

Paths in Graphs: Currency Exchange

Michael Levin

Higher School of Economics

Graph Algorithms
Data Structures and Algorithms

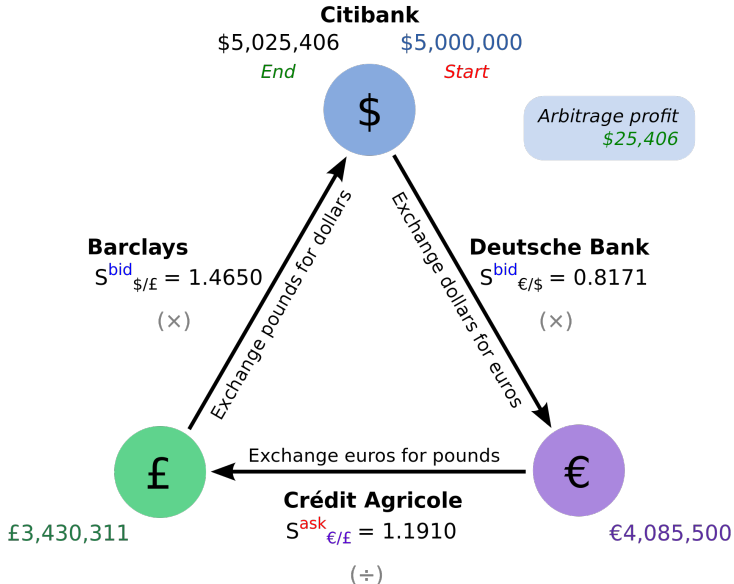
Outline

- 1 Currency Exchange
- 2 Bellman–Ford algorithm
- 3 Proof of Correctness
- 4 Negative Cycles
- 5 Infinite Arbitrage

Currency Exchange

You can convert some currencies into some others with given exchange rates. What is the maximum 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?

Arbitrage





$$1 \text{ USD} \rightarrow 0.88 \cdot 0.84 \cdot \dots \cdot 8.08 \text{ RUR}$$

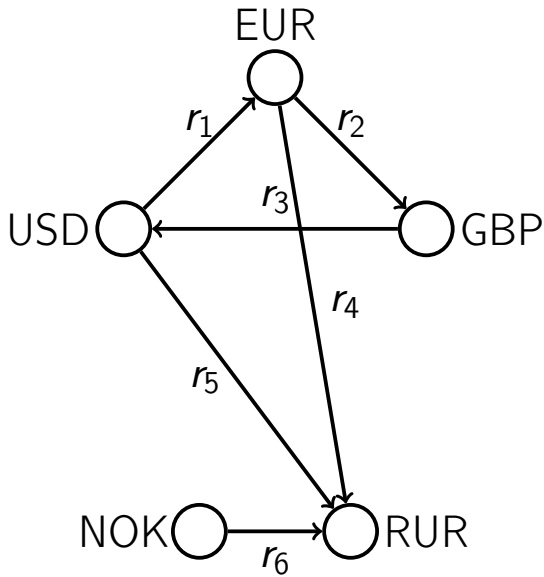
EUR
○

USD○

○GBP

NOK○

○RUR



Maximum product over paths

Input: Currency exchange graph with weighted directed edges e_i between some pairs of currencies with weights r_{e_i} corresponding to the exchange rate.

Output: Maximize $\prod_{j=1}^k r_{e_j} = r_{e_1} r_{e_2} \dots r_{e_k}$ over paths (e_1, e_2, \dots, e_k) from USD to RUR in the graph.

Reduction to shortest paths

So now we want to reduce the problem of maximum product to the problem about shortest paths

Use two standard approaches:

- Replace product with sum by taking logarithms of weights

Because instead of products of weights we want sums of weights like in shortest path algorithm

Reduction to shortest paths

Use two standard approaches:

- Replace product with sum by taking logarithms of weights
 - Negate weights to solve minimization instead of maximization
- Because shortest path is about minimizations

Taking the Logarithm

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

Taking the Logarithm

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

$$xy \rightarrow \max \Leftrightarrow \log_2(x) + \log_2(y) \rightarrow \max$$

Taking the Logarithm

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

$$xy \rightarrow \max \Leftrightarrow \log_2(x) + \log_2(y) \rightarrow \max$$

$$4 \times 1 \times \frac{1}{2} = 2 = 2^1$$

$$\log_2(4) + \log_2(1) + \log_2\left(\frac{1}{2}\right) = 2 + 0 + (-1) = 1$$

Taking the Logarithm

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

Hence we have reduced the problem of maximizing the product to maximizing the sum

$$xy \rightarrow \max \Leftrightarrow \log_2(x) + \log_2(y) \rightarrow \max$$

$$4 \times 1 \times \frac{1}{2} = 2 = 2^1$$

$$\log_2(4) + \log_2(1) + \log_2\left(\frac{1}{2}\right) = 2 + 0 + (-1) = 1$$

$$\prod_{j=1}^k r_{e_j} \rightarrow \max \Leftrightarrow \sum_{j=1}^k \log(r_{e_j}) \rightarrow \max$$

