Today:

- Review of:
  - Heaps, Priority Queues
  - Basic Graph Algs.
- Algs for SSSP (Bellman-Ford, Topological sort for DAGs, Dijkstra)

COSC 581, Algorithms

February 4, 2014

Many of these slides are adapted from several online sources

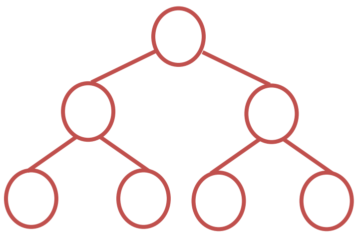
Reading Assignments

- Today's class:
  - Chapter 6, 22, 24.0, 24.1, 24.2, 24.3
- Reading assignment for next class:
  - Chapter 25.1-25.2

- Announcement: Exam 1 is on Tues, Feb. 18
  - Will cover everything up through dynamic programming

Heaps & Priority Queues

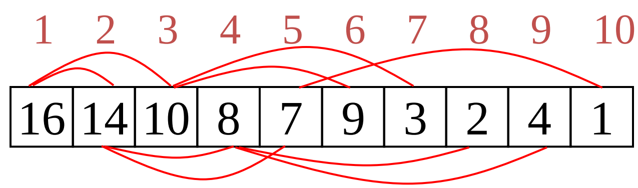
Complete binary tree:



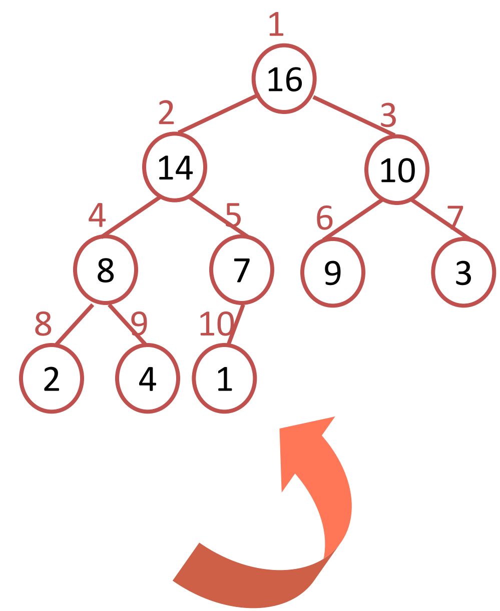
- All leaves have the same depth
- All internal nodes have 2 children

The (binary) heap data structure is:

an array object



that can be viewed as
a nearly complete binary tree



Parent(i) = $\lfloor i/2 \rfloor$
Left(i) = 2i
Right(i) = 2i+1

Heap Property:

- For a max-heap: child <= parent
- For a min-heap: child >= parent

Maintaining Heap Property

MAX-HEAPIFY(A,i)

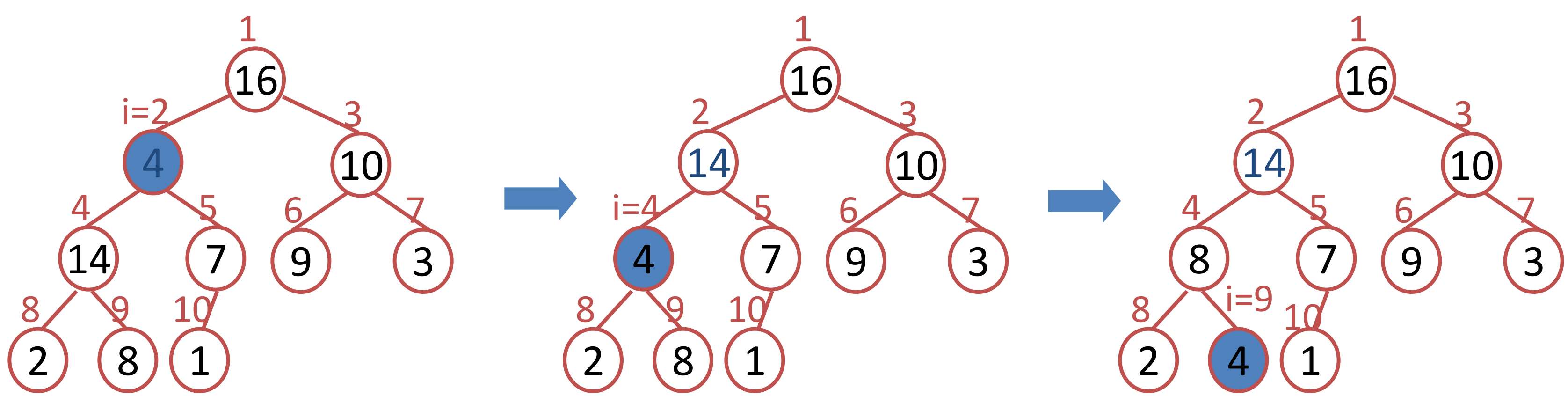
..

The binary trees rooted at LEFT(i) and RIGHT(i) are max-heaps

But A[i] may be smaller than its children.

MAX-HEAPIFY is to "float down" A[i] to make the subtree rooted at A[i] a max-heap.

