Lecture 11

Weighted Graphs: Dijkstra and Bellman-Ford
Announcements

• The reading for last Wednesday was garbled on the website.
  • It is now fixed to Ch 22.5, on SCCs.

• Midterm THIS WEDNESDAY
  • In class
  • You may bring 1 double-sided page of notes
  • Covers up through last week (SCCs)
    • There haven’t been any HWs about graphs yet, and we know that.
    • You should know the basics.

• Practice exams online
  • Disclaimer: some material (like Dijkstra) appeared on previous midterms but will not be on this midterm.
Last week

• Graphs!
• DFS
  • Topological Sorting
  • Strongly Connected Components
• BFS
  • Shortest Paths in unweighted graphs
Today

• What if the graphs are **weighted**?
  • All nonnegative weights: Dijkstra!
  • If there are negative weights: Bellman-Ford!
Just the graph

How do I get from Gates to the Union?

Run BFS ...

I should go to the dish and then back to the union!

That doesn’t make sense if I label the edges by walking time.
Just the graph

How do I get from Gates to the Union?

weighted graph

If I pay attention to the weights...

If I pay attention to the weights...

I should go to Packard, then CS161, then the union.

w(u,v) = weight of edge between u and v.

For now, edge weights are non-negative.

w(u,v) = weight of edge between u and v.

For now, edge weights are non-negative.
Shortest path problem

• What is the shortest path between u and v in a weighted graph?
  • the cost of a path is the sum of the weights along that path
  • The shortest path is the one with the minimum cost.

• The distance $d(u,v)$ between two vertices u and v is the cost of the shortest path between u and v.

• For this lecture all graphs are directed, but to save on notation I’m just going to draw undirected edges.
Q: What’s the shortest path from Packard to the Union?

This is the shortest path from Gates to the Union.

It has cost 6.
Warm-up

• A sub-path of a shortest path is also a shortest path.

• Say this is a shortest path from s to t.

• Claim: this is a shortest path from s to x.
  • Suppose not, this one is shorter.
  • But then that gives an even shorter path from s to t!

\[ s \rightarrow x \rightarrow t \]
Single-source shortest-path problem

- I want to know the shortest path from one vertex (Gates) to all other vertices.

<table>
<thead>
<tr>
<th>Destination</th>
<th>Cost</th>
<th>To get there</th>
</tr>
</thead>
<tbody>
<tr>
<td>Packard</td>
<td>1</td>
<td>Packard</td>
</tr>
<tr>
<td>CS161</td>
<td>2</td>
<td>Packard-CS161</td>
</tr>
<tr>
<td>Hospital</td>
<td>10</td>
<td>Hospital</td>
</tr>
<tr>
<td>Caltrain</td>
<td>17</td>
<td>Caltrain</td>
</tr>
<tr>
<td>Union</td>
<td>6</td>
<td>Packard-CS161-Union</td>
</tr>
<tr>
<td>Stadium</td>
<td>10</td>
<td>Stadium</td>
</tr>
<tr>
<td>Dish</td>
<td>23</td>
<td>Packard-Dish</td>
</tr>
</tbody>
</table>

(Not necessarily stored as a table – how this information is represented will depend on the application)
Example

• I regularly have to solve “what is the shortest path from Palo Alto to [anywhere else]” using BART, Caltrain, light rail, MUNI, bus, Amtrak, bike, walking, Uber/Lyft.

• Edge weights have something to do with time, money, hassle. (They also change depending on my mood and traffic...).
Example

- **Network routing**
- I send information over the internet, from my computer to all over the world.
- Each path (from a router to another router) has a cost which depends on link length, traffic, other costs, etc..
- How should we send packets?
A few things that make these examples even more difficult

- Costs may change
  - If it’s raining the cost of biking is higher
  - If a link is congested, the cost of routing a packet along it is higher
- The network might not be known
  - My computer doesn’t store a map of the internet
- We want to do these tasks really quickly
  - I have time to bike to Berkeley, but not to contemplate biking to Berkeley...
  - More seriously, the internet.

This is a joke.
Let’s ignore these for now

Back to this example:
Graph is known and does not change.
Dijkstra’s algorithm

• What are the shortest paths from Gates to everywhere else?
Dijkstra

intuition

YOINK!

Gates
Packard
Dish
CS161
Union
A vertex is done when it’s not on the ground anymore.

Dijkstra intuition
Dijkstra
intuition
Dijkstra intuition

YOINK!

Diagram with nodes labeled Gates, Packard, CS161, Dish, and Union.
Dijkstra intuition
Dijkstra intuition

This also creates a tree structure!

The shortest paths are the lengths along this tree.
How do we actually implement this?

- **Without** string and gravity?
Dijkstra by example

How far is a node from Gates?

- I’m not sure yet
- I’m sure

x is my best over-estimate for a vertex v.
We’ll say \( d[v] = x \)

That is, an estimate of \( d(v, \text{Gates}) \).

Initialize \( d[v] = \infty \)
for all non-starting vertices v, and \( v[\text{Gates}] = 0 \)

- Pick the not-sure node u with the smallest estimate \( d[u] \).
Dijkstra by example

How far is a node from Gates?

- I’m not sure yet
- I’m sure
- x is my best over-estimate for a vertex v. We’ll say $d[v] = x$
- Current node u

- Pick the **not-sure** node u with the smallest estimate $d[u]$.
- Update all u’s neighbors v:
  - $d[v] = \min(d[v], d[u] + \text{edgeWeight}(u,v))$
Dijkstra by example

How far is a node from Gates?

