Chapter 4 Paths in graphs

Joe Song

September 23, 2019

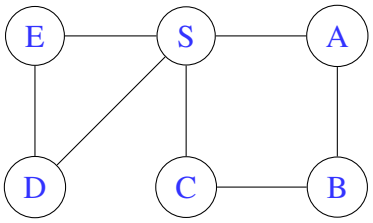
The notes are based on Chapter 4 of Dasgupta, Papadimitriou and Vazirani. Algorithms. 2008. McGraw-Hill. New York.

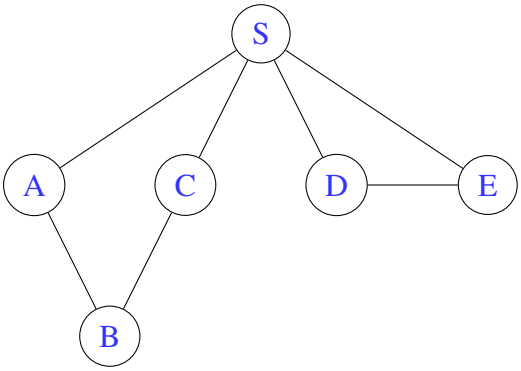
Contents

1 Distances

The *distance* between two nodes is the length of the shortest path between them.

2 Breadth-first search





The idea: Explore nodes by their distance to the source node. Maintain nodes to-be-explored in a queue.

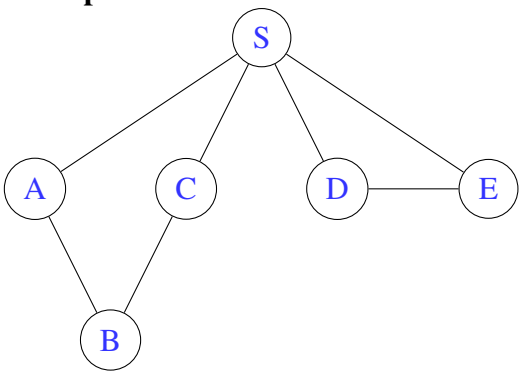
---

procedure bfs(G, s)
Input: Graph $G = (V, E)$, directed or undirected, vertex $s \in V$
Output: For all vertices u reachable from s, dist[u] is set to the distance from s to u

```
for all u ∈ V:
    dist(u) = ∞

dist(s)=0
Q = [s] (queue containing just s)
while Q is not empty
        u =eject(Q)
        for each edge (u,v) ∈ E:
            if dist(v)=∞
              inject(Q,v)
              dist(v) = dist(u) + 1
```

Example:



Order of visit	Queue after processing node
	[S]
S	[A C D E]
A	[C D E B]
C	[D E B]
D	[E B]
E	[B]
B	[]

Correctness: bfs(G, s) explores all nodes reachable from s with correct distance in both directed & undirected graphs.

For each $d = 0, 1, 2, \ldots$, there is a moment at which

1. all nodes at distance $\leq d$ from s have their distances correctly set;
2. all other nodes at distance $> d$ have their distances set to ∞; and
3. the queue contains exactly the nodes at distance d.

Proof. (By induction)

Base case: Before 1st while iteration, the queue contains only node s with distance $d = 0$.

Inductive step: Assume statements (1–3) true at $d-1$ levels and the queue contains all nodes of distance $d-1$ to s at time t_0:

$$t_0 : [d-1, d-1, \ldots, d-1]$$

Now let bfs() continue. After exactly the last node in the above queue with $d-1$ is processed, let the time be t. We must have the following:

- By definition, all nodes with distance d must be neighbors of those nodes with distance $d-1$;

  By the algorithm, all non-infinity neighbors of nodes with distance $d-1$ must have distance d.

  Also at time t_0, all nodes with distance $d-1$ or less were already set correctly. Together statement (1) is true at time t for all nodes with distance $\leq d$;

- All nodes with distance $> d$ were not touched and remain labeled with ∞ according to (2) at t_0; and thus statement (2) is correct at t;

- By the algorithm, neighbors of nodes with distance $d-1$ can only be d when they were injected into the queue, so the queue contains only nodes with distance d—statement (3) is true at time t. That is, the distances of nodes in the queue are

$$t : [d, d, \ldots, d]$$

□

Running time: Linear, i.e., $O(|V|+|E|)$

- Each node is put into the queue once and removed from the queue once.
- Each edge is visited exactly once (in directed graphs) or twice (in undirected graphs)

DFS versus BFS

- DFS has special properties regarding previsit and postvisit times
- DFS may take a longer route to reach a node than BFS
- DFS can be done iteratively using a stack, while iterative BFS uses a queue

3 Lengths on edges

Lengths of edges have been assumed to be 1 in DFS and BFS so far.