O(height of node i)
= O(lg n)



Heaps & Priority Queues

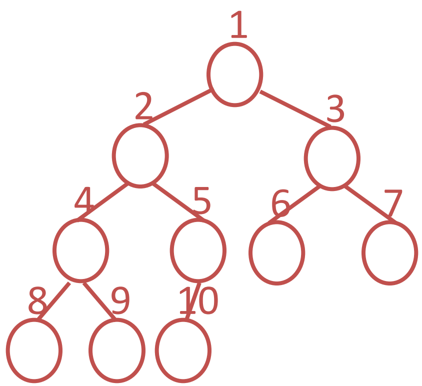
<u>Maximum No. of elements</u>

level 0: 1

level 1: 2

level 2: 4

level 3: 8



<u>Maximum No. of elements</u>

a one-level tree	(height=0):	1
a 2-level tree	(height=1):	3
a 3-level tree	(height=2):	7
a 4-level tree	(height=3):	15

Therefore, for a heap containing *n* elements :

Maximum no. of elements in level k = 2^k

Height of tree = $\lfloor \lg n \rfloor = \Theta(\lg n)$

Basic procedures:

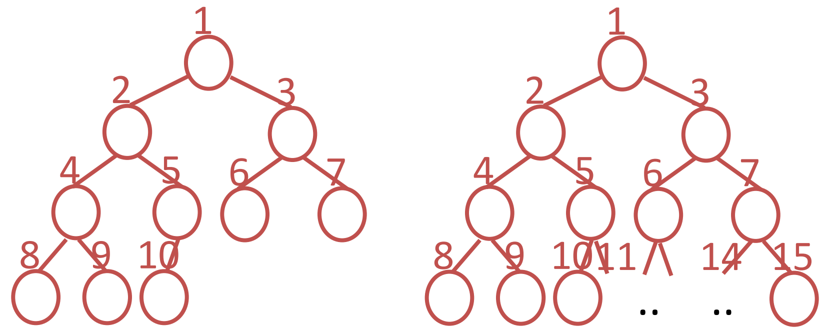
MAX-HEAPIFY	O(lg n)	HEAP-EXTRACT-MAX	O(lg n)
BUILD-MAX-HEAP	O(n)	HEAP-INCREASE-KEY	O(lg n)
MAX-HEAP-INSERT	O(lg n)	HEAP-MAXIMUM	O(lg n)

Heaps & Priority Queues

Building a heap:

```
BUILD-MAX-HEAP(Input_numbers)
    Copy Input_numbers to a heap
    For i = ⌊ n/2 ⌋ down to 1 /*all non-leaf nodes */
        MAX-HEAPIFY(A,i)
```

$\boxed{O(n)}$



Note that $\lceil n/2 \rceil$ the elements are leaf nodes

Illustration for a Complete-binary tree:

A complete-binary tree of height h has h+1 levels: 0,1,2,3,.. h.

The levels have $2^0,2^1,2^2,2^3,\ldots 2^h$ elements respectively.

Then, maximum total no. of "float down" carried out by MAX-HEAPIFY

= sum of maximum no. of "float down" of all non-leaf nodes (levels h-1, h-2, .. 0)

$= 1 \times 2^{h-1} + 2 \times 2^{h-2} + 3 \times 2^{h-3} + 4 \times 2^{h-4} + .. h \times 2^0$

$= 2^h (1/2 + 2/4 + 3/8 + 4/16\ldots)$ [note: $2^{h+1} = n+1$, thus $2^h=0.5*(n+1)$]

$= 0.5(n+1) (1/2 + 2/4 + 3/8 + 4/16\ldots)$ [note: $1/2 + 2/4 + 3/8 + 4/16.. <2$]

$< 0.5(n+1) * 2 = (n+1)$

$= O(n)$

Priority Queue

- Priority queue is a data structure for maintaining a set of elements each associated with a key.
- Maximum priority queue supports the following operations:

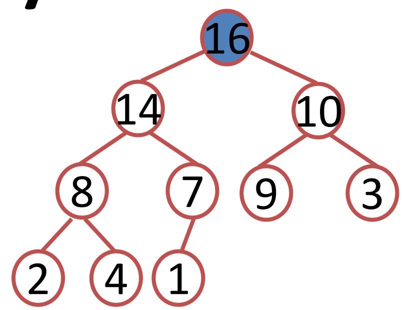
INSERT(S,x)	- Insert element x into the set S
MAXIMUM(S)	- Return the 'largest' element
EXTRACT-MAX(S)	- Remove and return the 'largest' element
INCREASE-KEY(S,x,v)	- Increase x's key to a new value, v

We can implement priority queues based on a heap structure.

Heaps & Priority Queues

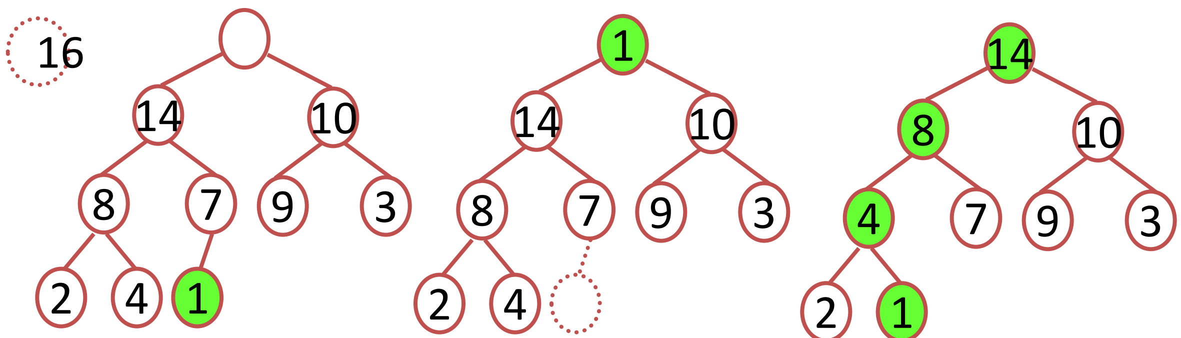
```
MAXIMUM(A)
    return A[1]
```

$\Theta(1)$



```
HEAP-EXTRACT-MAX(A)
```

$O(\lg n)$



Step 1. Save the value of the root that is to be returned.

