Shortest Paths

Adjacency Matrix or Adjacency List?

- **Adjacency Matrix**
  - Uses space $O(n^2)$
  - Can iterate over all edges in time $O(n^2)$
  - Can answer "Is there an edge from $u$ to $v$?" in $O(1)$ time
  - Better for *dense* graphs (lots of edges) ($m \sim n^2$)

- **Adjacency List**
  - Uses space $O(m+n)$
  - Can iterate over all edges in time $O(m+n)$
  - Can answer "Is there an edge from $u$ to $v$?" in $O(d(u))$ time
  - Better for *sparse* graphs (fewer edges) ($m \ll n^2$)

- $n$ = number of vertices
- $m$ = number of edges
- $d(u)$ = outdegree of $u$

Graph Representations

Relation in Java

- Like adjacency lists, but uses Sets instead of Lists
- *class* Digraph<N,E>*
- *Nodes<N>* and *Edges<E>*
  - $N$ is the type of data stored at the nodes (e.g. name)
  - $E$ is the type of data stored at the edges (e.g. distance)
- Each Edge knows its endpoints (two Nodes)
- Each Node knows the Set of its outgoing Edges and the Set of its incoming Edges
- This representation may look complicated, but it allows insertion and deletion of Nodes and Edges in constant time

Example

Example
Caveat: Deleting Nodes and Edges

• When you delete an Edge, you must remove it from the Edge sets of its endpoints.
• When you delete a Node, you must also delete all adjacent Edges.

Shortest Paths in Graphs

• Finding the shortest (min-cost) path in a graph is a problem that occurs often.
  – Find the shortest route between Ithaca and West Lafayette, IN.
  – Result depends on our notion of cost.
    • Least mileage
    • Least time
    • Cheapest
    • Least boring
  – All of these “costs” can be represented as edge weights.
• How do we find a shortest path?

Shortest Paths

• Let \( d(s,u) \) denote the distance (length of shortest path) from \( s \) to \( u \). In this example,
  • \( d(1,1) = 0 \) (distance from a node to itself is always 0)
  • \( d(1,2) = 1.6 \) (the shortest path goes through node 4)
  • \( d(1,3) = 2.5 \) (the shortest path goes through nodes 4 and 2)
  • \( d(1,4) = 1.5 \)

Dijkstra’s Algorithm

• Start with \( X = \{s\} \).
  – \( X \) is the set of nodes for which we have already determined the shortest path from \( s \).
• For each node \( u \not\in X \), initially set \( D(u) = w(s,u) \).
  – \( D(u) \) will be the shortest distance from \( s \) to \( u \) through only nodes in \( X \) (except for \( u \)).
  – \( D(2) = 2.4 \)
  – \( D(3) = \) incompletely visible
  – \( D(4) = 1.5 \)
Dijkstra's Algorithm

- Find \( u \notin X \) such that \( D(u) \) minimum, add it to \( X \)
  - at that point, we know \( d(s,u) = D(u) \)
- For each node \( v \notin X \) such that \( (u,v) \in E \), if \( D(u) + w(u,v) < D(v) \), set \( D(v) = D(u) + w(u,v) \)
  - i.e., check if going through \( u \) to get to \( v \) is better
  - \( D(2) = 2.4 \) \( D(3) = \infty \) \( D(4) = 1.5 \) = \( d(1,4) \)

• Find \( u \notin X \) such that \( D(u) \) minimum, add it to \( X \)
  - at that point, we know \( d(s,u) = D(u) \) \( u = 4 \)
• For each node \( v \notin X \) such that \( (u,v) \in E \), if \( D(u) + w(u,v) < D(v) \), set \( D(v) = D(u) + w(u,v) \)
  - i.e., check if going through \( u \) to get to \( v \) is better
  - \( D(2) = 1.6 \) \( D(3) = 4.6 \) \( D(4) = 1.5 \) = \( d(1,4) \)

• Find \( u \notin X \) such that \( D(u) \) minimum, add it to \( X \)
  - at that point, we know \( d(s,u) = D(u) \) \( u = 2 \)
• For each node \( v \notin X \) such that \( (u,v) \in E \), if \( D(u) + w(u,v) < D(v) \), set \( D(v) = D(u) + w(u,v) \)
  - i.e., check if going through \( u \) to get to \( v \) is better
  - \( D(2) = 1.6 \) = \( d(1,2) \) \( D(3) = 4.6 \) \( D(4) = 1.5 \) = \( d(1,4) \)
Dijkstra's Algorithm

• Find \( u \not\in X \) such that \( D(u) \) minimum, add it to \( X \)
  – at that point, we know \( d(s,u) = D(u) \)
• For each node \( v \not\in X \) such that \( (u,v) \in E \)
  if \( D(u) + w(u,v) < D(v) \), set \( D(v) = D(u) + w(u,v) \)
  – i.e., check if going through \( u \) to get to \( v \) is better
  – \( D(2) = 1.6 = d(1,2) \)
  – \( D(3) = 2.5 = d(1,3) \)
  – \( D(4) = 1.5 = d(1,4) \)

Proof of correctness – show by induction that the following are invariants of the loop:
• If \( u \in X \), then \( D(u) = d(s,u) \)
• If \( u \in X \) and \( v \not\in X \), then \( d(s,u) \leq d(s,v) \)
• For all \( u \), \( D(u) \) is the length of the shortest path from \( s \) to \( u \) such that all nodes on the path (except possibly \( u \)) are in \( X \)

Implementation:
• Use a priority queue for the nodes not yet taken – priority is \( D(u) \)

Complexity
• Every edge is examined once when its source is taken into \( X \)
• A vertex may be placed in the priority queue multiple times, but at most once for each incoming edge
• Number of insertions and deletions into priority queue = \( m + 1 \), where \( m = |E| \)
• Total complexity = \( O(m \log m) \)

Notes
• There are faster but more complicated algorithms for single-source, shortest-path problem that run in time \( O(n \log n + m) \) using something called Fibonacci heaps
• Dijkstra's algorithm does not work with negative weights, we need a more complicated algorithm called Warshall's algorithm

Dijkstra's Algorithm (pseudocode)

```plaintext
Dijkstra(s) {
  D[s] = 0;
  for (t ≠ s) \( D[t] = w(s,t) \):
    mark s;
    while (there exist unmarked nodes) {
      u = unmarked node with smallest \( D[u] \);
      mark u;
      for (each v adjacent to u) {
        \( D[v] = \min(D[v], D[u] + w(u,v)) \);
      }
    }
}```
Shortest Paths for Unweighted Graphs – A Special Case

- Use breadth-first search
- Time is $O(n + m)$ in adj list representation, $O(n^2)$ in adj matrix representation

![Graph Diagram]