Computer Science & Engineering 423/823
Design and Analysis of Algorithms
Lecture 05 — Single-Source Shortest Paths (Chapter 24)

Stephen Scott
(Adapted from Vinodchandran N. Variyam)

sscott@cse.unl.edu
Given a weighted, directed graph $G = (V, E)$ with weight function $w : E \rightarrow \mathbb{R}$, the weight of path $p = \langle v_0, v_1, \ldots, v_k \rangle$ is the sum of the weights of its edges:

$$w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

Then the shortest-path weight from $u$ to $v$ is

$$\delta(u, v) = \begin{cases} \min \{w(p) : u \xrightarrow{p} v\} & \text{if there is a path from } u \text{ to } v \\ \infty & \text{otherwise} \end{cases}$$

A shortest path from $u$ to $v$ is any path $p$ with weight $w(p) = \delta(u, v)$.

Applications: Network routing, driving directions
Types of Shortest Path Problems

Given $G$ as described earlier,

- **Single-Source Shortest Paths**: Find shortest paths from source node $s$ to every other node
- **Single-Destination Shortest Paths**: Find shortest paths from every node to destination $t$
  - Can solve with SSSP solution. How?
- **Single-Pair Shortest Path**: Find shortest path from specific node $u$ to specific node $v$
  - Can solve via SSSP; no asymptotically faster algorithm known
- **All-Pairs Shortest Paths**: Find shortest paths between every pair of nodes
  - Can solve via repeated application of SSSP, but can do better
The shortest paths problem has the **optimal substructure property**: If $p = \langle v_0, v_1, \ldots, v_k \rangle$ is a SP from $v_0$ to $v_k$, then for $0 \leq i \leq j \leq k$, $p_{ij} = \langle v_i, v_{i+1}, \ldots, v_j \rangle$ is a SP from $v_i$ to $v_j$.

**Proof:** Let $p = v_0 \xrightarrow{p_{0i}} v_i \xrightarrow{p_{ij}} v_j \xrightarrow{p_{jk}} v_k$ with weight $w(p) = w(p_{0i}) + w(p_{ij}) + w(p_{jk})$. If there exists a path $p'_{ij}$ from $v_i$ to $v_j$ with $w(p'_{ij}) < w(p_{ij})$, then $p$ is not a SP since $v_0 \xrightarrow{p_{0i}} v_i \xrightarrow{p'_{ij}} v_j \xrightarrow{p_{jk}} v_k$ has less weight than $p$.

This property helps us to use a greedy algorithm for this problem.
What happens if the graph $G$ has edges with negative weights?

Dijkstra’s algorithm cannot handle this, Bellman-Ford can, under the right circumstances (which circumstances?)
Negative-Weight Edges (2)
Cycles

- What kinds of cycles might appear in a shortest path?
  - Negative-weight cycle
  - Zero-weight cycle
  - Positive-weight cycle
Relaxation

- Given weighted graph $G = (V, E)$ with source node $s \in V$ and other node $v \in V$ ($v \neq s$), we'll maintain $d[v]$, which is upper bound on $\delta(s, v)$.

- Relaxation of an edge $(u, v)$ is the process of testing whether we can decrease $d[v]$, yielding a tighter upper bound.
Initialize-Single-Source(\(G, s\))

1. for each vertex \(v \in V\) do
   2. \(d[v] = \infty\)
   3. \(\pi[v] = \text{NIL}\)
4. end
5. \(d[s] = 0\)

How is the invariant maintained?
Relax\((u, v, w)\)

1. if \(d[v] > d[u] + w(u, v)\) then
2. \(d[v] = d[u] + w(u, v)\)
3. \(\pi[v] = u\)

How do we know that we can tighten \(d[v]\) like this?
Relaxation Example

Numbers in nodes are values of $d$
Bellman-Ford Algorithm

- Greedy algorithm
- Works with negative-weight edges and detects if there is a negative-weight cycle
- Makes $|V| - 1$ passes over all edges, relaxing each edge during each pass
Bellman-Ford($G, w, s$)

1. **Initialize-Single-Source**($G, s$)

2. for $i = 1$ to $|V| - 1$ do

3. for each edge $(u, v) \in E$ do

4. \hspace{1em} Relax($u, v, w$)

5. end

6. end

7. for each edge $(u, v) \in E$ do

8. \hspace{1em} if $d[v] > d[u] + w(u, v)$ then

9. \hspace{2em} \textbf{return} FALSE // $G$ has a negative-wt cycle

10. end

11. end

12. \textbf{return} TRUE // $G$ has no neg-wt cycle reachable frm $s$
Bellman-Ford Algorithm Example (1)

Within each pass, edges relaxed in this order:
(t, x), (t, y), (t, z), (x, t), (y, x), (y, z), (z, x), (z, s), (s, t), (s, y)
Bellman-Ford Algorithm Example (2)

