Shortest Paths Revisited 2/4

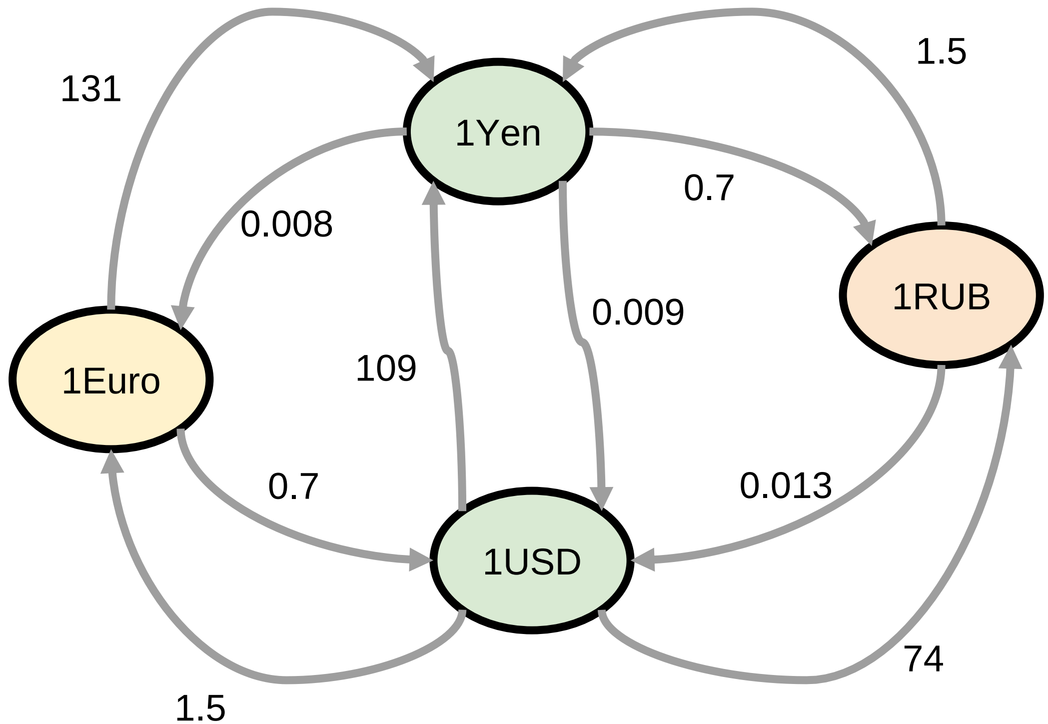
Lecture 07.07 by *Marina Barsky*

Bellman-Ford

Negative edge costs

- It is probably hard to imagine the cases in physical world when the costs of edges are negative: think of a network of roads
- However graphs model many different problems:
  in *decision problems modeled with graphs* we can easily get negative costs (penalties) and positive costs (rewards)
- The problem then is to find the shortest (min-cost) path that minimizes overall penalties – to make the best possible sequence of decisions

Example of a graph with negative edge weights

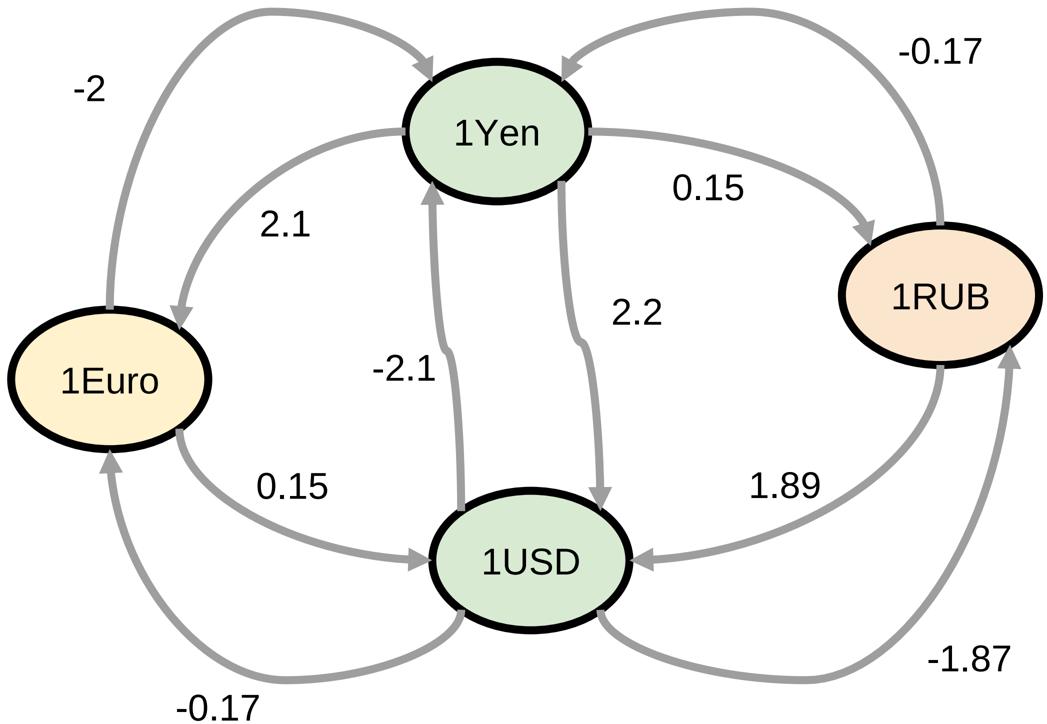


Graph of costs for buying and selling currencies. These are conversion rates

Goal: find the way to convert from RUB to EURO with the biggest loss (dream of a money-exchange agencies)

