Lecture 14: Single-Source Shortest Paths

Michael Dinitz

October 14, 2021
601.433/633 Introduction to Algorithms
Introduction

Setup:

- Directed graph $G = (V, E)$
- Length $\ell(x, y)$ on each edge $(x, y) \in E$ (equivalent: $\ell : E \to \mathbb{R}$)
- Length of path $P$ is $\ell(P) = \sum_{(x, y) \in P} \ell(x, y)$
- $d(x, y) = \min_{x \to y \text{ paths}} \ell(P)$
Introduction

Setup:

- Directed graph $G = (V, E)$
- Length $\ell(x, y)$ on each edge $(x, y) \in E$ (equivalent: $\ell : E \rightarrow \mathbb{R}$)
- Length of path $P$ is $\ell(P) = \sum_{(x, y) \in P} \ell(x, y)$
- $d(x, y) = \min_{x \rightarrow y \text{ paths } P} \ell(P)$

Today: source $v \in V$, want to compute shortest path from $v$ to every $u \in V$

- $d(u) = d(v, u)$ for all $u \in V$
- Representation: “shortest path tree” out of $v$.
- Often only care about distances – can reconstruct tree from distances.
Bellman-Ford
Dynamic Programming Approach

Subproblems:
- $\text{OPT}(u, i)$: shortest path from $v$ to $u$ that uses at most $i$ hops (edges)
- If no such path, set to “infinitely long” fake path.
- For simplicity, create loop (edge to and from the same node) at every node, length 0
Dynamic Programming Approach

Subproblems:
- \( \text{OPT}(u, i) \): shortest path from \( v \) to \( u \) that uses at most \( i \) hops (edges)
- If no such path, set to “infinitely long” fake path.
- For simplicity, create loop (edge to and from the same node) at every node, length 0

Theorem (Optimal Substructure)

\[
\ell(\text{OPT}(u, k)) = \begin{cases} 
0 & \text{if } u = v, k = 0 \\
\infty & \text{if } u \neq v, k = 0 \\
\text{otherwise} & \end{cases}
\]
Dynamic Programming Approach

Subproblems:

- $\text{OPT}(u, i)$: shortest path from $v$ to $u$ that uses at most $i$ hops (edges)
- If no such path, set to “infinitely long” fake path.
- For simplicity, create loop (edge to and from the same node) at every node, length 0

**Theorem (Optimal Substructure)**

\[
\ell(\text{OPT}(u, k)) = \begin{cases} 
0 & \text{if } u = v, k = 0 \\
\infty & \text{if } u \neq v, k = 0 \\
\min_{w: (w, u) \in E} (\ell(\text{OPT}(w, k - 1)) + \ell(w, u)) & \text{otherwise}
\end{cases}
\]
Proof of Optimal Substructure

\( k = 0 \): ✓. So let \( k \geq 1 \).
Proof of Optimal Substructure

\[ k = 0 : \checkmark. \] So let \( k \geq 1. \)

\[ \leq: \] Let \( x = \arg \min_{w: (w, u) \in E} \left( \ell(OPT(w, k - 1)) + \ell(w, u) \right) \)

\[ \implies OPT(x, k - 1) \circ (x, u) \text{ is a } v \to u \text{ path with at most } k \text{ edges, length } \ell(OPT(x, k - 1)) + \ell(x, u) \]

\[ \implies OPT(u, k) \leq \min_{w: (w, u) \in E} \left( \ell(OPT(w, k - 1)) + \ell(w, u) \right) \]
Proof of Optimal Substructure

\( k = 0 : \sqrt{\text{. So let } k \geq 1.} \)

\( \leq: \) Let \( x = \arg\min_{w:(w,u) \in E} (\ell(\text{OPT}(w, k - 1)) + \ell(w, u)) \)

\( \implies \) \( \text{OPT}(x, k - 1) \circ (x, u) \) is a \( v \to u \) path with at most \( k \) edges, length \( \ell(\text{OPT}(x, k - 1)) + \ell(x, u)) \)

\( \implies \) \( \text{OPT}(u, k) \leq \min_{w:(w,u) \in E} (\ell(\text{OPT}(w, k - 1)) + \ell(w, u)) \)

