Graph Algorithms
shortest paths, minimum spanning trees, etc.
Outline and Reading

Graphs (§12.1)
- Definition
- Applications
- Terminology
- Properties
- ADT

Data structures for graphs (§12.2)
- Edge list structure
- Adjacency list structure
- Adjacency matrix structure
Minimum Spanning Trees
Outline and Reading

- Minimum Spanning Trees (§12.7)
  - Definitions
  - A crucial fact
- The Prim-Jarnik Algorithm (§12.7.2)
- Kruskal's Algorithm (§12.7.1)
Reminder: Weighted Graphs

- In a weighted graph, each edge has an associated numerical value, called the weight of the edge.
- Edge weights may represent distances, costs, etc.
- Example:
  - In a flight route graph, the weight of an edge represents the distance in miles between the endpoint airports.
Minimum Spanning Tree

- **Spanning subgraph**
  - Subgraph of a graph $G$ containing all the vertices of $G$
- **Spanning tree**
  - Spanning subgraph that is itself a (free) tree
- **Minimum spanning tree (MST)**
  - Spanning tree of a weighted graph with minimum total edge weight
- **Applications**
  - Communications networks
  - Transportation networks

Graphs
6
Exercise: MST

Show an MSF of the following graph.
Cycle Property

Cycle Property:
- Let $T$ be a minimum spanning tree of a weighted graph $G$
- Let $e$ be an edge of $G$ that is not in $T$ and $C$ let be the cycle formed by $e$ with $T$
- For every edge $f$ of $C$, $weight(f) \leq weight(e)$

Proof:
- By contradiction
- If $weight(f) > weight(e)$ we can get a spanning tree of smaller weight by replacing $e$ with $f$
Partition Property

Partition Property:

- Consider a partition of the vertices of $G$ into subsets $U$ and $V$
- Let $e$ be an edge of minimum weight across the partition
- There is a minimum spanning tree of $G$ containing edge $e$

Proof:

- Let $T$ be an MST of $G$
- If $T$ does not contain $e$, consider the cycle $C$ formed by $e$ with $T$ and let $f$ be an edge of $C$ across the partition
- By the cycle property, $\text{weight}(f) \leq \text{weight}(e)$
- Thus, $\text{weight}(f) = \text{weight}(e)$
- We obtain another MST by replacing $f$ with $e$
Prim-Jarnik’s Algorithm

- (Similar to Dijkstra’s algorithm, for a connected graph)
- We pick an arbitrary vertex \( s \) and we grow the MST as a cloud of vertices, starting from \( s \)
- We store with each vertex \( v \) a label \( d(v) = \text{the smallest weight of an edge connecting } v \text{ to a vertex in the cloud} \)
- At each step:
  - We add to the cloud the vertex \( u \) outside the cloud with the smallest distance label
  - We update the labels of the vertices adjacent to \( u \)
Prim-Jarnik’s Algorithm (cont.)

- A priority queue stores the vertices outside the cloud
  - Key: distance
  - Element: vertex
- Locator-based methods
  - `insert(k,e)` returns a locator
  - `replaceKey(l,k)` changes the key of an item
- We store three labels with each vertex:
  - Distance
  - Parent edge in MST
  - Locator in priority queue

```
Algorithm PrimJarnikMST(G)
Q ← new heap-based priority queue
s ← a vertex of G
for all v ∈ G.vertices()
    if v = s
        setDistance(v, 0)
    else
        setDistance(v, ∞)
setParent(v, Ø)
l ← Q.insert(getDistance(v), v)
setLocator(v, l)
while ¬Q.isEmpty()
    u ← Q.removeMin()
    for all e ∈ G.incidentEdges(u)
        z ← G.opposite(u, e)
        r ← weight(e)
        if r < getDistance(z)
            setDistance(z, r)
            setParent(z, e)
    Q.replaceKey(getLocator(z), r)
```
Example