In a nutshell to maximize the product of the weights we need to maximize the sum of their logarithms. And note that this only works if all the numbers are positive because we cannot take logarithms of negative values and also we cannot take logarithms of 0

Negation

$$\sum_{j=1}^k \log(r_{e_j}) \rightarrow \max \Leftrightarrow - \sum_{j=1}^k \log(r_{e_j}) \rightarrow \min$$

Now maximizing the sum of the logarithms is also same as minimizing the NEGATIVE of this sum

And we will also just want to work with the sum and not minus sum

Negation

$$\sum_{j=1}^k \log(r_{e_j}) \rightarrow \max \Leftrightarrow - \sum_{j=1}^k \log(r_{e_j}) \rightarrow \min$$

$$\sum_{j=1}^k \log(r_{e_j}) \rightarrow \max \Leftrightarrow \sum_{j=1}^k (- \log(r_{e_j})) \rightarrow \min$$

maximizing the sum of the logarithms is also same as minimizing the sum of minus logarithms

Reduction

Joining both the ideas discussed above

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

Solved?

- Create currency exchange graph with weights r_{e_i} corresponding to exchange rates

Solved?

- Create currency exchange graph with weights r_{e_i} corresponding to exchange rates
- Replace $r_{e_i} \rightarrow (-\log(r_{e_i}))$

Solved?

- Create currency exchange graph with weights r_{e_i} corresponding to exchange rates
- Replace $r_{e_i} \rightarrow (-\log(r_{e_i}))$
- Find the shortest path from USD to RUR by Dijkstra's algorithm

Solved?

- Create currency exchange graph with weights r_{e_i} corresponding to exchange rates
- Replace $r_{e_i} \rightarrow (-\log(r_{e_i}))$
- Find the shortest path from USD to RUR by Dijkstra's algorithm
- Do the exchanges corresponding to the shortest path

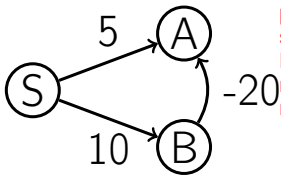
This doesn't work always as
Dijkstra Algorithm goes
wrong in sum conditions

Where Dijkstra's algorithm goes wrong?

- Dijkstra's algorithm relies on the fact that a shortest path from s to t goes only through vertices that are closer to s . closer to s then t

Where Dijkstra's algorithm goes wrong?

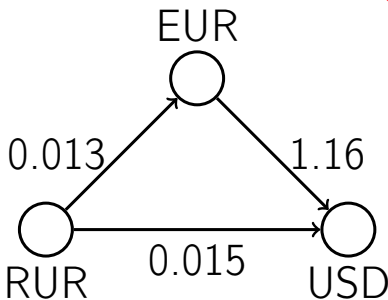
- Dijkstra's algorithm relies on the fact that a shortest path from s to t goes only through vertices that are closer to s .
- This is no longer the case for graphs with negative edges:



Observe over here Dijkstra's algorithm will find the shortest path as $S \rightarrow A$ but actually shortest path is $S \rightarrow B \rightarrow A$. Hence Dijkstra's algorithm does not work in case for graphs with negative edges

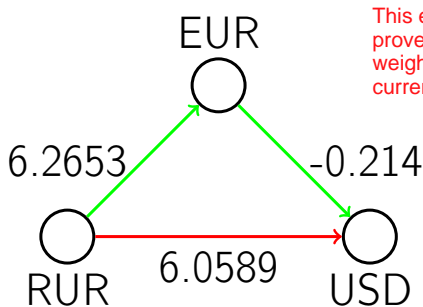
Currency exchange example

This example is discussed here to prove that such cases where weights are negative is possible in currency exchange problem



$$0.013 \times 1.16 = 0.01508 > 0.015$$

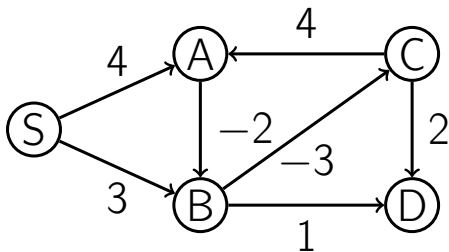
Currency exchange example



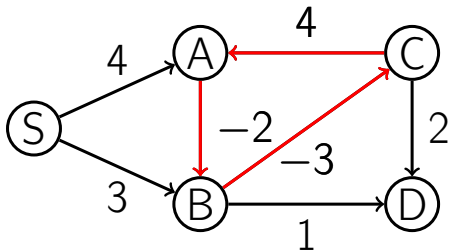
This example is discussed here to prove that such cases where weights are negative is possible in currency exchange problem

