Agenda

- The shortest-path problem
- Dijkstra's algorithm
- Bellman-ford algorithm
- Algorithm for DAGs

A real world problem

- Lars plans to go from Aalborg to Aarhus and he wants to save fuel.
  - Assuming fuel consumption is proportional to travel distance.
- One possible solution is
  - Model road network as a weighted graph where weights represent lengths of roads.
  - Enumerate all paths from Aalborg to Aarhus.
  - Add up the lengths of roads in each path, and select the path with the shortest sum of lengths.
- This solution is very inefficient because it examines a lot of paths that are not worth considering.
  - For example, a path via Skagen is a poor choice.
- Today, we will see a few efficient algorithms that are able to find shortest paths.

Path

- Path representations
  - A sequence of vertices $<v_0, v_1, \ldots, v_k>$ such that vertex v_{i+1} is adjacent to vertex v_i for i = 0, 1 ... k – 1.
  - A sequence of edges $<(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)>$.
  - A sequence of edges $<e_0, e_1, \ldots, e_{k-1}>$, where,
    - $e_0 = (v_0, v_1)$,
    - $e_1 = (v_1, v_2)$, …, and
    - $e_{k-1} = (v_{k-1}, v_k)$.
- If there is a path p from vertex u to vertex v, we say that v is reachable from u.

Shortest Path

- Weighted, directed graph G=(V, E) with a weight function w: E $\rightarrow \mathbb{R}$.
- A path on G: p = $<(v_0, v_1), (v_1, v_2), \ldots (v_{k-1}, v_k)>$
  $w(v_0, v_1)$ $w(v_1, v_2)$ $w(v_{k-1}, v_k)$
- Weight of a path is: w(p)=$\sum_{i=0}^{k-1} w(v_i, v_{i+1})$
- Given two vertices u and v in V,
  - More than one path exist to go from u to v, e.g., $p_1, p_2, \ldots, p_n$.
  - Each path has a weight: $w(p_1), w(p_2), \ldots, w(p_n)$.
- The shortest-path weight, denoted as δ (u, v), from u to v:
  - $\min(w(p_1), w(p_2), \ldots, w(p_n))$
  - Defined as $\delta(u, v) = \begin{cases} \min\{w(p) : u \overset{p}{\rightsquigarrow} v\} & \text{if there is a path from } u \text{ to } v, \\ \infty & \text{otherwise.} \end{cases}$
- The shortest path between u and v is a path p with the short-path weight w(p) = δ(u, v).

Shortest Path Problems

- **Single-source:** Find a shortest path from a given source (vertex *s*) to each of the vertices that are reachable from s.
  - The topic of this lecture.
- **Single-pair:** Given two vertices, find a shortest path between them.
  - Solution to single-source problem also solves this problem efficiently.
- **All-pairs:** Find shortest-paths for every pair of vertices.
  - Running a single-source algorithm once from each vertex.
  - More efficient method: chapter 25, not covered by this lecture (but covered in Advanced Algorithms (AALG)).
- Shortest-paths for **un-weighted** graphs.
  - BFS, covered in Lecture 10.

Optimal substructure

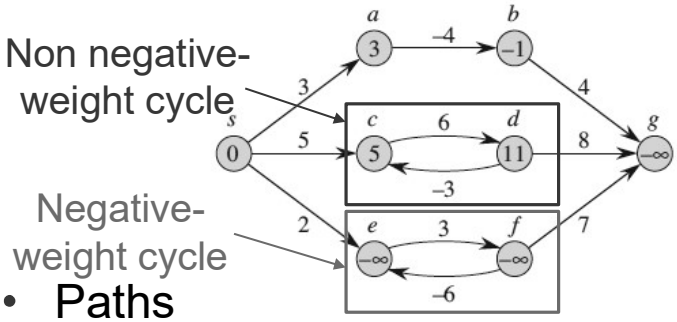
- Subpaths of shortest paths are shortest paths.
- Given a shortest path from v_0 to v_k
  - $p = \langle (v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k) \rangle$ or
  - $p = \langle v_0, v_1, v_2, \ldots, v_k \rangle$.
- For any i and j such that $0 \leq i \leq j \leq k$, a subpath of p from v_i to v_j:
  - $p_{ij} = \langle v_i, v_{i+1}, \ldots, v_j \rangle$
- Then, p_{ij} must be a shortest path from v_i to v_j.
- Proof by contradiction