Graphs
Example (contd.)
Exercise: Prim’s MST alg

- Show how Prim’s MST algorithm works on the following graph, assuming you start with SFO, i.e., s=SFO.
  - Show how the MST evolves in each iteration (a separate figure for each iteration).
Analysis

• Graph operations
  - Method incidentEdges is called once for each vertex
• Label operations
  - We set/get the distance, parent and locator labels of vertex $z$ $O(\text{deg}(z))$ times
  - Setting/getting a label takes $O(1)$ time
• Priority queue operations
  - Each vertex is inserted once into and removed once from the priority queue, where each insertion or removal takes $O(\log n)$ time
  - The key of a vertex $w$ in the priority queue is modified at most $\text{deg}(w)$ times, where each key change takes $O(\log n)$ time
• Prim-Jarnik’s algorithm runs in $O((n + m) \log n)$ time provided the graph is represented by the adjacency list structure
  - Recall that $\sum_v \text{deg}(v) = 2m$
  - The running time is $O(m \log n)$ since the graph is connected
Kruskal’s Algorithm

- A priority queue stores the edges outside the cloud
  - Key: weight
  - Element: edge
- At the end of the algorithm
  - We are left with one cloud that encompasses the MST
  - A tree $T$ which is our MST

**Algorithm** $\text{KruskalMST}(G)$

```
for each vertex $V$ in $G$ do
    define a $\text{Cloud}(v)$ of $\leftarrow \{v\}$
let $Q$ be a priority queue.
Insert all edges into $Q$ using their weights as the key
$T \leftarrow \emptyset$
while $T$ has fewer than $n-1$ edges do
    edge $e = T.removeMin()$
    Let $u, v$ be the endpoints of $e$
    if $\text{Cloud}(v) \neq \text{Cloud}(u)$ then
        Add edge $e$ to $T$
        Merge $\text{Cloud}(v)$ and $\text{Cloud}(u)$
return $T$
```
Data Structure for Kruskal Algorithm

- The algorithm maintains a forest of trees
- An edge is accepted if it connects distinct trees
- We need a data structure that maintains a partition, i.e., a collection of disjoint sets, with the operations:
  - `find(u)`: return the set storing u
  - `union(u,v)`: replace the sets storing u and v with their union
Representation of a Partition

- Each set is stored in a sequence
- Each element has a reference back to the set
  - operation \texttt{find}(u) takes $O(1)$ time, and returns the set of which $u$ is a member.
  - in operation \texttt{union}(u,v), we move the elements of the smaller set to the sequence of the larger set and update their references
  - the time for operation \texttt{union}(u,v) is $\min(n_u,n_v)$, where $n_u$ and $n_v$ are the sizes of the sets storing $u$ and $v$
- Whenever an element is processed, it goes into a set of size at least double, hence each element is processed at most $\log n$ times
Partition-Based Implementation

• A partition-based version of Kruskal’s Algorithm performs cloud merges as unions and tests as finds.

Algorithm Kruskal(G):

Input: A weighted graph G.
Output: An MST T for G.

Let \( P \) be a partition of the vertices of \( G \), where each vertex forms a separate set.
Let \( Q \) be a priority queue storing the edges of \( G \), sorted by their weights.
Let \( T \) be an initially-empty tree

while \( Q \) is not empty do
    \((u, v) \leftarrow Q\).removeMinElement()
    if \( P\).find(u) \( \neq \) \( P\).find(v) then
        Add \((u, v)\) to \( T \)
        \( P\).union(u, v)
return \( T \)

Running time: \( O((n+m)\log n) \)
Kruskal Example

Graphs
Example
Example
Example
Example
Example
Example

Graphs
Example

Graphs
Example
Example

Graphs
Example

Graphs
Example
Example
Exercise: Kruskal’s MST algorithm

- Show how Kruskal’s MST algorithm works on the following graph.
  - Show how the MST evolves in each iteration (a separate figure for each iteration).
Shortest Paths
Outline and Reading

