Currency Exchange

June 9, 2020

1 Bellman-Ford Algorithm

Dijkstra's algorithm allows us to find shortest path for any weighted graph but where all the edges have non-negative weights

Bellman-Ford Algorithm allows us to use negative weights

Currency Exchange: > You cannot convert some currencies into some others with given exchange rates. What is the manimum amount in Russian rubles you can get from 1000 US dollars using unlimited number of currency conversions? Is it possible to get as many Russian rubles as you want? Is it possible to get as many US dollars as you want?

1.1 Arbitrage

One of the reasonse this question is important is because of **Arbitrage**

After going through three banks, you got a profit for $ 25,406 $ due to the rates

So your starting out with 1 USD and going though diffrent nodes and ending up at $ RUR $. Your goal is to maxamize the return

Maximum Product Over Paths: - Input: - currency exchange graph with weighted edges $ e_i $ between some pairs of currencies with weights $ r_{e_i} $ corresponding to the exchange rate - Output: - Maximize: the product of weights of edges over all the paths from one node to another in the graph

`over paths $ (e_1, e_2, ..., e_k) $ from $ USD $ to $ RUR $ in the graph`

Weird thing about this is that we need to maxmize and not minmize. Even weirder is that we need to maximize the product

2 Reduction to Shortest Paths

Approaches: - Replace product with sum by taking logarithms of weights - Negate weights to solve minimization problem instead of maximization problem

2.1 Taking the Logarithm

$ xy = 2^{log_2(x)}*2^{log_2(y)} = 2^{log_2(x) + log_2(y)} $

The equation above is the correspondence between product $ X $ and $ Y $ and sum of their logarithms

More specifically, if you want to maximize the product of $ x $ and $ y $ is is equivalent to maximizing the sum of the logarithms of $ x $ and $ y $

$ xy => max <=> log_2(x) + log_2(y) -> max $

$ 4 * 1 * 1/2 = 2 = 2^1 $

$ log_2(4) + log_2(1) + log_2(1/2) = 2 + 0 + (-1) = 1 $

So basically our problem of maximizing the product of exchange rates over some paths is equilivant ot maximizing the sum of logarithms of those exchange rates over the same path

So now we can actually reduce our problem to maximize the total weight of the path from one node to another by replacing the initial weights

We take a minimization problem to a maximization problem by the logarithm route

2.2 Negation

so to reduce this further to a minimization problem, we should note that to maximize the sum of logarithms is equivalent to minimize minus the sum of logarithms

This doesn't yet give us convenient reformation though, so instead we will say that to maximize the sum of logarithms, is equivalent to minimize the sum of minus logarithms

So if we replace each logarithm with its negation, then we would need to solve the minimization problem

2.3 Reduction

Finally: replace edge weights $ r_{e_i} by (-log(r_{e_i})) $ and find the shortest path between $ USD $ and $ RUR $ in the graph

so we turn it into the shortest path problem

Steps: - Create currency exchange graphs with wrights $ r_{e_i} $ corresponding to exchange rates - Replace $ r_{e_i} -> (-log(r_{e_i})) $ - Find the shortest path from $ USD $ to $ RUR $ by Dijkstra's algorithm - Do the exchanges corresponding to the shortest path

The steps above actually do not work for this problem

Where Dijkstra's algorithm goes wrong? - Dijkstra's algorithm relies on the fact that a shortest path from $ s $ to $ t $ gors only through vertices that are closer to $ s $ than $ t $ - This is no longer the case for graphs with negative edges

- the algorithm is going to say shortest path is $ S $ to $ A $ (5) and not $ S $ to $ B $ to $ A $ (-10)

when we take the minus logrithm, we get the following edges

The shortest path is going through $ EUR $ but because it is negative, it says that the shortest is simply going straight to $ USD $

2.4 Negatice Weight Cycles

there is a nastier problem when you allow negative edges, which is negative cycles

In currency exchange, a negative cycle can make you a billionaire!

3 Bellman-Ford Algorthm Implementation

We will need to make sure that there are no negative weight cycles, negitive weight edges are ok

Naive Algorithm: - the naive algorithm we implmented where we relaxed edges works for even negative weights

It is standard if we can imporve the `dist[v]` by `dist[u]` and the weight of the edge connecting $ U $ and $ V $. we do update it and also update the previous node for $ V $ to become $ U $

3.1 Algorithm

Steps: - `for all u in V`: we loop through all nodes - `dist[u] <- inf`: we initialize the distance to infinity - `prev[u] <- None`: we dont have any path yet so we init to None - `dist[S] <- 0`: we init the distance of the starting Node to 0 - `repeat |V| - 1 times`: we do number of nodes in graph -1 iterations - `for all (u, v) in E`: for each iteration we try to relax all the edges

