6CCS3OME/7CCSMOME – Optimisation Methods

Lecture 1

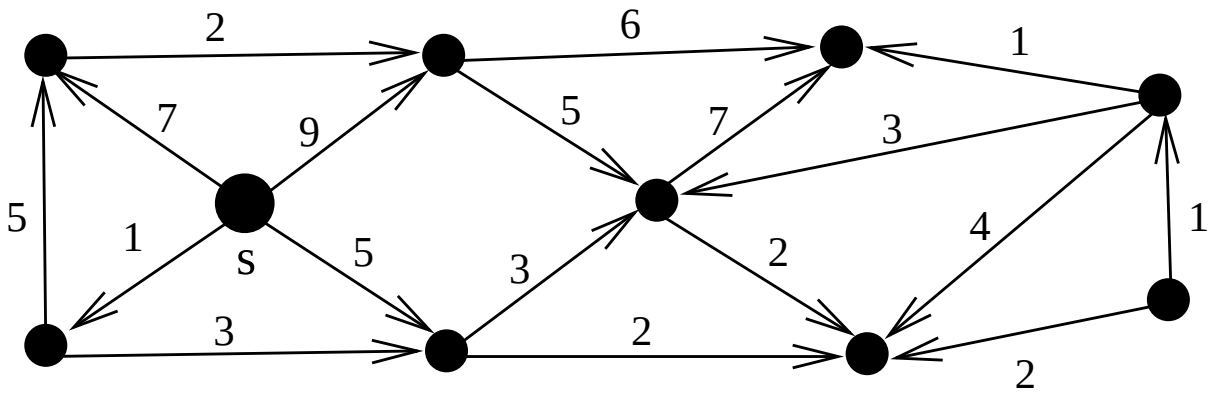
Single-source shortest-paths problem: Basic concepts, Relaxation technique, Bellman-Ford algorithm

Tomasz Radzik and Kathleen Steinhöfel

Department of Informatics, King's College London

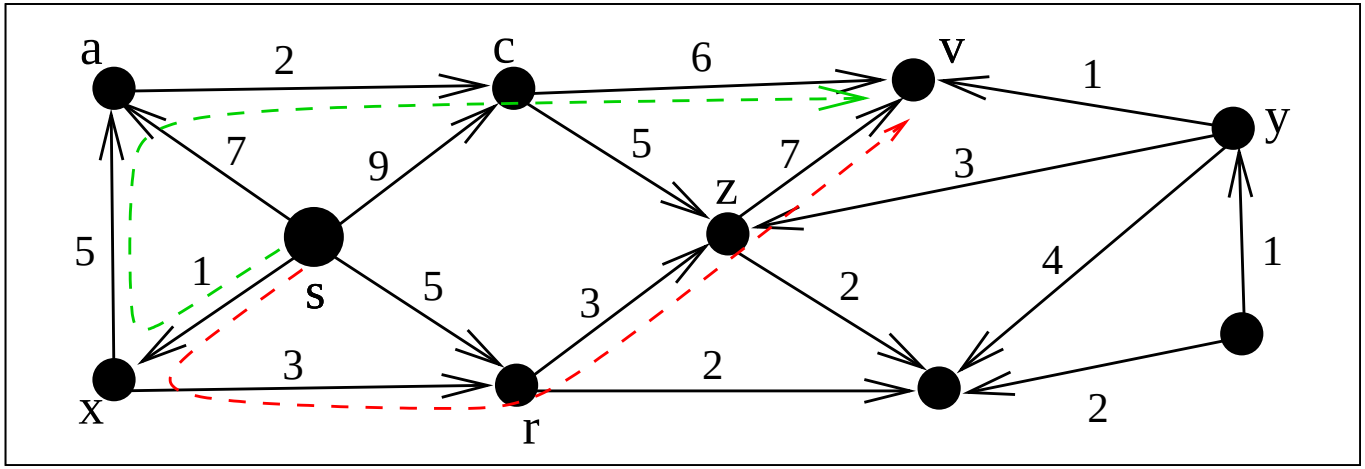
2020/21, Second term

Single-source shortest-paths problem



- $G = (V, E)$ – directed graph; $|V| = n$, $|E| = m$.
- $w(v, u)$ – the weight of edge (v, u)
- $s \in V$ – the source vertex
- Compute the shortest paths, that is, the paths with the smallest total weights, from s to all other vertices.
- The point-to-point shortest path problem is normally solved using a single-source shortest path algorithm (or an adaptation of such an algorithm).

Preliminaries



$\delta(s, v) = 14.$

Two shortest-paths from s to v.

$\delta(s, y) = +\infty.$

- A **path**: $p =< v_1, v_2, \ldots, v_k, v_{k+1} >$, where (v_i, v_{i+1}) is an edge for each $i = 1, 2, \ldots, k$. For example, path $Q =< s, x, r, z, v >$.
- The **weight** of a path $p =< v_1, v_2, \ldots, v_k, v_{k+1} >$: $w(p) = w(v_1, v_2) + w(v_2, v_3) + \cdots + w(v_k, v_{k+1})$. $w(Q) = 14.$
- The **shortest-path weight (distance)** from u to v:

$$\delta(u, v) = \begin{cases} \text{min } w(p) \text{ over all } p \text{ from } u \text{ to } v, \text{ if there is any such path;} \\ +\infty, \text{ if there is no path from } u \text{ to } v. \end{cases}$$

- A **shortest path** from u to v is any path p from u to v with weight $w(p) = \delta(u, v)$.

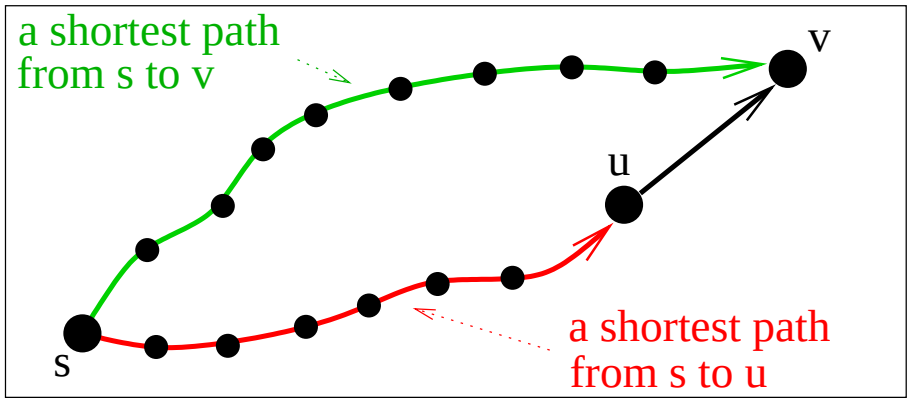
Preliminaries (cont.)

- Useful simple facts:

  1. A subpath of a shortest path is a shortest path.

     In the graph on the previous slide: path $\langle x, r, z \rangle$ is a subpath of a shortest path $\langle s, x, r, z, v \rangle$, so it must be a shortest path (from x to z).

  2. Triangle inequality:
     for each edge $(u, v) \in E$, $\delta(s, v) \leq \delta(s, u) + w(u, v)$.



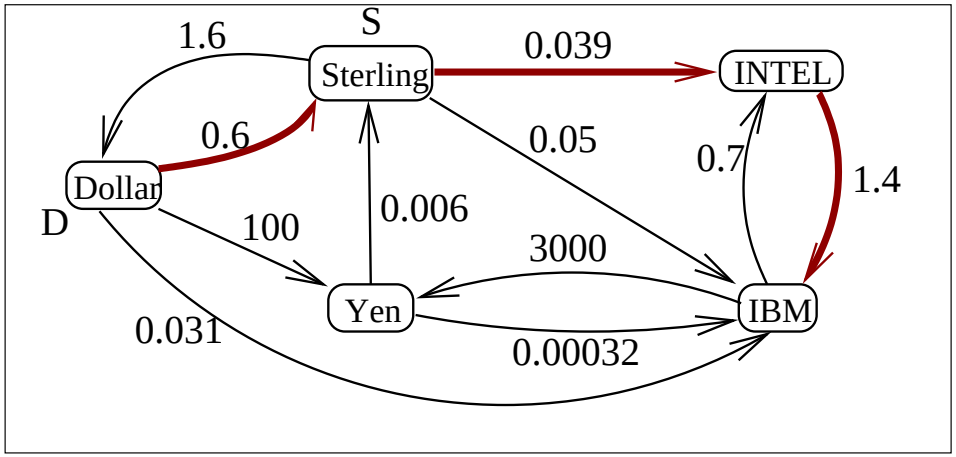
     Holds also if u or v is not reachable from s.

- First the general case:
  the weights of edges may be negative.