- Weighted graphs (§12.1)
  - Shortest path problem
  - Shortest path properties

- Dijkstra’s algorithm (§12.6.1)
  - Algorithm
  - Edge relaxation

- The Bellman-Ford algorithm

- Shortest paths in DAGs

- All-pairs shortest paths
**Weighted Graphs**

- In a weighted graph, each edge has an associated numerical value, called the weight of the edge.
- Edge weights may represent distances, costs, etc.
- Example:
  - In a flight route graph, the weight of an edge represents the distance in miles between the endpoint airports.
Shortest Path Problem

- Given a weighted graph and two vertices $u$ and $v$, we want to find a path of minimum total weight between $u$ and $v$.
  - Length of a path is the sum of the weights of its edges.
- Example:
  - Shortest path between Providence and Honolulu
- Applications
  - Internet packet routing
  - Flight reservations
  - Driving directions

Graphs
Shortest Path Problem

- If there is no path from \( v \) to \( u \), we denote the distance between them by \( d(v, u) = +\infty \).
- What if there is a negative-weight cycle in the graph?
Shortest Path Properties

Property 1:  
A subpath of a shortest path is itself a shortest path

Property 2:  
There is a tree of shortest paths from a start vertex to all the other vertices

Example:  
Tree of shortest paths from Providence
Dijkstra’s Algorithm

- The **distance** of a vertex \( v \) from a vertex \( s \) is the length of a shortest path between \( s \) and \( v \).

- Dijkstra’s algorithm computes the distances of all the vertices from a given start vertex \( s \) (**single-source** shortest paths).

- Assumptions:
  - the graph is connected
  - the edges are undirected
  - the edge weights are nonnegative

- We grow a “**cloud**” of vertices, beginning with \( s \) and eventually covering all the vertices.

- We store with each vertex \( v \) a label \( D[v] \) representing the distance of \( v \) from \( s \) in the subgraph consisting of the cloud and its adjacent vertices.

- The label \( D[v] \) is initialized to positive infinity.

- At each step:
  - We add to the cloud the vertex \( u \) outside the cloud with the smallest distance label, \( D[u] \).
  - We update the labels of the vertices adjacent to \( u \) (i.e. **edge relaxation**).
Edge Relaxation

Consider an edge \( e = (u, z) \) such that
- \( u \) is the vertex most recently added to the cloud
- \( z \) is not in the cloud

The relaxation of edge \( e \) updates distance \( D[z] \) as follows:
\[
D[z] \leftarrow \min\{D[z], D[u] + \text{weight}(e)\}
\]
1. Pull in one of the vertices with blue labels
2. The relaxation of edges updates the labels of LARGER font size
Example (cont.)
Exercise: Dijkstra’s alg

- Show how Dijkstra’s algorithm works on the following graph, assuming you start with SFO, i.e., $s = SFO$.
  - Show how the labels are updated in each iteration (a separate figure for each iteration).
Dijkstra’s Algorithm

- A priority queue stores the vertices outside the cloud
  - Key: distance
  - Element: vertex
- Locator-based methods
  - `insert(k, e)` returns a locator
  - `replaceKey(l, k)` changes the key of an item
- We store two labels with each vertex:
  - distance (`D[v]` label)
  - locator in priority queue

---

**Algorithm** \(\text{DijkstraDistances}(G, s)\)

1. \(Q \leftarrow \text{new heap-based priority queue}\)
2. \(\forall v \in G\text{.vertices()}:\)
   - if \(v = s\) \(\text{setDistance}(v, 0)\)
   - else \(\text{setDistance}(v, \infty)\)
3. \(l \leftarrow Q\text{.insert(getDistance}(v), v)\)
4. \(\text{setLocator}(v, l)\)
5. while \(\neg Q\text{.isEmpty()}\)
   - \{ pull a new vertex \(u\) into the cloud \}
   - \(u \leftarrow Q\text{.removeMin()}\)
   - \(\forall e \in G\text{.incidentEdges}(u)\)
     - \{ relax edge \(e\) \}
     - \(z \leftarrow G\text{.opposite}(u, e)\)
     - \(r \leftarrow \text{getDistance}(u) + \text{weight}(e)\)
     - if \(r < \text{getDistance}(z)\)
       - \(\text{setDistance}(z, r)\)
       - \(Q\text{.replaceKey(getLocator}(z), r)\)

