CMSC 132: Object-Oriented Programming II

Shortest Paths
Shortest Paths
Shortest paths

Given an edge-weighted digraph, find the shortest path from s to t.
Shortest path variants

- Which vertices?
  - Single source: from one vertex \( s \) to every other vertex.
  - Source-sink: from one vertex \( s \) to another \( t \).
  - All pairs: between all pairs of vertices.

- Restrictions on edge weights?
  - Nonnegative weights.
  - Arbitrary weights.

- Cycles?
  - No directed cycles.
  - No "negative cycles."

- Simplifying assumption: Shortest paths from \( s \) to each vertex \( v \) exist.
Weighted directed edge

public class DirectedEdge

    DirectedEdge(int v, int w, double weight)

int from()

int to()

double weight()

String toString()

weighted edge $v \rightarrow w$

vertex $v$

vertex $w$

weight of this edge

string representation

Idiom for processing an edge $e$: \texttt{int v = e.from()}, \texttt{w = e.to()};
Weighted directed edge implementation

```java
public class DirectedEdge{
    private final int v, w;
    private final double weight;

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

    public int from() { return v; }
    public int to() { return w; }
    public double weight() { return weight; }
}
```

![Directed Edge Diagram](image)
public class EdgeWeightedDigraph

EdgeWeightedDigraph(int V)

void addEdge(DirectedEdge e)

Iterable<DirectedEdge> adj(int v)

int V()

int E()

Iterable<DirectedEdge> edges()

String toString()

Conventions. Allow self-loops and parallel edges.
Edge-weighted digraph: adjacency-lists representation
Edge-weighted digraph implementation

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

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

    public void addEdge(DirectedEdge e) {
        int v = e.from();
        adj[v].add(e);
    }

    public Iterable<DirectedEdge> adj(int v) {
        return adj[v];
    }
}
```
Single-source shortest paths

What is the shortest distance and path from A to H?
Single-source shortest paths

- **Data structures:** Represent the **Shortest Path** with two vertex-indexed arrays:
  - `distTo[v]` is length of shortest path from `s` to `v`.
  - `edgeTo[v]` is last edge on shortest path from `s` to `v`.

```java
public double distTo(int v) {
    return distTo[v];
}
```

```java
public Iterable<DirectedEdge> pathTo(int v) {
    Stack<DirectedEdge> path = new Stack<DirectedEdge>();
    DirectedEdge e = edgeTo[v];
    while (e != null) {
        path.push(e);
        e = edgeTo[e.from()];
    }
    return path;
}
```
Edge relaxation

- Relax edge $e = v \rightarrow w$.
  - $\text{distTo}[v]$ is length of shortest known path from $s$ to $v$.
  - $\text{distTo}[w]$ is length of shortest known path from $s$ to $w$.
  - $\text{edgeTo}[w]$ is last edge on shortest known path from $s$ to $w$.
  - If $e = v \rightarrow w$ gives shorter path to $w$ through $v$, update both $\text{distTo}[w]$ and $\text{edgeTo}[w]$.
Edge relaxation

- Relax edge $e = v \to w$.
  - $\text{distTo}[v]$ is length of shortest known path from $s$ to $v$.
  - $\text{distTo}[w]$ is length of shortest known path from $s$ to $w$.
  - $\text{edgeTo}[w]$ is last edge on shortest known path from $s$ to $w$.
  - If $e = v \to w$ gives shorter path to $w$ through $v$, update both $\text{distTo}[w]$ and $\text{edgeTo}[w]$. 

private void relax(DirectedEdge e) {
    int v = e.from(), w = e.to();
    if (distTo[w] > distTo[v] + e.weight()) {
        distTo[w] = distTo[v] + e.weight();
        edgeTo[w] = e;
    }
}
Generic shortest-paths algorithm

**Generic algorithm (to compute SPT from s)**
- Initialize $\text{distTo}[s] = 0$ and $\text{distTo}[v] = \infty$ for all other vertices.
- Repeat until optimality conditions are satisfied:
  - Relax any edge.

**Efficient implementations:** How to choose which edge to relax?
- Dijkstra's algorithm (nonnegative weights).
- Topological sort algorithm (no directed cycles).
- Bellman-Ford algorithm (no negative cycles).
Dijkstra's algorithm

- Consider vertices in increasing order of distance from s (non-tree vertex with the lowest distTo[] value).
- Add vertex to tree and relax all edges pointing from that vertex.
Dijkstra's algorithm Demo

Pick vertex in List with minimum distance.
Update A’s neighbors

![Graph with labeled vertices and edges]

<table>
<thead>
<tr>
<th>V</th>
<th>distTo[]</th>
<th>edgeTo</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>--</td>
</tr>
<tr>
<td>B</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>C</td>
<td>∞</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>1</td>
<td>A</td>
</tr>
<tr>
<td>E</td>
<td>∞</td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>∞</td>
<td></td>
</tr>
</tbody>
</table>
Update D’s neighbors

