Lecture 12

Bellman-Ford, Floyd-Warshall,
and Dynamic Programming!
Announcements

• HW5 due Wednesday!
• HW6 released Wednesday!
• Ed Etiquette:
  • Please mark follow-up questions as “unresolved” so that we notice them!!
• We are almost done grading the midterm – grades will be released soon.
  • Great job everyone!
  • Once grades are released, please follow standard procedure for regrade requests.
Today

• Bellman-Ford Algorithm
• Bellman-Ford is a special case of Dynamic Programming!
• What is dynamic programming?
  • Warm-up example: Fibonacci numbers
• Another example:
  • Floyd-Warshall Algorithm
Recall

• A weighted directed graph:

Weights on edges represent costs.

The cost of a path is the sum of the weights along that path.

A shortest path from s to t is a directed path from s to t with the smallest cost.

The single-source shortest path problem is to find the shortest path from s to v for all v in the graph.

This is a path from s to t of cost 22.

This is a path from s to t of cost 10. It is the shortest path from s to t.
Last time

• Dijkstra’s algorithm!
  • Solves the single-source shortest path problem in weighted graphs.

• Bellman-Ford algorithm!
  • ALSO solves the single-source shortest path problem in weighted graphs.
Bellman-Ford vs. Dijkstra

- Dijkstra:
  - Find the $u$ with the smallest $d[u]$
  - Update $u$’s neighbors: $d[v] = \min( d[v], d[u] + w(u,v) )$

- Bellman-Ford:
  - Don’t bother finding the $u$ with the smallest $d[u]$
    - Everyone updates!
  - Slower, but more flexible:
    - Can handle negative edge weights (as long as there aren’t negative cycles)
    - Can do updates in a decentralized way.
Aside: Negative Cycles

• A **negative cycle** is a cycle whose edge weights sum to a negative number.

• Shortest paths aren’t defined when there are negative cycles!

The shortest path from A to B has cost...negative infinity?
Bellman-Ford vs. Dijkstra

• Dijkstra:
  • Find the u with the smallest d[u]
  • Update u’s neighbors: \( d[v] = \min( d[v], d[u] + w(u,v) ) \)

• Bellman-Ford:
  • Don’t bother finding the u with the smallest d[u]
    • Everyone updates!
  • Slower, but more flexible:
    • Can handle negative edge weights (as long as there aren’t negative cycles)
    • Can do updates in a decentralized way.
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></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 $v$ in $V$: 
    - $d^{(i+1)}[v] \leftarrow \min( d^{(i)}[v], d^{(i)}[u] + w(u,v) )$

where we are also taking the min over all $u$ in $v$.inNeighbors.
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>

- For $i=0,...,n-2$:
  - For $v$ in $V$:
    - $d^{(i+1)}[v] \leftarrow \min( d^{(i)}[v], d^{(i)}[u] + w(u,v) )$
where we are also taking the min over all $u$ in $v$.inNeighbors
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>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,...,n-2:
- For v in V:
  - \(d^{(i+1)}[v] \leftarrow \min(d^{(i)}[v], d^{(i)}[u] + w(u,v))\)
  where we are also taking the min over all u in v.inNeighbors
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 \(v\) in \(V:\)
    - \(d^{(i+1)}[v] \leftarrow \min( d^{(i)}[v] , d^{(i)}[u] + w(u,v) )\)
      where we are also taking the min over all \(u\) in \(v\).inNeighbors
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 $i=0,\ldots,n-2$:
  - For $v$ in $V$:
    - $d^{(i+1)}[v] \leftarrow \min (d^{(i)}[v], d^{(i)}[u] + w(u,v))$
      where we are also taking the min over all $u$ in $v$.inNeighbors
Bellman-Ford* algorithm

**Bellman-Ford**(G,s):

- Initialize arrays \(d^{(0)},...,d^{(n-1)}\) of length \(n\)
- \(d^{(0)}[v] = \infty\) for all \(v\) in \(V\)
- \(d^{(0)}[s] = 0\)
- For \(i=0,...,n-2:\)
  - For \(v\) in \(V\):
    - \(d^{(i+1)}[v] \leftarrow \min( d^{(i)}[v] , \min_{u \in v.inNbrs} \{d^{(i)}[u] + w(u,v)\} )\)
- Now, \(dist(s,v) = d^{(n-1)}[v]\) for all \(v\) in \(V\).
  - (Assuming no negative cycles)

Here, Dijkstra picked a special vertex \(u\) and updated \(u\)'s neighbors – Bellman-Ford will update all the vertices.