Proof If we decompose path p into $v_0 \overset{p_{0i}}{\rightsquigarrow} v_i \overset{p_{ij}}{\rightsquigarrow} v_j \overset{p_{jk}}{\rightsquigarrow} v_k$, then we have that $w(p) = w(p_{0i}) + w(p_{ij}) + w(p_{jk})$. Now, assume that there is a path p'_{ij} from v_i to v_j with weight $w(p'_{ij}) < w(p_{ij})$. Then, $v_0 \overset{p_{0i}}{\rightsquigarrow} v_i \overset{p'_{ij}}{\rightsquigarrow} v_j \overset{p_{jk}}{\rightsquigarrow} v_k$ is a path from v_0 to v_k whose weight $w(p_{0i}) + w(p'_{ij}) + w(p_{jk})$ is less than $w(p)$, which contradicts the assumption that p is a shortest path from v_0 to v_k. ■

Negative weights on Shortest Path?

- If weights are non-negative, shortest paths are well-defined.
- Negative weights effect shortest-path weights?

Negative-weight cycle effects.



- Paths
  - Path from s to a: <s, a>, δ(s, a) = w(s, a) = 3
  - Path from s to b: <s, a, b>, δ(s, b) = w(s, a) + w(a, b) = 3 + (-4) = -1
  - Paths from s to c: <s, c>, <s, c, d, c>, <s, c, d, c, d, c>, …

δ(s, c) = w(s, c) = 5, because w(<c, d, c>) = w(c, d) + w(d, c) = 6 + (-3) = 3

  - Paths from s to e: <s, e>, <s, e, f, e>, <s, e, f, e, f, e>, ….

δ(s, e) = -∞, because w(<e, f, e>) = w(e, f) + w(f, e) = 3 + (-6) = -3
No shortest path from s to e.

Cycles on Shortest Path?

- A path $<v_0, v_1, \ldots, v_i, \ldots, v_j, \ldots, v_k>$ forms a cycle, if $v_i = v_j$.
- Can shortest paths have cycles?
- Shortest paths have **NO** cycles.
  - Negative-weight cycles
    - No. Otherwise, shortest paths are not well-defined anymore.
  - Positive-weight cycles
    - No. Otherwise, we could get shorter paths by removing the cycles.
  - 0-weight cycles
    - No. We can just repeatedly remove the 0-weight cycles to form another shortest path, until the path becomes cycle-free.
- Any acyclic path in a graph G=(V, E) contains at most |V| distinct vertices and at most |V|-1 edges.
  - This property is used in Bellman-Ford algorithm.

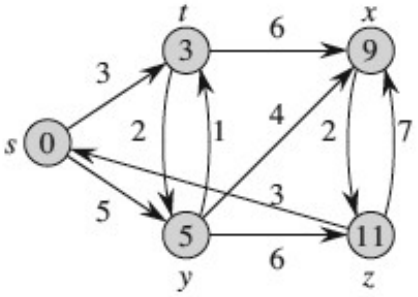
Shortest Paths Tree

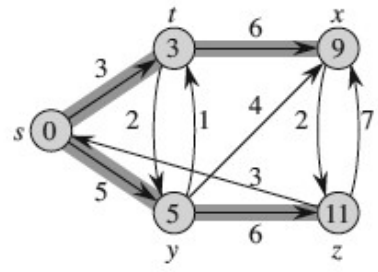
- The results of shortest path algorithms:
- A *shortest paths tree.*
  - The shortest paths tree is a tree with the source vertex s as the root, and it records a shortest path from the source vertex *s* to each vertex *v* that is reachable from s.

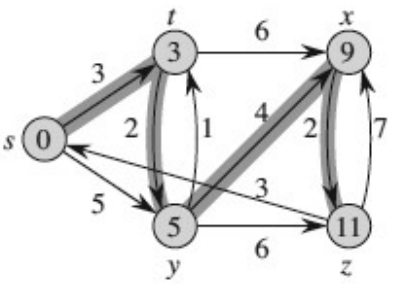
- Each vertex v
  - v.**parent()** records the predecessor of v in its shortest path.
  - v.**d()** records a shortest-path weight from s to v.

Uniqueness of shortest paths

- Shortest paths are not necessarily unique.
- Shortest paths trees are not necessarily unique.





