All-Pairs Shortest Paths

Lecture 07.08 by *Marina Barsky*

All-pairs Shortest Paths Problem

Input: directed graph G=(V,E) with edge costs C [no special source vertex]

Output: if G has no negative cycles, the length of a shortest path for each pair of vertices u,v ∈ V

All-pairs shortest paths: possible solutions

Use single-source shortest path algorithm:

Repeat n times (once for each vertex as a source)

1. If the costs are non-negative

$$n\text{*Dijkstra } (m \log n) = O(nm \log n) = \begin{cases} O(n^2 \log n) & \text{if } m=O(n) \text{ [sparse]} \\ O(n^3 \log n) & \text{if } m=O(n^2) \text{ [dense]} \end{cases}$$

2. If allowing negative costs:

$$n\text{*Bellman-Ford } (nm) = O(n^2 m) = \begin{cases} O(n^3) & \text{if } m=O(n) \text{ [sparse]} \\ O(n^4) & \text{if } m=O(n^2) \text{ [dense]} \end{cases}$$

Special Dynamic Programming algorithm:

1. Floyd-Warshall: always $O(n^3)$

All-Pairs Shortest Paths

Floyd-Warshall Algorithm

Dynamic Programming

Order of subproblems

Again – there is no “natural” ordering of subproblems: which subproblem is smaller than the other?

Idea: we invent our own order of subproblems:

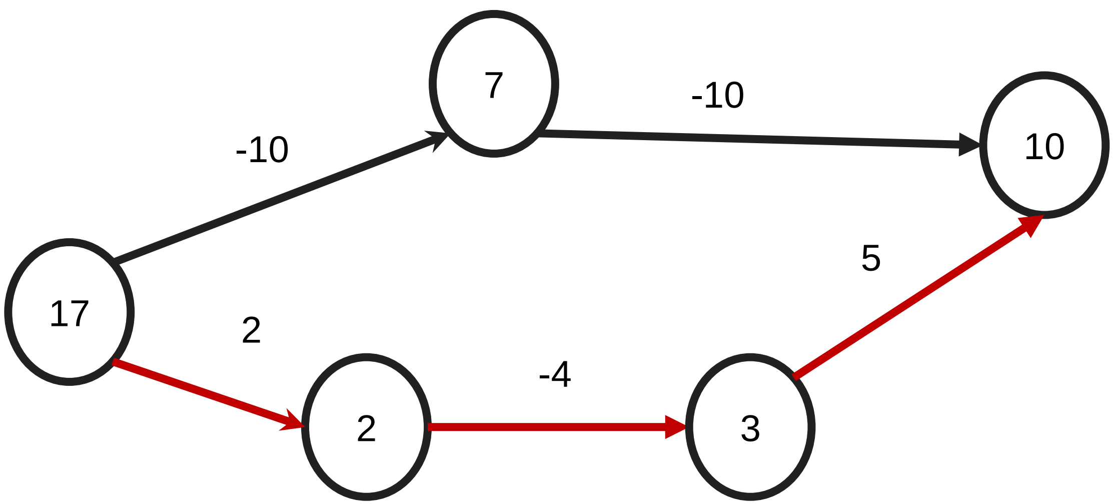
- We impose arbitrary ordering on vertices v_1, v_2, … v_n
- Each vertex gets a numeric id: V = {1,2,…,n}
- Now we have a sequence {1,2,…,n} of vertices

- Similar to knapsack problem, in each iteration k we will compute all shortest paths using only a subset of vertices {1,2,…k} as intermediate nodes on each shortest path

Subproblem

- V={1,2,…,n}
- We are allowed to use only {1,…,k}
- Each subproblem P(i, j, k) represents the cost of the shortest path from i to j using only the first 1…k vertices in the sequence

