COMPSCI 330: Design and Analysis of Algorithms **October 19, 2017**

Lecture 13: Shortest Path

Lecturer: Rong Ge *Scribe: Weiyao Wang*

13.1 Lecture Overview

In today's lecture we will be covering some algorithm for the shortest path problems. We will first introduce the problem setting for shortest path. Second we will describe and analyze Dijkastra's algorithm designed based on dynamic programming. Third, we will describe the situation of negative edge lengths and Bellman Ford algorithm to deal with this problem.

13.2 Problem Setting: Shortest Path

A most basic situation is when you want to get to some place, and you model the map as a graph and want to find the shortest way to the place. We may model the problem as follows: given a graph $G = (V, E)$ with edge length $w(u, v) > 0$, $\forall e(u, v) \in E$. This length can stand for different meanings, such as distance between two places, travel time between two locations, or cost. In this setting, a negative value may not make sense. Given two vertices in the graph, we may find a path between them, and the length of a path is equal to the sum of edge lengths. The goal we are trying to find the paths with minimum length, given a source $s \in V$ and a destination $t \in V$. Note that the graph can be either directed or undirected, and will not change the algorithms we will be seeing today.

13.3 Designing the Algorithm

To solve the problem, we will still be using the techniques we discussed before. To see what technique we need to use, we first want to have a better understanding of the shortest paths: what properties do shortest paths have? Note that an idea we used frequently is that we want to divide the problems into smaller problems.

Claim: Given a shortest paths from s to t, any sub-path is still a shortest path between its two end-points.

We may proof the claim by contradiction, as if we wre able to find a shorter subpath, we can switch the original subpath to the shorter one, and thus have a shorter path, contradicting to the shortest property. From this observation, we may consider a dynamic programming approach towards the problem.

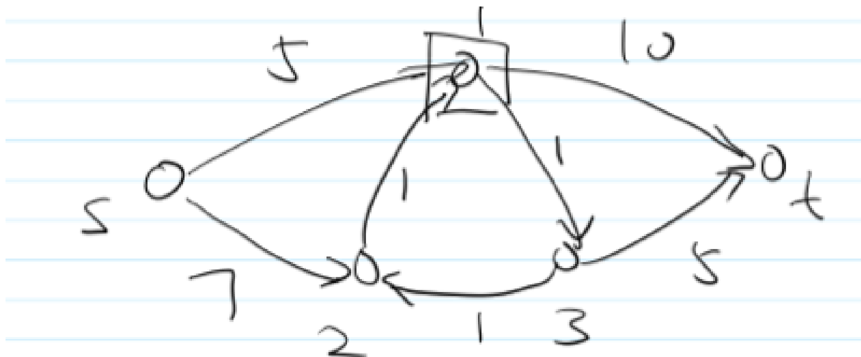
13.3.1 Shortest Path by Dynamic Programming

For dynamic programming, we always consider the decision for the last step. In this setting, it must be the last edge connecting to the terminal. Thus, we are comparing the final edge length and the shortest path to that point. Thus, we may write a transition function:

$$d_v = min_{(u,v)\in E} w(u,v) + d(u)$$

Here $w(u,v)$ is the length of last step and $d(u)$ is the shortest path to a predecessor. The reason is that for any (u,v), we only want he shortest path to u as choosing other paths will result in a higher paths length.

Now we have states and a transition function, the problem is still difficult since we want to know the ordering for the problem. The reason is that graph may have cycles. We may refer to the example below:



For the graph above, it's not possible to solve the dynamic programming by tracing back to a predecessor since we have a cycle in tracing the predecessor.

13.3.2 Dijkstra's Algorithm

The main idea is to find a correct ordering, and the correct ordering is an ascending order of distance from source. The intuition is that to get to a point v, if the last step of shortest path is (u,v), then u should always be closer to s than v. Then it remains on how we can compute the order from ascending order. The idea is that if I have computed shortest paths for all vertices that are closer to s than v, then I am ready to compute shortest paths to v.

Now let's understand how to design the algorithm.

1. We want to compute shortest paths from s to other vertices in ascending order of distance.
2. Initially, we only know $dis(s) = 0$
3. first step: try to find a vertex closest to s. A useful observation here is that the closet point must be a neighbor of s.
4. next step: neighbors of all vertices that we have computed before.

Following the idea, we are maintaining a set W, with property: we know the shortest path from s to any $u \in W$ and distance to any $u \in W$ no larger than distance to any $v \notin W$. Then a step for our algorithm will be finding a vertex $v \notin W$, add v to W such that v should be the one with minimum distance to s among all $v \notin W$. We have

Claim: if v is the one with minimum distance to s for $v \in W$, then the shortest path from s to v only uses points in W.

