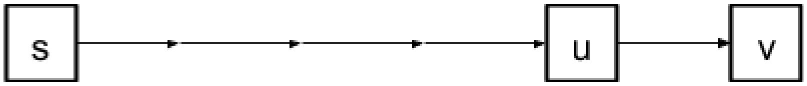
Basis of all shortest path algorithms: subpath of a shortest path is a shortest path. In other words, if a shortest path from s to v is composed of a path from s to u and the edge (u,v), then $\delta(s,v) = \delta(s,u) + w(u,v)$.



Therefore, shortest paths can be encoded as an up-tree, where each node stores its predecessor in a shortest path from s to that node. In the example above, we can set $v.\pi = u$. The following utility functions are used by all shortest path algorithms with edge weights:

```
initialize( s ):
    for u ∈ V do
        u.d ← ∞
        u.π ← null
        u.seen ← false
    s.d ← 0
```

```
boolean relax( u, v, e ):
    if v.d > u.d + e.weight then
        v.d ← u.d + e.weight
        v.π ← u
        return true
    return false
```

DAG-shortest-paths algorithm:In a DAG, the nodes in any path are in strictly increasing order of their topological numbers, in any topological ordering of V. This can be exploited to design the following efficient algorithm for shortest paths in DAGs:

```
// Pull algorithm
dagSP( g, s ):
    Find a topological ordering of g
    initialize( s )
    for u ∈ V in topological order do
        // LI: for predecessors p of u,  p.d = δ(s, p)
        for edge e = (p, u) into u do
            relax( p, u, e )
```

```
// Push algorithm
dagSP( g, s ):
    Find a topological ordering of g
    initialize( s )
    for u ∈ V in topological order do
        // LI: u.d = δ(s, u)
        for edge e = (u, v) out of u do
            relax( u, v, e )
```

Dijkstra's algorithm: applicable in graphs without any edges of negative weight.

Idea:

* Maintain a set of nodes S for which shortest paths are known.
* For $v \in V - S$, store in v.d, the length of a shortest path from s to v that goes through only nodes of S.
* In each iteration, select a node u in $V - S$ with minimum u.d, and add it to S.
* Relax edges out of u to update distance estimates of other nodes in $V - S$.

Code resembles Prim's algorithm that uses indexed priority queues:

```
dijkstraSP( G, s ):  // Implementation using indexed priority queue of vertices
    // v ∈ V−S stores in v.d, the weight of a shortest path from s to v that goes through only nodes of S
  initialize( s )   // for u ∈ V do { u.d ← ∞;  u.π ← null;  u.seen ← false }  s.d ← 0
  q ← new indexed priority queue of vertices with u.d as priority of u
  while q is not empty do
      u ← q.remove( )    // LI:  u.d = δ(s, u)
      u.seen ← true
      for Edge e = ( u, v ) incident on u do
          if not v.seen and relax( u, v, e ) then
              q.decreaseKey( v )
```

Dijkstra's algorithm for shortest paths

Input: $G = (V, E)$ (directed or undirected),

$w: E \to \mathbb{Z}^+$ ~ nonnegative weights on edges.

