Walchand College of Engineering
(Government Aided Autonomous Institute)
Vishrambag, Sangli. 416415

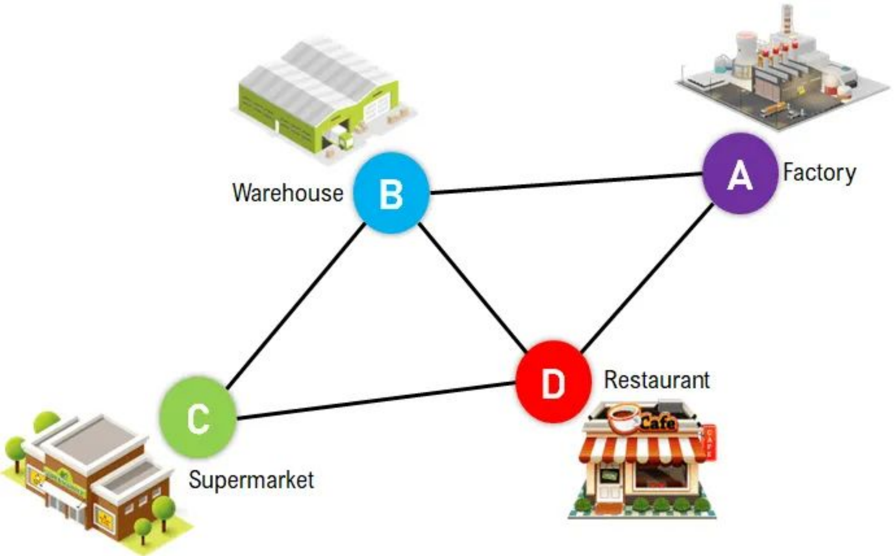
Computer Algorithms

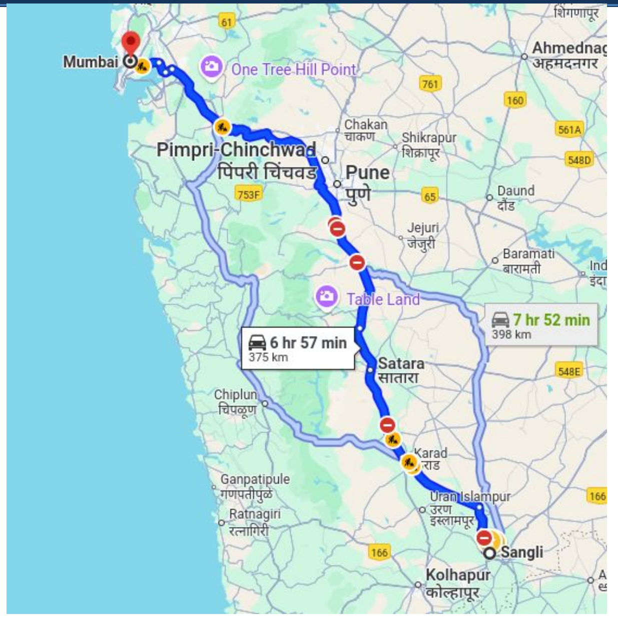
Shortest Path Algorithms

24-25

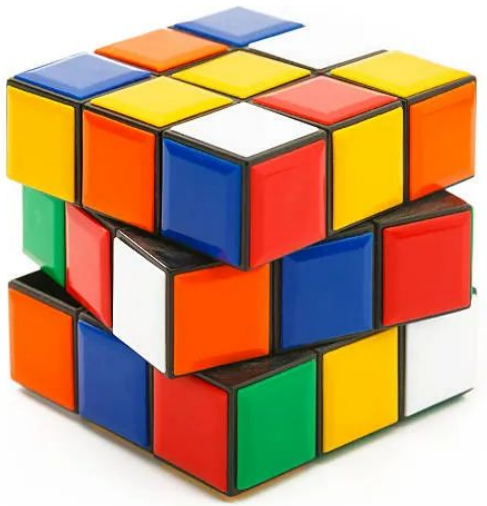
D B Kulkarni
Information Technology Department

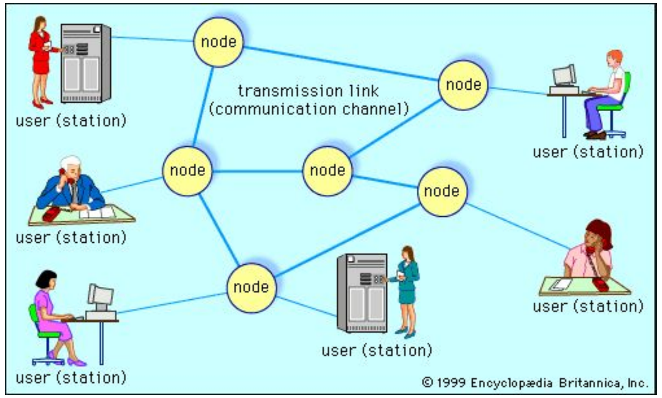
Shortest path: Applications



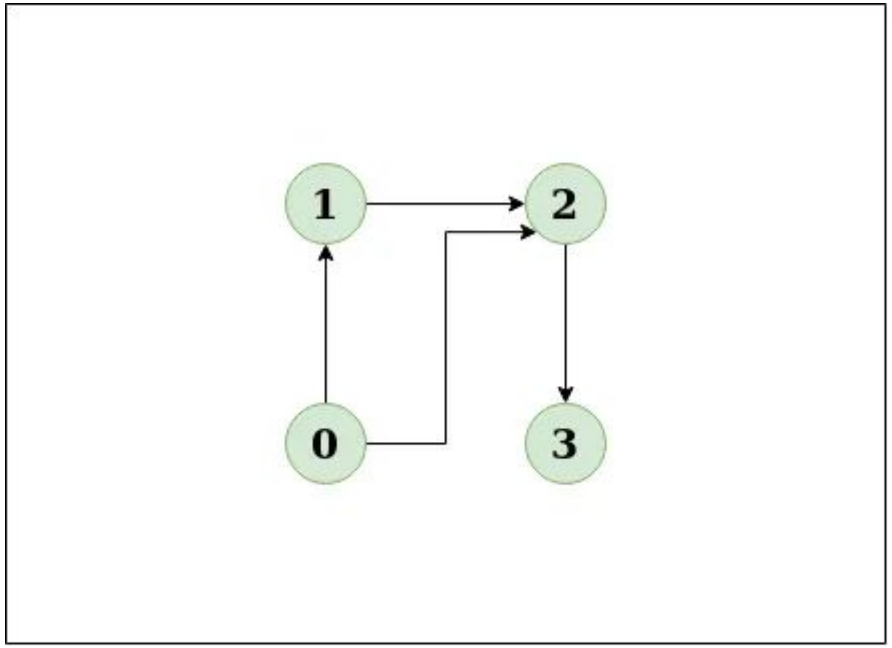


- ➢ Automatic direction finding between physical locations, such as driving directions on web mapping websites like Google Maps.
- ➢ Solving puzzles, like finding the minimum number of moves to solve a Rubik's Cube.
- ➢ Minimizing delay and maximizing bandwidth in telecommunications networks



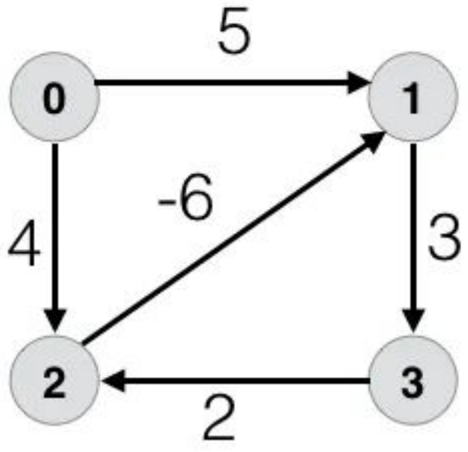


Shortest path: Definitions



Given weighted graph G

- ➢ Path: Weighted
  - ○ Shortest path (SP)
- ➢ Cycle (2,1,3,1)
- ➢ Negative weight w(2,1)
- ➢ Negative weight Cycle



Terminologies/ notations

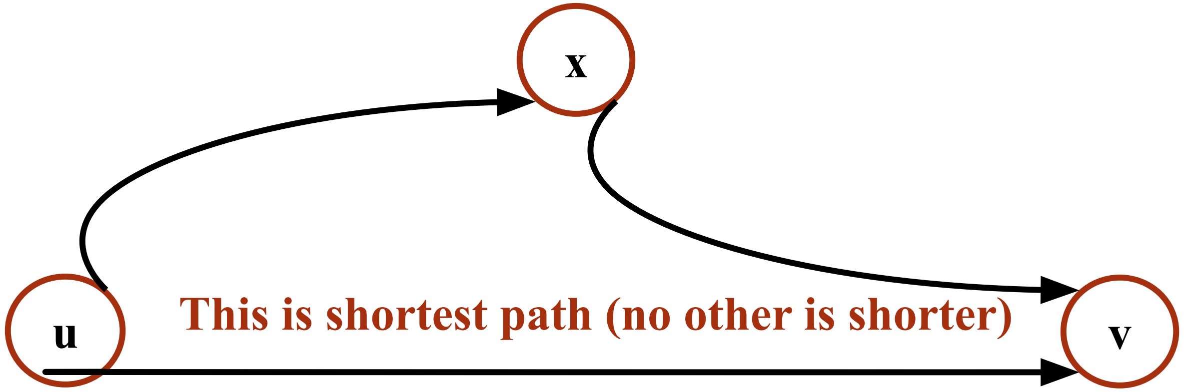
Weight- ***w(u,v)***, Path-***p(u→v)***,

Weight of path- ***w(p)=w(p(u→v))***

Weight of Shortest path (P)- ***δ(u,v)= min(w(p):u~v)***

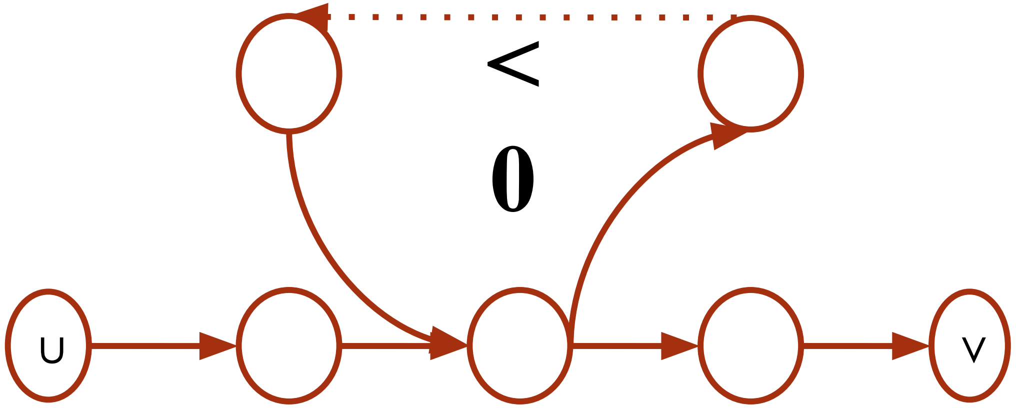
Properties of SP

- Triangle inequality
  δ(u,v)= δ(u,x)+ δ(x,v)
- Upper bound
  v.d≥δ(s,v)
- No path
- If there is no path from s to v , ***v.d=δ(s,v)=∞***
- Convergence
- Path relaxation
- Predecessor graph



Complexities

- Edge weights can be negative
- Negative weight cycle



Shortest path: Variants

- **S**ingle **P**air **S**hortest **P**ath (**SPSP**)
- **S**ingle **S**ource **S**hortest **P**ath **(SSSP)**
- **S**ingle **D**estinations **S**hortest **P**ath (SDSP)
- **A**ll **P**air **S**hortest **P**ath (**APSP)**

Can we use SPSP to solve other SP variants?

