Outline

- All Pairs Shortest Paths
- TSP Approximation Algorithm

All-Pairs Shortest Paths

- For the single-source shortest paths problem, we wanted to find the shortest path from a source vertex $s$ to all the other vertices in the graph.
- We will now generalize this problem further to that of finding the shortest path from every possible source to every possible destination.
- In particular, for every pair of vertices $u$ and $v$, we need to compute the following information:
  - $\text{dist}(u, v)$ is the length of the shortest path (if any) from $u$ to $v$.
  - $\text{pred}(u, v)$ is the second-to-last vertex (if any) on the shortest path (if any) from $u$ to $v$.

Example

- For any vertex $v$, we have $\text{dist}(v, v) = 0$ and $\text{pred}(v, v) = \text{NULL}$.
- If the shortest path from $u$ to $v$ is only one edge long, then $\text{dist}(u, v) = w(u \rightarrow v)$ and $\text{pred}(u, v) = u$.
- If there’s no shortest path from $u$ to $v$, then $\text{dist}(u, v) = \infty$ and $\text{pred}(u, v) = \text{NULL}$.
The output of our shortest path algorithm will be a pair of $|V| \times |V|$ arrays encoding all $|V|^2$ distances and predecessors. Many maps contain such a distance matrix - to find the distance from (say) Albuquerque to (say) Ruidoso, you look in the row labeled “Albuquerque” and the column labeled “Ruidoso.” In this class, we’ll focus only on computing the distance array. The predecessor array, from which you would compute the actual shortest paths, can be computed with only minor additions to the algorithms presented here.

Most obvious solution to APSP is to just run SSSP algorithm $|V|$ times, once for every possible source vertex. Specifically, to fill in the subarray $\text{dist}(s, \ast)$, we invoke either Dijkstra’s or Bellman-Ford starting at the source vertex $s$. We’ll call this algorithm ObviousAPSP.

```
ObviousAPSP(V, E, w) {
    for every vertex $s$ {
        $\text{dist}(s, \ast) = \text{SSSP}(V, E, w, s)$;
    }
}
```

The running time of this algorithm depends on which SSSP algorithm we use. If we use Bellman-Ford, the overall running time is $O(|V|^2 |E|) = O(|V|^4)$. If all the edge weights are positive, we can use Dijkstra’s instead, which decreases the run time to $\Theta(|V||E| + |V|^2 \log |V|) = O(|V|^3)$.
Problem

• We’d like to have an algorithm which takes \( O(|V|^3) \) but which can also handle negative edge weights
• We’ll see that a dynamic programming algorithm, the Floyd Warshall algorithm, will achieve this
• Note: the book discusses another algorithm, Johnson’s algorithm, which is asymptotically better than Floyd Warshall on sparse graphs. However we will not be discussing this algorithm in class.

Dynamic Programming

• Recall: Dynamic Programming = Recursion + Memorization
• Thus we first need to come up with a recursive formulation of the problem
• We might recursively define \( \text{dist}(u, v) \) as follows:

\[
\text{dist}(u, v) = \begin{cases} 
0 & \text{if } u = v \\
\min_x (\text{dist}(u, x) + w(x \rightarrow v)) & \text{otherwise}
\end{cases}
\]

The problem

• In other words, to find the shortest path from \( u \) to \( v \), try all possible predecessors \( x \), compute the shortest path from \( u \) to \( x \) and then add the last edge \( u \rightarrow v \)
• Unfortunately, this recurrence doesn’t work
• To compute \( \text{dist}(u, v) \), we first must compute \( \text{dist}(u, x) \) for every other vertex \( x \), but to compute any \( \text{dist}(u, x) \), we first need to compute \( \text{dist}(u, v) \)
• We’re stuck in an infinite loop!

