Lecture 11

Weighted Graphs: Dijkstra and Bellman-Ford
Announcements

• HW5 is out today!
• HW4 due FRIDAY.

• Lost and found from midterm:
  • Pencil case with a charger in it
  • Tote bag with a book in it
  • Email marykw@stanford.edu if either are yours.
Ed Heroes!

### Top hearted

<table>
<thead>
<tr>
<th>Name</th>
<th>Hearts</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shubham Anand Jain</td>
<td>157</td>
</tr>
<tr>
<td>Kevin Long Su</td>
<td>84</td>
</tr>
<tr>
<td>Rishu Garg</td>
<td>73</td>
</tr>
<tr>
<td>Jadon</td>
<td>73</td>
</tr>
<tr>
<td>Monica</td>
<td>68</td>
</tr>
<tr>
<td>Ingrid</td>
<td>51</td>
</tr>
<tr>
<td>Gunnar H</td>
<td>50</td>
</tr>
<tr>
<td>Ruiqi Wang</td>
<td>48</td>
</tr>
<tr>
<td>Zach Wi</td>
<td>46</td>
</tr>
<tr>
<td>Aditi T</td>
<td>45</td>
</tr>
</tbody>
</table>

### Top endorsed

<table>
<thead>
<tr>
<th>Name</th>
<th>Endorsements</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jadon G</td>
<td>79</td>
</tr>
<tr>
<td>Aditi T</td>
<td>36</td>
</tr>
<tr>
<td>Zach W</td>
<td>12</td>
</tr>
<tr>
<td>Thanawan A</td>
<td>11</td>
</tr>
<tr>
<td>Anna M</td>
<td>9</td>
</tr>
<tr>
<td>Giulia S</td>
<td>8</td>
</tr>
<tr>
<td>Maxton H</td>
<td>7</td>
</tr>
<tr>
<td>Rohan B</td>
<td>7</td>
</tr>
<tr>
<td>Claire M</td>
<td>6</td>
</tr>
<tr>
<td>Yasmine A</td>
<td>3</td>
</tr>
</tbody>
</table>
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.
Just the graph
Shortest path from Gates to Old Union?

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

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

Run BFS ...
Shortest path from Gates to Old Union?

**weighted graph**

\[ w(u,v) = \text{weight of edge between } u \text{ and } v. \]

For now, edge weights are non-negative.

If I pay attention to the weights, I should go to Packard, then CS161, then Old Union.
Shortest path problem

- 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 the shortest path between \( u \) and \( v \).

Note: For this lecture all graphs are directed, but to save on notation I’m just going to draw undirected edges.
Shortest paths

Q: What’s the shortest path from Packard to Old Union?

This is the shortest path from Gates to Old Union.

It has cost 6.
Warm-up

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

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

\begin{itemize}
  \item CONTRADICTION!!
\end{itemize}
Single-source shortest-path problem

• What is the shortest path from one vertex (e.g. 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>Old 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>

(The answer doesn’t necessarily need to be 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?
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!
Dijkstra intuition

YOINK!
Dijkstra intuition
Dijkstra intuition

YOINK!
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 \( \text{dist}(\text{Gates},v) \).

#### Initialize

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

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

![Graph diagram with nodes and edges labeled with distances.](image)

- Gates: 0
- CS161: 1
- Packard: \( \infty \)
- Union: 4
- Dish: 22
- 1
- 20
- 25
- 22
- 1
- 4
- \( \infty \)
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 \( v \):
  - \( d[v] = \min(d[v], d[u] + \text{edgeWeight}(u, v)) \)
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.
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] + edgeWeight(u,v))
- Mark u as sure.
- Repeat

Packard has three neighbors. What happens when we update them?
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] + edgeWeight(u,v))
• Mark u as **sure**.
• Repeat

Packard has three neighbors. What happens when we update them?
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
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 \( v \):
  • \( d[v] = \min\left( d[v], d[u] + \text{edgeWeight}(u, v) \right) \)
• 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 \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
- After all nodes are \textbf{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:** Let G be a directed, weighted graph with non-negative edge weights.
- 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] ≥ d(s, v) for all v.

• Inductive hypothesis.
  • After t iterations of Dijkstra, d[v] ≥ d(s, v) for all v.