\*Slightly different than some versions of Bellman-Ford...but this way is pedagogically convenient for today’s lecture."
Note on implementation

• Don’t actually keep all n arrays around.
• Just keep two at a time: “last round” and “this round”

Gates  Packard  CS161  Union  Dish

\[ \begin{array}{ccccc}
    & 0 & \infty & \infty & \infty & \infty \\
\text{d}^{(0)}
\end{array} \]

\[ \begin{array}{ccccc}
    & 0 & 1 & \infty & \infty & 25 \\
\text{d}^{(1)}
\end{array} \]

\[ \begin{array}{ccccc}
    & 0 & 1 & 2 & 45 & 23 \\
\text{d}^{(2)}
\end{array} \]

\[ \begin{array}{ccccc}
    & 0 & 1 & 2 & 6 & 23 \\
\text{d}^{(3)}
\end{array} \]

\[ \begin{array}{ccccc}
    & 0 & 1 & 2 & 6 & 23 \\
\text{d}^{(4)}
\end{array} \]

Only need these two in order to compute \( d^{(4)} \)
Interpretation of $d^{(i)}$

For all vertices $v$, $d^{(i)}[v]$ is equal to the cost of the shortest path between $s$ and $v$ with at most $i$ edges.
Why does Bellman-Ford work?

• Inductive hypothesis:
  • For all $v$, $d^{(i)}[v]$ is equal to the cost of the shortest path between $s$ and $v$ \textbf{with at most i edges}.

• Conclusion:
  • For all $v$, $d^{(n-1)}[v]$ is equal to the cost of the shortest path between $s$ and $v$ \textbf{with at most n-1 edges}.

Do the base case and inductive step!
Aside: simple paths

Assume there is no negative cycle.

• Then there is a shortest path from $s$ to $t$, and moreover there is a simple shortest path.

A simple path in a graph with $n$ vertices has at most $n-1$ edges in it.

• So there is a shortest path with at most $n-1$ edges.

“Simple” means that the path has no cycles in it.
Why does Bellman-Ford work?

• Inductive hypothesis:
  • For all $v$, $d^{(i)}[v]$ is equal to the cost of the shortest path between $s$ and $v$ with at most $i$ edges.

• Conclusion:
  • For all $v$, $d^{(n-1)}[v]$ is equal to the cost of the shortest path between $s$ and $v$ with at most $n-1$ edges.
  • If there are no negative cycles, $d^{(n-1)}[v]$ is equal to the cost of the shortest path.

Notice that negative edge weights are fine. Just not negative cycles.
Important thing about B-F for the rest of this lecture

For all vertices $v$, $d^{(i)}[v]$ is equal to the cost of the shortest path between $s$ and $v$ with at most $i$ edges.
Bellman-Ford is an example of... *Dynamic Programming!*

Today:

• Example of Dynamic programming:
  • Fibonacci numbers.
  • (And Bellman-Ford)

• What is dynamic programming, exactly?
  • And why is it called “dynamic programming”?

• Another example: Floyd-Warshall algorithm
  • An “all-pairs” shortest path algorithm
Pre-Lecture exercise: How not to compute Fibonacci Numbers

• Definition:
  • $F(n) = F(n-1) + F(n-2)$, with $F(1) = F(2) = 1$.
  • The first several are:
    • 1
    • 1
    • 2
    • 3
    • 5
    • 8
    • 13, 21, 34, 55, 89, 144,…

• Question:
  • Given $n$, what is $F(n)$?
Candidate algorithm

```python
• def Fibonacci(n):
  • if n == 0, return 0
  • if n == 1, return 1
  • return Fibonacci(n-1) + Fibonacci(n-2)
```

Running time?