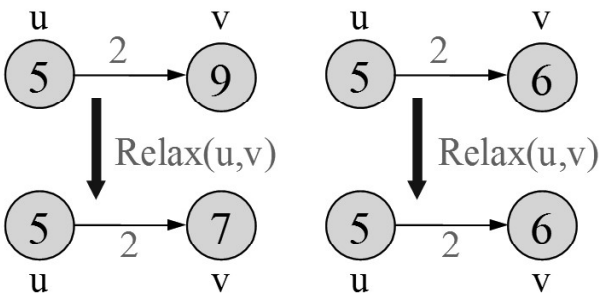


Relaxation Technique

- For each vertex *v* in the graph, we maintain *v.***d()**:
  - Estimate the weight of a shortest path from *s* to *v*,
  - Initialize *v.***d()** to ∞ in the beginning,
  - Update, i.e., decrease, the value of *v.***d()** during the search.
- Intuition behind relaxing an edge (*u, v*):
  - Check whether a new path from *s*, via *u*, to *v* can improve the existing shortest path from *s* to *v.*
  - w(*s, u*) + w(*u, v*) vs. w(*s, v*)

u.**d()** + w(u, v) < v.**d()**

?



If we can find a smaller shortest path weight to go from s to v?

```
Relax (u,v,G)
if v.d() > u.d()+G.w(u,v) then
   v.setd(u.d()+G.w(u,v))
   v.setparent(u)
```


Mini quiz on Moodle

3. (from 2003 re-exam)
Consider the following two statements about weighted graphs.

1. A shortest path between two vertices in a graph may include edges that do not belong to any minimum spanning tree of the graph.
2. An edge with the highest cost is never included in a minimum spanning tree.

Are these statements true or false?

- ☐ a) both statements true
- ☐ b) statement 1 true, statement 2 false
- ☐ c) statement 1 false, statement 2 true
- ☐ d) both statements false

b

Agenda

- The shortest-path problem
- Dijkstra's algorithm
- Bellman-ford algorithm
- Algorithm for DAGs

Dijkstra's algorithm

- It works for graphs with **non-negative** edge weights.
- Input:
  - Directed, weighted graph G = (V, E),
  - A weight function w: E $\rightarrow \mathbb{R}$,
  - A source vertex *s*.
- Output (**Single-source**):
  - A set of vertices *S,* |*S*| = |*V*|.
    - Each vertex $u \in S$ has a value for *u*.**d()** and for *u*.**parent()**.
    - If *u* is not reachable from *s*, *u*.**d()** = ∞.

Shortest-path weight from *s* to *u*

Shortest path from *s* to *u*

- Intuition:
  - Maintain a set *S* of visited vertices, and each time
  - 1) select a vertex *u* that is the "closest" from *s*, and add it to *S.*

Closest = Least shortest-path weights A priority queue prioritized on *u.d()*

  - 2) relax all edges from *u*, i.e., check whether going through u can improve the shortest path and the shortest-path weight of u's neighbors.

Algorithm

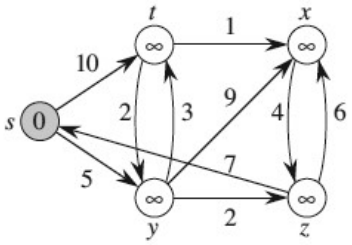
```
Dijkstra(G,s)
01 for each vertex u ∈ G.V()
02     u.setd(∞)
03     u.setparent(NIL)
04 s.setd(0)
05 S ←∅              // Set S is used to explain the algorithm
06 Q.init(G.V())    // Q is a priority queue ADT
07 while not Q.isEmpty()
08     u ← Q.extractMin()
09     S ←S ∪{u}
10     for each v ∈ u.adjacent() do
11         Relax(u, v, G)
12         Q.modifyKey(v)
```

Initialize all vertices

Relaxing edges for each u's adjacent vertices.

```
Relax (u,v,G)
if v.d() > u.d()+G.w(u,v) then
   v.setd(u.d()+G.w(u,v))
   v.setparent(u)
```

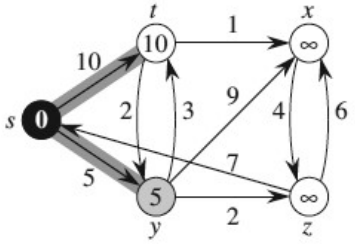

Running example



s	t	y	x	z
0/-	∞	∞	∞	∞

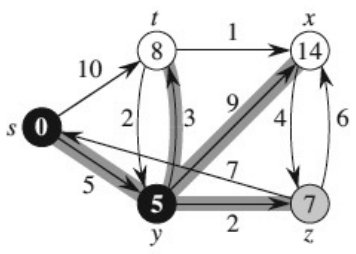
S=∅

Line 8: s is extracted with parent NIL. S={s}
s.**d()**=0
Relax (s, t), (s, y) { s.d() + w(s,t) < t.d()
s.d() + w(s,y) < y.d()



