Shortest Paths

Suppose $G(V, E)$ is a graph with weight function $w : E \to R$.

The *weight* of path $p = (v_0, \ldots, v_k)$ is

$$w(p) = w(v_0, v_1) + \cdots + w(v_{k-1}, v_k).$$

We'll be interested in paths of minimum weight from u to v

- usually talk about *shortest paths*, but they're only shortest in *unweighted graphs* (i.e., if all weights are 1)

BFS-SEARCH(s) gives shortest paths from s to any vertex v in unweighted graphs.

- We represent shortest paths implicitly using π, just as in BFS

Various Shortest Path Problems

Single-source shortest-path problem: Given s, find shortest path from s to every vertex $v \in V$

- Chapter 25: Dijkstra's algorithm, Bellman-Ford

Single-destination shortest-path problem: Given t, find shortest path from every vertex v to t

- Single-source shortest-path problem from s in G = single-destination shortest-path problem to s in G^T
- Single-destination = single-source in undirected graphs

Single-pair shortest-path problem: Find shortest path between s and t

- The best algorithms for this use the single-source shortest-path algorithm

All-pairs shortest-path problem: Find shortest path from s to t for all $s, t \in V$

- Could run single-source shortest-path algorithm for each s, but there are (sometimes) better ways
- Chapter 26

Negative-weight edges

We allow weights to be negative.

- E.g., the weight between u and v could be the gain/loss of taking the action that gets you from u to v

Shortest paths are not well defined in graphs with negative-weight cycles:

- The more times we go around the cycle, the "shorter" the path

We can take $\delta(s, v) = -\infty$ if there is a negative-weight cycle on a path from s to v.

Lemma: If there is an edge from $(u, v) \in E$, then

$$\delta(s, v) \leq \delta(s, u) + w(u, v).$$

Proof: One way of getting from s to v is to go from s to u and then from u to v.

- There may be better ways.

Relaxation

Both Dijkstra's algorithm and Bellman-Ford's algorithm use a technique called *relaxation*. Idea:

- Initialize $d[v]$ to ∞, $\pi[v]$ to NIL
- Test if we can improve $d[v]$ by going through u
  - This makes sense only if there is an edge (u, v)
  - The process of checking is called *relaxing* (u, v)

INITIALIZE-SINGLE-SOURCE(G, s)

```
1  for each vertex v ∈ V[G]
2      do d[v] ← ∞
3         π[v] ← NIL
4  d[s] ← 0
```

RELAX(u, v, w) [$(u, v) \in E[G]$, weight function w]

```
1  if d[v] > d[u] + w(u, v)
2     then d[v] ← d[u] + w(u, v)
3          π[v] ← u
```

Properties of Relaxation

Relaxation Property: If we start with INITIALIZE-SINGLE-SOURCE(G, s), then $d[v] \geq \delta(s, v)$, no matter how often we call RELAX. If $d[v]$ is ever $\delta(s, v)$, it never changes again.

Proof: This is true initially (since $d[v] = \infty$ unless $v = s$).

If the property was true before RELAX(u, v, w), it is true after:

- If $d[v] \leq d[u] + w(u, v)$, RELAX(u, v, w) doesn't change anything
- If $d[v] > d[u]+w(u, v)$ before, then after RELAX(u, v, w), $d[v] = d[u]+w(u, v) \geq \delta(s, u)+w(u, v) \geq \delta(s, v)$.

Since $d[v]$ never increases after a RELAX, if it hits $\delta(s, v)$, it does not change again.

The Relaxation Property holds even with negative weights.

Dijkstra's Algorithm

Dijkstra's algorithm solves single-source shortest-path problems if all weights are nonnegative.

Idea:

- Maintain list S of vertices whose shortest path has already been computed
  - We add elements to S in order of their distance from s
  - For $v \in S$, we have $d[v] = \delta(s, v)$
- Find vertex $v' \in V - S$ that has current minimum d value
  - Maintain elements of $V - S$ in a priority queue, keyed by d values
- Add v' to S
- Update all values of d for remaining elements of $V - S$

DIJKSTRA(G, w, s)

```
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  S ← ∅
3  Q ← V[G]
4  while Q ≠ ∅
5      do u ← EXTRACT-MIN(Q)
6         S ← S ∪ {u}
7         for each vertex v ∈ Adj[u]
8             do RELAX(u, v, w)
```

Example

Dijkstra's Algorithm: Analysis

In Dijkstra's algorithm, we do

- $|V|$ EXTRACT-MINs
- $\leq |E|$ RELAXes ($\leq 2|E|$ in the unordered case)

Thus, the running time depends on how we implement the priority queue.

- If we use an array
  - EXTRACT-MIN takes time $O(|V|)$
  - RELAX takes time $O(1)$
  - Total running time: $O(|V|^2 + |E|) = O(|V|^2)$
- If we use binary heap
  - EXTRACT-MIN takes time $O(\lg |V|)$
  - RELAX takes time $O(\lg |V|)$
    - Need to perform DECREASE-KEYs to update priority queue
  - Total running time: $O((|V| + |E|) \lg |V|)$
- If we use Fibonacci heaps (Chapter 21)
  - $|E|$ DECREASE-KEYs take time $O(|E|)$
  - Total running time: $O(|V| \lg |V| + |E|)$