Shortest path

- Shortest path: Given weighted directed graph, G(V,E) find min weight path from given source vertex u to another vertex v

- The weight w of path p^1 from u to v, $w(p^1_{u,v})=v_u,v_1,v_2.....v_{k,}v_v$ is the sum of weight of constituent edges

  $w(p(v_0\text{->}v_k))=\Sigma^k_{i=0}\ w(v_{i-1},v_i)$

- shortest path weight δ(u,v) to be the shortest path from u to v defined as

  $$\delta(u,v)= \begin{cases} \min\{w(p): u \sim v\} \\ \infty \quad Otherwise \end{cases}$$

SP : Solution formulation

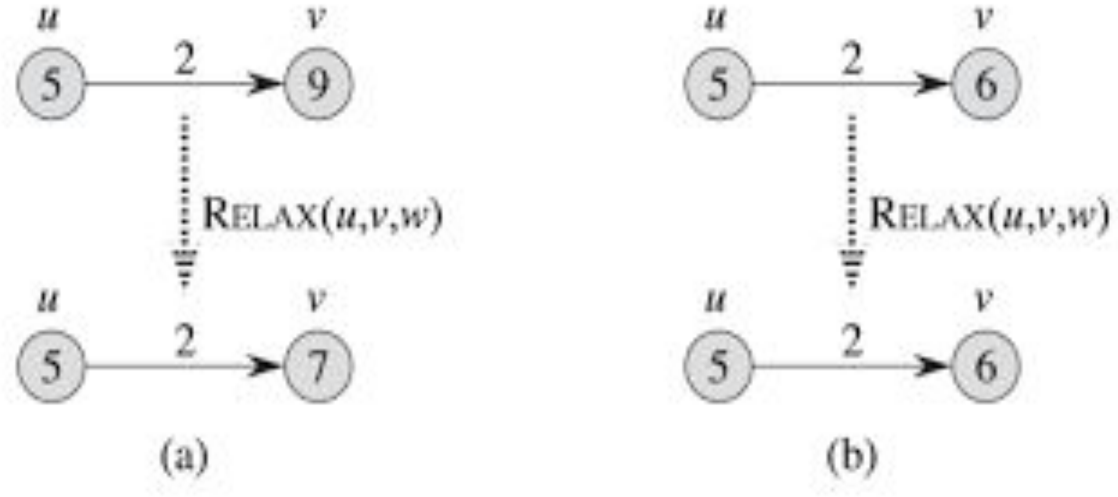
- For graph G=(V,E), S is the starting vertex , for each vertex vεV we maintain
  - predecessor vertex v.∏, which can be other vertex or NIL.
  - Shortest path estimate v.d, path length from s to v, is upper bound on weight of shortest path from source S to v, which is ∞ at the beginning.
  - Initialization: For each vertex v εG.V
    - v.d= ∞
    - v.∏=NIL
    - S.d=0
- (RELAX) Relaxing the edge (u,v) is testing whether we can improve the shortest path from S to v, found so far by going through u
  - if so updating v.d and V. ∏

Relaxation

- Relaxing the edge (u,v) is testing whether we can improve the shortest path from S to v, found so far by going through u
  - if so updating v.d and V. ∏

- RELAX (u,v,w)
  - if v.d>u.d+w(u,v)
  - { v.d=u.d+w(u,v)

    V. ∏=u }



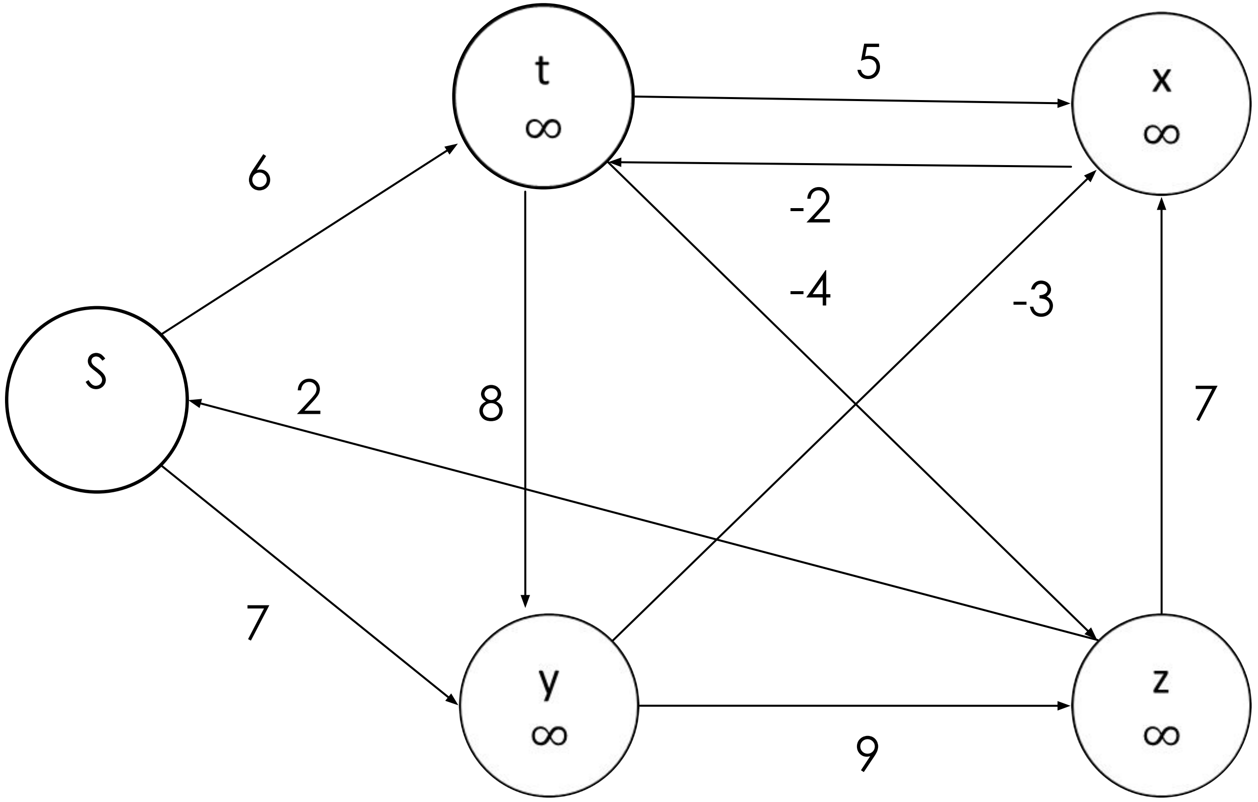
Bellman Ford (BF) Algorithm (SSSP)

- computes shortest paths from a single source vertex to all of the other vertices in a weighted digraph
  - If negative weight cycle is reachable, it exits
- Steps
  - Step 1: initialize
    - For all vertices v.∏=NIL, v.d= ∞, s.d=0
  - Step 2: relax edges repeatedly
    - For (#vertices-1)
      - For all edges - RELAX
  - Step 3: check for negative-weight cycles
    - For each edges <u,v>- if u.d+w <v.d then graph contains negative weight cycle

Bellman Ford Algorithm: Example

Edges : (t,x) (t,y) (t,z) (x,t) (y,x) (y,z) (z,x) (z,s) (s,t) (s,y)

Vetrices:5



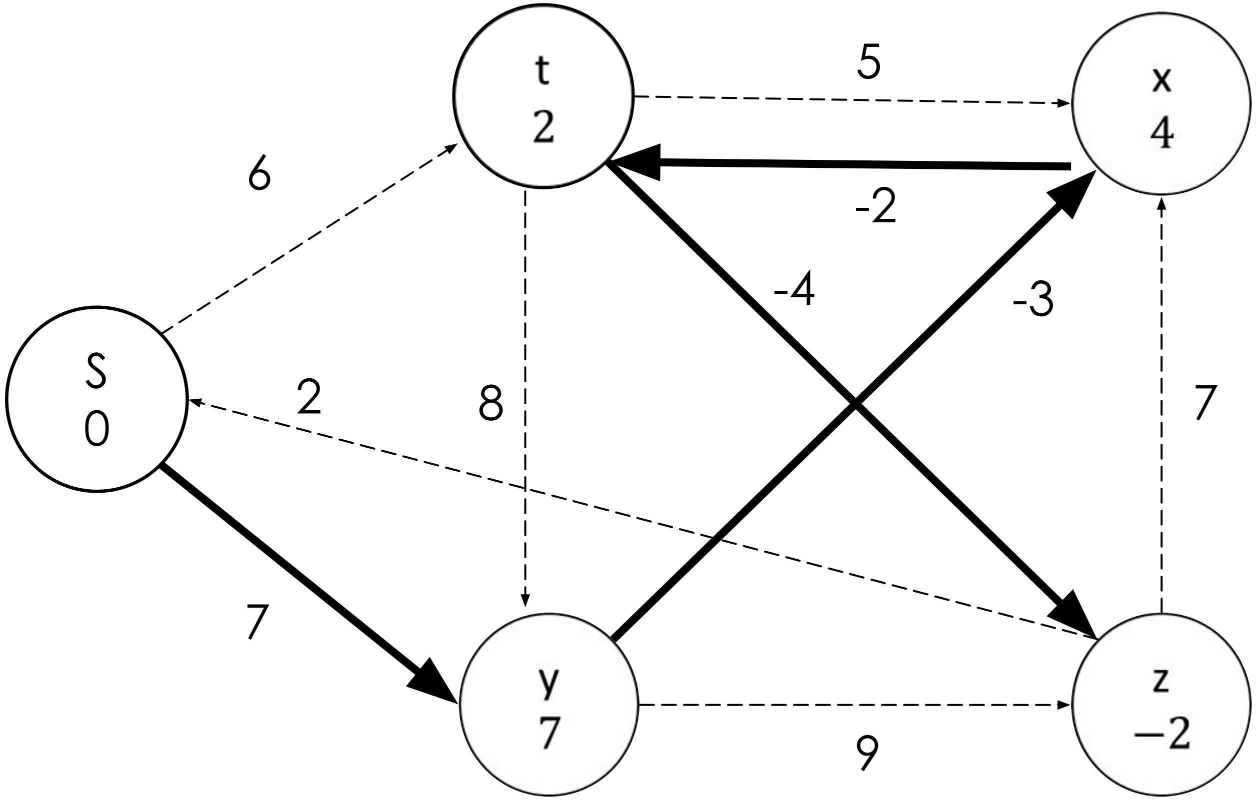
Steps

Step 1: Initialize graph
S.d=0
For all vertices v.d=∞, v.π=NIL

Step 2: Relax all edges repeatedly for (#vertices -1) times

Step 3: Check for negative weight cycle
for each edge <u,v>
if u.d+w <v.d
then -ve wt cycle present

