Optimization Methods Lecture 1

Jakub Uher

February 2020

1 Introduction

This lecture walks through an algorithm used to find single-source shortest-paths, all the basic concepts around it + Bellman-Ford Algorithm.

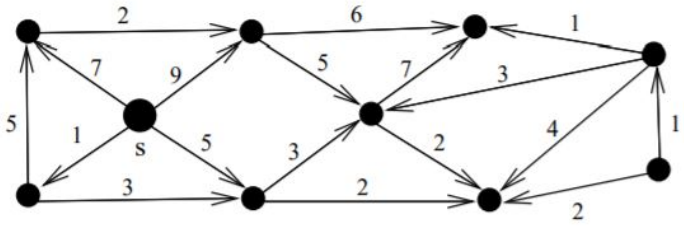


Figure 1: Single-source shortest-path problem

- $G = (V, E)$ - directed graph
- $|V| = n$, $|E| = m$
- $w(v, u)$ - the weight of edge (v, u)
- $s \epsilon V$ - the source vertex

2 Preliminaries

- A **path** $p =< v1, v2, ..., v_k, v_{k+1} >$, where (v_i, v_{i+1})
- The **weight** of a path $p =< v1, v2, ..., v_k, v_{k+1} >$:

$$w(p) = w(v_1, v_2) + w(v_2, v_3) + ... + w(v_k, v_{k+1})$$

- A **shortest-path weight(distance)** from u to v:
  $\delta(u, v) = \{$ min $w(p)$ for all p from u to v, if there is any such path. Otherwise $(-\infty, x_1]$
- A **shortest path** from u to v is any path p from u to v with weight $w(p) = \delta(u, v)$

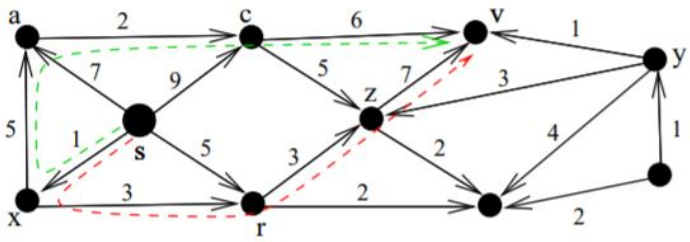


Figure 2: $\delta(s, v) = 14$, two shortest-paths from s to v. $\delta(s, y) = +\infty$

Useful facts:

- A sub-path of a shortest path is a shortest path. In the graph on the previous slide: path $< x, r, z >$ is a sub-path of a shortest path $< s, x, r, z, v >$, so it must be a shortest path (from x to z).
- Triangle inequality: for each edge (u, v) ϵ E

$$\delta(s, v) \leq \delta(s, u) + w(u, v)$$

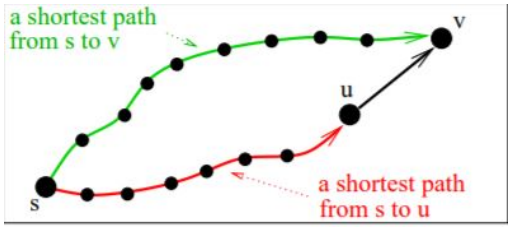


Figure 3: Holds also if u and v is not reachable from s.

- We first consider the general case: the weights of edges may be negative

3 Negative weights in an application

Example from financial analysis: a graph of exchange rates.

