

Graphs 1

Reading: CLR Sections 5.4 -- 5.5; Chapter 24, Sections 25.1 -- 25.2

Minimum Spanning Trees

Definitions:

A **weighted graph** is a graph (V, E) together with a weighting function $w: E \rightarrow \text{Reals}$.

The **weight** of a graph is $\sum_{e \in E} w(e)$.

A **(free) tree** is a connected acyclic undirected graph.

A **spanning tree** of a graph (V, E) is a free tree (V, E') where $E' \subseteq E$.

A **sub-spanning tree** of a graph (V, E) is a free tree (V', E') where $V' \subseteq V$ and $E' \subseteq E$.
(My terminology)

A **minimum (weight) spanning tree (MST)** of a connected, undirected graph G is a spanning tree of G with minimal weight. (It may not be unique.)

Skeleton of Greedy MST Algorithm

The following is the skeleton of a greedy MST algorithm. The skeleton can be instantiated to both Prim's algorithm and Kruskal's algorithm.

Idea: Grow a set of edges ST that is a subset of a spanning tree of G . At each step, extend ST by the "best" safe edge -- i.e., the "best" edge that maintains the invariant that ST is a subset of a minimum spanning tree of G .

```

MST( $G, w$ )
   $ST \leftarrow \{\}$ 
   $D \leftarrow \text{Init-Data}(G)$  {Initialize auxiliary data structure}
  while not Is-Spanning-Tree?( $ST, D$ ) do
    {Invariant:  $ST$  is the subset of a spanning tree.}
    ( $a, b$ )  $\leftarrow$  Find-Safe-Edge( $ST, w, D$ )
     $ST \leftarrow ST \cup \{(a, b)\}$ 
  return  $ST$ 

```

Note: The spanning tree is represented by the edge set ST , from which the vertices can be unambiguously derived.

Prim's Algorithm

Idea: Grow a single sub-spanning tree. At each step, add the least weight edge connecting a vertex not in the tree with a vertex in the tree.

```

Init-Data(G)
  root  Choose-Root(vertices(G))
  for v in vertices(G) do
    min-weight[v]
    min-parent[v]  nil
    in-tree?[v]   false
  min-weight[root]  0
  Q  Build-PQ(vertices(G)) {Priority queue ordered by min-weight.}
  Find-Safe-Vertex({}, w, Q)
  return Q

Is-Spanning-Tree?(ST, Q)
  return PQ-Empty?(Q)

Find-Safe-Edge(ST, w, Q)
  v  Find-Safe-Vertex(Q)
  return (min-parent(v), v)

Find-Safe-Vertex(ST, w, Q)
  a  PQ-Extract-Min(Q)
  in-tree?[a]  true
  for b in Adj[a] do
    if not in-tree?[b] and w(a,b) < min-weight[b] then
      PQ-Decrease-Key(Q, b, w(a,b))
      min-parent(b)  a
  return a

```

Analysis:

- Build-PQ called on $|V|$ vertices
- PQ-Extract-Min called once for each of $|V|$ vertices
- PQ-Decrease-Key called at most once for each of $|E|$ edges

Priority Queue implementation	Build-PQ	$ V \cdot$ PQ-Extract-Min	$ E \cdot$ PQ-Decrease-Key	Total
unsorted array/list	$O(V)$	$O(V^2)$	$O(E)$	$O(V^2)$
binary heap	$O(V)$	$O(V \cdot \lg(V))$	$O(E \cdot \lg(V))$	$O(E \cdot \lg(V))$
Fibonacci heap	$O(V)$	$O(V \cdot \lg(V))$	$O(E)$	$O(V \cdot \lg(V) + E)$

Note: A Fibonacci heap is a heap-like data structure in which Extract-Min takes $O(\lg(n))$ amortized cost for n nodes, and Decrease-Key takes $O(1)$ amortized cost for n nodes.

Kruskal's Algorithm

Idea: Grow a spanning forest = a set of sub-spanning trees whose vertex sets are disjoint. Initially, each vertex is a trivial sub-spanning tree. At each step, add the least-weight edge between two distinct sub-spanning trees. This "glues" the two trees into a single tree. Eventually there will be a single spanning tree.

```
Init-Data(G)
  sorted-edges  sort(edges[G]) {By increasing weight}
  partitions    {}
  for v in vertices(G) do
    partitions  partitions U Singleton-Partition(v)
  return (sorted-edges, partitions)

Is-Spanning-Tree?(ST, (edges, partitions))
  return Is-Singleton?(partitions)

Find-Safe-Edge(ST, w, (edges, partitions))
  (a, b)  Remove-First(sorted-edges)
  if not Same-Partition?(partition(a), partition(b))
    partitions  Union-Partitions(a, b, partitions)
  return (a, b)
```

Analysis:

- Initialization: $O(V)$
- Sorting edges: $O(E \lg(E))$
- At most $|E|$ calls to Same-Partition and Union, each of which costs $O(\lg(E))$: $O(E \lg(E))$. (Actually, the time per operation is the inverse Ackerman function of E and V , which grows far more slowly than a logarithm.)
- Total: $O(E \lg(E))$, which = $O(E \lg(V))$ since $E = O(V^2)$ and $O(\lg(V^2)) = O(\lg(V))$

Single-Source Shortest Paths

Definitions:

In a weighted, directed graph (G, w) , the **weight of a path** $p = (v_0, v_1, \dots, v_k)$ is

$$w(p) = \sum_{i=1}^k w(v_{i-1}, v_i).$$

The **shortest-path weight** from a to b is $w(a,b) = \min\{w(p) \mid p \text{ paths}(a,b)\}$.