Bellman Ford Algorithm: Final status



Steps

Step 1: Initialize graph
S.d=0
For all vertices v.d=∞, v.π=NIL

Step 2: Relax all edges repeatedly for (#vertices -1) times

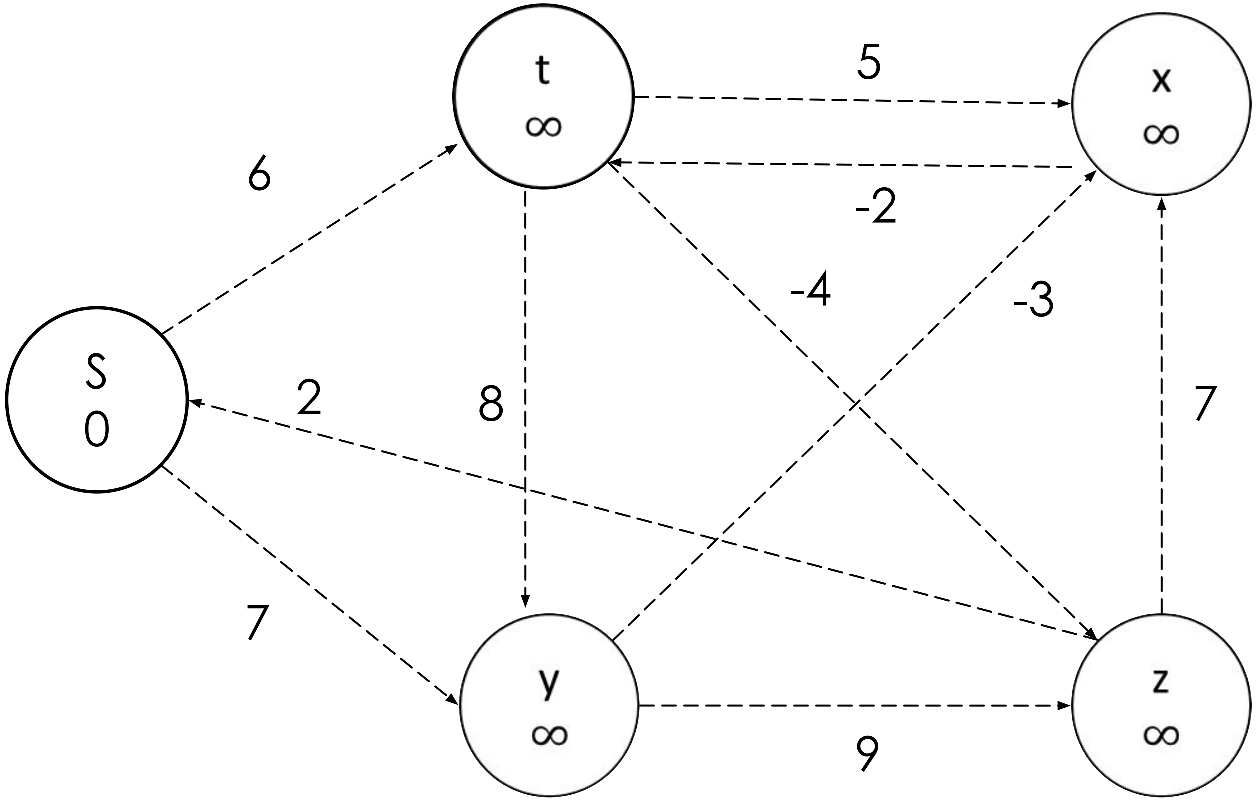
Step 3: Check for negative weight cycle
for each edge <u,v>
if u.d+w <v.d
then -ve wt cycle present

Bellman Ford Algorithm: Example

Edges :

1. (t,x)
2. (t,y)
3. (t,z)
4. (x,t)
5. (y,x)
6. (y,z)
7. (z,x)
8. (z,s)
9. (s,t)
10. (s,y)

Step : 1 Initialization



Vetrices:5

Steps

Step 1: Initialize graph
S.d=0
For all vertices v.d=∞, v.π=NIL

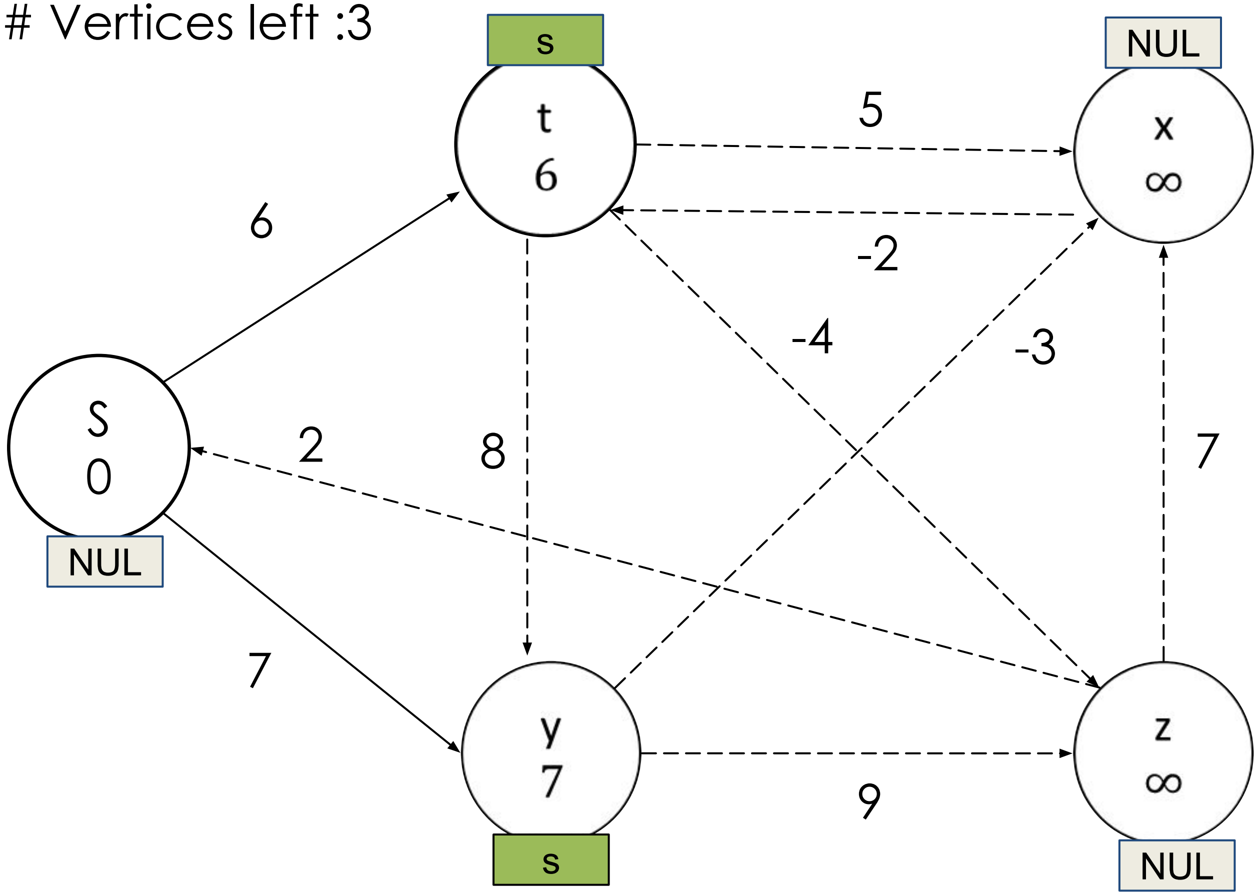
Step 2: Relax all edges repeatedly for (#vertices -1) times

Step 3: Check for negative weight cycle

for each edge <u,v>
if u.d+w <v.d
then -ve wt cycle present

Step 2: Iteration : 1 (Vertices-4)

Edges : (t,x) (t,y) (t,z) (x,t) (y,x) (y,z) (z,x) (z,s) **(s,t)(s,y)**
Vertices left :3



Steps

Step 1: Initialize graph
S.d=0
For all vertices v.d=∞, v.π=NIL

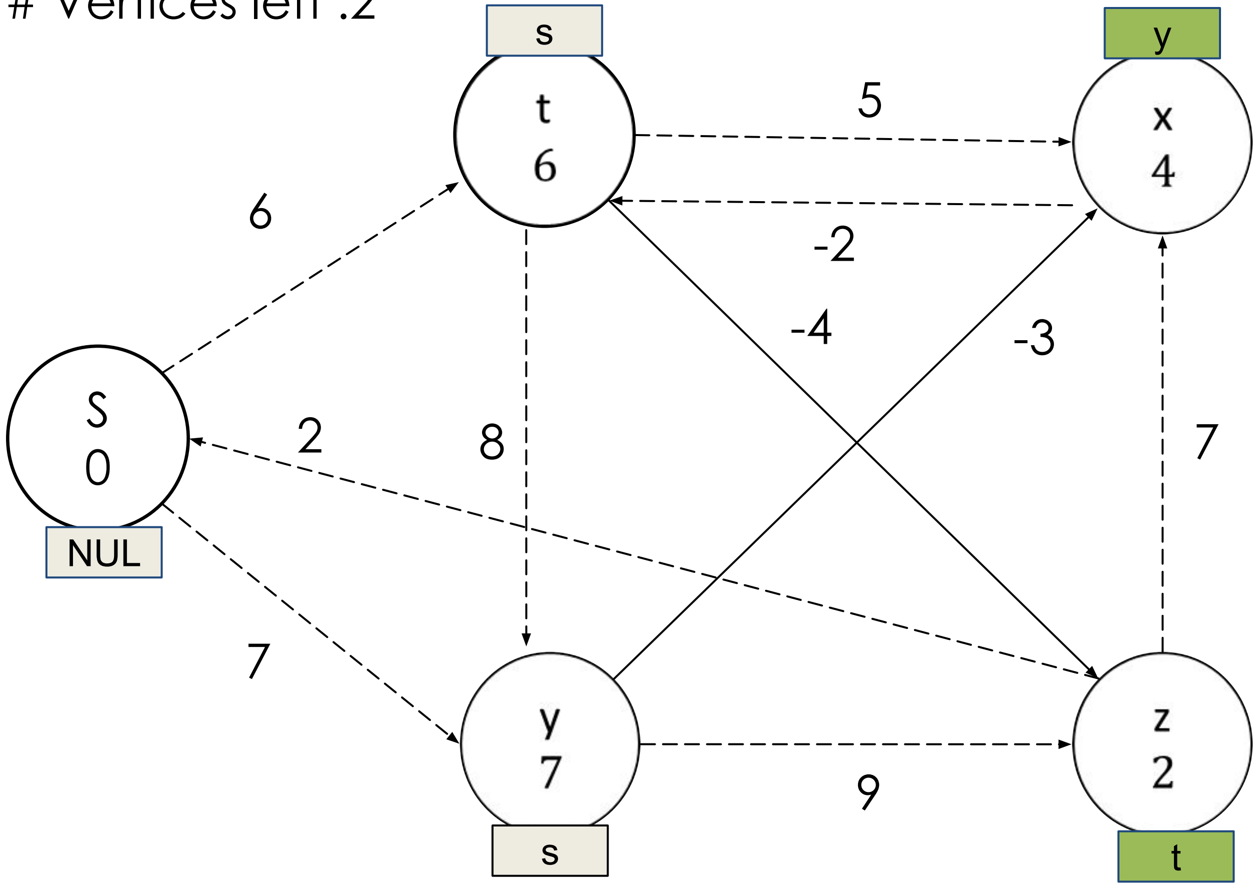
Step 2: Relax all edges repeatedly for (#vertices -1) times

Step 3: Check for negative weight cycle
for each edge <u,v>
if u.d+w <v.d
then -ve wt cycle present

Step 2: Iteration : 2 (Vertices-3)

Edges : (t,x) (t,y) **(t,z)** (x,t) **(y,x)** (y,z) (z,x) (z,s) (s,t) (s,y)