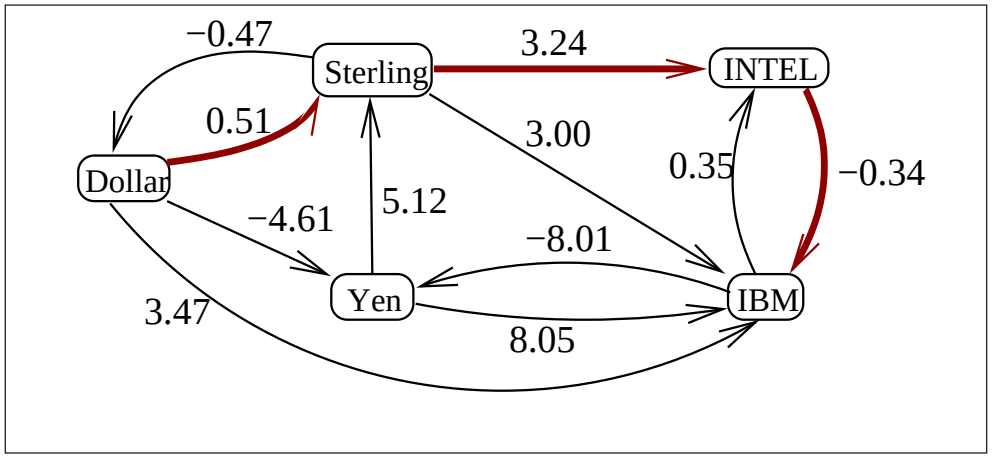
Negative weights in an application

Example from financial analysis: a graph of exchange rates.

Find paths maximising exchange rates:



Find shortest paths:



- For an edge (x, y): $\gamma(x, y) =$ the exchange rate from x to y. E.g., $\gamma(S, D) = 1.6$.
- Set the weight of edge (x, y): $w(x, y) = -\ln(\gamma(x, y))$

  (We use the natural logarithms, but any fixed base > 1 would do.)
- For example: $w(S, D) = -\ln(\gamma(S, D)) = -\ln(1.6) \approx -0.47$
- A path from v to u maximises the combined exchange rate from v to u, if and only if, this is a shortest path from v to u according to the these edge weights.
- If $\gamma(x, y) > 1$, then $w(x, y) < 0$.

  We cannot avoid negative weights here, so we have to solve the shortest-paths problem in a graph with (some) edge weights negative.

The exchange-rates example (cont.)

For a path $P = \langle v_1, v_2, \ldots, v_k \rangle$:

$$\begin{aligned}
w(P) &= w(v_1, v_2) + w(v_2, v_3) + \cdots w(v_{k-1}, v_k) \\
&= -\ln(\gamma(v_1, v_2)) - \ln(\gamma(v_2, v_3)) - \cdots - \ln(\gamma(v_{k-1}, v_k)) \\
&= -\left[\ln(\gamma(v_1, v_2)) + \ln(\gamma(v_2, v_3)) + \cdots + \ln(\gamma(v_{k-1}, v_k))\right] \\
&= -\ln\left(\gamma(v_1, v_2)\gamma(v_2, v_3)\cdots\gamma(v_{k-1}, v_k)\right) \qquad \{\ln(a) + \ln(b) + \ln(c) = \ln(abc)\} \\
&= -\ln\left(\gamma(P)\right).
\end{aligned}$$

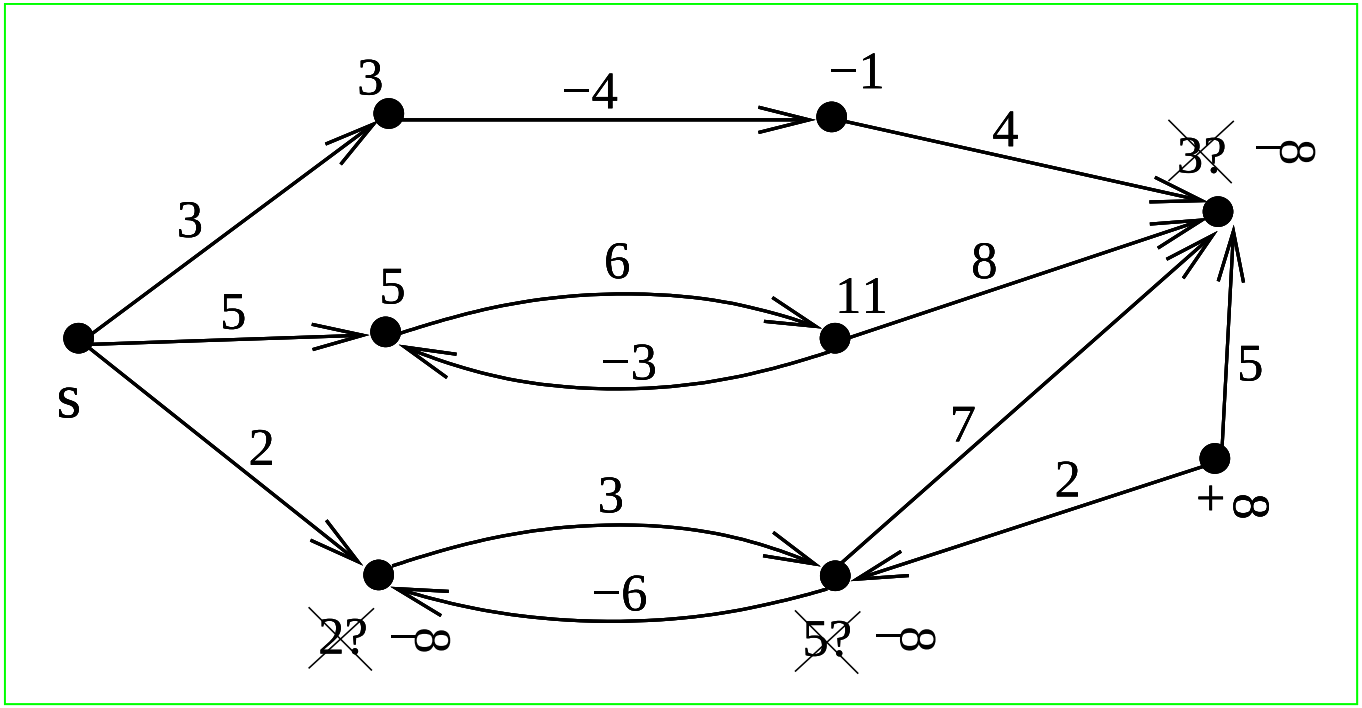
For two paths P and Q, the following relations are equivalent:

$$\begin{aligned}
\gamma(P) &> \gamma(Q) \\
\ln(\gamma(P)) &> \ln(\gamma(Q)) \\
-\ln(\gamma(P)) &< -\ln(\gamma(Q)) \\
w(P) &< w(Q)
\end{aligned}$$

That is, "$\gamma(P) > \gamma(Q)$" if and only if "$w(P) < w(Q)$"

Thus, a path P from v to u is a maximum exchange rate path from v to u, if and only if, P is a shortest path from v to u according to the edge weights w.

Negative-weight cycles (negative cycles)

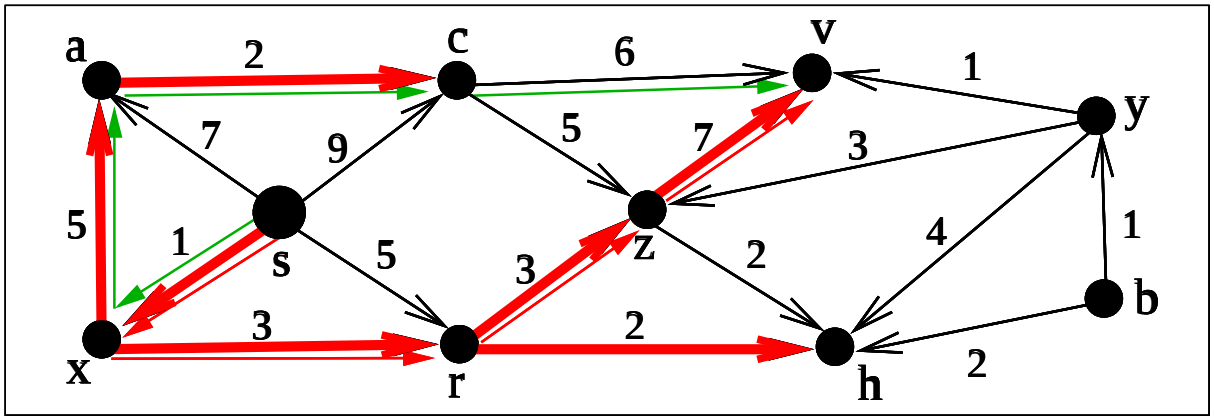


- If there is a negative cycle on a path from s to v, then by going increasingly many times around such a cycle,
  we get paths from s to v
  of arbitrarily small (negative) weights.
- If there is a negative cycle on a path from s to v,
  then, by convention, $\delta(s, v) = -\infty$.

