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.

Example:
- In a flight route graph, the weight of an edge represents the distance in miles between the endpoint airports.

Shortest Paths 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 Paths 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 \( x \) a label \( d(x) \) representing the distance of \( x \) from \( x \) in the subgraph consisting of the cloud and its adjacent vertices.
- At each step:
  - We add to the cloud the vertex \( y \) outside the cloud with the smallest distance label, \( d(y) \).
  - We update the labels of the vertices adjacent to \( y \).

Edge Relaxation
- Consider an edge \( e = (u, z) \) such that \( u \) is the vertex most recently added to the cloud and \( z \) 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(v) returns the index of v
  - replaceKey(v, l) changes the key of an item
- We store two labels with each vertex:
  - Distance (d(v)) label
  - Index in priority queue

```
Algorithm DijkstraDistance(G, s)
for all v ∈ G.vertices()
    setDistance(v, ∞)
    setLocator(v, 0)
end
setDistance(s, 0)
setLocator(s, 0)
def relaxEdge(e)
    u = G.opposite(e, r)
    z = G.opposite(e, z)
    getDistance(z) + weight(e)
    if r < getDistance(u)
        setDistance(u, getDistance(z) + weight(e))
        setLocator(u, z)
        Q.replaceKey(u, getLocator(u, z))
end
```

Analysis

- Graph operations
  - Method incidences is called once for each vertex
- Label operations
  - We set/get the distance and locator labels of vertex f O(deg(v)) times
- 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 deg(v) times, where each key change takes O(log n) time.
- Dijkstra’s algorithm runs in O(m + n) log n) time.

Recall that ∑ deg(v) = 2m.
The running time can also be expressed as O(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:
  - parent edge in the shortest path tree
- In the edge relaxation step, we update the parent label.

```
Algorithm DijkstraShortestPathsTree(G, s)
for all v ∈ G.vertices()
    setParent(v, 0)
end
for all e ∈ G.incidentEdges(a)
    relaxEdge(e)
end
```

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.
Shortest Path problem. Given a directed graph $G = (V, E)$, with edge weights $c_{vw}$, find shortest path from node $s$ to node $t$.

Ex. Nodes represent agents in a financial setting and $c_{vw}$ is cost of transaction in which we buy from agent $v$ and sell immediately to $w$.

Failed Attempts

Dijkstra: Can fail if negative edge costs.

Re-weighting: Adding a constant to every edge weight can fail.

Negative Cost Cycles

Observation. If some path from $s$ to $t$ contains a negative cost cycle, there does not exist a shortest $s$-$t$ path; otherwise, there exists one that is simple.

Dynamic Programming

Subproblem Property: The problem can be recursively defined by the subproblem of the same kind.

Trade space for time: A table is used to store the solutions of the subproblems (the meaning of “programming” before the age of computers is “table”).

Designing a DP solution

How are the subproblems defined?
Where are the solutions stored?
How are the base values computed?
How do we compute each entry from other entries in the table?
What is the order in which we fill in the table?

Two DP algorithms for All-pairs shortest paths

Both are correct. Both produce correct values for all-pairs shortest paths.
The difference is the subproblem formulation, and hence in the running time.
The reason both algorithms are given is to remind you how to do DP algorithms!
But, be prepared to provide one or both of these algorithms, and to be able to apply it to an input (on some exam, for example).
Dynamic Programming

First attempt: let \(1, 2, \ldots, n\) denote the set of vertices.

Subproblem formulation:
\[ M[i, j, k] = \min \text{length of any path from } i \text{ to } j \text{ that uses at most } k \text{ edges.} \]

All paths have at most \(n-1\) edges, so \(1 \leq k \leq n-1\).

When \(k=1\), \(M[i, j, 1] = w[i, j]\), the edge weight from \(i\) to \(j\).

Minimum paths from \(i\) to \(j\) are found in \(M[i, j, n-1]\)

Question: How to set \(M[i, j, k]\) from other entries?

How to set \(M[i, j, k]\) from other entries, for \(k>1\)?