y	t	x	z
5/s	10/s	∞	∞

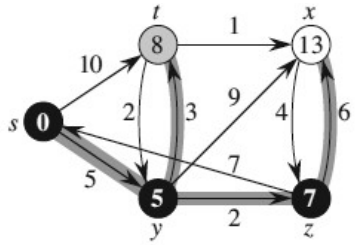
Line 8: y is extracted with parent s. S={s, y}
y.**d()**=5
Relax (y, z), (y, x), (y, t) { y.d() + w(y,z) < z.d()
y.d() + w(y,x) < x.d()
y.d() + w(y,t) < t.d()



z	t	x
7/y	8/y	14/y

Line 8: z is extracted with parent y. S={s, y, z}
z.**d()**=7
Relax (z, x), (z, s) { z.d() + w(z,x) < x.d()
z.d() + w(z,s) > s.d()

Running example (2)

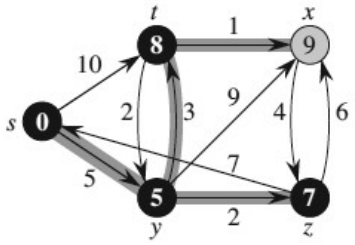


t	x
8/y	13/z

Line 8: t is extracted with parent y. S={s, y, z, t}

t.**d()**=8

Relax (t, x), (t, y) { t.d() + w(t,x) < x.d()
t.d() + w(t,y) > y.d()

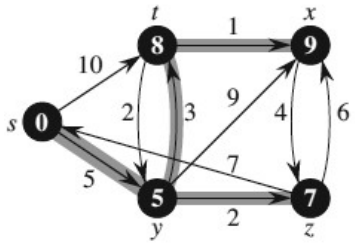


x
9/t

Line 8: x is extracted with parent t. S={s, y, z, t, x}

x.d()=9

Relax (x, z): x.d() + w(x,z) > z.d()

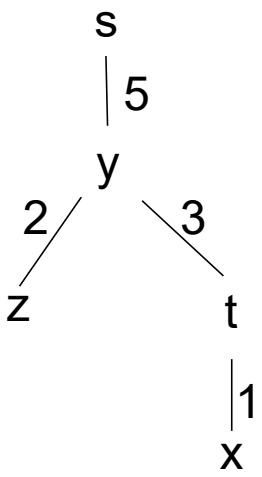


Q is empty and S has all vertices in V. Stop.

Think about what if a vertex is not reachable from s?

Shortest paths tree

- Line 8: s is extracted with parent NIL.
- Line 8: y is extracted with parent s.
- Line 8: z is extracted with parent y.
- Line 8: t is extracted with parent y.
- Line 8: x is extracted with parent t.



Mini quiz

- Try Dijkstra's alg on the following graph.

Run-time

Complexity depends on how to implement the min-priority queue.
- Binary min-heap (c.f. Lecture 6): O(|E|lg|V|)

```
Dijkstra(G,s)
01 for each vertex u ∈ G.V()
02     u.setd(∞)
03     u.setparent(NIL)
04 s.setd(0)
05 S ←∅            // Set S is used to explain the algorithm
06 Q.init(G.V())   // Q is a priority queue ADT
07 while not Q.isEmpty()
08     u ← Q.extractMin()
09     S ←S ∪{u}
10     for each v ∈ u.adjacent() do
11         Relax(u, v, G)
12         Q.modifyKey(v)
```

Initialize all vertices: Θ(|V|)

Initialize Q: O(|V|)

|V| times of Q.extractMin()

|E| times of edge relax

|E| times of Q.modifyKey()

Run-time (2)

- Initialization of vertices: Θ(|V|)
- Initialization of priority queue with |V| elements: O(|V|)
- Q.extract-Min executed |*V*| times, thus |*V*|* $T_{\text{Extract-Min}}$
- Q.modify-Key executed |*E*| times, thus |*E*|* $T_{\text{modify-Key}}$
- Relax edge |E| times, thus Θ(|E|)
- T = |*V*|* $T_{\text{extract-Min}}$ + |*E*|* $T_{\text{modify-Key}}$+O(|V|) + Θ(|E|)

- *T* depends on different priority queue implementations

Priority Queue Implementation	extract-Min	modify-Key	Total
Array	O(|V|)	O(1)	$O(\|V\|^2)$
Binary min heap	O(lg|V|)	O(lg|V|)	O((|E|+|V|)lg|V|) =O(|E|lg|V|)