$$0.013 \times 1.16 = 0.01508 > 0.015$$

Negative weight cycles



Negative weight cycles



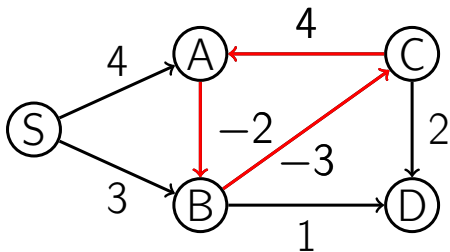
$$d(S, A) = d(S, B) = d(S, C) = d(S, D) = -\infty$$

because every time we go in a cycle highlighted the weight reduces

Same is also true for node B and C as they are in the loop

Same is also true for node D as it is also reachable from nodes A, B, C

Negative weight cycles



$$d(S, A) = d(S, B) = d(S, C) = d(S, D) = -\infty$$

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

Outline

- 1 Currency Exchange
- 2 Bellman–Ford algorithm
- 3 Proof of Correctness
- 4 Negative Cycles
- 5 Infinite Arbitrage

Naive algorithm

- Remember naive algorithm from the previous lesson?

Naive algorithm

- Remember naive algorithm from the previous lesson?
- Relax edges while dist changes

Naive algorithm

- Remember naive algorithm from the previous lesson?
- Relax edges while dist changes
- Turns out it works even for negative edge weights!

Bellman–Ford algorithm

BellmanFord(G, S)

{no negative weight cycles in G }

for all $u \in V$:

$\text{dist}[u] \leftarrow \infty$

$\text{prev}[u] \leftarrow \text{nil}$

$\text{dist}[S] \leftarrow 0$

repeat $|V| - 1$ times:

 for all $(u, v) \in E$:

 Relax(u, v)

If negative cycles then result obtained will be wrong for some nodes

However, we write this pseudo-code in this form just because it is easier to prove the algorithms correctness this way. But, you can just know that if at some iteration during these $|V| - 1$ iterations nothing changed, no edge was actually relaxed, we can just stop there and the distances will already be correct.

This part is only different from naive algorithm In naive algorithm we run the loop until we no longer can relax any edges. This will be even faster when there are no negative weight cycles

Running Time

Lemma

The running time of Bellman–Ford algorithm is $O(|V||E|)$. Longer than Dijkstra's but work for negative edges

Proof

- Initialize dist — $O(|V|)$

Running Time

Lemma

The running time of Bellman–Ford algorithm is $O(|V||E|)$.

Proof

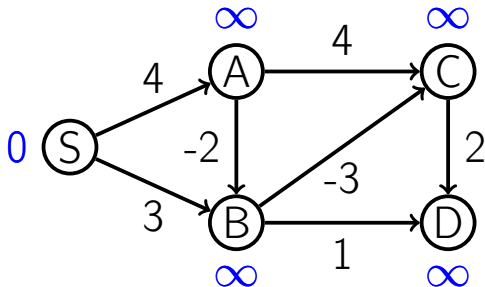
- Initialize dist — $O(|V|)$

$|V|-1$ is when the graph is a long line with V nodes. Hence the longest possible path in a graph has $V-1$ edges

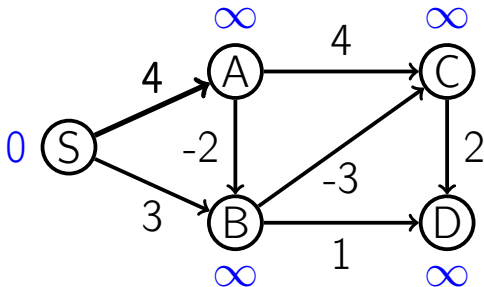
- $|V| - 1$ iterations, each $O(|E|)$ —
 $O(|V||E|)$

Often after just one iteration there will be no further relaxation possible hence most times this time will be just $|E|$ □