The solution

• To avoid this circular dependency, we need some additional parameter that decreases at each recursion and eventually reaches zero at the base case
• One possibility is to include the number of edges in the shortest path as this third magic parameter
• So define \( \text{dist}(u, v, k) \) to be the length of the shortest path from \( u \) to \( v \) that uses at most \( k \) edges
• Since we know that the shortest path between any two vertices uses at most \( |V| - 1 \) edges, what we want to compute is \( \text{dist}(u, v, |V| - 1) \)
The Recurrence

\[ dist(u, v, k) = \begin{cases} 
0 & \text{if } u = v \\
\infty & \text{if } k = 0 \text{ and } u \neq v \\
\min_x \left( dist(u, x, k-1) + w(x \rightarrow v) \right) & \text{otherwise} 
\end{cases} \]

The Algorithm

- It’s not hard to turn this recurrence into a dynamic programming algorithm
- Even before we write down the algorithm, though, we can tell that its running time will be \( \Theta(|V|^4) \)
- This is just because the recurrence has four variables — \( u, v, k \) and \( x \) — each of which can take on \( |V| \) different values
- Except for the base cases, the algorithm will just be four nested “for” loops

DP-APSP

\begin{align*}
\text{DP-APSP}(V, E, w) \{ \\
& \text{for all vertices } u \text{ in } V \{ \\
& \quad \text{for all vertices } v \text{ in } V \{ \\
& \quad \quad \text{if } (u=v) \\
& \quad \quad \quad \quad \text{dist}(u, v, 0) = 0; \\
& \quad \quad \quad \quad \text{else} \\
& \quad \quad \quad \quad \quad \quad \quad \text{dist}(u, v, 0) = \infty; \\
& \quad \quad \} \\
& \quad \text{for } k=1 \text{ to } |V|-1 \{ \\
& \quad \quad \text{for all vertices } u \text{ in } V \{ \\
& \quad \quad \quad \text{for all vertices } v \text{ in } V \{ \\
& \quad \quad \quad \quad \text{dist}(u, v, k) = \infty; \\
& \quad \quad \quad \quad \text{for all vertices } x \text{ in } V \{ \\
& \quad \quad \quad \quad \quad \text{if } (\text{dist}(u, v, k) > \text{dist}(u, x, k-1) + w(x, v)) \\
& \quad \quad \quad \quad \quad \quad \quad \text{dist}(u, v, k) = \text{dist}(u, x, k-1) + w(x, v); \\
& \quad \quad \} \\
& \quad \quad \} \\
& \quad \} \\
& \} \\
\end{align*}

The Problem

- This algorithm still takes \( O(|V|^4) \) which is no better than the ObviousAPSP algorithm
- If we use a certain divide and conquer technique, there is a way to get this down to \( O(|V|^3 \log |V|) \) (think about how you might do this)
- However, to get down to \( O(|V|^3) \) run time, we need to use a different third parameter in the recurrence
• Number the vertices arbitrarily from 1 to $|V|$
• Define $dist(u, v, r)$ to be the shortest path from $u$ to $v$ where all intermediate vertices (if any) are numbered $r$ or less
• If $r = 0$, we can’t use any intermediate vertices so shortest path from $u$ to $v$ is just the weight of the edge (if any) between $u$ and $v$
• If $r > 0$, then either the shortest legal path from $u$ to $v$ goes through vertex $r$ or it doesn’t
• We need to compute the shortest path distance from $u$ to $v$ with no restrictions, which is just $dist(u, v, |V|)$

The recurrence

We get the following recurrence:

$$dist(u, v, r) = \begin{cases} w(u \rightarrow v) & \text{if } r = 0 \\ \min\{dist(u, v, r - 1), \quad \text{if } r > 0 \\ dist(u, r, r - 1) + dist(r, v, r - 1) \} & \text{otherwise} \end{cases}$$

The Algorithm

```
FloydWarshall(V,E,w){
    for u=1 to |V|{
        for v=1 to |V|{
            dist(u,v,0) = w(u,v);
        }
    }
    for r=1 to |V|{
        for u=1 to |V|{
            for v=1 to |V|{
                if (dist(u,v,r-1) < dist(u,r,r-1) + dist(r,v,r-1))
                    dist(u,v,r) = dist(u,v,r-1);
                else
                    dist(u,v,r) = dist(u,r,r-1) + dist(r,v,r-1);
            }
        }
    }
}
```

Analysis

• There are three variables here, each of which takes on $|V|$ possible values
• Thus the run time is $\Theta(|V|^3)$
• Space required is also $\Theta(|V|^3)$
**Take Away**

