CS 362, Lecture 20

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

- Single Source Shortest Paths
- Dijkstra's Algorithm
- Bellman-Ford Algorithm

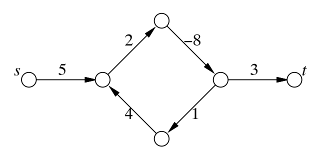
Shortest Paths Problem

- Another interesting problem for graphs is that of finding shortest paths
- Assume we are given a weighted *directed* graph $G = (V, E)$ with two special vertices, a source s and a target t
- We want to find the shortest directed path from s to t
- In other words, we want to find the path p starting at s and ending at t minimizing the function

$$w(p) = \sum_{e \in p} w(e)$$

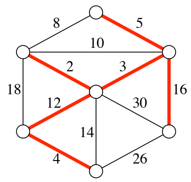
Negative Weights

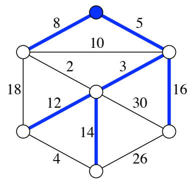
- We'll actually allow negative weights on edges
- The presence of a negative cycle might mean that there is no shortest path
- A shortest path from s to t exists if and only if there is *at least one* path from s to t but no path from s to t that touches a negative cycle
- In the following example, there is no shortest path from s to t



Single Source Shortest Paths

- Singles Source Shortest Paths (SSSP) is a more general problem
- SSSP is the following problem: find the shortest path from the source vertex s to every other vertex in the graph
- The problem is solved by finding a shortest path tree rooted at the vertex s that contains all the desired shortest paths
- A shortest path tree is *not* a MST





SSSP Algorithms

- We'll now go over some algorithms for SSSP on directed graphs.
- These algorithms will work for undirected graphs with slight modification
- In particular, we must specifically prohibit alternating back and forth across the same undirected negative-weight edge
- Like for graph traversal, all the SSSP algorithms will be special cases of a single generic algorithm

SSSP Algorithms

Each vertex v in the graph will store two values which describe a *tentative* shortest path from s to v

- $dist(v)$ is the length of the tentative shortest path between s and v
- $pred(v)$ is the predecessor of v in this tentative shortest path
- The predecessor pointers automatically define a tentative shortest path tree

Defns

Initially we set:

- $dist(s) = 0$, $pred(s) = NULL$
- For every vertex $v \neq s$, $dist(v) = \infty$ and $pred(v) = NULL$

Relaxation

- We call an edge (u, v) *tense* if $dist(u) + w(u, v) < dist(v)$
- If (u, v) is tense, then the tentative shortest path from s to v is incorrect since the path s to u and then (u, v) is shorter
- Our generic algorithm repeatedly finds a tense edge in the graph and *relaxes* it
- If there are no tense edges, our algorithm is finished and we have our desired shortest path tree

Relax

```
Relax(u,v){
  dist(v) = dist(u) + w(u,v);
  pred(v) = u;
}
```


Correctness

- The correctness of the relaxation algorithm follows directly from three simple claims
- The run time of the algorithm will depend on the way that we make choices about which edges to relax

Claim 1

- If $dist(v) \neq \infty$, then $dist(v)$ is the total weight of the predecessor chain ending at v:

$$s \rightarrow \cdots \rightarrow pred(pred(v)) \rightarrow pred(v) \rightarrow v.$$

- This is easy to prove by induction on the number of edges in the path from s to v. (left as an exercise)

Claim 2

- If the algorithm halts, then $dist(v) \leq w(s \rightsquigarrow v)$ for *any* path $s \rightsquigarrow v$.
- This is easy to prove by induction on the number of edges in the path $s \rightsquigarrow v$. (which you will do in the hw)

Claim 3

- The algorithm halts if and only if there is no negative cycle reachable from s.
- The 'only if' direction is easy—if there is a reachable negative cycle, then after the first edge in the cycle is relaxed, the cycle *always* has at least one tense edge.
- The 'if' direction follows from the fact that every relaxation step reduces either the number of vertices with $dist(v) = \infty$ by 1 or reduces the sum of the finite shortest path lengths by some positive amount.

Generic SSSP

- We haven't yet said how to detect which edges can be relaxed or what order to relax them in
- The following Generic SSSP algorithm answers these questions
- We will maintain a "bag" of vertices initially containing just the source vertex s
- Whenever we take a vertex u out of the bag, we scan all of its outgoing edges, looking for something to relax
- Whenever we successfully relax an edge (u, v), we put v in the bag

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;
      }
    }
  }
}
```


Generic SSSP

- Just as with graph traversal, using different data structures for the bag gives us different algorithms
- Some obvious choices are: a stack, a queue and a heap
- Unfortunately if we use a stack, we need to perform $\Theta(2^{|E|})$ relaxation steps in the worst case (an exercise for the diligent student)
- The other possibilities are more efficient

Diskstra's Algorithm

- If we implement the bag as a heap, where the key of a vertex v is $dist(v)$, we obtain Dijkstra's algorithm
- Dijkstra's algorithm does particularly well if the graph has no negative-weight edges
- In this case, it's not hard to show (by induction, of course) that the vertices are scanned in increasing order of their shortest-path distance from s
- It follows that each vertex is scanned at most once, and thus that each edge is relaxed at most once

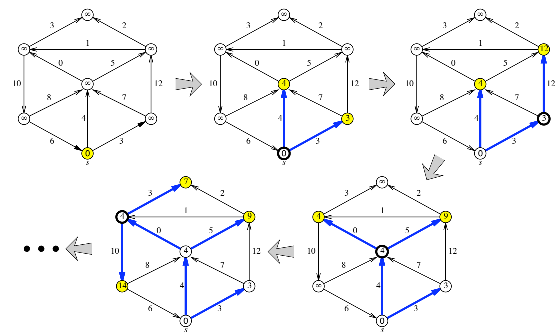
Dijktra's Algorithm

- Since the key of each vertex in the heap is its tentative distance from s, the algorithm performs a DecreaseKey operation every time an edge is relaxed
- Thus the algorithm performs at most $|E|$ DecreaseKey's
- Similarly, there are at most $|V|$ Insert and ExtractMin operations
- Thus if we store the vertices in a Fibonacci heap, the total running time of Dijkstra's algorithm is $O(|E| + |V| \log |V|)$

Negative Edges

- This analysis assumes that no edge has negative weight
- The algorithm given here is still correct if there are negative weight edges but the worst-case run time could be exponential
- The algorithm in our text book gives incorrect results for graphs with negative edges (which they make clear)

Example



Four phases of Dijkstra's algorithm run on a graph with no negative edges.
At each phase, the shaded vertices are in the heap, and the bold vertex has just been scanned.
The bold edges describe the evolving shortest path tree.

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

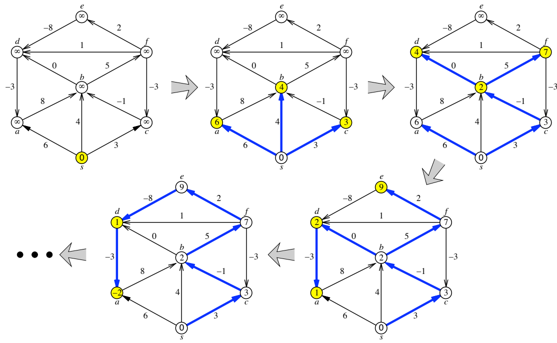
Analysis

- 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 $s \rightsquigarrow v$ consisting of i or fewer edges*

Example



Four phases of Bellman-Ford's 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
- Dijkstra'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 $dist(u, v) = w(u \to v)$ 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"
- 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.