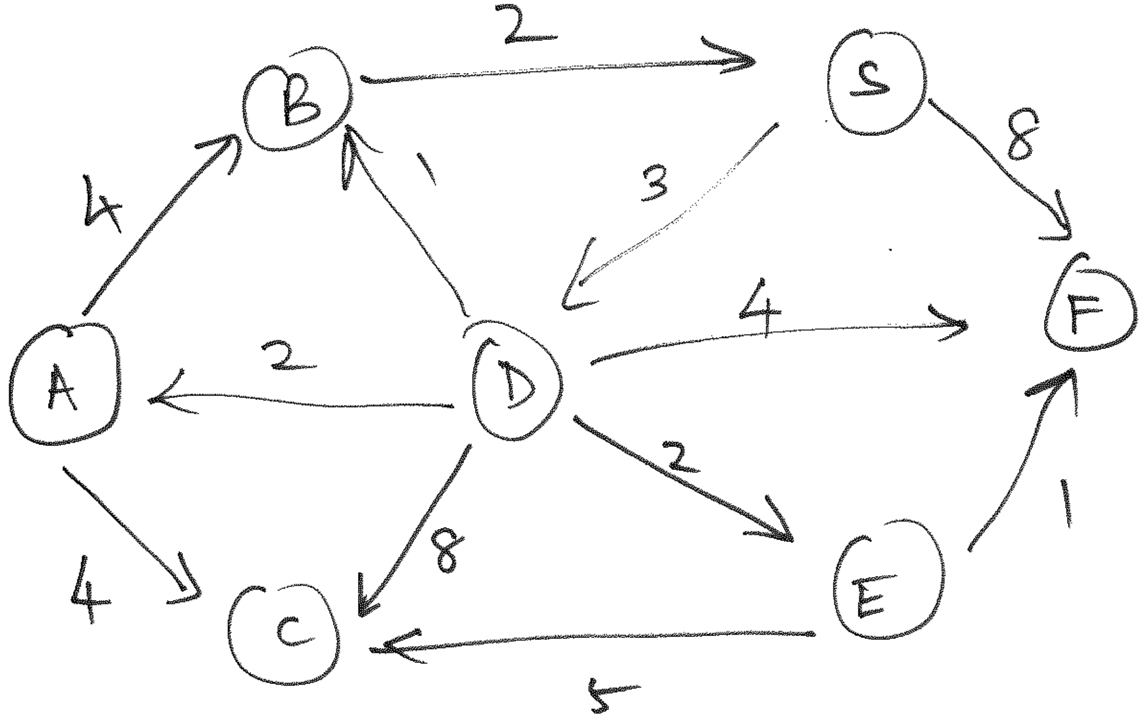
source $s \in V$ $(\mathbb{R}^+)$

Output: For all $u \in V$, $\delta(s,u)$, $\pi[u]$

shortest path weight from s to u

predecessor of u in this shortest path

Example:

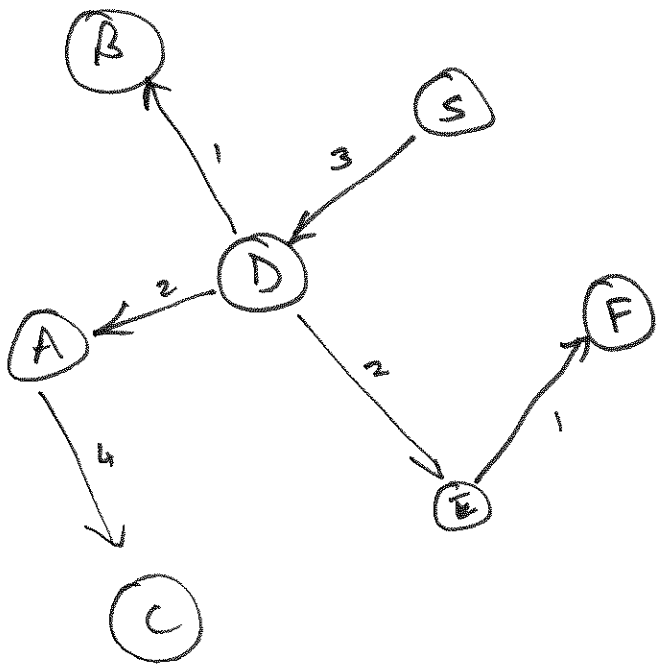


$u.d$ = estimate of how far u is from s.

Execution of Dijkstra's algorithm on this example:

Order in which nodes are added to S = {s, D, B, E, A, F, C}

<u>shortest path tree</u>



Vertex	d	π
S	0	–
A	~~∞~~ 5	D
B	~~∞~~ 4	D
C	~~∞~~ ~~11~~ ~~10~~ 9	~~D~~ ~~E~~ A
D	~~∞~~ 3	S
E	~~∞~~ 5	D
F	~~∞~~ ~~8~~ ~~7~~ 6	~~S~~ ~~D~~ E

Final Exam: 8:00–10:30 AM on Mon, Dec 17.

Topics: All topics discussed in class/assignments/projects

4 cheat sheets allowed, 8½" × 11" paper – both sides (8 pages in all)

Exam will have 3 sections.

A+/A/A− grades can be earned only by answering Sections 2 and 3. To get B+ … F grades, Section 3 need not be answered.

Idea: Maintain a set S of nodes, such that

for $u \in S$, $u.d = \delta(s,u)$.

for $v \in V - S$, $v.d$ = Length of a shortest path from s to v, all of whose internal nodes are in S..

Nodes of $V-S$ are kept in a priority queue with $u.d$ as priority of u.

In each iteration, remove node with smallest $u.d$, process edges out of u.