Now we define a length l_e for every edge $e \in E$. If $e = (u, v)$, the edge length can also be written as $l(u, v)$ or l_{uv}.

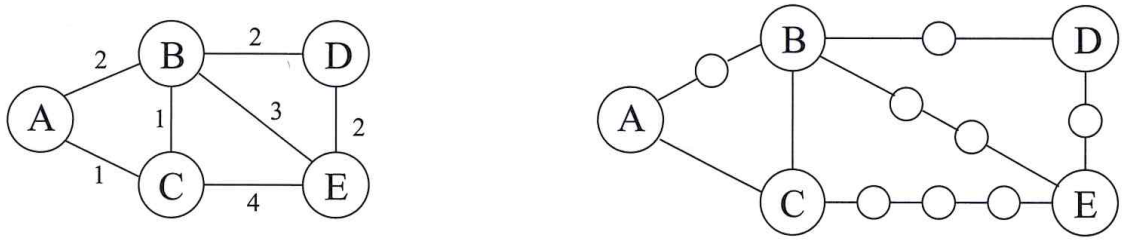
Length can mean time, cost, etc. They can be negative in some graphs.

4 Dijkstra's algorithm—Shortest paths in positively weighted graphs

$$$ Idea: Can we adapt the BFS algorithm to a more general graph $G = (V, E)$ whose edge lengths l_e are positive integers?

Extension of the BFS algorithm by adding a node for every unit length

Figure 4.6 Breaking edges into unit-length pieces.



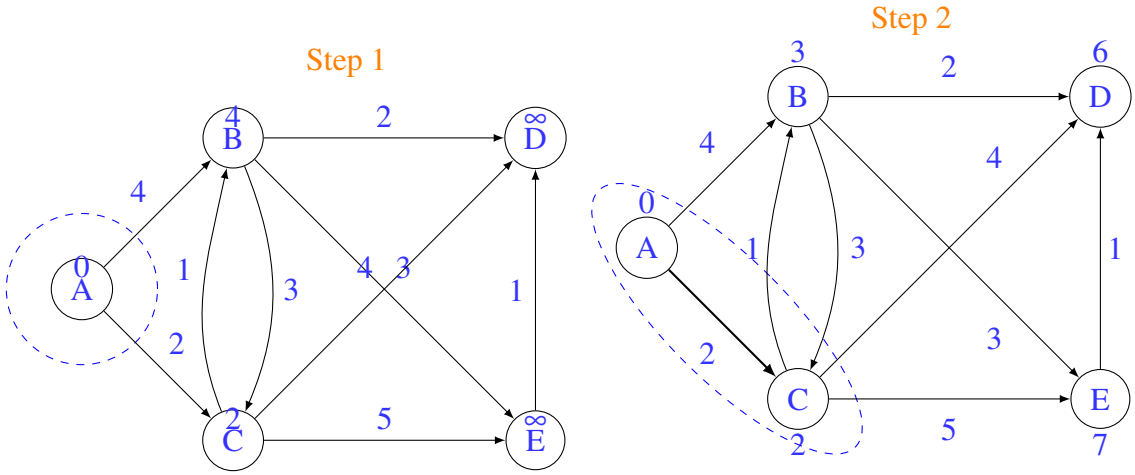
Problem: inefficient when edges have a large range of integer weights.

