Lecture 16: Single-Source Shortest Paths

Jessica Sorrell

October 23, 2025
601.433/633 Introduction to Algorithms
Slides by Michael Dinitz

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 $\ell(P) = \sum_{(x,y) \in P} \ell(x, y)$
- $d(x, y) = \min_{x \to y \text{ paths } P} \ell(P)$

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 $\ell(P) = \sum_{(x,y) \in P} \ell(x, y)$
- $d(x, y) = \min_{x \to y \text{ paths } P} \ell(P)$

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:

- $OPT(u, i)$: 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
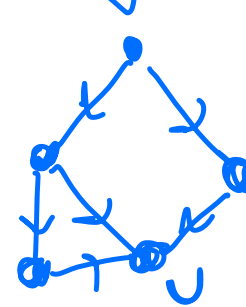
Dynamic Programming Approach

Subproblems:

- $OPT(u, i)$: 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)

$$\ell(OPT(u, k)) = \begin{cases} 0 & \text{if } u = v, k = 0 \\ \infty & \text{if } u \neq v, k = 0 \\ & \text{otherwise} \end{cases}$$

Dynamic Programming Approach

Subproblems:

- ▸ $OPT(u, i)$: 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)

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

Proof of Optimal Substructure

Induction on k.

$k = 0 : \checkmark$. So let $k \geq 1$.

Proof of Optimal Substructure

Induction on k.

$k = 0 : \checkmark$. So let $k \geq 1$.

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

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

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

Proof of Optimal Substructure

Induction on k.

$k = 0 : \checkmark$. So let $k \geq 1$.

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

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

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

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

$$\ell(OPT(u, k)) = \ell(P') + \ell(z, u) \geq \ell(OPT(z, k-1)) + \ell(z, u)$$

$$\geq \min_{w:(w,u)\in E}(\ell(OPT(w, k-1)) + \ell(w, u))$$

Bellman-Ford Algorithm

Obvious dynamic program!

$M[u, 0] = \infty$ 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!

$M[u, 0] = \infty$ for all $u \in V, u \neq v$
$M[v, 0] = 0$

```
for(k = 1 to n − 1) {
    for(u ∈ V) {
        M[u, k] = minw:(w,u)∈E(M[w, k − 1] + ℓ(w, u))
    }
}
```

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

Running Time:

Bellman-Ford Algorithm

Obvious dynamic program!

$M[u,0] = \infty$ 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)∈E}(M[w, k − 1] + ℓ(w, u))
    }
}
```

Running Time:

▸ Obvious: $O(n^3)$

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)∈E}(M[w, k − 1] + ℓ(w, u))
    }
}
```

Running Time:

▸ Obvious: $O(n^3)$

▸ Smarter: $O(mn)$

Bellman-Ford: Correctness

> **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, $M[u, k] = \ell(OPT(u, k))$ 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, $M[u, k] = \ell(OPT(u, k))$ for all $k \le n - 1$ and $u \in V$.

Proof.

Induction on k. Obviously true for $k = 0$.

$$M[u, k] = \min_{w:(w,u) \in E} (M[w, k-1]) + \ell(w, u)) \hspace{2cm} \text{(algorithm)}$$
$$= min_{w:(w,u) \in E} (\ell(OPT(w, k-1)) + \ell(w, u)) \hspace{1cm} \text{(induction)}$$
$$= \ell(OPT(u, k)) \hspace{2cm} \text{(optimal substructure)}$$
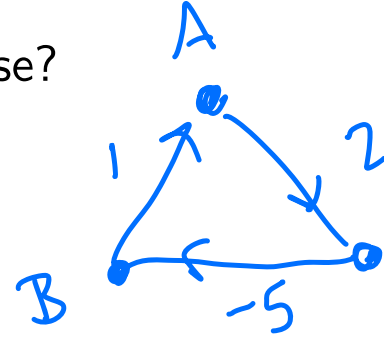
$\square$

Negative Weights and Cycle

Suppose weights are negative. Does the problem make sense?

Negative Weights and Cycle

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

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!

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!

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!

Fun fact: best-known algorithm with negative (real) edge weights until last year!

