Taking Stock

Last Time

- Single-Source Shortest Paths

This Time

- All-Pairs Shortest Paths

All-Pairs Shortest Paths

- Given directed graph $G = (V, E)$, $w : E \to \mathbb{R}^{|E|}$. (To ease notation, we let $V = \{1, 2, \ldots, n\}$.)
- Goal: Create an $n \times n$ matrix of shortest path distances $\delta(i, j)$
- We could run Bellman-Ford if negative weights edges
  - Running Time: $O(|V|^2|E|)$.
- We could run Dijkstra if no negative weight edges
  - Running Time: $O(|V|^3 \log |V|)$ (with binary heap implementation)
- We'll see how to do slightly better, by exploiting an analogy to matrix multiplication

New Graph Data Structure

- This is maybe the one and only time we are going to use an adjacency matrix graph representation.
- Given $G = (V, E)$ and weight function $w : E \to \mathbb{R}^{|E|}$, create $|V| \times |V|$ matrix $W$ as

$$w_{ij} = \begin{cases} 0 & i = j \\ w(i, j) & (i, j) \in E \\ \infty & (i, j) \notin E \end{cases}$$

- In this case it is useful to consider having 0 weight “loops” on the nodes ($w_{ii} = 0$)
- The output of an all pairs shortest path algorithm is a matrix $D = (d)_{ij}$, where $d_{ij} = \delta(i, j)$
**Dynamic Programming: Attempt #1**

- Subpaths of shortest paths are shortest paths
- Let $\ell_{ij}^{(m)}$ be the shortest path from $i \in V$ to $j \in V$ that uses $\leq m$ edges
- To initialize
  
  $\ell_{ij}^{(0)} = \begin{cases} 0 & \text{if } i = j \\ \infty & \text{if } i \neq j \end{cases}$

- What is the recursion we are looking for?

  $\ell_{ij}^{(m)} = \min \left( \ell_{ij}^{(n-1)}, \min_{1 \leq k \leq n} (\ell_{ik}^{(m-1)} + w_{kj}) \right)$

  (Since $w_{jj} = 0$)

**More Facts About Our DP**

- Note that $m = 1 \Rightarrow \ell_{ij}^{(1)} = w_{ij}$
- All simple shortest paths contain $\leq n - 1$ edges, so simply compute $\ell_{ij}^{n-1} = \delta(i, j)$
- We will keep a “label-matrix” $L^{(m)}$ which in the end will be $L^{(n-1)} = D$
- Initialize with $L^{(1)} = W$ by definition

**Incrementing $m$**

**EXTEND($L, W$)**

1. create $(n \times n)$ matrix $L'$
2. for $i \leftarrow 1$ to $n$
3. do for $j \leftarrow 1$ to $n$
4. do $\ell_{ij}' \leftarrow \infty$
5. for $k \leftarrow 1$ to $n$
6. do $\ell_{ij}' \leftarrow \min(\ell_{ij}', \ell_{ik} + w_{kj})$

**APSP$1(W)$**

1. $L^{(1)} = W$
2. for $m \leftarrow 2$ to $n - 1$
3. do $L^{(m)} = \text{EXTEND}(L^{(m-1)}, W)$
4. return $L^{(n-1)}$

**Let’s Compare**

**EXTEND($L, W$)**

1. create $(n \times n)$ matrix $L'$
2. for $i \leftarrow 1$ to $n$
3. do for $j \leftarrow 1$ to $n$
4. do $\ell_{ij}' \leftarrow \infty$
5. for $k \leftarrow 1$ to $n$
6. do $\ell_{ij}' \leftarrow \min(\ell_{ij}', \ell_{ik} + w_{kj})$

**MATRIXMUltiPLY($A, B$)**

1. create $(n \times n)$ matrix $C$
2. for $i \leftarrow 1$ to $n$
3. do for $j \leftarrow 1$ to $n$
4. do $c_{ij} \leftarrow 0$
5. for $k \leftarrow 1$ to $n$
6. do $c_{ij} \leftarrow c_{ij} + a_{ik}b_{kj}$
7
Observation!

Who Cares!?!?

- So what if `Extend` looks like `Matrix Multiply`?

**Key Insight**

We Only Care about computing $L^{(n-1)}$

- Suppose we wanted to compute the matrix $AAAAAAAA = A^8$
- Long way: 7 matrix multiplies
- Short Way: 3 matrix multiplies
  - $A, A^2, A^4 = A^2 A^2, A^8 = A^4 A^4$

Faster All-Pairs-Shortest-Paths

**Floyd-Warshall Algorithm**

- Again, a DP approach, but uses a different label definition.
- **Def:** For a path $(v_1, v_2, \ldots, v_k)$, an **intermediate vertex** is any vertex of $p$ other than $v_1$ and $v_k$.
- **Floyd-Warshall Labels:** Let $d_{ij}^{(k)}$ be the shortest path from $i$ to $j$ such that all intermediate vertices are in the set $\{1, 2, \ldots, k\}$.