• Floyd-Warshall solves the APSP problem in $\Theta(|V|^3)$ time even with negative edge weights
• Floyd-Warshall uses dynamic programming to compute APSP
• We’ve seen that sometimes for a dynamic program, we need to introduce an *extra variable* to break dependencies in the recurrence.
• We’ve also seen that the choice of this extra variable can have a big impact on the run time of the dynamic program

**TSP**

• A version of the TSP problem is: “Given a weighted graph $G$, what is the shortest Hamiltonian Cycle of $G$?”
• Where a Hamiltonian Cycle is a path that visits each node in $G$ exactly once and returns to the starting node
• This TSP problem is NP-Hard by a reduction from Hamiltonian Cycle
• However, there is a 2-approximation algorithm for this problem if the edge weights obey the *triangle inequality*

**Triangle Inequality**

• In many practical problems, it’s reasonable to make the assumption that the weights, $c$, of the edges obey the *triangle inequality*
• The triangle inequality says that for all vertices $u, v, w \in V$:
  $$c(u, w) \leq c(u, v) + c(v, w)$$
• In other words, the cheapest way to get from $u$ to $w$ is always to just take the edge $(u, w)$
• In the real world, this is often a pretty natural assumption. For example it holds if the vertices are points in a plane and the cost of traveling between two vertices is just the euclidean distance between them.

**Approximation Algorithm**

• Given a weighted graph $G$, the algorithm first computes a MST for $G$, $T$, and then arbitrarily selects a root node $r$ of $T$.
• It then lets $L$ be the list of the vertices visited in a depth first traversal of $T$ starting at $r$.
• Finally, it returns the Hamiltonian Cycle, $H$, that visits the vertices in the order $L$. 

Approximation Algorithm

Approx-TSP(G) {
    T = MST(G);
    L = the list of vertices visited in a depth first traversal
        of T, starting at some arbitrary node in T;
    H = the Hamiltonian Cycle that visits the vertices in the
        order L;
    return H;
}

Example Run

The top left figure shows the graph G (edge weights are just
the Euclidean distances between vertices); the top right figure
shows the MST T. The bottom left figure shows the depth
first walk on T, W = (a, b, c, b, h, b, a, d, e, f, e, g, e, d, a); the bottom
right figure shows the Hamiltonian cycle H obtained by deleting
repeat visits from W, H = (a, b, c, h, d, e, f, g).

Analysis

• The first step of the algorithm takes $O(|E| + |V| \log |V|)$ (if
  we use Prim’s algorithm)
• The second step is $O(|V|)$
• The third step is $O(|V|)$.
• Hence the run time of the entire algorithm is polynomial

An important fact about this algorithm is that: the cost of the
MST is less than the cost of the shortest Hamiltonian cycle.

• To see this, let T be the MST and let $H^*$ be the shortest
  Hamiltonian cycle.
• Note that if we remove one edge from $H^*$, we have a span-
  ning tree, $T'$
• Finally, note that $w(H^*) \geq w(T') \geq w(T)$
• Hence $w(H^*) \geq w(T)$
Now let $W$ be a depth first walk of $T$ which traverses each edge exactly twice (similar to what you did in the hw).

- In our example, $W = (a, b, c, b, h, b, a, d, e, f, e, g, e, d, a)$
- Note that $c(W) = 2c(T)$
- This implies that $c(W) \leq 2c(H^*)$

Unfortunately, $W$ is not a Hamiltonian cycle since it visits some vertices more than once.

- However, we can delete a visit to any vertex and the cost will not increase because of the triangle inequality. (The path without an intermediate vertex can only be shorter)
- By repeatedly applying this operation, we can remove from $W$ all but the first visit to each vertex, without increasing the cost of $W$.
- In our example, this will give us the ordering $H = (a, b, c, h, d, e, f, g)$

By the last slide, $c(H) \leq c(W)$.

- So $c(H) \leq c(W) = 2c(T) \leq 2c(H^*)$
- Thus, $c(H) \leq 2c(H^*)$
- In other words, the Hamiltonian cycle found by the algorithm has cost no more than twice the shortest Hamiltonian cycle.

Many real-world problems can be shown to not have an efficient solution unless $P = NP$ (these are the NP-Hard problems).

- However, if a problem is shown to be NP-Hard, all hope is not lost!
- In many cases, we can come up with an provably good approximation algorithm for the NP-Hard problem.