# Shortest Paths

Eric Price

UT Austin

CS 331, Spring 2020 Coronavirus Edition

# Talk Outline

1. Logistics
2. Shortest Paths: Bellman-Ford
3. Dijkstra's Algorithm

# Talk Outline

1. Logistics
2. Shortest Paths: Bellman-Ford
3. Dijkstra's Algorithm

# Welcome to 331 online!

- I'll be experimenting with different teaching methods.

# Welcome to 331 online!

- I'll be experimenting with different teaching methods.
  - Bear with me.

# Welcome to 331 online!

- I'll be experimenting with different teaching methods.
  - Bear with me.
  - Give feedback!

# Welcome to 331 online!

- I'll be experimenting with different teaching methods.
  - Bear with me.
  - Give feedback!
- Stay healthy

# Welcome to 331 online!

- I'll be experimenting with different teaching methods.
  - Bear with me.
  - Give feedback!
- Stay healthy
  - Let me know if you're sick & need extra time.

# Welcome to 331 online!

- I'll be experimenting with different teaching methods.
  - Bear with me.
  - Give feedback!
- Stay healthy
  - Let me know if you're sick & need extra time.
  - Please wear a mask if you interact with others.

# Welcome to 331 online!

- I'll be experimenting with different teaching methods.
  - Bear with me.
  - Give feedback!
- Stay healthy
  - Let me know if you're sick & need extra time.
  - Please wear a mask if you interact with others.
- Second exam will be take-home

# Welcome to 331 online!

- I'll be experimenting with different teaching methods.
  - Bear with me.
  - Give feedback!
- Stay healthy
  - Let me know if you're sick & need extra time.
  - Please wear a mask if you interact with others.
- Second exam will be take-home
  - Given out after class on April 8.

# Welcome to 331 online!

- I'll be experimenting with different teaching methods.
  - Bear with me.
  - Give feedback!
- Stay healthy
  - Let me know if you're sick & need extra time.
  - Please wear a mask if you interact with others.
- Second exam will be take-home
  - Given out after class on April 8.
  - Return before *10am* on Friday, April 10.

# Zoom details

- Mute your audio unless you have a question.

# Zoom details

- Mute your audio unless you have a question.
- I encourage sharing your video
  - Makes this seem more like a community

# Zoom details

- Mute your audio unless you have a question.
- I encourage sharing your video
  - Makes this seem more like a community
- You can "raise your hand" so I know to call on you.

# Zoom details

- Mute your audio unless you have a question.
- I encourage sharing your video
  - Makes this seem more like a community
- You can "raise your hand" so I know to call on you.
  - Press "View Participants", then the little blue hand.

# Zoom details

- Mute your audio unless you have a question.
- I encourage sharing your video
  - Makes this seem more like a community
- You can "raise your hand" so I know to call on you.
  - Press "View Participants", then the little blue hand.
- Videos will be recorded & available after class.

# Talk Outline

1. Logistics
2. Shortest Paths: Bellman-Ford
3. Dijkstra's Algorithm

# Single-Source Shortest Paths

- Problem setup:

# Single-Source Shortest Paths

- Problem setup:
  - Given a directed graph $G = (V, E)$

# Single-Source Shortest Paths

- Problem setup:
  - Given a directed graph $G = (V, E)$
  - Each edge $u \to v$ has *distance*: $w(u \to v) \in \mathbb{R}$

# Single-Source Shortest Paths

- Problem setup:
  - Given a directed graph $G = (V, E)$
  - Each edge $u \to v$ has *distance*: $w(u \to v) \in \mathbb{R}$
  - Distance of path is sum of distance of edges.

# Single-Source Shortest Paths

- Problem setup:
  - Given a directed graph $G = (V, E)$
  - Each edge $u \to v$ has *distance*: $w(u \to v) \in \mathbb{R}$
  - Distance of path is sum of distance of edges.
  - Given a source $s$

# Single-Source Shortest Paths

- Problem setup:
  - Given a directed graph $G = (V, E)$
  - Each edge $u \to v$ has *distance*: $w(u \to v) \in \mathbb{R}$
  - Distance of path is sum of distance of edges.
  - Given a source $s$
- Goal: for every $v$, compute $c^*(v)$, the distance of shortest $s \rightsquigarrow v$ path in $G$.