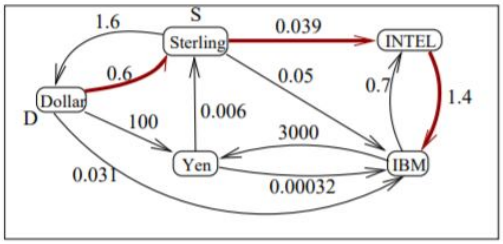


Figure 4: Find paths maximising exchange rates

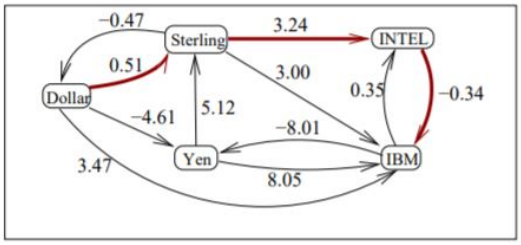


Figure 5: Find shortest paths

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

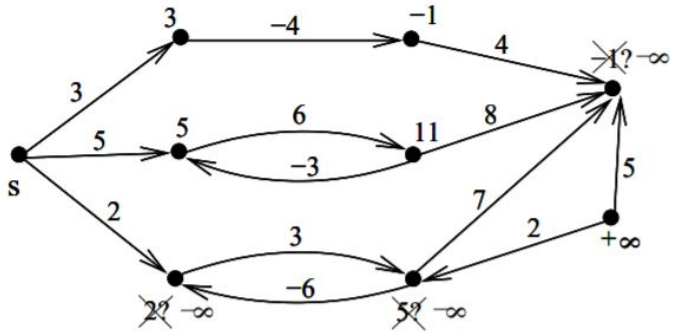
  if $\gamma(x, y)$ ¿ 1, then $w(x, y)$ ¡ 0.

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

3.1 The exchange-rates example

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.

4 Negative-weight cycles (negative cycles)



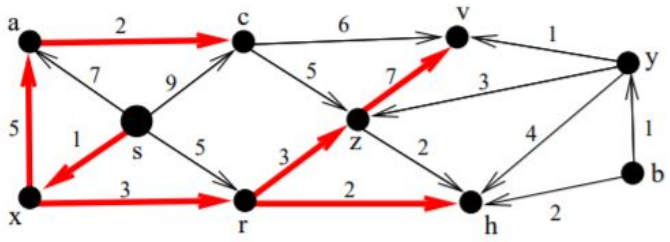
- If there is a negative cycle on a path from s to v, then by going increasingly many times around such a cycle, we get paths from s to v of arbitrarily small (negative) weights.
- If there is a negative cycle on a path from s to v, then by convention, $\delta(s, v) = -\infty$

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

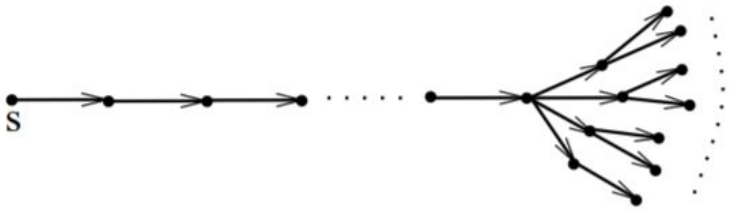
- We consider only shortest-paths algorithms which:
  - compute shortest paths for graphs with no negative cycles reachable from the source
  - for graphs with negative cycles reachable from the source, correctly detect that the input graph has a negative cycle (but are not required to compute anything else)
- If there is a negative cycle, then why don't we look for the shortest simple paths (not containing cycles)?
- This would be a valid, well-defined (there are always simple shortest-paths) 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 a $NP-hard$, that is computationally difficult (by reduction from the $NP-hard$ Hamiltonian Path problem), algorithms from lecture 1-3 don't apply.

5 Representation of computed shortest paths: A shortest-paths tree

- For each node v reachable from s, we want to find just one shortest path from s to v.
- If there are no negative cycles reachable from s, then shortest paths from s to all nodes reachable from s are all 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.

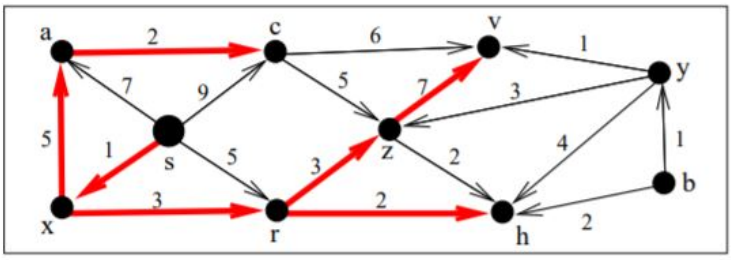


- A shortest-paths tree from s contains **exactly one shortest path from s to each v reachable from s.** There may be other shortest paths from s to v, which are not included in this tree.
- A shortest-paths tree T can be represented by an array
  $PARENT[v]$, $v \in V$
  In $\theta(n)$ space, where 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 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.



6 Output of a shortest-paths algorithm

- A shortest-paths algorithm computes the shortest-paths 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 the 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

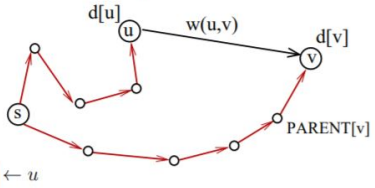
- Terminology and notation in book:
  - ($parent$, $PARENT[v]$, $d[v]$) $->$ ($predecessor$, $v.\pi$, $v.d$.