We give up, if we detect a negative cycle in the input graph

- We consider only shortest-paths algorithms which:
  - compute shortest paths for graphs with no negative cycles reachable from the source
  - for graphs with negative cycles reachable from the source, correctly detect that the input graph has a negative cycle (but are not required to compute anything else).
- Why don't we compute shortest *simple* paths (not containing cycles)? Such paths are always well defined, even if the graph has negative cycles.
- This would be a valid, and interesting computational problem (with some applications).
- The issue: we know efficient algorithms which compute shortest paths in graphs with no negative cycles, but we don't know any efficient algorithm for computing shortest simple paths in graphs with negative cycles.
- The problem of computing shortest simple paths in graphs containing negative cycles is NP-hard, that is, computationally difficult (by reduction from the Hamiltonian Path problem), and the algorithms which we discuss in Lectures 1–3 do not apply.

Representation of computed shortest paths: A shortest-paths tree



$\textsc{Parent}[a] = x$
$\textsc{Parent}[x] = s$
$\textsc{Parent}[s] = \text{nil}$
$\textsc{Parent}[y] = \text{nil}$
$\ldots$

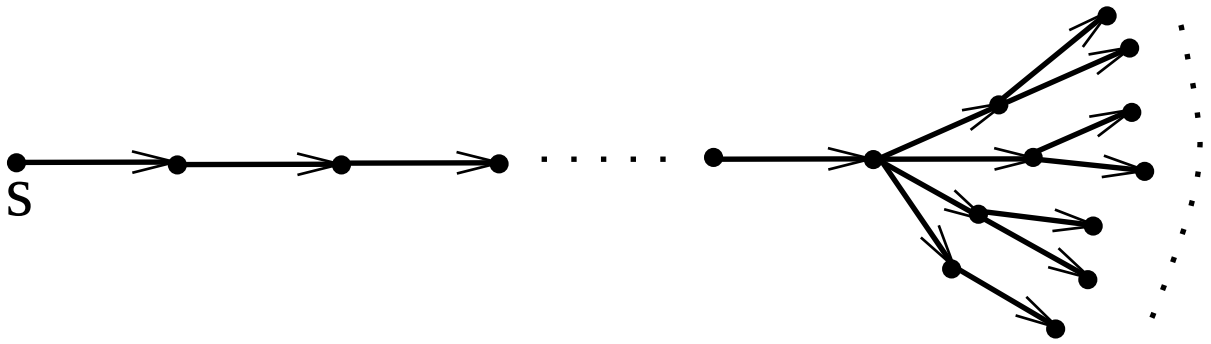
- If there are multiple shortest paths from s to v, do we want to find all of them or just one?

  For each node v reachable from s, we want to find just one shortest path from s to v.
- How should we represent the output, that is, the computed shortest paths?
- If there is no negative cycle reachable from s, then shortest paths from s to all nodes reachable from s are well defined and can be represented by a **shortest-paths tree**:

  a tree rooted at s such that for any node v reachable from s, the tree path from s to v is a shortest path from s to v.

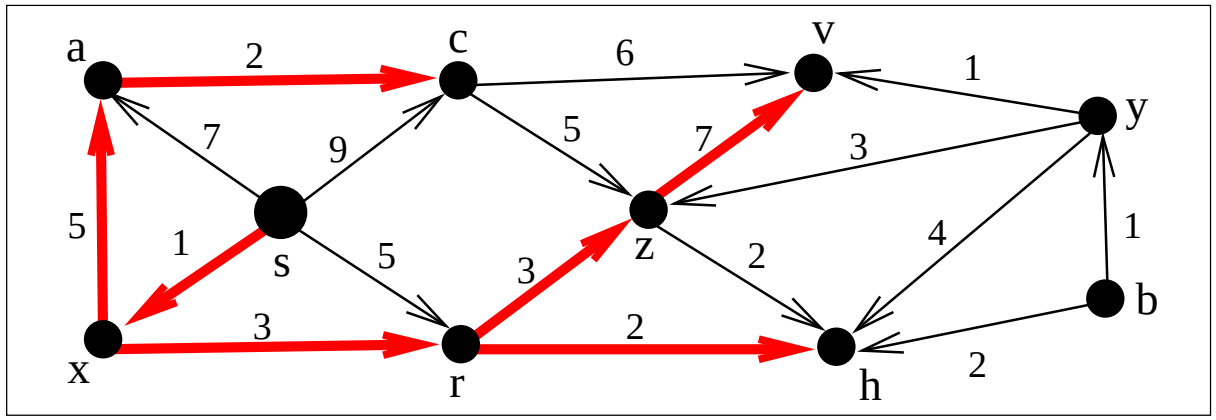
Representation of computed shortest paths: A shortest-paths tree (cont.)

- A shortest-paths tree from s contains exactly one shortest path from s to each v reachable from s. There may be other shortest paths from s to v, which are not included in this tree.
- A shortest-paths tree T can be represented by an array PARENT$[v]$, $v \in V$, in $\Theta(n)$ space (memory) , where n is the number of nodes in the graph.
- PARENT$[v]$ is the predecessor of node v in tree T.
- An explicit representation of shortest-paths from s (a list of shortest-paths, one path per one node reachable from s, and each path represented as a full sequence of all its edges) would take $\Theta(n^2)$ space in the worst case.



Output of a shortest-paths algorithm

- A shortest-paths algorithm computes the shortest-path weights and a shortest-paths tree, or detects a negative cycle reachable from s.
- Example. Input graph and the computed shortest-paths tree:



The output of a shortest-path algorithm: array PARENT[.] representing the computed shortest-paths tree and array $d[.]$ with the shortest-path weights:

node	a	b	c	h	r	s	v	x	y	z
PARENT[node]	x	nil	a	r	x	nil	z	s	nil	r
d[node]	6	∞	8	6	4	0	14	1	∞	7

- Terminology and notation in [CLRS]:

(parent, PARENT$[v]$, $d[v]$) $\longrightarrow$ (predecessor, $v.\pi$, $v.d$).

Relaxation technique (for computing shortest paths)

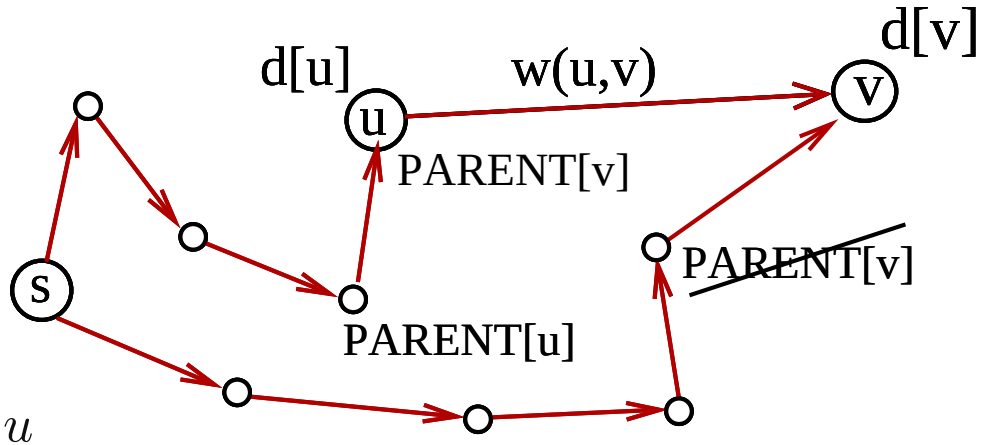
- For each node v, maintain:
  - $d[v]$: the **shortest-path estimate** for v – an upper bound on the weight of a shortest path from s to v;
  - PARENT$[v]$: the current predecessor of node v.
- The **relaxation technique**:

  INITIALIZATION followed by
  a sequence of RELAX operations.

node	a	b	c	h	r	s	v	x	y	z
PARENT	nil	nil	nil	nil	nil	nil	nil	nil	nil	nil
d	∞	∞	∞	∞	∞	0	∞	∞	∞	∞

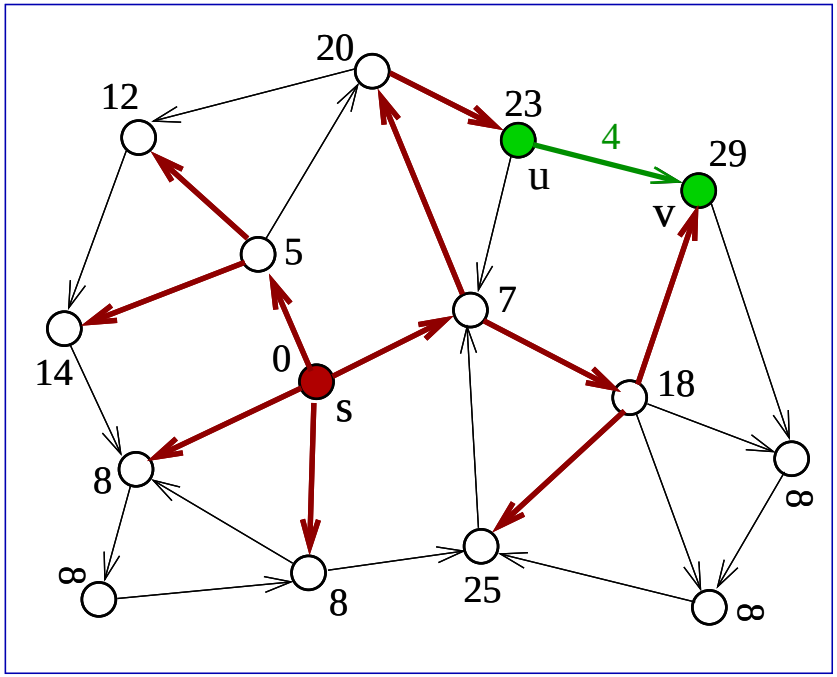
INITIALIZATION(G, s)
  $d[s] \leftarrow 0$; PARENT$[s] \leftarrow$ NIL
  for each node $v \in V - \{s\}$ **do**
    $d[v] \leftarrow \infty$; PARENT$[v] \leftarrow$ NIL