\( \geq: \) Let \( z \) be node before \( u \) in \( \text{OPT}(u, k) \), and let \( P' \) be the first \( k - 1 \) edges of \( \text{OPT}(u, k) \). Then

\[
\text{OPT}(u, k) = \ell(P') + \ell(z, u) \geq \ell(\text{OPT}(z, k - 1)) + \ell(z, u)
\]

\[
\geq \min_{w:(w,u) \in E} (\ell(\text{OPT}(w, k - 1)) + \ell(w, u))
\]
Bellman-Ford Algorithm

Obvious dynamic program!

\[
M[u, 0] = \infty \text{ for all } u \in V, u \neq v \\
M[v, 0] = 0
\]

for (k = 1 to n - 1) {
    for (u ∈ V) {
        \[M[u, k] = \min_{w:(w,u) \in E}(M[w, k - 1] + \ell(w, u))\]
    }
}
Bellman-Ford Algorithm

Obvious dynamic program!

\[
\begin{align*}
M[u, 0] &= \infty \quad \text{for all } u \in V, u \neq v \\
M[v, 0] &= 0 \\
\end{align*}
\]

\[
\text{for}(k = 1 \text{ to } n - 1) \quad \{
\begin{align*}
\text{for}(u \in V) \quad \{
M[u, k] &= \min_{w: (w, u) \in E} (M[w, k - 1] + \ell(w, u)) \\
\}
\}
\]

Running Time:

Obvious: \(O(n^3)\)

Smarter: \(O(mn)\)
Bellman-Ford Algorithm

Obvious dynamic program!

\[
\begin{align*}
M[u, 0] &= \infty \text{ for all } u \in V, u \neq v \\
M[v, 0] &= 0
\end{align*}
\]

\[
\text{for}(k = 1 \text{ to } n - 1) \{ \\
\text{for}(u \in V) \{ \\
M[u, k] &= \min_{w: (w, u) \in E}(M[w, k - 1] + \ell(w, u)) \\
\}
\}
\]

Running Time:
- Obvious: \(O(n^3)\)
Bellman-Ford Algorithm

Obvious dynamic program!

\[
\begin{align*}
M[u, 0] &= \infty \text{ for all } u \in V, u \neq v \\
M[v, 0] &= 0 \\
\end{align*}
\]

\[
\text{for}(k = 1 \text{ to } n - 1) \{ \\
\quad \text{for}(u \in V) \{ \\
\quad\quad M[u, k] = \min_{w: (w, u) \in E} (M[w, k - 1] + \ell(w, u)) \\
\quad \}\}
\]

Running Time:
- Obvious: \(O(n^3)\)
- Smarter: \(O(mn)\)
## Theorem

*After algorithm completes, $M[u, k] = \ell(OPT(u, k))$ for all $k \leq n - 1$ and $u \in V$.***
Bellman-Ford: Correctness

Theorem

After algorithm completes, $M[u, k] = \ell(\text{OPT}(u, k))$ for all $k \leq n - 1$ and $u \in V$.

Proof.

Induction on $k$. Obviously true for $k = 0$. 
Bellman-Ford: Correctness

Theorem

After algorithm completes, \( M[u, k] = \ell(\text{OPT}(u, k)) \) for all \( k \leq n - 1 \) and \( u \in V \).

Proof.

Induction on \( k \). Obviously true for \( k = 0 \).

\[
M[u, k] = \min_{w: (w, u) \in E} (M[w, k-1]) + \ell(w, u))
= \min_{w: (w, u) \in E} (\ell(\text{OPT}(w, k-1)) + \ell(w, u))
= \ell(\text{OPT}(u, k))
\]

(algorithm)

(induction)

[optimal substructure]
Negative Weights and Cycle

Suppose weights are negative. Does the problem make sense?
Suppose weights are negative. Does the problem make sense?

- Negative-weight cycle: not really!
Negative Weights and Cycle

Suppose weights are negative. Does the problem make sense?

- Negative-weight cycle: not really! Go around cycle forever, make distances arbitrarily negative
Suppose weights are negative. Does the problem make sense?