• Base case:
  • At step 0, d(s, s) = 0, and d(s, v) ≤ ∞

• Inductive step: say hypothesis holds for t.
  • At step t+1:
    • Pick u; for each neighbor v:
      • d[v] ← min( d[v], d[u] + w(u, v) ) ≥ d(s, v)

By induction, d(s, v) ≤ d[v]

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:
    - Want to show that \( d[u] = d(s,u) \).
    - 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

(Assuming edge weights are non-negative!)
Claim 2

Inductive step

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

Recall that we picked $u$ so that $d[u]$ is smallest (out of all not-sure vertices)

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 $u$.
• $z'$ is the vertex after $z$.

Recall that we picked $u$ so that $d[u]$ is smallest (out of all not-sure vertices)

**Temporary definition:**

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

- **Green circle:** means good
- **Red circle:** means not good

“by way of contradiction”

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

$z = s$, since $u$ is not good.

It may be that $z' = u$.

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.

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

Recall that we picked \( u \) so that \( d[u] \) is smallest (out of all not-sure vertices)

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

\( \) means good
\( \) means not good

\( z \) is good Subpaths of shortest paths are shortest paths.
(We’re also using that the edge weights are non-negative).
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.

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

• So $d[z] < d[u]$, so $z$ is \textbf{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.

$\text{Claim 1}$

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

- means good
- means not good

$\text{Recall that we picked } u \text{ so that } d[u] \text{ is smallest (out of all not-sure vertices)}$

$\text{z is good Subpaths of shortest paths are shortest paths.}$

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

- 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.

• If $z$ is sure then we’ve already updated $z'$:
  
  $d[z'] \leq d[z] + w(z, z')$
  
  $= d(s, z) + w(z, z')$
  
  $= d(s, z')$
  
  $\leq d[z']$ Claim 1

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

CONTRADICTION!!

Recall that we picked $u$ so that $d[u]$ is smallest (out of all not-sure vertices)

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, \( d[v] = \text{dist}(s,v) \).

- **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?

\textbf{Dijkstra}(G,s):

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

\begin{itemize}
  \item \(n\) iterations (one per vertex)
  \item How long does one iteration take? \textbf{Depends on how we implement it...}
\end{itemize}
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)`

Just the inner loop:

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

Better than an array if the graph is sparse!
aka if $m$ is much smaller than $n^2$
If 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*)

Running time of Dijkstra

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

\[
= O(n\log(n) + m) \text{ (amortized time)}
\]

We won’t cover heaps in this class! See CS166!
(You should know these supported operations and running times, but nothing else).

Compare:
- Array: \( O(n^2) \)
- RBTree: \( O((n+m)\log n) \)

*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 nlog(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

• Assumes 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.
  • We’ll see formal definitions/pseudocode next time.

• Basic idea:
  • Instead of picking the u with the smallest d[u] to update, just update all of the u’s simultaneously.
Start with the same graph, no negative weights.

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>

• 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>∞</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></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,...,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?

\begin{align*}
\begin{array}{cccccc}
\text{Gates} & \text{Packard} & \text{CS161} & \text{Union} & \text{Dish} \\
\text{d}^{(0)} & 0 & \infty & \infty & \infty & \infty \\
\text{d}^{(1)} & 0 & 1 & \infty & \infty & 25 \\
\text{d}^{(2)} & 0 & 1 & 2 & 45 & 23 \\
\text{d}^{(3)} & & & & & \\
\text{d}^{(4)} & & & & & \\
\end{array}
\end{align*}

Start with the same graph, no negative weights.

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

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

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 $u$'s 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...
  • $O(mn)$

### 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.)
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:**
**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.

- 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:

- 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 negative cycles!*

Skipped in class
Nice things about Bellman-Ford

• Flexible if the weights change
  • Each node continuously updates itself by querying its neighbors, and changes in the network will eventually propagate through.

• Can handle negative edge weights*

*As long as there aren’t negative cycles!
Caution: negative cycles

• What is the shortest path from Gates to Old Union?
Caution: negative cycles

• What is the shortest path from Gates to Old Union?
Caution: negative cycles

- What is the shortest path from Gates to Old Union?
- Shortest paths aren’t defined if there are negative cycles!

Cost: $-\infty$
Bellman-Ford and negative edge weights

• B-F works with negative edge weights...as long as there are not 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.

Figure out how! (Hint: if there are no negative cycles, the algorithm should stop updating after n-1 iterations. What happens if there are negative cycles?)
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, even 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 can be modified to return “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(nlog(n) + m) if you implement it with a Fibonacci heap
  • (-) no negative edge weights
  • (-) very “centralized” (need to keep track of all the vertices to know which to update).

• **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
  • Fibonacci numbers!