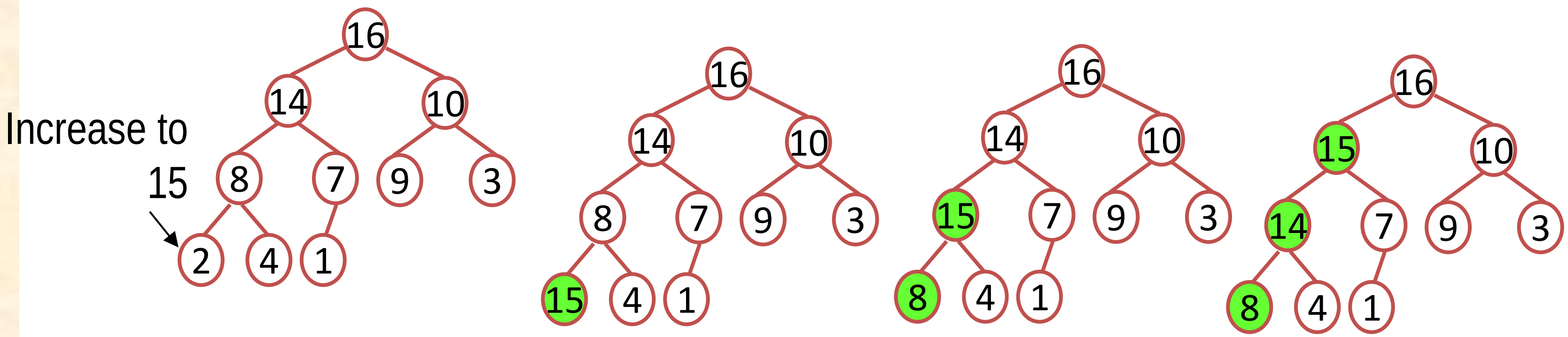
Step 2. Move the last value to the root node.

Step 3. MAX-HEAPIFY(A,1/*the root node*/).

Heaps & Priority Queues

HEAP-INCREASE-KEY(A,i,v)

O(lg n)

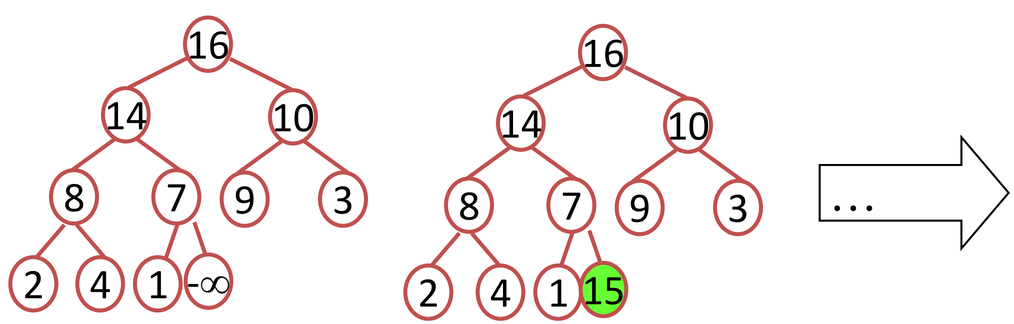


Keep on exchanging with parent until parent is greater than the current node.

O(lg n)

MAX-HEAP-INSERT(A,key)

```
n = n+1
A[n]= - ∞
HEAP-INCREASE-KEY(A,n,key)
```



Graph Representation

Given graph $G = (V, E)$.

- May be either directed or undirected.
- Two common ways to represent for algorithms:
  1. *Adjacency lists.*
  2. *Adjacency matrix.*

Expressing the running time of an algorithm is often in terms of both $|V|$ and $|E|$.

In asymptotic notation - and *only* in asymptotic notation - we'll drop the cardinality. Example: $O(V + E)$.

Adjacency lists

Array *Adj* of $|V|$ lists, one per vertex.

Vertex *u*'s list has all vertices *v* such that $(u, v) \in E$. (Works for both directed and undirected graphs.)

If edges have *weights*, can put the weights in the lists.

Weight: $w : E \rightarrow \mathrm{R}$

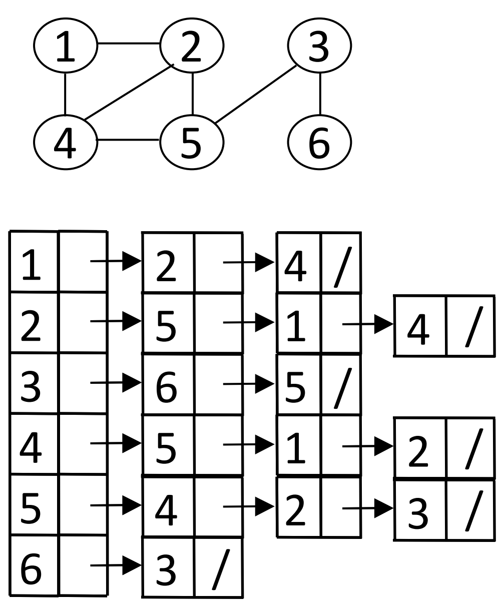
We'll use weights later on for shortest paths.