# Single-Source Shortest Paths

- Problem setup:
  - Given a directed graph $G = (V, E)$
  - Each edge $u \to v$ has *distance*: $w(u \to v) \in \mathbb{R}$
  - Distance of path is sum of distance of edges.
  - Given a source $s$
- Goal: for every $v$, compute $c^*(v)$, the distance of shortest $s \rightsquigarrow v$ path in $G$.
- Output two arrays: dist() and pred().

# Single-Source Shortest Paths

- Problem setup:
  - Given a directed graph $G = (V, E)$
  - Each edge $u \to v$ has *distance*: $w(u \to v) \in \mathbb{R}$
  - Distance of path is sum of distance of edges.
  - Given a source $s$
- Goal: for every $v$, compute $c^*(v)$, the distance of shortest $s \rightsquigarrow v$ path in $G$.
- Output two arrays: dist() and pred().
  - $\text{dist}(v) = c^*(v)$ for all $v$

# Single-Source Shortest Paths

- Problem setup:
  - Given a directed graph $G = (V, E)$
  - Each edge $u \to v$ has *distance*: $w(u \to v) \in \mathbb{R}$
  - Distance of path is sum of distance of edges.
  - Given a source $s$
- Goal: for every $v$, compute $c^*(v)$, the distance of shortest $s \leadsto v$ path in $G$.
- Output two arrays: dist() and pred().
  - $\text{dist}(v) = c^*(v)$ for all $v$
  - $\text{pred}(s) = \textsc{None}$

# Single-Source Shortest Paths

- Problem setup:
  - Given a directed graph $G = (V, E)$
  - Each edge $u \to v$ has *distance*: $w(u \to v) \in \mathbb{R}$
  - Distance of path is sum of distance of edges.
  - Given a source $s$
- Goal: for every $v$, compute $c^*(v)$, the distance of shortest $s \rightsquigarrow v$ path in $G$.
- Output two arrays: dist() and pred().
  - $\text{dist}(v) = c^*(v)$ for all $v$
  - $\text{pred}(s) = \textsc{None}$
  - $v \leftarrow \text{pred}(v) \leftarrow \text{pred}(\text{pred}(v)) \leftarrow \cdots \leftarrow s$ is shortest $s \rightsquigarrow v$ path.

# Single-Source Shortest Paths

- Problem setup:
  - Given a directed graph $G = (V, E)$
  - Each edge $u \to v$ has *distance*: $w(u \to v) \in \mathbb{R}$
  - Distance of path is sum of distance of edges.
  - Given a source $s$
- Goal: for every $v$, compute $c^*(v)$, the distance of shortest $s \rightsquigarrow v$ path in $G$.
- Output two arrays: dist() and pred().
  - $\text{dist}(v) = c^*(v)$ for all $v$
  - $\text{pred}(s) = \textsc{None}$
  - $v \leftarrow \text{pred}(v) \leftarrow \text{pred}(\text{pred}(v)) \leftarrow \cdots \leftarrow s$ is shortest $s \rightsquigarrow v$ path.
- Question: what if $w(u \to v) = 1$ for all $u \to v \in E$?

# Generic SSSP algorithm

- We maintain a vector dist that satisfies the invariant:

$$\text{dist}(v) \geq c^*(v)$$

for all $v$ at all times.

# Generic SSSP algorithm

- We maintain a vector dist that satisfies the invariant:

$$\text{dist}(v) \geq c^*(v)$$

  for all $v$ at all times.

- $\textsc{InitializeSSSP}(s)$:
  - $\text{dist}(v) = \infty \quad \forall v$
  - $\text{pred}(v) = \textsc{None} \quad \forall v$
  - $\text{dist}(s) = 0$.

# Generic SSSP algorithm

- We maintain a vector dist that satisfies the invariant:

$$\text{dist}(v) \geq c^*(v)$$

  for all $v$ at all times.
- $\textsc{InitializeSSSP}$(s):
  - $\text{dist}(v) = \infty \quad \forall v$
  - $\text{pred}(v) = \textsc{None} \quad \forall v$
  - $\text{dist}(s) = 0$.
- $\textsc{FordSSSP}$(s):
  - $\textsc{InitializeSSSP}$(s)
  - Repeat:
    - ★ Pick an edge
    - ★ If it is "tense", *relax* it.

