CMPSC 250

Analysis of algorithms

Dr. Aravind Mohan

Allegheny College
Weighted Graph

- Each edge has an associated numerical value, called **weight**
- Usually, the weights are non-negative integers
- Weighted graphs can be directed or undirected
Each edge has an associated numerical value, called **weight**.
- Usually, the weights are non-negative integers.
- Weighted graphs can be directed or undirected.
Weighted Edge API

- Edge abstraction needed for weighted edges
Weighted Edge API

- Edge abstraction needed for weighted edges

```java
public class Edge implements Comparable<Edge>

Edge(int v, int w, double weight)  // create a weighted edge v-w
int either()                      // either endpoint
int other(int v)                  // the endpoint that's not v
int compareTo(Edge that)          // compare this edge to that edge
double weight()                   // the weight
String toString()                 // string representation
```
Weighted edge: Java implementation

```java
public class Edge implements Comparable<Edge> {
    private final int v, w;
    private final double weight;

    public Edge(int v, int w, double weight) {
        this.v = v;
        this.w = w;
        this.weight = weight;
    }

    public int either() {
        return v;
    }

    public int other(int vertex) {
        if (vertex == v) return w;
        else return v;
    }

    public int compareTo(Edge that) {
        if (this.weight < that.weight) return -1;
        else if (this.weight > that.weight) return +1;
        else return 0;
    }
}
```
public class EdgeWeightedGraph

EdgeWeightedGraph(int V) 
create an empty graph with V vertices

EdgeWeightedGraph(In in) 
create a graph from input stream

void addEdge(Edge e) 
add weighted edge e to this graph

Iterable<Edge> adj(int v) 
edges incident to v

Iterable<Edge> edges() 
all edges in this graph

int V() 
number of vertices

int E() 
number of edges

String toString() 
string representation
Edge-weighted graph: adjacency-lists

Representation

Maintain vertex-indexed array of Edge lists.

```
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>6 0.58</td>
<td>0 2.26</td>
<td>0 4.38</td>
<td>0 7.16</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 3.29</td>
<td>1 2.36</td>
<td>1 7.19</td>
<td>1 5.32</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6 2.40</td>
<td>2 7.34</td>
<td>1 2.36</td>
<td>0 2.26</td>
<td>2 3.17</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3 6.52</td>
<td>1 3.29</td>
<td>2 3.17</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6 4.93</td>
<td>0 4.38</td>
<td>4 7.37</td>
<td>4 5.35</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 5.32</td>
<td>5 7.28</td>
<td>4 5.35</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6 4.93</td>
<td>6 0.58</td>
<td>3 6.52</td>
<td>6 2.40</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2 7.34</td>
<td>1 7.19</td>
<td>0 7.16</td>
<td>5 7.28</td>
<td>5 7.28</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
adjacency-lists representation

```java
public class EdgeWeightedGraph {
    private final int V;
    private final Bag<Edge>[] adj;

    public EdgeWeightedGraph(int V) {
        this.V = V;
        adj = (Bag<Edge>[]) new Bag[V];
        for (int v = 0; v < V; v++)
            adj[v] = new Bag<Edge>();
    }

    public void addEdge(Edge e) {
        int v = e.either(), w = e.other(v);
        adj[v].add(e);
        adj[w].add(e);
    }

    public Iterable<Edge> adj(int v) {
        return adj[v];
    }
}
```

- same as `Graph`, but adjacency lists of `Edges` instead of integers
- constructor
- add edge to both adjacency lists
Minimum Spanning Tree

Spanning Tree:
is a subgraph that is connected and acyclic

Goal: Find a minimum weight spanning tree
Minimum Spanning Tree

Spanning Tree:
is a subgraph that is connected and acyclic

Goal: Find a minimum weight spanning tree

![Graph with weights and labels](image)
Minimum Spanning Tree

Spanning Tree:

is a subgraph that is connected and acyclic

Goal: Find a minimum weight spanning tree
Minimum Spanning Tree

Spanning Tree:
is a subgraph that is connected and acyclic

Goal: Find a minimum weight spanning tree

spanning tree $T$: cost = $50 = 4 + 6 + 8 + 5 + 11 + 9 + 7$
MST of bicycle routes in North Seattle

Network design
public class MST

MST(EdgeWeightedGraph G) // constructor

Iterable<Edge> edges() // edges in MST

double weight() // weight of MST

% java MST tinyEWG.txt
0-7 0.16
1-7 0.19
0-2 0.26
2-3 0.17
5-7 0.28
4-5 0.35
6-2 0.40
1.81
Minimum spanning tree API

```java
public class MST {
    MST(EdgeWeightedGraph G) {
        constructor
    }
    Iterable<Edge> edges() {
        edges in MST
    }
    double weight() {
        weight of MST
    }
}

public static void main(String[] args) {
    In in = new In(args[0]);
    EdgeWeightedGraph G = new EdgeWeightedGraph(in);
    MST mst = new MST(G);
    for (Edge e : mst.edges())
        StdOut.println(e);
    StdOut.printf("%.2f\n", mst.weight());
}