Vertices left :2



Steps

Step 1: Initialize graph

S.d=0

For all vertices v.d=∞, v.π=NIL

Step 2: Relax all edges repeatedly for (#vertices -1) times

Step 3: Check for negative weight cycle

for each edge <u,v>

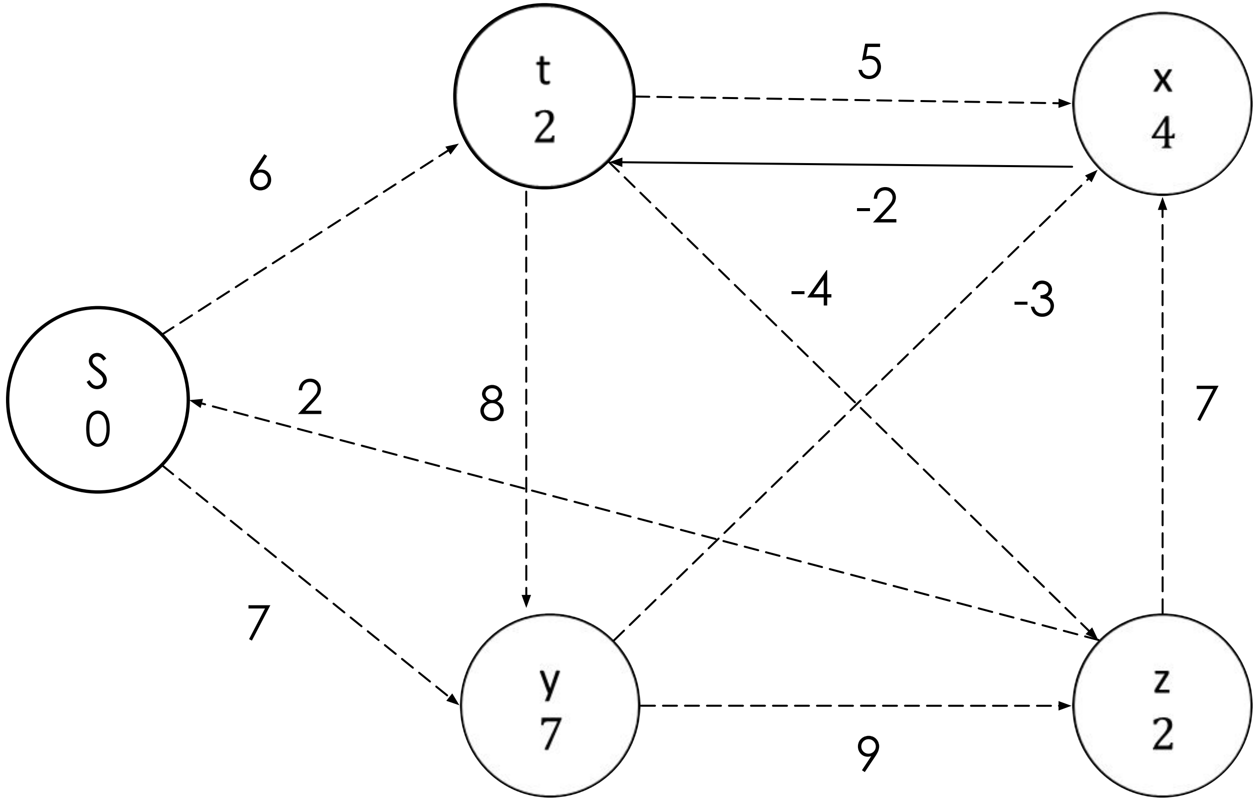
if u.d+w <v.d

then -ve wt cycle present

Step 2: Iteration : 3 (Vertices-2)

Edges : (t,x) (t,y) (t,z) **(x,t)** (y,x) (y,z) (z,x) (z,s) (s,t) (s,y)

Vertices left :1



Steps

Step 1: Initialize graph
S.d=0
For all vertices v.d=∞, v.π=NIL

Step 2: Relax all edges repeatedly for (#vertices -1) times

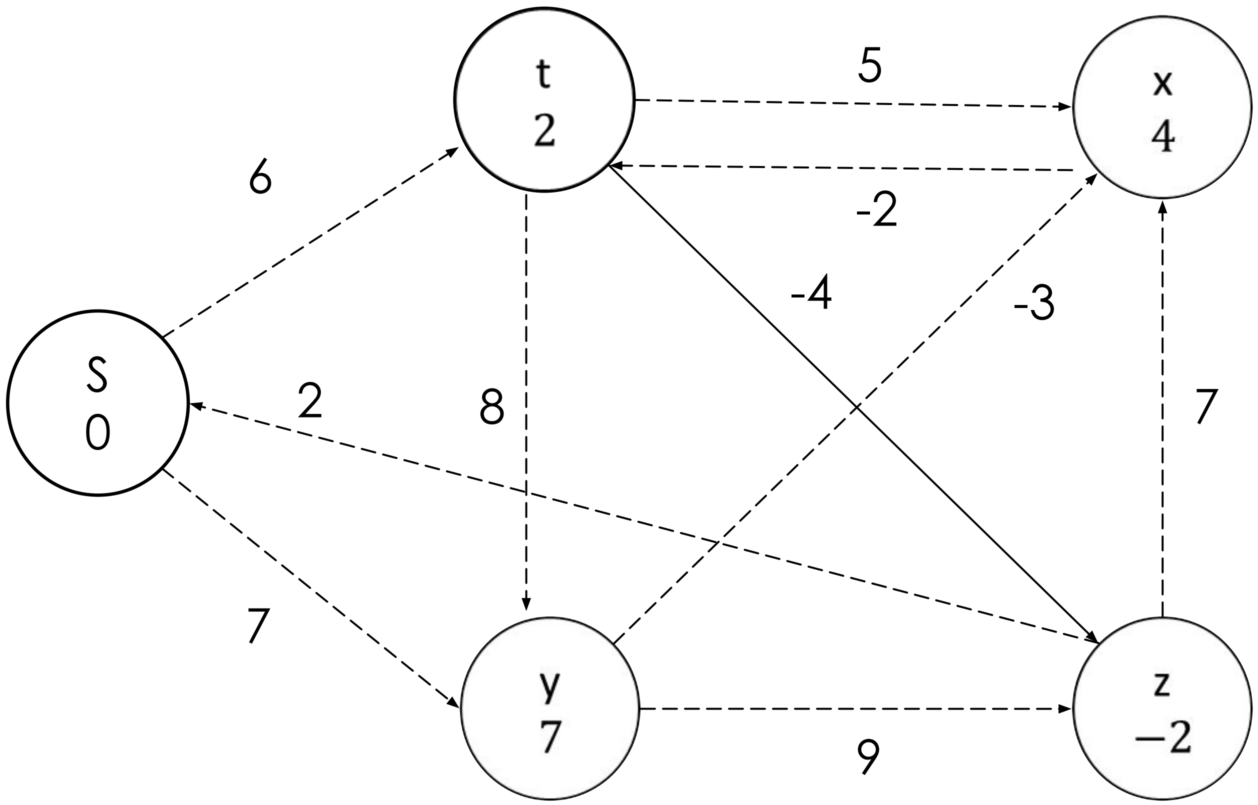
Step 3: Check for negative weight cycle
for each edge <u,v>
if u.d+w <v.d
then -ve wt cycle present

Bellman Ford Algorithm: Example

Step 2: Iteration : 4 (Vertices-1)

Edges : (t,x) (t,y) **(t,z)** (x,t) (y,x) (y,z) (z,x) (z,s) (s,t) (s,y)

Vertices left :0



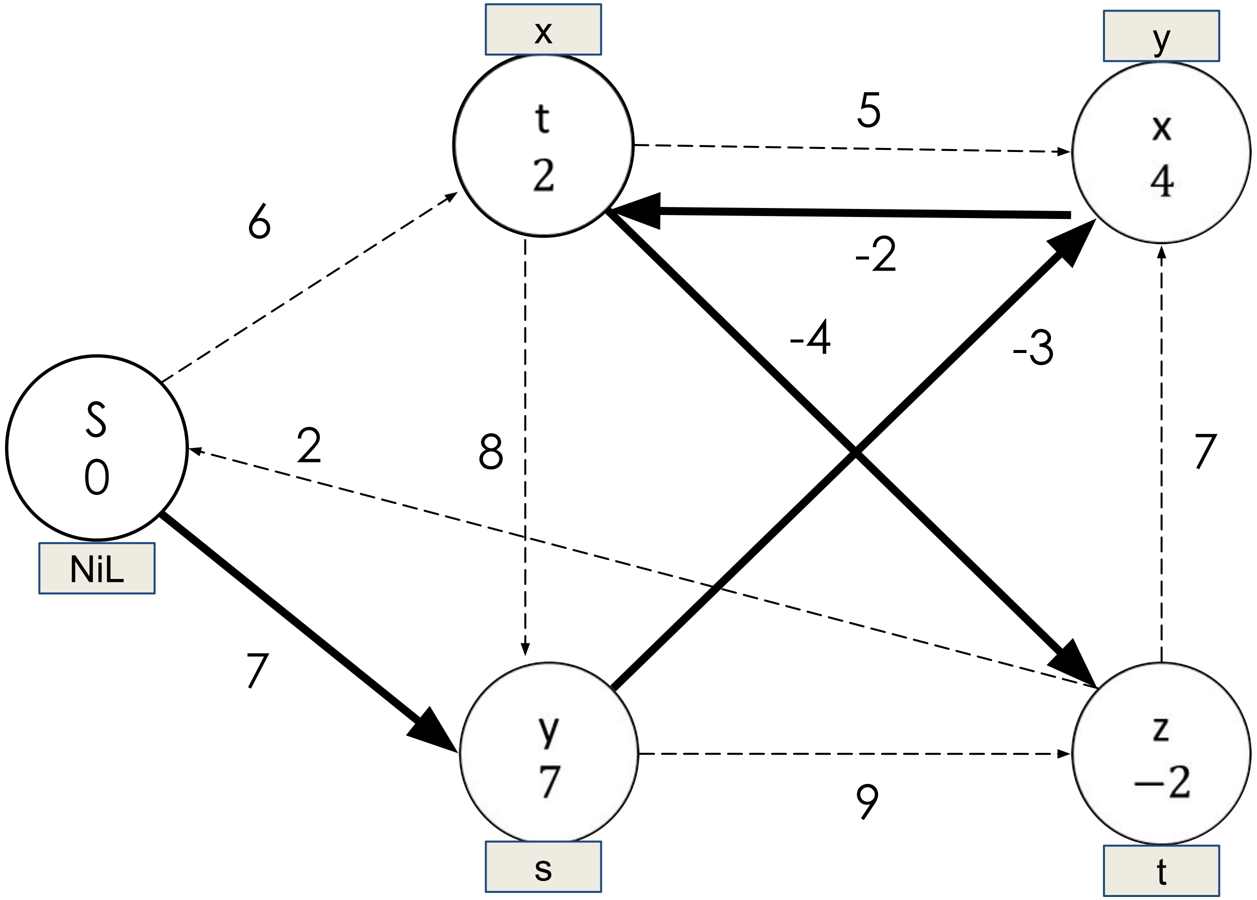
Steps

Step 1: Initialize graph
S.d=0
For all vertices v.d=∞, v.π=NIL

Step 2: Relax all edges repeatedly for (#vertices -1) times

Step 3: Check for negative weight cycle
for each edge <u,v>
if u.d+w <v.d
then -ve wt cycle present

Bellman Ford Algorithm: Example



Steps

Step 1: Initialize graph
S.d=0
For all vertices v.d=∞, v.π=NIL

Step 2: Relax all edges repeatedly for (#vertices -1) times

Step 3: Check for negative weight cycle
for each edge <u,v>
if u.d+w <v.d
then -ve wt cycle present