# Relaxing an edge

- $\textsc{Relax}(u \to v)$:
  - If $\text{dist}(v) > \text{dist}(u) + w(u \to v)$:
    - $\text{dist}(v) \leftarrow \text{dist}(u) + w(u \to v)$
    - $\text{pred}(v) \leftarrow u$.

**Triangle Inequality**

For any edge $u \to v$,

$$c^*(v) \le c^*(u) + w(u \to v).$$

**Lemma**

*If $dist(v) \ge c^*(v)$ for all $v$, then for any edge $u \to v$,*

$$c^*(v) \le dist(u) + w(u \to v).$$

*Hence $\textsc{Relax}$ preserves the invariant that $dist(v) \ge c^*(v) \forall v$.*

# Generic SSSP algorithm

- Invariant: $\text{dist}(v) \geq c^*(v)$ for all $v$ at all times.

# Generic SSSP algorithm

- Invariant: $\text{dist}(v) \geq c^*(v)$ for all $v$ at all times.
- $\text{InitializeSSSP}(s)$:
  - $\text{dist}(v) = \infty \quad \forall v$
  - $\text{pred}(v) = \text{None} \quad \forall v$
  - $\text{dist}(s) = 0$.

# Generic SSSP algorithm

- Invariant: $\text{dist}(v) \geq c^*(v)$ for all $v$ at all times.
- $\textsc{InitializeSSSP}(s)$:
  - $\text{dist}(v) = \infty \quad \forall v$
  - $\text{pred}(v) = \textsc{None} \quad \forall v$
  - $\text{dist}(s) = 0$.
- $\textsc{FordSSSP}(s)$:
  - $\textsc{InitializeSSSP}(s)$
  - Repeat some number of times:
    - ★ Pick an edge $u \to v$ (somehow)
    - ★ $\textsc{Relax}(u \to v)$

# Generic SSSP algorithm

- Invariant: $\text{dist}(v) \geq c^*(v)$ for all $v$ at all times.
- $\text{InitializeSSSP}(s)$:
  - $\text{dist}(v) = \infty \quad \forall v$
  - $\text{pred}(v) = \text{None} \quad \forall v$
  - $\text{dist}(s) = 0$.
- $\text{FordSSSP}(s)$:
  - $\text{InitializeSSSP}(s)$
  - Repeat some number of times:
    - ★ Pick an edge $u \to v$ (somehow)
    - ★ $\text{Relax}(u \to v)$
- $\text{Relax}(u \to v)$:
  - If $\text{dist}(v) > \text{dist}(u) + w(u \to v)$:
    - ★ $\text{dist}(v) \leftarrow \text{dist}(u) + w(u \to v)$
    - ★ $\text{pred}(v) \leftarrow u$.

# Analysis

- So far: $\text{dist}(v) \geq c^*(v)$.
- What we need: eventually $\text{dist}(v) = c^*(v)$.

**Lemma**

*Let $s = u_0 \to u_1 \to \cdots \to u_{k-1} \to u_k$ be a shortest $s \rightsquigarrow u_k$ path. After* Relax *has been called on every edge of this path in order—$u_0 \to u_1$, then $u_1 \to u_2$, until $u_{k-1} \to u_k$, with arbitrarily many other calls interleaved—then $\text{dist}(u_k) = c^*(u_k)$. Moreover, $u_k \leftarrow \text{pred}(u_k) \leftarrow \text{pred}(\text{pred}(u_k)) \leftarrow \cdots \leftarrow s$ is a shortest $s \rightsquigarrow u_k$ path.*

## Lemma

*Let* $s = u_0 \to u_1 \to \cdots \to u_{k-1} \to u_k$ *be a shortest* $s \rightsquigarrow u_k$ *path. After* Relax *has been called on every edge of this path in order—*$u_0 \to u_1$*, then* $u_1 \to u_2$*, until* $u_{k-1} \to u_k$*, with arbitrarily many other calls interleaved—then* $dist(u_k) = c^*(u_k)$.

## Proof.

Induct on $k$. Base case ($k = 0$) is easy.

## Lemma

*Let $s = u_0 \to u_1 \to \cdots \to u_{k-1} \to u_k$ be a shortest $s \rightsquigarrow u_k$ path. After* Relax *has been called on every edge of this path in order—$u_0 \to u_1$, then $u_1 \to u_2$, until $u_{k-1} \to u_k$, with arbitrarily many other calls interleaved—then $dist(u_k) = c^*(u_k)$.*

## Proof.

Induct on $k$. Base case ($k = 0$) is easy... or is it?