6.1 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. $d[v]$ array at the end of computation this array should store the weights of shortest path weights
  - $PARENT[v]$: the current predecessor of node v,
- The **relaxation technique**:
  Starting from s we initialise the array $d[s]$ to 0 or infinity if there is a self-loop, and all the other notes to positive infinity because they may not be reachable at all.

INITIALIZATION(G, s) followed by a sequence of RELAX operations.

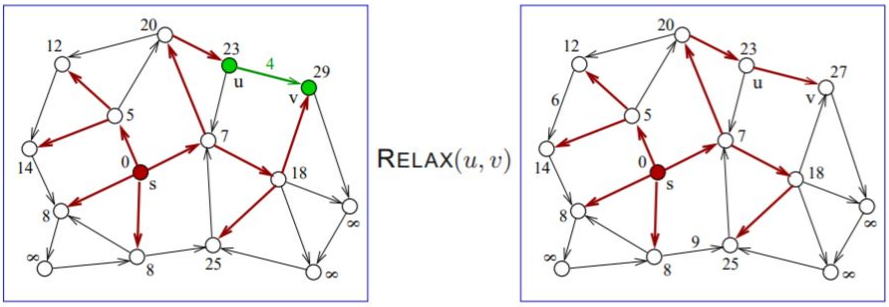
```
INITIALIZATION(G, s)
    d[s] ← 0; PARENT[s] ← NIL
    for each node v ∈ V − {s} do
        d[v] ← ∞; PARENT[v] ← NIL
RELAX(u, v, w)  { relax edge (u, v) }
    if d[v] > d[u] + w(u, v) then
        d[v] ← d[u] + w(u, v); PARENT[v] ← u
```



Relaxing an edge means applying relax operation to a given edge, we are trying to improve the shortest path estimate at v using the current shortest path estimate at node u.So relaxing operation checks if we are improving then $PARENT[v]$ updated with the better way

- All algorithms which we discuss in this module are based on the relaxation technique (most of the shortest-paths algorithms do). They differ in the order in which they relax edges. When should the computation stop?

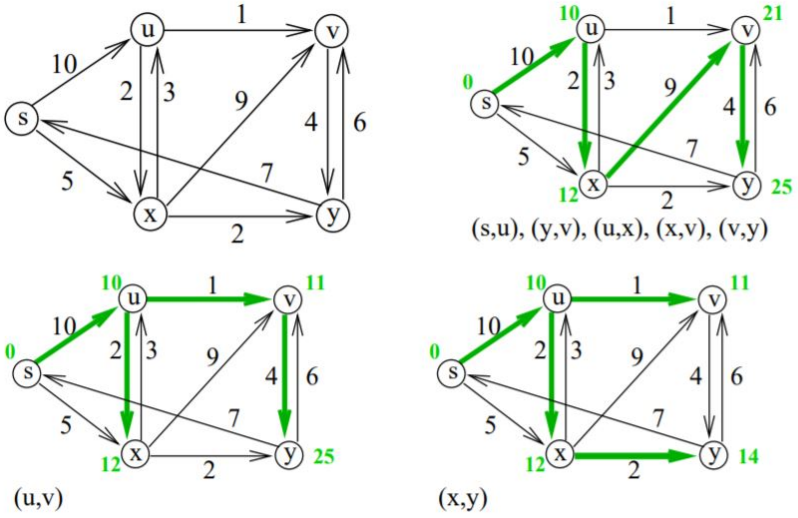
7 Relax operation



This shows how the edge $7- > 18- > 29$ can be improved (relaxed) by following the other path $7- > 20- > 23- > 27$

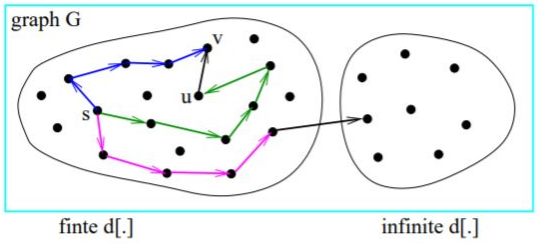
7.1 Example

- The first relaxed operation would improve shortest path estimate at node u from ∞ to 10 and would mark the edge as the current edge of u in $PARENT$ array. And so on.
- When applying relax operation from (u, v) there is a space for improvement from 21 to 11 when using edge between (u, v) shortest path estimate at node v can be improved by 10.
- The last step is to relax (x, y) by using edge between (x, y) which decreases shortest path estimate of node y by 11.