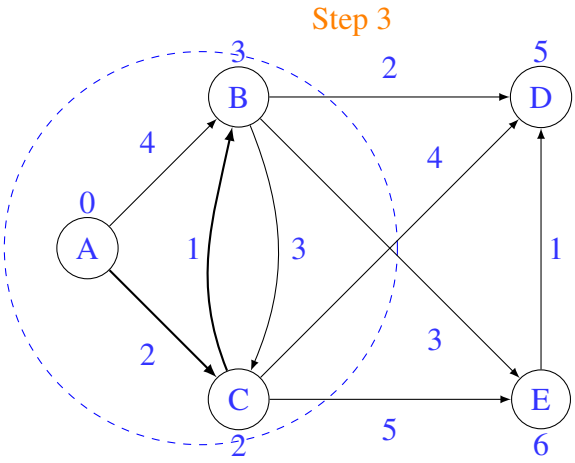
Alarm clock algorithm

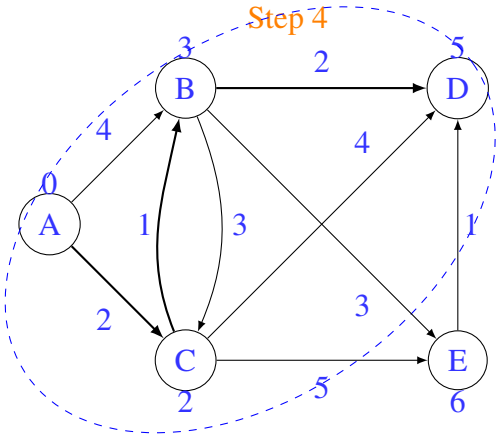
- Set an alarm clock for node s at time 0
- Repeat the following until no more alarms are left:

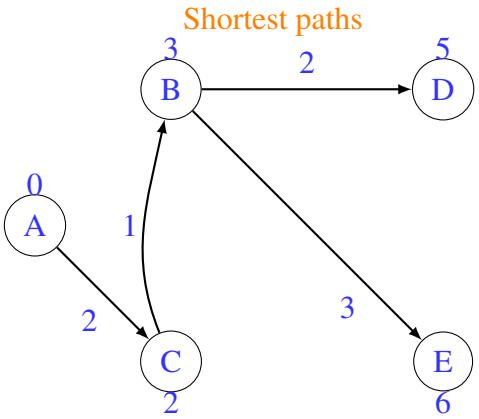
  Say the next alarm goes off at time T, for node u. Then:

  - The distance from s to u is T.
  - For each neighbor v of u in G:
    - If there is no alarm yet for v, set one for time $T + l(u,v)$.
    - If v's alarm is set earlier for a time later than $T + l(u,v)$, then reset it to $T + l(u,v)$.









Priority queue is a data structure that supports the following operations:

Insert: Add an element to the queue

Decrease-key: Accommodate the decrease in key value of a particular element

Remove-min: Return the element with the smallest key, and remove it from the set

Make-priority-queue: Build a priority queue out of the given elements with the give key values.

Procedure Dijkstra(G, l, s)
Input: Graph $G = (V, E)$, directed or undirected;
l positive edge lengths for each edge;
vertex $s \in V$

Output: For all vertices u, reachable from s, dist(u) is set to the distance from s to u

for all $u \in V$:
    dist(u) = ∞
    prev(u) = NIL

dist(s)=0
H = make-queue(V) (use distance values as keys)

while H is not empty:
        u =delete-min(H)
        for each edge $(u,v) \in E$:
            if dist(v) $>$ dist(u) + $l(u,v)$:
                dist(v) = dist(u) + $l(u,v)$
                prev(v) = u
                decrease-key(H,v)

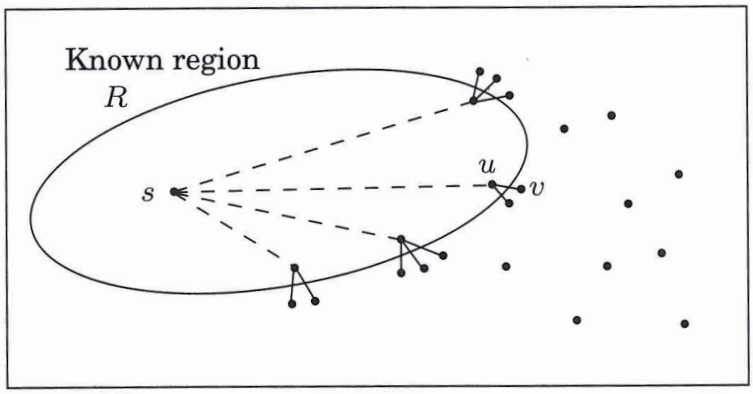
Dijkstra algorithm can be considered similar to the BFS, except that

- the Dijkstra algorithm uses a priority queue to arrange the nodes to be explored
- the BFS uses a regular queue to arrange the nodes to be explored

An alternative derivation

$$$ Idea: Grow a region of nodes R which are closer to the source node s in distance; then expand the region by including closest nodes to the region.

Figure 4.10 Single-edge extensions of known shortest paths.