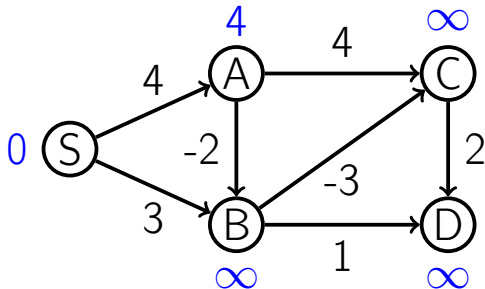
Example



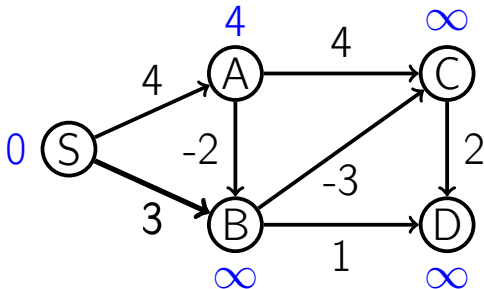
Example



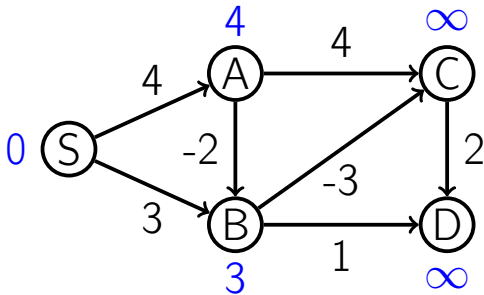
Example



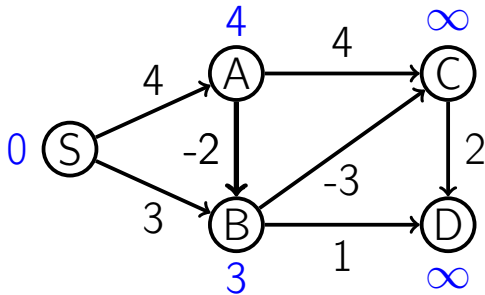
Example



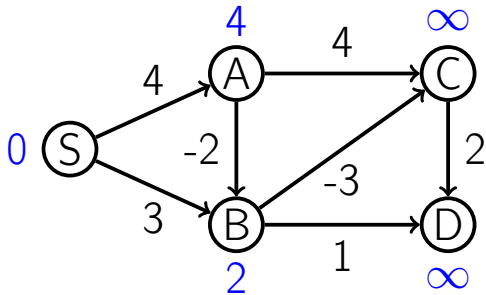
Example



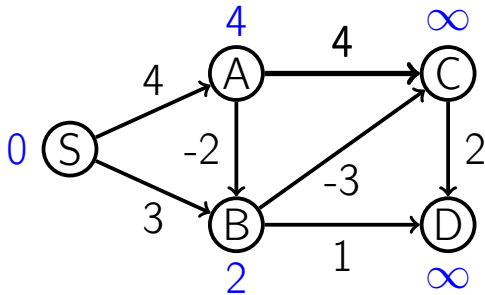
Example



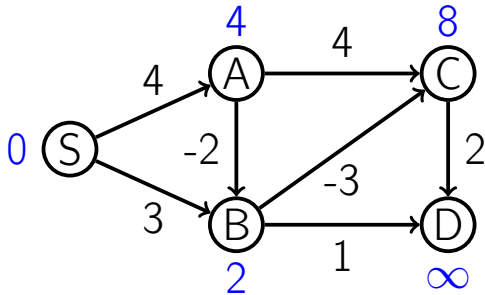
Example



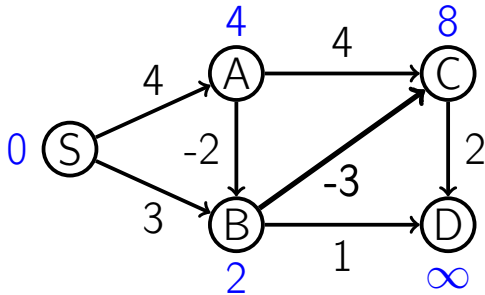
Example



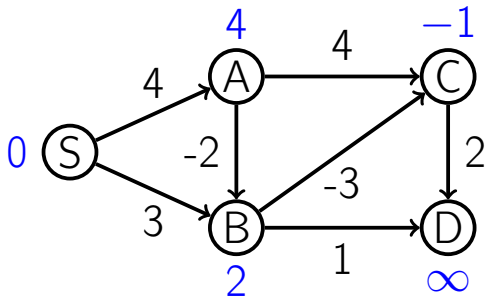
Example



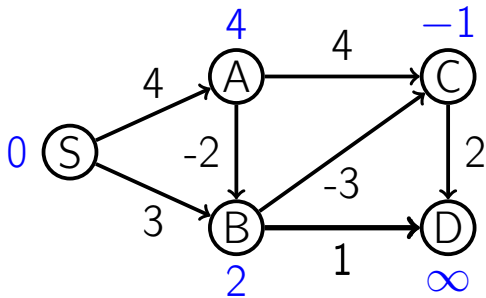
Example



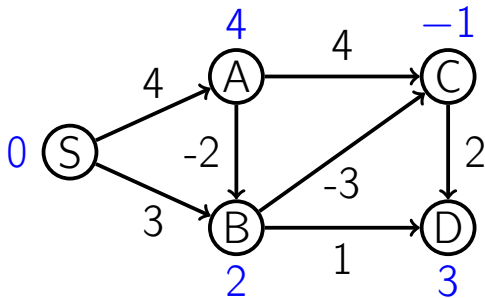
Example



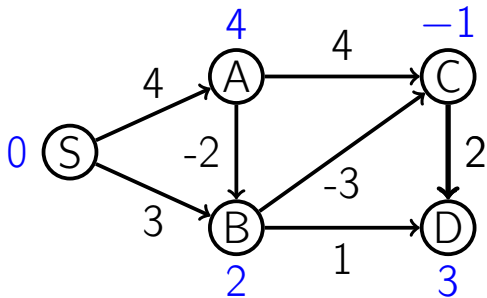
Example



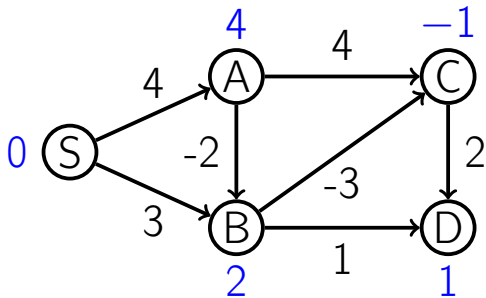
Example



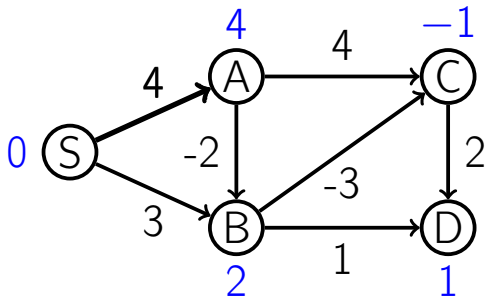
Example



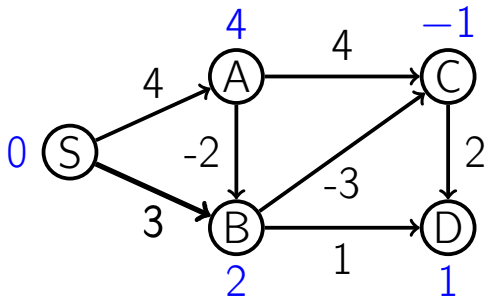
Example



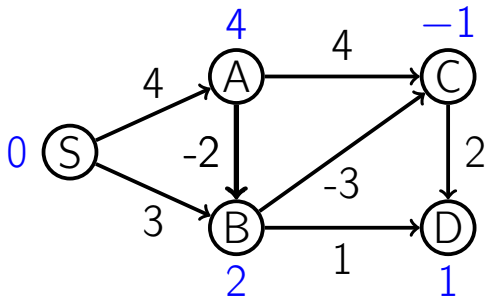
Example



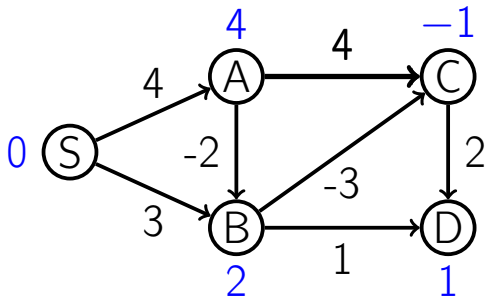
Example



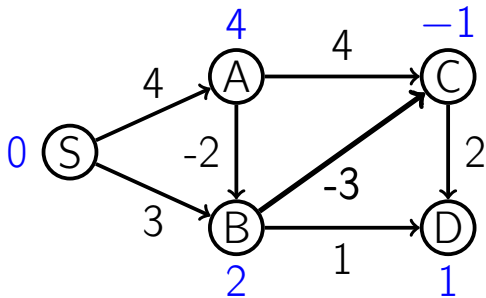
Example



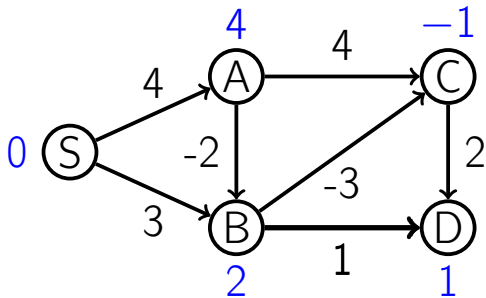
Example



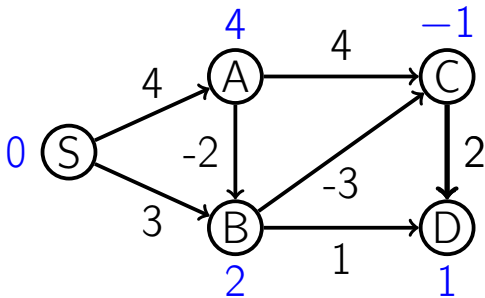
Example



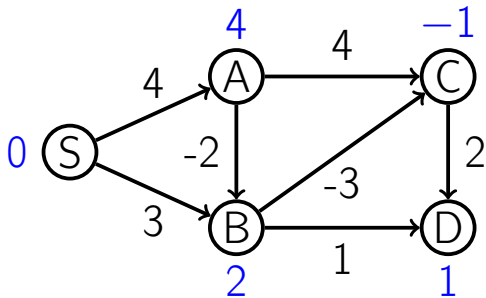
Example



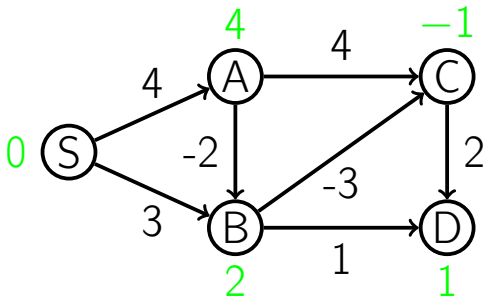
Example



Example



Example



Notice that after one iteration All vertices which are 1 edge away from the source are accurate. After two iterations 2 All vertices which are 2 edge away from the source are accurate. Likewise for n iterations

With $V-1$ iterations we can guarantee that the vertex furthest away from the source will have distance accurately calculated

Outline

- 1 Currency Exchange
- 2 Bellman–Ford algorithm
- 3 Proof of Correctness
- 4 Negative Cycles
- 5 Infinite Arbitrage

For example, after one iteration, we state that dist values of all the nodes will contain the best possible shortest path, which consists from 0 or 1 edges.