- Negative-weight cycle: not really! Go around cycle forever, make distances arbitrarily negative
- No negative-weight cycle: everything we did before is fine!
Negative Weights and Cycle

Suppose weights are negative. Does the problem make sense?

- Negative-weight cycle: not really! Go around cycle forever, make distances arbitrarily negative
- No negative-weight cycle: everything we did before is fine!

Detecting negative-weight cycle:
Negative Weights and Cycle

Suppose weights are negative. Does the problem make sense?

- Negative-weight cycle: not really! Go around cycle forever, make distances arbitrarily negative
- No negative-weight cycle: everything we did before is fine!

Detecting negative-weight cycle: One more round of Bellman-Ford!
Relaxations

Common primitive in shortest path algorithms
  - Reinterpret Bellman-Ford via relaxations
  - Use relaxations for Dijkstra’s algorithm

\[
\hat{d}(u) : \text{upper bound on } d(u)
\]

Initially:
\[
\hat{d}(v) = 0, \quad \hat{d}(u) = \infty \quad \text{for all } u \neq v
\]

Intuition for \( \text{relax}(x, y) \): can we improve \( \hat{d}(y) \) by going through \( x \)?

\[
\text{relax}(x, y) = \\
\begin{cases} 
\text{if } \hat{d}(y) > \hat{d}(x) + \ell(x, y) \\
\quad \hat{d}(y) = \hat{d}(x) + \ell(x, y) \\
\quad y.\text{parent} = x
\end{cases}
\]
Relaxations

Common primitive in shortest path algorithms
- Reinterpret Bellman-Ford via relaxations
- Use relaxations for Dijkstra’s algorithm

\( \hat{d}(u) \): upper bound on \( d(u) \)
- Initially: \( \hat{d}(v) = 0, \hat{d}(u) = \infty \) for all \( u \neq v \)
Relaxations

Common primitive in shortest path algorithms
  - Reinterpret Bellman-Ford via relaxations
  - Use relaxations for Dijkstra’s algorithm

\( \hat{d}(u) \): upper bound on \( d(u) \)
  - Initially: \( \hat{d}(v) = 0 \), \( \hat{d}(u) = \infty \) for all \( u \neq v \)

Intuition for \( \text{relax}(x, y) \): can we improve \( \hat{d}(y) \) by going through \( x \)?
Relaxations

Common primitive in shortest path algorithms

- Reinterpret Bellman-Ford via relaxations
- Use relaxations for Dijkstra’s algorithm

\( \hat{d}(u) \): upper bound on \( d(u) \)
- Initially: \( \hat{d}(v) = 0, \hat{d}(u) = \infty \) for all \( u \neq v \)

Intuition for \( \text{relax}(x, y) \): can we improve \( \hat{d}(y) \) by going through \( x \)?

```
relax(x, y) {
    if(\( \hat{d}(y) > \hat{d}(x) + \ell(x, y) \)) {
        \( \hat{d}(y) = \hat{d}(x) + \ell(x, y) \)
        y.parent = x
    }
}
```
Bellman-Ford as Relaxations

\[
\text{for}(i = 1 \text{ to } n) \{ \\
\quad \text{foreach}(u \in V) \{ \\
\quad \quad \text{foreach(edge } (x, u) \{ \\
\quad \quad \quad \text{relax}(x, u) \\
\quad \quad \} \\
\quad } \\
\} \\
\} \\
\]
Bellman-Ford as Relaxations

\[
\text{for(i = 1 to n) \{ }
\text{ \hspace{1cm} foreach(u ∈ V) \{ }
\text{ \hspace{2cm} foreach(edge (x, u)) \{ }
\text{ \hspace{3cm} relax(x, u) }
\text{ \hspace{2cm} \}}
\text{ \hspace{1cm} \}}
\text{ \}}
\\]

Not precisely the same: freezing/parallelism
Dijkstra's Algorithm
High Level

Intuition: “greedy starting at $v$”
  - BFS but with edge lengths: use priority queue (heap) instead of queue!