How to identify the closest node v? It must be a node v that

- with the smallest distance to s
- has a single edge (u,v) extension from a known shortest path from s to u. (Single edge: the path from u to v has a single edge)

Proof of the above statement by contradiction: As in the following picture, we can always identify a node u that immediately precedes v on a shortest path from s to v if dist(v) is not ∞.

Suppose u was not in the region R. Then dist(u) $<$ dist(v) contradicts that v is a node with the smallest distance outside R. So u must be in R and (u,v) is a single edge that extends from u in R to v.

Conceptual outline of growing the region R:

Initialize dist(s) to 0, other dist() values to ∞
$R = \{\}$ (the "known" region)
while $R \neq V$:
    Pick the node $v \notin R$ with the smallest dist($\cdot$)
    Add v to R
    for each edge $(u,v) \in E$:

if dist(v) $>$ dist(u) + $l(u,v)$:
  dist(v) = dist(u) + $l(u,v)$

Correctness proof by induction:

Induction hypothesis: At the end of each iteration of the while-loop, the following conditions hold:

1. there is a value d such that all nodes in R are at distance $\leq d$ from s and all nodes outside R are at distance $\geq d$ from s, and
2. for every node u in R, the value dist(u) is the length of the shortest path from s to u whose intermediate nodes are constrained to be in R (if no such path exists, the value is ∞)

Proof.

Base case: After the first while-loop, $R = \{s\}$.

1. $d = 0$. All nodes in R, which is s, have distance $0 \leq 0$; all nodes not in R, have distance $\infty \geq 0$.

2. dist(s)=0 is true obviously, representing the distance from s to s.

Induction step: Assume the induction hypothesis is true at the end of a while-loop. After we finish one more while-loop, we have the following:

1a. All nodes in R will have distance $\leq d \leq$ dist(v), as v was picked from outside R at the end of the previous while-loop.

1b. All nodes outside R will have distance $\geq$ dist(v), as v has the smallest distance when it was picked.

1c. Therefore the new d value will be dist(v) after the while-loop.

2a. As we have shown earlier, the path from s to v must be one-edge extension of some other path $s \rightsquigarrow u$ for some $u \in R$. Therefore, all intermediate nodes from s to v must be in R.

2b. When u was processed, node v must have the distance dist(v) set correctly, as it is impossible to have another path from s to v that is shorter than dist(v)

Termination: After the last while-loop, all nodes are in R and the distance are thus correctly set.

Running time

- $|V|$ `insert` operations in `make-queue`
- $|V|$ `delete-min` operations, one for each node
- $|E|$ `decrease-key` operations

Depending on the data structure used for the priority queue, the total running time will vary. Not all of the operations can be done in constant time typically. The following table summarizes different running time involved:

Data structure	`delete-min`	`insert`/ decrease-key	$\|V\|\times$ `#delete-min` + $\|V\|\times$ `#insert` + $\|E\|\times$ `#decrease-key`
Array	$O(\|V\|)$	$O(1)$	$O(\|V\|^2)$
Binary heap	$O(\log \|V\|)$	$O(\log \|V\|)$	$O((\|V\|+\|E\|)\log \|V\|)$
d-ary heap	$O(\frac{d\log \|V\|}{\log d})$	$O(\frac{\log \|V\|}{\log d})$	$O((\|V\|\cdot d+\|E\|)\frac{\log \|V\|}{\log d})$
Fibonacci heap	$O(\log \|V\|)$	$O(1)$ (amortized)	$O(\|V\|\log \|V\|+\|E\|)$

5 Priority queue implementation

5.1 Binary heap

One data structure that supports all operations of a priority queue:

- a binary heap is a complete binary tree – all levels must be filled to the full capacity except the bottom level.
- parent value is less than or equal to those of the children
- can be represented by an array (1-based):

