Shortest Paths

Outline

- Shortest Paths
- Single Source Shortest Path
  - Bellman-Ford Algorithm
  - Shortest Paths in a DAG

Shortest Path Algorithms

Algorithm	Comments	Graph Rep	Running Time	Space Used
Bellman-Ford	Single-Source	Adj List	$O(VE)$	$O(V)$
In DAG	Single-Source DAG	Adj List	$\mathrm{O}(V+E)$	$O(V)$
Dijkstra	Single-Source Non-Neg Weights	Adj List	$O(E \log V)$	$O(V)$
All-Pairs 1	All-Pairs	Adj Matrix	$O(V^4)$	$O(V^2)$
All-Pairs 2	All-Pairs	Adj Matrix	$O(V^3 \log V)$	$O(V^2)$
Floyd-Warshall	All Pairs	Adj Matrix	$O(V^3)$	$O(V^2)$

Space Used is in addition to space required to store the graph. For simplicity, we use V and E to denote $|V|$ and $|E|$, respectively, in complexity bounds.

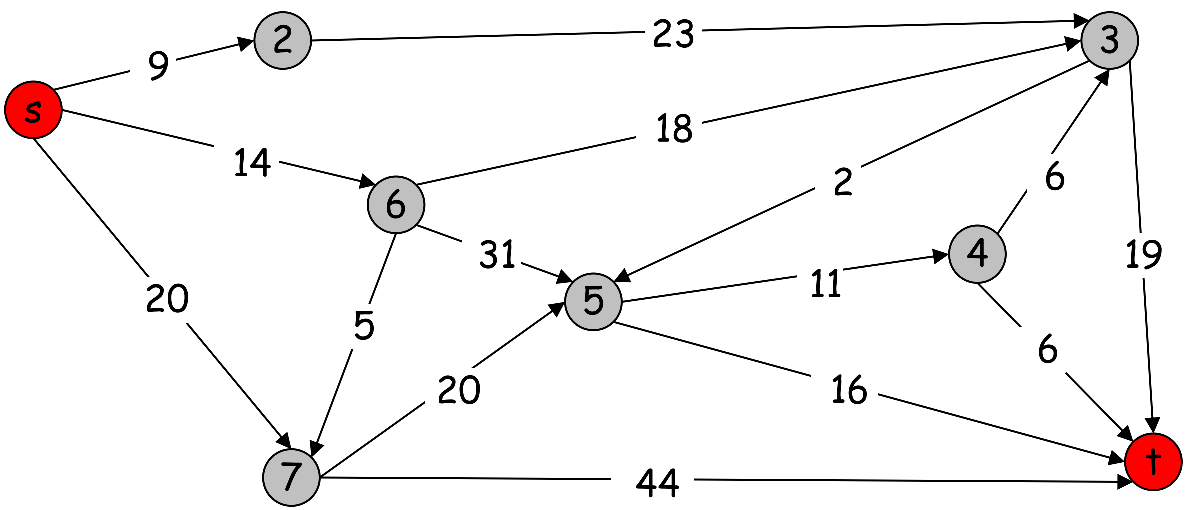
Shortest Path Problem

Input:

- Directed graph $G = (V, E)$.
  - An undirected edge can be considered as two directed edges.
- Source s, destination t.
- Weight $w(e)$ = length of edge e ($w(e)$ can be negative)

Shortest path problem: Find the shortest path from s to t.

Def: $\delta(u, v)$, the distance from u to v, is the length of the shortest path from u to v.