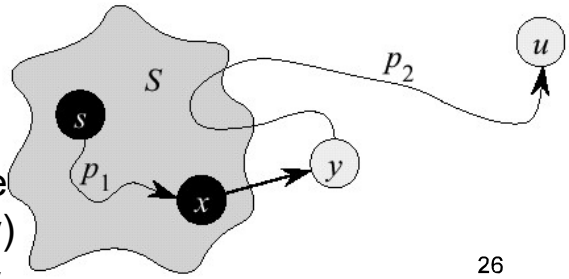
Correctness-1

- We need to prove that, in line 9, whenever vertex u is removed from the priority queue by Q.extractMin() and is added to S, we have u.**d()** = δ(s, u).
  - This means that every time when we put a vertex u into S, its distance u.**d()** is correctly set to the shortest-path weight δ(s, u).
- Assume:
  - Vertex u is the next vertex in Q to add to S.
  - When u is added to S, u is the first vertex in Q, for which u.**d()** ≠ δ(s, u).
- The assumption also means:
  - Vertex u has the least value u.**d()** among all vertices in Q.
  - At the time, u is added to S, each vertex x in S has x.**d()** = δ(s, x).
- We prove:
  - If u.**d()** > δ(s, u) happens, it leads to a contradiction, meaning that this case never happens. Thus, Dijkstra's algorithm is correct.

Correctness-2

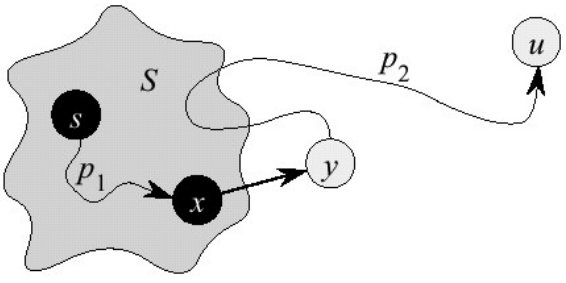
- Let's consider a shortest path p from s to u: $s \overset{p_1}{\rightsquigarrow} x \rightarrow y \overset{p_2}{\rightsquigarrow} u$
  - Path p_1 $s \overset{p_1}{\rightsquigarrow} x$: all vertices on p_1 are put to S.
  - An edge (x, y): y is the first vertex on path p that $y \in Q$ and $y \notin S$.
  - Path p_2 $y \overset{p_2}{\rightsquigarrow} u$: all vertices on p_2 are still in Q.
- When u is added to S, we claim:
  - x.**d()** = δ(s, x), i.e., x.**d()** is set correctly to the shortest-path weight from s to x.
    - Because for each vertex x in S, we have x.**d()** = δ(s, x) -- assumption.
  - y.**d()** = δ(s, y), i.e., y.**d()** is also set correctly: δ(s, y) = δ(s, x) + w(x, y)
    - Convergence property (CLRS page 672, Lemma 24.14).
    - When the algorithm added *x* into *S*, it also relaxed the edge (*x*, *y*), and set y.**d()** to δ(s, y) correctly.

If before relaxing: y.**d()** > x.**d()** + w(x, y)
After relaxing: y.**d()** = x.**d()** + w(x, y)
If before relax: y.**d()** ≤ x.**d()** + w(x, y)
After relaxing: y.**d()** ≤ x.**d()** + w(x, y), no change
y.**d()** ≤ x.**d()** + w(x, y) = δ(s, x) + w(x, y) = δ(s, y)
y.**d()** ≥ δ(s, y) Lemma 24.11 upper-bound property.



Correctness-3

- u.**d()** > δ(s, u) (initial assumption)
- = δ(*s, y*) + δ(y*, u*) (optimal substructure)
- = *y.***d()** + δ(y*, u*) (correctness of *y.***d()**)
- ≥ *y.***d()** (no negative weights)
- u.**d()** > y.**d()** is valid?
  - But *u.***d()** > *y.***d()** means that the algorithm would have chosen *y* (from the priority queue) to process next, not *u*.
  - This contradicts to the fact that u is the next vertex to be removed from the priority queue Q by the Q.extractMin() operation.



Correctness-4

- Thus, the case that **u.d()** > δ(s, u) and u is added into S won't happen.
- And u.**d()** = δ(s,u) at the time when u is added into *S*, meaning that Dijkstra's algorithm is correct.
- See CLRS pages 659 to 661 for a more formal proof using loop invariant.