Pros: faster than Bellman-Ford (super fast with appropriate data structures)

Cons: Doesn’t work with negative edge weights.
Dijkstra’s Algorithm

\[ T = \emptyset \]
\[ \hat{d}(v) = 0 \]
\[ \hat{d}(u) = \infty \text{ for all } u \neq v \]

while(not all nodes in \( T \)) {
    let \( u \) be node not in \( T \) with minimum \( \hat{d}(u) \)
    Add \( u \) to \( T \)
    foreach edge \((u, x)\) with \( x \notin T \) {
        relax\((u, x)\)
    }
}
Dijkstra Example

Figure 24.6: The execution of Dijkstra's algorithm. The source $s$ is the leftmost vertex. The shortest-path estimates appear within the vertices, and shaded edges indicate predecessor values.

(a) The situation just before the first iteration of the while loop of lines 4–8. The shaded vertex has the minimum $d$ value and is chosen as vertex $u$ in line 5.

(b)–(f) The situation after each successive iteration of the while loop. The shaded vertex in each part is chosen as vertex $u$ in line 5 of the next iteration.

The $d$ values and predecessors shown in part (f) are the final values.

Theorem 24.6 (Correctness of Dijkstra's algorithm)

Dijkstra's algorithm, run on a weighted, directed graph $G = \langle V, E \rangle$ with non-negative weight function $w$ and source $s$, terminates with $u : d_D(s, u)$ for all vertices $u \in V$.

Because Dijkstra's algorithm always chooses the "lightest" or "closest" vertex in $V \setminus S$ to add to set $S$, we use this strategy. Chapter 16 explains greedy strategies in detail, but you need not have read that chapter to understand Dijkstra's algorithm. Greedy strategies do not always yield optimal results in general, but as the following theorem and its corollary show, Dijkstra's algorithm does indeed compute shortest paths. The key is to show that each time it adds a vertex $u$ to set $S$, we have $u : d_D(s, u)$.
Dijkstra Correctness

**Theorem**

*Throughout the algorithm:*

1. $T$ is a shortest-path tree from $v$ to the nodes in $T$, and
2. $\hat{d}(u) = d(u)$ for every $u \in T$. 

Proof. Induction on $T$ (iterations of algorithm)

**Base Case:** After first iteration, $v$ is added to $T$ with $\hat{d}(v) = d(v) = 0$.
Dijkstra Correctness

**Theorem**

Throughout the algorithm:

1. $T$ is a shortest-path tree from $v$ to the nodes in $T$, and
2. $\hat{d}(u) = d(u)$ for every $u \in T$.

**Proof.** Induction on $|T|$ (iterations of algorithm)
Dijkstra Correctness

Theorem

Throughout the algorithm:

1. $T$ is a shortest-path tree from $v$ to the nodes in $T$, and
2. $\hat{d}(u) = d(u)$ for every $u \in T$.

Proof. Induction on $|T|$ (iterations of algorithm)

Base Case: After first iteration (when $|T| = 1$), added $v$ to $T$ with $\hat{d}(v) = d(v) = 0$ √
Correctness: Inductive Step (Sketch)

Consider iteration when \( u \) added to \( T \), let \( w = u\.parent \)

\[ \hat{d}(u) = \hat{d}(w) + \ell(w, u) = d(w) + \ell(w, u) \quad \text{(induction)} \]
Correctness: Inductive Step (Sketch)

Consider iteration when \( u \) added to \( T \), let \( w = u.parent \)

\[ \hat{d}(u) = \hat{d}(w) + \ell(w, u) = d(w) + \ell(w, u) \text{ (induction)} \]

- Red path \( P \) actual shortest path, black path found by Dijkstra
- \( w' \) predecessor of \( u \) on \( P \). Can’t be in \( T \).
  - If it was, would have \( \hat{d}(w') = d(w') \) by induction, would have relaxed \( (w', u) \), so would have \( w' = u.parent \)
- \( x \) first node of \( P \) outside \( T \), previous node \( y \)
Correctness: Inductive Step (Sketch)

Consider iteration when $u$ added to $T$, let $w = u\.\text{parent}$

$\implies \hat{d}(u) = \hat{d}(w) + \ell(w, u) = d(w) + \ell(w, u)$ (induction)