---

### Observations

<table>
<thead>
<tr>
<th>Extend</th>
<th>MatrixMultiply</th>
</tr>
</thead>
<tbody>
<tr>
<td>$L$</td>
<td>$A$</td>
</tr>
<tr>
<td>$W$</td>
<td>$B$</td>
</tr>
<tr>
<td>$L'$</td>
<td>$C$</td>
</tr>
<tr>
<td>min</td>
<td>+</td>
</tr>
<tr>
<td>+</td>
<td>$\times$</td>
</tr>
<tr>
<td>$\infty$</td>
<td>0</td>
</tr>
</tbody>
</table>

### Faster All-Pairs-Shortest-Paths

**Floyd-Warshall Algorithm**

1. $L^{(1)} = W$
2. $m \leftarrow 1$
3. while $m \leq n - 1$
4. do $L^{(2m)} = \text{Extend}(L^m, L^m)$
5. $m \leftarrow 2m$
6. return $L^m$

- OK to “overshoot” $n - 1$, since shortest path labels don’t change after $m = n - 1$ (since no negative cycles)
- “Repeated squaring” is a technique used to improve the efficiency of lots of other algorithms
- **Analysis:**
Another DP Recursion

- Consider a shortest path \( P \) from \( i \) to \( j \) such that all intermediate vertices are in \( \{1, 2, \ldots, k\} \).

There are two cases

1. \( k \) is not an intermediate vertex. Then all intermediate vertices of \( P \) are in \( \{1, 2, \ldots, k - 1\} \).
2. \( k \) is an intermediate vertex. Then for the paths \( P_{ik} \) and \( P_{kj} \), all intermediate vertices are in \( \{1, 2, \ldots, k - 1\} \).

Building the Algorithm

- This simple observation, immediately suggests a DP recursion

\[
d^{(k)}_{ij} = \begin{cases} 
  w_{ij} & k = 0 \\
  \min(d^{(k-1)}_{ij}, d^{(k-1)}_{ik} + d^{(k-1)}_{kj}) & k \geq 1 
\end{cases}
\]

- We look for \( D^{(n)} = (d^{(n)}_{ij}) \)

Floyd-Warshall (\( W \))

1. \( D^{(0)} = W \)
2. for \( k \leftarrow 1 \) to \( n \)
3. do for \( i \leftarrow 1 \) to \( n \)
4. \hspace{1em} do for \( j \leftarrow 1 \) to \( n \)
5. \hspace{2em} do \( d^{(k)}_{ij} \leftarrow \min(d^{(k-1)}_{ij}, d^{(k-1)}_{ik} + d^{(k-1)}_{kj}) \)
6. return \( D^{(n)} \)

- You don’t really need the superscripts (25.2.4)

Transitive Closure

- Given directed graph \( G = (V, E) \).
- Compute graph \( TC(G) = (V, E^*) \) such that \( e = (i, j) \in E^* \iff \exists \text{ path from } i \text{ to } j \text{ in } G \)

- Transitive closure can be thought of as establishing a data structure that makes it possible to solve reachability questions (can I get to \( x \) from \( y \)?) efficiently. After the preprocessing of constructing the transitive closure, all reachability queries can be answered in constant time by simply reporting a matrix entry.

- Transitive closure is fundamental in propagating the consequences of modified attributes of a graph \( G \).

Applications of Transitive Closure

- Consider the graph underlying any spreadsheet model, where the vertices are cells and there is an edge from cell \( i \) to cell \( j \) if the result of cell \( j \) depends on cell \( i \). When the value of a given cell is modified, the values of all reachable cells must also be updated. The identity of these cells is revealed by the transitive closure of \( G \).
- Many database problems reduce to computing transitive closures, for analogous reasons.
- Doing it fast is important
Transitive Closure Algorithms

1. Perform BFS or DFS from each vertex and keep track of the vertices encountered: $O(V(V + E))$. (Good for sparse graphs)

2. Find Strongly Connected Components. (All vertices in each component are mutually reachable). Do BFS or DFS on component graph. (In which component $A$ is connected to component $B$ if there exists an edge from a vertex in $A$ to a vertex in $B$)

3. You can use Warshall’s Algorithm with weights 1. (In fact you can use “bits” and make things very efficient as well)

Next Time

- Flows in Networks
- Continuation of TSP lab
- Quiz: April 4
- Programming Quiz: April 23