• $T(n) = T(n-1) + T(n-2) + O(1)$
• $T(n) \geq T(n-1) + T(n-2)$ for $n \geq 2$
• So $T(n)$ grows at least as fast as the Fibonacci numbers themselves...
• This is EXPONENTIALLY QUICKLY!
Why do the Fibonacci numbers grow exponentially quickly?

\[ T(n) = T(n-1) + T(n-2) \]
\[ \geq 2T(n-2) \]

Try unrolling this:

\[ T(n) \geq 2T(n-2) \]
\[ \geq 4T(n-4) \]
\[ \geq 8T(n-6) \]
\[ \ldots \geq 2^j T(n-2j) \text{ for any } j < \frac{n}{2} \]
\[ \ldots \geq 2^{n/2} T(1) \text{ by plugging in } j = \frac{n-1}{2} \]

So \[ T(n) \geq 2^{n/2} \], which is REALLY BIG!!!
What’s going on?
Consider Fib(8)

That’s a lot of repeated computation!
Maybe this would be better:

```python
def fasterFibonacci(n):
    F = [0, 1, None, None, ..., None]
    \ F has length n + 1
    for i = 2, ..., n:
        F[i] = F[i-1] + F[i-2]
    return F[n]
```

Much better running time!
This was an example of...

Dynamic programming!
What is *dynamic programming*?

• It is an algorithm design paradigm
  • like divide-and-conquer is an algorithm design paradigm.
• Usually it is for solving *optimization problems*
  • eg, *shortest* path
  • (Fibonacci numbers aren’t an optimization problem, but they are a good example of DP anyway...)


Elements of dynamic programming

1. Optimal sub-structure:

• Big problems break up into sub-problems.
  • Fibonacci: $F(i)$ for $i \leq n$
  • Bellman-Ford: Shortest paths with at most $i$ edges for $i < n$

• The optimal solution to a problem can be expressed in terms of optimal solutions to smaller sub-problems.
  • Fibonacci:
    \[ F(i+1) = F(i) + F(i-1) \]
  • Bellman-Ford:
    \[ d^{(i+1)}[v] \leftarrow \min \{ d^{(i)}[v], \ \min_u \{ d^{(i)}[u] + \text{weight}(u,v) \} \} \]

*The word “optimal” makes sense in the context of optimization problems like shortest path, and is why this is called “Optimal Sub-structure.”*
Elements of dynamic programming

2. Overlapping sub-problems:

• The sub-problems overlap.
  • Fibonacci:
    • Both $F[i+1]$ and $F[i+2]$ directly use $F[i]$.
    • And lots of different $F[i+x]$ indirectly use $F[i]$.
  • Bellman-Ford:
    • Many different entries of $d^{(i+1)}$ will directly use $d^{(i)}[v]$.
    • And lots of different entries of $d^{(i+x)}$ will indirectly use $d^{(i)}[v]$.

• This means that we can save time by solving a sub-problem just once and storing the answer.
Elements of dynamic programming

• Optimal substructure.
  • Optimal solutions to sub-problems can be used to find the optimal solution of the original problem.

• Overlapping subproblems.
  • The subproblems show up again and again

• Using these properties, we can design a dynamic programming algorithm:
  • Keep a table of solutions to the smaller problems.
  • Use the solutions in the table to solve bigger problems.
  • At the end we can use information we collected along the way to find the solution to the whole thing.
Two ways to **think about and/or implement** DP algorithms

- Top down
- Bottom up
Bottom up approach
what we just saw.

• For Fibonacci:
• Solve the small problems first
  • fill in F[0], F[1]
• Then bigger problems
  • fill in F[2]
• ...
• Then bigger problems
  • fill in F[n-1]
• Then finally solve the real problem.
  • fill in F[n]
Bottom up approach

what we just saw.