RELAX(u, v, w) { relax edge (u, v) }
  if $d[v] > d[u] + w(u, v)$ **then**
    $d[v] \leftarrow d[u] + w(u, v)$; PARENT$[v] \leftarrow u$

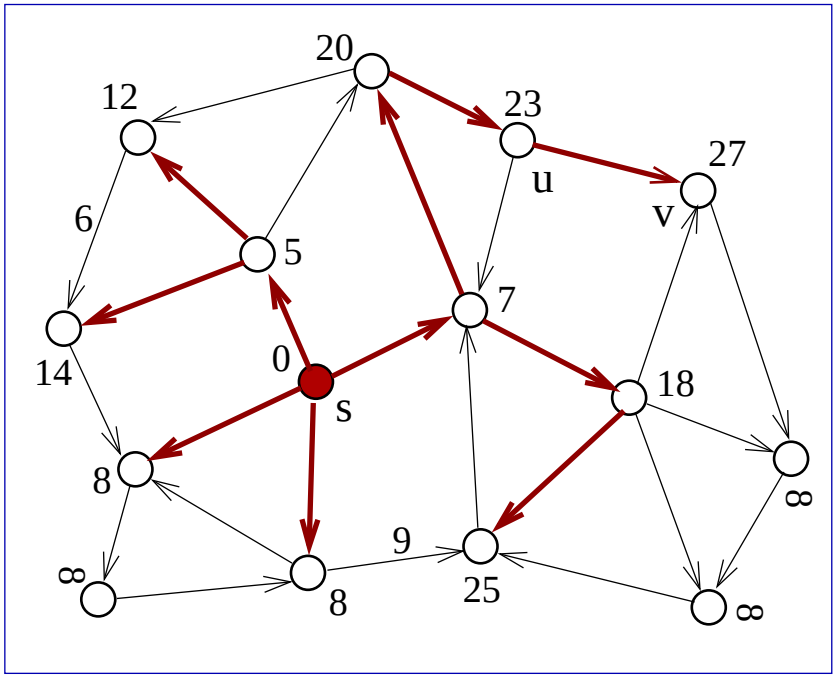


- All algorithms discussed in this module are based on the relaxation technique (as most of the SP algorithms). They differ in the order of RELAX operations.
- When should the computation stop?

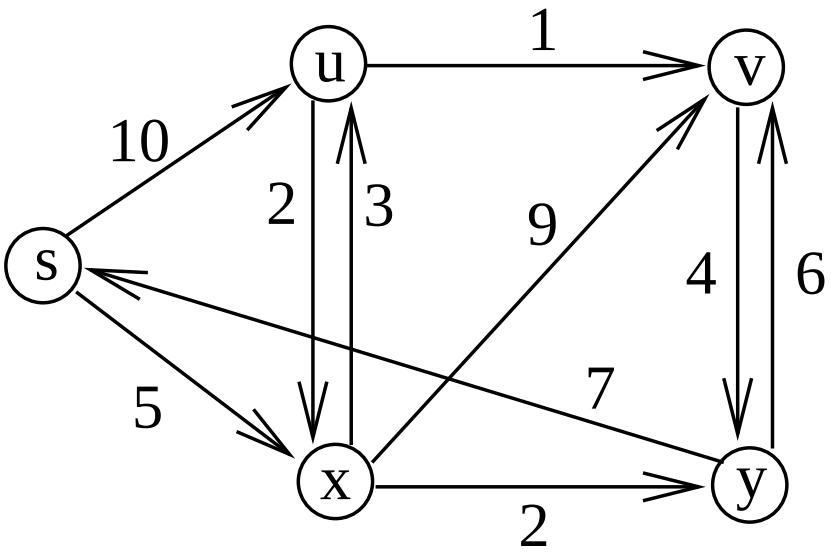
Relax operation

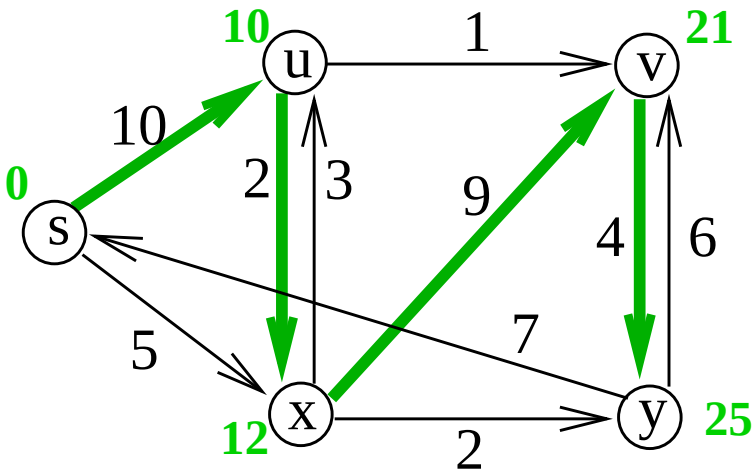


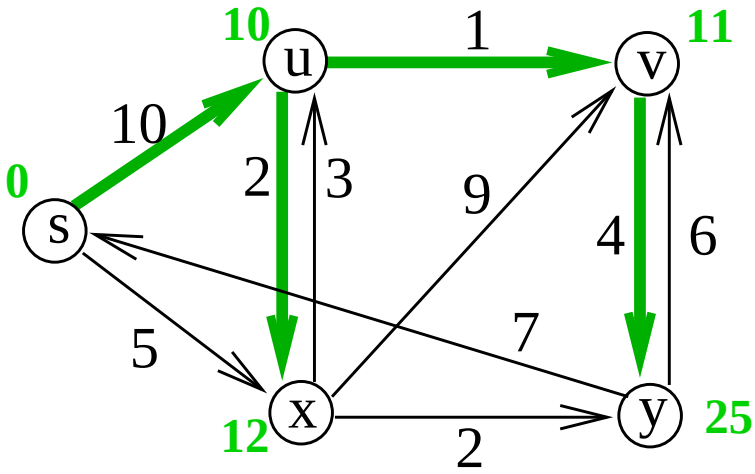
RELAX(u, v)