## Lemma

*Let* $s = u_0 \to u_1 \to \cdots \to u_{k-1} \to u_k$ *be a shortest* $s \rightsquigarrow u_k$ *path. After* RELAX *has been called on every edge of this path in order—*$u_0 \to u_1$*, then* $u_1 \to u_2$*, until* $u_{k-1} \to u_k$*, with arbitrarily many other calls interleaved—then* $dist(u_k) = c^*(u_k)$*.*

## Proof.

Induct on $k$. Base case ($k = 0$) is easy... or is it? Be careful about negative edges!

## Lemma

*Let* $s = u_0 \to u_1 \to \cdots \to u_{k-1} \to u_k$ *be a shortest* $s \rightsquigarrow u_k$ *path. After* $\textsc{Relax}$ *has been called on every edge of this path in order—*$u_0 \to u_1$*, then* $u_1 \to u_2$*, until* $u_{k-1} \to u_k$*, with arbitrarily many other calls interleaved—then* $dist(u_k) = c^*(u_k)$*.*

## Proof.

Induct on $k$. Base case ($k = 0$) is easy... or is it? Be careful about negative edges!

For the inductive step, assume it holds for all paths of length $k - 1$.

## Lemma

*Let $s = u_0 \to u_1 \to \cdots \to u_{k-1} \to u_k$ be a shortest $s \rightsquigarrow u_k$ path. After* Relax *has been called on every edge of this path in order—$u_0 \to u_1$, then $u_1 \to u_2$, until $u_{k-1} \to u_k$, with arbitrarily many other calls interleaved—then $dist(u_k) = c^*(u_k)$.*

## Proof.

Induct on $k$. Base case ($k = 0$) is easy... or is it? Be careful about negative edges!

For the inductive step, assume it holds for all paths of length $k - 1$. So the last time $\textsc{Relax}(u_{k-1} \to u_k)$ is called, $\text{dist}(u_{k-1}) = c^*(u_{k-1})$.

## Lemma

*Let* $s = u_0 \to u_1 \to \cdots \to u_{k-1} \to u_k$ *be a shortest* $s \rightsquigarrow u_k$ *path. After* $\textsc{Relax}$ *has been called on every edge of this path in order—*$u_0 \to u_1$*, then* $u_1 \to u_2$*, until* $u_{k-1} \to u_k$*, with arbitrarily many other calls interleaved—then* $dist(u_k) = c^*(u_k)$*.*

## Proof.

Induct on $k$. Base case ($k = 0$) is easy... or is it? Be careful about negative edges!

For the inductive step, assume it holds for all paths of length $k - 1$. So the last time $\textsc{Relax}(u_{k-1} \to u_k)$ is called, $\text{dist}(u_{k-1}) = c^*(u_{k-1})$. Therefore after this,

$$\text{dist}(u_k) \leq c^*(u_{k-1}) + w(u_{k-1} \to u_k).$$

## Lemma

*Let $s = u_0 \to u_1 \to \cdots \to u_{k-1} \to u_k$ be a shortest $s \leadsto u_k$ path. After* Relax *has been called on every edge of this path in order—$u_0 \to u_1$, then $u_1 \to u_2$, until $u_{k-1} \to u_k$, with arbitrarily many other calls interleaved—then $dist(u_k) = c^*(u_k)$.*

## Proof.

Induct on $k$. Base case ($k = 0$) is easy... or is it? Be careful about negative edges!

For the inductive step, assume it holds for all paths of length $k - 1$. So the last time $\textsc{Relax}(u_{k-1} \to u_k)$ is called, $\text{dist}(u_{k-1}) = c^*(u_{k-1})$. Therefore after this,

$$\text{dist}(u_k) \leq c^*(u_{k-1}) + w(u_{k-1} \to u_k).$$

Since $u_0 \to u_1 \to \cdots \to u_{k-1} \to u_k$ is a shortest path, this RHS is $c^*(u_k)$. □