Note: $\min\{\} = \infty$.

A **shortest path** from a to b is any path p such that $w(p) = w(a,b)$. (May not be unique.)

The **single-source shortest path problem**: given a weighted directed graph $((V, E), w)$ and a source vertex s in V , find a shortest path from s to every vertex of V .

Note: If a path has negative weight edges in a cycle, then shortest path is not defined. Some algorithms (like the Dijkstra algorithm we will study) assume non-negative weights. Other algorithms (such as Bellman-Ford, which we will not study) can handle negative weight edges as long as they don't appear in cycles.

Breadth First Search

In the simple case where all weights = 1, can expand a "frontier" outward from the source, level by level. We can color the nodes according to the following scheme:

- white = undiscovered
- gray = frontier node = discovered node whose edges have not been processed
- black = fully processed = discovered node directly connected only to other discovered nodes

```
BFS(G,s)
  {Initialization}
  for v in vertices(G) do
    color[v]  white  {All nodes originally undiscovered}
    d[v]
    parent[v]  nil
  color[s]  gray
  d[s]  0
  Q  Enq(s, Empty-Queue)  {Invariant: Q contains only gray nodes.}
  while not Empty-Queue?(Q) do
    f  Deq(Q)  {Next frontier node to process.}
    for g in Adj[f] do
      if color[g] = white then
        color[g] = gray
        d[g] = d[f] + 1
        parent[g] = f
        Enq(g, Q)
    color[f] = black  {Color node black when completely processed.}
```

Analysis:

- Each vertex enqueued and dequeued once at $O(1)$ time per operation: $O(V)$
- Each edge scanned once: $O(E)$
- Total = $O(V + E)$ (= $O(E)$ for a connected graph)

Relaxation

Shortest path algorithms maintain for each vertex in the graph:

- a **shortest-path estimate** $d_s[v]$ (s,v) and
- a **shortest-path parent** $\text{parent}[v]$ such that $d_s[v] = d_s[\text{parent}[v]] + w(\text{parent}[v], v)$.

```
Initialize-Single-Source(G,s)
  for v vertices(G) do
     $d_s[v] = \infty$ 
     $\text{parent}[v] \leftarrow \text{nil}$ 
   $d[s] = 0$ 
```

Relaxation is an operation on edges (a, b) that attempts to reduce the shortest-path estimate for b .

```
Relax((a, b) w)
  if  $d_s[b] > d_s[a] + w(a,b)$  then
     $d_s[b] = d_s[a] + w(a,b)$ 
     $\text{parent}[b] = a$ 
```

Shortest path algorithms work by initializing $d_s[v]$ as above and then repeatedly relaxing edges until $d_s[v] = d_s^*(s,v)$. A **shortest-path tree** is induced by the parent fields.

Dijkstra's Algorithm

Idea: Grow a shortest path tree from the source. Every node maintains a shortest path estimate and parent that indicates the final edge of the estimate shortest path. At each step, add the node with the smallest estimate to the tree via the edge to its parent.

```

{Note: PQ stands for Priority Queue}
Dijkstra(G, w, s)
  Initialize-Single-Source(G,s)
  shortest  {}                                {vertices at which  $d_s[v] = (s,v)$ }
  Q        Build-PQ (vertices(G))             {Invariant: Q contains
                                              (V - shortest) ordered by  $d_s[v]$ }

  while not PQ-Empty?(Q) do
    a      PQ-Extract-Min(Q)
    shortest  shortest  {a}
    for b  Adj[a] do
      Relax((a,b), w)
      PQ-Decrease-Key(Q, b,  $d_s[b]$ )

```

Analysis:

- Build-PQ called once on $|V|$ elements
- PQ-Extract-Min called $|V|$ times
- Relax and Decrease-Key called $|E|$ times.

Priority Queue implementation	Build-PQ	$ V \cdot$ PQ-Extract-Min	$ E \cdot$ PQ-Decrease-Key	Total
unsorted array/list	$O(V)$	$O(V^2)$	$O(E)$	$O(V^2)$
binary heap	$O(V)$	$O(V \cdot \lg(V))$	$O(E \cdot \lg(V))$	$O(E \cdot \lg(V))$
Fibonacci heap	$O(V)$	$O(V \cdot \lg(V))$	$O(E)$	$O(V \cdot \lg(V) + E)$