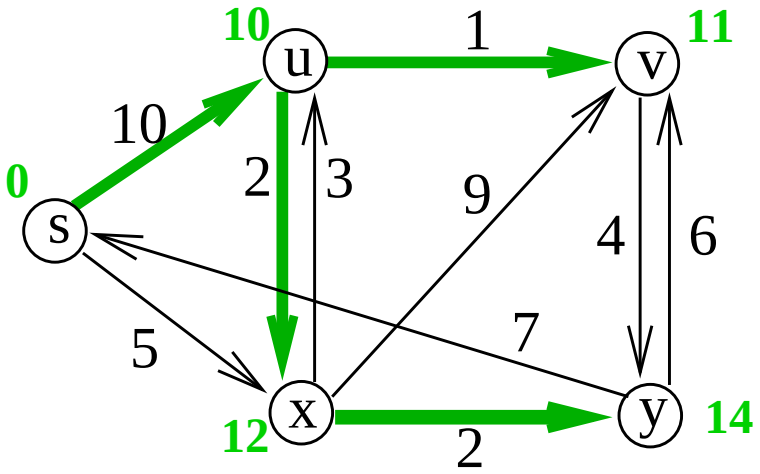


Example of Relaxation Technique – LGT





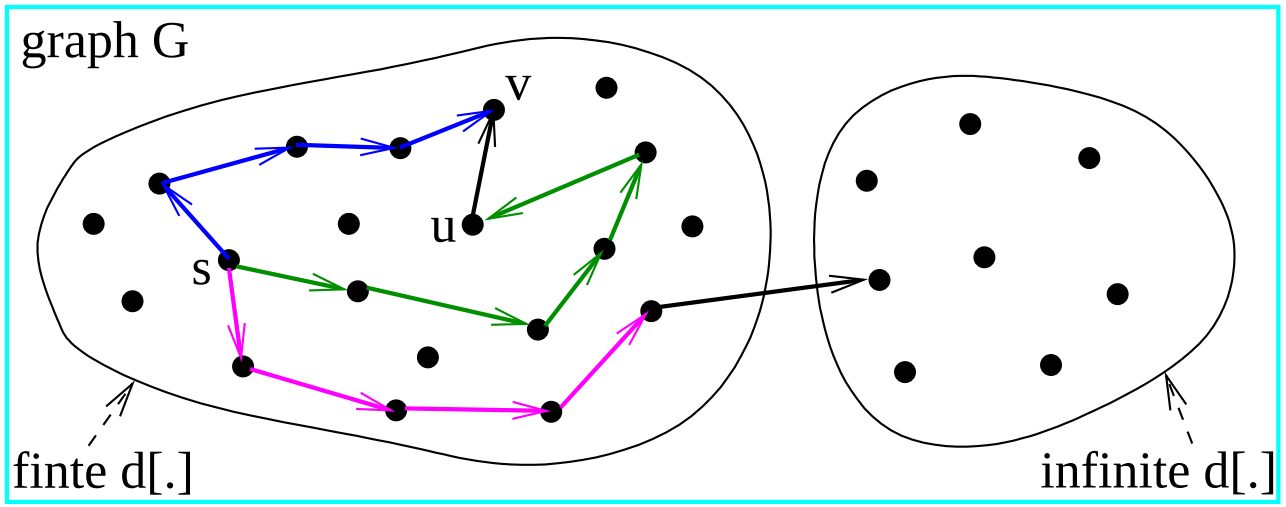




. . . and so on.

Properties of the relaxation technique

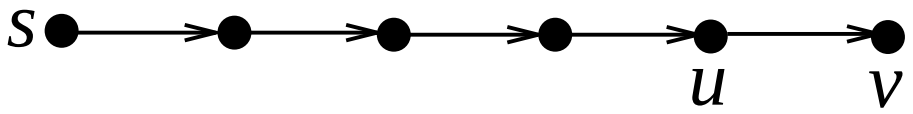
- (1) **Non-increasing shortest-path estimates**. Throughout the computation, for each node v, the shortest-path estimate $d[v]$ can only decrease (it never increases). (from definition of operation)
- (2) For each node v, the **shortest-path estimate** $d[v]$ is always either equal to ∞ (at the beginning) or **equal to the weight of some path from** s **to** v. (by induction)



- (3) **Upper bound property**. For each node v, we always have $d[v] \geq \delta(s, v)$. (directly from (2))
- (4) **No-path property**. If there is no path from s to v, then we always have $d[v] = \delta(s, v) = \infty$. (directly from (2))

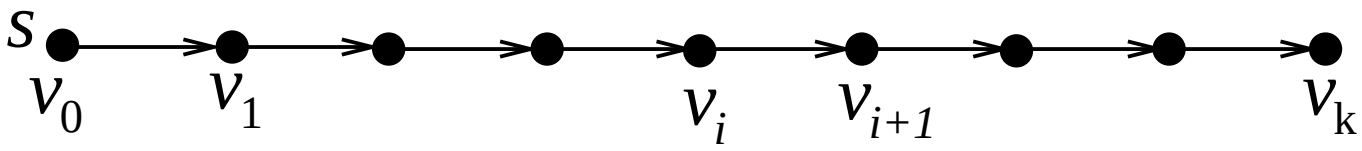
Properties of the relaxation technique (cont. 1)

- (5) **Convergence property**. If $(s, \ldots, u, v)$ is a shortest path from s to v and if $d[u] = \delta(s, u)$ at any time prior to relaxing edge (u, v), then $d[v] = \delta(s, v)$ at all times afterward.



- (6) **Path-relaxation property**. If $p = (v_0, v_1, \ldots, v_k)$, is a shortest path from $s = v_0$ to v_k, and we relax the edges of p in the order (v_0, v_1), $(v_1, v_2), \ldots, (v_{k-1}, v_k)$, then $d[v_k] = \delta(s, v_k)$.

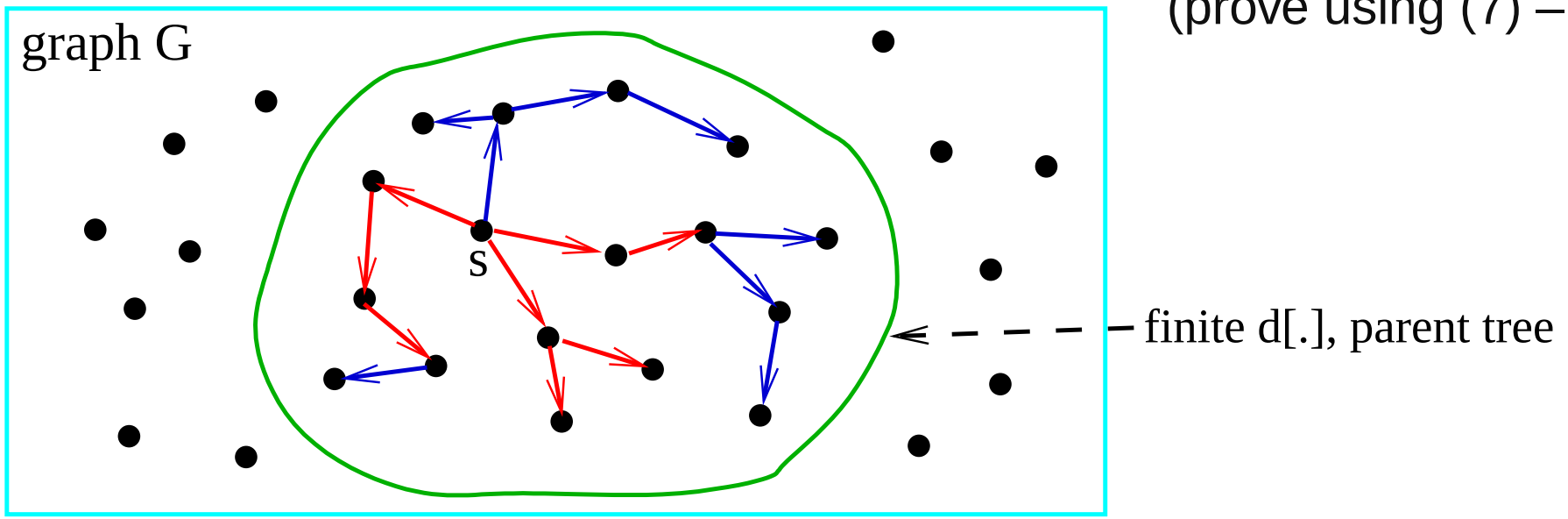
  This property holds regardless of any other relaxation steps that occur, even if they are intermixed with relaxations of the edges of path p.



Relaxation technique: properties of the parent subgraph

- (7) For each edge (u, v) in the current parent subgraph (that is, $u = \text{PARENT}[v]$), $d[u] + w(u, v) \leq d[v]$. (by induction – LGT)
- (8) If the graph contains no negative-weight cycle reachable from s, then
  - the parent subgraph is always a tree T rooted at s,
  - for each node v in tree T, $d[v] \geq$ the weight of the path in T from s to v.

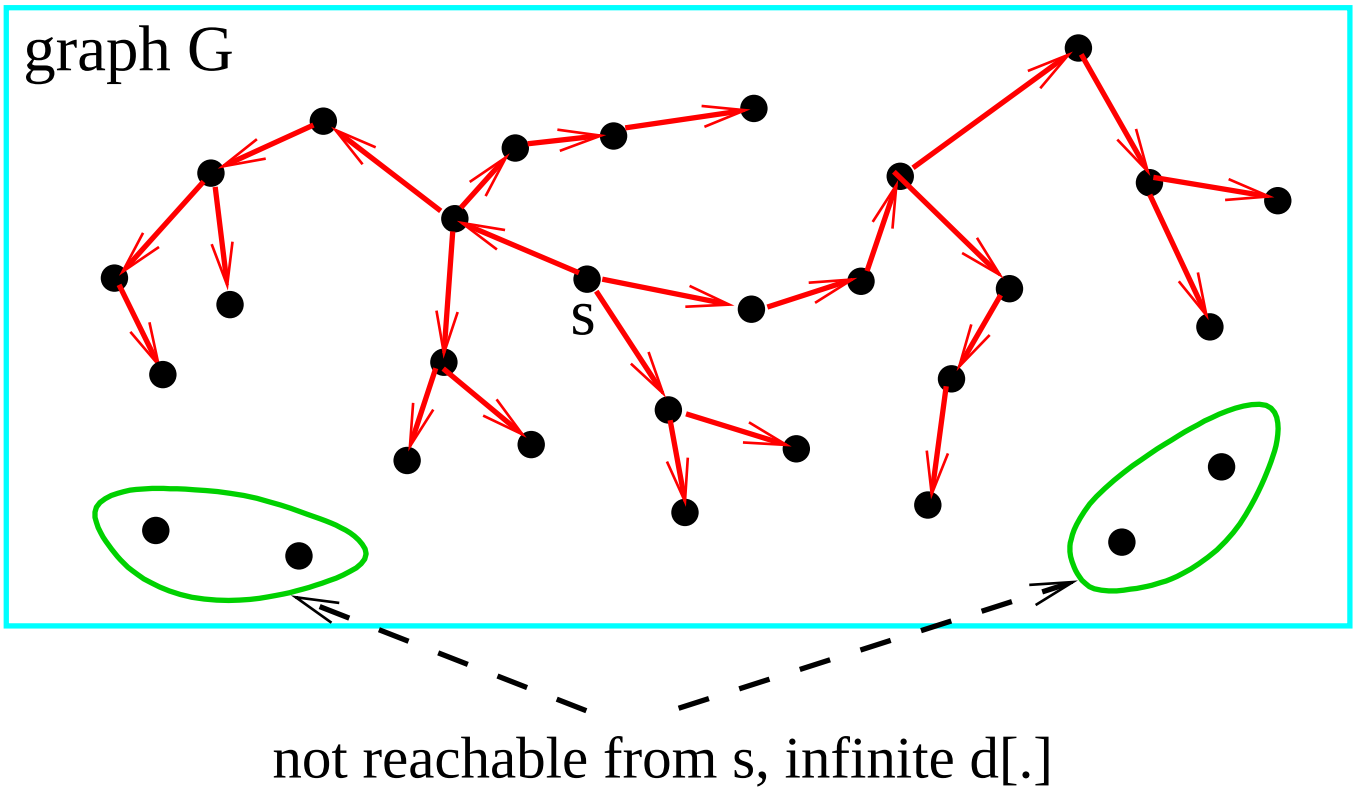
  (prove using (7) – LGT)