- I’m not sure yet
- I’m sure
- \( x \) is my best over-estimate for a vertex \( v \). We’ll say \( d[v] = x \)
- Current node \( u \)

- Pick the not-sure node \( u \) with the smallest estimate \( d[u] \).
- Update all \( u \)’s neighbors \( v \):
  - \( d[v] = \min(d[v], d[u] + \text{edgeWeight}(u,v)) \)
- Mark \( u \) as sure.
Dijkstra by example

How far is a node from Gates?

- I’m not sure yet
- I’m sure
- x is my best over-estimate for a vertex v. We’ll say d[v] = x
- Current node u

- Pick the **not-sure** node u with the smallest estimate d[u].
- Update all u’s neighbors v:
  - d[v] = min( d[v] , d[u] + edgeWeight(u,v))
- Mark u as **sure**.
- Repeat
Dijkstra by example

How far is a node from Gates?

- I’m not sure yet
- I’m sure
- x is my best over-estimate for a vertex v. We’ll say $d[v] = x$
- Current node u

- Pick the **not-sure** node u with the smallest estimate $d[u]$.
- Update all u’s neighbors v:
  - $d[v] = \min( d[v] , d[u] + \text{edgeWeight}(u,v))$
- Mark u as **sure**.
- Repeat
Dijkstra by example

How far is a node from Gates?

- I’m not sure yet
- I’m sure
- x is my best over-estimate for a vertex v. We’ll say \( d[v] = x \)
- Current node \( u \)

- Pick the **not-sure** node \( u \) with the smallest estimate \( d[u] \).
- Update all \( u \)’s neighbors \( v \):
  - \( d[v] = \min( d[v] , d[u] + \text{edgeWeight}(u,v) ) \)
- Mark \( u \) as **sure**.
- Repeat
Dijkstra by example

How far is a node from Gates?

- I’m not sure yet
- I’m sure
- x is my best over-estimate for a vertex v. We’ll say $d[v] = x$
- Current node u

- Pick the **not-sure** node u with the smallest estimate $d[u]$.
- Update all u’s neighbors v:
  - $d[v] = \min( d[v], d[u] + \text{edgeWeight}(u,v) )$
- Mark u as **sure**.
- Repeat
Dijkstra by example

How far is a node from Gates?

- I’m not sure yet
- I’m sure
- $x$ is my best over-estimate for a vertex $v$. We’ll say $d[v] = x$
- Current node $u$

- Pick the **not-sure** node $u$ with the smallest estimate $d[u]$.
- Update all $u$’s neighbors $v$:
  - $d[v] = \min(d[v], d[u] + \text{edgeWeight}(u,v))$
- Mark $u$ as **sure**.
- Repeat
Dijkstra by example

How far is a node from Gates?

- I’m not sure yet
- I’m sure
- x is my best over-estimate for a vertex v. We’ll say d[v] = x
- Current node u

• Pick the **not-sure** node u with the smallest estimate d[u].
• Update all u’s neighbors v:
  • d[v] = min( d[v] , d[u] + edgeWeight(u,v) )
• Mark u as **sure**.
• Repeat
How far is a node from Gates?

- I’m not sure yet
- I’m sure
- $x$ is my best over-estimate for a vertex $v$. We’ll say $d[v] = x$
- Current node $u$

- Pick the **not-sure** node $u$ with the smallest estimate $d[u]$.
- Update all $u$’s neighbors $v$:
  - $d[v] = \min(d[v], d[u] + \text{edgeWeight}(u,v))$
- Mark $u$ as **sure**.
- Repeat
Dijkstra by example

How far is a node from Gates?

- I’m not sure yet
- I’m sure
- x is my best over-estimate for a vertex v.
  We’ll say \( d[v] = x \)
- Current node u

- Pick the not-sure node u with the smallest estimate \( d[u] \).
- Update all u’s neighbors v:
  - \( d[v] = \min(d[v], d[u] + \text{edgeWeight}(u,v)) \)
- Mark u as sure.
- Repeat
Dijkstra by example

How far is a node from Gates?

- I’m not sure yet
- I’m sure
- X: x is my best over-estimate for a vertex v. We’ll say d[v] = x
- Current node u

- Pick the not-sure node u with the smallest estimate $d[u]$.
- Update all u’s neighbors v:
  - $d[v] = \min( d[v], d[u] + \text{edgeWeight}(u,v) )$
- Mark u as sure.
- Repeat
Dijkstra by example

I’m not sure yet
I’m sure

x is my best over-estimate for a vertex v.
We’ll say $d[v] = x$

Current node u

• Pick the **not-sure** node u with the smallest estimate $d[u]$.
• Update all u’s neighbors v:
  • $d[v] = \min(d[v], d[u] + \text{edgeWeight}(u,v))$
• Mark u as **sure**.
• Repeat
Dijkstra by example

How far is a node from Gates?

- I’m not sure yet
- I’m sure
- x is my best over-estimate for a vertex v. We’ll say d[v] = x
- Current node u

- Pick the **not-sure** node u with the smallest estimate d[u].
- Update all u’s neighbors v:
  - d[v] = min( d[v] , d[u] + edgeWeight(u,v))
- Mark u as **sure**.
- Repeat

More formal pseudocode on board (or see CLRS)!
Why does this work?

• **Theorem:**
  - Run Dijkstra on G = (V, E).
  - At the end of the algorithm, the estimate \( d[v] \) is the actual distance \( d(\text{Gates}, v) \).

Let’s rename “Gates” to “s”, our starting vertex.