Jeremy Fineman. *Single-Source Shortest Paths with Negative Real Weights in $\tilde{O}(mn^{8/9})$ Time.*
STOC '24

Relaxations

Common primitive in shortest path algorithms

- ▶ Reinterpret Bellman-Ford via relaxations
- ▶ Use relaxations for Dijkstra's algorithm

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

```
relax(x, y) {
    if(d̂(y) > d̂(x) + ℓ(x, y)) {
        d̂(y) = d̂(x) + ℓ(x, y)
        y.parent = x
    }
}
```

Bellman-Ford as Relaxations

```
for(i = 1 to n) {
    foreach(u ∈ V) {
        foreach(edge (x, u)) {
            relax(x, u)
        }
    }
}
```

Bellman-Ford as Relaxations

```
for(i = 1 to n) {
    foreach(u ∈ V) {
        foreach(edge (x, u)) {
            relax(x, u)
        }
    }
}
```

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

$T = \varnothing$
$\hat{d}(v) = 0$
$\hat{d}(u) = \infty$ 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
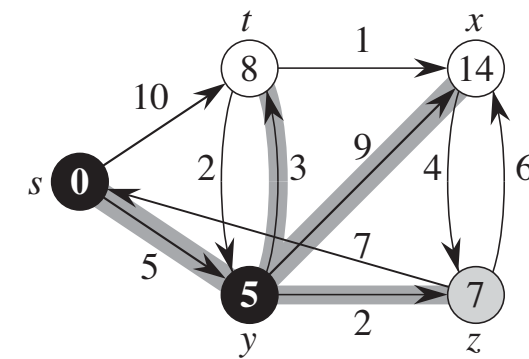   foreach edge (u, x) with $x \notin T$ {
        relax(u,x)
    }
}

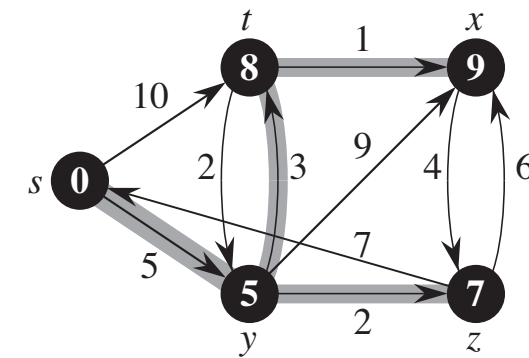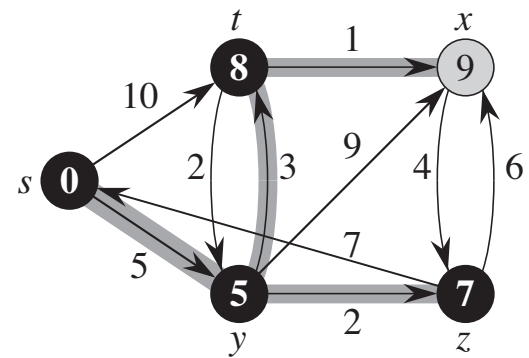# Dijkstra Example
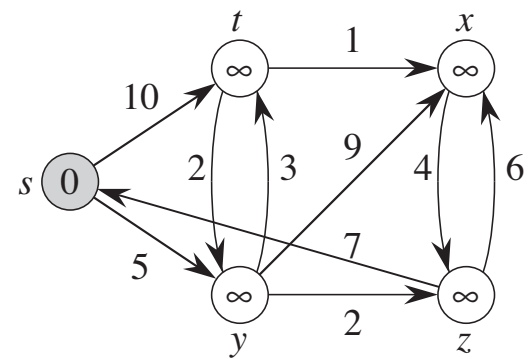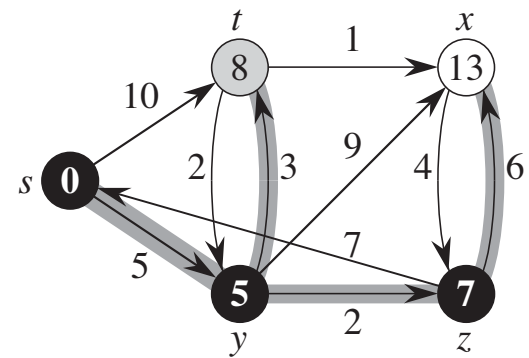

(a)
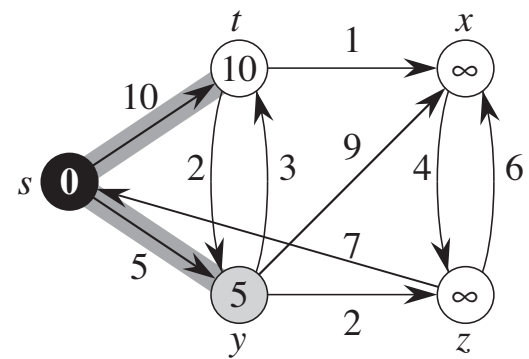
(b)

(c)

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* $u \in T$.