Example

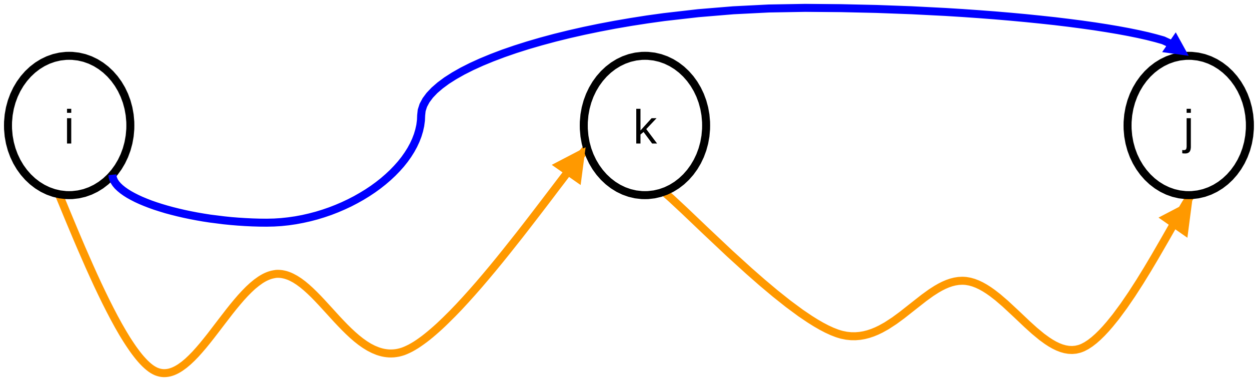


i=17, j= 10, k= 5
P(i,j,k) = 3

Optimal subproblems: intuition

When we allow the next k to be included as intermediate vertex on the path i~>j, we have the following choices:

- New vertex k is not included as part of the shortest path from i to j. The cost of the shortest path i~>j remains P(i,j,k-1)
- If vertex k is used to improve P(i,j,k-1), then k is internal to path P(i,j,k). In this case both P(i,k,k-1) and P(k,j,k-1) are shortest paths which use first k-1 vertices [which we already computed as subproblems for k-1]



We choose min between P(i, j, k-1) and [P(i, k, k-1) + P(k, j, k-1)]
All these min-cost paths are already computed in iteration k-1

Recurrence relation

- Input: directed graph G={V,E} – where vertices are numbered: V={1, …n}, and the cost matrix C with all edge costs.
- For each pair (i,j) ∈ V, let P(i, j, k) be the cost of the shortest path i~>j which uses only k first vertices from V as intermediate nodes on the path.

- Base case: no intermediate vertices are allowed

$$P(i,j,0) = \begin{cases} 0 \text{ if } i=j \\ C_{ij} \text{ if edge}(i,j) \in E \\ \infty \text{ otherwise} \end{cases}$$

Recurrence relation

- Input: directed graph G={V,E} – where vertices are numbered: V={1, …n}, and the cost matrix C with all edge costs.
- For each pair (i,j) ∈ V, let P(i, j, k) be the cost of the shortest path i~>j which uses only k first vertices from V as intermediate nodes on the path.

- Base case: no intermediate vertices are allowed

$$P(i,j,0) = \begin{cases} 0 \text{ if } i=j \\ C_{ij} \text{ if edge}(i,j) \in E \\ \infty \text{ otherwise} \end{cases}$$

- Recurrence: for any k, 0 < k ≤ n

$$P(i, j, k) = \min \begin{cases} P(i, j, k-1) \\ P(i, k, k-1) + P(k, j, k-1) \end{cases}$$

Pseudocode

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

```
A: = nxnxn 3D array indexed by k, i, and j
# base case
for each i ∈ V:
        for each j ∈ V:
                if i=j                  A[0, i, i] := 0
                else if (i, j) ∈ E      A[0, i, j] := Cij
                else                    A[0, i, j] := ∞

# DP table
for k from 1 to n:
        for i from 1 to n:
                for j from 1 to n:
                        A[k,i,j] = min A[k-1,i,j], A[k-1,i,k] + A[k-1,k,j]
return A[n]             # last matrix contains all-pair shortest path costs
```

Total n^3 subproblems with O(1) work per subproblem

Running time $O(n^3)$

Floyd-Warshall algorithm: notes

- **Negative cycles:**
  - To trust the results – we need to check that graph does not have negative cycles
  - If we scan the diagonal of the final matrix A[n], then all values A[n, i, i] must be 0.
  - If any of distances from node i to itself is < 0 – graph contains negative cycles

- **Space improvement:**
  - We do not have to store the entire 3D array to recover actual shortest path between a pair of vertices
  - It is enough for each pair of vertices (i, j) to store the max index of an internal node on the path from i to j: the last value of k which was used to improve the cost of i~>j
  - Knowing this vertex, we can recursively obtain shortest paths i~>k and k~j and recover the entire path

- **Undirected graphs:**
  - The Floyd-Warshall algorithm also works for undirected graphs, but only when there are no negative-weight edges

Results: All-Pairs Shortest Paths

For sparse graphs with non-negative edges: use n*Dijkstra

The best!

