Lecture 12

Weighted Graphs: Dijkstra and Bellman-Ford
Announcements

• Your midterms look great!
  • We’re still grading...

• I’ll be away next week (11/12-16)
  • Special guest lecturer: Prof. Mary Wootters 😊
  • My 11/15 OH are moved to Monday 11/26 at noon
  • I’ll still mostly be available by email
  • I’ll really miss you!
Last week

• Dynamic Programming:
  • Identify optimal sub-structure
  • Reuse solutions to sub-problems

• Weighted graphs:
  • Weight on each edge
  • Cost of path = sum of weights

• Floyd-Warshall algorithm:
  • Solves All Pairs Shortest Path on weighted graphs
  • Runs in $O(n^3)$ time
  • Uses Dynamic Programming

$n = \# \text{ of vertices}$
Today

• Back to *Single Source* Shortest Path...
  • BFS does that!
  • ... on weighted graphs (← BFS doesn’t do that)

• Dijkstra’s algorithm

• Bellman-Ford Algorithm

*We’ll see more Dynamic Programming 😊*
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 $t$!

$\text{CONTRACTION!!}$
Stanford’s weighted graph
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>Main Quad</td>
<td>2</td>
<td>Packard-Main Quad</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-Main Quad-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, lightrail, 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 has a cost which depends on link length, traffic, other costs, etc..

• How should we send packets?
Aside: These are difficult problems

• 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.

But let’s ignore them for now.
BFS – we already solved the Single Source Shortest Paths Problem!

• Set $L_i = []$ for $i=1,...,n$
• $L_0 = \{w\}$, where $w$ is the start node
• For $i = 0, ..., n-1$:
  • For $u$ in $L_i$:
    • For each $v$ which is a neighbor of $u$:
      • If $v$ isn’t yet visited:
        • mark $v$ as visited, and put it in $L_{i+1}$

$L_i$ is the set of nodes we can reach in $i$ steps from $w$

Go through all the nodes in $L_i$ and add their unvisited neighbors to $L_{i+1}$
BFS: the good and the bad...

1. Super high level idea of BFS:
   - Bottom-up Dynamic Programming
   - Optimization - order of sub-problems:
     start from source, then distance 1, distance 2, etc.

2. How can we generalize this idea to weights?
   - Still bottom-up Dynamic Programming
   - Optimization - order of sub-problems:
     start from source, then closest, 2\textsuperscript{nd} closest, etc.
Dijkstra’s algorithm

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

YOINK!

Gates
Packard
Dish
Main Quad
Union
A vertex is done when it’s not on the ground anymore.
Dijkstra intuition

YOINK!
Dijkstra intuition

YOINK!
Dijkstra's intuition
Dijkstra intuition

YOINK!
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

\[ x = d[v] \] is my best over-estimate for \( \text{dist}(\text{Gates}, v) \).

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

Initialize \( d[v] = \infty \) for all non-starting vertices \( v \), and \( d[\text{Gates}] = 0 \).
How far is a node from Gates?

- I’m not sure yet
- I’m sure
- $x = d[v]$ is my best over-estimate for $\text{dist}(\text{Gates},v)$.
- 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
- \( x = d[v] \) is my best \textbf{over-estimate} for \( \text{dist}(\text{Gates,v}) \).
- Current node \( u \)

- Pick the \textbf{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 \textbf{sure}.
**Dijkstra by example**

How far is a node from Gates?

- I’m not sure yet
- I’m sure
- $x = d[v]$ is my best over-estimate for $\text{dist(Gates,v)}$.
- 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
- \(x = d[v]\) is my best over-estimate for \(\text{dist}(\text{Gates}, v)\).
- Current node \(u\)

• Pick the \textbf{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 \textbf{sure}.
• Repeat
Dijkstra by example

How far is a node from Gates?