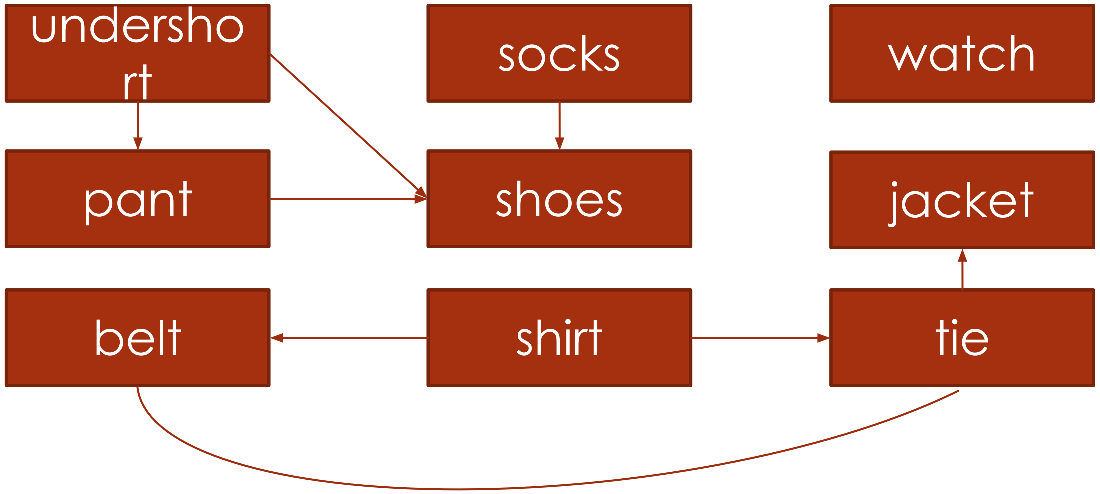
Bellman Ford Algorithm: Analysis

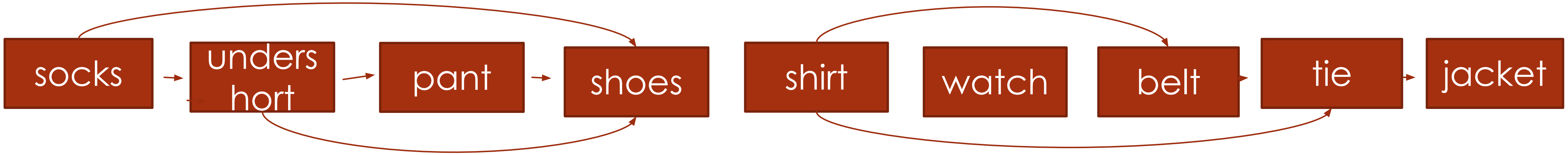
- Complexity
  V*E

- Order in which the edges are processed affects how quickly the algorithm converges

SSSP in DAG

- If edge relaxation is carried out in some order (as per topological sort order of the vertices), SP can be computed in (V+E) time
- Topological sort: For DAG , its linear ordering of all vertices such that for edge (u,v)u appears before v in ordering.



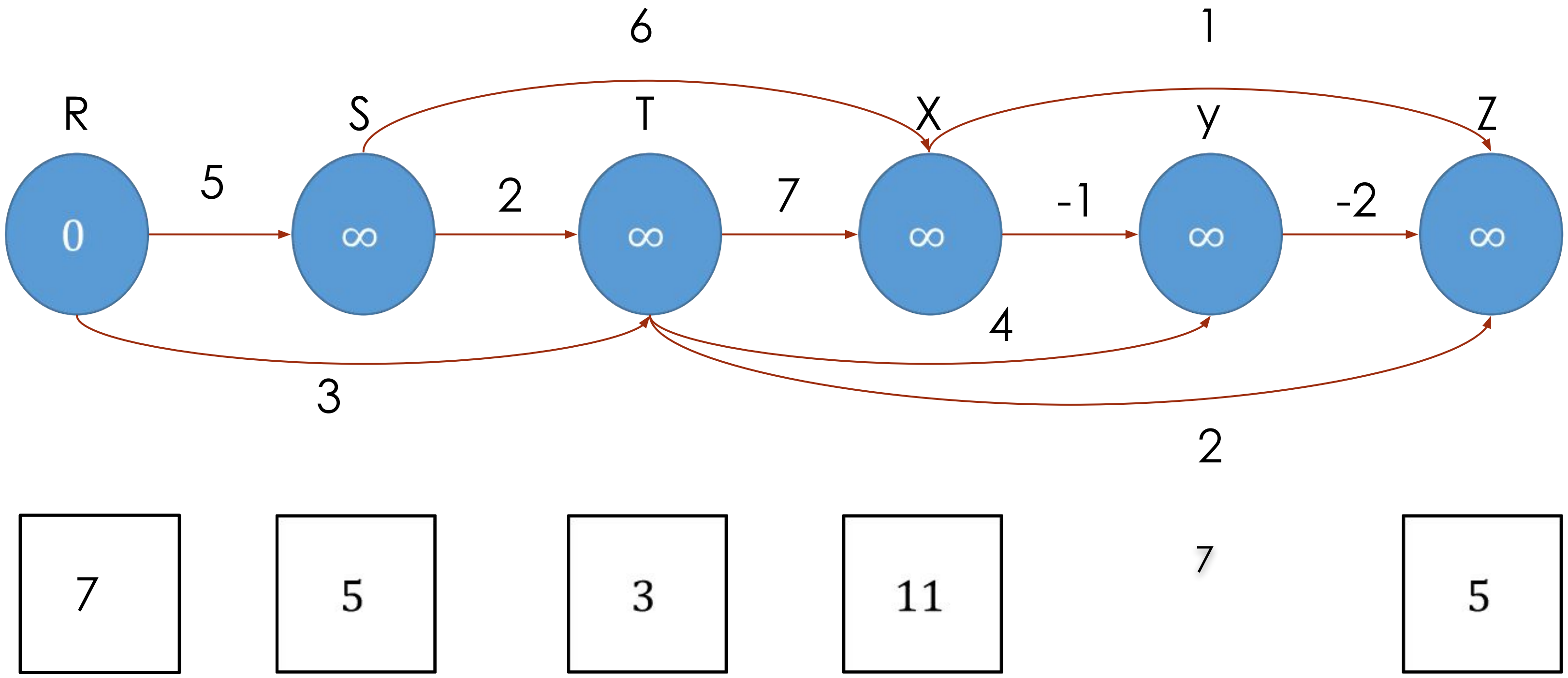


SSSP in DAG: Example

Algorithm

For each vertex u taken in topological order

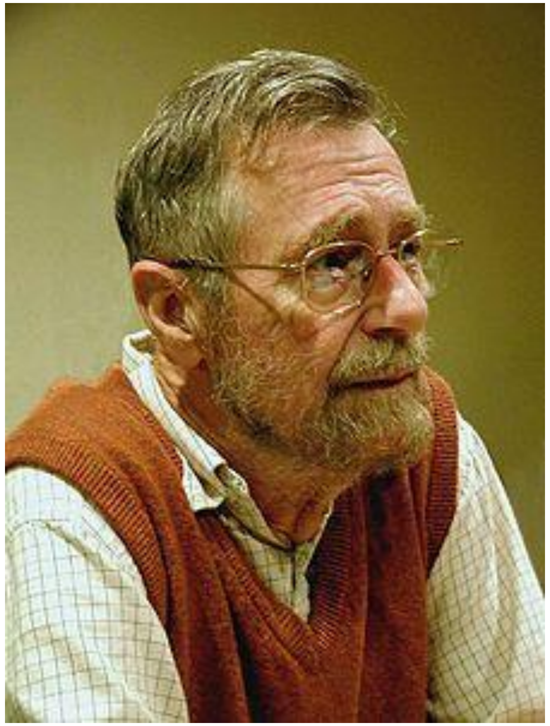
- For each vertex vεG Adj[u]
- RELAX(u,v,w)
- Complexity- V+E (home work)



EDSGER WYBE DIJKSTRA

May 11, 1930 – August 6, 2002

"Computer Science is no more about computers than astronomy is about telescopes."



Dijkstra's Algorithm

- If no negative edge weights, we can beat BF
  - No negative weight cycle?

  - Use a priority queue keyed on d[v]

SSSP: DIJKSTRA'S ALGORITHM

- ➢ Works on both directed and undirected graphs.
- ➢ However, all edges must have nonnegative weights.

Approach: Greedy

- ➢ Similar to breadth-first search
  - ■ Grow a tree gradually, advancing from vertices taken from a queue
- ➢ Also similar to Prim's algorithm for MST

Input: Weighted graph ***G={E,V}*** and source vertex $v \in V$, such that all edge weights are nonnegative

Output: Lengths of shortest paths

Dijkstra's Algorithm - pseudocode

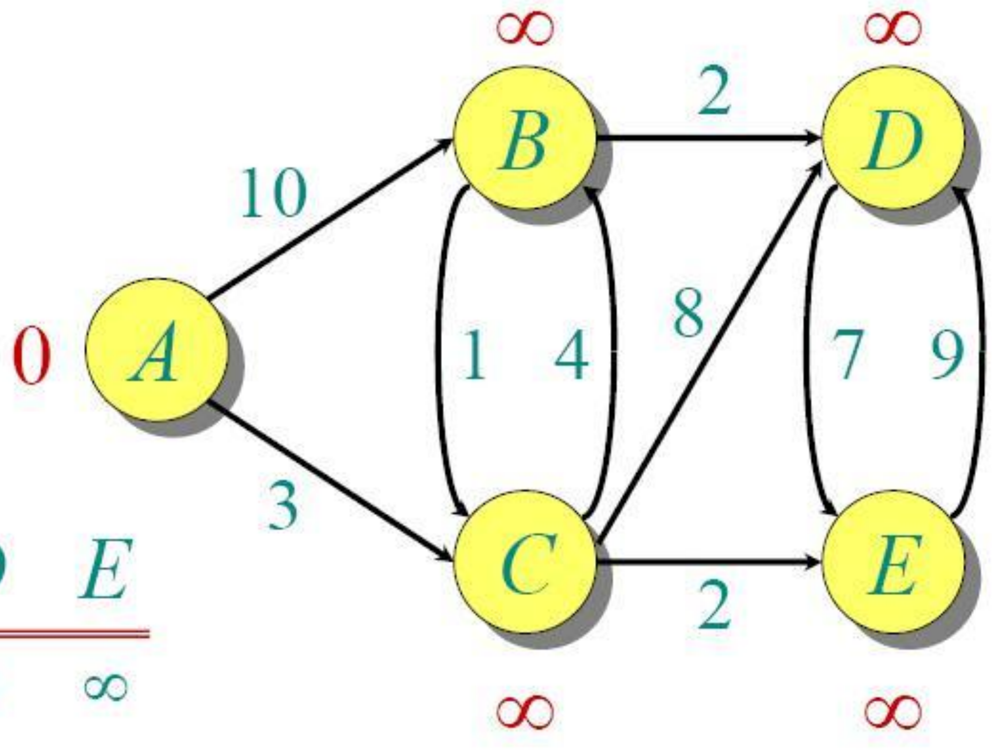
s- starting vertex (root) S- Set containing vertices

$S \leftarrow \varnothing$	(S, the set of visited vertices is initially empty)
$Q \leftarrow V$	(Q, the queue initially contains all vertices)
$s.d \leftarrow 0$	(distance to source vertex is zero)
for all $v \in V-\{s\}$	
do $v.d \leftarrow \infty$	(set all other distances to infinity)

while $Q \neq \varnothing$	(while the queue is not empty)
do $u \leftarrow$ mindistance(Q,dist)	(select the element of Q with the min. distance)
$S \leftarrow S \cup \{u\}$	(add u to list of visited vertices)
for all $v \in$ neighbors[u]	
do if $v.d > u.d + w(u, v)$	(if new shortest path found)
then $v.d \leftarrow u.d + w(u, v)$	(set new value of shortest path)
return v.d	