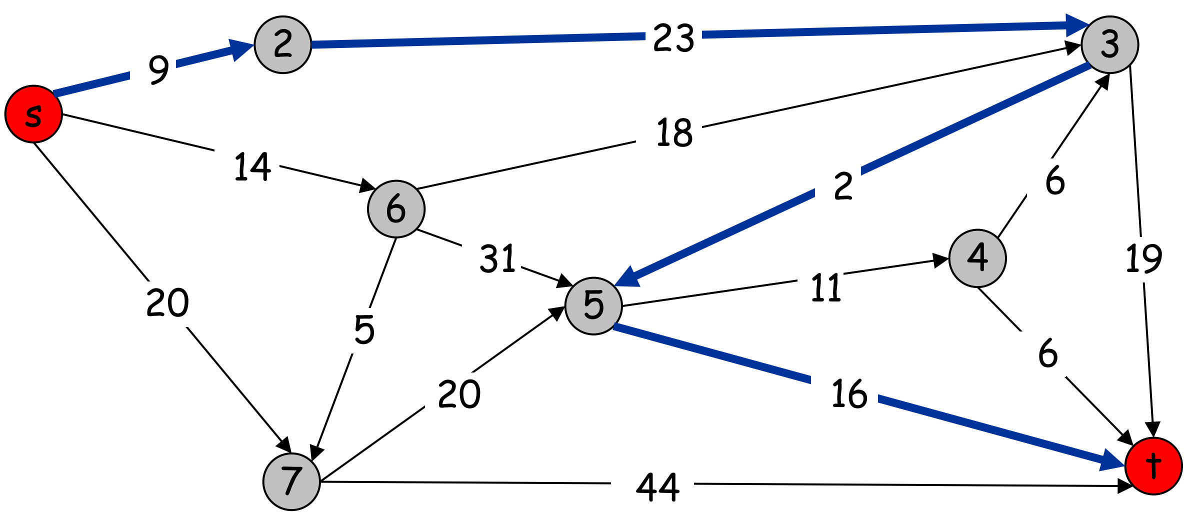
Shortest Path Problem

Input:

- Directed graph $G = (V, E)$.
  - An undirected edge can be considered as two directed edges.
- Source s, destination t.
- Weight $w(e)$ = length of edge e ($w(e)$ can be negative)

Shortest path problem: Find the shortest path from s to t.

Def: $\delta(u, v)$, the distance from u to v, is the length of the shortest path from u to v.



$\delta(s,t)$ = 9 + 23 + 2 + 16
= 50.

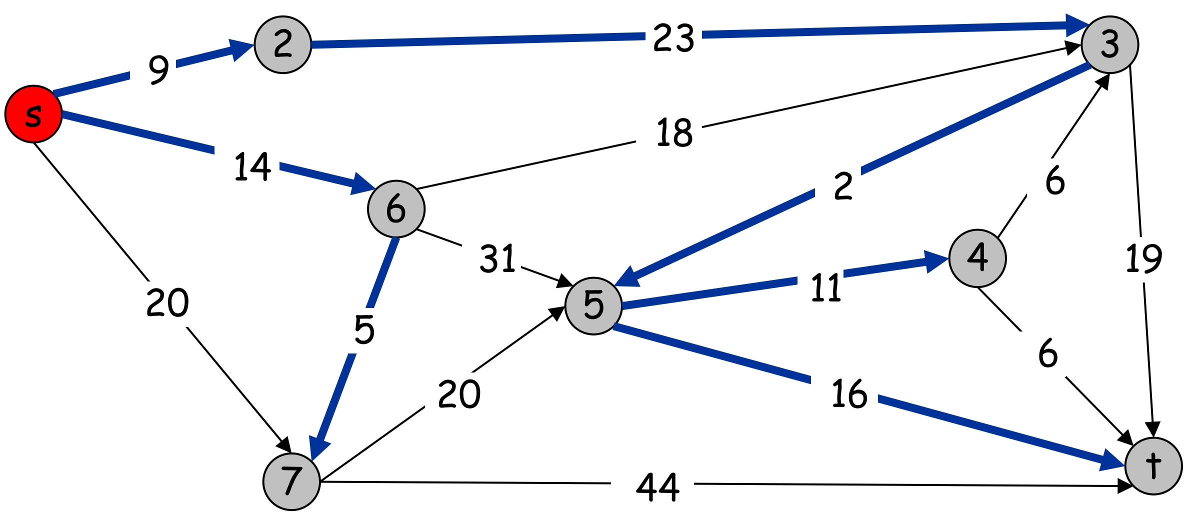
Shortest Path Problem

Input:

- Directed graph $G = (V, E)$.
  - An undirected edge can be considered as two directed edges.
- Source s, destination t.
- Weight $w(e)$ = length of edge e ($w(e)$ can be negative)

Shortest path problem: Find the shortest path from s to t.

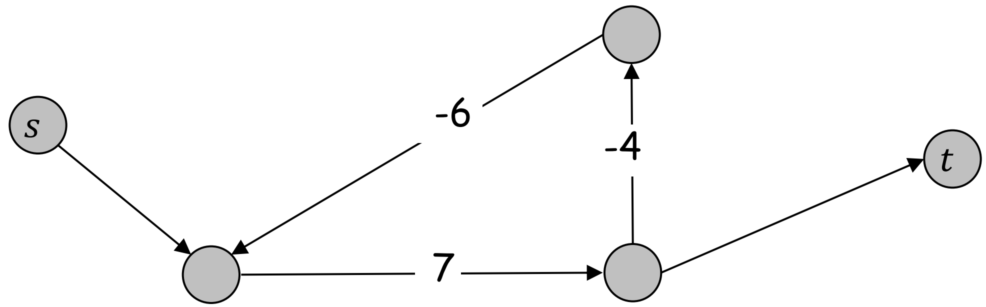
Single-source shortest path: Find the shortest path from s to every node.