1. **Graphs with non-negative edge costs**:

n*Dijkstra (m log n) = O(nm log n) =
- $O(n^2 \log n)$ if $m=O(n)$ [sparse]
- $O(n^3 \log n)$ if $m=O(n^2)$ [dense]

2. **General graphs:**

n*Bellman-Ford (nm) = $O(n^2m)$ =
- $O(n^3)$ if $m=O(n)$ [sparse]
- $O(n^4)$ if $m=O(n^2)$ [dense]

1*Floyd-Warshall: $O(n^3)$

Can we do better for general graphs?

Motivation

- APSP = n*SSSP
- n*Dijkstra's algorithm = O(nm log n)

  for sparse graphs: $O(n^2 \log n)$

- **Idea:** use n*Dijkstra for general graphs
- **Problem:** we need to get rid of negative edge costs

Johnson's algorithm

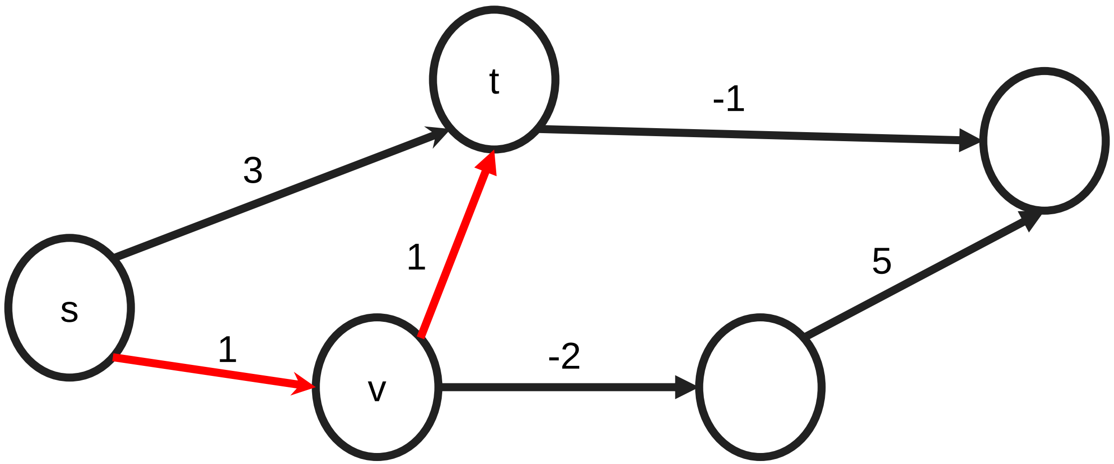
- Invoke Bellman-Ford SSSP: O(nm)
- Use n times Dijkstra: O(nm log n)
- Total running time: O(nm log n)

This will transform G into the graph with non-negative edge weights

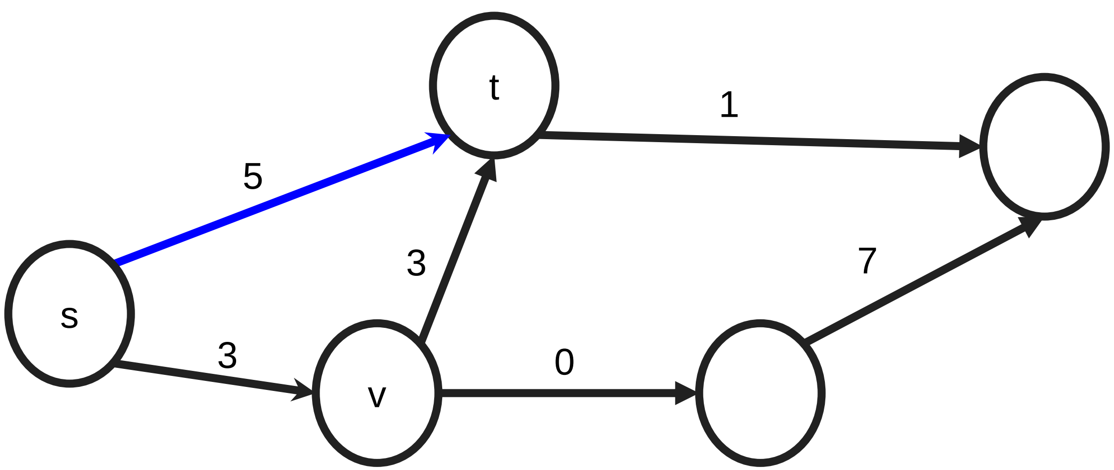
For general graphs!