Using this claim, we are now able to compute the transition function. We may prove the claim by contradiction. If we use a point outside W, u, to compute such $d(v)$, then we have $d(u) < d(v)$, contradicting to the property that $d(v)$ is the shortest for points outside W.

Based the discussions above, we may construct Dijkstra's algorithm:

Dijkstra (s) ;
Initialize $dis[u]$ to be all infinity, $prev[u]$ to be NULL ;
for *neighbors of* s **do**
  Initialize $dis[u] = w[s, u]$ and $prev[u] = s$;
end
Mark s as visited ;
for i*=2 to* n **do**
  Among all vertices that are not visited, find the one with smallest distance, call it u ;
  Mark u as visited ;
  for *all edges* (u, v) **do**
    if $dis[u] + w[u, v] < dis[v]$ **then**
      $dis[v] = dis[u] + w[u, v]$;
      $prev[v] = u$
    end
  end
end

13.3.3 Implementing Dijkstra's algorithm

We would like to explain further beyond the Pseudo-code on how the algorithm is implemented.

1. maintain W (set of vertices with known shortest path)

2. maintain $dis[v]$: for $v \in W$, $dis[v]$ = length of shortest path; for $v \in W$, $dis[v]$ = length of shortest path to v where all vertices are in W.

3. For every iteration: find $v \notin W$ with smallest $d[v]$, add v to W, update $dis[u]$.

13.3.4 Running Time: Dijkstra's algorithm

To analyze the running time of a graph algorithm, usually we want to analyze what is the average amount of time spent on each edge/vertex.

For Dijkstra, we need to find the closest vertex n times and we need to update weight of edges m times.

A naive implementation uses an array to store the distances. Updating one entry of the array takes $O(1)$ time. However, finding the closest vertex requires looking through all the remaining vertices and may take $O(n)$ per iteration.

A better way is to use a binary heap. The heap contains all vertices and is ordered according to their distances. Finding the closest vertex (and remove it from the heap) corresponds to the ExtractMin operation for the heap, which takes $O(\log n)$ time. Updating the weight of an edge requires adjusting the structure of the heap, which is similar to an Insert operation and also takes $O(\log n)$ time. Therefore the total running time is $O((m + n) \log n)$.

The best implementation is to use Fibonacci heap (`https://en.wikipedia.org/wiki/Fibonacci_heap`). This is a heap that supports DecreaseKey operation in $O(1)$ amortized running time. Therefore the running time is $O(\log n)$ per vertex, $O(1)$ per edge. Total runtime is $O(m + n \log n)$.

13.4 Shortest Path with Negative edge length

13.4.1 Situation when $w(u, v)$ to be negative

The motivation here is arbitrage when you exchange currencies. In rare cases, the exchange rates are not carefully tuned, it's possible after cycles we increase the amount we have: more money than you start with. When such thing happens, we call it arbitrage.

13.4.2 Modeling Arbitrage

Suppose u, v are different currencies, exchange rate is $C(u, v)$ (1 unit of v worth $C(u, v)$ units of u). We take the length to be $w(u, v) = logC(u, v)$. The reason for using log is that we want to convert multiplication in currency exchange to addition in calculating path length. With the log operation, the length of a path represents the log of the total currency rate. If I got more money, we have negative cycle, which indicates arbitrage. A shortest path is the best way to exchange money. When there is a negative cycle, the shortest path is not well-defined, then we can go through the cycle infinitely to have infinitely short path.

13.4.3 How to compute shortest path with negative edges?

We first consider the claim we developed for previous case: that given a shortest paths from s to t, any sub-path is still a shortest path between its two end-points. This is still true, since the replacing argument we use is still true. This is good since we can still use the same transition function.

Second, we consider the claim/ observation we have from Dijkstra's algorithm, that if I have computed shortest paths for all vertices that are closer to s than v, then I am ready to compute shortest paths to v. This no longer holds, since we may have negative edge connecting two vertices outside W and our argument no longer holds.

13.4.4 Approach: dynamic programming with steps

The reason we can no longer use the original algorithm is that the ordering no longer holds. Thus, we introduce Bellman-Ford algorithm: we add a dimension to our table: $d[u, i]$ = length of shortest path to get to u with i steps. And our transition becomes

$$d(v, i + 1) = min_{(u,v)\in E} w(u, v) + d(u, i)$$

The number of steps we need to take is going to be n, since we do not want to repeat calculation on a vertex. In this way, we avoid cycles.