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 $w = u.parent$

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

Correctness: Inductive Step (Sketch)

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

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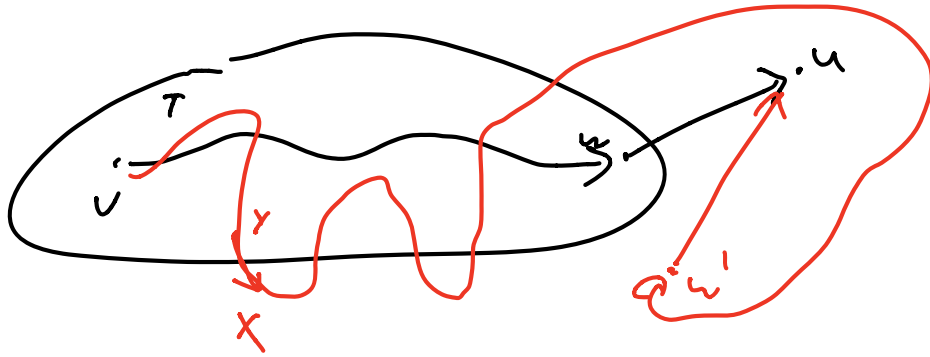
- ▸ Red path P actual shortest path, black path found by Dijkstra
- ▸ w' predecessor of u on P. Can't be in T.
  - ▸ If it was, would have $\hat{d}(w') = d(w')$ by induction, would have relaxed (w', u), so would have $w' = u.parent$
- ▸ x first node of P outside T, previous node y

Correctness: Inductive Step (Sketch)

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

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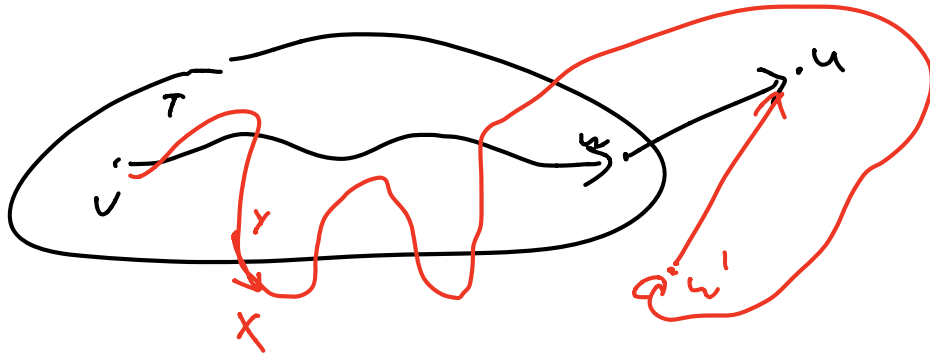

- Red path P actual shortest path, black path found by Dijkstra
- w' predecessor of u on P. Can't be in T.
  - If it was, would have $\hat{d}(w') = d(w')$ by induction, would have relaxed (w', u), so would have $w' = u.parent$
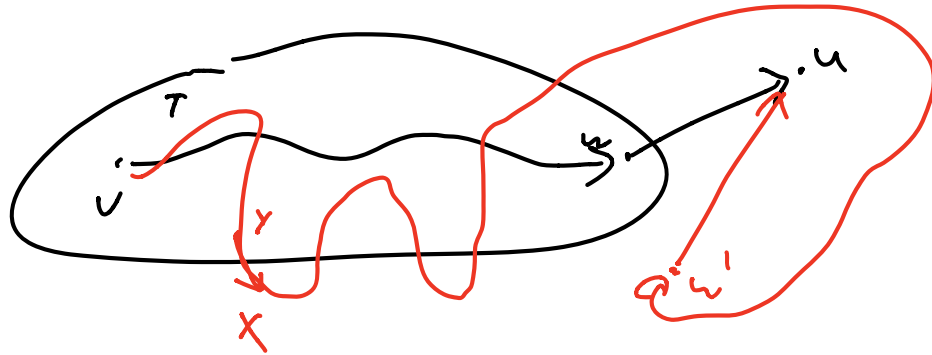- x first node of P outside T, previous node y

$$\hat{d}(x) \le \hat{d}(y) + \ell(y, x) = d(y) + \ell(y, x) < \ell(P) = d(u) \le \hat{d}(u)$$

Correctness: Inductive Step (Sketch)

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

$\implies \hat{d}(u) = \hat{d}(w) + \ell(w, u) = d(w) + \ell(w, u)$ (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 $w' = u.parent$

▸ 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 $\implies \leq m$ 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

Binary heap: $O(\log n)$ per operation (amortized)
$\implies O((m + n)\log n)$ 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 $\implies \leq m$ 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

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

Fibonacci Heap:

- ▶ Insert, Decrease-Key $O(1)$ amortized
- ▶ Extract-Min $O(\log n)$ amortized
- $\implies O(m + n \log n)$ running time