# Question for you all

**Lemma**

*Let* $s = u_0 \to u_1 \to \cdots \to u_{k-1} \to u_k$ *be a shortest* $s \rightsquigarrow u_k$ *path. After* RELAX *has been called on every edge of this path in order—*$u_0 \to u_1$*, then* $u_1 \to u_2$*, until* $u_{k-1} \to u_k$*, with arbitrarily many other calls interleaved—then* $dist(u_k) = c^*(u_k)$*.*

# Question for you all

**Lemma**

*Let $s = u_0 \to u_1 \to \cdots \to u_{k-1} \to u_k$ be a shortest $s \rightsquigarrow u_k$ path. After* Relax *has been called on every edge of this path in order—$u_0 \to u_1$, then $u_1 \to u_2$, until $u_{k-1} \to u_k$, with arbitrarily many other calls interleaved—then $dist(u_k) = c^*(u_k)$.*

What happens with negative edges?

# Question for you all

**Lemma**

*Let $s = u_0 \to u_1 \to \cdots \to u_{k-1} \to u_k$ be a shortest $s \rightsquigarrow u_k$ path. After* RELAX *has been called on every edge of this path in order—$u_0 \to u_1$, then $u_1 \to u_2$, until $u_{k-1} \to u_k$, with arbitrarily many other calls interleaved—then $dist(u_k) = c^*(u_k)$.*

What happens with negative edges?

What happens with negative cycles?

# Back to the algorithm

- $\textsc{InitializeSSSP}(s)$:
  - $\text{dist}(v) = \infty \quad \forall v$
  - $\text{pred}(v) = \textsc{None} \quad \forall v$
  - $\text{dist}(s) = 0$.
- $\textsc{FordSSSP}(s)$:
  - $\textsc{InitializeSSSP}(s)$
  - Repeat some number of times:
    - Pick an edge $u \to v$ (somehow)
    - $\textsc{Relax}(u \to v)$
- $\textsc{Relax}(u \to v)$:
  - If $\text{dist}(v) > \text{dist}(u) + w(u \to v)$:
    - $\text{dist}(v) \leftarrow \text{dist}(u) + w(u \to v)$
    - $\text{pred}(v) \leftarrow u$.

# Back to the algorithm

- $\textsc{InitializeSSSP}(s)$:
  - $\text{dist}(v) = \infty \quad \forall v$
  - $\text{pred}(v) = \textsc{None} \quad \forall v$
  - $\text{dist}(s) = 0$.
- $\textsc{FordSSSP}(s)$:
  - $\textsc{InitializeSSSP}(s)$
  - Repeat some number of times:
    - Pick an edge $u \to v$ (somehow)
    - $\textsc{Relax}(u \to v)$
- $\textsc{Relax}(u \to v)$:
  - If $\text{dist}(v) > \text{dist}(u) + w(u \to v)$:
    - $\text{dist}(v) \leftarrow \text{dist}(u) + w(u \to v)$
    - $\text{pred}(v) \leftarrow u$.

# Back to the algorithm

- $\textsc{InitializeSSSP}(s)$:
  - $\text{dist}(v) = \infty \quad \forall v$
  - $\text{pred}(v) = \textsc{None} \quad \forall v$
  - $\text{dist}(s) = 0$.
- $\textsc{FordSSSP}(s)$:
  - $\textsc{InitializeSSSP}(s)$
  - Repeat some number of times:
    - Pick an edge $u \to v$ (somehow)
    - $\textsc{Relax}(u \to v)$
- $\textsc{Relax}(u \to v)$:
  - If $\text{dist}(v) > \text{dist}(u) + w(u \to v)$:
    - $\text{dist}(v) \leftarrow \text{dist}(u) + w(u \to v)$
    - $\text{pred}(v) \leftarrow u$.
- Lemma states: need to call $\textsc{Relax}$ *in order* for every shortest path.

# Bellman-Ford Algorithm

- Lemma states: need to call RELAX *in order* for every shortest path.

# Bellman-Ford Algorithm

- Lemma states: need to call RELAX *in order* for every shortest path.
- Every shortest path has length at most $V - 1$.

# Bellman-Ford Algorithm

- Lemma states: need to call RELAX *in order* for every shortest path.
- Every shortest path has length at most $V - 1$.
- If we relax *every* edge, we'll surely relax the first edge of the path.