- I’m not sure yet
- I’m sure
- \( x = d[v] \) is my best over-estimate for \( \text{dist}(\text{Gates}, v) \).
- 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
- $x = d[v]$ is my best over-estimate for $\text{dist}(\text{Gates}, v)$.
- 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
- $x = d[v]$ is my best over-estimate for $\text{dist}(\text{Gates}, v)$.
- 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
- \( x = d[v] \) is my best over-estimate for \( \text{dist}(\text{Gates},v) \).
- 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
- $x = d[v]$ is my best over-estimate for $\text{dist}(\text{Gates}, v)$.
- 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
- \(x = d[v]\) is my best over-estimate for \(\text{dist}(\text{Gates},v)\).
- 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
- \( x = d[v] \) is my best over-estimate for \( \text{dist}(\text{Gates}, v) \).
- 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
- \( x = d[v] \) is my best over-estimate for \( \text{dist}(\text{Gates},v) \).
- 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
- \( x = d[v] \) is my best over-estimate for \( \text{dist}(\text{Gates}, v) \).
- 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’s algorithm

Dijkstra(G,s):

• Set all vertices to not-sure
• d[v] = ∞ for all v in V
• d[s] = 0
• While there are not-sure nodes:
  • Pick the not-sure node u with the smallest estimate d[u].
  • For v in u.neighbors:
    • d[v] ← min( d[v] , d[u] + edgeWeight(u,v))
  • Mark u as sure.
• Now d(s, v) = d[v]

Lots of implementation details left un-explained. We’ll get to that!
As usual

• Does it work?
  • Yes.

• Is it fast?
  • Depends on how you implement it.
Why does this work?

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

  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.**
  • By the time we are sure about $v$, $d[v] = d(s, v)$.
  • $d[v]$ never increases, so after $v$ is sure, $d[v]$ stops changing.
  • All vertices are eventually sure. (Stopping condition in algorithm)
  • So all vertices end up with $d[v] = d(s, v)$.

Next let’s prove the claims!
Claim 1
\[ d[v] \geq d(s,v) \text{ for all } v. \]
Claim 1

$\text{d}[v] \geq \text{d}(s,v)$ for all $v$.

Informally:

- Every time we update $\text{d}[v]$, we have a path in mind:
  
  $$\text{d}[v] \leftarrow \min( \text{d}[v], \text{d}[u] + \text{edgeWeight}(u,v) )$$

  Whatever path we had in mind before

  The shortest path to $u$, and then the edge from $u$ to $v$.

- $\text{d}[v] = \text{length of the path we have in mind} \geq \text{length of shortest path} = \text{d}(s,v)$

Formally:

- We should prove this by induction.
  - (See hidden slide or do it yourself)
Claim 1
\[ d[v] \geq d(s,v) \] 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 \( t \).
  - At step \( t+1 \):
    - Pick \( u \); for each neighbor \( v \):
      - \( d[v] \leftarrow \min( d[v], d[u] + w(u,v) ) \geq d(s,v) \)

By induction, \( d(s,v) \leq d[u] + w(u,v) \) 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, $d[u] = d(s,u)$

• For $s$ (the start vertex):
  • The first vertex marked sure has $d[s] = d(s,s) = 0$.

• For all the other vertices:
  • 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 $v$:
      • $d[v] \leftarrow \min(d[v], d[u] + \text{edgeWeight}(u,v))$
    • Mark $u$ as sure.
  • Repeat

• Want to show that $d[u] = d(s,u)$. 
Intuition
When a vertex \( u \) is marked sure, \( d[\text{u}] = d(s, \text{u}) \)

• The first path that lifts \( u \) off the ground is the shortest one.

But let’s actually prove it.
Claim 2

• Want to show that $u$ is good.

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

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

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

True shortest path.
Claim 2

Want to show that $u$ is good. **BWOC, suppose $u$ isn’t good.**

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”

$z \neq u$, since $u$ is not good.

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.

True shortest path.
**Claim 2**

- Want to show that $u$ is good. BWOC, suppose $u$ isn’t good.

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

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

Algorithm chose $u$ so that $d[u]$ was smallest of the unsure vertices.

So therefore $z$ is **sure**.

True shortest path.
**Claim 2**

- Want to show that \( u \) is good. BWOC, suppose \( u \) isn’t good.
- 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 \( d[v] = d(s, v) \)
  - **紫色** means good
  - **橙色** means not good

So everything is equal!

\( d(s, z') = d[z'] \)

And \( z' \) is good.

**CONTRADICTION!!**
Claim 2

- Want to show that $u$ is good. BWOC, suppose $u$ isn’t good.