Agenda

- The shortest-path problem
- Dijkstra's algorithm
- Bellman-ford algorithm
- Algorithm for DAGs

Bellman-Ford Algorithm - 1

- Dijkstra's doesn't work when there are negative edges.
  - Intuition – we cannot guarantee that the length of a path increases when more edges are included into the path.
- Bellman-Ford algorithm can handle a graph where edges have negative weights (but no negative-weight cycles).
- Input:
  - Directed, weighted graph G = (V, E),
  - A weight function w: E $\rightarrow \mathbb{R}$,
  - A source vertex *s*.
- Output (**Single-source**):
  - Boolean value: *false* = detects negative-weight cycles; or *true* = returns the shortest path-tree.
  - If Boolean value = *true*, a set of vertices *S,* |*S*| = |*V*|.
    - Each vertex $u \in S$ has a value for *u*.**d()** and for *u*.**parent()**.
    - If *u* is not reachable from *s*, *u*.**d()** = ∞.

Bellman-Ford Algorithm - 2

- Dijkstra's
  - All vertices in Q
  - *s*=Q.extractMin(), add to S
  - Visit the neighbors of *s*, e.g., *u, v*
  - Relax (*s, u,* G), Q ← u
    Relax (*s, v,* G), Q ← v
  - Q.extractMin(), add to S
  - Relax…
  - ……
  - Relax(.) follows an order

```
Relax (u,v,G)
if v.d() > u.d()+G.w(u,v) then
   v.setd(u.d()+G.w(u,v))
   v.setparent(u)
```

- Bellman-Ford
  - Repeat |V|-1 iterations
  - Each iteration,
    Relax(.) for each edge in E.
  - Since edges follow no order in E, it is possible:

edge (*s, v*): *v*.**d()** > *s*.**d()** + w(*s, v*)?
∞ > 0 + w(*s, v*)

edge (*u, t*): *t*.**d()** > *u*.**d()** + w(*u, t*)?
∞ ⊁ ∞ + w(*u, t*)

  - Over iterations, keep relaxing edges, progressively decreasing an estimate on *t*.**d()**.
  - Relax(.) does not follow an order.

Algorithm

Bellman-Ford(G, s)

```
01 for each vertex u ∈ G.V()
02    u.setd(∞)
03    u.setparent(NIL)
04 s.setd(0)
05 for i ← 1 to |G.V()|-1 do
06     for each edge (u,v) ∈ G.E() do
07        Relax (u,v,G)
08 for each edge (u,v) ∈ G.E() do
09     if v.d() > u.d() + G.w(u,v) then
10        return false
11 return true
```

```
Relax (u,v,G)
if v.d() > u.d()+G.w(u,v) then
   v.setd(u.d()+G.w(u,v))
   v.setparent(u)
```

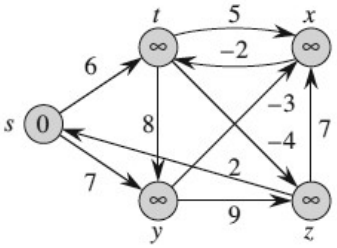
Initialize all vertices: Θ(|V|)

Keep relaxing edges: Θ(|V|*|E|)

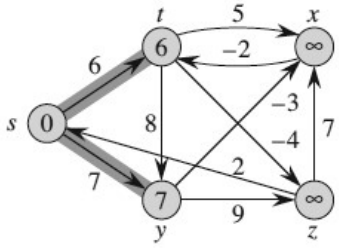
Check negative-weight cycles: O(|E|)
false: there is a negative-weight cycle

In total: Θ(|V|*|E|)

Running Example (1)

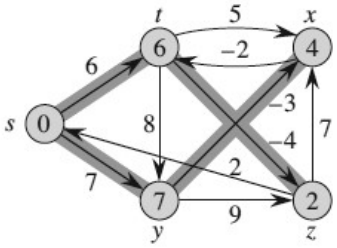


|V|=5, thus 5-1=4 iterations.
In each iteration, check all edges.
Assume the order of checking edges is
(t, x), (t, y), (t, z), (x, t), (y, x),
(y, z), (z, x), (z, s), (s, t), (s, y).