The red part of the current parent tree ("near" the source vertex s) is already part of the computed shortest-paths tree.
The blue part may change during the subsequent relax operations.

Relaxation technique: properties of the parent subgraph (cont.)

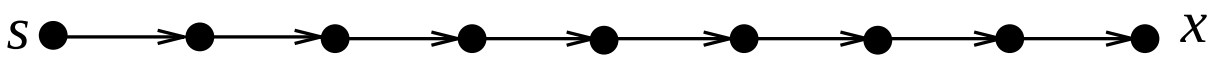
- (9) When $d[v] = \delta(s, v)$ for all $v \in V$, then
  - the parent subgraph is a shortest-paths tree rooted at s, (from (8))
  - no further updates possible. (from (3) - the upper bound property)



Effective relax operations

- Definition:
  an **effective relax operation** is a relax operation which decreases $d[v]$.
- (10) The computation can progress, if the s.p. weights not reached yet:
  - (a) There exists a vertex $x \in V$ such that $d[x] > \delta(s, x)$, if and only if,
  - (b) There exists an edge $(u, v) \in E$ such that $d[v] > d[u] + w(u, v)$ (that is, another effective relax operation is possible).
- (b) $\Rightarrow$ (a): from (3) and the Triangle Inequality.
- (a) $\Rightarrow$ (b): Assume $d[x] > \delta(s, x)$ for some node x (so $\delta(s, x) < +\infty$).
  - Case $\delta(s, x) > -\infty$ (no negative cycle on a path from s to x).

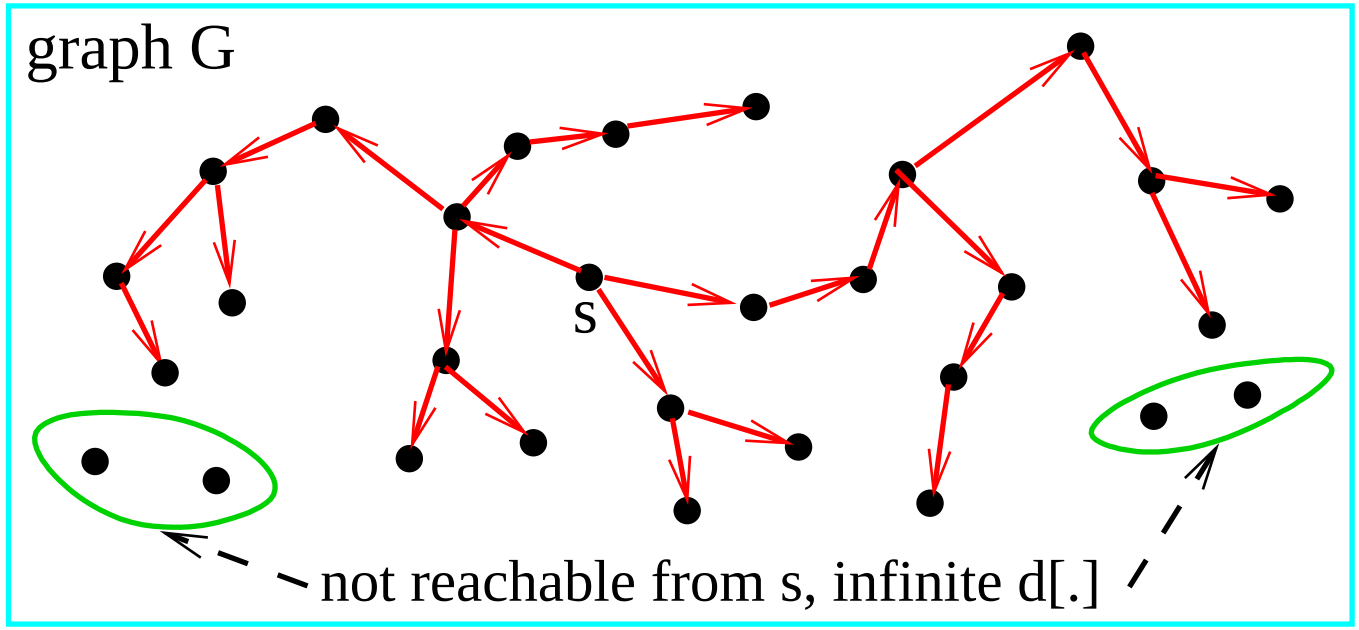
    Consider a shortest s-to-x path P:



$d[s] = \delta(s, s)$ $\qquad$ $d[x] > \delta(s, x)$

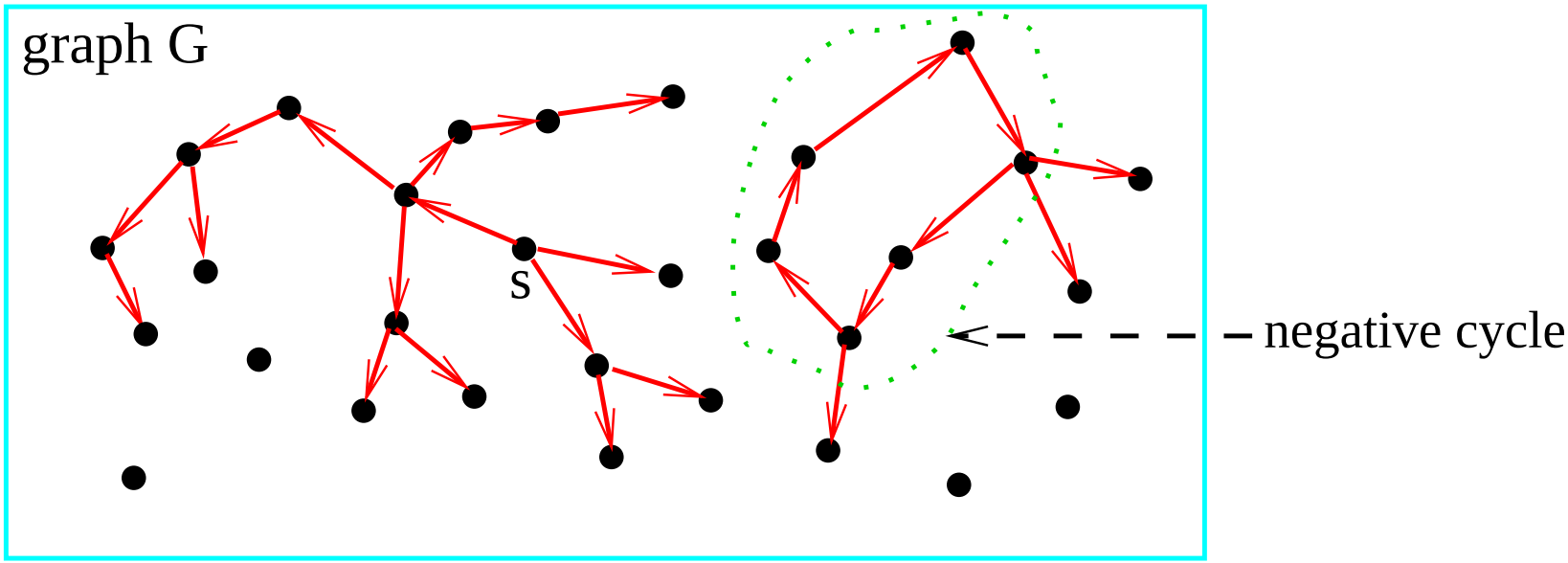
    Must have an edge (u, v) on P such that $d[u] = \delta(s, u)$ but $d[v] > \delta(s, v)$.
    For such an edge: $d[v] > \delta(s, v) = \delta(s, u) + w(u, v) = d[u] + w(u, v)$.
  - Case $\delta(s, x) = -\infty$ for some vertex x: exercise (consider a path $(s, u_1, \ldots, u_k, v_1, \ldots, v_r, v_1)$, where $(v_1, \ldots, v_r, v_1)$ is a negative cycle).

