Shortest Paths

- Weighted Digraphs
- Shortest paths

Diagram showing cities with distances between them.
Weighted Graphs

- **weights** on the edges of a graph represent distances, costs, etc.

- An example of an undirected weighted graph:
Shortest Path

- BFS finds paths with the minimum number of edges from the start vertex.
- Hence, BFS finds shortest paths assuming that each edge has the same weight.
- In many applications, e.g., transportation networks, the edges of a graph have different weights.
- How can we find paths of minimum total weight?
- Example - Boston to Los Angeles:
Dijkstra’s Algorithm

• Dijkstra’s algorithm finds shortest paths from a start vertex $s$ to all the other vertices in a graph with
  - undirected edges
  - nonnegative edge weights

• Dijkstra’s algorithm uses a greedy method (sometimes greed works and is good ...)

• the algorithm computes for each vertex $v$ the distance of $v$ from the start vertex $s$, that is, the weight of a shortest path between $s$ and $v$.

• the algorithm keeps track of the set of vertices for which the distance has been computed, called the cloud $C$

• the algorithm uses a label $D[v]$ to store an approximation of the distance between $s$ and $v$

• when a vertex $v$ is added to the cloud, its label $D[v]$ is equal to the actual distance between $s$ and $v$

• initially, the cloud $C$ contains $s$, and we set
  - $D[s] = 0$
  - $D[v] = \infty$ for $v \neq s$
Expanding the Cloud

- **meaning of D[z]**: length of shortest path from s to z that uses only intermediate vertices in the cloud

- after a new vertex u is added to the cloud, we need to check whether u is a better routing vertex to reach z

- let u be a vertex not in the cloud that has smallest label D[u]
  - we add u to the cloud C
  - we update the labels of the adjacent vertices of u as follows
    
    ```
    for each vertex z adjacent to u do
      if z is not in the cloud C then
        if D[u] + weight(u, z) < D[z] then
          D[z] = D[u] + weight(u, z)
    ```

- the above step is called a **relaxation** of edge (u, z)
Pseudocode

• we use a priority queue $Q$ to store the vertices not in the cloud, where $D[v]$ the key of a vertex $v$ in $Q$

Algorithm ShortestPath($G$, $v$):

Input: A weighted graph $G$ and a distinguished vertex $v$ of $G$.

Output: A label $D[u]$, for each vertex that $u$ of $G$, such that $D[u]$ is the length of a shortest path from $v$ to $u$ in $G$.

initialize $D[v] \leftarrow 0$ and $D[u] \leftarrow +\infty$ for each vertex $v \neq u$

let $Q$ be a priority queue that contains all of the vertices of $G$ using the $D$ lables as keys.

while $Q \neq \emptyset$ do

{pull $u$ into the cloud $C$}

$u \leftarrow Q$.removeMinElement()

for each vertex $z$ adjacent to $u$ such that $z$ is in $Q$ do

{perform the relaxation operation on edge $(u, z)$}

if $D[u] + w((u, z)) < D[z]$ then

$D[z] \leftarrow D[u] + w((u, z))$

change the key value of $z$ in $Q$ to $D[z]$

return the label $D[u]$ of each vertex $u$. 
Example

• shortest paths starting from BWI
• JFK is the nearest...
• followed by sunny PVD.
• BOS is just a little further.
• ORD: Chicago is my kind of town.
• MIA, just after Spring Break.
• DFW is huge like Texas.
• SFO: the 49’ers will take the prize next year.
• LAX is the last stop on the journey.
Running Time

• Let’s assume that we represent G with an adjacency list. We can then step through all the vertices adjacent to u in time proportional to their number (i.e. O(j) where j in the number of vertices adjacent to u)

• The priority queue Q:
  - A Heap: Implementing Q with a heap allows for efficient extraction of vertices with the smallest D label (O(logN)). If Q is implemented with locators, key updates can be performed in O(logN) time. The total run time is O((n+m)logn) where n is the number of vertices in G and m in the number of edges. In terms of n, worst case time is (On2log)
  - Unsorted Sequence: O(n) when we extract minimum elements, but fast key updates (O(1)). There are only n-1 extractions and m relaxations. The running time is O(n2+m)

• In terms of worst case time, heap is good for small data sets and sequence for larger.