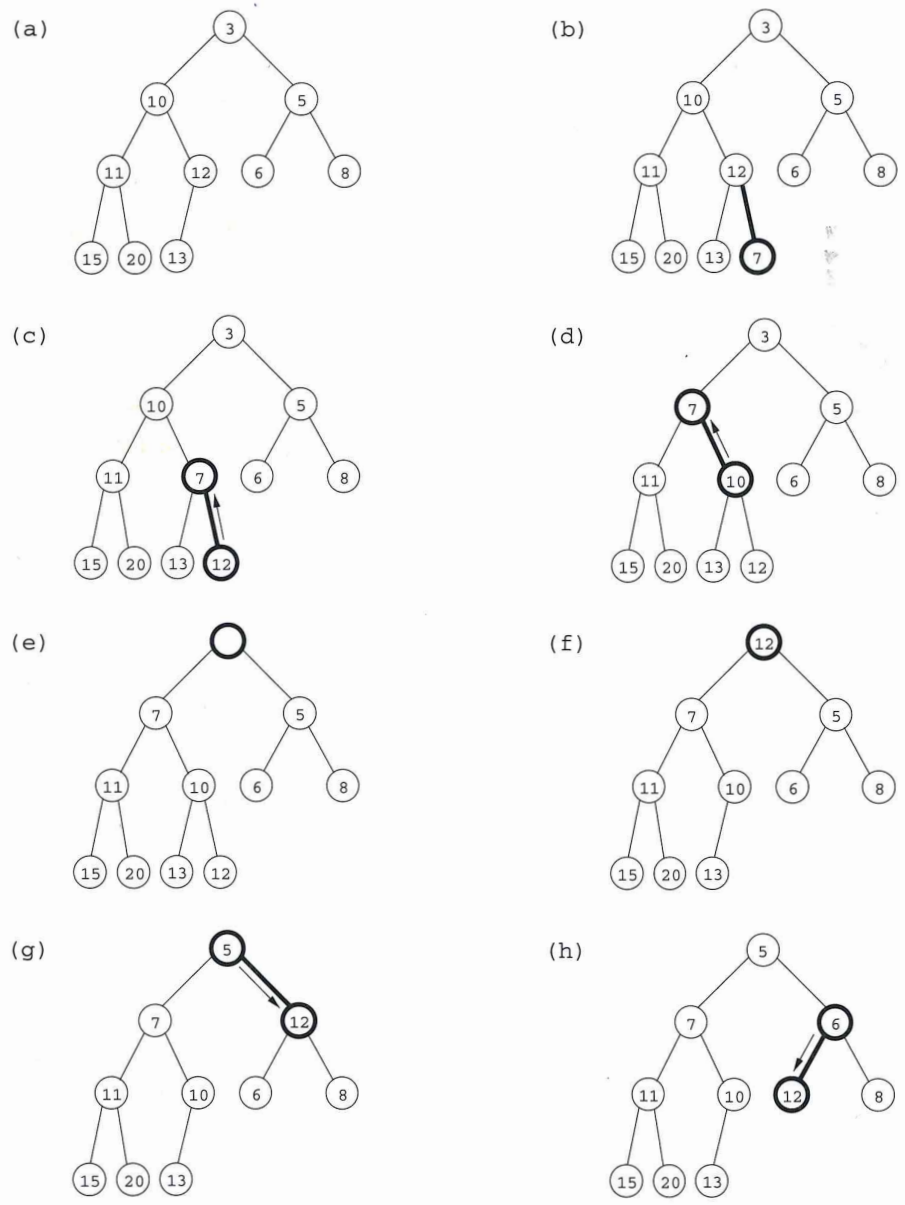
  node at location j has

  parent $\lfloor j/2 \rfloor$

  children $2j$ and $2j+1$

  Example: 3, 10, 5, 11, 12, 6, 8, 15, 20, 13

Figure 4.11 (a) A binary heap with 10 elements. Only the key values are shown. (b)–(d) The intermediate "bubble-up" steps in inserting an element with key 7. (e)–(g) The "sift-down" steps in a delete-min operation.



If we use an binary heap for the following operations, we have

Insert: Insert the the end of the array and Bubble-up, $O(\log n)$

Decrease-key: Bubble-up, $O(\log n)$

Remove-min: Move the last element to the top and Bubble-down, $O(\log n)$

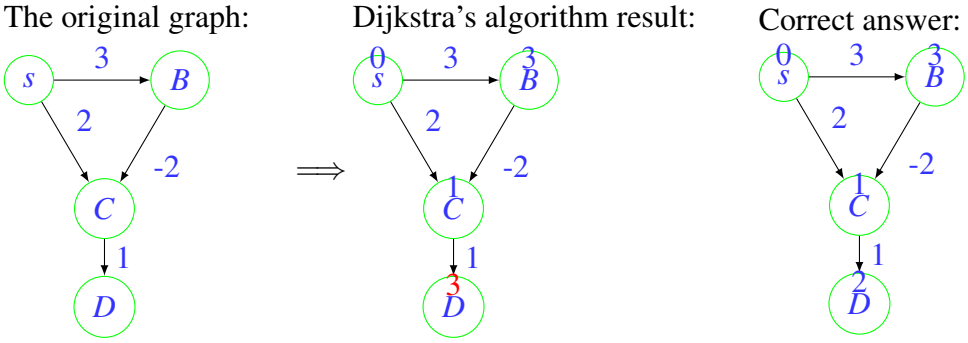
Make-queue: By n insertions, $O(n \log n)$