Lemma

After k iterations of relaxations, for any node u , $\text{dist}[u]$ is the smallest length of a path from S to u that contains at most k edges.

Here we will prove that Bellman-Ford's algorithm returns correct distances from origin node to all the nodes in the graph in the absence of negative weight cycles.

To understand this lemma watch a video in this link

<https://www.udemy.com/tutorial/from-0-to-1-data-structures/the-bellman-ford-algorithm-visualized/>

Proof

- Use mathematical induction

Proof

- Use mathematical induction
- Base: after 0 iterations, all dist-values are ∞ , but for $\text{dist}[S] = 0$, which is correct.

Proof

- Use mathematical induction
- Base: after 0 iterations, all dist-values are ∞ , but for $\text{dist}[S] = 0$, which is correct.
- Induction: proved for $k \rightarrow$ prove for $k + 1$

Proof

- Before $k + 1$ -th iteration, $\text{dist}[u]$ is the smallest length of a path from S to u containing at most k edges

Proof

- Before $k + 1$ -th iteration, $\text{dist}[u]$ is the smallest length 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)

Proof

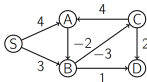
- Before $k + 1$ -th iteration, $\text{dist}[u]$ is the smallest length of a path from S to u containing at most k edges
 - Each path from S to u goes through one of the incoming edges (v, u)
 - Relaxing by (v, u) is comparing it with the smallest length of a path from S to u through v containing at most $k + 1$ edge
- hence if we are finding the $K+1$ edge by smallest k edge then $k+1$ edge will be smallest



Corollary

In a graph without negative weight cycles, Bellman–Ford algorithm correctly finds all distances from the starting node S .