\(\mathcal{O}(n)\) iter’s \(\mathcal{O}(\log n)\) iter’s
\(\mathcal{O}(n)\) iter’s
\(\sum_v \deg(u)\) iter’s
\(\mathcal{O}(n)\) iter’s
\(\mathcal{O}(\log n)\)
Analysis

- **Graph operations**
  - Method incidentEdges is called once for each vertex

- **Label operations**
  - We set/get the distance and locator labels of vertex \( z \) \( O(\text{deg}(z)) \) times
  - Setting/getting a label takes \( O(1) \) time

- **Priority queue operations**
  - Each vertex is inserted once into and removed once from the priority queue, where each insertion or removal takes \( O(\log n) \) time
  - The key of a vertex in the priority queue is modified at most \( \text{deg}(w) \) times, where each key change takes \( O(\log n) \) time

- Dijkstra’s algorithm runs in \( O((n + m) \log n) \) time provided the graph is represented by the adjacency list structure
  - Recall that \( \sum_v \text{deg}(v) = 2m \)

- The running time can also be expressed as \( O(m \log n) \) since the graph is connected
- The running time can be expressed as a function of \( n, O(n^2 \log n) \)
Exercise: Analysis

Algorithm $DijkstraDistances(G, s)$

$q \leftarrow \text{new unsorted-sequence-based priority queue}$

for all $v \in G.vertices()$

if $v = s$ setDistance($v, 0$)
else setDistance($v, \infty$)

$l \leftarrow Q.insert(\text{getDistance}(v), v)$

setLocator($v, l$)

while $\neg Q.isEmpty()$

{ pull a new vertex $u$ into the cloud }

$u \leftarrow Q.removeMin()$

for all $e \in G.incidentEdges(u)$ //use adjacency list

{ relax edge $e$ }

$z \leftarrow G.opposite(u, e)$

$r \leftarrow \text{getDistance}(u) + \text{weight}(e)$

if $r < \text{getDistance}(z)$

setDistance($z, r$)

$Q.replaceKey(\text{getLocator}(z), r)$ $O(1)$
Extension

Using the template method pattern, we can extend Dijkstra’s algorithm to return a tree of shortest paths from the start vertex to all other vertices.

We store with each vertex a third label:
- parent edge in the shortest path tree

In the edge relaxation step, we update the parent label.

Algorithm $DijkstraShortestPathsTree(G, s)$

```java
for all $v \in G.vertices()$
  \[setParent(v, \emptyset)\]
  \[
  \text{for all } e \in G.incidentEdges(u) \{ \text{relax edge } e \} \\
  z \leftarrow G.opposite(u, e) \\
  r \leftarrow getDistance(u) + weight(e) \\
  \text{if } r < getDistance(z) \\
  \text{setDistance}(z, r) \\
  \text{setParent}(z, e) \\
  Q.replaceKey(getLocator(z), r)
  \]
```
Why Dijkstra’s Algorithm Works

Dijkstra’s algorithm is based on the greedy method. It adds vertices by increasing distance.

- Suppose it didn’t find all shortest distances. Let F be the first wrong vertex the algorithm processed.
- When the previous node, D, on the true shortest path was considered, its distance was correct.
- But the edge (D,F) was relaxed at that time!
- Thus, so long as $D[F] > D[D]$, F’s distance cannot be wrong. That is, there is no wrong vertex.
Why It Doesn’t Work for Negative-Weight Edges

Dijkstra’s algorithm is based on the greedy method. It adds vertices by increasing distance.