• Proof outline:
  - **Claim 1:** For all \( v \), \( d[v] \geq d(s, v) \).
  - **Claim 2:** When a vertex \( v \) is marked sure, \( d[v] = d(s, v) \).

• **Claims 1 and 2** imply the **theorem.**
  - \( d[v] \) never increases, so Claims 1 and 2 imply that \( d[v] \) weakly decreases until \( d[v] = d(s, v) \), then never changes again.
  - By the time we are sure about \( v \), \( d[v] = d(s, v) \). (Claim 1 again)
  - All vertices are eventually sure. (Stopping condition in algorithm)
  - So all vertices end up with \( d[v] = d(s, v) \).
Claim 1
\[ d[v] \geq d(s,v) \text{ for all } v. \]

- **Inductive hypothesis.**
  - After \( t \) iterations of Dijkstra, \( d[v] \geq d(s,v) \) for all \( v \).

- **Base case:**
  - At step 0, \( d(s,s) = 0 \), and \( d(s,v) \leq \infty \).

- **Inductive step:** say hypothesis holds for \( t \).
  - Then at step \( t+1 \):
    - We pick \( u \); for each neighbor \( v \):
      - \( d[v] \leftarrow \min( d[v] , d[u] + w(u,v) ) \geq d(s,v) \)

By induction, \( d(s,v) \leq d[v] \)

\[ d(s,v) \leq d(s,u) + d(u,v) \leq d[u] + w(u,v) \]
using induction again for \( d[u] \)

So the inductive hypothesis holds for \( t+1 \), and Claim 1 follows.
Claim 2
When a vertex u is marked sure, d[u] = d(s,u)

• To begin with:
  • The first vertex marked sure has d[s] = d(s,s) = 0.
• For t > 0:
  • Suppose that we are about to add u to the sure list.
  • That is, we picked u in the first line here:
    • Pick the not-sure node u with the smallest estimate d[u].
    • Update all u’s neighbors v:
      • d[v] ← min( d[v] , d[u] + edgeWeight(u,v))
    • Mark u as sure.
    • Repeat
Claim 2

- Want to show that $u$ is good.

Consider a **true** shortest path from $s$ to $u$:

The vertices in between are beige because they may or may not be **sure**.

Temporary definition:
$\nu$ is “good” means that $d[\nu] = d(s,\nu)$

True shortest path.
Claim 2

- Want to show that \( u \) is good. **BWOC, suppose it’s not.**
- Say \( z \) is the last good vertex before \( u \).
- \( z' \) is the vertex after \( z \).

Temporary definition:

\( v \) is “good” means that \( d[v] = d(s, v) \)

- means good
- means not good

“by way of contradiction”

It may be that \( z' = u \).

It may be that \( z = s \).

The vertices in between are beige because they may or may not be **sure.**

\( z \neq u \), since \( u \) is not good.

True shortest path.
Claim 2

- Want to show that \( u \) is good. \textbf{BWOC, suppose it’s not.}

\[
d[z] = d(s, z) \leq d(s, u) \leq d[u]
\]

- If \( d[z] = d[u] \), then \( u \) is good.
- If \( d[z] < d[u] \), then \( z \) is \textbf{sure.}
Claim 2

- Want to show that $u$ is good. **BWOC, suppose it’s not.**
- If $z$ is **sure** then we’ve already updated $z'$:
  - $d[z'] \leftarrow \min\{ d[z'], d[z] + w(z, z') \}$, so

\[
d[z'] \leq d[z] + w(z, z') = d(s, z') \leq d[z']
\]

**Temporary definition:**
- $v$ is “good” means that $d[v] = d(s,v)$
- Purple means good
- Red means not good

**CONTRADICTION!!**
Claim 2

- Want to show that $u$ is good. **BWOC, suppose it’s not.**

$$d[z] = d(s, z) \leq d(s, u) \leq d[u]$$

**Def. of $z$**

This is the shortest path from $s$ to $x$

**Claim 1**

- If $d[z] = d[u]$, then $u$ is good.
- If $d[z] < d[u]$, then $z$ is **sure**.

**So $u$ is good!**

aka $d[u] = d(s,v)$
Claim 2
When a vertex is marked sure, \( d[u] = d(s,u) \)

• To begin with:
  • The first vertex marked **sure** has \( d[s] = d(s,s) = 0 \).

• For \( t > 0 \):
  • Suppose that we are about to add \( u \) to the **sure** list.
  • That is, we picked \( u \) in the first line here:

    Then \( u \)

    **is good!**

    *Pick the **not-sure** node \( u \) with the smallest estimate \( d[u] \).*
    *Update all \( u \)'s neighbors \( v \):*
      * \( d[v] \leftarrow \min( d[v] , d[u] + \text{edgeWeight}(u,v)) \)*
    *Mark \( u \) as **sure**.*
    *Repeat*
Why does this work?

• **Theorem**: At the end of the algorithm, the estimate $d[v]$ is the actual distance $d(s,v)$.

• Proof outline:
  • **Claim 1**: For all $v$, $d[v] \geq d(s,v)$.
  • **Claim 2**: When a vertex is marked `sure`, $d[v] = d(s,v)$.

• **Claims 1 and 2** imply the **theorem**.
  • We will never mess up $d[v]$ after $v$ is marked `sure`, because $d[v]$ is a **decreasing over-estimate**.
Why does this work?

- **Theorem:**
  - Run Dijkstra on G = (V,E).
  - At the end of the algorithm, the estimate $d[v]$ is the actual distance $d(s,v)$.

- **Proof outline:**
  - **Claim 1:** For all $v$, $d[v] \geq d(s,v)$.
  - **Claim 2:** When a vertex $v$ is marked *sure*, $d[v] = d(s,v)$.