The relaxation technique: summary for the case of no negative cycles



- If **no negative cycle** is reachable from s:
  - Only finitely many effective relax operations during the computation. (If no neg. cycles, Prop. (2) becomes: $d[v]$ is ∞ or the weight of a simple s-to-v path. Hence $d[v]$ takes on only finitely many different values.)
  - The PARENT-subgraph is always a tree rooted at s. (property (8))
  - When eventually no effective relax operation possible, then for each node v, $d[v] = \delta(s, v)$ (property (10)), and the PARENT pointers form a shortest-paths tree (property(9)).

The relaxation technique: summary for the case with negative cycles



- If there is **a negative cycle** reachable from s:
  - There is always an edge (u, v) such that: $d[v] > d[u] + w(u, v)$, that is, an effective RELAX operation is always possible. (from property (10))
  - The PARENT subgraph eventually contains a cycle. (not easy to prove)
  - Thus we can detect the existence of a negative cycle by periodically checking if the PARENT pointers form a cycle.
    This way we can avoid infinite loop, so we can turn any relaxation technique computation into an algorithm.

    How to check for cycles in the parent subgraph and how often?

The Bellman-Ford algorithm (for single-source shortest-paths problem)

- Edge weights may be negative.

BELLMAN-FORD(G, w, s) $\{\ G = (V, E)\ \}$
  INITIALIZATION(G, s)
1: **for** $i \leftarrow 1$ **to** $n - 1$ **do** $\{\ n = |V|\ \}$
    for each edge $(u, v) \in E$ **do** { edges considered in arbitrary order}
      RELAX(u, v, w)

representation of input:
nodes indexed by $1, 2, \ldots, n$;
one node index as the source;
list of edges

2: **for** each edge $(u, v) \in E$ **do**
    if $d[v] > d[u] + w(u, v)$ **then**
      return FALSE { a negative cycle is reachable from s}
  return TRUE { no negative cycles; for each $v \in V$, $d[v] = \delta(s, v)$ }

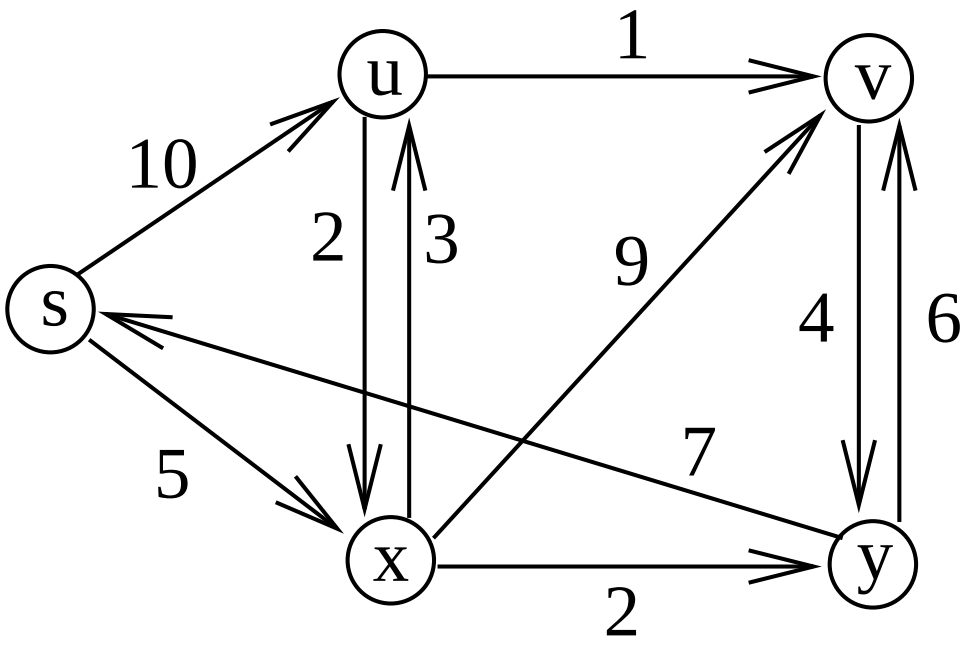
- This algorithm is based on the relaxation technique: INITIALIZATION followed by a (finite) sequence of RELAX operations.
- The running time is $\Theta(mn)$, where n is the number of nodes and m is the number of edges.

exactly of the order of

- The worst case running time of any algorithm for the single-source shortest paths problem with negative weights is $\Omega(mn)$.

at least of the order of

Example of the computation by the Bellman-Ford algorithm – LGT



Assume that the edges are given in this order:

$$(s,u),\ \ (s,x),\ \ (y,s),\ \ (v,y),\ \ (u,v),\ \ (x,v),\ \ (y,v),\ \ (x,u),\ \ (x,y),\ \ (u,x).$$

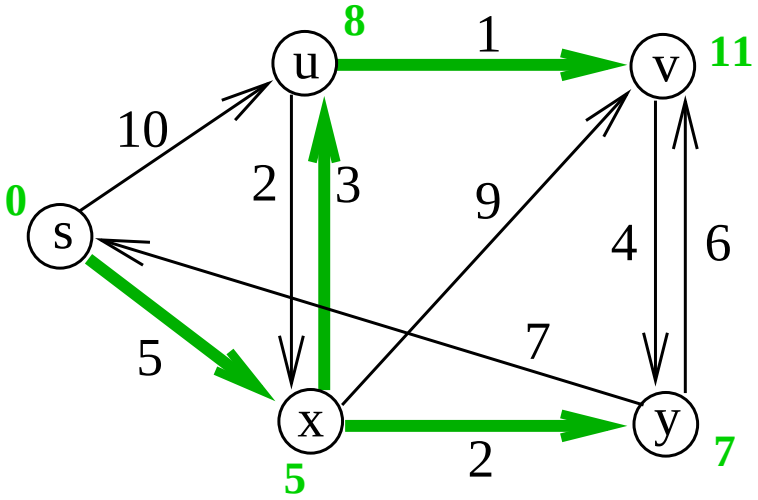
Initially:
(relaxation technique initialization)

node	s	u	v	x	y
PARENT[.]	nil	nil	nil	nil	nil
$d[.]$	0	∞	∞	∞	∞

Example the computation by the Bellman-Ford algorithm (cont.)

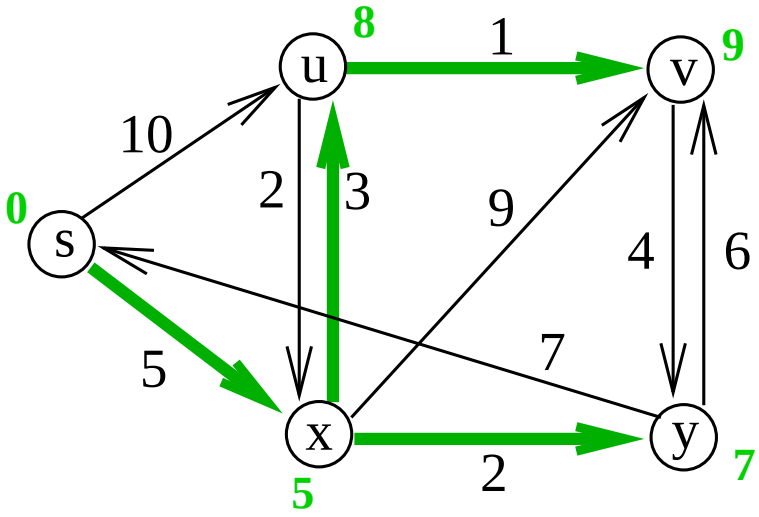
At the end of the **1-st iteration** of the main loop 1:

node	s	u	v	x	y
PARENT[.]	nil	x	u	s	x
$d[.]$	0	8	11	5	7



At the end of the **2-nd iteration** of the main loop 1:

node	s	u	v	x	y
PARENT[.]	nil	x	u	s	x
$d[.]$	0	8	9	5	7



The shortest paths and the shortest-paths weights are now computed, but the algorithm will execute the remaining two iterations of the main loop.

Correctness of the Bellman-Ford algorithm

- If there is **a negative cycle** reachable from s, then Bellman-Ford algorithm returns FALSE (because an effective relax operation will always be possible).

  That is, in this case the algorithm is correct.

- If **no negative cycle** reachable from s, then the following claim is true.

  Claim: At the termination of loop 1, $d[v] = \delta(s, v)$ for each vertex $v \in V$.

