Last week: Breadth-First Search

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

How come BFS finds the shortest path?

There are exponentially many candidates to check...
Lecture 10
Dynamic Programming and Floyd-Warshall!
Last week

- Graphs!
- DFS
  - Topological Sorting
  - Strongly Connected Components
- BFS
  - Shortest Paths from $u$ in unweighted graphs

i.e. all edges were identical
Last week: Breadth-First Search
Exploring the world with a bird’s-eye view

- Not been there yet
- Can reach there in zero steps
- Can reach there in one step
- Can reach there in two steps
- Can reach there in three steps
Last week: Breadth-First Search
Exploring the world with a bird’s-eye view

- Not been there yet
- Can reach there in zero steps
- Can reach there in one step
- Can reach there in two steps
- Can reach there in three steps

start
Last week: Breadth-First Search
Exploring the world with a bird’s-eye view

- Not been there yet
- Can reach there in zero steps
- Can reach there in one step
- Can reach there in two steps
- Can reach there in three steps

start
Last week: Breadth-First Search
Exploring the world with a bird’s-eye view
Last week: Breadth-First Search
Exploring the world with a bird’s-eye view

![Diagram showing Breadth-First Search]

- Not been there yet
- Can reach there in zero steps
- Can reach there in one step
- Can reach there in two steps
- Can reach there in three steps

World: explored!
Last week: Breadth-First Search
Exploring the world with pseudocode

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

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

Go through all the nodes in $L_i$ and add their unvisited neighbors to $L_{i+1}$
Today

• Dynamic programming
• All-pairs shortest path (APSP)
• What if the graphs are weighted?
• Floyd-Warshall algorithm!
Example: Fibonacci Numbers

• Definition:
  • $F(n) = F(n-1) + F(n-2)$, with $F(0) = F(1) = 1$.
  • 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

- **def** Fibonacci(n):
  - **if** n == 0 or 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...
- Fun fact, that’s like \( \phi^n \) where \( \phi = \frac{1+\sqrt{5}}{2} \) is the golden ratio.
- aka, **EXponentially Quickly 😞**

See CLRS Problem 4-4 for a walkthrough of how fast the Fibonacci numbers grow!
What’s going on?
Consider Fib(8)

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

```python
def fasterFibonacci(n):
    • F = [1, 1, None, None, ..., None]
        \ F has length n
    • 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...)

Elements of dynamic programming

1. Optimal sub-structure:

• Big problems break up into sub-problems.
  • Fibonacci: $F(i)$ for $i \leq n$

• The solution to a problem can be expressed in terms of solutions to smaller sub-problems.
  • Fibonacci:

$$F(i+1) = F(i) + F(i-1)$$
Elements of dynamic programming

2. Overlapping sub-problems:

- The sub-problems overlap a lot.
  - Fibonacci:
    - Lots of different $F[j]$ will use $F[i]$.

- 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 are sub-solutions to 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]
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:
  • Memo-ization
  • Keep track of what small problems you’ve already solved to prevent re-solving the same problem twice.
Example of top-down Fibonacci

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

etc
Memo-ization Visualization ctd

- define a global list $F = [1,1,None, None, …, None]$
- **def** Fibonacci(n):
  - **if** $F[n] != None$:
    - **return** $F[n]$
  - **else**:
    - $F[n] = \text{Fibonacci}(n-1) + \text{Fibonacci}(n-2)$
    - **return** $F[n]$
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!
Wait, what about BFS? ... is it also “Dynamic Programming”? 

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

Memoize:
Vertices at distance $i$

Clever DP:
only updates neighbors of $L_i$
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.

Manipulating computer code in an action movie?
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.”
Today

• Dynamic programming
• All-pairs shortest path (APSP)
• What if the graphs are weighted?
• Floyd-Warshall algorithm!
All-Pairs Shortest Path Problem

• **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>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>u</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>v</td>
<td>∞</td>
<td>∞</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>t</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>0</td>
</tr>
</tbody>
</table>
All-Pairs Shortest Path Problem

- **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 \( s \).

Candidate algorithm:

Run BFS from every node!

Complexity: \( O(n^2 + nm) \)

In theory, yes.

There are faster algorithms that use “fast matrix multiplication” (which you saw in Section 1).

Learn faster APSP alg’s in CS367 😊

Can we do better?
All-Pairs Shortest Path Problem

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

Candidate algorithm:
Run BFS from every node!

Complexity: \( O(n^2 + nm) \)

Can we do \textbf{more}?

Can we do better?
Today

- Dynamic programming
- All-pairs shortest path (APSP)
- What if the graphs are *weighted*?
- Floyd-Warshall algorithm!
YOU ARE HERE
Just the graph

How do I get from Gates to the Union?

- Dish
- Hospital
- Gates
- Packard
- STLC
- Union
- Stadium
- Caltrain

Run BES ...
I should go to the dish and then back to the union!

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

How do I get from Gates to the Union?

weighted graph

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

If I pay attention to the weights...

I should go to STLC, 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 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.
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.

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.
Weighted All-Pairs Shortest Path

• **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>
```
Today

• Dynamic programming
• All-pairs shortest path (APSP)
• What if the graphs are weighted?
• Floyd-Warshall algorithm!
Weighted APSP:
How can we generalize BFS to weighted graphs?