Note that we need to multiply here

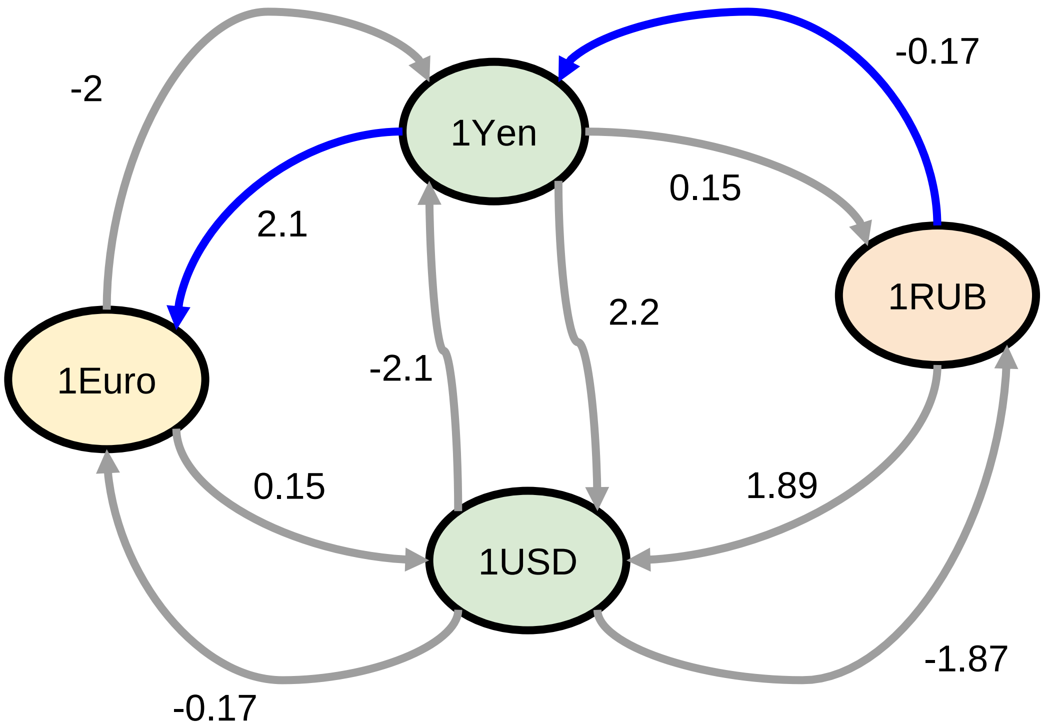
Example of a graph with negative edge weights



To reduce the problem to the shortest (min-cost) path problem:
Represent weights as -log of conversion rates

Now the product will become a sum, and we can compute the shortest (cheapest) path, which will bring us max profit (or smallest loss) with exchanges
However some weights are negative!

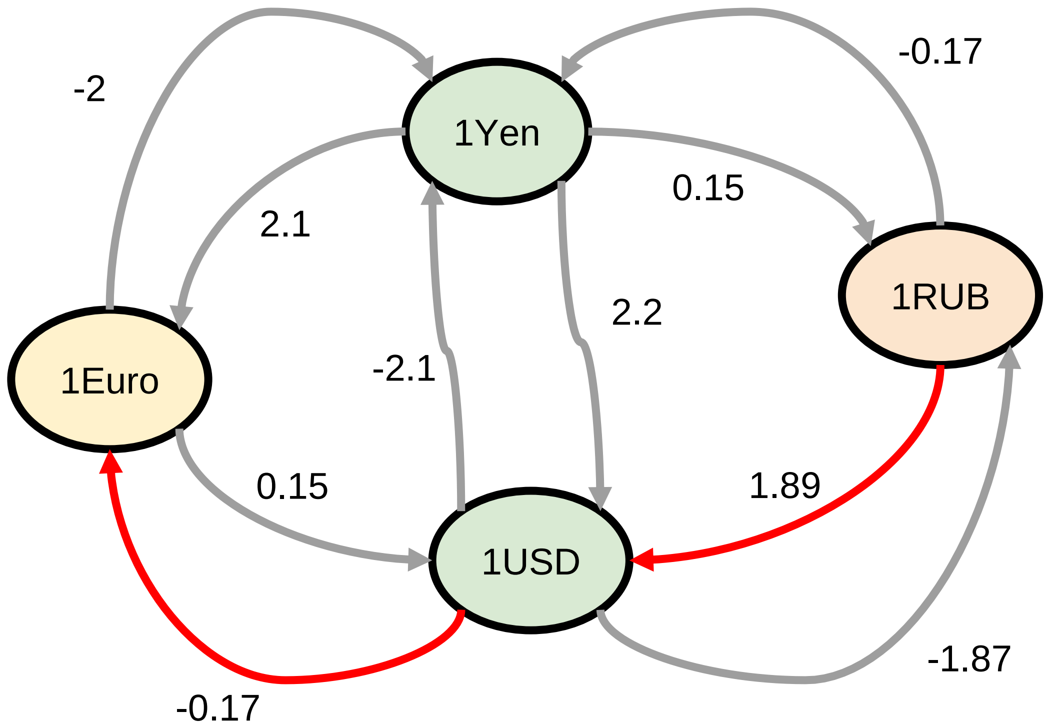
Example of a graph with negative edge weights



What is the min-cost path from RUB to EUR?

