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

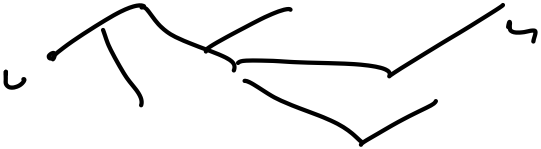
Introduction

Setup:

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

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

- $\mathbf{d}(\mathbf{u}) = \mathbf{d}(\mathbf{v}, \mathbf{u})$ for all $\mathbf{u} \in \mathbf{V}$
- Representation: "shortest path tree" out of $\mathbf{v}$.
- Often only care about distances – can reconstruct tree from distances.



Bellman-Ford

Dynamic Programming Approach

Subproblems:

- $\mathbf{OPT(u,i)}$: shortest path from $\mathbf{v}$ to $\mathbf{u}$ that uses at most $\mathbf{i}$ 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 $\mathbf{0}$

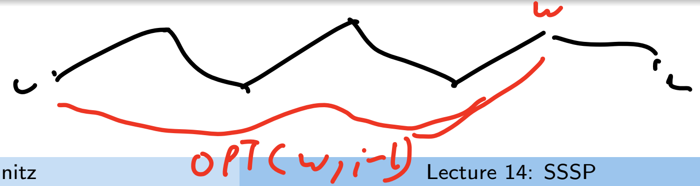
Dynamic Programming Approach

Subproblems:

- $\mathbf{OPT(u,i)}$: shortest path from $\mathbf{v}$ to $\mathbf{u}$ that uses at most $\mathbf{i}$ 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 $\mathbf{0}$

Theorem (Optimal Substructure)

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



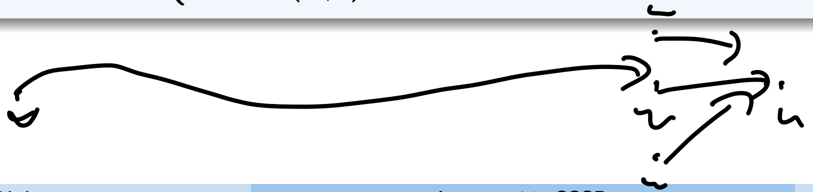
Dynamic Programming Approach

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)

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



Proof of Optimal Substructure

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

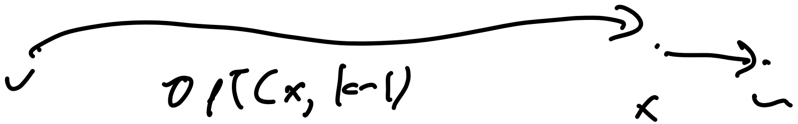
Proof of Optimal Substructure

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

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

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

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



Proof of Optimal Substructure

$\mathbf{k = 0}$: ✓. So let $\mathbf{k \geq 1}$.

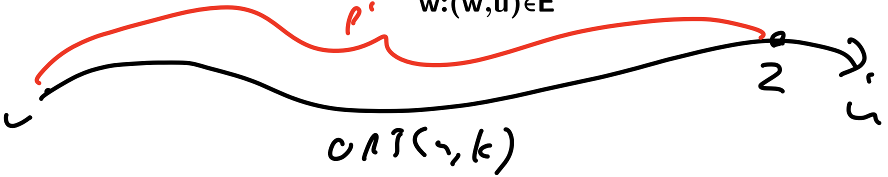
- $\leq$: Let $\mathbf{x = \arg\min_{w:(w,u) \in E} (\ell(OPT(w, k-1)) + \ell(w,u))}$