- **Claims 1 and 2 imply the theorem.**
  - $d[v]$ never increases, so Claims 1 and 2 imply that $d[v]$ weakly decreases until $d[v] = d(s,v)$, then never changes again.
  - By the time we are *sure* about $v$, $d[v] = d(s,v)$. (Claim 1 again)
  - All vertices are eventually *sure*. (Stopping condition in algorithm)
  - So all vertices end up with $d[v] = d(s,v)$. 
What did we just learn?

• Dijkstra’s algorithm can find **shortest paths** in weighted graphs with non-negative edge weights.

• Along the way, it constructs a nice tree.
  
  • We could post this tree in Gates, and it would be easy for anyone in Gates to figure out what the shortest path is to wherever they want to go.
Running time?

- Pick the **not-sure** node \( u \) with the smallest estimate \( d[u] \).
- Update all \( u \)'s neighbors \( v \):
  - \( d[v] \leftarrow \min( d[v] , d[u] + \text{edgeWeight}(u,v) ) \)
- Mark \( u \) as **sure**.
- Repeat

This will run for \( n \) iterations, since there’s one iteration per vertex.

How long does an iteration take?

Depends on how we implement it...
We need a data structure that:

- Stores unsure vertices \( v \)
- Keeps track of \( d[v] \)
- Can find \( v \) with minimum \( d[v] \)
  - `findMin()`
- Can remove that \( v \)
  - `removeMin(v)`
- Can update the \( d[v] \)
  - `updateKey(v,d)`

Pick the **not-sure** node \( u \) with the smallest estimate \( d[u] \).
- Update all \( u \)’s neighbors \( v \):
  - \( d[v] \leftarrow \min(d[v] , d[u] + \text{edgeWeight}(u,v)) \)
- Mark \( u \) as **sure**.
- Repeat

Total running time is big-oh of:

\[
\sum_{u \in V} \left( T(\text{findMin}) + \left( \sum_{v \in u.\text{neighbors}} T(\text{updateKey}) \right) + T(\text{removeMin}) \right)
\]

\( n(T(\text{findMin}) + T(\text{removeMin})) + m T(\text{updateKey}) \)
If we use an array

- $T(\text{findMin}) = O(n)$
- $T(\text{removeMin}) = O(n)$
- $T(\text{updateKey}) = O(1)$

- Running time of Dijkstra
  
  
  $= O(n(T(\text{findMin}) + T(\text{removeMin}))) + m T(\text{updateKey}))$

  $= O(n^2) + O(m)$

  $= O(n^2)$
If we use a red-black tree

- \( T(\text{findMin}) = O(\log(n)) \)
- \( T(\text{removeMin}) = O(\log(n)) \)
- \( T(\text{updateKey}) = O(\log(n)) \)

- **Running time of Dijkstra**
  
  \[
  O(n(T(\text{findMin}) + T(\text{removeMin})) + m T(\text{updateKey}))
  \]
  
  \[
  = O(n \log(n)) + O(m \log(n))
  \]
  
  \[
  = O((n + m) \log(n))
  \]

Better than an array if the graph is sparse!
aka \( m \) is much smaller than \( n^2 \)
Is a hash table a good idea here?

• Not really:
  • `Search(v)` is fast (in expectation)
  • But `findMin()` will still take time $O(n)$ without more structure.

$O(n(T(\text{findMin}) + T(\text{removeMin})) + mT(\text{updateKey}))$
Can also use a Fibonacci Heap

• This can do all operations in amortized time* O(1).
• Except deleteMin which takes amortized time* O(log(n)).
• See CS166 for more! (or CLRS)

• This gives (amortized) runtime $O(m + n\log(n))$ for Dijkstra’s algorithm.

*Any sequence of d deleteMin calls takes time at most $O(d \log(n))$. But some of the d may take longer and some may take less time.
Dijkstra is used in practice

- $O(n \log(n) + m)$ is really fast!
- eg, OSPF (Open Shortest Path First), a routing protocol for IP networks, uses Dijkstra.

But there are some things it’s not so good at.
Dijkstra Drawbacks

• Needs non-negative edge weights.
• If the weights change, we need to re-run the whole thing.
  • in OSPF, a vertex broadcasts any changes to the network, and then every vertex re-runs Dijkstra’s algorithm from scratch.
WE STOPPED HERE IN LECTURE

• Bonus slides follow, but material on the Bellman-Ford algorithm is also in slides for lecture 12.

• The slides below are different than those in lecture 12 (in order to maintain internal consistency within lectures), so they might be interesting for a different perspective.
Bellman-Ford algorithm

• Slower than Dijkstra’s algorithm

• Can handle negative edge weights.

• Allows for some flexibility if the weights change.
  • We’ll see what this means later
Drawbacks of Dijkstra:

- Can’t handle negative edge weights
- Need to know the network topology and weights in advance.

Example: in OSPF on the previous slide, if there are any changes to the network, a node broadcasts that change to everybody and everybody re-runs Dijkstra from scratch.

Why negative edge weights?

I often choose to take these long paths to the dish and back for recreation! It costs me negative happiness!

There’s frequently free food over here, this costs me negative deliciousness to walk by it.
Problem
with negative edge weights

• What is the shortest path from Gates to the Union?
• Should still be Gates—Packard—CS161—Union
• But what about • G—P—D—G—P—CS161—Union
• That costs • 1-2-3+1+1+4 = 2.
• And why not
  
Let’s put that aside for a moment

Onwards!
To the Bellman-Ford algorithm!
Bellman-Ford

How far is a node from Gates?