Reweighting technique which does not work

- Natural instinct: add max negative cost to the weight of each edge
- However this does not preserve the original shortest paths



Before reweighting:
Shortest path s~>t is s-v-t
Most negative m=-2

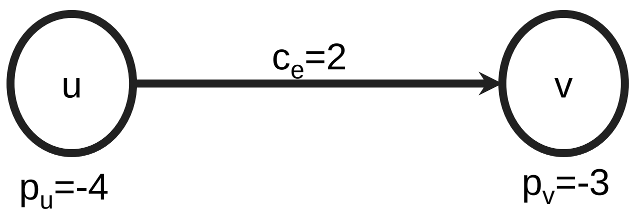


Add -m to each edge weight.
After reweighting:
Shortest path s~>t is s-t

Reweighting technique: vertex tokens

- Let G=(V,E) be a directed graph with general edge lengths (including negative)
- Fix a token p_v for each vertex v ∈ V (any real number)

- Transform the cost c_e of every edge e=(u,v) to $c_e' = c_e + p_u – p_v$



$c_e' = 2 + (-4) – (-3) = 1$

- Then the cost of any path P with original length L between two vertices s,t in G will be modified by exactly the same amount:

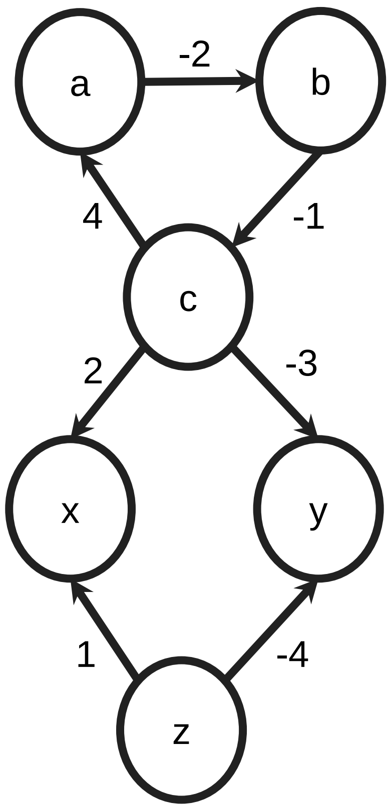
$$L' = L + p_u – p_v$$

$$L' = \sum_{all\ (u,v) \epsilon\ P} [c_e + p_u - p_v]$$

The tokens of all intermediate nodes cancel themselves and leave only the tokens of the source and the destination vertices

