CS 362, All Pairs Shortest Paths

Jared Saia
University of New Mexico
Outline

- All Pairs Shortest Paths Problem
- Floyd Warshall 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:
  – $dist(u, v)$ is the length of the shortest path (if any) from $u$ to $v$
  – $pred(u, v)$ 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 $dist(v, v) = 0$ and $pred(v, v) = NULL$
• If the shortest path from $u$ to $v$ is only one edge long, then $dist(u, v) = w(u \rightarrow v)$ and $pred(u, v) = u$
• If there’s no shortest path from $u$ to $v$, then $dist(u, v) = \infty$ and $pred(u, v) = NULL$
• The output of our shortest path algorithm will be a pair of $n \times n$ arrays encoding all $n^2$ 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
Lots of Single Sources

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

ObviousAPSP(V,E,w){
    for every vertex s{
        dist(s,*) = SSSP(V,E,w,s);
    }
}

Analysis

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

- We’d like to have an algorithm which takes $O(n^3)$ 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:

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

• **Un fortunately, this recurrence doesn’t work**
• To compute $dist(u, v)$, we first must compute $dist(u, x)$ for every other vertex $x$, but to compute any $dist(u, x)$, we first need to compute $dist(u, v)$
• 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 $dist(u, v, k)$ 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 $n - 1$ edges, what we want to compute is $dist(u, v, n - 1)$. 
The Recurrence

\[ \text{dist}(u, v, k) = \begin{cases} 
0 & \text{if } u = v \\
\infty & \text{if } k = 0 \text{ and } u \neq v \\
\min_x \left( \text{dist}(u, x, k - 1) + w(x \rightarrow v) \right) & \text{otherwise}
\end{cases} \]
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 $\Theta(n^4)$
- This is just because the recurrence has four variables — $u$, $v$, $k$ and $x$ — each of which can take on $n$ different values
- Except for the base cases, the algorithm will just be four nested “for” loops
DP-APSP

DP-APSP(V,E,w){
    for all vertices u in V{
        for all vertices v in V{
            if(u=v)
                dist(u,v,0) = 0;
            else
                dist(u,v,0) = infinity;
        }
    }
    for k=1 to n-1{
        for all vertices u in V{
            for all vertices u in V{
                dist(u,v,k) = infinity;
                for all vertices x in V{
                    if (dist(u,v,k)>dist(u,x,k-1)+w(x,v))
                        dist(u,v,k) = dist(u,x,k-1)+w(x,v);
                }
            }
        }
    }
}
The Problem

• This algorithm still takes $O(n^4)$ 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 $O(n^3 \log n)$ (think about how you might do this)
• However, to get down to $O(n^3)$ run time, we need to use a different third parameter in the recurrence
Number the vertices arbitrarily from 1 to $n$

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 $dist(u, v, n)$.
We get the following recurrence:

\[
dist(u, v, r) = \begin{cases} 
  w(u \rightarrow v) & \text{if } r = 0 \\
  \min\{dist(u, v, r - 1), \\
  dist(u, r, r - 1) + dist(r, v, r - 1)\} & \text{otherwise}
\end{cases}
\]
The Algorithm

FloydWarshall(V,E,w){
    for u=1 to n{
        for v=1 to n{
            dist(u,v,0) = w(u,v);
        }
    }
    for r=1 to n{
        for u=1 to n{
            for v=1 to n{
                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 $n$ possible values
- Thus the run time is $\Theta(n^3)$
- Space required is also $\Theta(n^3)$
Take Away

- Floyd-Warshall solves the APSP problem in $\Theta(n^3)$ 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