Current edge

x is my best over-estimate for a vertex v. We’ll say d[v] = x

• For v in V:
  • d[v] = ∞
  • d[s] = 0
• For i = 1, .., n-1:
  • For each edge e = (u, v) in E:
    • d[v] ← min( d[v] , d[u] + edgeWeight(u, v))

Start with the same graph, no negative weights.
Bellman-Ford

How far is a node from Gates?

Current edge

\[ x \]

\( x \) is my best over-estimate for a vertex \( v \).
We’ll say \( d[v] = x \)

• For \( v \) in \( V \):
  • \( d[v] = \infty \)
  • \( d[s] = 0 \)

• For \( i = 1,\ldots,n-1 \):
  • For each edge \( e = (u,v) \) in \( E \):
    • \( d[v] \leftarrow \min( d[v], d[u] + \text{edgeWeight}(u,v) ) \)
Bellman-Ford

How far is a node from Gates?

Current edge:

- $x$ is my best over-estimate for a vertex $v$. We'll say $d[v] = x$

- For $v$ in $V$:
  - $d[v] = \infty$
  - $d[s] = 0$

- For $i = 1, \ldots, n-1$:
  - For each edge $e = (u,v)$ in $E$:
    - $d[v] \leftarrow \min(d[v], d[u] + \text{edgeWeight}(u,v))$
Bellman-Ford

How far is a node from Gates?

Current edge

x is my best over-estimate for a vertex v. We’ll say d[v] = x

• For v in V:
  • d[v] = ∞
  • d[s] = 0
• For i = 1,...,n-1:
  • For each edge e = (u,v) in E:
    • d[v] ← min( d[v] , d[u] + edgeWeight(u,v))
• For $v$ in $V$:
  • $d[v] = \infty$
  • $d[s] = 0$
• For $i = 1,..,n-1$:
  • For each edge $e = (u,v)$ in $E$:
    • $d[v] \leftarrow \min( d[v], d[u] + \text{edgeWeight}(u,v))$

How far is a node from Gates?

$x$ is my best over-estimate for a vertex $v$. We’ll say $d[v] = x$
Bellman-Ford

How far is a node from Gates?

- For \( v \) in \( V \):
  - \( d[v] = \infty \)
  - \( d[s] = 0 \)
- For \( i = 1, \ldots, n-1 \):
  - For each edge \( e = (u, v) \) in \( E \):
    - \( d[v] \leftarrow \min( d[v], d[u] + \text{edgeWeight}(u,v) ) \)
Bellman-Ford

How far is a node from Gates?

Current edge

x is my best over-estimate for a vertex v. We’ll say d[v] = x

• For v in V:
  • d[v] = ∞
• d[s] = 0
• For i = 1,..,n-1:
  • For each edge e = (u,v) in E:
    • d[v] ← min( d[v], d[u] + edgeWeight(u,v))
Bellman-Ford

How far is a node from Gates?

x is my best over-estimate for a vertex v. We’ll say d[v] = x

• For v in V:
  • d[v] = ∞
  • d[s] = 0
• For i = 1,..,n-1:
  • For each edge e = (u,v) in E:
    • d[v] ← min( d[v] , d[u] + edgeWeight(u,v))
Bellman-Ford

How far is a node from Gates?

• For $v$ in $V$:
  • $d[v] = \infty$
  • $d[s] = 0$
• For $i = 1,\ldots, n-1$:
  • For each edge $e = (u,v)$ in $E$:
    • $d[v] \leftarrow \min(d[v], d[u] + \text{edgeWeight}(u,v))$

- $x$ is my best over-estimate for a vertex $v$.
  - We’ll say $d[v] = x$
Bellman-Ford

How far is a node from Gates?

Current edge

$x$ is my best over-estimate for a vertex $v$. We’ll say $d[v] = x$

- For $v$ in $V$:
  - $d[v] = \infty$
  - $d[s] = 0$
- For $i = 1,..,n-1$:
  - For each edge $e = (u,v)$ in $E$:
    - $d[v] \leftarrow \min(d[v], d[u] + \text{edgeWeight}(u,v))$
**Bellman-Ford**

How far is a node from Gates?

- Current edge

- $x$ is my best over-estimate for a vertex $v$. We'll say $d[v] = x$

- For $v$ in $V$:
  - $d[v] = \infty$
  - $d[s] = 0$
- For $i = 1,..,n-1$:
  - For each edge $e = (u,v)$ in $E$:
    - $d[v] \leftarrow \min(d[v], d[u] + \text{edgeWeight}(u,v))$
Bellman-Ford

How far is a node from Gates?

Current edge

x is my best over-estimate for a vertex v. We’ll say \( d[v] = x \)

• For \( v \) in \( V \):
  • \( d[v] = \infty \)
  • \( d[s] = 0 \)
• For \( i = 1,..,n-1 \):
  • For each edge \( e = (u,v) \) in \( E \):
    • \( d[v] \leftarrow \min( d[v] , d[u] + \text{edgeWeight}(u,v) ) \)
Bellman-Ford

How far is a node from Gates?

- Current edge

  x is my best over-estimate for a vertex v.
  We’ll say $d[v] = x$

- For $v$ in $V$:
  - $d[v] = \infty$
  - $d[s] = 0$

- For $i = 1,..,n-1$:
  - For each edge $e = (u,v)$ in $E$:
    - $d[v] \leftarrow \min( d[v] , d[u] + \text{edgeWeight}(u,v))$
• For $v$ in $V$:
  • $d[v] = \infty$
  • $d[s] = 0$
• For $i = 1, \ldots, n-1$:
  • For each edge $e = (u,v)$ in $E$:
    • $d[v] \leftarrow \min(d[v], d[u] + \text{edgeWeight}(u,v))$

How far is a node from Gates?