```

V    distTo[]  edgeTo
A    0         --
B    2         A
C    3         D
D    1         A
E    3         D
F    9         D
G    5         D
```
Update B’s neighbors

No Update
Update E’s neighbors

No Update
Update C’s neighbors

```
<table>
<thead>
<tr>
<th>V</th>
<th>distTo[]</th>
<th>edgeTo</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>--</td>
</tr>
<tr>
<td>B</td>
<td>2</td>
<td>A</td>
</tr>
<tr>
<td>C</td>
<td>3</td>
<td>D</td>
</tr>
<tr>
<td>D</td>
<td>1</td>
<td>A</td>
</tr>
<tr>
<td>E</td>
<td>3</td>
<td>D</td>
</tr>
<tr>
<td>F</td>
<td>8</td>
<td>C</td>
</tr>
<tr>
<td>G</td>
<td>5</td>
<td>D</td>
</tr>
</tbody>
</table>
```
Update G’s neighbors
Update F’s neighbors

No Update
Dijkstra's algorithm Demo

- Consider vertices in increasing order of distance from s (non-tree vertex with the lowest distTo[] value).
- Add vertex to tree and relax all edges pointing from that vertex.
Dijkstra's algorithm

- Consider vertices in increasing order of distance from s (non-tree vertex with the lowest distTo[] value).
- Add vertex to tree and relax all edges pointing from that vertex.
Dijkstra's algorithm Implementation

```java
public class DijkstraSP{
    private DirectedEdge[] edgeTo;
    private double[] distTo;
    private IndexMinPQ<Double> pq;

    public DijkstraSP(EdgeWeightedDigraph G, int s) {
        edgeTo = new DirectedEdge[G.V()];
        distTo = new double[G.V()];
        pq = new IndexMinPQ<Double>(G.V());
        for (int v = 0; v < G.V(); v++)
            distTo[v] = Double.POSITIVE_INFINITY;
        distTo[s] = 0.0;
        pq.insert(s, 0.0);
        while (!pq.isEmpty()){
            int v = pq.delMin();
            for (DirectedEdge e : G.adj(v))
                relax(e);
        }
    }
}
```
Shortest Path Demo
Shortest Path Demo
Shortest Path Demo

If the graph has negative weighted edges, Dijkstra's algorithm does not work.
Acyclic shortest paths

- Consider vertices in topological order. Relax all edges pointing from that vertex.
Acyclic shortest paths

- Consider vertices in topological order.
- Relax all edges pointing from that vertex.
Longest paths in edge-weighted DAGs

- Formulate as a shortest paths problem in edge-weighted DAGs.
  - Negate all weights.
  - Find shortest paths.
  - Negate weights in result
- Key point. Topological sort algorithm works even with negative weights.
Longest paths in edge-weighted DAGs

- **Parallel job scheduling.**
  - Given a set of jobs with durations and precedence constraints, schedule the jobs (by finding a start time for each) so as to achieve the minimum completion time, while respecting the constraints.
Critical path method

- To solve a parallel job-scheduling problem, create edge-weighted DAG:
  - Source and sink vertices.
  - Two vertices (begin and end) for each job.
  - Three edges for each job.
    - Begin to end (weighted by duration)
    - Source to begin (0 weight)
    - End to sink (0 weight)
  - One edge for each precedence constraint (0 weight).
Critical path method

Use longest path from the source to schedule each job.
There are multiple shortest paths between vertices S and T. Which one will be reported by Dijkstra’s shortest path algorithm?

A. SDT
B. SBDT
C. SACDT
D. SACET
Quiz 1

There are multiple shortest paths between vertices S and T. Which one will be reported by Dijkstra’s shortest path algorithm?

A. SDT
B. SBDT
C. SACDT
D. SACET
In an unweighted, undirected connected graph, the shortest path from a node $S$ to every other node is computed most efficiently, in terms of time complexity by

A. Dijkstra’s algorithm starting from $S$.
B. Performing a DFS starting from $S$.
C. Performing a BFS starting from $S$.
D. None of the above
In an unweighted, undirected connected graph, the shortest path from a node S to every other node is computed most efficiently, in terms of time complexity by

A. Dijkstra’s algorithm starting from S.
B. Performing a DFS starting from S.
C. Performing a BFS starting from S.
D. None of the above