-0.17 + 2.1 = 1.93

Example of a graph with negative edge weights

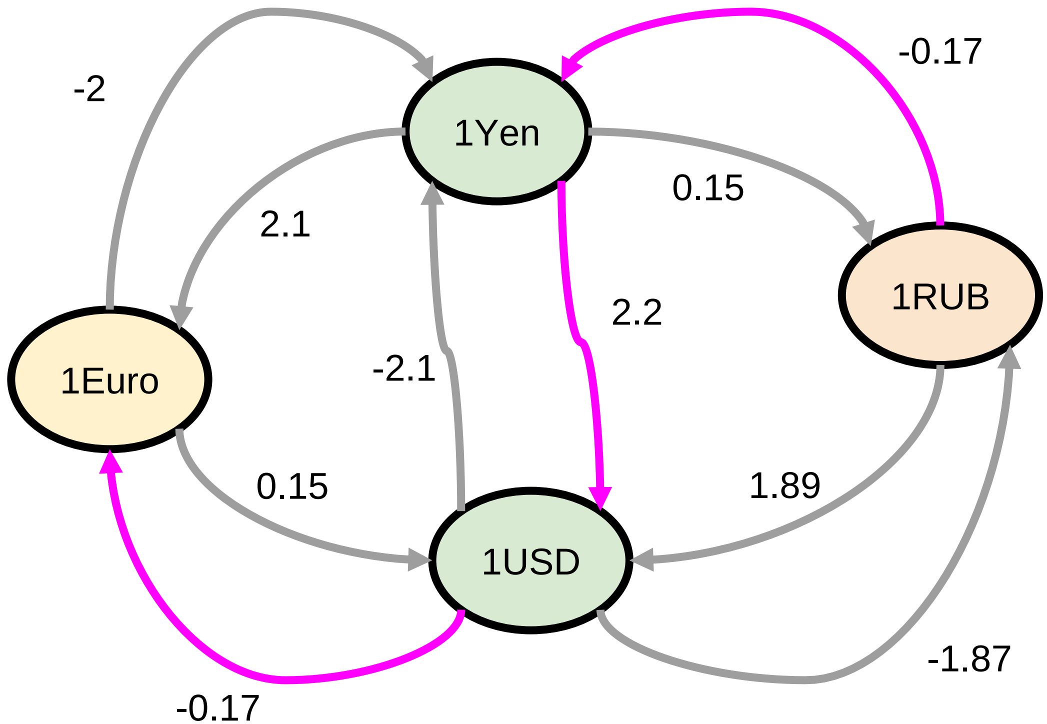


What is the best path from RUB to EUR?

-0.17 +2.1 = 1.93

1.89 - 0.17 = 1.72

Example of a graph with negative edge weights



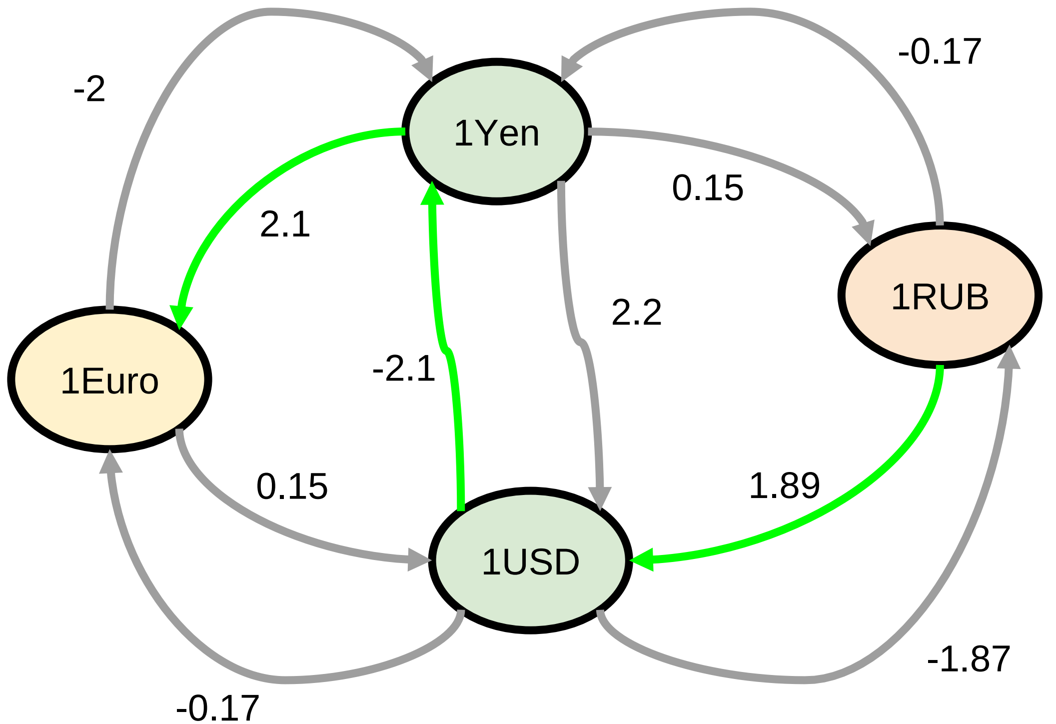
What is the best path from RUB to EUR?

-0.17 + 2.1 = 1.93

1.89 - 0.17 = 1.72

-0.17 + 2.2 - 0.17 = 2.2

Example of a graph with negative edge weights



What is the best path from RUB to EUR?

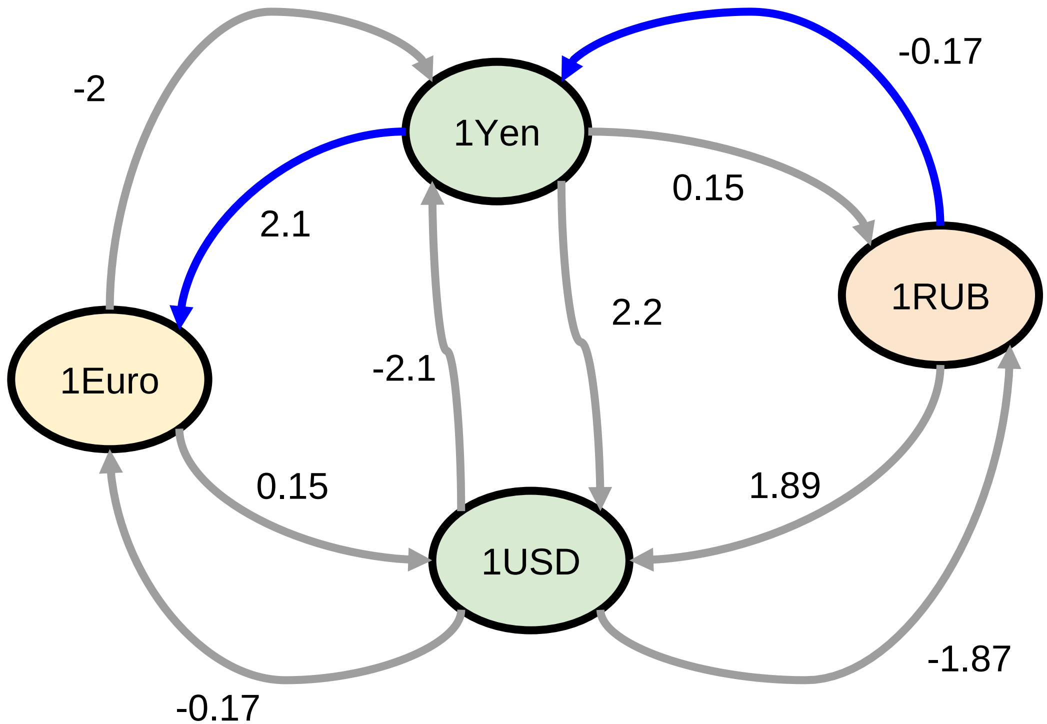
-0.17 + 2.1 = 1.93

+1.89 - 0.17 = 1.72

-0.17 + 2.2 - 0.17 = 2.2

1.89 - 2.1 + 2.1 = 1.89

Example of a graph with negative edge weights



We will lose less money if we exchange this way

The min-cost path:

-0.17 + 2.1 = 1.93

+1.89 - 0.17 = 1.72

-0.17 + 2.2 - 0.17 = 2.2

1.89 - 2.1 + 2.1 = 1.89

Luckily we have only 4 nodes: Dijkstra does not help here!

Use Bellman-Ford

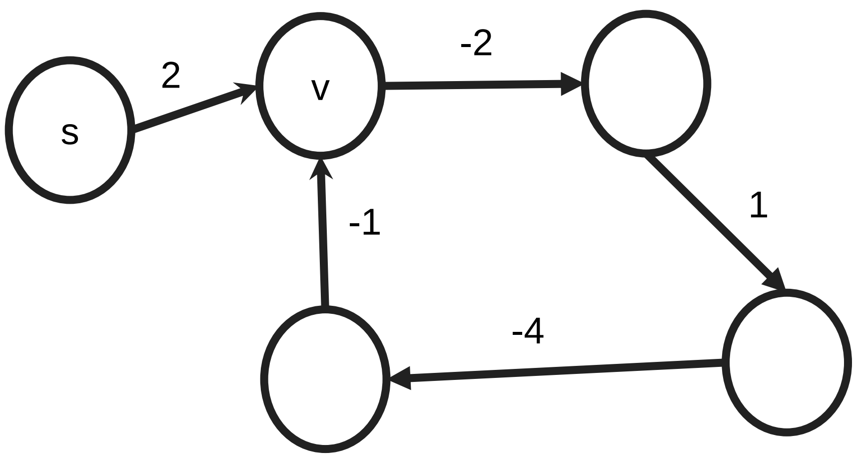
Single-Source shortest paths with positive **and negative** edge costs

Bellman-Ford Algorithm

Dynamic Programming!

Negative edge costs: problem!

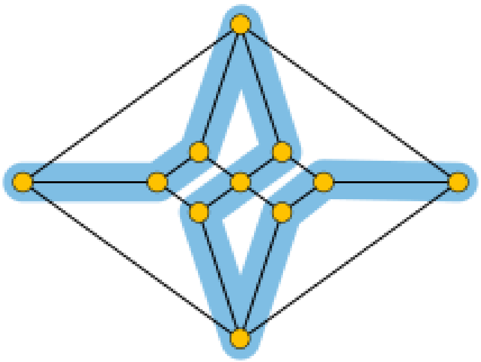
- If we allow some weights be negative, we facing the problem of a **negative cycle:** a cycle with the total cost < 0
- All shortest-path algorithms based on iterative improvement will fail here, because the cost of a path can be improved indefinitely!



The cost of path s~>v can be improved indefinitely!

Avoiding cycles: even bigger problem!

- We may think of limiting the search to paths that avoid traversing cycles, but that leads to an even bigger problem:
  - If we do not allow paths to use cycles, we are asking for something which is called *a simple path*: a path that repeats no vertex.
  - If we need a path to every vertex – then we are asking for nothing else but a Hamiltonian Path – and no efficient algorithm is known for computing it!



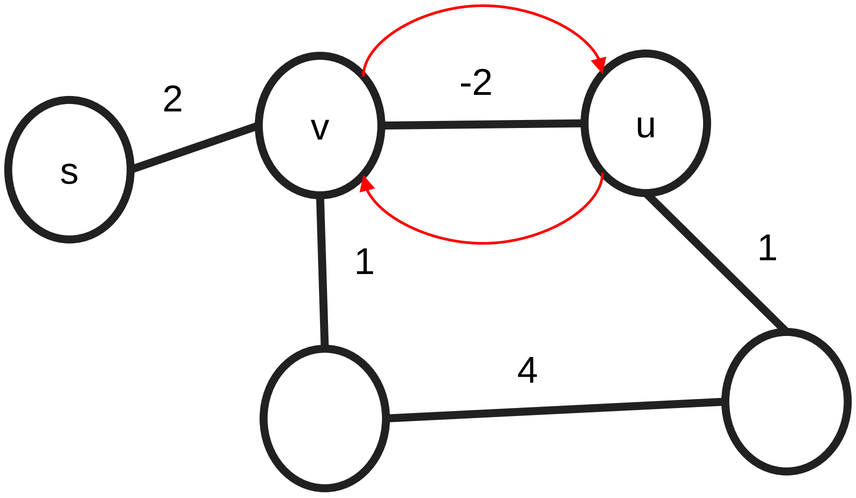
A Hamiltonian cycle visits every **node** of a graph exactly once

Unfortunately, no polynomial-time algorithm is known for finding Hamiltonian paths!