- Thus the relative lengths of different paths between s and t remain the same

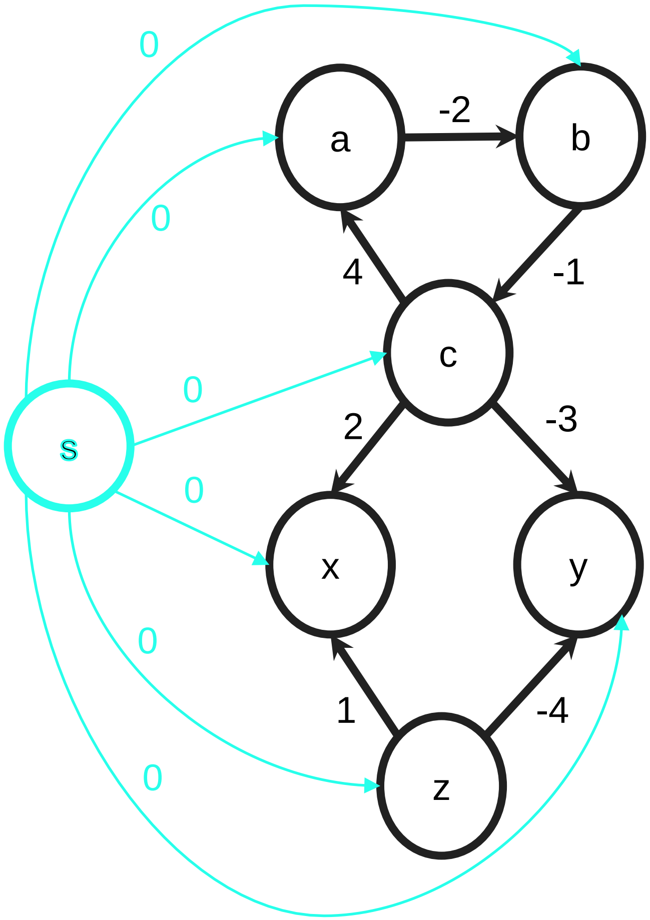
Computing magical vertex tokens



Sample graph with negative edge lengths but without negative cycles

- Compute magical vertex tokens running SSSP Bellman-Ford algorithm once

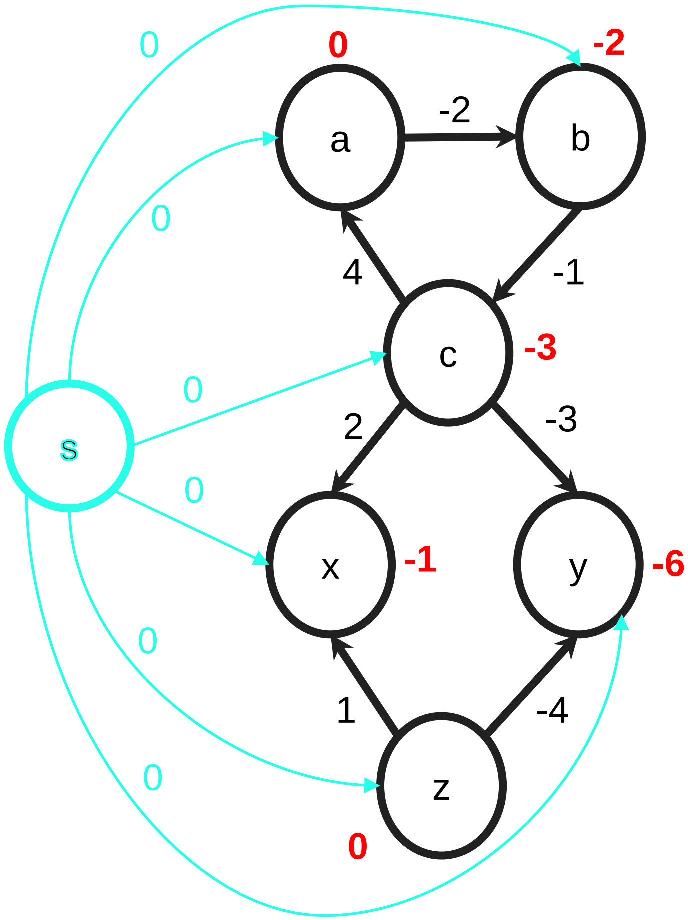
Computing magical vertex tokens



Adding artificial source vertex s with edges of cost 0 to every vertex in G

- Compute magical vertex tokens running SSSP Bellman-Ford algorithm once
- Add artificial source vertex s which has an outgoing edge of cost 0 to every vertex in G. Adding s will not change any shortest paths between original vertices of G, because s has no incoming edges