# Bellman-Ford Algorithm

- Lemma states: need to call RELAX *in order* for every shortest path.
- Every shortest path has length at most $V - 1$.
- If we relax *every* edge, we'll surely relax the first edge of the path.
- If we relax every edge again, we'll get the second edge.

# Bellman-Ford Algorithm

- Lemma states: need to call Relax *in order* for every shortest path.
- Every shortest path has length at most $V - 1$.
- If we relax *every* edge, we'll surely relax the first edge of the path.
- If we relax every edge again, we'll get the second edge.
- Do this $V - 1$ times.

# Bellman-Ford Algorithm

- Lemma states: need to call RELAX *in order* for every shortest path.
- Every shortest path has length at most $V - 1$.
- If we relax *every* edge, we'll surely relax the first edge of the path.
- If we relax every edge again, we'll get the second edge.
- Do this $V - 1$ times.
- BELLMANFORD(s):
  - INITIALIZESSSP(s)
  - Repeat $V - 1$ times:
    - For every edge $u \to v$ in $E$:
      RELAX($u \to v$)

# Bellman-Ford Algorithm

- Lemma states: need to call RELAX *in order* for every shortest path.
- Every shortest path has length at most $V - 1$.
- If we relax *every* edge, we'll surely relax the first edge of the path.
- If we relax every edge again, we'll get the second edge.
- Do this $V - 1$ times.
- BELLMANFORD(s):
  - INITIALIZESSSP(s)
  - Repeat $V - 1$ times:
    - For every edge $u \to v$ in $E$:
      RELAX($u \to v$)
- $O(EV)$ time for SSSP.

# Bellman-Ford Algorithm

- Bellman-Ford solves SSSP in $O(EV)$ time.
- It works with negative edges.
- It's the fastest known algorithm in general!
- Can use to find *negative cycles*:
  - Repeat one more time. If no negative cycles, no edge should change in the $V$th iteration.
  - Follow the predecessor chain to find a negative cycle.
- Can go faster if edge lengths *nonnegative*: Dijkstra's algorithm.

# Talk Outline

# Dijkstra's Algorithm

- DIJKSTRA(s):
  - INITIALIZESSSP(s)
  - Repeat $V$ times:
    - ★ Find the unvisited vertex $u$ of minimal dist($u$).
    - ★ For every edge $u \to v$ out from $u$:
      RELAX($u \to v$)
- Alternative view: WHATEVERFIRSTSEARCH that visits the *nearest vertex to s*.
- Another alternative view: a small variant on Prim's algorithm.

# Dijkstra's Algorithm

```
function Dijkstra(s)
    pred, dist ← {}, {}
    q ← PriorityQueue([(0, s, None)])                ▷ dist, vertex, pred
    while q do
        d, u, parent ← q.pop()
        if u ∈ pred then
            continue
        pred[u] ← parent
        dist[u] ← d
        for u → v ∈ E do
            q.push( (dist[u] + w(u → v), v, u) )
    return dist, pred
```

# ~~Dijkstra's~~ Prim's Algorithm

```
function PRIM(s)
    pred, dist ← {}, {}
    q ← PRIORITYQUEUE([(0, s, None)])            ▷ dist, vertex, pred
    while q do
        d, u, parent ← q.pop()
        if u ∈ pred then
            continue
        pred[u] ← parent
        dist[u] ← d
        for u → v ∈ E do
            q.push( (dist[u] + w(u → v), v, u) )
    return dist, pred
```

# Dijkstra's Algorithm

- Just like Prim: visits each vertex once and scans through outgoing edges, so looks at each edge once.

# Dijkstra's Algorithm

- Just like Prim: visits each vertex once and scans through outgoing edges, so looks at each edge once.
  - Time from $E$ pushes/pops, for $O(E \log V)$ with binary heap.

# Dijkstra's Algorithm

- Just like Prim: visits each vertex once and scans through outgoing edges, so looks at each edge once.
  - Time from $E$ pushes/pops, for $O(E \log V)$ with binary heap.
  - Modifying the algorithm slightly and using a Fibonacci heap can bring this down to $O(E + V \log V)$.

# Dijkstra's Algorithm

- Just like Prim: visits each vertex once and scans through outgoing edges, so looks at each edge once.
  - Time from $E$ pushes/pops, for $O(E \log V)$ with binary heap.
  - Modifying the algorithm slightly and using a Fibonacci heap can bring this down to $O(E + V \log V)$.
- Tricky part: correctness.