1. What are the sub-problems of BFS?

2. How can you re-word your answer to #1, so that it makes sense for weighted graphs?

Think-Pair-Share!
Weighted APSP:
How can we generalize BFS to weighted graphs?

1. What are the sub-problems of BFS?
   - $L(u, v, i)$: “Is distance $(u, v) \leq i$?”
   - Recursion: $L(u, v, i) = \max_{(w, v) \in E} L(u, w, i - 1)$

2. How can you re-word your answer to #1, so that it makes sense for weighted graphs?
   - $D(u, v, k)$: “Shortest path from $u$ to $v$ using $\leq k$ vertices?”
   - Recursion: $D(u, v, k) = \min_{(w, v) \in E} D(u, w, k - 1) + \text{cost}(w, v)$
Is this a good idea?

- $D(u, v, k)$: “Shortest path from $u$ to $v$ using $\leq k$ vertices?”
- Recursion: $D(u, v, k) = \min_{(w, v) \in E} D(u, w, k - 1) + cost(w, v)$

- It’s actually not a bad starting point
  (next week we’ll see it’s called “Bellman-Ford”)
- For $(u, v, k)$, we’re only using $(u, w, k - 1)$ sub-problems
  ...but we also know $(w, v, k - 1)$...
- Enumerating over $\text{deg}(v)$ sub-problems is expensive
- Complexity: $O(n^2 m)$ 😞
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).

Our DP algorithm will fill in the \( n \)-by-\( n \) arrays \( D^{(0)}, D^{(1)}, \ldots, D^{(n)} \) iteratively and then we'll be done.

Label the vertices 1,2,...,n (We omit some edges in the picture below).

This is the shortest path from \( u \) to \( v \) through the blue set. It has length \( D^{(k-1)}[u,v] \).
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).

**Question:** How can we find \(D^{(k)}[u,v]\) using \(D^{(k-1)}\)?

Our DP algorithm will fill in the n-by-n arrays \(D^{(0)}, D^{(1)}, \ldots, D^{(n)}\) iteratively and then we’ll be done.
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 $\{1, \ldots, k\}$. 

.Vertices $1, \ldots, k$

.Vertices $1, \ldots, k-1$
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 $\{1, \ldots, k\}$.

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

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

- If **that path** passes through k, it must look like this:

- **This path** is the shortest path from u to k through \{1,...,k-1\}.
  - sub-paths of shortest paths are shortest paths

- Same 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 $D^{(k-1)}$?

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

  **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 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 $\{1,\ldots,k-1\}$

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

• Using our *Dynamic programming* paradigm, this immediately gives us an algorithm!
Floyd-Warshall algorithm

• Initialize n-by-n arrays $D^{(k)}$ for $k = 0, \ldots, n$
  - $D^{(k)}[u,u] = 0$ for all $u$, for all $k$
  - $D^{(k)}[u,v] = \infty$ for all $u \neq v$, for all $k$
  - $D^{(0)}[u,v] = \text{weight}(u,v)$ 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 $D^{(n)}$

This is a bottom-up **Dynamic programming** algorithm.

The base case checks out: the only path through zero other vertices are edges directly from $u$ to $v$. 
We’ve basically just shown

• 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: $O(n^3)$
  • For dense graphs $(m = \Theta(n^2))$, it’s as good as running BFS from every vertex

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

We don’t really need to store all $n$ of the $D^{(k)}$. 

Work out the details of the proof! (Or see Lecture Notes for a few more details).
What if there are negative cycles?

- Floyd-Warshall can detect negative cycles:
  - Negative cycle $\iff \exists v$ s.t. there is a path from $v$ to $v$ that goes through all $n$ vertices that has cost $< 0$.
  - Negative cycle $\iff \exists v$ s.t. $D^{(n)}[v,v] < 0$.

- 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 $O(n^3)$. 
Another Example of DP?

- Longest simple path (say all edge weights are 1):

What is the longest simple path from s to t?
This is an optimization problem...

- Can we use Dynamic Programming?
- Optimal Substructure?
  - Longest path from s to t = longest path from s to a + longest path from a to t?

NOPE!
This doesn’t give optimal sub-structure

Optimal solutions to subproblems don’t give us an optimal solution to the big problem. (At least if we try to do it this way).

• The subproblems we came up with aren’t independent:
  • Once we’ve chosen the longest path from a to t
    • which uses b,
  • our longest path from s to a shouldn’t be allowed to use b
    • since b was already used.

• Actually, the longest simple path problem is NP-complete.
  • We don’t know of any polynomial-time algorithms for it, DP or otherwise!
Recap

• Floyd-Warshall for weighted all-pairs shortest path

• *Dynamic programming*!
  • This is a fancy name for:
    • Break up an optimization problem into smaller problems
      • The optimal solutions to the sub-problems should be sub-solutions to the original problem.
    • Build the optimal solution iteratively by filling in a table of sub-solutions.
      • Take advantage of overlapping sub-problems!
Next time

• More examples of *dynamic programming*!

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

Before next time

• Go to section!

• No HW this week!!