• For Bellman-Ford:

• Solve the small problems first
  • fill in $d^{(0)}$

• Then bigger problems
  • fill in $d^{(1)}$

• ...

• Then bigger problems
  • fill in $d^{(n-2)}$

• Then finally solve the real problem.
  • fill in $d^{(n-1)}$
Top down approach

• Think of it like a recursive algorithm.

• To solve the big problem:
  • Recurse to solve smaller problems
    • Those recurse to solve smaller problems
    • etc..

• The difference from divide and conquer:
  • Keep track of what small problems you’ve already solved to prevent re-solving the same problem twice.
  • Aka, “memo-ization”
Example of top-down Fibonacci

- define a global list `F = [0, 1, None, None, ..., None]`

- `def` Fibonacci(n):
  - `if` `F[n]` `!=` `None`:
    - `return` `F[n]`
  - `else`:
    - `F[n]` `=` `Fibonacci(n-1) + Fibonacci(n-2)`
  - `return` `F[n]`

Memo-ization: Keeps track (in `F`) of the stuff you’ve already done.
Memo-ization visualization

Collapse repeated nodes and don’t do the same work twice!
Memo-ization Visualization

ctd

- define a global list \( F = [0,1,\text{None}, \text{None}, ..., \text{None}] \)
- \textbf{def} Fibonacci(n):
  - \textbf{if} \( F[n] \neq \text{None} \):
    - return \( F[n] \)
  - \textbf{else}:
    - \( F[n] = \text{Fibonacci}(n-1) + \text{Fibonacci}(n-2) \)
    - return \( F[n] \)

Collapse repeated nodes and don’t do the same work twice!

But otherwise treat it like the same old recursive algorithm.
What have we learned?

• **Dynamic programming:**
  • Paradigm in algorithm design.
  • Uses *optimal substructure*
  • Uses *overlapping subproblems*
  • Can be implemented *bottom-up* or *top-down*.
  • It’s a fancy name for a pretty common-sense idea:
    
    Don’t duplicate work if you don’t have to!
Why “dynamic programming”?

• Programming refers to finding the optimal “program.”
  • as in, a shortest route is a *plan* aka a *program*.
• Dynamic refers to the fact that it’s multi-stage.
• But also it’s just a fancy-sounding name.
Why “dynamic programming”? 

• Richard Bellman invented the name in the 1950’s. 
• At the time, he was working for the RAND Corporation, which was basically working for the Air Force, and government projects needed flashy names to get funded. 
• From Bellman’s autobiography:
  • “It’s impossible to use the word, dynamic, in the pejorative sense…I thought dynamic programming was a good name. It was something not even a Congressman could object to.”
Floyd-Warshall Algorithm
Another example of DP

• This is an algorithm for **All-Pairs Shortest Paths (APSP)**
  • That is, I want to know the shortest path from u to v for **ALL pairs** u,v of vertices in the graph.
  • Not just from a special single source s.

<table>
<thead>
<tr>
<th>Source</th>
<th>s</th>
<th>u</th>
<th>v</th>
<th>t</th>
</tr>
</thead>
<tbody>
<tr>
<td>s</td>
<td>0</td>
<td>2</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>u</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>v</td>
<td>∞</td>
<td>∞</td>
<td>0</td>
<td>-2</td>
</tr>
<tr>
<td>t</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>0</td>
</tr>
</tbody>
</table>
Floyd-Warshall Algorithm
Another example of DP

• This is an algorithm for **All-Pairs Shortest Paths (APSP)**
  • That is, I want to know the shortest path from u to v for **ALL pairs** u,v of vertices in the graph.
  • Not just from a special single source s.

• Naïve solution (if we want to handle negative edge weights):
  • For all s in G:
    • Run Bellman-Ford on G starting at s.

  • Time $O(n \cdot nm) = O(n^2m)$,
    • may be as bad as $n^4$ if $m=n^2$
Recipe for applying Dynamic Programming

• **Step 1:** Identify **optimal substructure.**
  • What are our subproblems?

• **Step 2:** Find a **recursive formulation** for the subproblems
  • How can we solve larger problems using smaller ones?