Space: $\Theta(V + E)$.

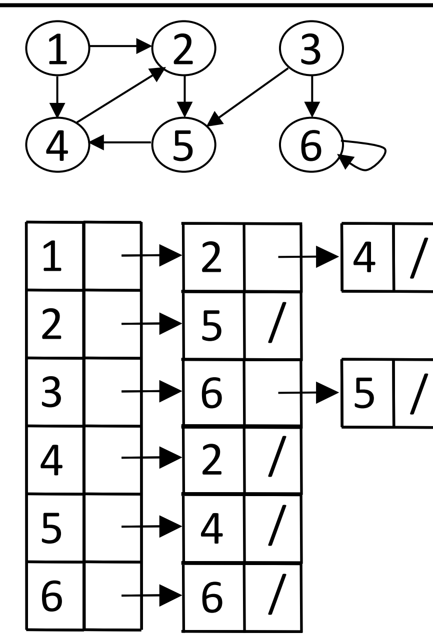
Time: to list all vertices adjacent to *u*: Θ(degree(u)).

Time: to determine if $(u, v) \in E$: O(degree(u)).

Undirected graph:



Directed graph:



Adjacency Matrix

$|V| \times |V|$ matrix $A = (a_{ij})$

$$a_{ij} = \begin{cases} 1 & \text{if } (i, j) \in E, \\ 0 & \text{otherwise}. \end{cases}$$

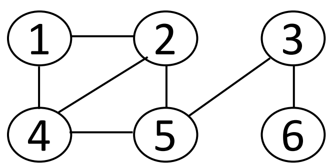
Space: $\Theta(V^2)$

Time: to list all vertices adjacent to u: $\Theta(V)$.

Time: to determine if $(u, v) \in E$: $O(1)$.

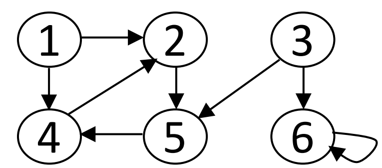
Can store weights instead of bits for weighted graph.

Undirected graph:



$$\begin{array}{c|cccccc|} a & 1 & 2 & 3 & 4 & 5 & 6 \\ \hline 1 & 0 & 1 & 0 & 1 & 0 & 0 \\ 2 & 1 & 0 & 0 & 1 & 1 & 0 \\ 3 & 0 & 0 & 0 & 0 & 1 & 1 \\ 4 & 1 & 1 & 0 & 0 & 1 & 0 \\ 5 & 0 & 1 & 1 & 1 & 0 & 0 \\ 6 & 0 & 0 & 1 & 0 & 0 & 0 \\ \hline \end{array}$$

Directed graph:



$$\begin{array}{c|cccccc|} & 1 & 2 & 3 & 4 & 5 & 6 \\ \hline 1 & 0 & 1 & 0 & 1 & 0 & 0 \\ 2 & 0 & 0 & 0 & 0 & 1 & 0 \\ 3 & 0 & 0 & 0 & 0 & 1 & 1 \\ 4 & 0 & 1 & 0 & 0 & 0 & 0 \\ 5 & 0 & 0 & 0 & 1 & 0 & 0 \\ 6 & 0 & 0 & 0 & 0 & 0 & 1 \\ \hline \end{array}$$

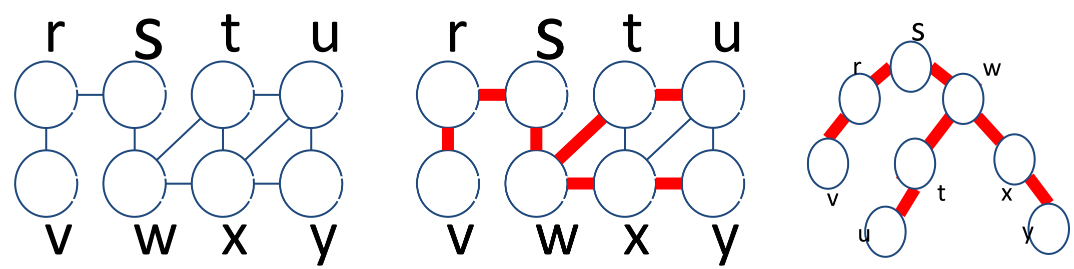
Breadth-First Search

- **Input:**
  Graph $G = (V, E)$, either directed or undirected, and ***source vertex*** $s \in V$.
- **Output:**
  $d[v]$ = distance (smallest # of edges) from s to v, for all $v \in V$.
  Also $\pi[v] = u$ such that (u, v) is last edge on shortest path $s \rightsquigarrow v$
  - u is v's ***predecessor***.
  - set of edges $\{(\pi[v], v) : v = s\}$ forms a tree.
- Later, a breadth-first search will be generalized with edge weights.
  Now, let's keep it simple.
  – Compute only $d[v]$, not $\pi[v]$.
  – Omitting colors of vertices.
- ***Idea:*** Send a wave out from s.
  – First hits all vertices 1 edge from s.
  – From there, hits all vertices 2 edges from s.
  – Etc.
- Use FIFO queue Q to maintain wavefront.
  – $v \in Q$ if and only if wave has hit v but has not come out of v yet.