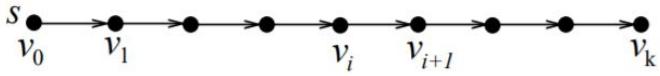
7.2 Properties of the relaxation technique

- **Non-increasing shortest-path estimates**. Throughout the computation, for each node v, the shortest-path estimate $d[v]$ can only decrease **(it never increases)**.
- 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.
- **Upper bound property** For each node v, we always have $d[v] \geq \delta(s, v)$.
- **No-path property**. If there is no path from s to v, then we always have $d[v] = \delta(s, v) = \infty$ (Equivalently, if $d[v]$ becomes finite, then there must be a path from s to v.)



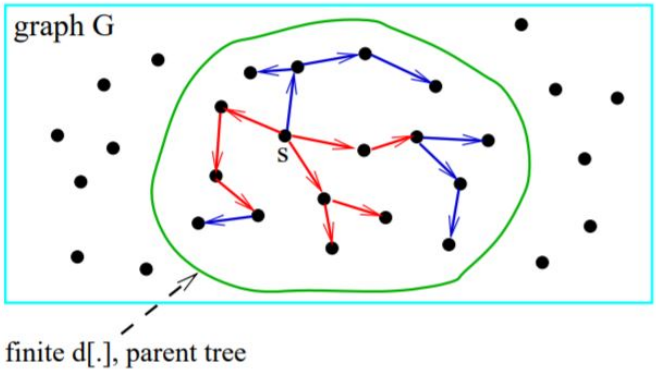
- **Convergence property**. If $(s, ..., u, v)$ is a shortest path from s to v and if $d[u] = \delta(s, u)$ at any time prior to relaxing edge (u, v), then $d[v] = \delta(s, v)$ at all times afterward. The progress is there to be made

- **Path-relaxation property**. if $p = (v_0, v_1, ..., v_k$, 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), ..., (v_{k-1}, v_k) then $d[v_k] = \delta(s, v_k)$.
This property holds regardless of any other relaxation step that occur, even if they are intermixed with relaxations of the edges of p.

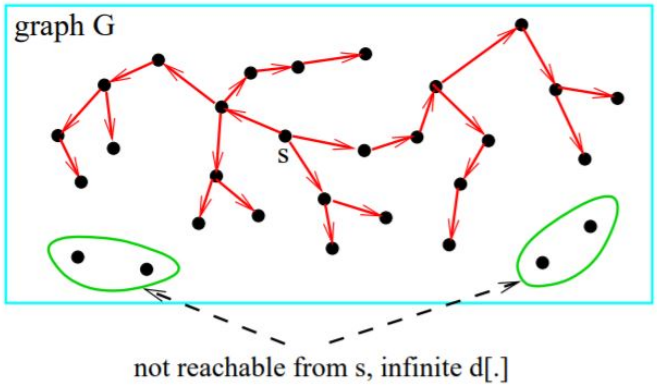


7.3 Relaxation technique: Parent sub-graph properties

- if the graph contains no negative-weight cycle reachable from s. then the parent graph (formed by the edge $(PARENT[v], v)$) is always a rooted tree with root s.



- The red part of the current parent tree is already part of the computed shortest-paths tree.
- The blue par may change during the subsequent relax operations.
- Once $d[v] = \delta(s, v)$ for all $v\epsilon$ V, the parent sub-graph is a shortest-paths tree rooted as s



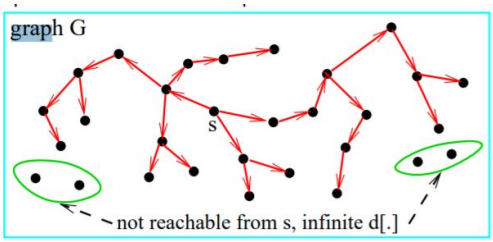
7.4 Effective relax operations

- **This relax operation always decreases** $d[v]$
- An **effective relax operation** is a relax operation which decreases $d[v]$.
- The computation can keep progressing, if the s.p. weights not reached yet:
  - There exists a vertex $x \epsilon E$ such that $d[x]$ ¿ $\delta(s, x)$,(This means that the estimate is higher then the actual weights)
  - There exists an edge (u, v) ϵ E such that $d[v] > d[u] + w(u, v)$ (that is, another effective relax operation is possible).