$x$ is my best over-estimate for a vertex $v$. We’ll say $d[v] = x$
Bellman-Ford

How far is a node from Gates?

- Current edge

- $x$ is my best over-estimate for a vertex $v$. We’ll say $d[v] = x$

- For $v$ in $V$:
  - $d[v] = \infty$
  - $d[s] = 0$

- For $i = 1,..,n-1$:
  - For each edge $e = (u,v)$ in $E$:
    - $d[v] \leftarrow \min( d[v], d[u] + \text{edgeWeight}(u,v))$
Bellman-Ford

How far is a node from Gates?

- For \( v \) in \( V \):
  - \( d[v] = \infty \)
  - \( d[s] = 0 \)
- For \( i = 1, \ldots, n-1 \):
  - For each edge \( e = (u, v) \) in \( E \):
    - \( d[v] \leftarrow \min (d[v], d[u] + \text{edgeWeight}(u,v)) \)
Bellman-Ford

How far is a node from Gates?

Current edge

x is my best over-estimate for a vertex v. We’ll say d[v] = x

• For v in V:
  • d[v] = \infty
  • d[s] = 0

• For i = 1,..,n-1:
  • For each edge e = (u,v) in E:
    • d[v] ← \min( d[v] , d[u] + \text{edgeWeight}(u,v))

• For v in V:
  • d[v] = \infty
  • d[s] = 0

• For i = 1,..,n-1:
  • For each edge e = (u,v) in E:
    • d[v] ← \min( d[v] , d[u] + \text{edgeWeight}(u,v))
Bellman-Ford

How far is a node from Gates?

• For \( v \) in \( V \):
  • \( d[v] = \infty \)
  • \( d[s] = 0 \)
• For \( i = 1, \ldots, n-1 \):
  • For each edge \( e = (u,v) \) in \( E \):
    • \( d[v] \leftarrow \min( d[v], d[u] + \text{edgeWeight}(u,v) ) \)
Bellman-Ford

How far is a node from Gates?

Current edge

x is my best over-estimate for a vertex v. We’ll say \( d[v] = x \)

This will keep on running until \( i=4 \), but nothing more will happen.

we say it’s converged.

- For \( v \) in \( V \):
  - \( d[v] = \infty \)
- \( d[s] = 0 \)
- For \( i = 1, \ldots, n-1 \):
  - For each edge \( e = (u,v) \) in \( E \):
    - \( d[v] \leftarrow \min( d[v], d[u] + \text{edgeWeight}(u,v) ) \)
This seems much slower than Dijkstra

• And it is:

Running time $O(mn)$

• However, it’s also more flexible in a few ways.
  • Can handle negative edges
  • If we keep on doing these iterations, then changes in the network will propagate through.

• For $i = 1,..,n-1$:
  • For each edge $e = (u,v)$ in $E$:
    • $d[v] \leftarrow \min(d[v], d[u] + \text{edgeWeight}(u,v))$
But first

• Why does it work as is?

We will show:

• After iteration i, for each v,
  • \( d[v] \) is equal to the shortest path between s and v ...
  • ...with at most i edges.

In particular:

• After iteration \( n-1 \), for each v,
  • \( d[v] \) is equal to the shortest path between s and v ...
  • ...with at most \( n-1 \) edges.

All paths in a graph with \( n \) vertices have at most \( n-1 \) edges.
Proof by induction

• **Inductive Hypothesis:**
  • After iteration $i$, for each $v$, $d[v]$ is equal to the cost of the shortest path between $s$ and $v$ with at most $i$ edges.

• **Base case:**
  • After iteration 0...

• **Inductive step:**
Inductive step

- Suppose the inductive hypothesis holds for \( i \).
- We want to establish it for \( i+1 \).

Say this is the shortest path between \( s \) and \( v \) of with at most \( i+1 \) edges:

\[ S \rightarrow \text{vertex} \rightarrow \ldots \rightarrow u \rightarrow v \]  

Let \( u \) be the vertex right before \( v \) in this path.

- By induction, \( d[u] \) is the cost of a shortest path between \( s \) and \( u \) of \( i \) edges.
- By setup, \( d[u] + w(u,v) \) is the cost of a shortest path between \( s \) and \( v \) of \( i+1 \) edges.
- In the \( i+1 \)'st iteration, when \( (u,v) \) is active, we ensure \( d[v] \leq d[u] + w(u,v) \).
- So \( d[v] \leq \text{cost of shortest path between } s \text{ and } v \text{ with } i+1 \text{ edges} \).
- But \( d[v] = \text{cost of a particular path of at most } i+1 \text{ edges} \geq \text{cost of shortest path} \).
- So \( d[v] = \text{cost of shortest path with at most } i+1 \text{ edges} \).

**Hypothesis:** After iteration \( i \), for each \( v \), \( d[v] \) is equal to the cost of the shortest path between \( s \) and \( v \) with at most \( i \) edges.
Proof by induction

• **Inductive Hypothesis:**
  • After iteration \( i \), for each \( v \), \( d[v] \) is equal to the cost of the shortest path between \( s \) and \( v \) of length at most \( i \) edges.

• **Base case:**
  • After iteration 0...