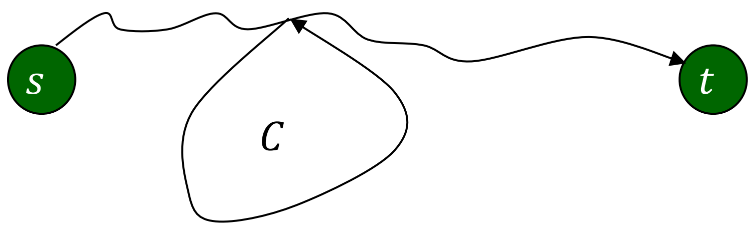


x	Shortest path from s to x	$\delta(s, x)$
2	s,2	9
3	s,2,3	32
4	s,2,3,5,4	45
5	s,2,3,5	34
6	s,6	14
7	s,6,7	19
t	s,2,3,5,t	50

Shortest Paths: Negative Weight Cycles

Negative weight cycles.





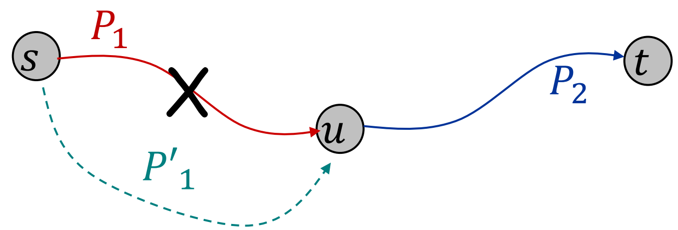
$w(C) < 0$

Note. The shortest path problem is not well defined if the graph contains negative-weight cycles.

(Repeating C *can create arbitrarily negative* $s{-}t$ *paths.)*

So we will always assume no negative cycles exist.

Subpath Optimality



Lemma (Cut and Paste Argument):
Let $P = (s, \ldots, u, \ldots, t)$ be a shortest $s-t$ path. Then the subpaths

$$P_1 = (s, \ldots, u) \quad \text{and} \quad P_2 = (u, \ldots, t)$$

must also be, respectively, shortest $s-u$ and $u-t$ paths.

Pf: (by contradiction)

- Suppose the subpath $P_1 = (s, \ldots, u)$ is not the shortest $s-u$ path; i.e., there is another path P'_1 from s to u that is shorter than P_1.
- Then we can replace P_1 with P'_1, this creates $P' = P'_1\, P_2$, a $s-t$ path shorter than P.
- This contradicts the choice of P as a shortest $s-t$ path. Impossible!
- Same proof works for the subpath from u to t.

Concept of Relaxation

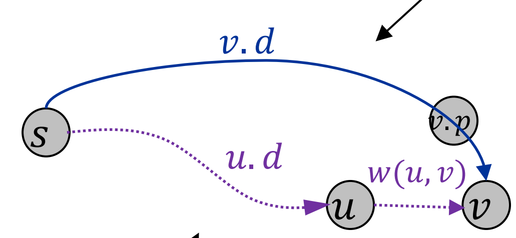
Let $v.d$ be shortest distance found so far from starting node s to node v, and
$v.p$ be the last node in the current shortest path from s to node v.

Relaxing edge (u, v) means checking whether
taking shortest path to u and then edge (u, v)
gives an even shorter path to v
improving known shortest path to v.

If this occurs, we *update* $v.d = u.d + w(u, v)$ and $v.p = u$

Old candidate shortest path

Relax(u, v)
If $u.d + w(u, v) < v.d$ *Then*
$v.d = u.d + w(u, v)$
$v.p = u$



New candidate shortest path

Bellman-Ford Algorithm

- Initially, we set $v.d = \infty$ for all nodes,
  except the starting node s for which $s.d = 0$
- Relax all edges once, in no particular order.
  After finishing, $v.d < \infty$ for all neighbors of s, or equivalently for all nodes that are connected with s through a path with length 1 edge.
- Relax all edges a 2nd time (in no particular order).
  After finishing, $v.d < \infty$ for all nodes that can be reached from s through a path with length 1 or 2. A relaxation, may only decrease distances so $v.d$ is the shortest distance for paths with maximum length 2.
- In general, after relaxing all edges for the i-th time, $v.d$ is the shortest distance for paths with maximum length i edges.
- Assuming no negative cycles, what is the max number of edges in a path?
- A path may have at most $V - 1$ edges.
- Thus, after relaxing all edges $V - 1$ times, $v.d$ is the **actual** shortest distance between v and s.

Bellman-Ford- Basic Implementation

```
Shortest-Path(G,s):
for each node v ∈ V do
        v.d ← ∞
s.d←0
for i ← 1 to V − 1
    for each edge (u,v) ∈ E
        if u.d + w(u,v) < v.d then      ⎫
            v.d ← u.d + w(u,v)          ⎬ Relax(u,v)
            v.p ← u                     ⎭
```

Analysis. $\Theta(VE)$ time, $\Theta(V)$ space.