Within each pass, edges relaxed in this order:
(t, x), (t, y), (t, z), (x, t), (y, x), (y, z), (z, x), (z, s), (s, t), (s, y)
Time Complexity of Bellman-Ford Algorithm

- Initialize-Single-Source takes how much time?
- Relax takes how much time?
- What is time complexity of relaxation steps (nested loops)?
- What is time complexity of steps to check for negative-weight cycles?
- What is total time complexity?
Correctness of Bellman-Ford Algorithm

- Assume no negative-weight cycles
- Since no cycles appear in SPs, every SP has at most $|V| - 1$ edges
- Then define sets $S_0, S_1, \ldots S_{|V|-1}$:

$$S_k = \{v \in V : \exists s \xrightarrow{p} v \text{ s.t. } \delta(s, v) = w(p) \text{ and } |p| \leq k\}$$

- **Loop invariant:** After $i$th iteration of outer relaxation loop (Line 1), for all $v \in S_i$, we have $d[v] = \delta(s, v)$
  - Can prove via induction
- Implies that, after $|V| - 1$ iterations, $d[v] = \delta(s, v)$ for all $v \in V = S_{|V|-1}$
Correctness of Bellman-Ford Algorithm (2)

- Let \( c = \langle v_0, v_1, \ldots, v_k = v_0 \rangle \) be neg-weight cycle reachable from \( s \):
  \[
  \sum_{i=1}^{k} w(v_{i-1}, v_i) < 0
  \]

- If algorithm incorrectly returns `TRUE`, then (due to Line 8) for all nodes in the cycle \( (i = 1, 2, \ldots, k) \),
  \[
  d[v_i] \leq d[v_{i-1}] + w(v_{i-1}, v_i)
  \]

- By summing, we get
  \[
  \sum_{i=1}^{k} d[v_i] \leq \sum_{i=1}^{k} d[v_{i-1}] + \sum_{i=1}^{k} w(v_{i-1}, v_i)
  \]

- Since \( v_0 = v_k \), \( \sum_{i=1}^{k} d[v_i] = \sum_{i=1}^{k} d[v_{i-1}] \)

- This implies that \( 0 \leq \sum_{i=1}^{k} w(v_{i-1}, v_i) \), a contradiction
SSSPs in Directed Acyclic Graphs

- Why did Bellman-Ford have to run $|V| - 1$ iterations of edge relaxations?
- To confirm that SP information fully propagated to all nodes

- What if we knew that, after we relaxed an edge just once, we would be completely done with it?
- Can do this if $G$ a dag and we relax edges in correct order (what order?)
Dag-Shortest-Paths($G, w, s$)

1. topologically sort the vertices of $G$
2. Initialize-Single-Source($G, s$)
3. for each vertex $u \in V$, taken in topo sorted order do
   4. for each $v \in Adj[u]$ do
      5. Relax($u, v, w$)
   6. end
4. end
SSSP dag Example (1)

(a) Initial graph with source node r.

(b) After processing vertex s, the distance to s is updated.

(c) After processing vertex t, the distance to t is updated.

(d) After processing vertex y, the distance to y is updated.

The graph shows the progression of the shortest path algorithm from the source node r to other vertices in the graph.
SSSP dag Example (2)
Time Complexity of SSSP in dag

- Topological sort takes how much time?
- `initialize-single-source` takes how much time?
- How many calls to `relax`?
- What is total time complexity?
Dijkstra’s Algorithm

- Faster than Bellman-Ford
- Requires all edge weights to be nonnegative
- Maintains set $S$ of vertices whose final shortest path weights from $s$ have been determined
  - Repeatedly select $u \in V \setminus S$ with minimum SP estimate, add $u$ to $S$, and relax all edges leaving $u$
- Uses min-priority queue
Dijkstra($G, w, s$)

1. **Initialize-Single-Source**($G, s$)
2. $S = \emptyset$
3. $Q = V$
4. while $Q \neq \emptyset$ do
5.   $u = \text{Extract-Min}(Q)$
6.   $S = S \cup \{u\}$
7.   for each $v \in Adj[u]$ do
8.     $\text{Relax}(u, v, w)$
9.   end
10. end
Dijkstra’s Algorithm Example (1)
Dijkstra’s Algorithm Example (2)
Time Complexity of Dijkstra’s Algorithm

- Using array to implement priority queue,
  - `INITIALIZE-SINGLE-SOURCE` takes how much time?
  - What is time complexity to create $Q$?
  - How many calls to `EXTRACT-MIN`?
  - What is time complexity of `EXTRACT-MIN`?
  - How many calls to `RELAX`?
  - What is time complexity of `RELAX`?
  - What is total time complexity?

- Using heap to implement priority queue, what are the answers to the above questions?