Consider a minimum weight path from \(i\) to \(j\) that has at most \(k\) edges.

- Case 1: The minimum weight path has at most \(k-1\) edges.
  \[ M[i, j, k] = M[i, j, k-1] \]

- Case 2: The minimum weight path has exactly \(k\) edges.
  \[ M[i, j, k] = \min \{ M[i, x, k-1] + w(x, j) : x \in V \} \]

Combining the two cases:
\[ M[i, j, k] = \min \{ \min \{ M[i, x, k-1] + w(x, j) : x \in V \}, M[i, j, k-1] \} \]

Finishing the design

Where is the answer stored?
How are the base values computed?
How do we compute each entry from other entries?
What is the order in which we fill in the matrix?
Running time?

Pseudo-Code and Running time analysis

for \(j = 1\) to \(n\)
for \(i = 1\) to \(n\)
\[ M[i, j, 1] = W[i, j] \]
for \(k = 2\) to \(n-1\)
for \(j = 1\) to \(n\)
for \(i = 1\) to \(n\)
\[ M[i, j, k] = \min \{ \min \{ M[i, x, k-1] + w(x, j) : x \in V \}, M[i, j, k-1] \} \]

How many entries do we need to compute? \(O(n^3)\)
How much time does it take to compute each entry? \(O(n)\)
Total time: \(O(n^4)\)

Next DP approach

Try a new subproblem formulation!
\[ Q[i, j, k] = \text{minimum weight of any path from } i \text{ to } j \text{ that uses internal vertices drawn from } \{1, 2, \ldots, k\}. \]

Designing a DP solution

How are the subproblems defined?
Where is the answer stored?
How are the base values computed?
How do we compute each entry from other entries?
What is the order in which we fill in the matrix?
Q[i,j,k] = minimum weight of any path from i to j that uses internal vertices (other than i and j) drawn from \{1,2,...,k\).

Base cases: Q[i,j,0] = w[i,j] for all i,j

Minimum paths from i to j are found in Q[i,j,n]

Once again, O(n^3) entries in the matrix

Solving subproblems

Q[i,j,k] = minimum weight of any path from i to j that uses internal vertices drawn from \{1,2,...,k\}.

P is a minimum cost path from i to j that uses vertex k, and has all internal vertices from \{1,2,...,k\}.

Path P1 from i to k, and P2 from k to j.

The weight of P1 is Q[i,k,k-1] (why??).

The weight of P2 is Q[k,j,k-1] (why??).

Thus the weight of P is Q[i,k,k-1] + Q[k,j,k-1].

New DP algorithm

for j = 1 to n
  for i = 1 to n
    Q[i,j,0] = w[i,j]
  for k = 1 to n
    for j = 1 to n
      for i = 1 to n
        Q[i,j,k] = min\{Q[i,j,k-1], Q[i,k,k-1] + Q[k,j,k-1]\}

Each entry only takes O(1) time to compute
There are O(n^3) entries
Hence, O(n^3) time.

Reusing the space

// Use R[i,j] for Q[i,j,0], Q[i,j,1], ..., Q[i,j,n].
for j = 1 to n
  for i = 1 to n
    R[i,j] = W[i,j];
  for k = 1 to n
    for j = 1 to n
      for i = 1 to n
        R[i,j] = min\{R[i,j], R[i,k] + R[k,j]\}

How to check negative cycles

// Use R[i,j] for Q[i,j,0], Q[i,j,1], ..., Q[i,j,n].
for j = 1 to n
  for i = 1 to n
    R[i,j] = W[i,j];
  for k = 1 to n
    for j = 1 to n
      for i = 1 to n
        R[i,j] = min\{R[i,j], R[i,k] + R[k,j]\};
    for i = 1 to n
      if (R[i,i] < 0) print("There is a negative cycle");
Detecting Negative Cycles: Application

Currency conversion. Given n currencies and exchange rates between pairs of currencies, is there an arbitrage opportunity?

Remark: Fastest algorithm very valuable!