Bellman-Ford as Dynamic Programming

Def. $v.d[i]$ = length of shortest path from s to v using up to i edges.

Recurrence:

- Suppose (u, v) is the last edge of the shortest path from s to v. By the cut and paste argument, the subpath from s to u must also be shortest, using at most $i-1$ edges, followed by (u, v).

$$v.d[i] = \min_{u,(u,v)\in E} \{u.d[i-1] + w(u, v)\}$$
$$v.d[0] = \infty$$

- Final solution: $v.d[n-1]$ = length of the actual shortest path from s to v, since no shortest path can have n edges or more.

- Bellman-Ford uses a single $v.d$ instead of $v.d[i]$
  - After the i-th iteration, $v.d \le v.d[i]$

Bellman-Ford: Efficient Implementation

```
Bellman-Ford(G, s):
for each node v ∈ V
    v.d ← ∞, v.p ← nil
s.d ← 0
for i ← 1 to V − 1
    for each node u ∈ V
        if u.d changed in previous iteration then
            for each v ∈ Adj[u]
                if u.d + w(u,v) < v.d then          ⎫
                    v.d ← u.d + w(u,v)              ⎬ Relax(u,v)
                    v.p ← u                         ⎭
    if no v.d changed in this iteration then terminate
```

Analysis.

- $O(VE)$ time in the worst case, but can be much faster in practice
- $O(V)$ space.

Remark:

- Can be run in parallel.
- Used on massive graphs (even if no negative edges).
- Can also detect whether there is a negative cycle.

Exercise Bellman-Ford for Negative Cycle Detection

How you can use Bellman-Ford to detect negative cycles?

Solution:

Assuming no negative cycles, the max number of edges in a path is $V - 1$.

What happens if there are negative cycles?

Some nodes distances will continue decreasing after relaxing all edges for V times.

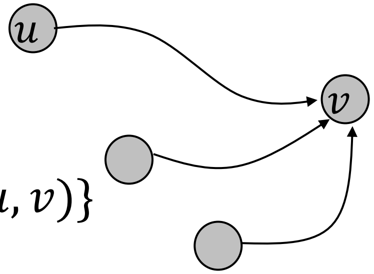
Apply Bellman-Ford and add another round of relaxations:

For each edge (u, v) // check for negative cycles

    If $d[u] + w(u, v) < d[v]$ then return "Negative Cycle"

Shortest path in a DAG

- Input is a DAG, a ***Directed Acyclic Graph***
- $\delta(s, v)$ will store shortest path distance from s to v .
- By subpath optimality, we have



$$\delta(s,v) = \min_{u,(u,v)\in E}\{\delta(s,u) + w(u,v)\}$$

- Unlike in Bellman-Ford, each edge will only be relaxed once.
- We need to ensure that when v is processed,
  ALL u with $(u, v) \in E$ have already been processed,
  so $\delta(s, u)$ holds the correct value when v is processed,
- We can do that by processing v *(and thus the* $\delta(s, v)$) in the topological order of the nodes.

Shortest path in a DAG: algorithm

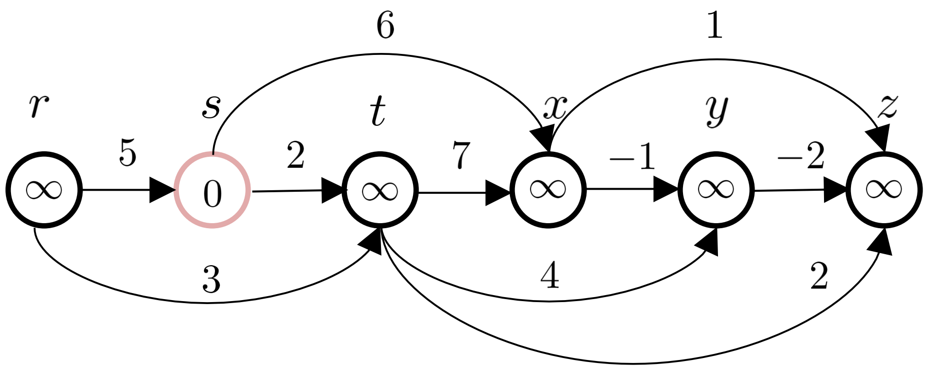
<u>**DAG-Shortest-Path**(G, s)</u>
topologically sort the vertices of G
for each vertex $v \in V$
    $v.d \leftarrow \infty$
    $v.p \leftarrow nil$
$s.d \leftarrow 0$
for each vertex u **in topological order**
    for each vertex $v \in Adj[u]$
        if $v.d > u.d + w(u, v)$ **then**
            $v.d \leftarrow u.d + w(u, v)$
            $v.p \leftarrow u$