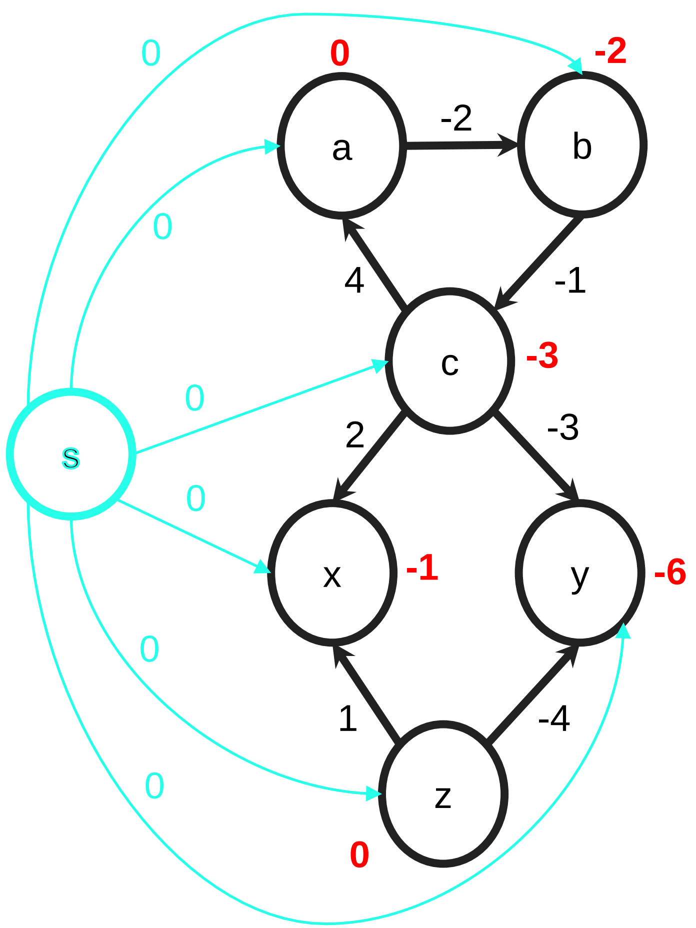
Computing magical vertex tokens



For each vertex: costs of single-source shortest paths from s

- Compute magical vertex tokens running SSSP Bellman-Ford algorithm once
- Add artificial source vertex s which has an outgoing edge of cost 0 to every vertex in G
- Run Bellman-Ford and compute the costs of shortest paths from s to every other vertex

Computing magical vertex tokens



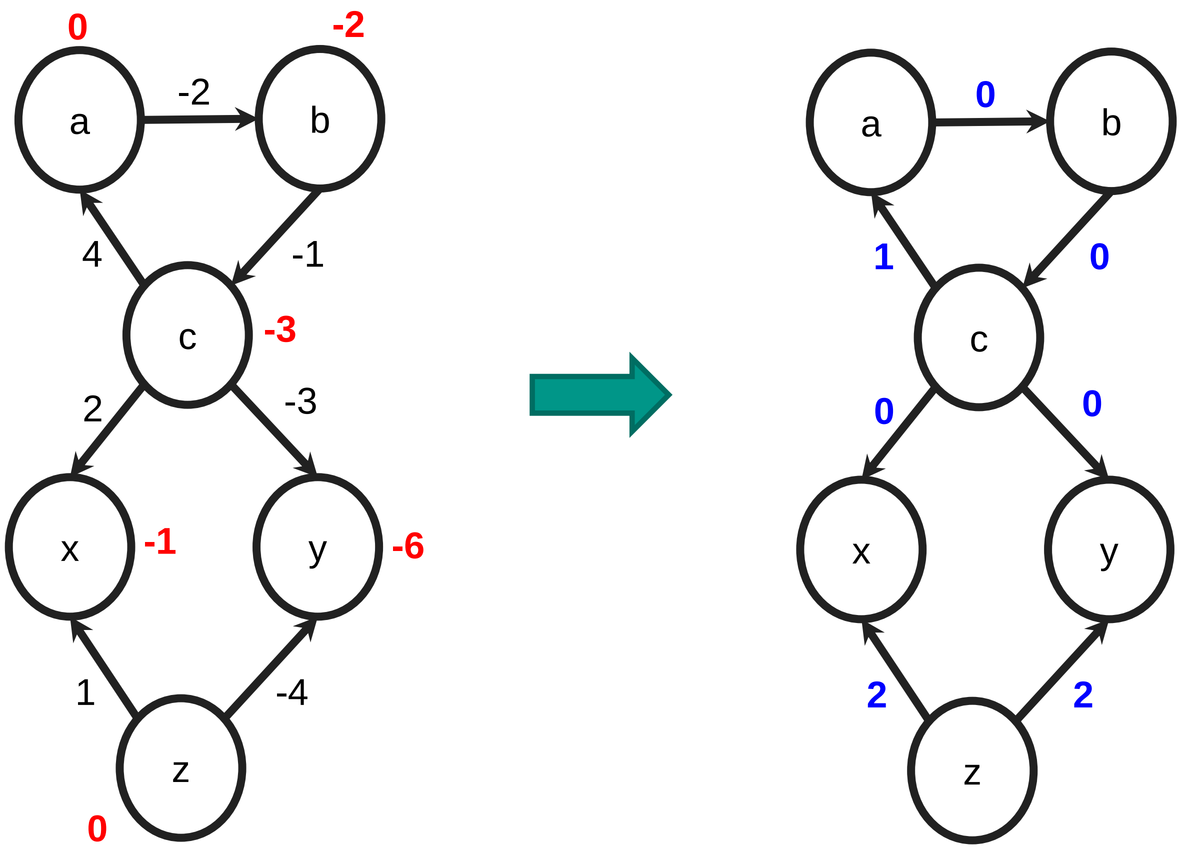
For each vertex: costs of single-source shortest paths from s