• **Inductive step:**

• **Conclusion:**
  • After iteration \( n-1 \), for each \( v \), \( d[v] \) is equal to the cost of the shortest path between \( s \) and \( v \) of length at most \( n-1 \) edges.
  • *Aka, \( d[v] = d(s,v) \) for all \( v \).*
Something is wrong

• We never used that there weren’t any negative cycles!!
Proof by induction

• **Inductive Hypothesis:**
  • After iteration i, for each v, d[v] is equal to the cost of the shortest path between s and v of length at most i edges.

• **Base case:**
  • After iteration 0...

• **Inductive step:**

• **Conclusion:**
  • After iteration n-1, for each v, d[v] is equal to the cost of the shortest path between s and v of length at most n-1 edges.
  • Aka, d[v] = d(s,v) for all v.
Some paths have more than n-1 edges.

• So we’ve correctly concluded:
  • After iteration n-1, for each v, d[v] is equal to the cost of the shortest path between s and v of length at most n-1 edges.

• But that’s not what we wanted to show.
This is a problem if there are negative cycles.

- A **negative cycle** is a cycle so that the sum of the edges is negative:
  
  - If there is a **negative cycle** in G, then there are always **shorter paths** of length \( n \) if we can always make a path shorter by going around the cycle.
  
  - We kind of want to ignore this case, though, because “**shortest path**” doesn’t even make sense...

![Graph Diagram]
Suppose there are no negative cycles.

• Then all shortest paths are simple paths.
  • A simple path has no cycles.

• It’s true that all simple paths on n vertices have length at most n-1.

• So then we can make the conclusion that we want.
Proof by induction

• **Inductive Hypothesis:**
  • After iteration i, for each v, d[v] is equal to the cost of the shortest path between s and v of length at most i edges.

• **Base case:**
  • After iteration 0...

• **Inductive step:**

• **Conclusion:**
  • After iteration n-1, for each v, d[v] is equal to the cost of the shortest path between s and v of length at most n-1 edges.
  • Aka, the d[v] = d(s,v).

Suppose there are no negative cycles
Theorem

• The Bellman-Ford algorithm runs in time $O(nm)$ on a graph $G$ with $n$ vertices and $m$ edges.

• If there are no negative cycles in $G$, then the BF algorithm terminates with $d[v] = d(s,v)$.

• Notice, negative weights are okay.

Okay, so what if there are negative cycles?
What does B-F do?

- For \( i = 1, \ldots, n-1 \):
  - For each edge \( e = (u,v) \) in \( E \):
    - \( d[v] \leftarrow \min(d[v], d[u] + \text{edgeWeight}(u,v)) \)
What does B-F do?

- For \( i = 1, \ldots, n-1 \):
  - For each edge \( e = (u,v) \) in \( E \):
    - \( d[v] \leftarrow \min(d[v], d[u] + \text{edgeWeight}(u,v)) \)

\( x \) is my best over-estimate for a vertex \( v \). We’ll say \( d[v] = x \)
What does B-F do?

Current edge

x is my best over-estimate for a vertex v. We’ll say d[v] = x

• For i = 1,..,n-1:
  • For each edge e = (u,v) in E:
    • $d[v] \leftarrow \min(d[v], d[u] + \text{edgeWeight}(u,v))$

i=1
What does B-F do?

- For $i = 1,..,n-1$:
  - For each edge $e = (u,v)$ in $E$:
    - $d[v] \leftarrow \min(d[v] , d[u] + \text{edgeWeight}(u,v))$

Current edge

$x$ is my best over-estimate for a vertex $v$.
We’ll say $d[v] = x$
**What does B-F do?**

- Current edge

\[ x \text{ is my best over-estimate for a vertex } v. \]  
\[ \text{We’ll say } d[v] = x \]

• **For** \( i = 1, \ldots, n-1: \)
  • **For** each edge \( e = (u,v) \) in \( E: \)
    • \( d[v] \leftarrow \min(d[v], d[u] + \text{edgeWeight}(u,v)) \)
What does B-F do?

- For $i = 1, \ldots, n-1$:
  - For each edge $e = (u, v)$ in $E$:
    - $d[v] \leftarrow \min(d[v], d[u] + \text{edgeWeight}(u, v))$

$x$ is my best over-estimate for a vertex $v$. We’ll say $d[v] = x$
What does B-F do?

- For each edge $e = (u,v)$ in $E$:
  - $d[v] \leftarrow \min(d[v], d[u] + \text{edgeWeight}(u,v))$
What does B-F do?

- Current edge

x is my best over-estimate for a vertex v. We’ll say d[v] = x

For i = 1,..,n-1:
  - For each edge e = (u,v) in E:
    - d[v] ← min(d[v], d[u] + edgeWeight(u,v))
What does B-F do?

Current edge

x is my best over-estimate for a vertex v. We’ll say \( d[v] = x \)

For \( i = 1, \ldots, n-1 \):
  * For each edge \( e = (u, v) \) in \( E \):
    * \( d[v] \leftarrow \min(d[v], d[u] + \text{edgeWeight}(u, v)) \)
What does B-F do?

x is my best over-estimate for a vertex v. We’ll say d[v] = x

• For i = 1,..,n-1:
  • For each edge e = (u,v) in E:
    • d[v] ← min(d[v], d[u] + edgeWeight(u,v))
What does B-F do?

- $\text{Current edge}$

  $x$ is my best over-estimate for a vertex $v$. We’ll say $d[v] = x$

• For $i = 1,..,n-1$:
  • For each edge $e = (u,v)$ in $E$:
    • $d[v] \leftarrow \min(d[v], d[u] + \text{edgeWeight}(u,v))$

• And again...

  $i=2$
What does B-F do?