• **Step 3:** Use **dynamic programming** to find the thing you want.
  • Fill in a table, starting with the smallest sub-problems and building up.

• *(Steps 4 and 5 coming next lecture!)*
Optimal substructure

Label the vertices 1, 2, ..., n
**Optimal substructure**

**Sub-problem(k-1):**
For all pairs, $u,v$, find the cost of the shortest path from $u$ to $v$, so that all the internal vertices on that path are in $\{1,\ldots,k-1\}$.

Let $D^{(k-1)}[u,v]$ be the solution to Sub-problem(k-1).

Label the vertices 1,2,...,n (We omit some edges in the picture below – meant to be a cartoon, not an example).

This is the shortest path from $u$ to $v$ through the blue set. It has cost $D^{(k-1)}[u,v]$. 
Recipe for applying Dynamic Programming

• **Step 1:** Identify **optimal substructure.**
  • What are our subproblems?

• **Step 2:** Find a **recursive formulation** for the subproblems
  • How can we solve larger problems using smaller ones?

• **Step 3:** Use dynamic programming to find the thing you want.
  • Fill in a table, starting with the smallest sub-problems and building up.
How can we find $D^{(k)}[u,v]$ using $D^{(k-1)}$?

$D^{(k)}[u,v]$ is the cost of the shortest path from $u$ to $v$ so that all internal vertices on that path are in $\{1, \ldots, k\}$. 
How can we find $D^{(k)}[u,v]$ using $D^{(k-1)}$?

$D^{(k)}[u,v]$ is the cost of the shortest path from $u$ to $v$ so that all internal vertices on that path are in $\{1, \ldots, k\}$.

**Case 1**: we don’t need vertex $k$.

$D^{(k)}[u,v] = D^{(k-1)}[u,v]$
How can we find $D^{(k)}[u,v]$ using $D^{(k-1)}$?

$D^{(k)}[u,v]$ is the cost of the shortest path from $u$ to $v$ so that all internal vertices on that path are in \{1, ..., $k$\}.

**Case 2:** we need vertex $k$.
Case 2 continued

• Suppose there are no negative cycles. WLOG the shortest path from $u$ to $v$ through $\{1,\ldots,k\}$ is simple.

• The shortest path from $u$ to $v$ looks like this:
  • **This path** is the shortest path from $u$ to $k$ through $\{1,\ldots,k-1\}$.
    • sub-paths of shortest paths are shortest paths
  • Similarly for **this path**.

\[
D^{(k)}[u,v] = D^{(k-1)}[u,k] + D^{(k-1)}[k,v]
\]
How can we find $D^{(k)}[u,v]$ using $D^{(k-1)}$?

**Case 1:** we don’t need vertex $k$.

$$D^{(k)}[u,v] = D^{(k-1)}[u,v]$$

**Case 2:** we need vertex $k$.

$$D^{(k)}[u,v] = D^{(k-1)}[u,k] + D^{(k-1)}[k,v]$$
How can we find $D^{(k)}[u,v]$ using $D^{(k-1)}$?

- $D^{(k)}[u,v] = \min\{ D^{(k-1)}[u,v], D^{(k-1)}[u,k] + D^{(k-1)}[k,v] \}$

Case 1: Cost of shortest path through $\{1,\ldots,k-1\}$

Case 2: Cost of shortest path from $u$ to $k$ and then from $k$ to $v$ through $\{1,\ldots,k-1\}$

- Optimal substructure:
  - We can solve the big problem using solutions to smaller problems.

- Overlapping sub-problems:
  - $D^{(k-1)}[k,v]$ can be used to help compute $D^{(k)}[u,v]$ for lots of different $u$’s.
How can we find $D^{(k)}[u,v]$ using $D^{(k-1)}$?