- When might you choose one queue implementation over another?
Correctness of Dijkstra’s Algorithm

- **Invariant:** At the start of each iteration of the while loop, $d[v] = \delta(s, v)$ for all $v \in S$
  - Prove by contradiction (p. 660)
- Since all vertices eventually end up in $S$, get correctness of the algorithm
Linear Programming

- Given an \( m \times n \) matrix \( A \) and a size-\( m \) vector \( b \) and a size-\( n \) vector \( c \), find a vector \( x \) of \( n \) elements that maximizes \( \sum_{i=1}^{n} c_i x_i \) subject to \( Ax \leq b \)

- E.g. \( c = [ 2 \ -3 ] \), \( A = \begin{bmatrix} 1 & 1 \\ 1 & -2 \\ -1 & 0 \end{bmatrix} \), \( b = \begin{bmatrix} 22 \\ 4 \\ -8 \end{bmatrix} \) implies:

maximize \( 2x_1 - 3x_2 \) subject to

\[
\begin{align*}
x_1 + x_2 & \leq 22 \\
x_1 - 2x_2 & \leq 4 \\
x_1 & \geq 8
\end{align*}
\]

- Solution: \( x_1 = 16, \ x_2 = 6 \)
Difference Constraints and Feasibility

- **Decision version of this problem:** No objective function to maximize; simply want to know if there exists a **feasible solution**, i.e. an $x$ that satisfies $Ax \leq b$

- Special case is when each row of $A$ has exactly one 1 and one $-1$, resulting in a set of **difference constraints** of the form

  $$x_j - x_i \leq b_k$$

- **Applications:** Any application in which a certain amount of time must pass between events ($x$ variables represent times of events)
Difference Constraints and Feasibility (2)

\[ A = \begin{bmatrix}
1 & -1 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & -1 \\
0 & 1 & 0 & 0 & -1 \\
-1 & 0 & 1 & 0 & 0 \\
-1 & 0 & 0 & 1 & 0 \\
0 & 0 & -1 & 1 & 0 \\
0 & 0 & -1 & 0 & 1 \\
0 & 0 & 0 & -1 & 1 \\
\end{bmatrix} \quad \text{and} \quad b = \begin{bmatrix}
0 \\
-1 \\
1 \\
5 \\
4 \\
-1 \\
-3 \\
-3 \\
\end{bmatrix} \]
Difference Constraints and Feasibility (3)

Is there a setting for $x_1, \ldots, x_5$ satisfying:

\[
\begin{align*}
  x_1 - x_2 & \leq 0 \\
  x_1 - x_5 & \leq -1 \\
  x_2 - x_5 & \leq 1 \\
  x_3 - x_1 & \leq 5 \\
  x_4 - x_1 & \leq 4 \\
  x_4 - x_3 & \leq -1 \\
  x_5 - x_3 & \leq -3 \\
  x_5 - x_4 & \leq -3
\end{align*}
\]

One solution: $x = (-5, -3, 0, -1, -4)$
Constraint Graphs

- Can represent instances of this problem in a **constraint graph** $G = (V, E)$
- Define a vertex for each variable, plus one more: If variables are $x_1, \ldots, x_n$, get $V = \{v_0, v_1, \ldots, v_n\}$
- Add a directed edge for each constraint, plus an edge from $v_0$ to each other vertex:

$$E = \{(v_i, v_j) : x_j - x_i \leq b_k \text{ is a constraint}\}$$

$$\cup \{(v_0, v_1), (v_0, v_2), \ldots, (v_0, v_n)\}$$

- Weight of edge $(v_i, v_j)$ is $b_k$, weight of $(v_0, v_\ell)$ is 0 for all $\ell \neq 0$
Constraint Graph Example
Theorem: Let $G$ be the constraint graph for a system of difference constraints. If $G$ has a negative-weight cycle, then there is no feasible solution to the system. If $G$ has no negative-weight cycle, then a feasible solution is

$$x = [\delta(v_0, v_1), \delta(v_0, v_2), \ldots, \delta(v_0, v_n)]$$

For any edge $(v_i, v_j) \in E$, \(\delta(v_0, v_j) \leq \delta(v_0, v_i) + w(v_i, v_j) \Rightarrow \delta(v_0, v_j) - \delta(v_0, v_i) \leq w(v_i, v_j)\)

If there is a negative-weight cycle $c = \langle v_i, v_{i+1}, \ldots, v_k \rangle$, then there is a system of inequalities $x_{i+1} - x_i \leq w(v_i, v_{i+1})$, $x_{i+2} - x_{i+1} \leq w(v_{i+1}, v_{i+2})$, \ldots, $x_k - x_{k-1} \leq w(v_{k-1}, v_k)$. Summing both sides gives $0 \leq w(c) < 0$, implying that a negative-weight cycle indicates no solution.

Can solve this with Bellman-Ford in time $O(n^2 + nm)$.