Dijkstra's Algorithm: Animated Example

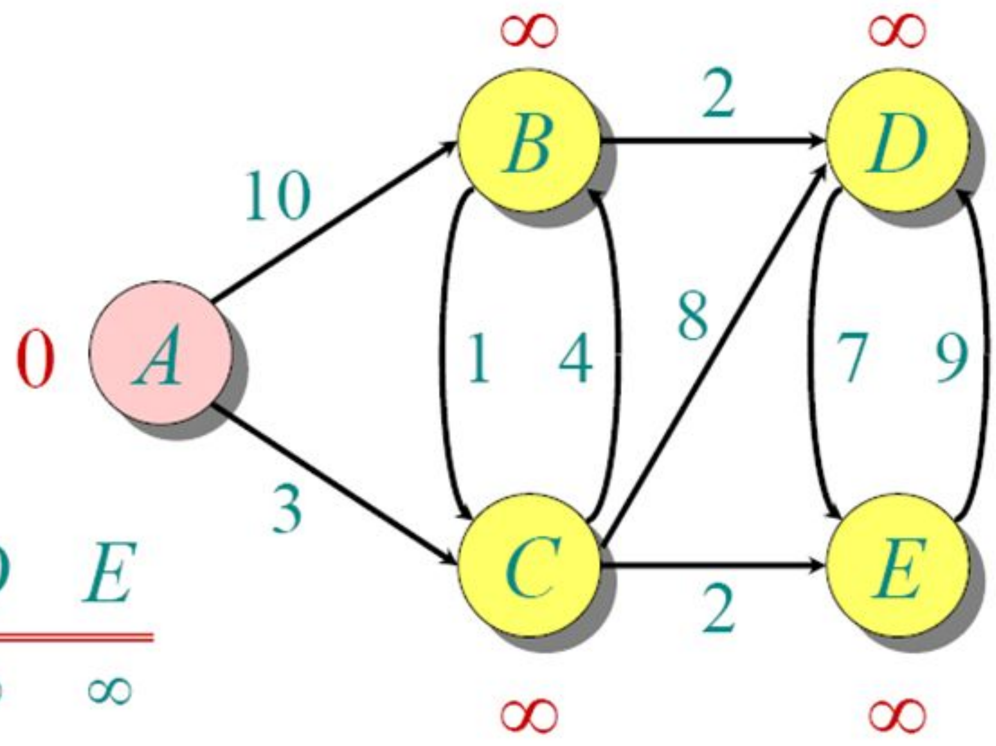
Initialize:



