Reminders

- Shortest Paths with Negative Edge Costs

Negative Edge Costs

- Allowing negative edge costs in Dijkstra’s algorithm can yield incorrect result. DO EXAMPLE
- Attempts to massage data (add large value to every edge) don’t help. DO EXAMPLE

Worse, a directed graph might have negative cost cycles, suggesting there is no shortest path!

Why might we care?

- Vertices could represent one’s position in the stock market, with edges representing possible trades, and edge weights a profit or loss.
- Vertices could represent state of a chemical system, with edges being reaction pathways and edge weights representing energy loss or gain.

Dynamic Programming To the Rescue

We’ll describe an algorithm, the Bellman-Ford Algorithm, that can compute shortest paths even with negative edge costs as long as there are no negative cost cycles.

Finding a good quantity to optimize takes a little work, but the idea is pretty reasonable once you see it.

We’ll let $opt(i, v)$ represent the optimal (minimum) cost of a path from $v$ to $t$ that uses at most $i$ edges.

- Clearly $opt(i, t) = 0$ and $opt(0, v) = -\infty$ if $v \neq t$.
- And clearly $opt(1, v) = c(v, t)$ if $(v, t) \in E$ and $-\infty$ otherwise.

Now let $P$ be a minimum-cost path from $v$ to $t$ using at most $i$ edges.

- If $P$ has length less than $i$, then $opt(i, v) = opt(i - 1, v)$.
- Otherwise, $P$ consists of some edge $(v, u)$ and a path of length $i - 1$ from $u$ to $t$, so

$$opt(i, v) = c(v, u) + opt(i - 1, u).$$

- Therefore,

$$opt(i, v) = \min_{(v, u) \in E} \{opt(i - 1, v), c(v, u) + opt(i - 1, u)\}$$

How big does $i$ need to get before we can be sure $opt(i, s)$ is the cost of a minimum-cost path from $s$ to $t$?

Well, since there are no negative cycles, there is a shortest path with no more than $n$ vertices, so $i \leq n - 1$.

**proof** A path of length greater than $n$ contains a cycle and the cycle has length at least 0. Removing it gives a path at least as short.
Complexity Analysis

Time Complexity

- The table is of size \( n^2 \).
- It appears as if it could take \( O(n^3) \) time to construct, since a vertex \( v \) of large degree may have \( O(n) \) neighbors.
- Let’s be more careful:
  - Computing \( \text{opt}[i,v] \) requires looking at each neighbor of \( v \), so it requires, \( \text{outDegree}(v) \) table accesses of \( \text{opt}[\cdot] \).
  - So, for each \( i \), filling in row \( i \) requires \( \sum_{v \in V \setminus \{i\}} \text{outDegree}(v) \) accesses.
  - This sum is at most \( m \), since each edge is used at most once.
  - Since there are \( n \) rows to the table, we get a more accurate \( O(mn) \) running time.

Space Complexity

- The \( i \)th row of \( \text{opt}[\cdot] \) depends only on the previous row, so we could use an array \( \text{opt}[v] \) which begins by holding row 1 and then uses those values (and a second temporary array \( \text{next} \)) to compute the next row from the current row.
- This gives \( O(n) \) space complexity beyond the storing of the graph.
- Add an array \( \text{next}(v) \) which holds the next vertex after \( v \) on the current shortest path from \( v \) to \( t \).
  - \( \text{next}(v) \) is initialized to \( \text{null} \) for all \( v \)
  - Whenever \( \text{opt}[i,v] \) changes, we change \( \text{next}[v] \) to hold the next vertex on the new (shorter) path from \( v \) to \( t \).
  - Let \( T \) be the graph containing all edges \((v, \text{next}(v))\). \( T \) is dynamically changing.
  - Claim: \( T \) is a tree. We show that \( T \) doesn’t contain cycles and that \( |V(T)| - 1 = |E(T)| \).
    * First, show \( |V(T)| - 1 = |E(T)| \).
    * Well, \( T \) begins by containing \( \{t\} \) and no edges, so the condition holds.
    * Consider a point at which \( \text{opt}[v] \) is being changed. Then \( \text{opt}[v] > c(v,u) + \text{opt}[u] \) for some neighbor \( u \) of \( v \). So \( u \) is in \( T \) (or else \( \text{opt}[u] = \infty \)). If \( v \) is in \( T \), then \( \text{next}[v] = w \neq \text{null} \) so we are just replacing \((v,w)\) in \( T \) with \((v,u)\), so condition still holds. If \( v \) is not in \( T \), then we are adding a new vertex and a new edge to \( T \) so condition still holds.
    * Second, show that \( T \) contains no cycles\(^1\).
      * If updating \( \text{opt}[v] \) creates a cycle in \( T \), then the cycle looks like \( v = v_0, v_1, \ldots, v_n = v \), where, for each \( i < n \), \( v_{i+1} = \text{next}[v_i] \). So, by definition of \( \text{next}[\cdot] \),
        \[
        \text{opt}[v_0] > c(v_0, v_1) + \text{opt}[v_1] \quad \text{and} \quad \text{opt}[v_i] = c(v_i, v_{i+1}) + \text{opt}[v_{i+1}], \forall i < n.
        \]
      * Thus
        \[
        \text{opt}[v_0] > \left( \sum_{i=0}^{n-1} c(v_i, v_{i+1}) \right) + \text{opt}[v_n] = \left( \sum_{i=0}^{n-1} c(v_i, v_{i+1}) \right) + \text{opt}[v_0]
        \]
        since \( v_0 = v_n \). But this is a negative weight cycle!

---

\(^1\)Instead, and more easily, one could show that \( T \) is connected. That fact along with \( |V(T)| - 1 = |E(T)| \) imply that the edges of \( T \) form a tree.