# CS 461, Lecture 21

Jared Saia
University of New Mexico

## Today's Outline

*"The path that can be trodden is not the enduring and unchanging Path. The name that can be named is not the enduring and unchanging Name." - Tao Te Ching*

- Bellman-Ford Wrapup
- All-Pairs Shortest Paths



## InitSSSP

```
InitSSSP(s){
  dist(s) = 0;
  pred(s) = NULL;
  for all vertices v != s{
    dist(v) = infinity;
    pred(v) = NULL;
  }
}
```



## GenericSSSP

```
GenericSSSP(s){
  InitSSSP(s);
  put s in the bag;
  while the bag is not empty{
    take u from the bag;
    for all edges (u,v){
      if (u,v) is tense{
        Relax(u,v);
        put v in the bag;
      }
    }
  }
}
```

## Bellman-Ford

- If we replace the bag in the GenericSSSP with a queue, we get the Bellman-Ford algorithm
- Bellman-Ford is efficient even if there are negative edges and it can be used to quickly detect the presence of negative cycles
- If there are no negative edges, however, Dijkstra's algorithm is faster than Bellman-Ford



## Analysis

- The easiest way to analyze this algorithm is to break the execution into phases
- Before we begin the alg, we insert a token into the queue
- Whenever we take the token out of the queue, we begin a new phase by just reinserting the token into the queue
- The 0-th phase consists entirely of scanning the source vertex $s$
- The algorithm ends when the queue contains only the token

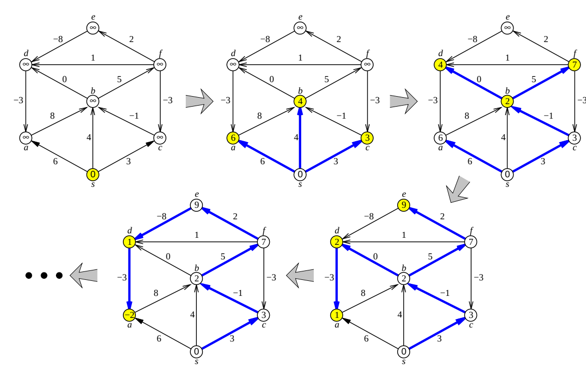


## Invariant

- A simple inductive argument (left as an exercise) shows the following invariant:
- *At the end of the $i$-th phase, for each vertex $v$, $dist(v)$ is less than or equal to the length of the shortest path $s \leadsto v$ consisting of $i$ or fewer edges*
- This implies that the algorithm ends in $O(|V|)$ phases



## Example



Four phases of the Bellman-Ford algorithm run on a directed graph with negative edges.
Nodes are taken from the queue in the order
$s \diamond a\ b\ c \diamond d\ f\ b \diamond a\ e\ d \diamond d\ a \diamond \diamond$, where $\diamond$ is the token.
Shaded vertices are in the queue at the end of each phase.
The bold edges describe the evolving shortest path tree.

## Analysis

- Since a shortest path can only pass through each vertex once, either the algorithm halts before the $|V|$-th phase or the graph contains a negative cycle
- In each phase, we scan each vertex at most once and so we relax each edge at most once
- Hence the run time of a single phase is $O(|E|)$
- Thus, the overall run time of Bellman-Ford is $O(|V||E|)$



## Book Bellman-Ford

- Now that we understand how the phases of Bellman-Ford work, we can simplify the algorithm
- Instead of using a queue to perform a partial BFS in each phase, we will just scan through the adjacency list directly and try to relax every edge in the graph
- This will be much closer to how the textbook presents Bellman-Ford
- The run time will still be $O(|V||E|)$
- To show correctness, we'll have to show that are earlier invariant holds which can be proved by induction on $i$



## Book Bellman-Ford

```
Book-BF(s){
  InitSSSP(s);
  repeat |V| times{
    for every edge (u,v) in E{
      if (u,v) is tense{
        Relax(u,v);
      }
    }
  }
  for every edge (u,v) in E{
    if (u,v) is tense, return ''Negative Cycle''
  }
}
```



## Take Away

- Dijkstra's algorithm and Bellman-Ford are both variants of the GenericSSSP algorithm for solving SSSP
- Dijkstra's algorithm uses a Fibonacci heap for the bag while Bellman-Ford uses a queue
- Diskstra's algorithm runs in time $O(|E| + |V| \log |V|)$ if there are no negative edges
- Bellman-Ford runs in time $O(|V||E|)$ and can handle negative edges (and detect negative cycles)

## 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 $pred(u, v) = NULL$



## APSP

- 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 matric - 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 $|V|$ timnes, once for every possible source vertex
- Specifically, to fill in the subarray $dist(s, *)$, 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 $O(|V|^2|E|) = O(|V|^4)$
- 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 recursive define $dist(u, v)$ as follows:

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

## 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 $u \to v$
- **Unfortunately, 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 $|V| - 1$ edges, what we want to compute is $dist(u, v, |V| - 1)$



## The Recurrence

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



## 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

```
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 |V|-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 $O(|V|^4)$ 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



## Floyd-Warshall

- 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:

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

## 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