- Compute magical vertex tokens running SSSP Bellman-Ford algorithm once
- Add artificial source vertex s which has an outgoing edge of cost 0 to every vertex in G
- Run Bellman-Ford and compute the costs of shortest paths from s to every other vertex
- At the end - set p_v = cost of the shortest path s~>v

These are your magical vertex tokens, which will make the cost of each edge non-negative!

Transforming edges

- p_v = cost of a shortest path s~>v
- For every edge e=(u,v) new cost $c_e' = c_e + p_u – p_v$



Transformed graph with non-negative edge costs:
ready to run n*Dijkstra to compute all-pair shortest paths

Johnson's algorithm

- Convert G(V,E) into G' by adding a new vertex s and n edges (s,v) of cost 0 to every vertex v ∈ V

- Run Bellman-Ford (G' with source s) [if it reports a negative-cost cycle – halt]

- For each v ∈ V define pv = cost of the shortest path s~>v in G'
  For each edge e=(u,v) ∈ E, define new cost $c_e' = c_e + p_u - p_v$

- Run Dijkstra n times on G using new edge costs and starting from every vertex v ∈ V

- Extract the cost of the original path for each pair of vertices

easy?
Think how

Reduction of the APSS problem for general graph to:
1 SSSP for general graphs + n SSSP for graphs with non-negative edge costs

Johnson's algorithm: running time

- Convert G(V,E) into G' by adding a new vertex s and n edges (s,v) of cost 0 to every vertex v ∊ V — O(n)
- Run Bellman-Ford (G' with source s) [if it reports a negative-cost cycle – halt] — O(nm)
- For each v ∊ V define pv = cost of the shortest path s~>v in G'
  For each edge e=(u,v) ∊ E, define new cost $c_e' = c_e + p_u - p_v$ — O(m)
- Run Dijkstra n times on G using new edge costs and starting from every vertex v ∊ V — n*O(m log n)
- Extract the cost of the original path for each pair of vertices — $O(n^2)$

O(mn log n)

Much better than $O(n^3)$ Floyd-Warshall for sparse graphs

Johnson's algorithm: correctness

- We have already proven that using tokens of each vertex to reweigh edges does not change the order of paths u~>v: the shortest path remains the shortest even after reweighting: see Reweighting technique slide

- What remains is to prove the following:

Lemma

For every edge e=(u,v) of G, the reweighted edge cost $c_e' = c_e + p_u - p_v$ is non-negative.

Lemma

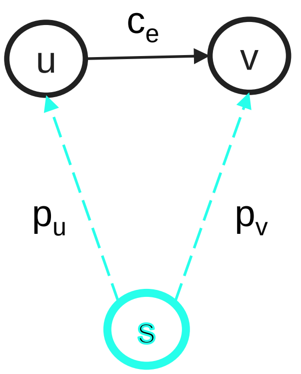
For every edge e=(u,v) of G, the reweighted edge cost $c_e' = c_e + p_u - p_v$ is non-negative.

Proof

- Let (u,v) be an arbitrary pair of vertices in G connected by an edge e u→v with cost c_e.
- By construction,

  p_u = cost of a shortest path from s to u

  p_v = cost of a shortest path from s to v



- If p_u is the cost of a shortest path s~>u
- Then $p_u + c_e$ is the length of some path from s to v. This may be a shortest path from s to v, but there could be an even shorter path from s to v which does not pass through vertex u.
- Hence, $p_u + c_e \geq p_v$
- Therefore, $c_e' = c_e + p_u - p_v \geq 0$ ∎