% java MST tinyEWG.txt
0-7 0.16
1-7 0.19
0-2 0.26
2-3 0.17
5-7 0.28
4-5 0.35
6-2 0.40
1.81
```
Efficient implementations

- Prim’s algorithm
- Kruskal’s algorithm
Removing assumptions

Q. What if edge weights are not all distinct?
A. Greedy MST algorithm still correct if equal weights are present! (our correctness proof fails, but that can be fixed)

Q. What if graph is not connected?
A. Compute minimum spanning forest = MST of each component.
Prim’s algorithm

- Start with vertex 0 and greedily grow tree $T$
- At each step, add to $T$ the min weight edge with exactly one endpoint in $T$
Prim’s algorithm demo

Initialize $S = \text{any node}$.
Repeat $n - 1$ times:
- Add to tree the min weight edge with one endpoint in $S$.
- Add new node to $S$. 

Diagram:

- Nodes: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13
- Edges: 1-6, 6-8, 3-5, 11-2, 5-10, 3-12, 7-3, 7-1, 7-4
Prim’s algorithm demo

Initialize $S = \text{any node.}$

Repeat $n - 1$ times:
  - Add to tree the min weight edge with one endpoint in $S$.
  - Add new node to $S$. 

![Diagram of a network with Prim's algorithm steps demonstrated.](image-url)
Prim’s algorithm demo

Initialize $S = \text{any node.}$
Repeat $n - 1$ times:
- Add to tree the min weight edge with one endpoint in $S$.
- Add new node to $S$. 
Prim’s algorithm demo

Initialize $S =$ any node.
Repeat $n - 1$ times:
• Add to tree the min weight edge with one endpoint in $S$.
• Add new node to $S$. 

![Diagram of a network graph with weights on edges and a shaded path starting from a node.](image)
Prim’s algorithm demo

Initialize $S$ = any node.
Repeat $n-1$ times:
  • Add to tree the min weight edge with one endpoint in $S$.
  • Add new node to $S$. 
Prim’s algorithm demo

Initialize $S =$ any node.
Repeat $n - 1$ times:
  • Add to tree the min weight edge with one endpoint in $S$.
  • Add new node to $S$. 

![Graph](image)
Prim’s algorithm demo

Initialize $S = \text{any node.}$
Repeat $n - 1$ times:
  • Add to tree the min weight edge with one endpoint in $S$. 
  • Add new node to $S$. 

Diagram of a network with selected edges highlighted.
Prim’s algorithm demo

Initialize $S = \text{any node}$.
Repeat $n - 1$ times:
• Add to tree the min weight edge with one endpoint in $S$.
• Add new node to $S$. 

![Graph diagram with Prim's algorithm application]
Prim’s algorithm demo

Initialize $S$ = any node.
Repeat $n - 1$ times:
- Add to tree the min weight edge with one endpoint in $S$.
- Add new node to $S$. 
Prim’s algorithm demo

Initialize $S =$ any node.
Repeat $n - 1$ times:
• Add to tree the min weight edge with one endpoint in $S$.
• Add new node to $S$. 

![Graph with Prim's algorithm applied](image)
Prim’s algorithm demo

Initialize $S = \text{any node.}$
Repeat $n - 1$ times:
• Add to tree the min weight edge with one endpoint in $S$.
• Add new node to $S$. 

---

Diagram of the Prim's algorithm process.
Prim’s algorithm demo

Initialize $S = \text{any node}$. 
Repeat $n - 1$ times:
  * Add to tree the min weight edge with one endpoint in $S$.
  * Add new node to $S$. 

![Diagram of Prim's algorithm](image)
Prim’s algorithm demo

Initialize $S$ = any node.
Repeat $n - 1$ times:
  - Add to tree the min weight edge with one endpoint in $S$.
  - Add new node to $S$. 

Diagram showing the steps of Prim’s algorithm.
Prim’s algorithm demo

Initialize $S = $ any node.
Repeat $n - 1$ times:
- Add to tree the min weight edge with one endpoint in $S$.
- Add new node to $S$. 

![Diagram of Prim's algorithm process]
Prim’s algorithm demo

Initialize $S = $ any node.
Repeat $n - 1$ times:
- Add to tree the min weight edge with one endpoint in $S$.
- Add new node to $S$. 
Prim’s algorithm demo

Initialize $S = \text{any node.}$
Repeat $n - 1$ times:
  * Add to tree the min weight edge with one endpoint in $S$.
  * Add new node to $S$. 

[Diagram of a network with nodes and edges]
Prim’s algorithm demo

Initialize $S = \text{any node}$.

Repeat $n - 1$ times:
