Outline and Reading
- Weighted graphs (§7.1)
  - Shortest path problem
  - Shortest path properties
- Dijkstra’s algorithm (§7.1.1)
  - Algorithm
  - Edge relaxation
- The Bellman-Ford algorithm (§7.1.2)
- Shortest paths in dags (§7.1.3)
- All-pairs shortest paths (§7.2.1)

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

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 $x$ is the length of a shortest path between $x$ and $v$
- Dijkstra’s algorithm computes the distances of all the vertices from a given start vertex $x$
- Assumptions:
  - the graph is connected
  - the edges are undirected
  - the edge weights are nonnegative
- We grow a “cloud” of vertices, beginning with $x$ and eventually covering all the vertices
- We store with each vertex $v$ a label $d(v)$ representing the distance of $v$ from $x$ in the subgraph consisting of the cloud and its adjacent vertices
- 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 $u$
**Shortest Path**

**Analysis**

- **Graph operations**
  - Method `incidentEdges` is called once for each vertex.
- **Label operations**
  - We get the distance and locator labels of vertex \( z \) \( \Theta(\deg(z)) \) times.
  - Setting/getting a label takes \( \Theta(1) \) time.
- **Priority queue operations**
  - Each vertex is inserted once into and removed once from the priority queue, where each insertion or removal takes \( \Theta(\log m) \) time.
  - The key of a vertex in the priority queue is modified at most \( \deg(z) \) times, where each key change takes \( \Theta(\log m) \) time.
- **Dijkstra's algorithm**
  - Runs in \( \Theta(n + m \log n) \) time, provided the graph is represented by the adjacency list structure.
  - Recall that \( \sum \deg(v) = 2m \).
  - The running time can also be expressed as \( \Theta(m \log n) \) since the graph is connected.

**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:**
  - Distance \( (d(v)) \) label
  - Locator in priority queue
- In the edge relaxation step, we update the parent label

**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:
  \[
  d(z) \leftarrow \min\{d(z), d(u) + \text{weight}(e)\}
  \]

**Dijkstra's Algorithm**

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

**Example**

**Example (cont.)**

**Algorithm**

- **Dijkstra's Distance** \([G, s]\)
  - new heap-based priority queue
  - for all \( v \in G.vertices() \)
    - if \( v = s \)
      - `setDistance(v, 0)`
    - else
      - `setDistance(v, \infty)`
      - `setLocator(v, (s, e))`
  - while \(-Q.isEmpty()\)
    - \( u \leftarrow Q.removeMin() \)
    - for all \( e \in G.incidentEdges(u) \)
      - `relaxEdge(e)`
    - \( z \leftarrow G.opposite(u) \)
    - \( r \leftarrow \text{getDistance}(z) + \text{weight}(e) \)
    - if \( r < \text{getDistance}(z) \)
      - `replaceKey(getLocator(z), r)`

- **Dijkstra's Shortest Paths Tree** \([G, s]\)
  - for all \( v \in G.vertices() \)
    - `setParent(v, s)`
  - for all \( e \in G.incidentEdges(u) \)
    - `relaxEdge(e)`
    - \( z \leftarrow G.opposite(u) \)
    - \( r \leftarrow \text{getDistance}(z) + \text{weight}(e) \)
    - if \( r < \text{getDistance}(z) \)
      - `setDistance(z, r)`
      - `setParent(z, u)`
      - `replaceKey(getLocator(z), r)`
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.

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: \( O(nm) \).

```
Algorithm BellmanFord(G, s)
for all \( v \in G.\text{vertices} \)
if \( v = s \)
    setDistance(\( v, 0 \))
else
    setDistance(\( v, \infty \))
for \( i ← 1 \) 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 \( r < \text{getDistance}(z) \)
        setDistance(\( z, r \))
```

Bellman-Ford Example

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

DAG-based Algorithm

- Works even with negative-weight edges
- Uses topological order
- Doesn’t use any fancy data structures
- Is much faster than Dijkstra’s algorithm
- Running time: \( O(n+m) \).

```
Algorithm DAGDistance(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 \( u ← 1 \) 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
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,...,$n$]

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 $D_{n}$