Negative-sum cycles

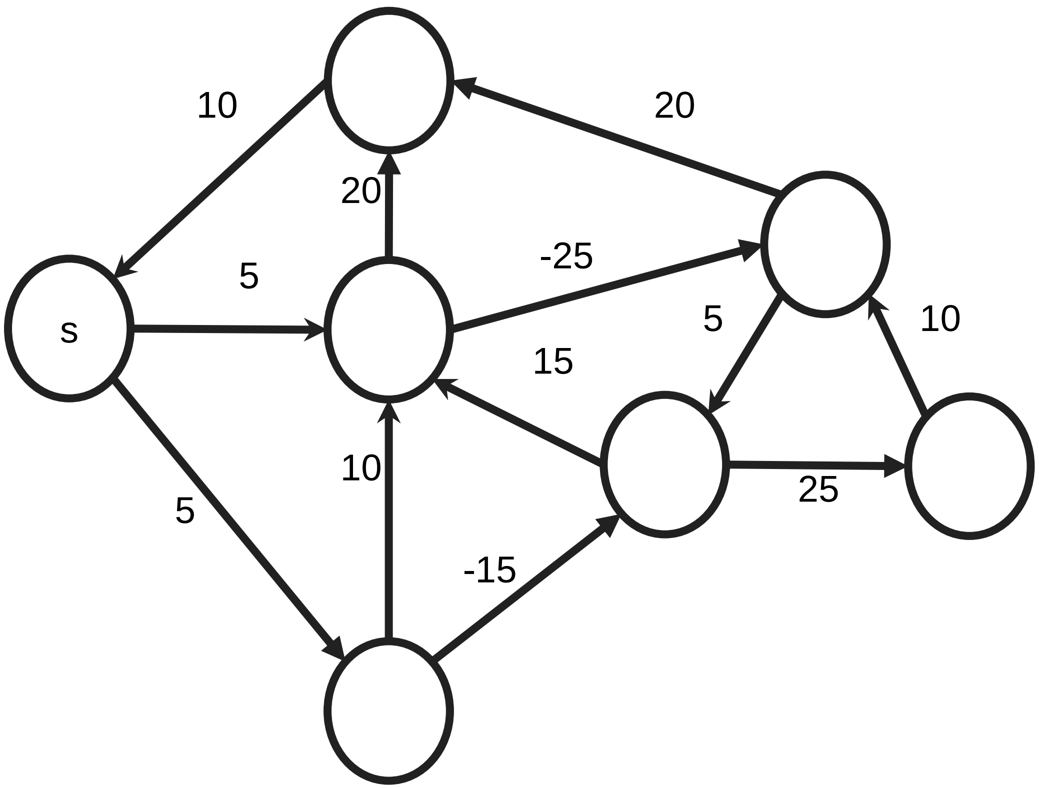
- If the graph contains a negative cycle, then all the shortest paths produced by any of the shortest paths algorithms are unreliable (may be not the shortest)
- Thus we either believe that our input graph does not contain negative-weight cycles, or we ask the algorithm to at least inform us if such cycle is present

- For the same reason, while working with negative-edge weights **we cannot** really **work with undirected graphs**: each negative-cost edge can be considered as a negative-weight cycle of 2 nodes



We cannot work with **undirected** graphs with **negative edge costs:**
Move back and forth between v and u and the cost will decrease indefinitely

Quiz: how many edges in any shortest path?

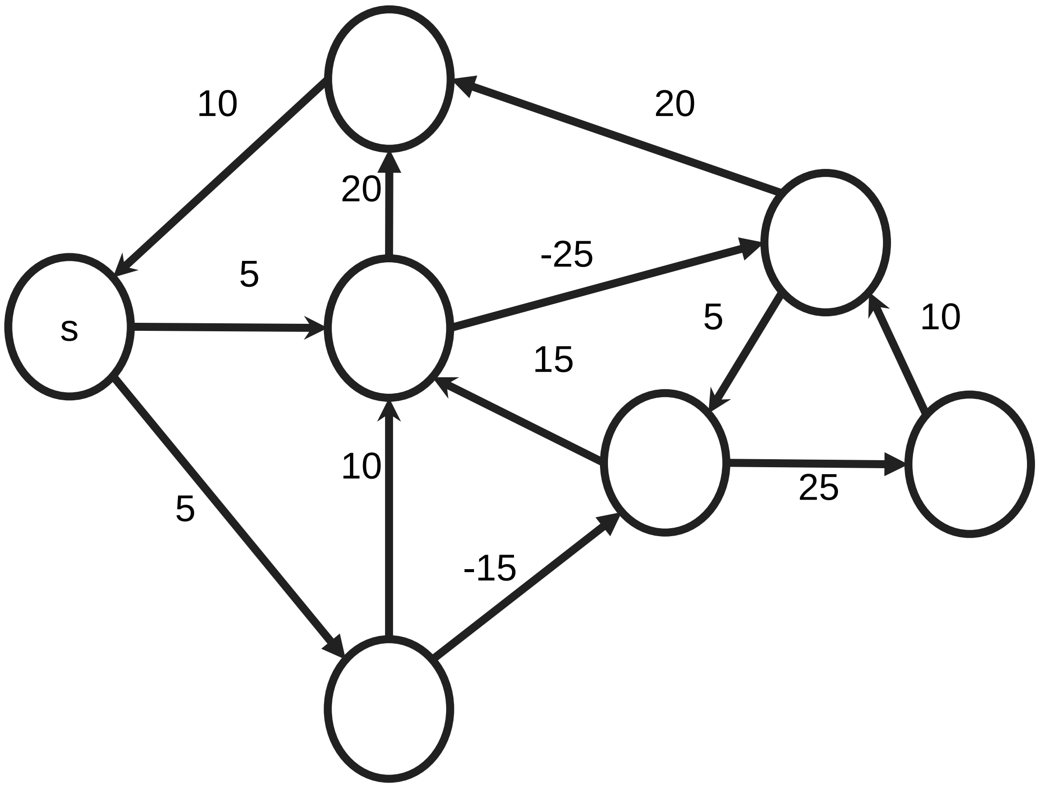


Given directed graph G=(V,E) without negative cost cycles, what is the maximum number of edges in a shortest path u~>v?

- Total number of edges:

A. At most n
B. At most n-1
C. At most n+1
D. At most n^2

Quiz: how many edges in any shortest path?



Given directed graph G=(V,E) without negative cost cycles, what is the maximum number of edges in a shortest path u~>v?

- Total number of edges:

A. At most n
B. At most n-1
C. At most n+1
D. At most n^2

A shortest path from s to v will contain in total no more that n vertices and n-1 edges, because these shortest paths would not contain cycles: the only cycles that could improve the path cost are negative-weight cycles, and they are not allowed

Generic Single-Source Shortest Paths problem

Input: directed graph G=(V,E), array C of edge costs [possibly negative], source vertex s.

Output: if G has no negative-weight cycles, then for every vertex v ∈ V, shortest path s~>v.