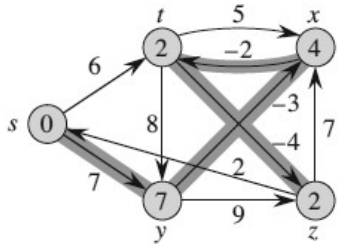
Iteration 1:
Relaxing (t, x), (t, y), (t, z), (x, t), (y, x),
(y, z), (z, x), (z, s), no changes.
Relaxing (s, t): t.d() = 6, t.parent()= s.
Relaxing (s, y): y.d() =7, y.parent()= s.

Running example (2)

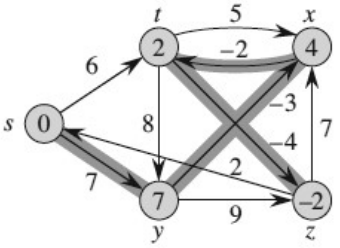


Relaxing order: (t, x), (t, y), (t, z), (x, t), (y, x), (y, z), (z, x), (z, s), (s, t), (s, y).

Iteration 2
Relaxing (t, x): x.d() = 11, x.parent()=t
Relaxing (t, z): z.d()=2, z.parent()=t
Relaxing (y, x): x.d()=4, x.parent()=y



Iteration 3
Relaxing (x, t): t.d()=2, t.parent()=x



Iteration 4
Relaxing (t, z): z.d()=-2, z.parent()=t

Correctness - Path Relaxation Property

- Path Relaxation Property
  - If path p=<$v_0, v_1, v_2, \ldots v_k$> is a shortest path from $s = v_0$ to v_k, and
  - If we relax the edges of p in the order (v_0, v_1), (v_1, v_2), …, (v_{k-1}, v_k),
  - Then, we have $v_k.d() = \delta(s, v_k)$.

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

- Prove by induction.
  - We show by induction that after the i-th edge of path p is relaxed, we have $v_i.d() = \delta(s, v_i)$.

Correctness - Path Relaxation Property

- For the base case, when i=0:
  - Before any edge of p have been relaxed.
  - From the initialization of the algorithm,
  - We know that $v_0.d() = s.d() = \delta(s, v_0)=0$.
- Relaxation never increases d values, no matter what edge relaxations will happen in the future.
  - $v_0.d = \delta(s, v_0) = 0$.
- So it holds for the base case.

```
Relax (u,v,G)
if v.d() > u.d()+G.w(u,v) then
   v.setd(u.d()+G.w(u,v))
   v.setparent(u)
```


Correctness - Path Relaxation Property

- Inductive to prove: $v_i.d() = \delta(s, v_i)$
  - Assume $v_{i-1}.d() = \delta(s, v_{i-1})$,
  - After relaxing the i-th edge (v_{i-1}, v_i), we prove $v_i.d() = \delta(s, v_i)$ still holds.
  - Then the path relaxation property is proved.
- According to the optimal substructure, since $p=<v_0, v_1, v_2, \ldots v_k>$ is a shortest path from v_0 to v_k, we have
  - $p_1=<v_0, v_1, v_2, \ldots, v_{i-1}, v_i>$ is a shortest path from v_0 to v_i ,
  - $p_2=<v_0, v_1, v_2, \ldots, v_{i-1}>$ is a shortest path from v_0 to v_{i-1},
  - $p_3=<v_{i-1}, v_i>$ is a shortest path from v_{i-1} to v_i.
  - And we have $w(p_1)=w(p_2)+w(p_3)$
- By relaxing edge (v_{i-1}, v_i), we set $v_i.d() = v_{i-1}.d() + w(v_{i-1}, v_i)$
  - $v_i.d() = v_{i-1}.d() + w(v_{i-1}, v_i) = \delta(s, v_{i-1}) + w(v_{i-1}, v_i) = w(p_2) + w(v_{i-1}, v_i)$
  - $v_i.d() = w(p_2) + w(v_{i-1}, v_i) = w(p_2) + w(p_3) = w(p_1)$.
  - Thus, $v_i.d() = \delta(s, v_i)$, i.e., the shortest path distance from v_0 to v_i.
- This completes the proof.

Agenda

- The shortest-path problem
- Dijkstra's algorithm
- Bellman-ford algorithm
- Algorithm for DAGs

Shortest Path in DAGs

- Topologically sort the DAG.
- Relax the edges of vertices according to the topologically sorted order of vertices.