- This claim follows from the lemma given on the next slide.

- This claim implies that at the termination of loop 1, no effective relax operation is possible so the algorithm returns TRUE.

  At the end of the computation:

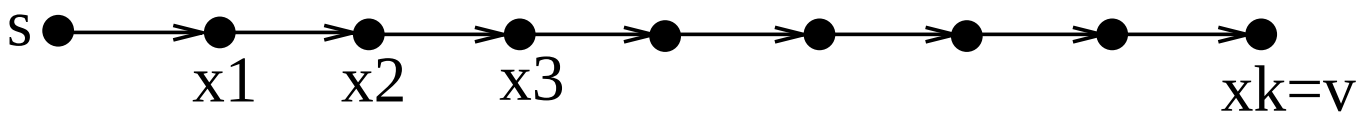
  (a) array d contains the shortest path distances from s to all other nodes;

  (b) array PARENT contains a shortest-paths tree with node s as the source (from (a) and property (9) of the relaxation technique).

- That is, in the case of "no negative cycles," the algorithm is also correct.

Correctness of the Bellman-Ford algorithm (cont.)

- **Lemma:** If the length (the number of edges) of a shortest (simple) path from s to a node v is k, then at the end of iteration k (of the main loop 1) in the Bellman-Ford algorithm, $d[v] = \delta(s, v)$.
- This lemma follows from the path-relaxation property of the relaxation technique (property (6)).
- A shortest path from s to v of lenght k (k edges on the path, $k \leq n - 1$):

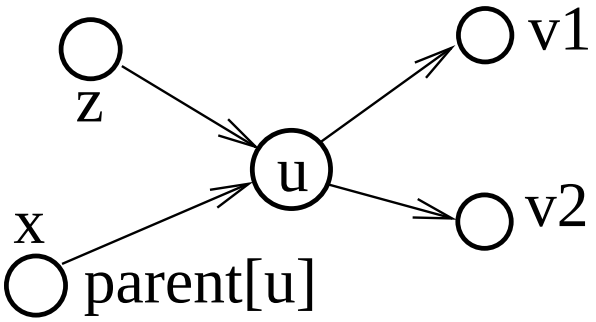


$d[x1] = \delta(s, x1)$ by the end of the 1st iteration,
$d[x2] = \delta(s, x2)$ by the end of the 2nd iteration,
$d[x3] = \delta(s, x3)$ by the end of the 3rd iteration,
$\ldots$
$d[v] = \delta(s, v)$ by the end of the k-th iteration.

- This lemma implies the claim from the previous slide, because each simple shortest path has at most $n - 1$ edges.

Speeding-up the Bellman-Ford algorithm

- Try to decrease the number of iterations of loop 1:
  - If no effective relax operation in the current iteration, then terminate: the shortest-path weights are already computed.
  - Check periodically (at the end of each iteration of loop 1?) if the PARENT pointers form a cycle. If they do, then terminate and return FALSE: there is a negative cycle reachable from s.
- Try to decrease the running time of one iteration of loop 1: consider only edges which may give effective relax operations.
  - We say a vertex u is **active**, if the edges outgoing from u have not been relaxed since the last time $d[u]$ has been decreased.



  - Perform RELAX only on edges outgoing from active vertices.
  - Store active vertices in the Queue data structure.

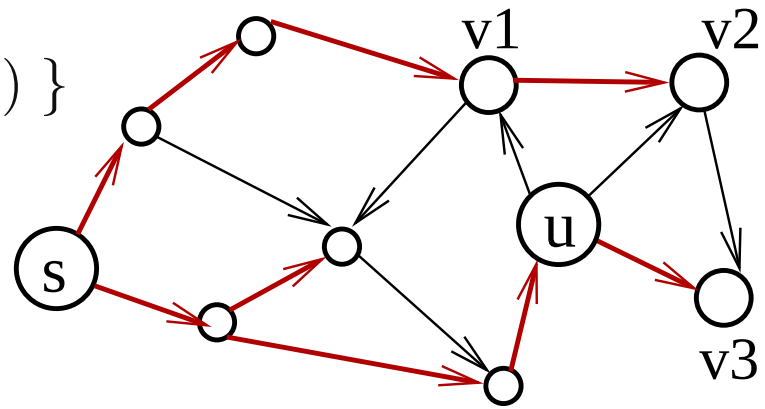
    (What is the Queue data structure?)

The Bellman-Ford algorithm with FIFO Queue

representation of input:
the array Adj of adjacency lists;
$Adj[u]$ – the list of u's neighbours

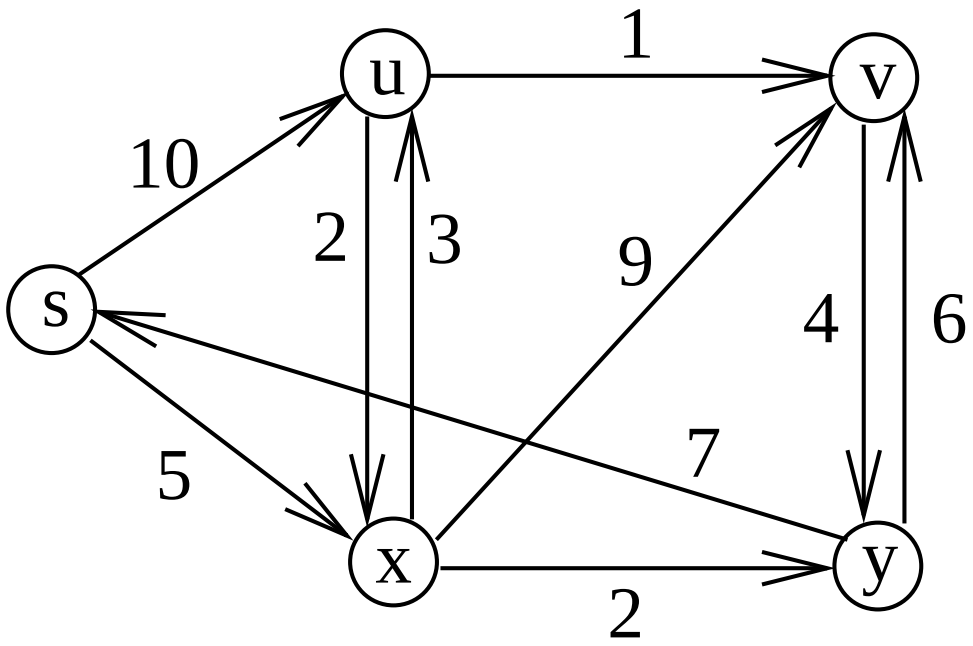
BF-WITH-FIFO(G, w, s) $\quad \{ G = (V, E) \}$
INITIALIZATION(G, s)
$Q \leftarrow \emptyset$; ENQUEUE(Q, s) $\{Q$ - FIFO queue of active vertices$\}$
while $Q \neq \emptyset$ **do**
$u \leftarrow head(Q)$; DEQUEUE(Q)
for each $v \in Adj[u]$ $\quad \{$ for each node v adjacent to u $\}$
do RELAX(u, v, w)
return TRUE $\quad \{$ for each $v \in V$, $d[v] = \delta(s, v)$ $\}$

RELAX(u, v, w):
if $d[v] > d[u] + w(u, v)$ **then**
$d[v] \leftarrow d[u] + w(u, v)$; PARENT$[v] \leftarrow u$
if v is not in Q **then** ENQUEUE(Q, v)



- A mechanism for detecting negative cycles must be added (Q never empty, if there is a negative cycle): periodically check PARENT pointers for a cycle.
- The running time: $O(mn)$, if properly implemented. But $\Theta(mn)$ worst case.

Example of the computation of Bellman-Ford with FIFO Queue – LGT



Assume that the edges outgoing from nodes are given in this order:
(s,u), (s,x);
(u,x), (u,v);
(x,u), (x,v), (x,y);
(v,y);
(y,s), (y,v).

Examples and Exercises – LGT

1. Example of the relaxation technique - slide 14.

2. Show the properties of the relaxation technique given on slide 17:

   - Property (7): For each edge (u, v) in the current parent subgraph (that is, $u = \text{PARENT}[v]$), $d[u] + w(u, v) \leq d[v]$.
   - Property (8):
     If the graph contains no negative-weight cycle reachable from s, then
     - the parent subgraph is always a tree T rooted at s,
     - for each node v in T, $d[v] \geq$ the weight of the path in T from s to v.

3. Example of the computation by the Bellman-Ford algorithm – slide 23.

4. Example of the computation of Bellman-Ford with FIFO Queue – slide 29.

Exercises – LGT (cont.)

5. Design an efficient algorithm for checking whether the array PARENT[.] represents a tree (or it has a cycle). Check how your algorithm works on these two arrays:

NODE	a	b	c	g	h	s	x	z
PARENT:	x	s	s	c	x	NIL	b	c

NODE	a	b	c	g	h	s	x	z
PARENT:	x	h	s	c	x	NIL	b	c