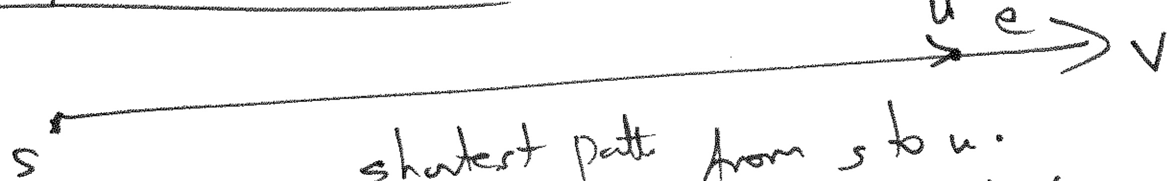
RT of Dijkstra's algorithm:

$O(|E| \log |V|)$ with indexed binary heaps

$O(|E| + |V| \log |V|)$ with Fibonacci heaps

---

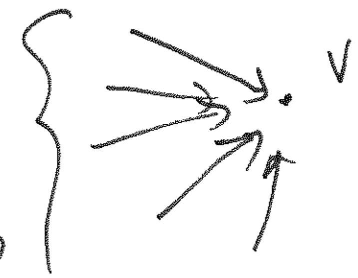
shortest paths in DAGs: $\delta(s,v) = \delta(s,u) + w(u,v)$.



shortest path from s to u.

Suppose, we know that $u.d = \delta(s,u)$. $\rightarrow$ Relax(u,v,e) : $v.d = \delta(s,v)$

which one of these nodes is predecessor of v in $\delta(s,v)$?

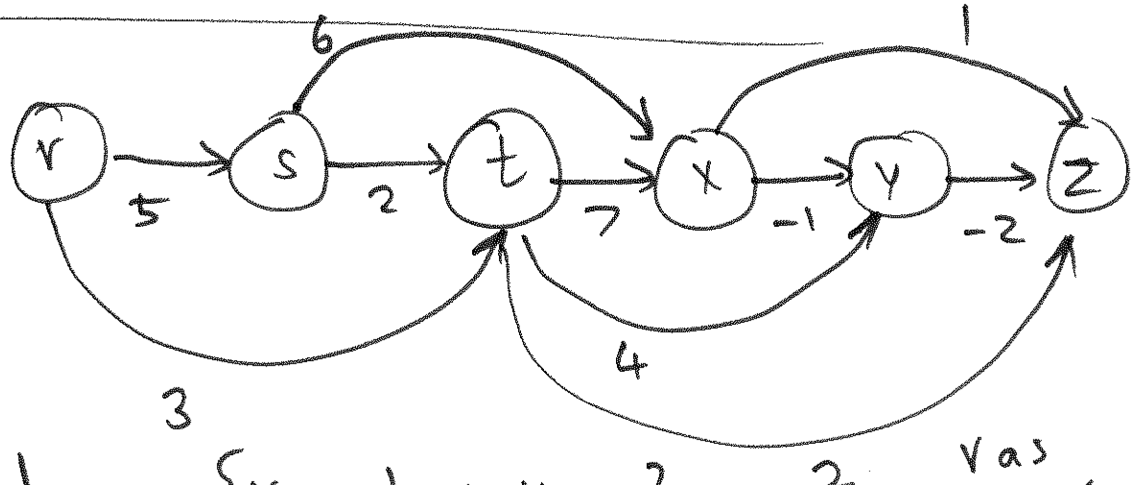


What if we wait until all predecessors of v have been processed?

Then we relax all edges into v - one of these edges is the right one.

Process nodes in topological order. $RT = O(|E| + |V|)$

Example:



Topological order = $\{r, s, t, x, y, z\}$

s as source →

		d	π
✓	r	∞	—
✓	s	0	—
✓	t	$\cancel{\infty}$ 2	s
✓	x	$\cancel{\infty}$ 6	s
	y	$\cancel{\infty}$ $\cancel{6}$ 5	$\cancel{t}$ x
	z	$\cancel{\infty}$ $\cancel{4}$ 3	$\cancel{t}$ y

r as source ↙

✓	r	0	—
✓	s	$\cancel{\infty}$ 5	r
✓	t	$\cancel{\infty}$ 3	r
✓	x	$\cancel{\infty}$ $\cancel{11}$ 10	$\cancel{s}$ t
✓	y	$\cancel{\infty}$ 7	t
✓	z	$\cancel{\infty}$ 5	t

Bellman-Ford Algorithm

Input: Graph $G = (V, E)$ – directed graph

$w : E \to \mathbb{Z}$ – positive and negative weights

$(\mathbb{R})$

source $s \in V$

Output: either: (1) For each $u \in V$: $\delta(s,u)$, $\pi[u]$ if G has no negative cycles

or (2) Discover a negative cycle in G.

Negative cycle: cycle C: $\sum_{e \in C} w(e) < 0$.

Idea: (1) Def: $d_k(u)$ = length of a shortest (simple) path from s to u using at most k edges.

(2) If a graph has no negative cycles, a path P from s to u that is not simple can be used to get a path P' from s to u that is simple, $w(P') \leq w(P)$.

$\Rightarrow$ shortest path with no constraints on simplicity is fine.