• Add to tree the min weight edge with one endpoint in $S$.
• Add new node to $S$.
Prim’s Algorithm: running time

Recall

... some constant-time steps omitted ...
repeat until all vertices are marked:
  add edges adjacent to \( v \) to the priority queue
  delete minimum-weight edge \( e \) from priority queue
  if \( e \)'s endpoints are both marked, go back to previous step
... some constant-time steps omitted ...
Proposition

Kruskal’s algorithm computes the MST in time proportional to $E \log E$ and extra space proportional to $E$ in the worst case.
Proposition

Kruskal’s algorithm computes the MST in time proportional to $E \log E$ and extra space proportional to $E$ in the worst case.

<table>
<thead>
<tr>
<th>operation</th>
<th>frequency</th>
<th>binary heap</th>
</tr>
</thead>
<tbody>
<tr>
<td>delete min</td>
<td>$E$</td>
<td>$\log E$</td>
</tr>
<tr>
<td>insert</td>
<td>$E$</td>
<td>$\log E$</td>
</tr>
</tbody>
</table>
### Prim’s Algorithm: Priority Queue implementation

<table>
<thead>
<tr>
<th>PQ implementation</th>
<th>insert</th>
<th>delete-min</th>
<th>decrease-key</th>
<th>total</th>
</tr>
</thead>
<tbody>
<tr>
<td>array</td>
<td>1</td>
<td>V</td>
<td>1</td>
<td>$V^2$</td>
</tr>
<tr>
<td>binary heap</td>
<td>$\log V$</td>
<td>$\log V$</td>
<td>$\log V$</td>
<td>$E \log V$</td>
</tr>
<tr>
<td>d-way heap (Johnson 1975)</td>
<td>$d \log_d V$</td>
<td>$d \log_d V$</td>
<td>$\log_d V$</td>
<td>$E \log_{E/V} V$</td>
</tr>
<tr>
<td>Fibonacci heap</td>
<td>$1^\dagger$</td>
<td>$\log V^\dagger$</td>
<td>$1^\dagger$</td>
<td>$E + V \log V$</td>
</tr>
</tbody>
</table>
Kruskal’s Algorithm [1956]

- Consider edges in ascending order of weight
- Add the next edge to the tree $T$ unless doing so would create a cycle.
Correctness

Proposition

Kruskal’s algorithm computes the MST
Correctness

Proposition

Kruskal’s algorithm computes the MST
Kruskal’s algorithm: Java implementation

class KruskalMST
{
  private Queue<Edge> mst = new Queue<Edge>();
  public KruskalMST(EdgeWeightedGraph G)
  {
    MinPQ<Edge> pq = new MinPQ<Edge>();
    for (Edge e : G.edges())
      pq.insert(e);
    UF uf = new UF(G.V());
    while (!pq.isEmpty() && mst.size() < G.V()-1)
    {
      Edge e = pq.delMin();
      int v = e.either(), w = e.other(v);
      if (!uf.connected(v, w))
      {
        uf.union(v, w);
        mst.enqueue(e);
      }
    }
  }
  public Iterable<Edge> edges()
  {
    return mst;
  }
}
Kruskal’s algorithm: running time

Proposition

Kruskal’s algorithm computes MST in time proportional to $E \log E$ in the worst case.

<table>
<thead>
<tr>
<th>operation</th>
<th>frequency</th>
<th>time per op</th>
</tr>
</thead>
<tbody>
<tr>
<td>build pq</td>
<td>1</td>
<td>$E$</td>
</tr>
<tr>
<td>delete-min</td>
<td>$E$</td>
<td>$\log E$</td>
</tr>
<tr>
<td>union</td>
<td>$V$</td>
<td>$\log^* V$</td>
</tr>
<tr>
<td>connected</td>
<td>$E$</td>
<td>$\log^* V$</td>
</tr>
</tbody>
</table>

Note: If edges are already sorted, then order of growth is $E \log^* V$, where $\log^* V \leq 5$. 
Does linear MST exist?

<table>
<thead>
<tr>
<th>year</th>
<th>worst case</th>
<th>discovered by</th>
</tr>
</thead>
<tbody>
<tr>
<td>1975</td>
<td>$E \log \log V$</td>
<td>Yao</td>
</tr>
<tr>
<td>1976</td>
<td>$E \log \log V$</td>
<td>Cheriton-Tarjan</td>
</tr>
<tr>
<td>1984</td>
<td>$E \log^* V, E + V \log V$</td>
<td>Fredman-Tarjan</td>
</tr>
<tr>
<td>1986</td>
<td>$E \log (\log^* V)$</td>
<td>Gabow-Galil-Spencer-Tarjan</td>
</tr>
<tr>
<td>1997</td>
<td>$E \alpha(V) \log \alpha(V)$</td>
<td>Chazelle</td>
</tr>
<tr>
<td>2000</td>
<td>$E \alpha(V)$</td>
<td>Chazelle</td>
</tr>
<tr>
<td>2002</td>
<td>optimal</td>
<td>Pettie-Ramachandran</td>
</tr>
<tr>
<td>20xx</td>
<td>$E$</td>
<td>???</td>
</tr>
</tbody>
</table>
Textbook Reading

- Chapter 4: pages 604 - 629
Class activity, post your solutions in slack

Graph G

- Given graph above, find MST using the Prims algorithm
Class activity, post your solutions in slack

Given graph above, find MST using the Kruskals algorithm