Negative weight cycles



Corollary

If there is no negative weight cycle reachable from S such that u is reachable from this negative weight cycle, Bellman–Ford algorithm correctly finds $\text{dist}[u] = d(S, u)$.

1. First if a graph doesn't have any negative weight cycles, then Bellman-Ford algorithm correctly finds all distances from the starting node S
2. second even if there is a negative cycle in the graph, that doesn't mean that there is no correct distance estimation from origin to some particular node because that particular node may be not reachable from any of the negative weight cycles.

Outline

- 1 Currency Exchange
- 2 Bellman–Ford algorithm
- 3 Proof of Correctness
- 4 Negative Cycles
- 5 Infinite Arbitrage

Negative weight cycles

Lemma

A graph G contains a negative weight cycle if and only if $|V|$ -th (additional) iteration of $\text{BellmanFord}(G, S)$ updates some dist -value.

Lemma to help find if there is a negative cycle in the graph

Proof

Because if there is a cycle and if it is non negative then it can be removed from the shortest path. So NO dist value can be updated on V th iteration

⇐ If there are no negative cycles, then all shortest paths from S contain at most $|V| - 1$ edges (any path with $\geq |V|$ edges contains a cycle, ^{if} it is non-negative, ^{and if} ~~then~~ it can be removed from the shortest path), so no dist-value can be updated on $|V|$ -th iteration.

Proof

\Rightarrow There's a negative weight cycle, say $a \rightarrow b \rightarrow c \rightarrow a$, but no relaxations on $|V|$ -th iteration.

Proof

\Rightarrow There's a negative weight cycle, say $a \rightarrow b \rightarrow c \rightarrow a$, but no relaxations on $|V|$ -th iteration.