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

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

So $u$ is good!

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

- For \(s\) (the starting vertex):
  - The first vertex marked sure has \(d[s] = d(s,s) = 0\).

- For all other vertices:
  - 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 \(v\):
  - \(d[v] \leftarrow \min(d[v], d[u] + \text{edgeWeight}(u,v))\)
- Mark \(u\) as sure.
- Repeat

Then \(u\) is good! aka \(d[u] = d(s,u)\)
Why does this work?

• **Theorem:**
  • Run Dijkstra on $G = (V, E)$ starting from $s$.
  • 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**.
What did we just learn?

• **Dijkstra’s algorithm finds 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!
  • Then people would know how to get places quickly.
As usual

• Does it work?
  • Yes.

• Is it fast?
  • Depends on how you implement it.
Running time?

Dijkstra(G,s):

- Set all vertices to not-sure
- \( d[v] = \infty \) for all \( v \) in \( V \)
- \( d[s] = 0 \)
- While there are not-sure nodes:
  - Pick the not-sure node \( u \) with the smallest estimate \( d[u] \).
  - For \( v \) in \( u \).neighbors:
    - \( d[v] \leftarrow \min( d[v] , d[u] + \text{edgeWeight}(u,v) ) \)
  - Mark \( u \) as sure.
- Now \( \text{dist}(s, v) = d[v] \)

- \( n \) iterations (one per vertex)
- How long does one 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 u with minimum d[u]
  • findMin()
• Can remove that u
  • removeMin(u)
• Can update (decrease) 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] ← min( d[v] , d[u] + edgeWeight(u,v))
  • Mark u as sure.

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 if \( m \) is much smaller than \( n^2 \)
Is a hash table a good idea here?

• Not really:
  
  • Search\( (v) \) is fast (in expectation)

  • But \texttt{findMin()} will still take time \( O(n) \) without more structure.
Heaps support these operations

- $T(\text{findMin})$
- $T(\text{removeMin})$
- $T(\text{updateKey})$

- A **heap** is a tree-based data structure that has the property that every node has a smaller key than its children.

- Not covered in this class – see CS166! (Or CLRS).
- But! We will use them.
Many heap implementations

Nice chart on Wikipedia:

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>find-min</td>
<td>Θ(1)</td>
<td>Θ(1)</td>
<td>Θ(log n)</td>
<td>Θ(1)</td>
<td>Θ(1)</td>
<td>Θ(1)</td>
<td>Θ(1)</td>
<td>Θ(1)</td>
</tr>
<tr>
<td>delete-min</td>
<td>Θ(log n)</td>
<td>Θ(log n)</td>
<td>Θ(log n)</td>
<td>O(log n)[c]</td>
<td>O(log n)[c]</td>
<td>O(log n)</td>
<td>O(log n)[c]</td>
<td>O(log n)</td>
</tr>
<tr>
<td>insert</td>
<td>O(log n)</td>
<td>Θ(log n)</td>
<td>Θ(1)[c]</td>
<td>Θ(1)</td>
<td>Θ(1)</td>
<td>Θ(1)</td>
<td>Θ(1)</td>
<td>Θ(1)</td>
</tr>
<tr>
<td>decrease-key</td>
<td>Θ(log n)</td>
<td>Θ(n)</td>
<td>Θ(log n)</td>
<td>Θ(1)[c]</td>
<td>O(log n)[c][d]</td>
<td>Θ(1)</td>
<td>Θ(1)[c]</td>
<td>Θ(1)</td>
</tr>
<tr>
<td>merge</td>
<td>Θ(n)</td>
<td>Θ(log n)</td>
<td>O(log n)[e]</td>
<td>Θ(1)</td>
<td>Θ(1)</td>
<td>Θ(1)</td>
<td>Θ(1)</td>
<td>Θ(1)</td>
</tr>
</tbody>
</table>
Say we use a Fibonacci Heap

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

- See CS166 for more! (or CLRS)

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

*This means that any sequence of $d$ $\text{removeMin}$ calls takes time at most $O(d\log(n))$. But a few of the $d$ may take longer than $O(\log(n))$ and some may take less time..
In practice

Dijkstra using a Python list to keep track of vertices has quadratic runtime.

Dijkstra using a heap looks a bit more linear (actually $n \log(n)$).

BFS is really fast by comparison! But it doesn’t work on weighted graphs.
Dijkstra is used in practice

- 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.
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.

Examples:

- 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.

Why negative edge weights?
One problem with negative edge weights

• What is the shortest path from Gates to the Union?
• Should it still be Gates—Packard—Main Quad—Union?
• But what about G—P—D—G—P—Main Q.—Union?
• That costs 1-2-3+1+1+4 = 2.

Shortest Paths aren’t well-defined if there are negative cycles!
Let’s put that aside for a moment

Onwards!

To the Bellman-Ford algorithm!

This is Richard Bellman who coined “Dynamic Programming” to get $$
Dijkstra: the good and the bad...

1. Super high level idea of Dijkstra:
   - Bottom-up Dynamic Programming
   - Optimization - order of sub-problems: start from source, then closest, next closest, etc.
   - This optimization doesn’t work with negative edges:
Dijkstra: what’s wrong w/ negative weights?

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

  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**.
  - By the time we are **sure** about \( v \), \( d[v] = d(s,v) \).
  - \( d[v] \) never increases, so after \( v \) is **sure**, \( d[v] \) stops changing.
  - All vertices are eventually **sure**. (Stopping condition in algorithm)
  - So all vertices end up with \( d[v] = d(s,v) \).

Next let’s prove the claims!
Dijkstra: the good and the bad...

1. Super high level idea of Dijkstra:
   - Bottom-up Dynamic Programming
   - Optimization - order of sub-problems:
     start from source, then closest, 2\(^{nd}\) closest, etc.
   - This optimization doesn’t work with negative edges:

2. How can we generalize to \textit{negative} weights?
   - Still Bottom-up Dynamic Programming
   - Skip the optimization :-(

![Graph diagram]
Bellman-Ford algorithm

Bellman-Ford(G,s):

- \(d[v] = \infty\) for all \(v\) in \(V\)
- \(d[s] = 0\)
- For \(i = 0, \ldots, n-1:\)
  - For \(u\) in \(V\):
    - For \(v\) in \(u\).neighbors:
      - \(d[v] \leftarrow \min(d[v], d[u] + \text{edgeWeight}(u,v))\)

While there are not-sure nodes:
- Pick the not-sure node \(u\) with the smallest estimate \(d[u]\).
- For \(v\) in \(u\).neighbors:
  - \(d[v] \leftarrow \min(d[v], d[u] + \text{edgeWeight}(u,v))\)
- Mark \(u\) as sure.
For pedagogical reasons

• We are actually going to change this to be dumber.
• Keep n arrays: \(d^{(0)}, d^{(1)}, \ldots, d^{(n-1)}\)

Bellman-Ford*(G,s):

• \(d^{(0)}[v] = \infty\) for all \(v\) in \(V\)
• \(d^{(0)}[s] = 0\)
• For \(i=0,\ldots,n-1:\)
  • For \(u\) in \(V\):
    • For \(v\) in \(u\).neighbors:
      • \(d^{(i+1)}[v] \leftarrow \min( d^{(i)}[v] , d^{(i)}[u] + \text{edgeWeight}(u,v))\)
• Then \(\text{dist}(s,v) = d^{(n-1)}[v]\)
Bellman-Ford

How far is a node from Gates?

<table>
<thead>
<tr>
<th></th>
<th>Gates</th>
<th>Packard</th>
<th>M.Q.</th>
<th>Union</th>
<th>Dish</th>
</tr>
</thead>
<tbody>
<tr>
<td>d(0)</td>
<td>0</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td>d(1)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>d(2)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>d(3)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>d(4)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

• For i=0,...,n-2:
  • For u in V:
    • For v in u.neighbors:
      • \( d^{(i+1)}[v] \leftarrow \min( d^{(i)}[v], d^{(i)}[u] + \text{edgeWeight}(u,v) ) \)

Start with the same graph, no negative weights.
How far is a node from Gates?

<table>
<thead>
<tr>
<th></th>
<th>Gates</th>
<th>Packard</th>
<th>M.Q.</th>
<th>Union</th>
<th>Dish</th>
</tr>
</thead>
<tbody>
<tr>
<td>(d^{(0)})</td>
<td>0</td>
<td>(\infty)</td>
<td>(\infty)</td>
<td>(\infty)</td>
<td>(\infty)</td>
</tr>
<tr>
<td>(d^{(1)})</td>
<td>0</td>
<td>1</td>
<td>(\infty)</td>
<td>(\infty)</td>
<td>25</td>
</tr>
<tr>
<td>(d^{(2)})</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(d^{(3)})</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(d^{(4)})</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

For \(i=0,...,n-2\):
  - For \(u\) in \(V\):
    - For \(v\) in \(u\text{.neighbors}\):
      - \(d^{(i+1)}[v] \leftarrow \min(d^{(i)}[v], d^{(i)}[u] + \text{edgeWeight}(u,v))\)
Bellman-Ford

How far is a node from Gates?

<table>
<thead>
<tr>
<th></th>
<th>Gates</th>
<th>Packard</th>
<th>M.Q.</th>
<th>Union</th>
<th>Dish</th>
</tr>
</thead>
<tbody>
<tr>
<td>(d^{(0)})</td>
<td>0</td>
<td>(\infty)</td>
<td>(\infty)</td>
<td>(\infty)</td>
<td>(\infty)</td>
</tr>
<tr>
<td>(d^{(1)})</td>
<td>0</td>
<td>1</td>
<td>(\infty)</td>
<td>(\infty)</td>
<td>25</td>
</tr>
<tr>
<td>(d^{(2)})</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>45</td>
<td>23</td>
</tr>
<tr>
<td>(d^{(3)})</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(d^{(4)})</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[
\begin{align*}
&\text{For } i=0, \ldots, n-2: \\
&\quad \text{For } u \text{ in } V: \\
&\quad\quad \text{For } v \text{ in } u.\text{neighbors:} \\
&\quad\quad\quad \text{For } i+1: d^{(i+1)}[v] \leftarrow \min(d^{(i)}[v], d^{(i)}[u] + \text{edgeWeight}(u, v))
\end{align*}
\]
How far is a node from Gates?

<table>
<thead>
<tr>
<th></th>
<th>Gates</th>
<th>Packard</th>
<th>M.Q.</th>
<th>Union</th>
<th>Dish</th>
</tr>
</thead>
<tbody>
<tr>
<td>$d^{(0)}$</td>
<td>0</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
</tr>
<tr>
<td>$d^{(1)}$</td>
<td>0</td>
<td>1</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>25</td>
</tr>
<tr>
<td>$d^{(2)}$</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>45</td>
<td>23</td>
</tr>
<tr>
<td>$d^{(3)}$</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>6</td>
<td>23</td>
</tr>
<tr>
<td>$d^{(4)}$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- For $i=0,\ldots,n-2$:
  - For $u$ in $V$:
    - For $v$ in $u$.neighbors:
      - $d^{(i+1)}[v] \leftarrow \min(d^{(i)}[v], d^{(i)}[u] + \text{edgeWeight}(u,v))$
Bellman-Ford

How far is a node from Gates?

<table>
<thead>
<tr>
<th></th>
<th>Gates</th>
<th>Packard</th>
<th>M.Q.</th>
<th>Union</th>
<th>Dish</th>
</tr>
</thead>
<tbody>
<tr>
<td>$d^{(0)}$</td>
<td>0</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
</tr>
<tr>
<td>$d^{(1)}$</td>
<td>0</td>
<td>1</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>25</td>
</tr>
<tr>
<td>$d^{(2)}$</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>45</td>
<td>23</td>
</tr>
<tr>
<td>$d^{(3)}$</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>6</td>
<td>23</td>
</tr>
<tr>
<td>$d^{(4)}$</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>6</td>
<td>23</td>
</tr>
</tbody>
</table>

For $i=0,...,n-2$:
  - For $u$ in $V$:
    - For $v$ in $u$.neighbors:
      - $d^{(i+1)}[v] \leftarrow \min(d^{(i)}[v], d^{(i)}[u] + \text{edgeWeight}(u,v))$
As usual

• Does it work?
  • Yes
  • Idea to the right.
  • (Base case and inductive step similar to Dijkstra)
  • (See hidden slides for details)

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>d&lt;sup&gt;(0)&lt;/sup&gt;</td>
<td>0</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td>d&lt;sup&gt;(1)&lt;/sup&gt;</td>
<td>0</td>
<td>1</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td>d&lt;sup&gt;(2)&lt;/sup&gt;</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>45</td>
</tr>
<tr>
<td>d&lt;sup&gt;(3)&lt;/sup&gt;</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>d&lt;sup&gt;(4)&lt;/sup&gt;</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>6</td>
</tr>
</tbody>
</table>

Idea: proof by induction.

Inductive Hypothesis:

\[ d^{(i)}[v] \] is equal to the cost of the shortest path between \( s \) and \( v \) with at most \( i \) edges.

Conclusion:

\[ d^{(n-1)}[v] \] is equal to the cost of the shortest path between \( s \) and \( v \).

(Since all simple paths have at most \( n-1 \) edges).
Proof by induction

• **Inductive Hypothesis:**
  • After iteration $i$, for each $v$, $d^{(i)}[v]$ 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:**
Skipped in class

**Inductive step**

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

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

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

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

- **Inductive Hypothesis:**
  - After iteration $i$, for each $v$, $d^{(i)}[v]$ 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 $v$ as long as there are no cycles!
Bellman-Ford:
what happens with negative cycles?

Bellman-Ford*(G,s):

• $d^{(0)}[v] = \infty$ for all $v$ in $V$
• $d^{(0)}[s] = 0$
• For $i=0,...,n-1$:
  • For $u$ in $V$:
    • For $v$ in $u$.neighbors:
      • $d^{(i+1)}[v] \leftarrow \min( d^{(i)}[v] , d^{(i)}[u] + \text{edgeWeight}(u,v))$
• Then $\text{dist}(s,v) = d^{(n-1)}[v]$
**Negative edge weights**

<table>
<thead>
<tr>
<th></th>
<th>Gates</th>
<th>Packard</th>
<th>M.Q.</th>
<th>Union</th>
<th>Dish</th>
</tr>
</thead>
<tbody>
<tr>
<td>$d^{(0)}$</td>
<td>0</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td>$d^{(1)}$</td>
<td>0</td>
<td>1</td>
<td>∞</td>
<td>∞</td>
<td>-3</td>
</tr>
<tr>
<td>$d^{(2)}$</td>
<td>0</td>
<td>-5</td>
<td>2</td>
<td>7</td>
<td>-3</td>
</tr>
<tr>
<td>$d^{(3)}$</td>
<td>-4</td>
<td>-5</td>
<td>-4</td>
<td>6</td>
<td>-3</td>
</tr>
</tbody>
</table>

This is not looking good!

- For $i=0,\ldots,n-2$:
  - For $u$ in $V$:
    - For $v$ in $u$.neighbors:
      - $d^{(i+1)}[v] \leftarrow \min(d^{(i)}[v], d^{(i)}[u] + \text{edgeWeight}(u,v))$
Negative edge weights

But we can tell that it’s not looking good:

Some stuff changed!

- For \( i=0,...,n-1 \):
  - For \( u \) in \( V \):
    - For \( v \) in \( u.\text{neighbors} \):
      - \( d^{(i+1)}[v] \leftarrow \min( d^{(i)}[v] , d^{(i)}[u] + \text{edgeWeight}(u,v) ) \)
Back to the correctness

• Does it work?
  • Yes
  • Idea to the right.
  • (Base case and inductive step similar to Dijkstra)

If there are negative cycles, then non-simple paths matter!

Idea: proof by induction.
Inductive Hypothesis: $d^{(i)}[v]$ is equal to the cost of the shortest path between $s$ and $v$ with at most $i$ edges.

Conclusion: $d^{(n-1)}[v]$ is equal to the cost of the shortest path between $s$ and $v$. (Since all simple paths have at most $n-1$ edges).
How Bellman-Ford deals with negative cycles

• If there are no negative cycles:
  • Everything works as it should.
  • The algorithm stabilizes after n-1 rounds.
  • Note: Negative **edges** are okay!!

• If there are negative cycles:
  • Not everything works as it should...
    • Note: it couldn’t possibly work, since shortest paths aren’t well-defined if there are negative cycles.
  • The d[v] values will keep changing.

• Solution:
  • Go one round more and see if things change.
Bellman-Ford algorithm

Bellman-Ford*(G,s):

• \( d^{(0)}[v] = \infty \) for all \( v \) in \( V \)
• \( d^{(0)}[s] = 0 \)
• For \( i=0,...,n-1 \):
  • For \( u \) in \( V \):
    • For \( v \) in \( u \).neighbors:
      • \( d^{(i+1)}[v] \leftarrow \min( d^{(i)}[v] , d^{(i)}[u] + \text{edgeWeight}(u,v)) \)
  • If \( d^{(n-1)} \neq d^{(n)} \):
    • Return NEGATIVE CYCLE 😞
  • Otherwise, \( \text{dist}(s,v) = d^{(n-1)}[v] \)
As usual

• Does it work?
  • Yes
  • Idea to the right.
  • (Base case and inductive step similar to Dijkstra)
  • (See hidden slides for details)

• Is it fast?
  • Not really...

<table>
<thead>
<tr>
<th></th>
<th>Gates</th>
<th>Packard</th>
<th>M.Q.</th>
<th>Union</th>
<th>Dish</th>
</tr>
</thead>
<tbody>
<tr>
<td>d^{(0)}</td>
<td>0</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td>d^{(1)}</td>
<td>0</td>
<td>1</td>
<td>∞</td>
<td>∞</td>
<td>25</td>
</tr>
<tr>
<td>d^{(2)}</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>45</td>
<td>23</td>
</tr>
<tr>
<td>d^{(3)}</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>6</td>
<td>23</td>
</tr>
<tr>
<td>d^{(4)}</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>6</td>
<td>23</td>
</tr>
</tbody>
</table>

Idea: proof by induction.
Inductive Hypothesis: 
\( d^{(i)}[v] \) is equal to the cost of the shortest path between \( s \) and \( v \) with at most \( i \) edges.

Conclusion: 
\( d^{(n-1)}[v] \) is equal to the cost of the shortest path between \( s \) and \( v \). (Since all simple paths have at most \( n-1 \) edges).
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 $i=0,\ldots,n-1$:
  • For $u$ in $V$:
    • For $v$ in $u$.neighbors:
      • $d^{(i+1)}[v] \leftarrow \min( d^{(i)}[v] , d^{(i)}[u] + \text{edgeWeight}(u,v))$
What have we learned?

- The Bellman-Ford algorithm:
  - Finds shortest paths in weighted graphs with negative edge weights
  - runs in time $O(nm)$ on a graph $G$ with $n$ vertices and $m$ edges.
- If there are no negative cycles in $G$:
  - the BF algorithm terminates with $d^{(n-1)}[v] = d(s,v)$.
- If there are negative cycles in $G$:
  - 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: single-source shortest paths

• **BFS:**
  - (+) $O(n+m)$
  - (-) only unweighted graphs

• **Dijkstra’s algorithm:**
  - (+) weighted graphs
  - (+) $O(n\log(n) + m)$ if you implement it right.
  - (-) no negative edge weights
  - (-) very “centralized” (need to keep track of all the vertices to know which to update).

• **The Bellman-Ford algorithm:**
  - (+) weighted graphs, even with negative weights
  - (+) can be done in a distributed fashion, every vertex using only information from its neighbors.
  - (-) $O(nm)$
Next Time

• Greedy algorithms with Prof. Wootters!

Before next time

• Take a walk and think about Dijkstra and B-F algorithms
Reminder for next week: post-midterm flow chart

• Your grade on the midterms is not so important. (Remember: Midterms 1+2 worth less HW+Final)

• I care about your feelings!

Review
Office hours
Piazza
Classmates

Now that you’re smarter, could you solve it on a better day*, with a little more time?

NO

We have a PROBLEM!

YES

Still OK!

Note:
• “panic” is not on the chart b/c you shouldn’t.
• The red ▼ reads “we have a problem”.
• Most solutions are correct, but some are faster than others. You should aim to have the fastest runtime!

* - for example, if we ask a similar q on final
Mini-topic (bonus slides; not on exam)

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.

• 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.