Relax $(\boldsymbol{u}, \boldsymbol{v})$

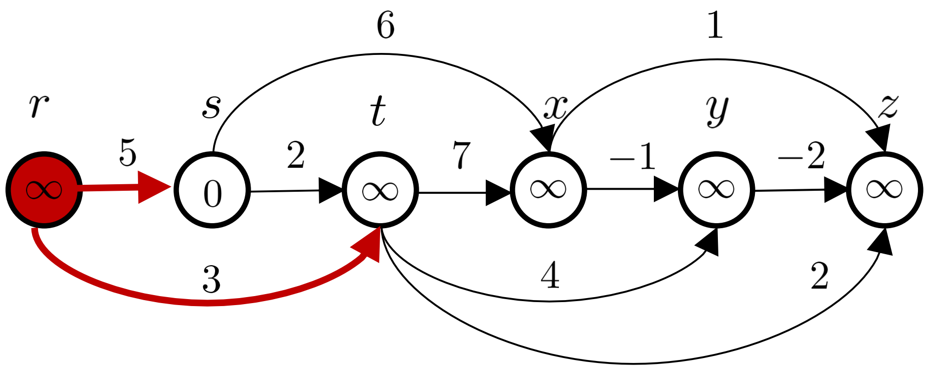
Running time: $\Theta(V + E)$

Note:

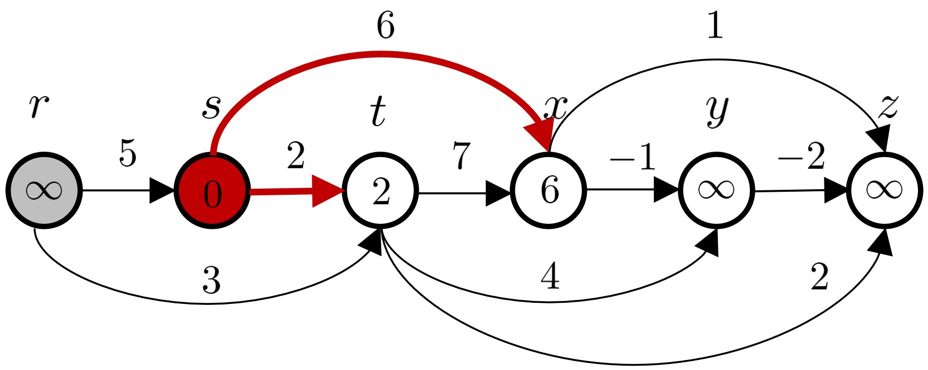
- Can find the actual shortest path by tracing the parent pointers.
- If we just want to find the shortest path from s to t, can stop the algorithm when $u = t$. But this does not reduce the running time asymptotically.