# Dijkstra's Algorithm

- Just like Prim: visits each vertex once and scans through outgoing edges, so looks at each edge once.
  - Time from $E$ pushes/pops, for $O(E \log V)$ with binary heap.
  - Modifying the algorithm slightly and using a Fibonacci heap can bring this down to $O(E + V \log V)$.
- Tricky part: correctness.
- Need to argue: if edge weights nonnegative, for any shortest path, will visit the vertices *in order*.

# Dijkstra's Algorithm

- Just like Prim: visits each vertex once and scans through outgoing edges, so looks at each edge once.
  - Time from $E$ pushes/pops, for $O(E \log V)$ with binary heap.
  - Modifying the algorithm slightly and using a Fibonacci heap can bring this down to $O(E + V \log V)$.
- Tricky part: correctness.
- Need to argue: if edge weights nonnegative, for any shortest path, will visit the vertices *in order*.
  - Bellman-Ford relaxes each edge $V$ times.

# Dijkstra's Algorithm

- Just like Prim: visits each vertex once and scans through outgoing edges, so looks at each edge once.
  - Time from $E$ pushes/pops, for $O(E \log V)$ with binary heap.
  - Modifying the algorithm slightly and using a Fibonacci heap can bring this down to $O(E + V \log V)$.
- Tricky part: correctness.
- Need to argue: if edge weights nonnegative, for any shortest path, will visit the vertices *in order*.
  - Bellman-Ford relaxes each edge $V$ times.
  - Dijkstra only relaxes each edge once, so it better happen at the right time.

# Dijkstra's Algorithm: Correctness

- The distances $d$ popped from the queue are nondecreasing.

# Dijkstra's Algorithm: Correctness

- The distances $d$ popped from the queue are nondecreasing.
  - At each step, values pushed aren't smaller than the one just popped.

# Dijkstra's Algorithm: Correctness

- The distances $d$ popped from the queue are nondecreasing.
  - At each step, values pushed aren't smaller than the one just popped.

**Lemma**

*For any (not necessarily shortest) path $s = v_0 \rightarrow v_1 \rightarrow \ldots \rightarrow v_j$ of length $L_j$, then $dist[v_j]$ is at most $L_j$ when it is set.*

# Dijkstra's Algorithm: Correctness

- The distances $d$ popped from the queue are nondecreasing.
  - At each step, values pushed aren't smaller than the one just popped.

**Lemma**

*For any (not necessarily shortest) path $s = v_0 \to v_1 \to \ldots \to v_j$ of length $L_j$, then $dist[v_j]$ is at most $L_j$ when it is set.*

**Proof.**

Induct on $j$. For $j = 0$, trivially true.
If true for $j - 1$, then $dist[v_{j-1}] \leq L_{j-1}$. So when $v_{j-1}$ is visited, we will push $(d, v_j, v_{j-1})$ for

$$d = dist[v_{j-1}] + w(v_{j-1}, v_j) \leq L_{j-1} + w(v_{j-1}, v_j) = L_j$$

onto the queue. At some point this gets popped from the queue. Since the distances popped are nondecreasing, the *first* time we pop $v_j$ from the queue it must also be with a distance at most $L_j$. □

# Dijkstra's Algorithm: Conclusion

- Takes $O(E + V \log V)$ time.

# Dijkstra's Algorithm: Conclusion

- Takes $O(E + V \log V)$ time.
- Outputs the correct answer if all edge weights nonnegative.

# Dijkstra's Algorithm: Conclusion

- Takes $O(E + V \log V)$ time.
- Outputs the correct answer if all edge weights nonnegative.
- Alternative version:

# Dijkstra's Algorithm: Conclusion

- Takes $O(E + V \log V)$ time.
- Outputs the correct answer if all edge weights nonnegative.
- Alternative version:
  - Outputs the correct answer always.

# Dijkstra's Algorithm: Conclusion

- Takes $O(E + V \log V)$ time.
- Outputs the correct answer if all edge weights nonnegative.
- Alternative version:
  - Outputs the correct answer always.
  - Takes $O(E + V \log V)$ time if all edge weights nonnegative.