3.2 Run Time:

Lemma: > The running time of Bellman-Ford algorithm is $ O (|V||E|)

Proof: - Initalize dist - $ O(|V|) $ - $ |V| - 1 $ iterations, each $ O(|E|) - O(|V||E|)$

3.3 Example

Steps: - in the first iteration, it tried to update all of the edges to relax them - $ S $ to $ A $ and update edge

- Update $ S $ to $ B $ to $ 3 $ and update $ A $ to $ B $ to $ 2 $
- update $ C $ to $ - 1 $ because $ 2 (B) - 3 = -1 $
- with all of that our firtst iteration is complete
- for our graph, further iterations change nothing and we get this as the final result

4 Proof of Correctness

Lemma: > After $ k $ iterations of relaxations, for any node $ u $, $ dist[u] $ is less than or equal to the smallest lenght of a path form $ S $ to $ u $ that contains at most $ k $ edges

Proof: - use mathematical induction - Base: after $ 0 $ iterations, all dist-values are inf, but for `dist[S] = 0`, which is correct - Induction: proved for $ k $ -> prove for $ k + 1 $ - Before $ k + 1-th$ iteration, `dst[u]` is less than or equal to the smallest lenght of a path from $ S $ to $ u $ containing at most $ k $ edges - Each path from $ S $ to $ u $ goes through one of the incoming edges $ (v, u) $ - Relaxing by $ (v, u)$ is comparing it with the smallest lenght of a path from $ S $ to $ u $ through $ v $ containing at most $ k + 1 $ edges

Corollary: > In a graph without negative weight cycle, Bellman-Ford algorithm correctly finds all distances from the starting node $ S $

Proof: - Any path with at least $ V $ edges contains a cycle - The cycle can be removed without making the path longer - Shortest path contains at most $ V - 1 $ edges and will be found after $ V - 1 $ iterations

Corollary: > If there is no negative weight cycle reachable form $ S $ such that $ u $ is reachable from this negative-weight cycle, Bellman-Ford algorithm correctly finds `dist[u] = d(S, u)`

5 Negative Cycles

Lemma: > A graph $ G $ contains a negative weight cycle if and only if $ |V| $-th (additional) iteration of `BellmanFord(G, S)` updates some dist-value

Proof: - If there are no negative cycles, then all shortest paths from $ S $ contain at most $ |V| - 1 $ edges, so no dist-value can be updated on $ |V|$-th iteration - There's a negative weight cycle, say $ a -> b -> c -> a $ but no relaxations on $ |V| $-th iteration

- what happens if we sum these inequalities

5.1 Algorithm for finding Negative Cycle

Algorithms: - Run $ |V| $ iterations of Bellman-Ford algorithm, save node $ v $ relaxed on the last iteration - $ v $ is reachable from a negative cycle - Start from $ x <- v $, follow the link $ x <- prev[x] $ for $ |V| $ times - we will definitely on the cylce - Save $ y <- x $ and go $ x <- prev[x] $ until $ x = y $ again

Even if you can detech the negative cycle, that does not mean you can make the exchange. Because what happens if you are in a negative cycle and have no way of leaving the cycle to go to $ RUR $

6 Infinite Arbitrage

Lemma: > It is possible to get any amount of currency $ u $ from currency $ S $ if and only if $ u $ is reachable from some node $ w $ for which `dist[w]` decreased on iteration $ v $ of Bellman-Ford

Proof:

- `dist[w]` decreased on iteration $ V => w $ is reachable from a negative weight cylce
- $ w $ is reachable => $ u $ is also reachable => infinite arbitrage

Proof: - After $ V - 1 $ iterations, `dist[u] = L` - infinite arbitrage to $ u $ => there exists a path shorter than $ L $ - Thus `dist[u]` will decreased on some iteration $ k >= V $ - if $ edge(x, y) $ was not relaxed and `dist[x]` did not decrease on $ i-th $ iteration, then $ edge(x, y) $ will not be relaxed on $ i + 1-st$ iteration - Only nodes reachable from those relaxed on previous iterations can be relaxed - `dist[u]` is decreased on iteration $ k >= V => u $ is reachable from some node relaxed on $ V-th $ iteration

6.1 Detect Infinite Arbitrage

Steps: - Do $ |V| $ iterations of Bellman-Ford, save all noded relaxed on $ V $-th iterations - set $ A $ - Put all nodes from $ A $ in a queue $ Q $ - do BFS with queue $ Q $ and find all nodes

reachable from $ A $ - All those nodes and only those can have infinte arbitrage - During BFS, remember the parent of each visited node - Reconstruct the path to $ u $ from some node $ w $ relazed on iteration $ V $ - Go back from $ w $ to find negative cycle from which $ w $ is reachable - Use this negative cycle to achive infinite arbitrage from $ S $ to $ u $