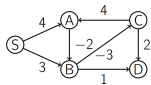
$$\text{dist}[b] \leq \text{dist}[a] + w(a, b)$$

$$\text{dist}[c] \leq \text{dist}[b] + w(b, c)$$

$$\text{dist}[a] \leq \text{dist}[c] + w(c, a)$$

Proof

Negative weight cycles



There's a negative weight cycle, say $a \rightarrow b \rightarrow c \rightarrow a$, but no relaxations on $|V|$ -th iteration.

Here consider these situations step by step

1. There is a negative cycle hence we know that relaxation will happen on V th iteration
2. But for the sake of contradiction let's say relaxation does not happen
3. Now let's consider the sum of these dist values. Note that these equations states that edges (a,b) , (b,c) , (c,a) cannot be relaxed on v th iteration
4. Now if we take sum of these distances the equation in the bottom is obtained which states that the sum of weights is greater than 0
5. But we know that if sum is > 0 then there is no negative cycle but we know that there is a negative cycle hence proved

$$\text{dist}[b] \leq \text{dist}[a] + w(a, b)$$

$$\text{dist}[c] \leq \text{dist}[b] + w(b, c)$$

$$\text{dist}[a] \leq \text{dist}[c] + w(c, a)$$

This won't be true for negative cycle

$$w(a, b) + w(b, c) + w(c, a) \geq 0 \quad \text{---}$$

a contradiction.

Finding Negative Cycle

Algorithm:

- Run $|V|$ iterations of Bellman–Ford algorithm, save node v relaxed on the last iteration

Very Interesting. Algo to find the negative cycle (and not just recognize if the graph contains one or not)

Finding Negative Cycle

Algorithm:

- Run $|V|$ iterations of Bellman–Ford algorithm, save node v relaxed on the last iteration
- v is reachable from a negative cycle

Finding Negative Cycle

Algorithm:

- Run $|V|$ iterations of Bellman–Ford algorithm, save node v relaxed on the last iteration
- v is reachable from a negative cycle
- Start from $x \leftarrow v$, follow the link $x \leftarrow \text{prev}[x]$ for $|V|$ times — will be definitely on the cycle

Finding Negative Cycle

Algorithm:

Very Interesting. Algo to find the negative cycle (and not just recognize if the graph contains one or not)

- Run $|V|$ iterations of Bellman–Ford algorithm, save node v relaxed on the last iteration

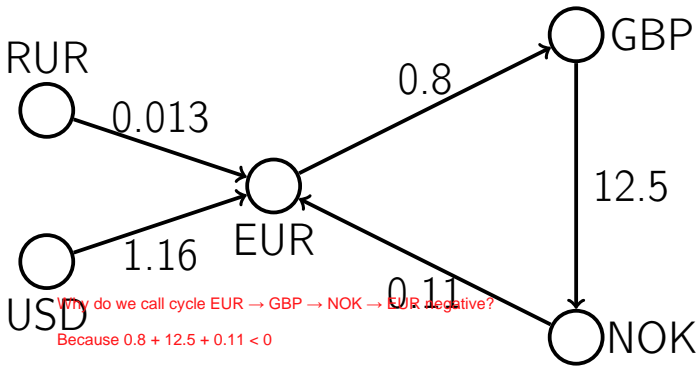
Running time same as that of Bellman-Ford Algorithm

- v is reachable from a negative cycle
- Start from $x \leftarrow v$, follow the link $x \leftarrow \text{prev}[x]$ for $|V|$ times — will be definitely on the cycle
- Save $y \leftarrow x$ and go $x \leftarrow \text{prev}[x]$ until $x = y$ again

Is it possible to get as many rubles as you want from 1000 USD?

Is it possible to get as many rubles as you want from 1000 USD?

Not always, even if there is a negative cycle



Why do we call cycle EUR \rightarrow GBP \rightarrow NOK \rightarrow EUR negative?

Because $0.8 + 12.5 + 0.11 < 0$

Because $0.8 \times 12.5 \times 0.11 = 1.1 > 10.8 \cdot 12.5 \cdot 0.11 = 1.1 > 1$, so $(-\log_2(0.8)) + (-\log_2(12.5)) + (-\log_2(0.11)) = -\log_2(1.1) < 0$

Cannot exchange USD into rubles via
(negative) cycle EUR \rightarrow GBP \rightarrow NOK.

Correct!

Correct! Remember that we replace conversion rates r_{ij} with negative logarithms $-\log_2(r_{ij})$ before proceeding with finding shortest paths.

Outline

- 1 Currency Exchange
- 2 Bellman–Ford algorithm
- 3 Proof of Correctness
- 4 Negative Cycles
- 5 Infinite Arbitrage

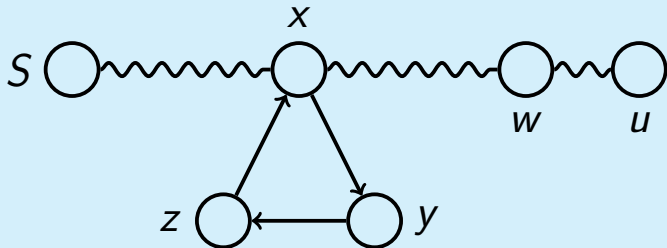
Detect Infinite Arbitrage

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 $\text{dist}[w]$ decreased on iteration V of Bellman-Ford.