Recap: when to use Dynamic Programming

- **There is a “natural” ordering of subproblems from smallest to largest such that you can obtain the solution for a subproblem by only looking at smaller subproblems.**

- It is easy to decide which subproblem is smaller when the input is a sequence: array (knapsack items) or strings (edit distance)
- It is much harder to imagine a “natural” ordering of subproblems on graphs: they have no particular order on vertices or edges

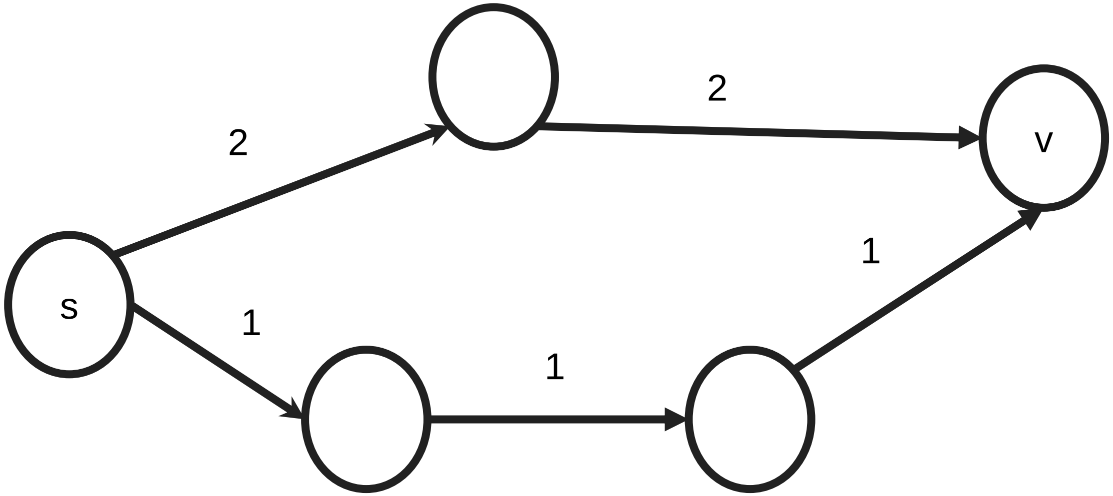
- **If we do not have a “natural” ordering we need to impose an artificial ordering: this is the main step in designing DP algorithms on graphs**

Order of subproblems

- We will exploit the sequential nature of a path: if a path is optimal, then every sub-path must also be optimal

- **Issue:** not clear how to define smaller and larger subproblems
- **Key idea:** artificially restrict the number of edges in the path

- Subproblems are ordered by the number of edges allowed in the path



Example of subproblems:

The shortest path s~>v with edge budget = 2 has cost 4

The shortest path s~>v with edge budget = 3 has cost 3

First subproblem will be considered smaller than the second and will be solved first

Optimal subproblems

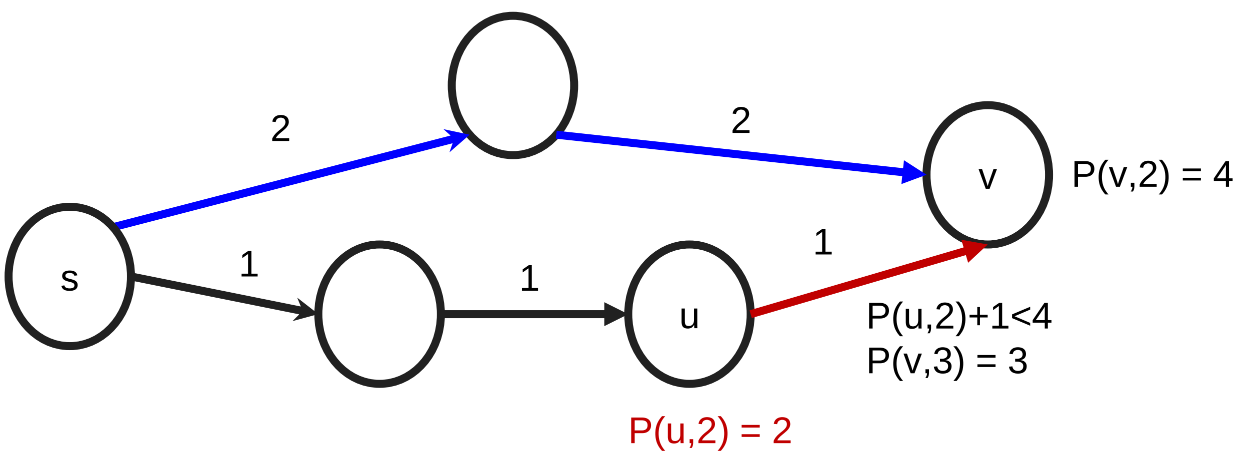
Let P(v,k-1) be the cost of shortest path from the source vertex s to v using at most k-1 edges

We increase the edge budget by allowing one more edge and want to compute P(v, k)

What are possible choices?

- For each incoming edge (u,v) we extend all (already computed) paths P(u, k-1) by edge (u,v)
- If adding any of these edges to paths P(u, k-1) does not result in a shorter path: then
  P(v, k) = P(v, k-1) [we keep the previous shortest path]
- Otherwise we get a shorter path using one of the incoming (u,v) edges:
  $P(v,k) = P(u, k-1) + c_{uv}$

For each vertex v we need to consider at most 1 + in-degree(v) candidate paths with the edge budget <= k



Recurrence relation

- Let P(v,k) be the cost of the shortest path s~>v with the total budget k of allowed edges [path s~>v contains ≤ k edges]

Base case: k=0 [0 edges allowed]

$$P(v,0) = \begin{cases} 0 & \text{if } v=s \\ \infty & \text{if } v \neq s \end{cases}$$

Recurrence relation

- Let P(v,k) be the cost of the shortest path s~>v with the total budget k of allowed edges [path s~>v contains ≤ k edges]

Base case: k=0 [0 edges allowed]

$$P(v,0) = \begin{cases} 0 & \text{if } v=s \\ \infty & \text{if } v \neq s \end{cases}$$