- $D^{(k)}[u,v] = \min\{ D^{(k-1)}[u,v], \, D^{(k-1)}[u,k] + D^{(k-1)}[k,v] \}$

  **Case 1**: Cost of shortest path through \{1,...,k-1\}

  **Case 2**: Cost of shortest path from \(u\) to \(k\) and then from \(k\) to \(v\) through \{1,...,k-1\}

- Using our **Dynamic programming** paradigm, this gives us an algorithm!
Recipe for applying Dynamic Programming

• **Step 1:** Identify *optimal substructure.*
  • What are our subproblems?

• **Step 2:** Find a *recursive formulation* for the subproblems
  • How can we solve larger problems using smaller ones?

• **Step 3:** Use *dynamic programming* to find the thing you want.
  • Fill in a table, starting with the smallest sub-problems and building up.
Floyd-Warshall algorithm

- Initialize n-by-n arrays $D^{(k)}$ for $k = 0,\ldots,n$
  - $D^{(0)}[u,v] = \infty$ for all pairs $(u,v)$
  - $D^{(0)}[u,u] = 0$ for all $u$
  - $D^{(0)}[u,v] = \text{weight}(u,v)$ for all $(u,v)$ in $E$.
- For $k = 1, \ldots, n$:
  - For pairs $u,v$ in $V^2$:
    - $D^{(k)}[u,v] = \min\{D^{(k-1)}[u,v], D^{(k-1)}[u,k] + D^{(k-1)}[k,v]\}$
- Return $D^{(n)}$

The base case checks out: the only path through zero other vertices are edges directly from $u$ to $v$.

This is a bottom-up Dynamic programming algorithm.
Our earlier logic shows

• Theorem:

  If there are no negative cycles in a weighted directed graph $G$, then the Floyd-Warshall algorithm, running on $G$, returns a matrix $D^{(n)}$ so that:

  $$D^{(n)}[u,v] = \text{distance between } u \text{ and } v \text{ in } G.$$  

• Running time: $O(n^3)$
  • Better than running Bellman-Ford $n$ times!

• Storage:
  • Need to store two $n$-by-$n$ arrays, and the original graph.

  As with Bellman-Ford, we don’t really need to store all $n$ of the $D^{(k)}$. 

Work out the details of a proof!
What if there *are* negative cycles?

- Just like Bellman-Ford, Floyd-Warshall can detect negative cycles:
  - “Negative cycle” means that there’s some \( v \) so that there is a path from \( v \) to \( v \) that has cost \( < 0 \).
  - Aka, \( D^{(n)}[v,v] < 0 \).

- Algorithm:
  - Run Floyd-Warshall as before.
  - If there is some \( v \) so that \( D^{(n)}[v,v] < 0 \):
    - *return* negative cycle.
What have we learned?

• The Floyd-Warshall algorithm is another example of dynamic programming.

• It computes All Pairs Shortest Paths in a directed weighted graph in time $O(n^3)$. 
Can we do better than $O(n^3)$?
Nothing on this slide is required knowledge for this class

- There is an algorithm that runs in time $O(n^3/\log^{100}(n))$.
  - [Williams, “Faster APSP via Circuit Complexity”, STOC 2014]
- If you can come up with an algorithm for All-Pairs-Shortest-Path that runs in time $O(n^{2.99})$, that would be a really big deal.
  - Let me know if you can!
  - See [Abboud, Vassilevska-Williams, “Popular conjectures imply strong lower bounds for dynamic problems”, FOCS 2014] for some evidence that this is a very difficult problem!
Recipe for applying Dynamic Programming

• **Step 1:** Identify optimal substructure.
• **Step 2:** Find a recursive formulation for the thing you want.
  • E.g., length of shortest paths
• **Step 3:** Use dynamic programming to find the thing you want.
  • Fill in a table, starting with the smallest sub-problems and building up.
• *(Steps 4 and 5 coming next lecture...)*
Recap

• Two shortest-path algorithms:
  • Bellman-Ford for single-source shortest path
  • Floyd-Warshall for all-pairs shortest path

• *Dynamic programming*!
  • This is a fancy name for not repeating work!
Next time

• More examples of *dynamic programming*!

We will stop bullets with our action-packed coding skills, and also maybe find longest common subsequences.

• No pre-lecture exercise for next time: go over your exam instead!