Q:	A	B	C	D	E
	0	∞	∞	∞	∞

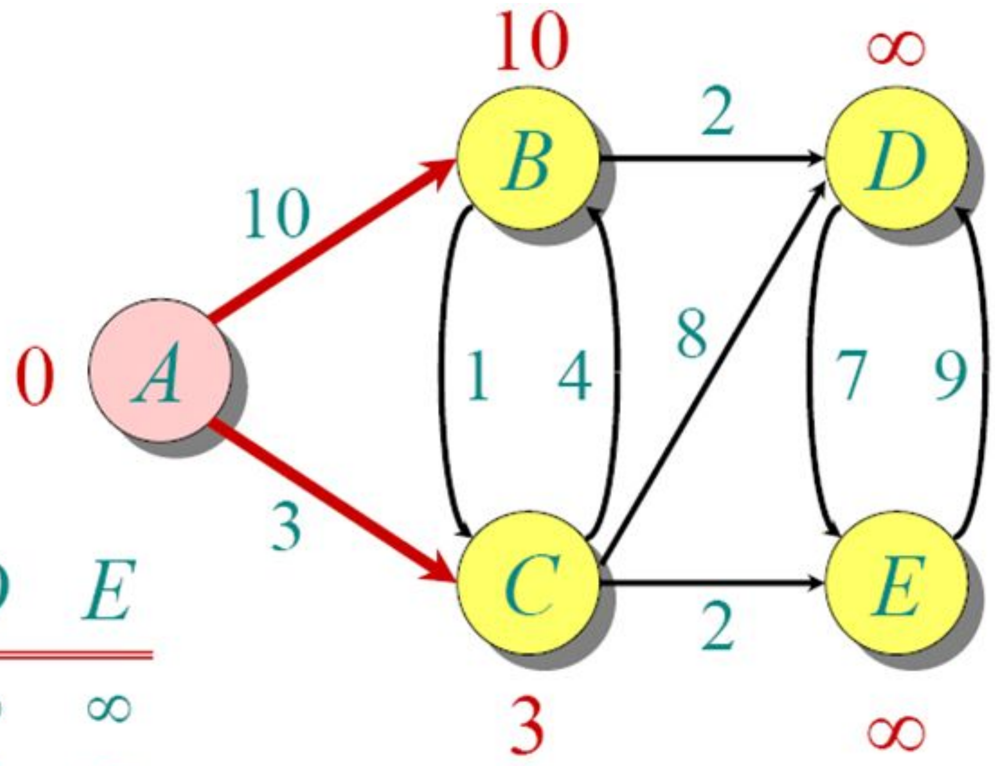
S: {}

Dijkstra’s Algorithm: Animated Example



Q:	A	B	C	D	E
	0	∞	∞	∞	∞

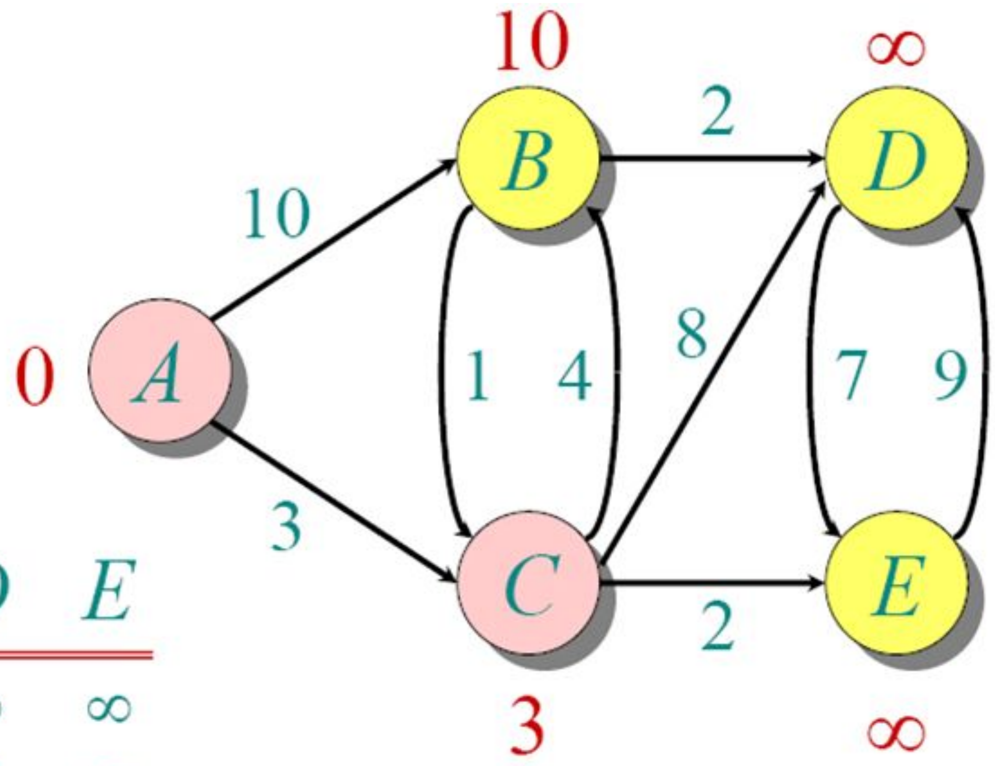
Dijkstra’s Algorithm: Animated Example



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞

S: { A }

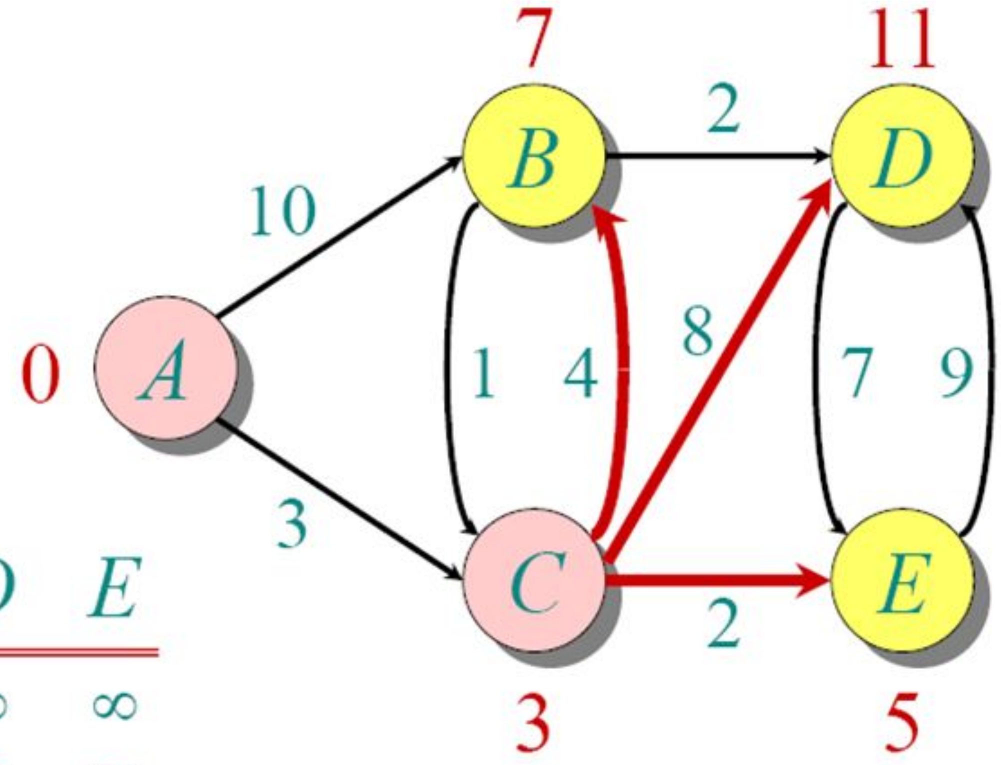
Dijkstra’s Algorithm: Animated Example



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞

S: { A, C }

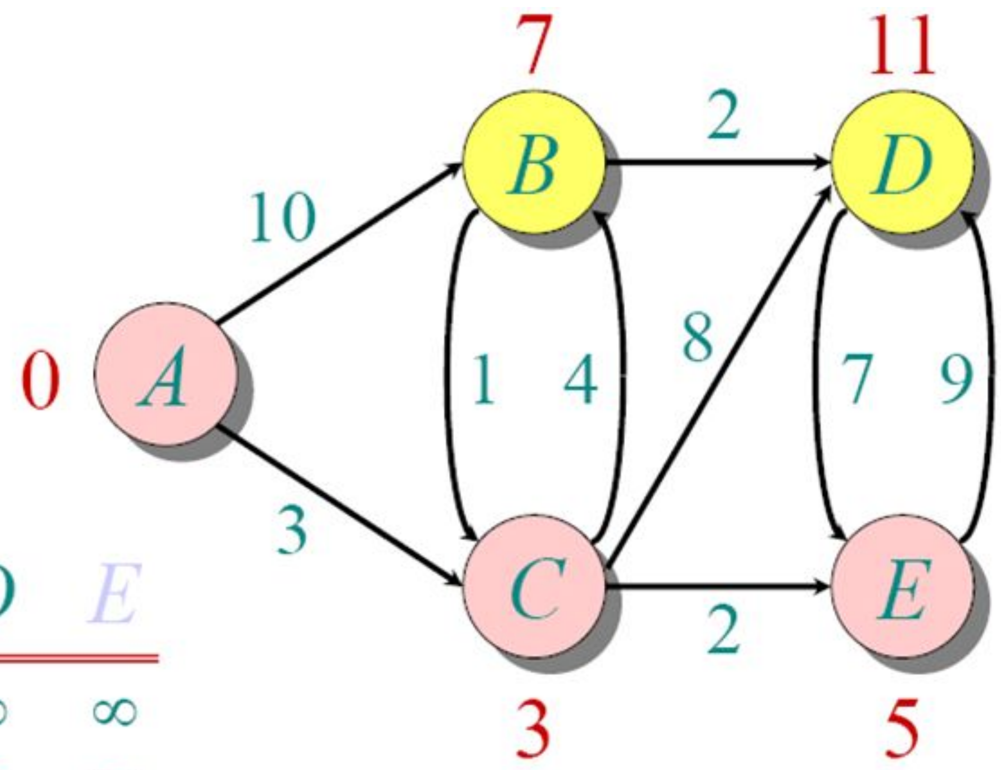
Dijkstra’s Algorithm: Animated Example



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5

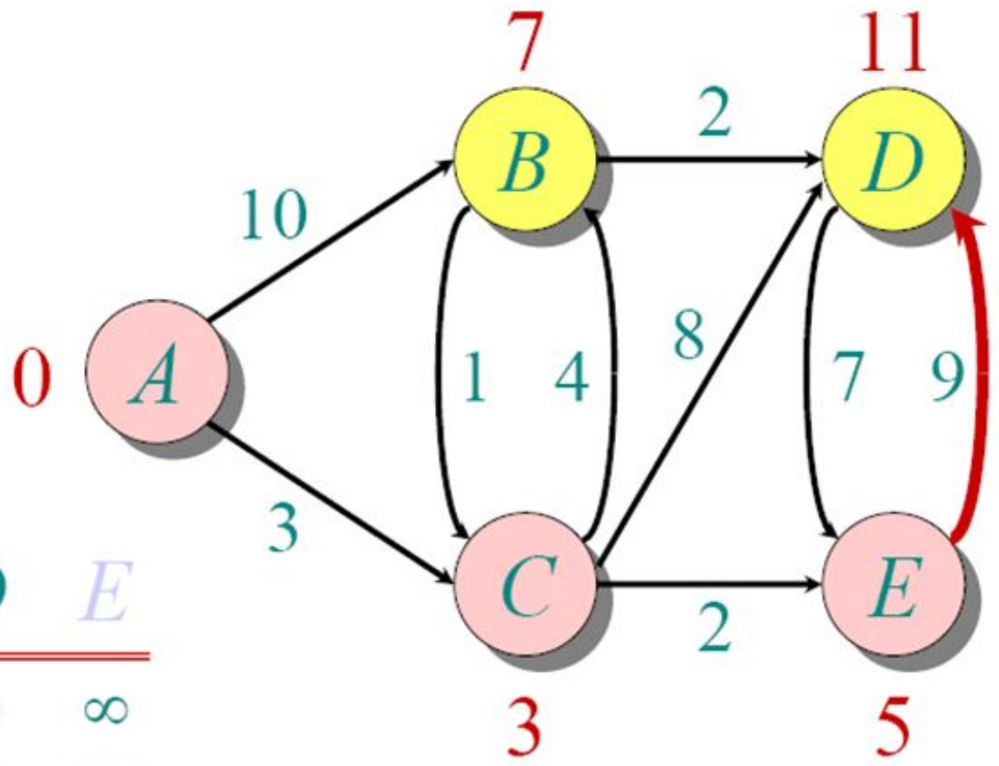
S: { A, C }

Dijkstra’s Algorithm: Animated Example



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5

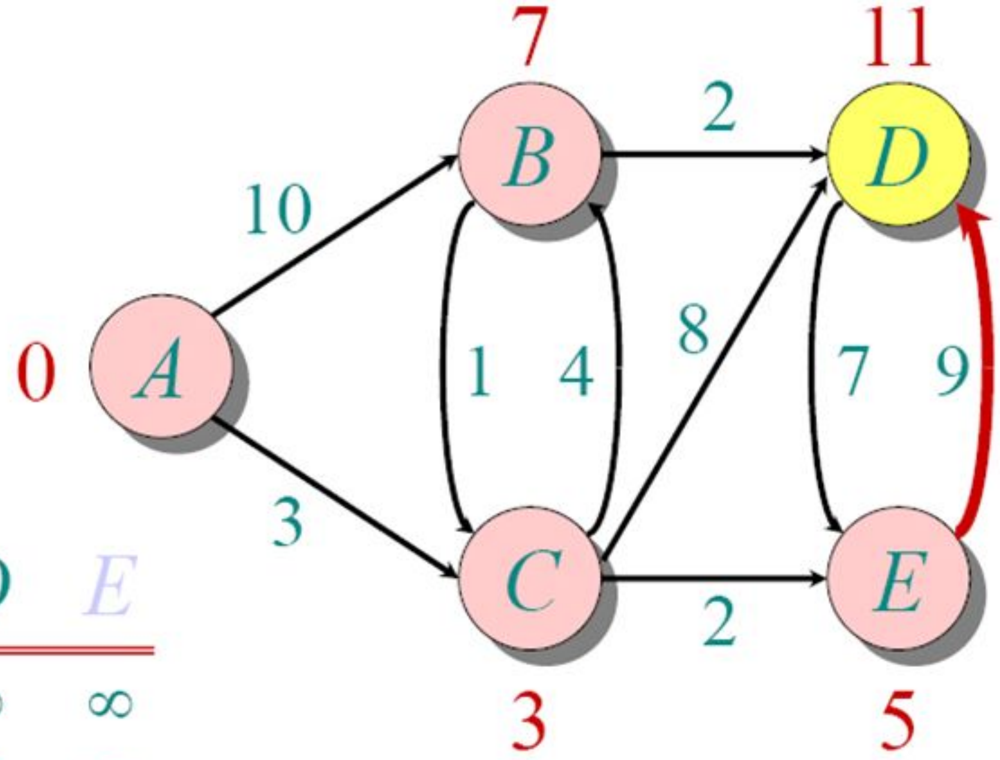
$S: \{ A, C, E \}$



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5
		7		11	

S: { A, C, E }

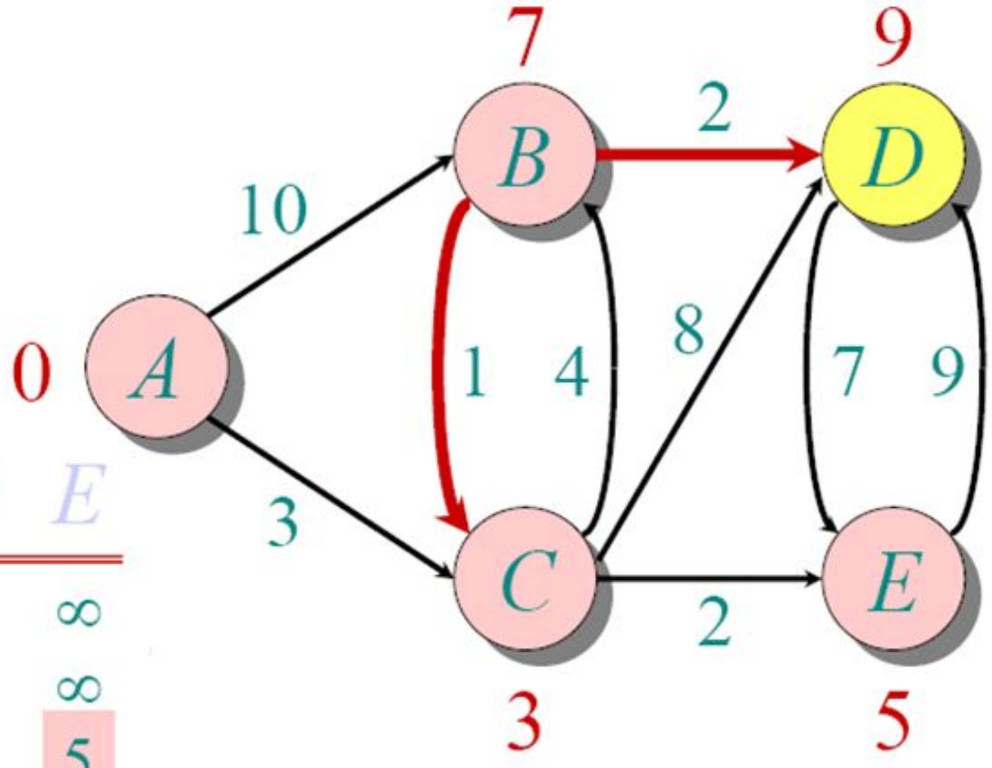
Dijkstra’s Algorithm: Animated Example



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5
		7		11	

$S: \{ A, C, E, B \}$

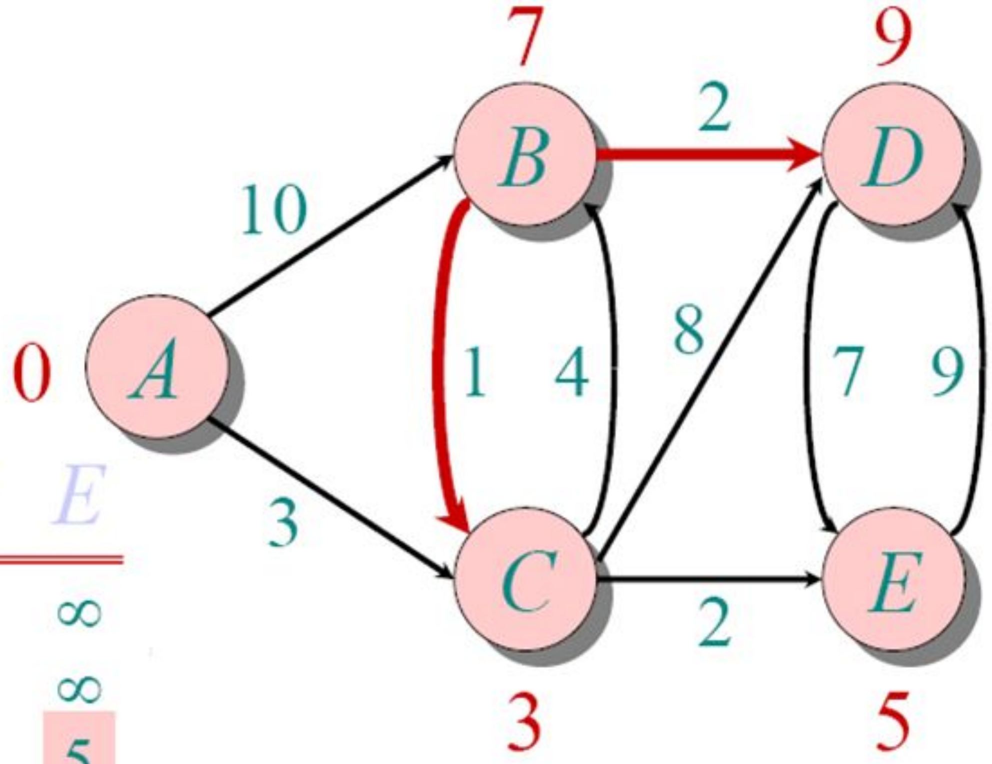
Dijkstra's Algorithm: Animated Example



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5
		7		11	
				9	

S: { A, C, E, B }

Dijkstra’s Algorithm: Animated Example



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5
		7		11	
				9	

S: { A, C, E, B, D }

Difference constraints

Linear Programming

- Given mxn matrix A, m-vector b and n vector c
- Find vector x of n elements that maximises objective function $\sum^n c_i x_i$ subject to m constraints given by $Ax \leq b$

System difference constraints

- Each row of linear programming matrix A contains one 1 and one -1, all other entries of A are 0
- So constraints $Ax \leq b$ are set of m different constraints involving n unknowns of the form $x_j - x_i \leq b_k$