Recurrence: $0 < k \leq n-1$

Max number of edges n-1

$$P(v,k) = \min \begin{cases} P(v, k-1) \\ \underset{\text{over all edges}(u,v)}{\min} \left(P(u, k-1) + c_{uv}\right) \end{cases}$$

Pseudocode

Algorithm BellmanFord (digraph G=(V, E), edge costs C, start node s)

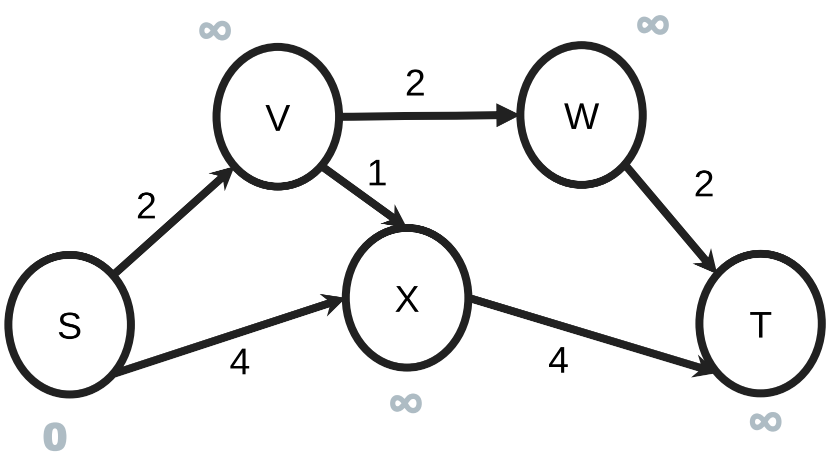
```
A: = nxn 2D array indexed by k and v

# base case
A[0, s] := 0
for each v ∈ V:
        A[0, v] := ∞

# DP table
for k from 1 to n-1:
        for each v ∈ V:
                A[k,v]:= A[k-1][v]
                for each edge (u, v): # check all incoming edges of v
                        if  A[k-1][u] + C[u,v] < A[k,v]:
                                A[k,v]: = A[k-1][u] + C[u,v]

return A[n-1]      # the last row contains final shortest paths from s
```

Bellman-Ford: illustration

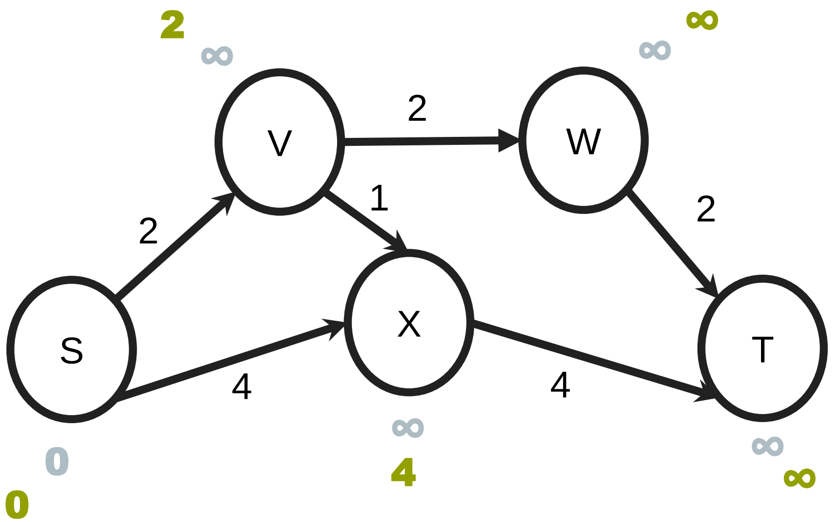


- k=0 [zero edges allowed]

k	S	T	V	W	X
0	0	∞	∞	∞	∞
1					
2					
3					
4					

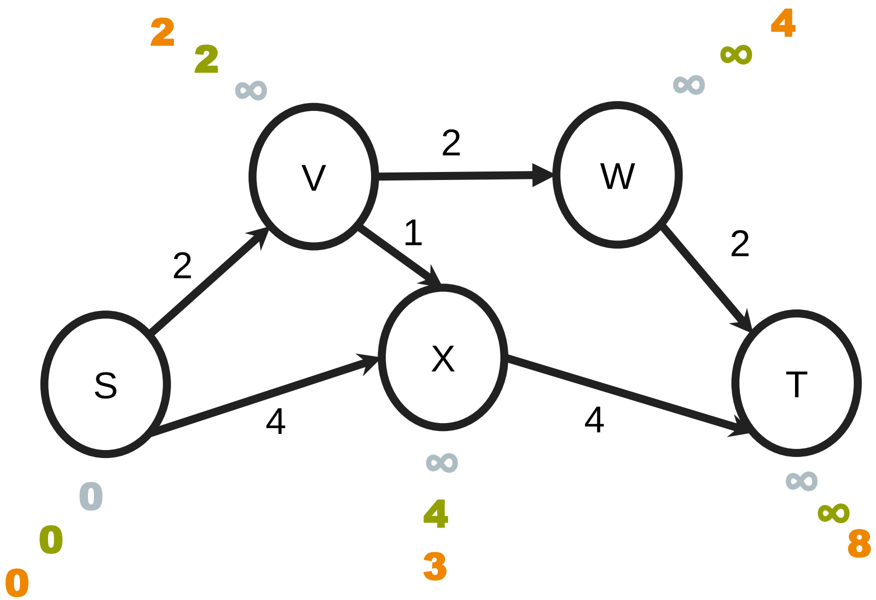
Bellman-Ford: illustration

- k=1 [shortest paths with 1 edge]