Current edge

x is my best over-estimate for a vertex v. We’ll say d[v] = x

For i = 1,..,n-1:
  • For each edge e = (u,v) in E:
    • d[v] ← min(d[v], d[u] + edgeWeight(u,v))

And again...

• For i = 2:
What does B-F do?

Current edge

x is my best over-estimate for a vertex v. We’ll say d[v] = x

• For i = 1,..,n-1:
  • For each edge e = (u,v) in E:
    • d[v] ← min(d[v] , d[u] + edgeWeight(u,v))

And again...

i=2
What does B-F do?

- Current edge
- $x$ is my best over-estimate for a vertex $v$. We’ll say $d[v] = x$

- For $i = 1, \ldots, n-1$:
  - For each edge $e = (u, v)$ in $E$:
    - $d[v] \leftarrow \min(d[v], d[u] + \text{edgeWeight}(u, v))$

- $i=2$

And again...
What does B-F do?

Current edge

x is my best over-estimate for a vertex v. We’ll say $d[v] = x$

You can see where this is going: this will never converge.

• For $i = 1,..,n-1$:
  • For each edge $e = (u,v)$ in $E$:
    • $d[v] \leftarrow \min(d[v], d[u] + \text{edgeWeight}(u,v))$
What does B-F do?

x is my best over-estimate for a vertex v. We’ll say d[v] = x

You can see where this is going: this will never converge.
After n-1 iterations, we stop and get something like this.

• For i = 1,..,n-1:
  • For each edge e = (u,v) in E:
    • d[v] ← min(d[v], d[u] + edgeWeight(u,v))
How can we tell that this didn’t work?

- If we had converged and the algorithm had worked, if we kept going to $i=n$, **nothing would happen**

- But if we keep going, then **something does happen**.

- **For** $i = 1, .., n-1$:
  - **For** each edge $e = (u,v)$ in $E$:
    - $d[v] \leftarrow \min(d[v], d[u] + \text{edgeWeight}(u,v))$
This suggests:

**Bellman-Ford Algorithm:**

- **For** $v$ in $V$:
  - $d[v] = \infty$
  - $d[s] = 0$

- **For** $i = 1, \ldots, n-1$:
  - **For** each edge $e = (u,v)$ in $E$:
    - $d[v] \leftarrow \min(d[v], d[u] + \text{weight}(u,v))$

- **For** each edge $e = (u,v)$ in $E$:
  - if $d[v] < d[u] + \text{weight}(u,v)$:
    - return **negative cycle**
What have we just learned?

Theorem

- The Bellman-Ford algorithm runs in time $O(nm)$ on a graph $G$ with $n$ vertices and $m$ edges.
- If there are no negative cycles in $G$, then the BF algorithm terminates with $d[v] = d(s,v)$.
- If there are negative cycles in $G$, then the BF algorithm returns negative cycle.
Bellman-Ford is also used in practice.

- eg, Routing Information Protocol (RIP) uses something like Bellman-Ford.
  - Older protocol, not used as much anymore.

- Each router keeps a table of distances to every other router.
- Periodically we do a Bellman-Ford update.
- This means that if there are changes in the network, this will propagate. (maybe slowly...)

<table>
<thead>
<tr>
<th>Destination</th>
<th>Cost to get there</th>
<th>Send to whom?</th>
</tr>
</thead>
<tbody>
<tr>
<td>172.16.1.0</td>
<td>34</td>
<td>172.16.1.1</td>
</tr>
<tr>
<td>10.20.40.1</td>
<td>10</td>
<td>192.168.1.2</td>
</tr>
<tr>
<td>10.155.120.1</td>
<td>9</td>
<td>10.13.50.0</td>
</tr>
</tbody>
</table>
Recap: shortest paths

• BFS can do it in unweighted graphs
• In weighted graphs:
  • **Dijkstra’s algorithm** is real fast but:
    • doesn’t work with negative edge weights
    • is very “centralized”
  • **The Bellman-Ford algorithm** is slower but:
    • works with negative edge weights
    • can be done in a distributed fashion, every vertex using only information from its neighbors.
Mini-topic (if time)

Amortized analysis!

• We mentioned this when we talked about implementing Dijkstra.

*Any sequence of \(d\) deleteMin calls takes time at most \(O(d \log(n))\). But some of the \(d\) may take longer and some may take less time.

• What’s the difference between this notion and expected runtime?
Example

• Incrementing a binary counter $n$ times.

• Say that flipping a bit is costly.
  • Above, we’ve noted the cost in terms of bit-flips.
Example

- Incrementing a binary counter $n$ times.

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

- Say that flipping a bit is costly.
  - Some steps are very expensive.
  - Many are very cheap.

- **Amortized** over all the inputs, it turns out to be pretty cheap.
  - $O(n)$ for all $n$ increments.
This is different from expected runtime.

• The statement is **deterministic**, no randomness here.

• But it is still weaker than **worst-case** runtime.
  • We may need to wait for a while to start making it worth it.
Recap

• BFS can do it in unweighted graphs
• In weighted graphs:
  • Dijkstra’s algorithm
  • The Bellman-Ford algorithm

• One can implement Dijkstra’s algorithm using a fancy data structure (a Fibonacci heap) so that it has good amortized time, $O(m + n\log(n))$.
  • And now we have a slightly better idea what amortized time means.
Next time: **MIDTERM**

**Resources:**

- Some practice exams online
- Book, lecture notes, slides
- Office hours
- You can prepare one 2-sided reference sheet to use during the exam
- Here’s a baby hippo: 

![Baby Hippo Image]