CSCI270 Week 8

Jacob Ma

November 1, 2022

1 Pseudo-Polynomial Time

Definition 1.1: Pseudo-Polynomial Time

An algorithm A runs in pseudo-polynomial time if for every input, if all the numbers in the input were written in unary, the algorithm would run in polynomial time in the input size.

Example 1.2: Unary.

$$8 = 11111111$$

What about other algorithms involving numbers?

Example 1.3: Adding numbers in a array a.

```
for (int sum = 0, i = 0; i<n; ++i){ sum += a[i]; }
```

Largest number is K.

Solution

Input Size:

$$\sum_{i=1}^{n} \log\left(a\left[i\right]\right)$$

Upper bound: $n \cdot \log(K)$ - n numbers, each at most K

Lower bound: $n + \log(K)$

Adding two numbers a, b takes $O(\max(\log a, \log b)) = O(\log a + \log b)$

The largest number ever involved in additions is at most $n \cdot K$.

So each addition takes at most $O(\log(nK)) = O(\log n + \log K)$.

Input size is at least $n + \log(K)$.

So the running time is polynomial in the input size.

In general, numbers are not a problem so long as we just do arithmetic and comparisons with them. (See Shaddin Dughmi's write up on the web page.)

The distinction of polynomial vs. pseudo-polynomial is only meaningful when the input contains only numbers. When input is an object, we just take the size.

2 Shortest Path in a Graph - Dynamic Progamming

Problem 1: Shortest Path, Revisited

Given a directed graph G with edge costs $c(e)$ [which could be negative], and start node s, end node t. [Assume that s-t exists.]
Goal: find a shortest path from s to t with respect to the sum of edge costs.

Dijkstra fails. If there were a negative-sum cycle reachable from s and which can reach t, then the notion of "shortest path" would not even be well-defined: for every path, there is a shorter path which goes around the cycle one more time.

We will use a Dynamic Programming Approach.
Sub problems: getting from any node v to t as cheaply as possible. [Could also use getitng from s to v as cheaply as posible]

$$\text{OPT}(t) := \text{ minimum total cost to get from } v \text{ to } t$$

So we have

$$\begin{aligned} \text{OPT}(t) &= 0 \\ \text{OPT}(v) &= \min\left(c\left(v, u(i)\right) + \text{OPT}\left(u(i)\right) \mid i = 1, \cdots, d(v)\right) \end{aligned}$$

The optimum path from v to t takes some first hop to some $u(i)$, and then takes the optimum path form $u(i)$ to t. The total cost of doing this is $c(v, u(i)) + \text{OPT}(u(i))$. Among all of the options, the optimum chooses the best (cheapest) one.
This recurrence is completely correct. But it is not at all clear how to convert it to a tabular or recursive algorithm, because we do not know in what order ton compute the $\text{OPT}(v)$ [or corresponding $a[v]$].
Figuring out an order in which to write the OPT(v) is about as difficult as computing the shortest-path distances in the first place. To circumvent this, we define a new version of OPT.

$$\text{OPT}(v, k) := \text{shortest-path distance/cost from } v \text{ to } t \text{ if the path is allowed to use at most } k \text{ hops.}$$

(We write $d(v)$ for the number of outgoing edges of v.)

$$\begin{aligned} \text{OPT}(t, k) &= 0 \qquad \text{for all } k. \\ \text{OPT}(v, 0) &= \infty \qquad \text{for all } v! = t. \\ \text{OPT}(v, k+1) &= \min_{i=1,\cdots,d(v)} \left(c(v, u_i) + \text{OPT}(u_i, k)\right) \end{aligned}$$

[The optimum path of at most $k+1$ hops takes a first hop to a neighbor of v, then takes an optimum path of at most k hops from there to t. Among all of the $d(v)$ options, the optimum chooses the one minimizing the sum of the costs of getting to u_i, and then getting from u_i to t.]

Tabular Implementation [Bellman-Ford Algorithm]:

```
for (all v){
    a[v,0] = infinity;
}
a[t,0] = 0;
for (int k = 1; k <= n; k ++){
    for (all v != t){
        set a[v][k] = min_{u: (v,u) is an edge} c[v,u] + a[u][k-1];
    }
}
return a[s][n];
```

The shortest path will have at most n nodes (at most $n-1$ hops). If it had more, then by Pigeon Hole Principle, at least one node would repeat. Therefore, it would contain a cycle. By assumption, there are no negative cycles, so this cycle has non-negative weight. So we could remove it and make the path cheaper (or the same).

> *Correctness Proof.* By induction on k, probe that $a[v,k] = \text{OPT}(v,k)$, for all nodes v and all k. Base case uses base case or recurrence.
>
> Induction step uses □

In addition to being able to deal with negative edges, Bellman-Ford is also naturally parallelizable.