Breadth-First Search

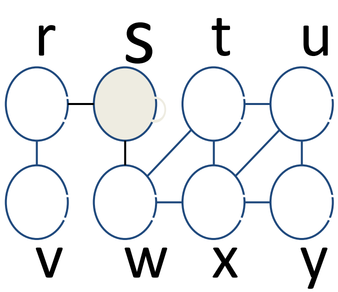
Breadth-First Search (BFS)

Explores the edges of a graph to reach every vertex from a vertex **s**, with “shortest paths”



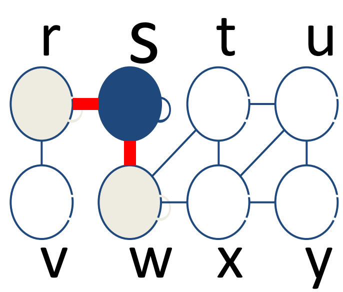
The algorithm:

Start by inspecting the source vertex S:



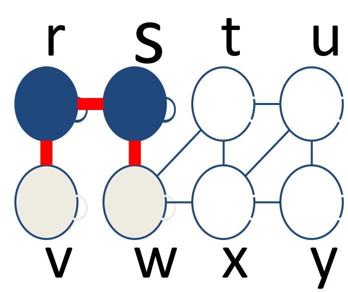
For s, its 2 neighbors are not yet searched

So we connect them:



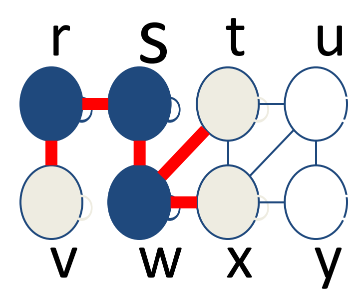
Now r and w join our solution

For r, we do the same to its white color neighbors:



Now v joins our solution

For w, we do the same to its white color neighbors:



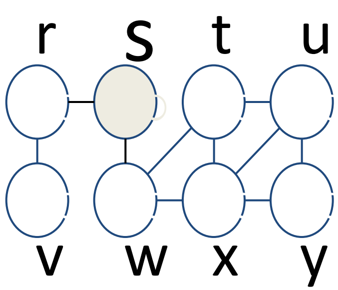
Now t and x join our solution

…

Breadth-First Search

Using 3 colors: white / gray / black

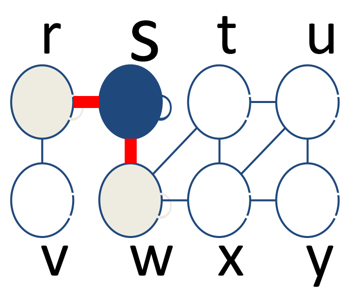
Start by inspecting the source vertex S:



For s, its 2 neighbors are not yet searched

Since s is in our solution, and it is to be inspected, we mark it <u>gray</u>

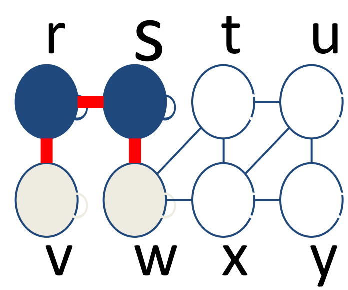
So we connect them:



Now r and w join our solution

No more need to check s, so mark it <u>black</u>. r and w join our solution, we need to check them later on, so mark them <u>gray</u>.

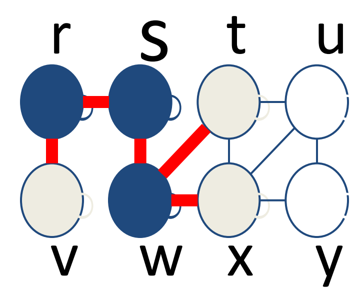
For r, we do the same to its <u>white</u> color neighbors:



Now v joins our solution

No more need to check r, so mark it <u>black</u>. v joins our solution, we need to check it later on, so mark it <u>gray</u>.

For w, we do the same to its <u>white</u> color neighbors:



Now t and x join our solution

No more need to check w, so mark it <u>black</u>. t and x join our solution, we need to check them later on, so mark them <u>gray</u>.

...

Breadth-First Search Algorithm

```
BFS(G,s)  /*G=(V,E)*/
For each vertex u in V - {s}
        u.color = white
        u.distance = ∞
        u.pred = NIL
s.color = gray
s.distance = 0
s.pred = NIL
Q = ∅
ENQUEUE(Q,s)
while Q ≠ ∅
        u = DEQUEUE(Q)
        for each v adjacent to u
                if v.color = white
                        v.color = gray
                        v.distance = u.distance + 1
                        v.pred = u
                        ENQUEUE(Q,v)
        u.color = black
```

$\Theta(V)$

The running time of BFS is: O(V+E)

Total number of edges kept by the adjacency list is $\Theta(E)$

Total time spent in the adjacency list is O(E)