v	$v.d$	$v.p$
r	∞	nil
s	0	nil
t	∞	nil
x	∞	nil
y	∞	nil
z	∞	nil



v	$v.d$	$v.p$
r	∞	nil
s	0	nil
t	∞	nil
x	∞	nil
y	∞	nil
z	∞	nil

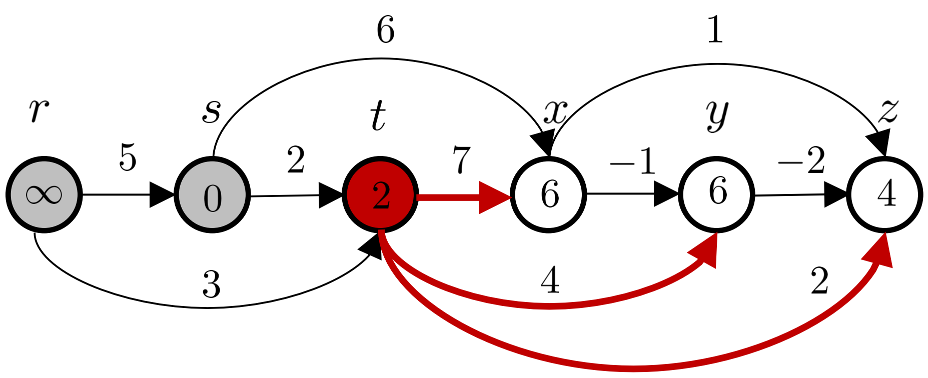


OLD

v	$v.d$	$v.p$
r	∞	nil
s	0	nil
t	∞	nil
x	∞	nil
y	∞	nil
z	∞	nil

NEW

v	$v.d$	$v.p$
r	∞	nil
s	0	nil
t	2	s
x	6	s
y	∞	nil
z	∞	nil

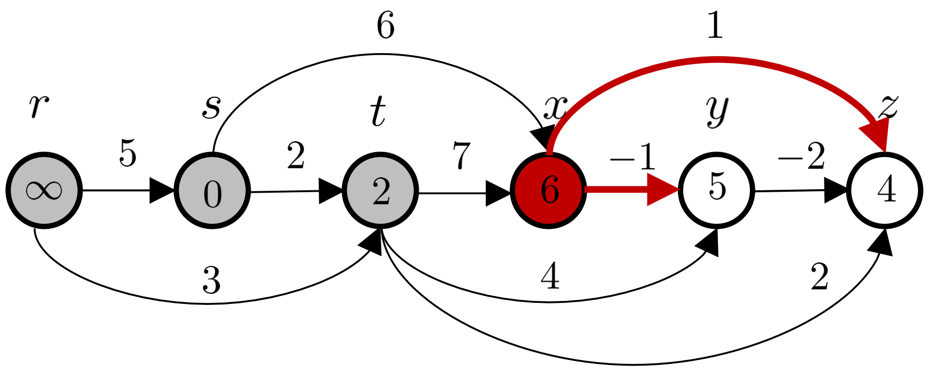


OLD

v	$v.d$	$v.p$
r	∞	nil
s	0	nil
t	2	s
x	6	s
y	∞	nil
z	∞	nil

NEW

v	$v.d$	$v.p$
r	∞	nil
s	0	nil
t	2	s
x	6	s
y	6	t
z	4	t

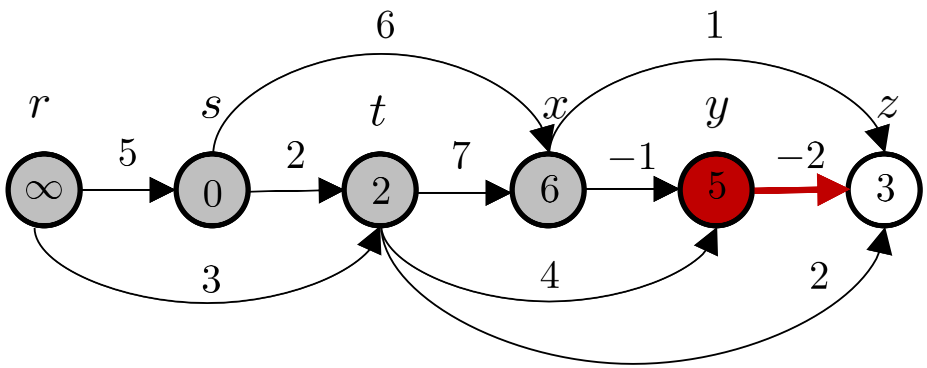


OLD

v	$v.d$	$v.p$
r	∞	nil
s	0	nil
t	2	s
x	6	s
y	6	t
z	4	t

NEW

v	$v.d$	$v.p$
r	∞	nil
s	0	nil
t	2	s
x	6	s
y	5	x
z	4	t

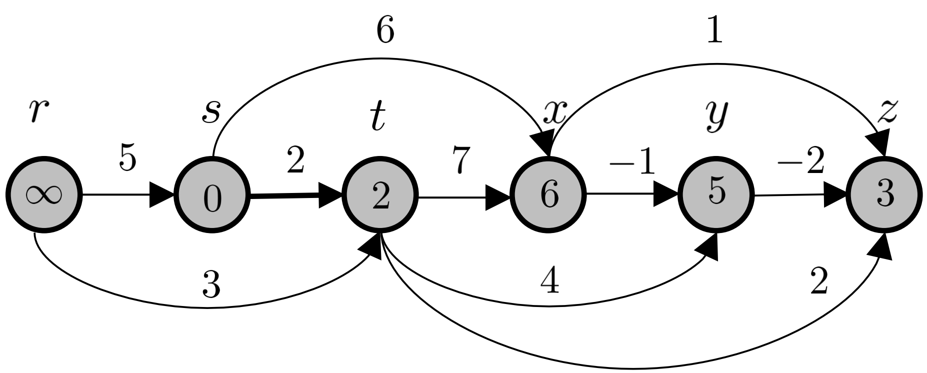


OLD

v	$v.d$	$v.p$
r	∞	nil
s	0	nil
t	2	s
x	6	s
y	5	x
z	4	t

NEW

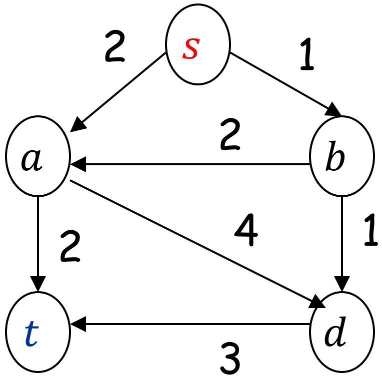
v	$v.d$	$v.p$
r	∞	nil
s	0	nil
t	2	s
x	6	s
y	5	x
z	3	y



v	$v.d$	$v.p$
r	∞	nil
s	0	nil
t	2	s
x	6	s
y	5	x
z	3	y

Exercise on Longest Path in DAG

Given a directed acyclic graph with real-valued edge weights and two vertices s, t, describe a dynamic programming algorithm for finding the **longest** weighted simple path from s to t.