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

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

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

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



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

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

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($u \in V$) {
        $M[u,k] = \min_{w:(w,u)\in E}(M[w,k-1] + \ell(w,u))$   $\}$ $O(m)$ (assuming $m \geq n-1$)
    }
}
```

Running Time:

- Obvious: $O(n^3)$

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

Running Time:

- Obvious: $O(n^3)$
- Smarter: $O(mn)$

Bellman-Ford: Correctness

Theorem

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

Bellman-Ford: Correctness

Theorem

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

Proof.

Induction on $\mathbf{k}$. Obviously true for $\mathbf{k} = \mathbf{0}$.

Bellman-Ford: Correctness

Theorem

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

Proof.

Induction on $\mathbf{k}$. Obviously true for $\mathbf{k = 0}$.

$$\begin{aligned}
\mathbf{M[u,k]} &= \min_{\mathbf{w:(w,u)\in E}} \mathbf{(M[w,k-1]) + \ell(w,u))} && \text{(algorithm)} \\
&= \mathbf{\min_{w:(w,u)\in E}(\ell(OPT(w,k-1)) + \ell(w,u))} && \text{(induction)} \\
&= \mathbf{\ell(OPT(u,k))} && \text{(optimal substructure)}
\end{aligned}$$

☐

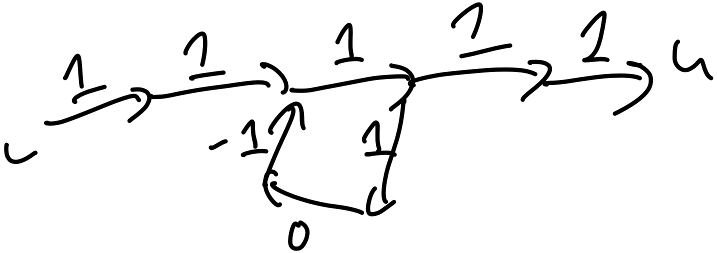
Negative Weights and Cycle

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!

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

$\hat{\mathbf{d}}(\mathbf{u})$: upper bound on $\mathbf{d}(\mathbf{u})$

- Initially: $\hat{\mathbf{d}}(\mathbf{v}) = \mathbf{0}$, $\hat{\mathbf{d}}(\mathbf{u}) = \infty$ for all $\mathbf{u} \neq \mathbf{v}$

Relaxations

Common primitive in shortest path algorithms

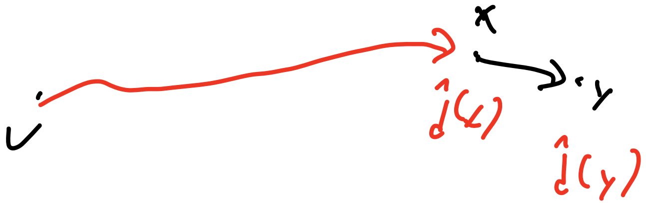
- Reinterpret Bellman-Ford via relaxations
- Use relaxations for Dijkstra's algorithm

$\hat{\mathbf{d}}(\mathbf{u})$: upper bound on $\mathbf{d}(\mathbf{u})$

- Initially: $\hat{\mathbf{d}}(\mathbf{v}) = \mathbf{0}$, $\hat{\mathbf{d}}(\mathbf{u}) = \infty$ for all $\mathbf{u} \neq \mathbf{v}$

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

Relaxations



Common primitive in shortest path algorithms

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

$\hat{\mathbf{d}}(\mathbf{u})$: upper bound on $\mathbf{d}(\mathbf{u})$

- Initially: $\hat{\mathbf{d}}(\mathbf{v}) = \mathbf{0}$, $\hat{\mathbf{d}}(\mathbf{u}) = \infty$ for all $\mathbf{u} \neq \mathbf{v}$

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

```
relax($\mathbf{x}, \mathbf{y}$) {
    if($\hat{\mathbf{d}}(\mathbf{y}) > \hat{\mathbf{d}}(\mathbf{x}) + \ell(\mathbf{x}, \mathbf{y})$) {
        $\hat{\mathbf{d}}(\mathbf{y}) = \hat{\mathbf{d}}(\mathbf{x}) + \ell(\mathbf{x}, \mathbf{y})$
        y.parent = x
    }
}
```

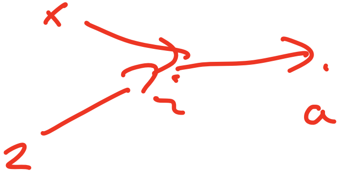
Bellman-Ford as Relaxations

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

Bellman-Ford as Relaxations