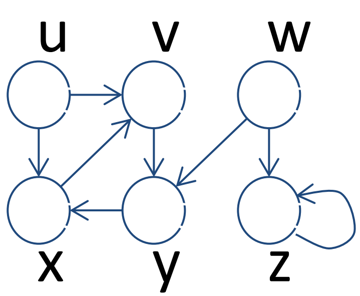
Depth-First Search

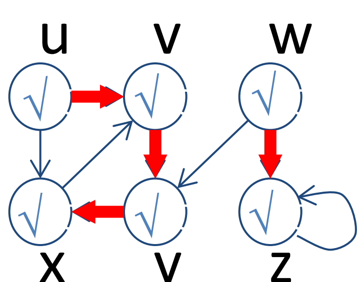
- **Input:**
  Graph $G = (V, E)$, either directed or undirected. No source vertex given.
- **Output:** 2 ***timestamps*** on each vertex:
  - $d[v]$ = *discovery time*.
  - $f[v]$ = *finishing time.*
  - $\pi[v]$: *v's predecessor field.*
- Will methodically explore *every* edge.
  – Start over from different vertices as necessary.
- As soon as we discover a vertex, explore from it.
  – Unlike BFS, which puts a vertex on a queue so that we explore from it later.
- As DFS progresses, every vertex has a ***color***:
  – WHITE = undiscovered
  – GRAY = discovered, but not finished (not done exploring from it)
  – BLACK = finished (have found everything reachable from it)
- Discovery and finish times:
  – Unique integers from 1 to 2 $|V|$.
  – For all v, $\mathbf{d[v] < f[v]}$.
- In other words, $\mathbf{1 \leq d[v] < f[v] \leq 2\,|V|}$.

Depth-First Search

Depth-First Search (BFS)

Explores the edges of a graph by searching "deeper" whenever possible.






```
DFS(G) /*G = (V,E) */
    for each vertex u in V
        u.color = white
        u.pred = NIL
    for each vertex u in V
        if u.color = white
            DFS-VISIT(u)
```

Θ(V)

Θ(V) + Time to execute calls to DFS-VISIT

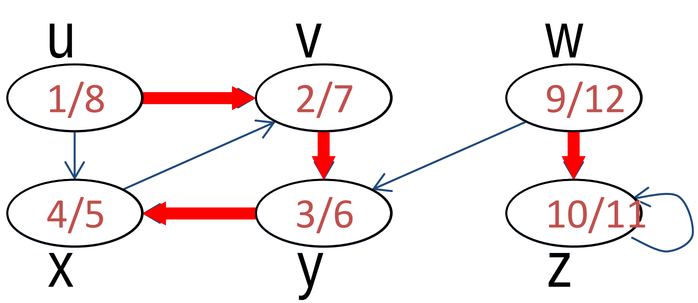
The running time of DFS is: Θ(V+E)

```
DFS-VISIT(u)
    u.color = gray
    for each v adjacent to u
        if v.color = white
            v.pred = u
            DFS-VISIT(v)
    u.color = black
```

Total number of edges kept by the adjacency list is Θ(E). Total time spent in the adjacency list is Θ(E).

Depth-First Search

On many occasions it is useful to keep track of the discovery time and the finishing time while checking each node.




```
DFS(G) /*G = (V,E) */
for each vertex u in V
    u.color = white
    u.pred = NIL
time = 0
for each vertex u in V
    if u.color = white
        DFS-VISIT(u)
```

```
DFS-VISIT(u)
u.color = gray
time = time + 1
u.discover = time
for each v adjacent to u
    if v.color = white
        v.pred = u
        DFS-VISIT(v)
u.color = black
time = time + 1
u.finish = time
```

Properties of Depth-First Search

Parenthesis theorem

For all *u, v*, exactly one of the following holds:

1. $d[u] < f[u] < d[v] < f[v]$ or $d[v] < f[v] < d[u] < f[u]$ and neither of *u* and *v* is a descendant of the other.
2. $d[u] < d[v] < f[v] < f[u]$ and *v* is a descendant of *u*.
3. $d[v] < d[u] < f[u] < f[v]$ and *u* is a descendant of *v*.

So $d[u] < d[v] < f[u] < f[v]$ *cannot* happen.

Like parentheses:

– OK: ( ) [ ] ([ ]) [( )]

– Not OK: ([ )] [( ])

Corollary

– *v* is a proper descendant of *u* if and only if $d[u] < d[v] < f[v] < f[u]$.

White-path theorem

v is a descendant of *u* if and only if at time $d[u]$, there is a path $u \rightsquigarrow v$ consisting of only white vertices.

(Except for *u*, which was *just* colored gray.)