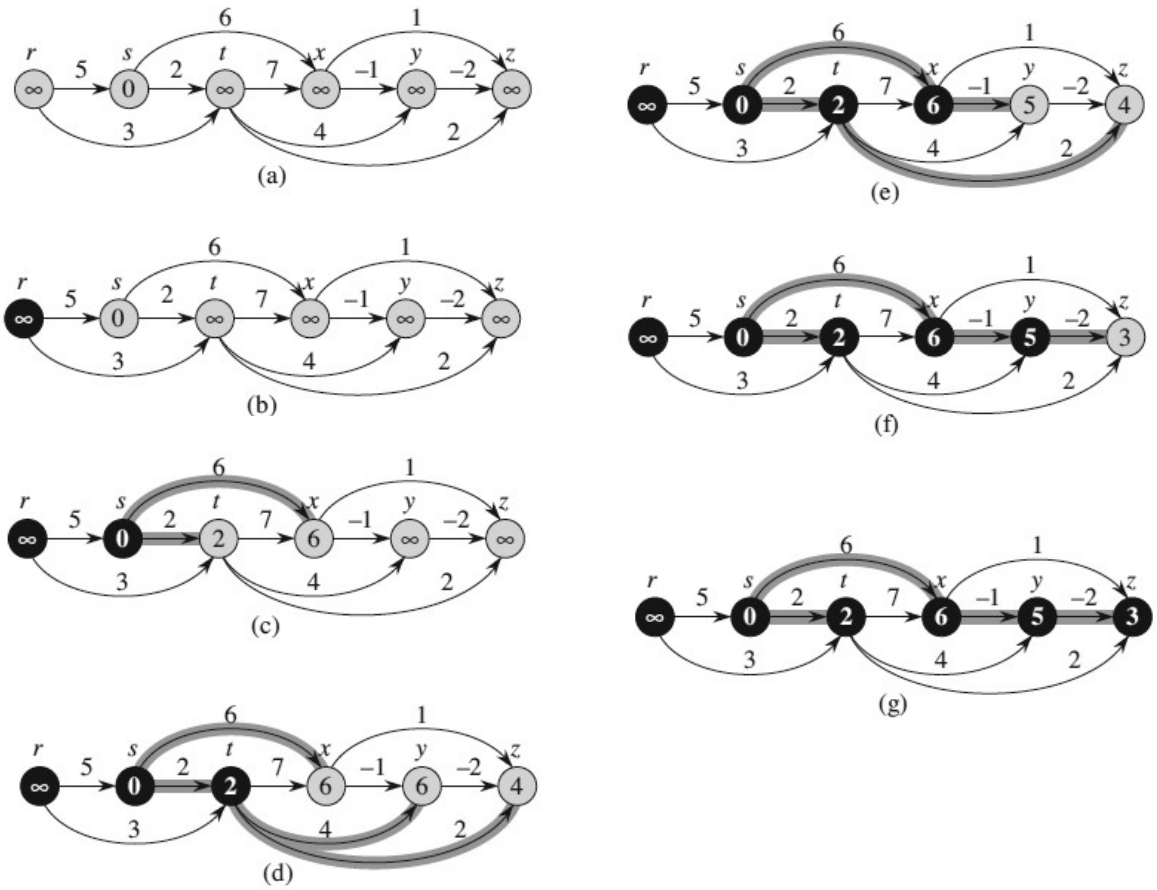
```
DAG-Shortest-Paths(G,s)
01 for each vertex u ∈ G.V()
02     u.setd(∞)
03     u.setparent(NIL)
04 s.setd(0)
05 topologically sort G
06 for each vertex u, taken in topolog. order do
07     for each v ∈ u.adjacent() do
08         Relax(u, v, G)
```

Initialize all vertices: Θ(|V|)

Topological sort Θ(|V|+|E|)

Relax all edges Θ(|E|)
In total: Θ(|V|+|E|)

Running Example



Correctness

- Also based on path relaxation property.

- After topological sort, vertices are ordered from “left” to “right”, according to reversed finishing time:
  - Finishing time of a “left” vertex (e.g., v_i) > Finishing time of a “right” vertex (e.g., v_j)
  - It means, DFS starts from s, next it reaches v_i, then v_j, finishes v_j, and finally goes back to and finishes v_i.
- When find a shortest path from s:
  - A shortest path starting from s, first go to v_i, and then v_j
  - $\delta(s, v_i) < \delta(s, v_j)$
- When we relax edges, as long as we follow the topologically sorted order from “left” to “right”, we main the path relaxation property.

Mini quiz

- What about calling Bellman-Ford algorithm on a DAG? Does Bellman-Ford also returns the shortest path on the DAG?
- Why do we need to have a new algorithm for finding shortest paths on DAGs?

- $\Theta(|V|+|E|)$ vs. $\Theta(|V|*|E|)$