- If a node with a negative incident edge were to be added late to the cloud, it could mess up distances for vertices already in the cloud.

C’s true distance is 1, but it is already in the cloud with $D[C]=2$!
Bellman-Ford Algorithm

- Works even with negative-weight edges
- Must assume directed edges (for otherwise we would have negative-weight cycles)
- Iteration $i$ finds all shortest paths that use $i$ edges.
- Running time: $O(nm)$.
- Can be extended to detect a negative-weight cycle if it exists
  - How?

Algorithm $\text{BellmanFord}(G, s)$

```
for all $v \in G.\text{vertices}()$
  if $v = s$
    $\text{setDistance}(v, 0)$
  else
    $\text{setDistance}(v, \infty)$
for $i \leftarrow 1$ to $n-1$ do
  for each $e \in G.\text{edges}()$
    \{ relax edge $e$ \}
    $u \leftarrow G.\text{origin}(e)$
    $z \leftarrow G.\text{opposite}(u,e)$
    $r \leftarrow \text{getDistance}(u) + \text{weight}(e)$
    if $r < \text{getDistance}(z)$
      $\text{setDistance}(z,r)$
```
Bellman-Ford Example

Nodes are labeled with their $d(v)$ values

First round

Second round

Third round

Graphs
Exercise: Bellman-Ford’s alg

- Show how Bellman-Ford’s algorithm works on the following graph, assuming you start with the top node.
  - Show how the labels are updated in each iteration (a separate figure for each iteration).
DAG-based Algorithm

- Works even with negative-weight edges
- Uses topological order
- Is much faster than Dijkstra’s algorithm
- Running time: $O(n+m)$.

Algorithm $\text{DagDistances}(G, s)$

for all $v \in G\text{.vertices()}$

if $v = s$

setDistance($v, 0$)

else

setDistance($v, \infty$)

Perform a topological sort of the vertices

for $u \leftarrow 1$ to $n$ do  
  {in topological order}

for each $e \in G\text{.outEdges}(u)$  
  {relax edge $e$}

$z \leftarrow G\text{.opposite}(u,e)$

$r \leftarrow \text{getDistance}(u) + \text{weight}(e)$

if $r < \text{getDistance}(z)$

setDistance($z,r$)
DAG Example

Nodes are labeled with their $d(v)$ values

Graphs (two steps)
Exercize: DAG-based Alg

- Show how DAG-based algorithm works on the following graph, assuming you start with the second rightmost node
  - Show how the labels are updated in each iteration (a separate figure for each iteration).
Summary of Shortest-Path Algs

- Breadth-First-Search
- Dijkstra’s algorithm (§12.6.1)
  - Algorithm
  - Edge relaxation
- The Bellman-Ford algorithm
- Shortest paths in DAGs
All-Pairs Shortest Paths

- Find the distance between every pair of vertices in a weighted directed graph G.
- We can make n calls to Dijkstra’s algorithm (if no negative edges), which takes $O(nm \log n)$ time.
- Likewise, n calls to Bellman-Ford would take $O(n^2m)$ time.
- We can achieve $O(n^3)$ time using dynamic programming (similar to the Floyd-Warshall algorithm).

**Algorithm** $AllPair(G)$ \{assumes vertices 1,\ldots,n\}

for all vertex pairs $(i,j)$
    if $i = j$
        $D_0[i,i] \leftarrow 0$
    else if $(i,j)$ is an edge in G
        $D_0[i,j] \leftarrow$ weight of edge $(i,j)$
    else
        $D_0[i,j] \leftarrow +\infty$

for $k \leftarrow 1$ to $n$ do
    for $i \leftarrow 1$ to $n$ do
        for $j \leftarrow 1$ to $n$ do
            $D_k[i,j] \leftarrow \min\{D_{k-1}[i,j], D_{k-1}[i,k]+D_{k-1}[k,j]\}$
        return $D_n$