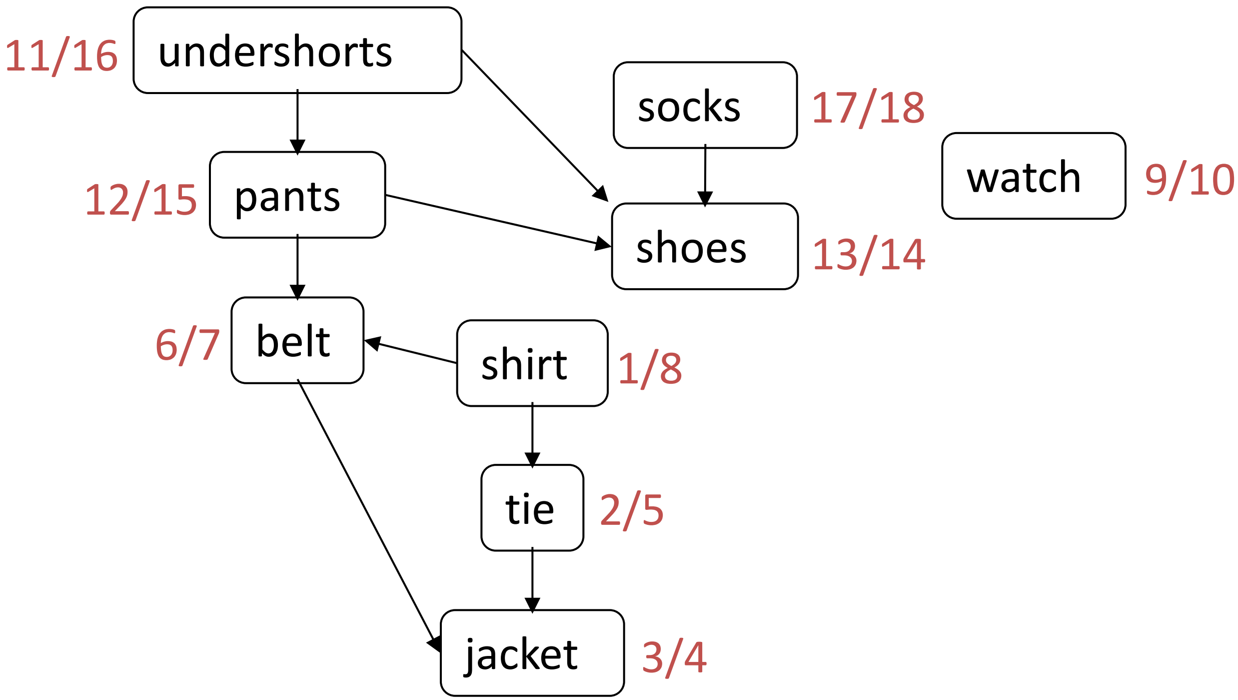
Classification of edges

– ***Tree edge:*** in the depth-first forest. Found by exploring *(u, v)*.
– ***Back edge:*** *(u, v)*, where *u* is a descendant of *v*.
– ***Forward edge:*** *(u, v)*, where *v* is a descendant of *u*, but not a tree edge.
– ***Cross edge:*** any other edge.
Can go between vertices in same depth-first tree or
in different depth-first trees.

In an undirected graph, there may be some ambiguity since *(u, v)* and *(v, u)* are the same edge. Classify by the first type above that matches.

Theorem
In DFS of an *un*directed graph, we get only tree and back edges.
No forward or cross edges.