- Red path $P$ actual shortest path, black path found by Dijkstra
- $w'$ predecessor of $u$ on $P$. Can’t be in $T$.
  - If it was, would have $\hat{d}(w') = d(w')$ by induction, would have relaxed $(w', u)$, so would have $w' = u\.\text{parent}$
- $x$ first node of $P$ outside $T$, previous node $y$

$\hat{d}(x) \leq \hat{d}(y) + \ell(y, x) = d(y) + \ell(y, x) < \ell(P) = d(u) \leq \hat{d}(u)$
Correctness: Inductive Step (Sketch)

Consider iteration when $u$ added to $T$, let $w = u.parent$

\[ \hat{d}(u) = \hat{d}(w) + \ell(w, u) = d(w) + \ell(w, u) \]  
(induction)

- Red path $P$ actual shortest path, black path found by Dijkstra
- $w'$ predecessor of $u$ on $P$. Can't be in $T$.
  - If it was, would have $\hat{d}(w') = d(w')$ by induction, would have relaxed $(w', u)$, so would have $w' = u.parent$
- $x$ first node of $P$ outside $T$, previous node $y$

\[ \hat{d}(x) \leq \hat{d}(y) + \ell(y, x) = d(y) + \ell(y, x) < \ell(P) = d(u) \leq \hat{d}(u) \]

Contradiction! Algorithm would have chosen $x$ next, not $u$. 
Running Time

Algorithm needs to:

- Select node with minimum $\hat{d}$ value $n$ times
- Decrease a $\hat{d}$ value at most once per relaxation $\implies \leq m$ times.
Running Time

Algorithm needs to:
- Select node with minimum $\hat{d}$ value $n$ times
- Decrease a $\hat{d}$ value at most once per relaxation $\implies \leq m$ times.

Nothing fancy, keep $\hat{d}(u)$ in adjacency list: selecting min $\hat{d}$ value takes $O(n)$ time $\implies O(n^2 + m) = O(n^2)$ total.
Running Time

Algorithm needs to:
- Select node with minimum $\hat{d}$ value $n$ times
- Decrease a $\hat{d}$ value at most once per relaxation $\implies \leq m$ times.

Nothing fancy, keep $\hat{d}(u)$ in adjacency list: selecting min $\hat{d}$ value takes $O(n)$ time $\implies O(n^2 + m) = O(n^2)$ total.

Keep $\hat{d}$ values in a heap!
- Insert $n$ times
- Extract-Min $n$ times
- Decrease-Key $m$ times
Running Time

Algorithm needs to:

- Select node with minimum \(\hat{d}\) value \(n\) times
- Decrease a \(\hat{d}\) value at most once per relaxation \(\Rightarrow \leq m\) times.

Nothing fancy, keep \(\hat{d}(u)\) in adjacency list: selecting min \(\hat{d}\) value takes \(O(n)\) time \(\Rightarrow O(n^2 + m) = O(n^2)\) total.

Keep \(\hat{d}\) values in a heap!

- Insert \(n\) times
- Extract-Min \(n\) times
- Decrease-Key \(m\) times

Binary heap: \(O(\log n)\) per operation (amortized)
\(\Rightarrow O((m + n) \log n)\) running time.
Running Time

Algorithm needs to:
- Select node with minimum $\hat{d}$ value $n$ times
- Decrease a $\hat{d}$ value at most once per relaxation $\Longrightarrow \leq m$ times.

Nothing fancy, keep $\hat{d}(u)$ in adjacency list: selecting min $\hat{d}$ value takes $O(n)$ time $\Longrightarrow O(n^2 + m) = O(n^2)$ total.

Keep $\hat{d}$ values in a heap!
- Insert $n$ times
- Extract-Min $n$ times
- Decrease-Key $m$ times

Binary heap: $O(\log n)$ per operation (amortized) $\Longrightarrow O((m + n) \log n)$ running time.

Fibonacci Heap:
- Insert, Decrease-Key $O(1)$ amortized
- Extract-Min $O(\log n)$ amortized
$\Longrightarrow O(m + n \log n)$ running time