorithm

```java
6  PrimMST(G, s):
7      foreach (Vertex v : G):
8          d[v] = +inf
9          p[v] = NULL
10         d[s] = 0
11
12         PriorityQueue Q // min distance, defined by d[v]
13         Q.buildHeap(G.vertices())
14         Graph T         // "labeled set"
15
16         repeat n times:
17             Vertex u = Q.removeMin()
18             T.add(u)
19             foreach (Vertex v : neighbors of u not in T):
20                 if cost(v, u) < d[v]:
21                     d[v] = cost(v, u)
22                     p[v] = u
```

<table>
<thead>
<tr>
<th>Adj. Matrix</th>
<th>Adj. List</th>
</tr>
</thead>
<tbody>
<tr>
<td>Heap</td>
<td></td>
</tr>
<tr>
<td>Unsorted Array</td>
<td></td>
</tr>
</tbody>
</table>
Prim’s Algorithm

PrimMST(G, s):

foreach (Vertex v : G):
    d[v] = +inf
    p[v] = NULL
    d[s] = 0

PriorityQueue Q // min distance, defined by d[v]
Q.buildHeap(G.vertices())
Graph T         // "labeled set"

repeat n times:
    Vertex u = Q.removeMin()
    T.add(u)
    foreach (Vertex v : neighbors of u not in T):
        if cost(v, u) < d[v]:
            d[v] = cost(v, u)
            p[v] = u

<table>
<thead>
<tr>
<th>Adj. Matrix</th>
<th>Adj. List</th>
</tr>
</thead>
<tbody>
<tr>
<td>Heap</td>
<td>$O(n^2 + m \lg(n))$</td>
</tr>
<tr>
<td>Unsorted Array</td>
<td>$O(n^2)$</td>
</tr>
</tbody>
</table>
MST Algorithm Runtime:

- Kruskal’s Algorithm: $O(n + m \ lg(n))$
- Prim’s Algorithm: $O(n \ lg(n) + m \ lg(n))$

- What must be true about the connectivity of a graph when running an MST algorithm?

- How does $n$ and $m$ relate?
MST Algorithm Runtime:

• Kruskal’s Algorithm: \( O(n + m \lg(n)) \)

• Prim’s Algorithm: \( O(n \lg(n) + m \lg(n)) \)

Sparse Graph:

Dense Graph:
Suppose I have a new heap:

```
PrimMST(G, s):
  foreach (Vertex v : G):
    d[v] = +inf
    p[v] = NULL
    d[s] = 0
  PriorityQueue Q // min distance, defined by d[v]
  Q.buildHeap(G.vertices())
  Graph T         // "labeled set"
  repeat n times:
    Vertex m = Q.removeMin()
    T.add(m)
    foreach (Vertex v : neighbors of m not in T):
      if cost(v, m) < d[v]:
        d[v] = cost(v, m)
        p[v] = m
```

What’s the updated running time?

<table>
<thead>
<tr>
<th></th>
<th>Binary Heap</th>
<th>Fibonacci Heap</th>
</tr>
</thead>
<tbody>
<tr>
<td>Remove Min</td>
<td>O( lg(n) )</td>
<td>O( lg(n) )</td>
</tr>
<tr>
<td>Decrease Key</td>
<td>O( lg(n) )</td>
<td>O(1)*</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Binary Heap</th>
<th>Fibonacci Heap</th>
</tr>
</thead>
<tbody>
<tr>
<td>Remove Min</td>
<td>O( lg(n) )</td>
</tr>
<tr>
<td>Decrease Key</td>
<td>O(1)*</td>
</tr>
</tbody>
</table>
Final Big-O MST Algorithm Runtimes:

• Kruskal’s Algorithm: \( O(m \lg(n)) \)

• Prim’s Algorithm: \( O(n \lg(n) + m) \)

Sparse Graph:

Dense Graph: