Lecture 11

Weighted Graphs: Dijkstra and Bellman-Ford

NOTE: We may not get to Bellman-Ford!
We will spend more time on it next time.
Announcements

• The midterm is over!
  • for most of you

• Don’t talk about it just yet – we will tell you when it is ok to discuss the midterm!

• HW5 is out today!
Ed Heroes

• < Your name here >

• Bonus points for the most endorsed students on Ed
Previous two lectures

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

• What if the graphs are weighted?

• Part 1: Dijkstra!
  • This will take most of today’s class

• Part 2: Bellman-Ford!
  • Real quick at the end if we have time!
  • We’ll come back to Bellman-Ford in more detail, so today is just a taste.
YOU ARE HERE
Just the graph
Shortest path 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.
If I pay attention to the weights, I should go to Packard, then CS161, then the union.
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?
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 a shorter path from s to x.
  • But then that gives an even shorter path from s to t!

CONTRADICTION!!
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

• “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.
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 think about whether I should bike to Berkeley...
  • More seriously, the internet.

This is a joke.

But let’s ignore them for now.
Dijkstra’s algorithm

- Finds 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

YOINK!

Gates

Packard

Dish

Union

CS161
Dijkstra intuition
Dijkstra
intuition

YOINK!

Gates

Packard

CS161

Union

Dish
Dijkstra intuition
Dijkstra intuition

This creates a tree!

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 \( d[\text{Gates}] = 0 \).
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 dist(Gates,v).
- Current node u

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

How far is a node from Gates?

- I’m not sure yet
- I’m sure
- Current node u

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

• 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.
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
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
**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 \( \text{dist}(Gates,v) \).
- 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

\[x = \text{over-estimate}\]
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
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 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
- After all nodes are **sure**, say that \( d(\text{Gates, v}) = d[v] \) for all v
Dijkstra’s algorithm

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 \( d(s, v) = d[v] \)

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

See IPython Notebook for code!
As usual

• Does it work?
  • Yes.

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

• **Theorem:**
  • Suppose we 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 $v$ is marked **sure**, $d[v] = d(s,v)$.

• **Claims 1 and 2 imply the theorem.**
  • When $v$ is marked **sure**, $d[v] = d(s,v)$.
  • $d[v] \geq d(s,v)$ and never increases, so after $v$ is **sure**, $d[v]$ stops changing.
  • This implies that at any time after $v$ is marked **sure**, $d[v] = d(s,v)$.
  • All vertices are **sure** at the end, so all vertices end up with $d[v] = d(s,v)$.

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

Next let’s prove the claims!
Claim 1

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

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

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

  \[ = d(s,v) \]

Formally:
- We should prove this by induction.
  
  - (See skipped slide or do it yourself)
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) \text{ for all } v.

• Base case:
  • At step 0, d(s,s) = 0, and \begin{align*} d(s,v) \leq \infty \end{align*}

• 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, \begin{align*} d(s,v) \leq d(s,u) + d(u,v) \leq d[u] + w(u,v) \end{align*}
using induction again for d[u]

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

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

- Let’s prove it!
  - Or at least see a proof outline.
Claim 2
When a vertex \( u \) is marked sure, \( d[u] = d(s,u) \)

- **Inductive Hypothesis:**
  - When we mark the \( t \)’th vertex \( v \) as sure, \( d[v] = \text{dist}(s,v) \).

- **Base case (\( t=1 \)):**
  - The first vertex marked sure is \( s \), and \( d[s] = d(s,s) = 0 \).

- **Inductive step:**
  - Assume by induction that every \( v \) already marked sure has \( d[v] = d(s,v) \).

  - 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 \( d[u] = d(s,u) \).
Claim 2
Inductive step

• Want to show that u is good.
• Consider a true shortest path from s to u:

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

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

True shortest path.
Claim 2
Inductive step

• Want to show that $u$ is good. **BWOC, suppose $u$ isn’t good.**
• Say $z$ is the last good vertex before $u$ (on shortest path to $u$).
• $z'$ 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”

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

It may be that $z = s$.

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

It may be that $z' = u$.

True shortest path.
Claim 2

Inductive step

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

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

$z$ is good

Subpaths of shortest paths are shortest paths.
(We’re also using that the edge weights are non-negative here).
Claim 2
Inductive step

• Want to show that \( u \) is good. BWOC, suppose \( u \) isn’t good.

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

\( z \) is good
Subpaths of shortest paths are shortest paths.
Claim 1

• Since \( u \) is not good, \( d[z] \neq d[u] \).

• So \( d[z] < d[u] \), so \( z \) is **sure**. We chose \( u \) so that \( d[u] \) was smallest of the unsure vertices.
Claim 2

Inductive step

- 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.
- So \( d[z] < d[u] \), so z is sure.

Temporary definition:

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

- means good
- means not good

Subpaths of shortest paths are shortest paths.

Claim 1

But u is not good!

We chose u so that \( d[u] \) was smallest of the unsure vertices.
Claim 2
Inductive step

- Want to show that \( u \) is good. BWOC, suppose \( u \) isn’t good.
- If \( z \) is sure then we’ve already updated \( z' \):
  - \( d[z'] \leq d[z] + w(z, z') \) (def of update)
    - \( = d(s, z) + w(z, z') \) (by induction when \( z \) was added to the sure list it had \( d(s, z) = d[z] \))
    - \( = d(s, z') \) (sub-paths of shortest paths are shortest paths)
    - \( \leq d[z'] \) (Claim 1)
  - By induction when \( z \) was added to the sure list it had \( d(s, z) = d[z] \)
  - So \( d(s, z') = d[z'] \) and so \( z' \) is good.

\[ d[z'] \leftarrow \min\{ d[z'], d[z] + w(z, z') \} \]

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

- means good
- means not good

---

That is, the value of \( d[z] \) when \( z \) was marked sure...
Claim 2
When a vertex u is marked sure, d[u] = d(s,u)

• Inductive Hypothesis:
  • When we mark the t’th vertex v as sure, d[v] = dist(s,v).
• Base case:
  • The first vertex marked sure is s, and d[s] = d(s,s) = 0.
• Inductive step:
  • Suppose that we are about to add u to the sure list.
  • That is, we picked u in the first line here:
    • Assume by induction that every v already marked sure has d[v] = d(s,v).
    • Want to show that d[u] = d(s,u).

Conclusion: Claim 2 holds!
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 have we learned?

• 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 \( d[u] \)
  - \( \text{findMin()} \)
- Can remove that \( u \)
  - \( \text{removeMin}(u) \)
- Can update (decrease) \( d[v] \)
  - \( \text{updateKey}(v,d) \)

Pick the \textbf{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 \textbf{sure}.

Total running time is big-oh of:

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

\[= n (T(\text{findMin}) + T(\text{removeMin})) + m \cdot T(\text{updateKey})\]
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
  \[
  = O(n(T(\text{findMin}) + T(\text{removeMin})) + m \cdot T(\text{updateKey}))
  = O(n \log(n)) + O(m \log(n))
  = O((n + m) \log(n))
  \]

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:
  
  • \texttt{Search(v)} is fast (in expectation)

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

- `findMin`
- `removeMin`
- `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
- But! We will use them.
Many heap implementations

Nice chart on Wikipedia:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Binary\textsuperscript{[7]}</th>
<th>Leftist</th>
<th>Binomial\textsuperscript{[7]}</th>
<th>Fibonacci\textsuperscript{[7][8]}</th>
<th>Pairing\textsuperscript{[9]}</th>
<th>Brodal\textsuperscript{[10][b]}</th>
<th>Rank-pairing\textsuperscript{[12]}</th>
<th>Strict Fibonacci\textsuperscript{[13]}</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))\textsuperscript{[c]}</td>
<td>(O(\log n))\textsuperscript{[c]}</td>
<td>(O(\log n))</td>
<td>(O(\log n))\textsuperscript{[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)\textsuperscript{[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)\textsuperscript{[c]}</td>
<td>(o(\log n))\textsuperscript{[c][d]}</td>
<td>Θ(1)</td>
<td>Θ(1)\textsuperscript{[c]}</td>
<td>Θ(1)</td>
</tr>
<tr>
<td>merge</td>
<td>Θ((n))</td>
<td>Θ(log (n))</td>
<td>(O(\log n))\textsuperscript{[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!
- Running time of Dijkstra
  \[
  = O(n( T(\text{findMin}) + T(\text{removeMin}) ) + m T(\text{updateKey})) \\
  = O(n \log(n) + m) \text{ (amortized time)}
  \]

*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

See IPython Notebook for Lecture 11
The heap is implemented using `heapdict`

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.
  • Can be useful if you want to say that some edges are actively good to take, rather than costly.
  • Can be useful as a building block in other algorithms.

• (+) Allows for some flexibility if the weights change.
  • We’ll see what this means later
Today: *intro* to Bellman-Ford

• We’ll see a definition by example.

• We’ll come back to it next lecture with more rigor.
  • Don’t worry if it goes by quickly today.
  • There are some skipped slides with pseudocode, but we’ll see them again next lecture.

• Basic idea:
  • Instead of picking the \( u \) with the smallest \( d[u] \) to update, just update all of the \( u \)’s simultaneously.
Bellman-Ford algorithm

Bellman-Ford(G,s):

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

Instead of picking $u$ cleverly, just update for all of the $u$’s.

Compare to Dijkstra:

• 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 which we will see next lecture

- We are actually going to change this to be less smart.
- Keep n arrays: $d^{(0)}$, $d^{(1)}$, ..., $d^{(n-1)}$

Bellman-Ford*(G,s):

- $d^{(i)}[v] = \infty$ for all $v$ in $V$, for all $i=0,...,n-1$
- $d^{(0)}[s] = 0$
- For $i=0,...,n-2$:
  - For $u$ in $V$:
    - For $v$ in $u$.neighbors:
      - $d^{(i+1)}[v] \leftarrow \min(d^{(i)}[v], d^{(i+1)}[v], d^{(i)}[u] + \text{edgeWeight}(u,v))$
    - Then $\text{dist}(s,v) = d^{(n-1)}[v]$

Slightly different than the original Bellman-Ford algorithm, but the analysis is basically the same.
How far is a node from Gates?

<table>
<thead>
<tr>
<th></th>
<th>Gates</th>
<th>Packard</th>
<th>CS161</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></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:
      - $d^{(i+1)}[v] \leftarrow \min(d^{(i)}[v], d^{(i+1)}[v], d^{(i)}[u] + \text{edgeWeight}(u,v))$
Bellman-Ford

How far is a node from Gates?

<table>
<thead>
<tr>
<th></th>
<th>Gates</th>
<th>Packard</th>
<th>CS161</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>

Start with the same graph, no negative weights.

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

How far is a node from Gates?

<table>
<thead>
<tr>
<th>d(0)</th>
<th>Gates</th>
<th>Packard</th>
<th>CS161</th>
<th>Union</th>
<th>Dish</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>d(1)</th>
<th>Gates</th>
<th>Packard</th>
<th>CS161</th>
<th>Union</th>
<th>Dish</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>25</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>d(2)</th>
<th>Gates</th>
<th>Packard</th>
<th>CS161</th>
<th>Union</th>
<th>Dish</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>45</td>
<td>23</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>d(3)</th>
<th>Gates</th>
<th>Packard</th>
<th>CS161</th>
<th>Union</th>
<th>Dish</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>d(4)</th>
<th>Gates</th>
<th>Packard</th>
<th>CS161</th>
<th>Union</th>
<th>Dish</th>
</tr>
</thead>
</table>

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

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

How far is a node from Gates?

<table>
<thead>
<tr>
<th></th>
<th>Gates</th>
<th>Packard</th>
<th>CS161</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 $i=0,...,n-2$:
  - For $u$ in $V$:
    - For $v$ in $u$.neighbors:
      - $d^{(i+1)}[v] \leftarrow \min(d^{(i)}[v], d^{(i+1)}[v], d^{(i)}[u] + \text{edgeWeight}(u,v))$

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

How far is a node from Gates?

<table>
<thead>
<tr>
<th></th>
<th>Gates</th>
<th>Packard</th>
<th>CS161</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>

These are the final distances!

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

- Does it work?
  - Yes
  - Idea to the right.
  - (See hidden slides for details)

- Is it fast?
  - Not really...
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 \( i+1 \).

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

```
S -----> u -----> v
```

- 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 \( d^{(i+1)}[v] \leq \text{cost of shortest path between } s \text{ and } v \text{ with } i+1 \text{ edges} \).
- But \( d^{(i+1)}[v] = \text{cost of a particular path of at most } i+1 \text{ edges} \geq \text{cost of shortest path} \).
- So \( d[v] = \text{cost of shortest path with at most } i+1 \text{ edges} \).
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 negative cycles!
Pros and cons of Bellman-Ford

• Running time: \( O(mn) \) running time
  • For each of \( n \) steps we update \( m \) edges
  • Slower than Dijkstra

• However, it’s also more flexible in a few ways.
  • Can handle negative edges
  • If we constantly do these iterations, any changes in the network will eventually propagate through.
Wait a second...

• What is the shortest path from Gates to the Union?
Wait a second...

- What is the shortest path from Gates to the Union?
Negative edge weights?

- What is the shortest path from Gates to the Union?
- Shortest paths aren’t defined if there are negative cycles!
Bellman-Ford and negative edge weights

• B-F works with negative edge weights...as long as there are no negative cycles.
  • A negative cycle is a path with the same start and end vertex whose cost is negative.

• However, B-F can detect negative cycles.
Back to the correctness

- Does it work?
  - Yes
  - Idea to the right.

<table>
<thead>
<tr>
<th></th>
<th>Gates</th>
<th>Packard</th>
<th>CS161</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>

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 simple path between $s$ and $v$. *(Since all simple paths have at most $n-1$ edges).*

If there are negative cycles, then non-simple paths matter!
So the proof breaks for negative cycles.
Negative edge weights

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

This is not looking good!

<table>
<thead>
<tr>
<th></th>
<th>Gates</th>
<th>Packard</th>
<th>CS161</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>-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>

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

For $i=0,\ldots,n-1$:
  - For $u$ in $V$:
    - For $v$ in $u$'s neighbors:
      - $d^{(i+1)}[v] \leftarrow \min(d^{(i)}[v], d^{(i+1)}[v], d^{(i)}[u] + \text{edgeWeight}(u,v))$
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...
    • 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.
    • If so, return NEGATIVE CYCLE 😞
  • (Pseudocode on skipped slide)
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:
      • $d^{(i+1)}[v] \leftarrow \min(d^{(i)}[v], d^{(i+1)}[v], d^{(i)}[u] + \text{edgeWeight}(u,v))$
• If $d^{(n-1)} \neq d^{(n)}$:
  • Return NEGATIVE CYCLE 😞
• Otherwise, $\text{dist}(s,v) = d^{(n-1)}[v]$
Summary
It’s okay if that went by fast, we’ll come back to Bellman-Ford

- 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: 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

• Dynamic Programming!!!

Before next time

• Pre-lecture exercise for Lecture 12
  • Remember the Fibonacci numbers from HW1?
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.

\[
\begin{array}{cccccccccccccccc}
0 & 1 & 10 & 11 & 100 & 101 & 110 & 111 & 1000 & 1001 & 1010 & 1011 & 1100 & 1101 & 1110 & 1111 \\
1 & 2 & 1 & 3 & 1 & 2 & 1 & 4 & 1 & 2 & 1 & 3 & 1 & 2 & 1 \\
\end{array}
\]

• 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.