Proof

(\Leftarrow)

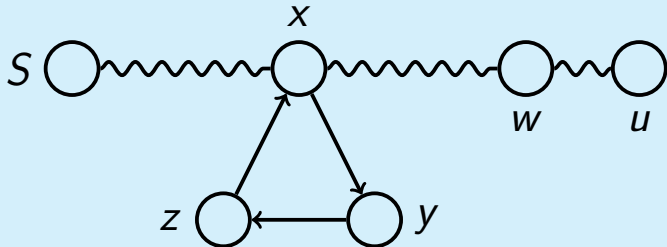


- $\text{dist}[w]$ decreased on iteration $V \Rightarrow w$ is reachable from a negative weight cycle

Proof

PROVES THAT if there is an infinite arbitrage then there is a negative cycle

(\Leftarrow)



- $\text{dist}[w]$ decreased on iteration $V \Rightarrow w$ is reachable from a negative weight cycle
- w is reachable $\Rightarrow u$ is also reachable \Rightarrow infinite arbitrage



Proof

(\Rightarrow)

PROVES THAT if there is an infinite arbitrage then there is a negative cycle

- Let L be the length of the shortest path to u with at most $V - 1$ edges

Proof

(\Rightarrow)

- Let L be the length of the shortest path to u with at most $V - 1$ edges
- After $V - 1$ iterations, $\text{dist}[u]$ is equal to L

Proof

(\Rightarrow)

- Let L be the length of the shortest path to u with at most $V - 1$ edges
- After $V - 1$ iterations, $\text{dist}[u]$ is equal to L This is according to Bellman Ford's Algorithm. That is after $V-1$ iterations all the distances are accurate
- Infinite arbitrage to $u \Rightarrow$ there exists a path shorter than L

Proof

(\Rightarrow)

- Let L be the length of the shortest path to u with at most $V - 1$ edges
- After $V - 1$ iterations, $\text{dist}[u]$ is equal to L
- Infinite arbitrage to $u \Rightarrow$ there exists a path shorter than L
- Thus $\text{dist}[u]$ will be decreased on some iteration $k \geq V$

Proof

(\Rightarrow continued)

- If edge (x, y) was not relaxed and $\text{dist}[x]$ did not decrease on i -th iteration, then edge (x, y) will not be relaxed on $i + 1$ -st iteration

Proof

(\Rightarrow continued)

- If edge (x, y) was not relaxed and $\text{dist}[x]$ 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

Proof

(\Rightarrow continued)

- If edge (x, y) was not relaxed and $\text{dist}[x]$ 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
- $\text{dist}[u]$ is decreased on iteration $k \geq V \Rightarrow u$ is reachable from some node relaxed on V -th iteration



Detect Infinite Arbitrage

- Do $|V|$ iterations of Bellman–Ford, save all nodes relaxed on V -th iteration — set A

Detect Infinite Arbitrage

- Do $|V|$ iterations of Bellman–Ford, save all nodes relaxed on V -th iteration — set A
- Put all nodes from A in queue Q

Detect Infinite Arbitrage

- Do $|V|$ iterations of Bellman–Ford, save all nodes relaxed on V -th iteration — set A
- Put all nodes from A in queue Q
- Do breadth-first search with queue Q and find all nodes reachable from A

Detect Infinite Arbitrage

- Do $|V|$ iterations of Bellman–Ford, save all nodes relaxed on V -th iteration — set A
- Put all nodes from A in queue Q
- Do breadth-first search with queue Q and find all nodes reachable from A
- All those nodes and only those can have infinite arbitrage

Reconstruct Infinite Arbitrage

- During Breadth-First Search, remember the parent of each visited node

Reconstruct Infinite Arbitrage

- During Breadth-First Search, remember the parent of each visited node
- Reconstruct the path to u from some node w relaxed on iteration V

Reconstruct Infinite Arbitrage

- During Breadth-First Search, remember the parent of each visited node
- Reconstruct the path to u from some node w relaxed on iteration V
- Go back from w to find negative cycle from which w is reachable

Reconstruct Infinite Arbitrage

- During Breadth-First Search, remember the parent of each visited node
- Reconstruct the path to u from some node w relaxed on iteration V
- Go back from w to find negative cycle from which w is reachable
- Use this negative cycle to achieve infinite arbitrage from S to u

Conclusion

- Can implement best possible exchange rate
- Can determine whether infinite arbitrage is possible
- Can implement infinite arbitrage
- Can find shortest paths in graphs with negative edge weights