k	S	T	V	W	X
0	0	∞	∞	∞	∞
1	0	∞	2	∞	4
2					
3					
4					

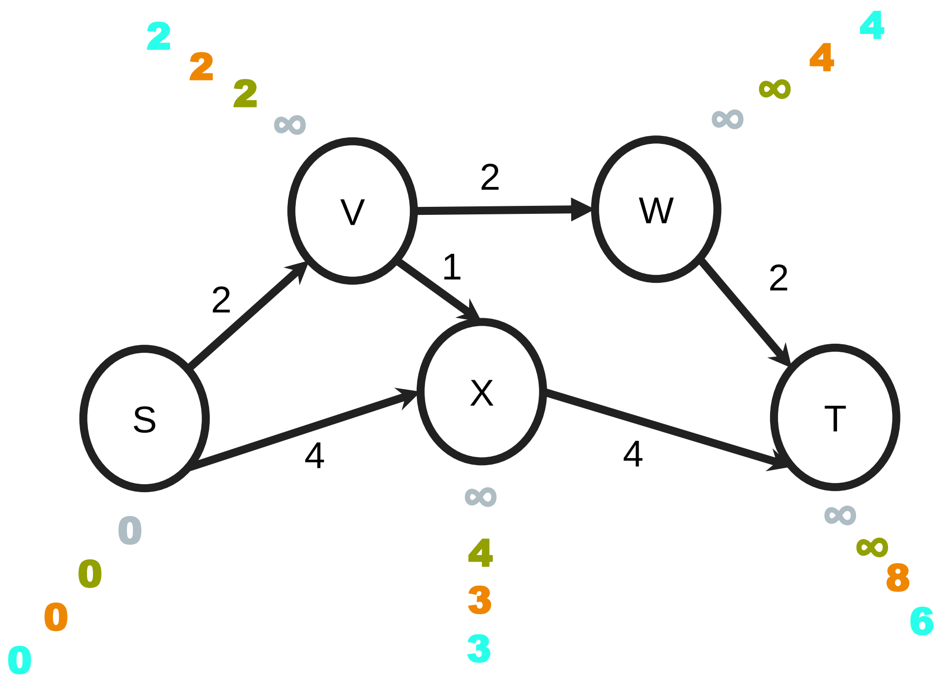
Bellman-Ford: illustration



- k=2

k	S	T	V	W	X
0	0	∞	∞	∞	∞
1	0	∞	2	∞	4
2	0	8	2	4	3
3					
4					

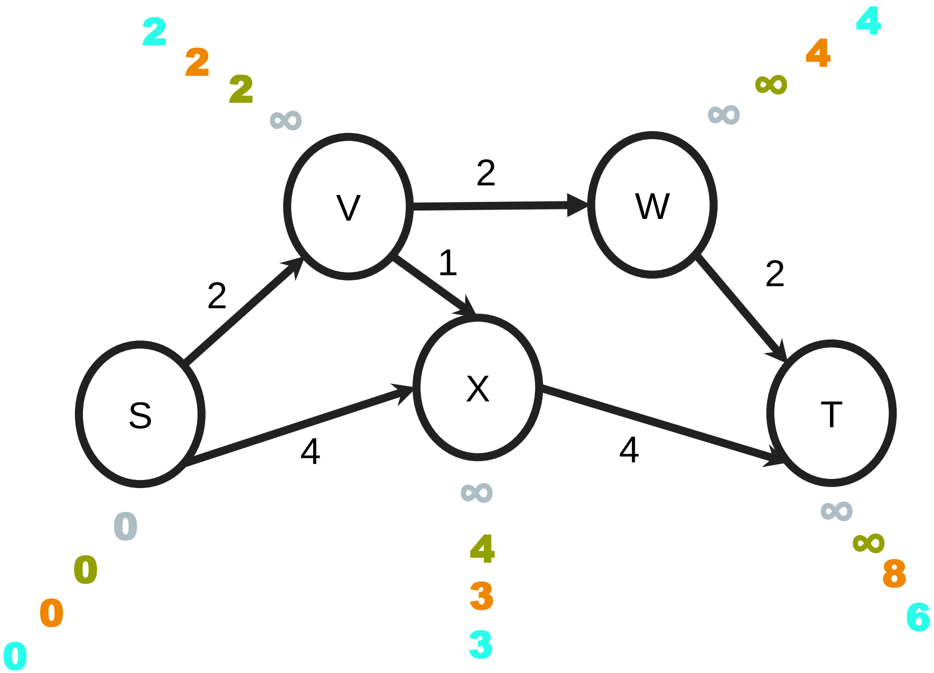
Bellman-Ford: illustration



- k=3

i	S	T	V	W	X
0	0	∞	∞	∞	∞
1	0	∞	2	∞	4
2	0	8	2	4	3
3	0	6	2	4	3
4					

Bellman-Ford: illustration



- k=4

i	S	T	V	W	X
0	0	∞	∞	∞	∞
1	0	∞	2	∞	4
2	0	8	2	4	3
3	0	6	2	4	3
4	0	6	2	4	3

Running Time

Algorithm BellmanFord(digraph G=(V, E), edge costs C)

```
A: = nxn 2D array indexed by k and v

# base case
A[0, s] := 0
for each v ∈ V:
        A[0, v] := ∞

# DP table
for k from 1 to n-1:
        for each v ∈ V:
                A[k,v]:= A[k-1][v]
                for each edge (u, v): # check all incoming edges of v
                        if  A[k-1][u] + C[u,v] < A[k,v]:
                                A[k,v]: = A[k-1][u] + C[u,v]

return A[n-1]        # the last row contains final shortest paths from s
```

Loop is executed n times

At each iteration – total O(m) edges are checked for all the subproblems at iteration k
Sum(in-degree(V)) = O(m

The amortized cost of this inner loop is O(m)

Running time: O(nm)

Bellman-Ford algorithm: notes

- **Early stopping:**
  - We can run less than n-1 iterations
  - If there is no improvements between iteration k-1 and iteration k, then the algorithm computed all shortest paths

- **Detecting negative-weight cycles:**
  - If algorithm continues until iteration n-1, then we run one more iteration
  - If we have improvements in iteration n, then G contains a negative-cost cycle
  - Conclusion: all the shortest paths are unreliable

- **Space improvement:**
  - We can reconstruct the shortest paths by a regular traceback: but this requires to store all n^2 cells of the DP table
  - However due to sequential nature of a path and the fact that each sub-path of the optimal path is by itself optimal – we just need to store the predecessor node for each destination vertex v: when the path gets improved, we store the source node u which caused this improvement
  - Because the sub-path s~>u is by itself optimal, we can continue recovering the path by looking at predecessor of u etc., until we reach node s