```
for($\mathbf{i} = \mathbf{1}$ to $\mathbf{n}$) {
    foreach($\mathbf{u} \in \mathbf{V}$) {
        foreach(edge $(\mathbf{x}, \mathbf{u})$) {
            relax($\mathbf{x}, \mathbf{u}$)
        }
    }
}
```

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)

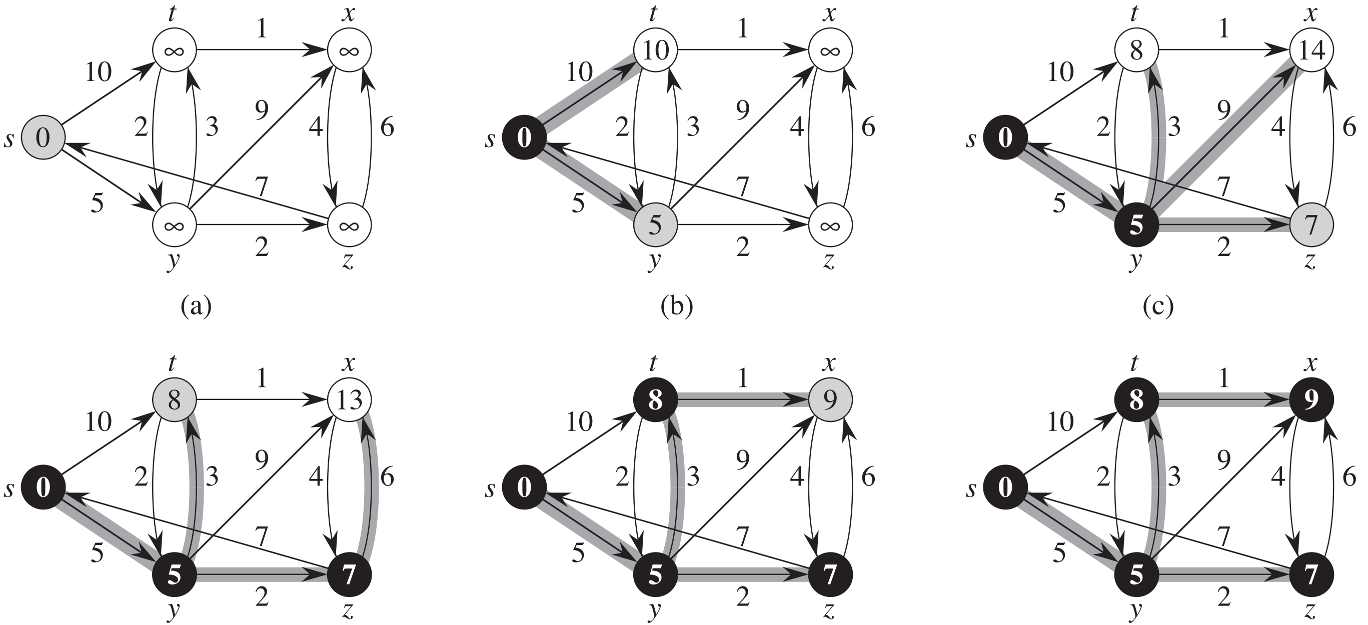
Cons: Doesn’t work with negative edge weights.

Dijkstra's Algorithm

```
$\mathbf{T} = \varnothing$
$\hat{\mathbf{d}}(\mathbf{v}) = \mathbf{0}$
$\hat{\mathbf{d}}(\mathbf{u}) = \infty$ for all $\mathbf{u} \neq \mathbf{v}$

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

Dijkstra Example



Dijkstra Correctness

Theorem

Throughout the algorithm:

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

Dijkstra Correctness

Theorem

Throughout the algorithm:

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

Proof. Induction on $|\mathbf{T}|$ (iterations of algorithm)

Dijkstra Correctness

Theorem

Throughout the algorithm:

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

Proof. Induction on $|\mathbf{T}|$ (iterations of algorithm)

Base Case: After first iteration (when $|\mathbf{T}| = \mathbf{1}$), added $\mathbf{v}$ to $\mathbf{T}$ with $\mathbf{\hat{d}(v) = d(v) = 0}$ ✓

Correctness: Inductive Step (Sketch)

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

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

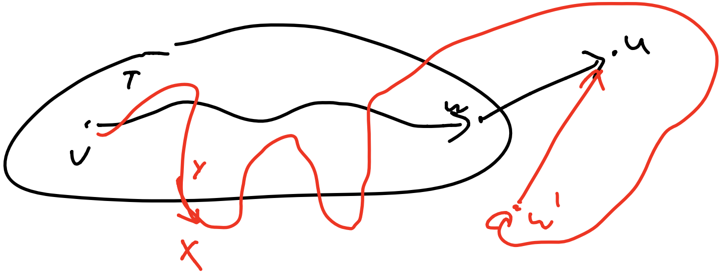
↗

induction

Correctness: Inductive Step (Sketch)