Longest part:
s, b, a, d, t
Total weight=
1+2+4+3=10

Let $ld[v]$ be the weight of the longest path from s to v:

$$ld[v] = 0, \text{ if } s = v$$

$$ld[v] = \max\{w(u,v) + ld[u] : (u,v) \in E\}, \text{ otherwise}$$

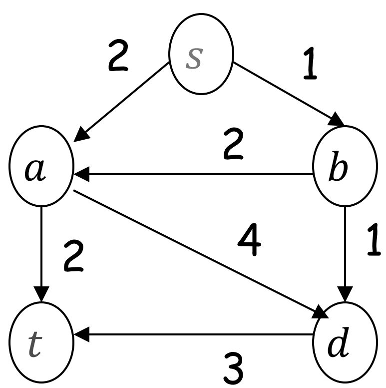
How do we make sure that when we reach v, we have computed the longest distance for every u such that there is an edge (u,v)?

Answer: We use *topological sort* starting from s.

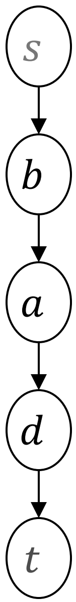
Algorithm on Longest Path in DAG

DP-LD(G, s, t)
Topologically sort the vertices of G, starting from s
For each vertex v, set $ld[v] := -\infty$
$ld[s] := 0$
For each vertex u in the topological order
    For each vertex v in adjacency list of u
        if $ld[u] + w(u, v) > ld[v]$ then
            $ld[v] := ld[u] + w(u, v)$

Running time $\Theta(V + E)$



topological sort



$ld[s] = 0$

$ld[b] = 1$

$ld[a] = 3$

$ld[d] = 7$

$ld[f] = 10$

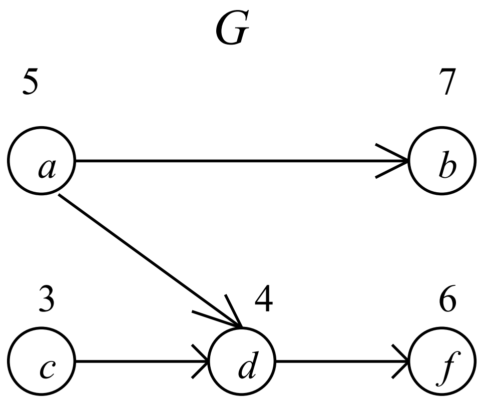
Exercise on Critical Paths

Let $G = (V, E)$ be a DAG, where vertices correspond to jobs and edges are sequence constraints: (u, v) means that job u should be performed before v (in other words, u must finish before v starts). Each *vertex* is associated with a positive weight that indicates the time to complete the corresponding job.

- Find the minimum time to perform all the jobs.

For instance, in the following graph, to finish d, we must first complete a and c; total time 9 (4 for d and 5 for a; c can be performed in parallel with a).

Minimum time is the time required to finish f, i.e., 15.

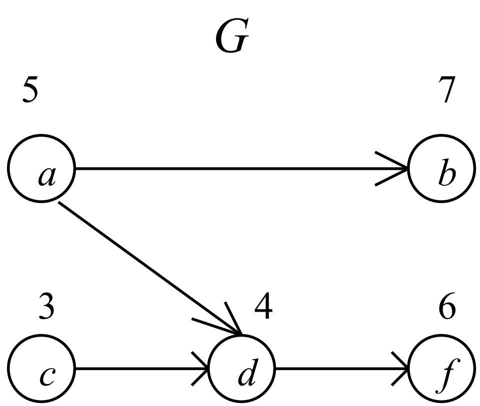


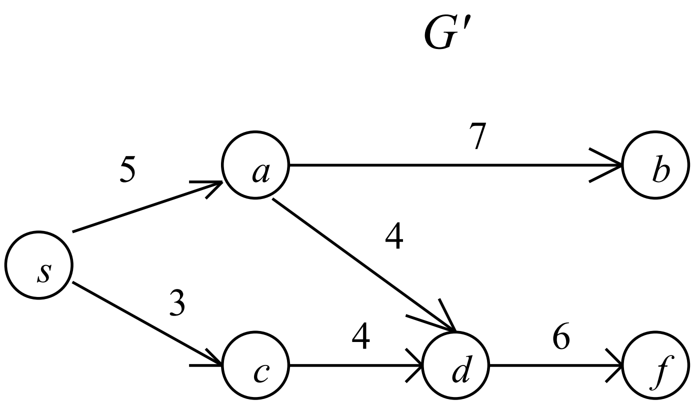
Solution by Longest Path

I generate a new graph $G' = (V', E')$ as follows.