# Dijkstra's Algorithm: Conclusion

- Takes $O(E + V \log V)$ time.
- Outputs the correct answer if all edge weights nonnegative.
- Alternative version:
  - Outputs the correct answer always.
  - Takes $O(E + V \log V)$ time if all edge weights nonnegative.
  - Exponential time in general.

# Alternative Dijkstra: correct but slow with negative weights

```
function Dijkstra(s)
    pred, dist ← {}, {}
    q ← PriorityQueue([(0, s, None)])        ▷ dist, vertex, pred
    while q do
        d, u, parent ← q.pop()
        if u ∈ pred then
            continue
        pred[u] ← parent
        dist[u] ← d
        for u → v ∈ E do
            q.push( (dist[u] + w(u → v), v, u) )
    return dist, pred
```

# Alternative Dijkstra: correct but slow with negative weights

```
function DIJKSTRA(s)
    pred, dist ← {}, {}
    q ← PRIORITYQUEUE([(0, s, None)])                ▷ dist, vertex, pred
    while q do
        d, u, parent ← q.pop()
        if d ≥ dist[u] then
            continue
        pred[u] ← parent
        dist[u] ← d
        for u → v ∈ E do
            q.push( (dist[u] + w(u → v), v, u) )
    return dist, pred
```

# Summary of shortest paths

- DAGs: DP for $O(E)$ time.

# Summary of shortest paths

- DAGs: DP for $O(E)$ time.
- Unweighted graphs: BFS for $O(E)$ time.

# Summary of shortest paths

- DAGs: DP for $O(E)$ time.
- Unweighted graphs: BFS for $O(E)$ time.
- Bellman-Ford: $O(EV)$ time

# Summary of shortest paths

- DAGs: DP for $O(E)$ time.
- Unweighted graphs: BFS for $O(E)$ time.
- Bellman-Ford: $O(EV)$ time
  - Works with negative edge weights

# Summary of shortest paths

- DAGs: DP for $O(E)$ time.
- Unweighted graphs: BFS for $O(E)$ time.
- Bellman-Ford: $O(EV)$ time
  - Works with negative edge weights
  - Can detect cycles

# Summary of shortest paths

- DAGs: DP for $O(E)$ time.
- Unweighted graphs: BFS for $O(E)$ time.
- Bellman-Ford: $O(EV)$ time
  - Works with negative edge weights
  - Can detect cycles
- Dijkstra: $O(E + V \log V)$ time

# Summary of shortest paths

- DAGs: DP for $O(E)$ time.
- Unweighted graphs: BFS for $O(E)$ time.
- Bellman-Ford: $O(EV)$ time
  - Works with negative edge weights
  - Can detect cycles
- Dijkstra: $O(E + V \log V)$ time
  - but only with nonnegative edge weights

# Summary of shortest paths

- DAGs: DP for $O(E)$ time.
- Unweighted graphs: BFS for $O(E)$ time.
- Bellman-Ford: $O(EV)$ time
  - Works with negative edge weights
  - Can detect cycles
- Dijkstra: $O(E + V \log V)$ time
  - but only with nonnegative edge weights
  - either wrong or exponential time with negative edges

# Summary of shortest paths

- DAGs: DP for $O(E)$ time.
- Unweighted graphs: BFS for $O(E)$ time.
- Bellman-Ford: $O(EV)$ time
  - Works with negative edge weights
  - Can detect cycles
- Dijkstra: $O(E + V \log V)$ time
  - but only with nonnegative edge weights
  - either wrong or exponential time with negative edges
- Next class:

# Summary of shortest paths

- DAGs: DP for $O(E)$ time.
- Unweighted graphs: BFS for $O(E)$ time.
- Bellman-Ford: $O(EV)$ time
  - Works with negative edge weights
  - Can detect cycles
- Dijkstra: $O(E + V \log V)$ time
  - but only with nonnegative edge weights
  - either wrong or exponential time with negative edges
- Next class:
  - $A^*$ search: Dijkstra with a twist

# Summary of shortest paths

- DAGs: DP for $O(E)$ time.
- Unweighted graphs: BFS for $O(E)$ time.
- Bellman-Ford: $O(EV)$ time
  - Works with negative edge weights
  - Can detect cycles
- Dijkstra: $O(E + V \log V)$ time
  - but only with nonnegative edge weights
  - either wrong or exponential time with negative edges
- Next class:
  - $A^*$ search: Dijkstra with a twist
  - Exercises