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

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

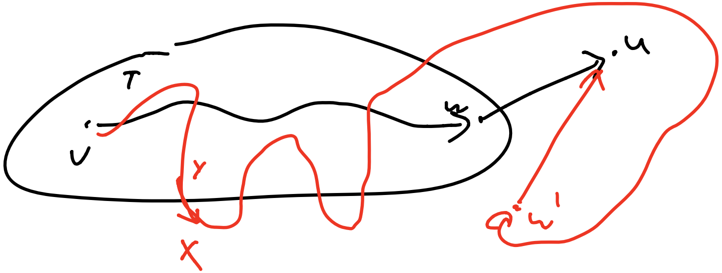


- Red path **P** actual shortest path, black path found by Dijkstra
- **w′** predecessor of **u** on **P**. Can't be in **T**.
  - If it was, would have $\hat{\mathbf{d}}(\mathbf{w}') = \mathbf{d}(\mathbf{w}')$ by induction, would have relaxed $(\mathbf{w}',\mathbf{u})$, so would have **w′** = **u.parent**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

$\implies \hat{\mathbf{d}}(\mathbf{u}) = \hat{\mathbf{d}}(\mathbf{w}) + \ell(\mathbf{w},\mathbf{u}) = \mathbf{d}(\mathbf{w}) + \ell(\mathbf{w},\mathbf{u})$ (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 $\hat{\mathbf{d}}(\mathbf{w'}) = \mathbf{d}(\mathbf{w'})$ by induction, would have relaxed $(\mathbf{w'},\mathbf{u})$, so would have **w′** = **u.parent**
- **x** first node of **P** outside **T**, previous node **y**

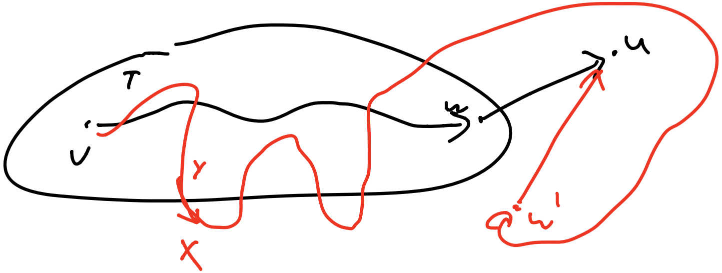
$$\hat{\mathbf{d}}(\mathbf{x}) \le \hat{\mathbf{d}}(\mathbf{y}) + \ell(\mathbf{y},\mathbf{x}) = \mathbf{d}(\mathbf{y}) + \ell(\mathbf{y},\mathbf{x}) < \ell(\mathbf{P}) = \mathbf{d}(\mathbf{u}) \le \hat{\mathbf{d}}(\mathbf{u})$$

relaxed (y,x) induction

Correctness: Inductive Step (Sketch)

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

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



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

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

Contradiction! Algorithm would have chosen $\mathbf{x}$ next, not $\mathbf{u}$.

Running Time

Algorithm needs to:

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

Running Time

Algorithm needs to:

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

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

Running Time

Algorithm needs to:

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

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

Keep $\hat{\mathbf{d}}$ values in a heap!

- Insert $\mathbf{n}$ times
- Extract-Min $\mathbf{n}$ times
- Decrease-Key $\mathbf{m}$ times

Running Time

Algorithm needs to:

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

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

Keep $\hat{\mathbf{d}}$ values in a heap!

- Insert $\mathbf{n}$ times
- Extract-Min $\mathbf{n}$ times
- Decrease-Key $\mathbf{m}$ times

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

Running Time

Algorithm needs to:

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

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

Keep $\hat{\mathbf{d}}$ values in a heap!

- Insert $\mathbf{n}$ times
- Extract-Min $\mathbf{n}$ times
- Decrease-Key $\mathbf{m}$ times

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

Fibonacci Heap:

- Insert, Decrease-Key $\mathbf{O(1)}$ amortized
- Extract-Min $\mathbf{O(\log n)}$ amortized

$\implies \mathbf{O(m+n\log n)}$ running time