Difference constraints: Example

1	-1	0	0	0				0
1	0	0	0	-1		x_1		-1
0	1	0	0	-1		x_2		1
1	0	-1	0	0	X	x_3	≤	5
1	0	0	-1	0		x_4		4
0	0	-1	1	0		x_5		-1
0	0	-1	0	1				-3
0	0	0	-1	1				-3

$x_1-x_2 \le 0$
$x_1-x_5 \le -1$
$x_2-x_5 \le 1$
$x_1-x_3 \le 5$
$x_1-x_4 \le 4$
$x_4-x_3 \le -1$
$x_5-x_3 \le -3$
$x_5-x_4 \le -3$

Application

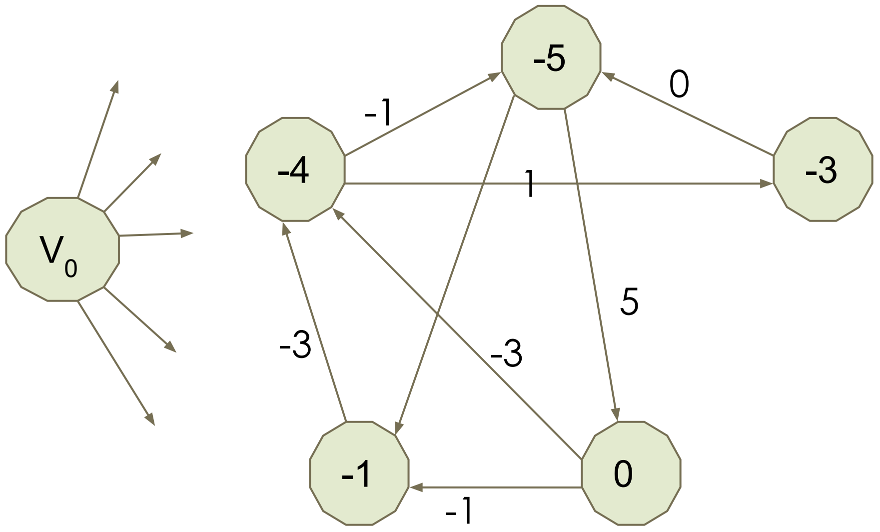
- Let event be jobs to be performed during assembly of product
- e.g. 5 hrs between x_1 and x_3 i.e. $x_3-x_1 \ge 5$

Feasible?? Solution? Yes, (-5,-3,0,-1,-4)

Difference constraints: Graph

Mapping to Graph Algorithm

- Let n (variables) be no. of vertices and m(constraints, equations) be edges
- Add an edge from x_j to x_i (if x_j should follow x_i)with minimum difference of b_k, with weight b_k from k^{th} constraints
- Add vertex v_0 with edges from v_0 to all other vertices with weight 0



Solving the problem

Bellman-Ford returns

- False, negative cycle present , no solution exits.
- True, SSSP from v_0 , by Bellman-Ford provides solution

Proofs of shortest path

- Upper bound property
- No path property
- Convergence property
- Path relaxation property
- Predecessor subgraph property

Examples

- Gift wrapping: Nesting boxes

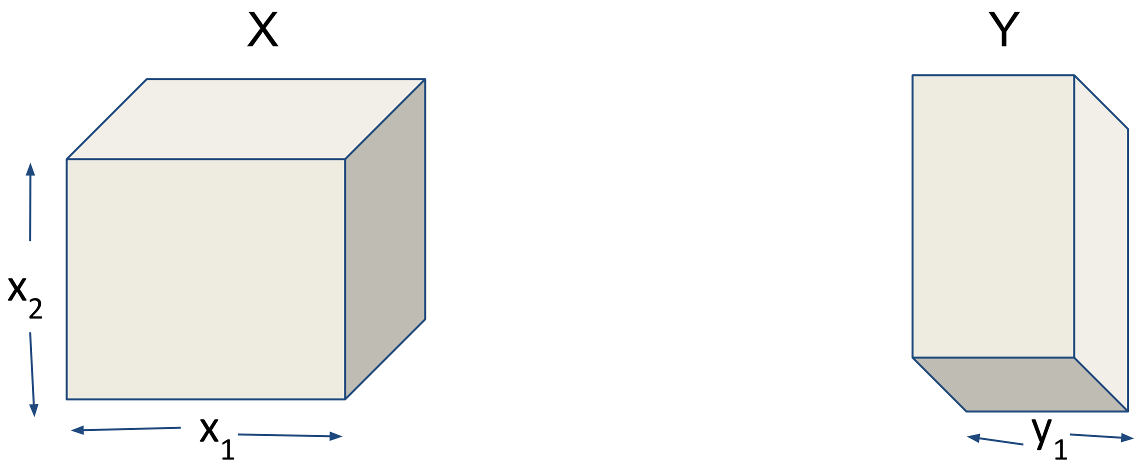
- Arbitrage: Use of discrepancies in currency exchange rates to transform one unit of currency into more that one unit of same currency.
    a. e.g. Ind Rs -> US $ -> Jap yen -> Euro -> Ind Rs

Some problems: Nesting boxes

- Given n boxes, all D dimensional, needs to be nested within one-other (recursively). What is maximum nesting level possible?

Base problem

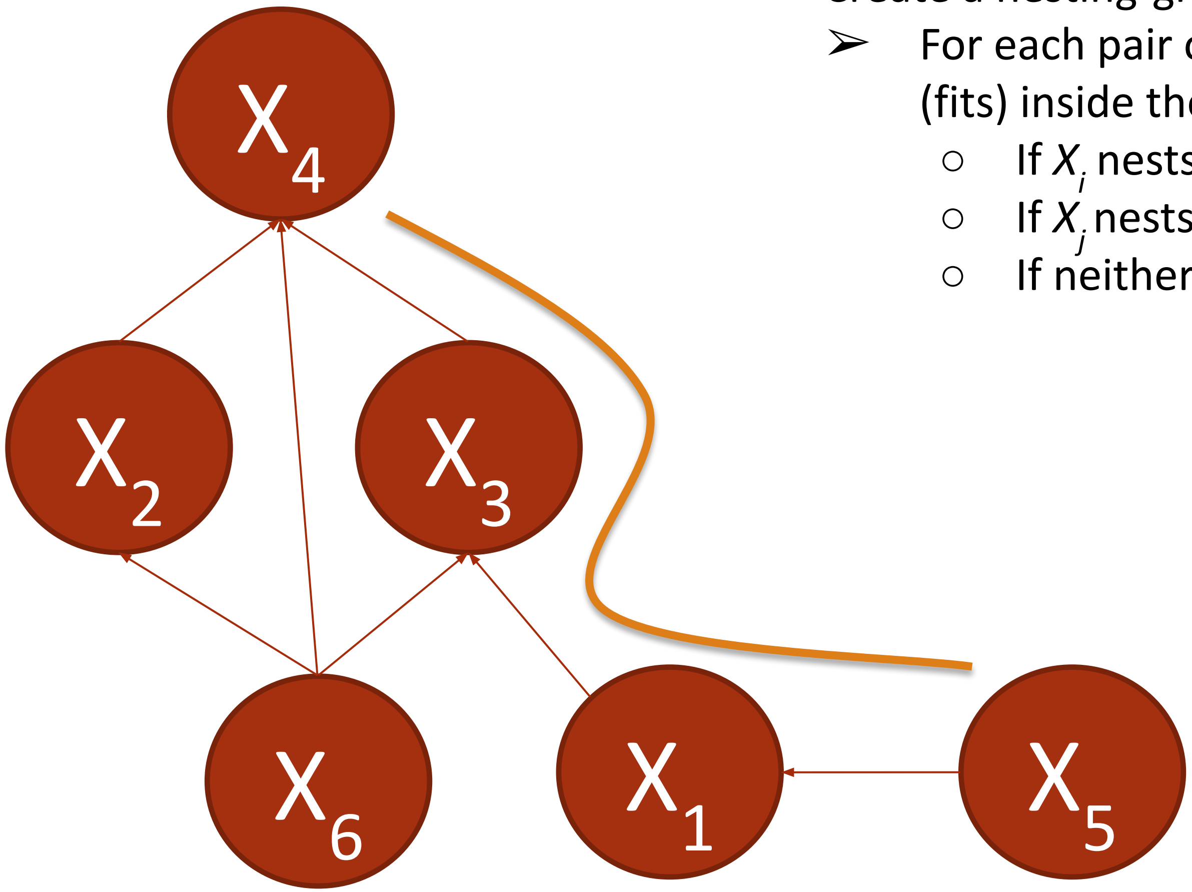
- Given two boxes X and Y with D dimensions as $(x_1, x_2, \ldots x_D)$ and $(y_1, y_2, \ldots y_D)$ how to decide if X fits into Y ?
    - Example $X(x_1, x_2, x_3)$ and $Y(y_1, y_2, y_3)$



Some problems: Nesting boxes

- Box X nests inside box Y if and only if the increasing sequence of dimensions of X is component-wise strictly less than the increasing sequence of dimensions of Y. Thus, it will suffice to sort both sequences of dimensions and compare them.
- With boxes as vertices, indicate for each pair if one can be contained in other, with a directed edge.

Nesting boxes: Example



Create a nesting-graph G with vertices $X_1,\ldots,X_n$ as follows.

➢ For each pair of boxes X_i & X_j, we decide if one nests (fits) inside the other.
  - If X_i nests in X_j, draw an arrow from X_i to X_j.
  - If X_j nests in X_i, draw an arrow from X_j to X_i.
  - If neither nests, draw no arrow.

Nesting boxes: Complexity

- Sorting both length d sequences is done in O(d log d), and comparing their elements is done in O(d), so the total time is O(d lg d).
- To determine the arrows efficiently, after sorting each list of dimensions in $O(nd \lg d)$ time, compare all pairs of boxes using the algorithm in $O(n^2*d)$. The resulted graph is acyclic, which allows us to easily find the **longest chain** in it in O(n^2) in a bottom-up manner. Thus, the total time is $O(nd \max(\lg d, n))$.

Some problems: Arbitrage

- Arbitrage is defined as near simultaneous purchase and sale of securities or foreign exchange in different markets in order to profit from price discrepancies. Rs -> US$ -> EURO ->AUS $ ->Rs ?
- Use of discrepancies in currency exchange rate.
- For the sake of simplicity, let’s assume there are no transaction costs and you can trade any currency amount in fractional quantities.
- 1 U.S. dollar bought 0.82 Euro, 1 Euro bought 129.7 Japanese Yen, 1 Japanese Yen bought bought 0.70 Indian Rupee, and 1 Indian Rupee bought 0.0135489 USD U.S. dollars. Then, by converting currencies, a trader can start with 1 U.S. dollar and buy U.S. dollars, thus turning a 0.82*129.7*0.7* 0.0135489 USD =1.042 US dollars, thus making a 4.2% profit.
- Weighted directed graphs can be represented as an **adjacency matrix**.

Arbitrage: Analysis

- Arbitrage opportunities arise when a cycle is determined such that the edge weights satisfy the following expression **$w_1 * w_2 * w_3 * \ldots * w_n > 1$**
- The above constraint of finding the cycles is harder in graphs.

Let's take the logarithm on both sides, such that

$\log(w_1) + \log(w_2) + \log(w_3) + \ldots + \log(w_n) > 0$

Taking the negative log, this becomes

$(-\log(w_1)) + (-\log(w_2)) + (-\log(w_3)) + \ldots + (-\log(w_n)) < 0$

Therefore we can conclude that if we can find a cycle of vertices such that the sum of their weights if negative, then we can conclude there exists an opportunity for currency arbitrage. Luckily, Bellman-Ford algorithm is a standard graph algorithm that can be used to easily detect negative weight cycles in O(|V*E|) time.

Thank You