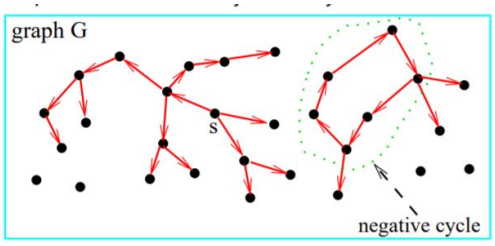
7.5 The relaxation technique: summary for the case of no negative cycles

Repeat until there is no more effective relax operation

- If **no negative cycle** is reachable from s:
  - There may be only finitely many effective relax operations throughout the computation. (Finitely many different parent trees and the parent tree has to be updated after finitely many effective relax operations.)
  - The PARENT sub-graph is always a tree rooted at s.
  - When eventually no effective relax operation possible, then for each node v, $d[v] = \delta(s, v)$, and the PARENT pointers from a shortest-paths tree.



- 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).
  - The PARENT will eventually form a cycle



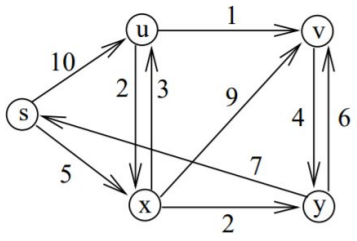
This implies that we can detect the existence of a negative cycle (and halt the computation) by periodically checking if the PARENT pointers form a cycle.

7.6 The Bellman-Ford algorithm

- Edge weights may be negative.

```
BELLMAN-FORD(G, w, s)                    { G = (V, E) }
    INITIALIZATION(G, s)
1: for i ← 1 to n − 1 do                 { n = |V| }
       for each edge (u, v) ∈ E do       { edges considered in arbitrary order}
           RELAX(u, v, w)
2: for each edge (u, v) ∈ 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 ∈ V, d[v] = δ(s, v) }
```

- This algorithm is based on the relaxation technique: INITIALIZATION(G,s) 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.
- The worst case running time of any algorithm for the single-source shortest paths problem with negative weights is $\Omega(mn)$



Assume that the edges are given in this order:

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

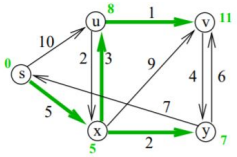
Initially:
(relaxation technique initialization)

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

7.7 Example of the computation by the Bellman-Ford algorithm

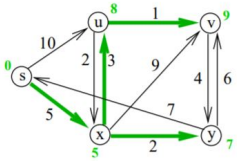
At the end of the **1-st iteration** of the main loop 1:

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.

7.8 Correctness of the Bellman-Ford algorithm

- **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] = $\delta(s, v)$
- The lemma follows from the path-relaxation property of the relaxation technique.
- A shortest path from s to v of length 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,
...
$d[v] = \delta(s, v)$ by the end of the k-th iteration.

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

7.9 Speeding-up the Bellman-Ford algorithm

- Try to decrease the number of iterations of loop 1:
  - If no effective relax operation in the current interaction, then terminate: the shortest-path weights are already computed.
  - Check periodically (at the end of each iteration of loop 1?) if the PARENT pointers from 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.
  - 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.

7.10 The Bellman-Ford algorithm with a FIFO queue

```
BF-WITH-FIFO(G, w, s)          { G = (V, E) }
  INITIALIZATION(G, s)
  Q ← ∅;  ENQUEUE(Q, s)   {Q - FIFO queue of active vertices}
  while Q ≠ ∅ do
      u ← head(Q);  DEQUEUE(Q)
      for each v ∈ Adj[u]     { for each node v adjacent to u }
          do RELAX(u, v, w)
  return TRUE    { for each v ∈ V, d[v] = δ(s, v) }
RELAX(u, v, w):
      if d[v] > d[u] + w(u, v) then
          d[v] ← d[u] + w(u, v);  PARENT[v] ← u
          if v is not in Q then ENQUEUE(Q, v)
```