• For each vertex, its neighbors are pulled into the cloud in random order. There are only O(logn) updates to the key of a vertex. Under this
Running Time (cont)

assumption, the run time of the head is $O(n \log n + m)$, which is always $O(n^2)$ the heap implementation is thus preferable for all but degenerate cases.
Java Implementation

• we use a priority queue \( Q \) supporting locator-based methods in the implementation of Dijkstra’s shortest path algorithm

• when we insert a vertex \( u \) into \( Q \), we associate with \( u \) the locator returned by \texttt{insert} (e.g., via a dictionary)

\[
\text{Locator } u\_\text{loc} = Q.\text{insert}((u\_\text{dist}), u); \\
\text{setLocator}(u, u\_\text{loc});
\]

• in the relaxation of an edge \((u,z)\), the update of the distance of \( z \) is performed with operation \texttt{replaceKey}

\[
\text{for } (\text{Enumeration } u\_\text{edges} = \text{graph.incidentEdges}(u); \\
\text{u\_edges.hasMoreElements()}; ) \{ \\
\text{Edge } e = (\text{Edge}) u\_\text{edges}.nextElement(); \\
\text{Vertex } z = \text{graph.opposite}(u,e); \\
\text{Locator } z\_\text{loc} = \text{getLocator}(z); \\
\text{if } (z\_\text{loc}.isContained()) \{ // test whether } z \text{ is in } Q \\
\text{int } e\_\text{weight} = \text{weight}(e); \\
\text{int } z\_\text{dist} = \text{value}(z\_\text{loc}); \\
\text{if } (u\_\text{dist} + e\_\text{weight} < z\_\text{dist}) \\
\text{Q.replaceKey}(z\_\text{loc}, \text{new Integer}(u\_\text{dist} e\_\text{weight})); \\
\}
\]
public abstract class Dijkstra {
    private static final int INFINITE = Integer.MAX_VALUE;
    protected InspectableGraph graph;
    // priority queue used by the algorithm
    protected PriorityQueue Q;
    public Object execute(InspectableGraph g, Vertex start) {
        graph = g;
        dijkstraVisit(start);
        return distances();
    }
    // initialization
    abstract void init();
    // create an empty priority queue
    abstract PriorityQueue initPQ(Comparator comp);
    // return the weight of edge e
    abstract int weight(Edge e);
    // attach to u its locator loc in Q
    abstract void setLocator(Vertex u, Locator loc);
    // return the locator attached to u
    abstract Locator getLocator(Vertex u);
Java Implementation (cont)

// attach to u its distance dist
abstract void setDistance(Vertex u, int dist);

// return the vertex distances in a data structure
abstract Object distances();

// return as an int the key of a vertex in Q
private int value(Locator u_loc) {
   return ((Integer) u_loc.key()).intValue();
}

Shortest Paths
protected void dijkstraVisit (Vertex v) {
    // initialize the priority queue Q and store all the vertices in it
    init();
    Q = initPQ(new IntegerComparator());
    for (Enumeration vertices = graph.vertices();
        vertices.hasMoreElements(); ) {
        Vertex u = (Vertex) vertices.nextElement();
        int u_dist;
        if (u==v)
            u_dist = 0;
        else
            u_dist = INFINITE;
        Locator u_loc  = Q.insert (new Integer(u_dist), u);
        setLocator (u, u_loc);
    }
    // grow the cloud, one vertex at a time
    while (! Q.isEmpty()) {
        // remove from Q and insert into cloud a vertex with minimum distance
        Locator u_loc = Q.min();
    }
Q.remove(u_loc);
setDistance(u, u_dist); // the distance of u is final
// examine all the neighbors of u and update their distances
for (Enumeration u_edges = graph.incidentEdges(u);
     u_edges.hasMoreElements(); ) {
    Edge e = (Edge) u_edges.nextElement();
    Vertex z = graph.opposite(u,e);
    Locator z_loc = getLocator(z);
    // check if z is not in the cloud, i.e., z is in Q
    if ( z_loc.isContained() ) {
        // relaxation of edge e = (u,z)
        int e_weight = weight(e);
        int z_dist = value(z_loc);
        if ( u_dist + e_weight < z_dist )
            Q.replaceKey(z_loc, new Integer(u_dist + e_weight));
    }
}
public class MyDijkstra extends Dijkstra {
    protected Hashtable locators = new Hashtable();
    protected Hashtable distances = new Hashtable();
    protected Hashtable weights = new Hashtable();
    public void init() {}
    public PriorityQueue initPQ(Comparator comp) {
        return (PriorityQueue) new SequenceLocPriorityQueue(comp);
    }
    public int weight(Edge e) {
        return ((Integer) weights.get(e)).intValue();
    }
    public void setWeight(Edge e, int w) {
        weights.put(e, new Integer(w));
    }
    public void setLocator(Vertex u, Locator loc) {
        locators.put(u, loc);
    }
    public Locator getLocator(Vertex u) {
        return (Locator) locators.get(u);
    }
}
Java Implementation (cont.)

```java
public void setDistance(Vertex u, int dist) {
    distances.put(u, new Integer(dist));
}
public int distance(Vertex u) {
    return ((Integer) distances.get(u)).intValue();
}
public Object distances() {
    return distances;
}
```