- I add a new vertex s so that $V' = V \cup \{s\}$.
- All edges in E are included in E'.
- I add an edge from s to each vertex that has in-degree 0; (only s vertex has in-degree 0 in E').
- Then, I assign the weight of each edge (u, v) to be the weight of v. Thus, $V' = V + 1$ and $E' \leq E + V$ (since no more than V vertices have in-degree 0).

The earliest time that I can finish all jobs corresponds to the longest path in G'.





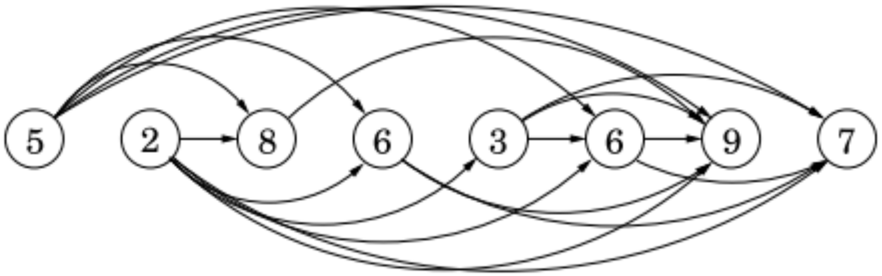
Exercise on Longest Increasing Subsequence

Input: a sequence of numbers: $a_1, a_2, \dots, a_n$

- Example: 5, 2, 8, 6, 3, 6, 9, 7
- Increasing sequence : 5, 2, 8, 6, 3, 6, 9, 7
- Longest increasing sequence : 5, 2, 8, 6, 3, 6, 9, 7 or 5, 2, 8, 6, 3, 6, 9, 7

Convert to a directed graph $G = (V, E)$

- $V = \{a_1, a_2, \dots, a_n\}$
- $E = \{(a_i, a_j) : i < j \text{ and } a_i < a_j\}$

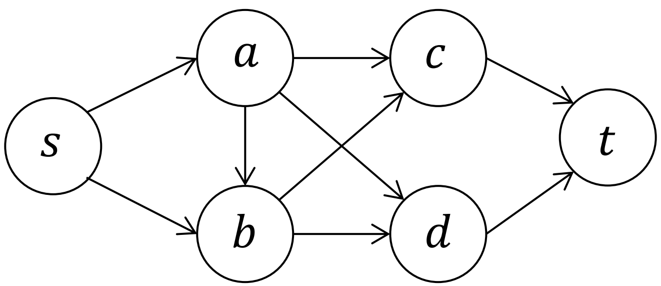


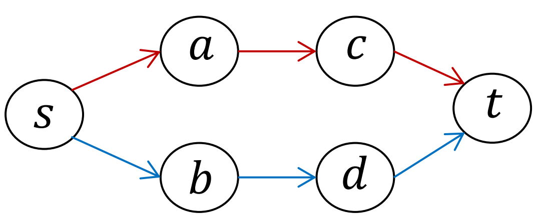
Equivalent graph problem: find longest path in G starting from every node with in-degree 0.

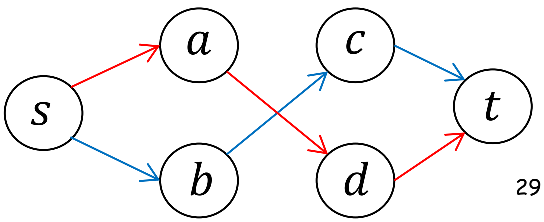
Exercise on Count of Distinct Paths

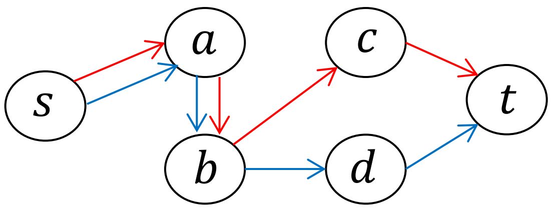
Let s, t be two vertices in a connected, directed, acyclic graph (DAG) G, where s is the first and t the last vertex in the topological order. Design an algorithm that counts the number of different paths from s to t (no need to find the paths, just their count). Two paths are different if they differ in at least one edge)

How many different paths exist from s to t in following graph?









Algorithm for Count of Distinct Paths

Let $d[u]$ be number of distinct paths from source s to u. Initially set $d[s] = 1$, and $d[u] = 0$ for $u \neq s$.

Do a topological sort on G

For each u in topological order

$$d[v] = d[v] + d[u] \text{ for every } v \in \mathrm{Adj}(u).$$