6 Shortest paths in general weighted graphs

Assumptions on edge lengths in a graph:

- Edge can have negative weight
- No negative cycles are allowed

Dijkstra's algorithm does not always work when some edges have negative weights in a graph. An example where Dijkstra's algorithm does not work is given below:



The distance from s to D was wrong by the Dijkstra's algorithm (middle). The distance of C was 2 when removed from the priority queue but was correctly updated to 1 when B was processed. The correct distances are shown on the right.

The counter example of Figure 4.12 on PAGE 117 DPV 2006 is wrong. The weight of edge (A,B) should be -2, not 2.

---

procedure update$((u,v) \in E)$
$\text{dist}(v) = \min\{\text{dist}(v), \text{dist}(u) + l(u,v)\}$

Properties:

1. It gives the correct distance to v in the particular case where u is the second-last node on a shortest path from s to v and when dist(u) is correctly set.
2. It will never make dist(v) smaller than the actual distance to v, and in this sense it is *safe*.

$$$ Idea: Can we operate using a special sequence of nodes, different from the Dijkstra's, so that we can handle negative edges?

s — u_1 — u_2 — u_3 - - - - - - - - - - - u_k — t

Observations:

1. A shortest path has at most $|V|-1$ edges, assuming the graph has no negative cycles.
2. Updating in the order of edges on the path would give the right distance to v, if the path is known
3. Updating all edges $|V|-1$ times will hit all the shortest paths, even if the paths are unknown

---

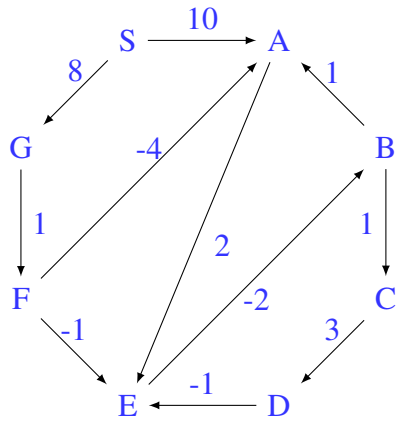
procedure Bellman-Ford(G, l, s)
Input: Graph $G = (V, E)$, directed or undirected; no negative cycles
l lengths for each edge
vertex $s \in V$
Output: For all vertices u reachable from s, dist(u) is set to the distance from s to u

for all $u \in V$:
dist(u) = ∞
prev(u) = NIL

dist(s)=0

repeat $|V|-1$ times:
for each edge $e = (u, v) \in E$
update(e)

Example:



Node	0	1	2	3	4	5	6	7
S	0	0	0	0	0	0	0	0
A	∞	10	10	5	5	5	5	5
B	∞	∞	∞	10	6	5	5	5
C	∞	∞	∞	∞	11	7	6	6
D	∞	∞	∞	∞	∞	14	10	9
E	∞	∞	12	8	7	7	7	7
F	∞	∞	9	9	9	9	9	9
G	∞	8	8	8	8	8	8	8

Practical concerns:

- Stop early when no change happens after a round of update for the edges
- Check one more time after the last round ($|V|-1$), if and only if distance to any node continuous to decrease, then negative cycle exists.

7 Shortest paths in dags

Observation: In any path of a dag, the vertices appear in increasing linearized order.

Idea: linearize the nodes by DFS (topological-sort); update the edges out of each node in the linear order.

---

procedure Shortest-path-in-dag(G, l, s)
Input: Dag $G = (V, E)$
l lengths for each edge;
vertex $s \in V$
Output: For all vertices u reachable from s, dist(u) is set to the distance from s to u

for all $u \in V$:
dist(u) = ∞
prev(u) = NIL

dist(s)=0

Linearize G (by DFS first and sorting nodes in decreasing order of *post* time)

for each $u \in V$, in linearized order:
for each edge $(u, v) \in E$
update((u, v))