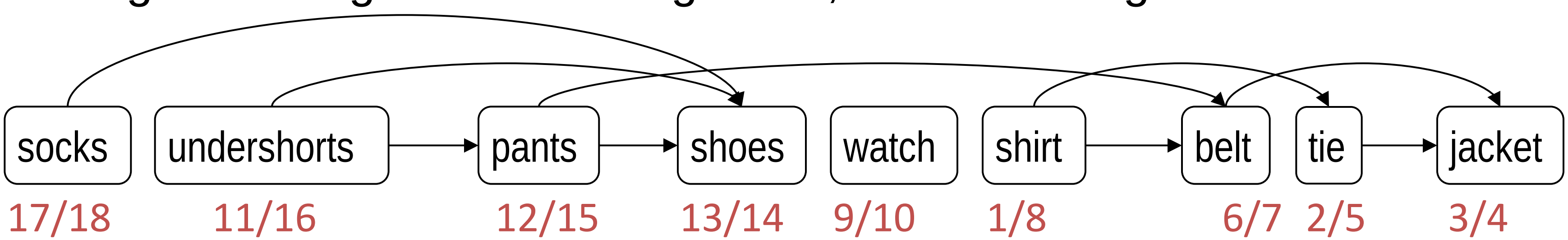
Topological Sort of a DAG



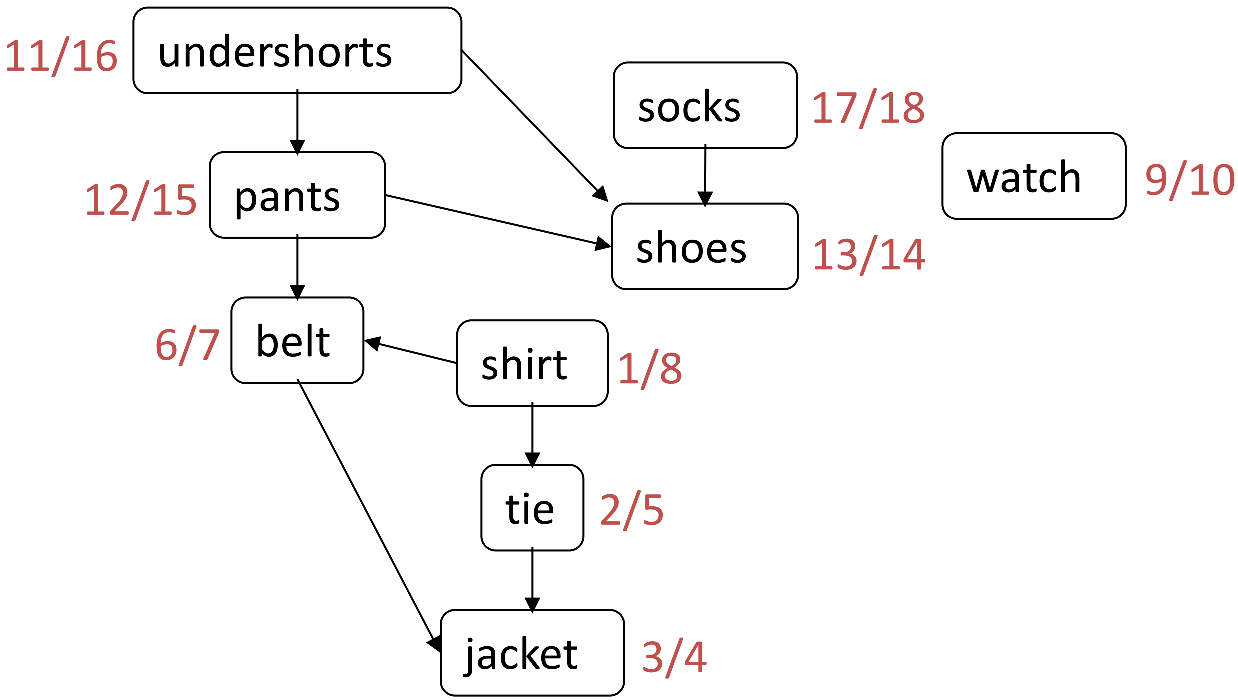
Topological Sort

- A linear ordering of vertices : if the graph contains an edge (u,v), then u appears before v.
- Applied to directed acyclic graphs (DAG)

Sorting according to the finishing times, in descending order:



Topological Sort of a DAG

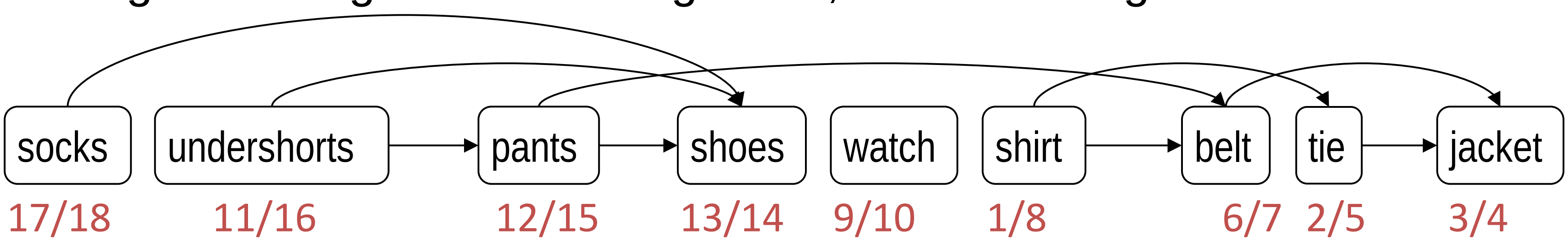


TOPOLOGICAL-SORT(G)

1. Call DFS(G) to compute finishing times v.finish for each vertex v
2. As each vertex is finished, insert it onto the front of a linked list
3. Return the linked list of vertices

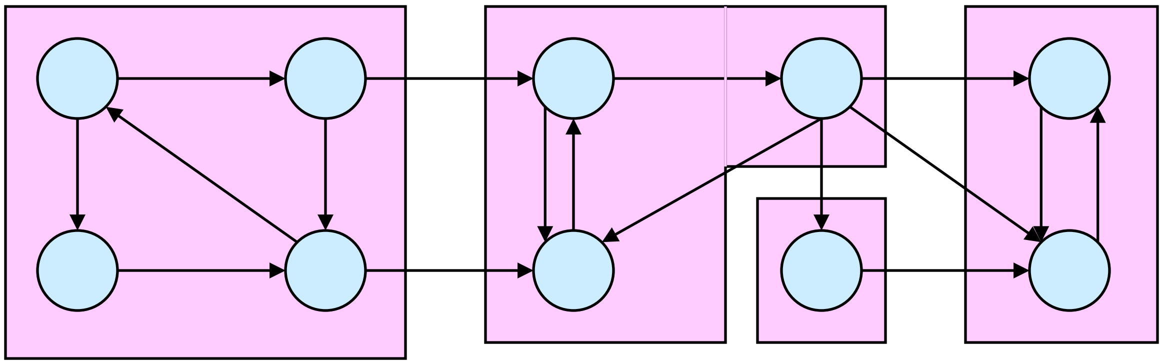
$\boxed{\Theta(V+E)}$

Sorting according to the finishing times, in descending order:



Strongly Connected Components

- Given directed graph $G = (V, E)$.
- A ***strongly connected component*** (***SCC***) of G is a maximal set of vertices $C \subseteq V$ such that for all $u, v \in C$, both $u \rightsquigarrow v$ and $v \rightsquigarrow u$
- ***Example:***



- Algorithm uses G^T = ***transpose*** of G:
  - $G^T = (V, E^T)$, $E^T = \{(u, v) : (v, u) \in E\}$.
  - G^T is G with all edges reversed.
- Can create G^T in $(V + E)$ time if using adjacency lists.
- ***Observation:*** G and G^T have the *same* SCC's. (u and v are reachable from each other in G if and only if reachable from each other in G^T.)

Algorithm For Strongly Connected Components

STRONGLY-CONNECTED-COMPONENTS(G)

call DFS(G) to compute finishing times *u.f* for each vertex *u*

compute G^T

call DFS(G^T), but in the main loop of DFS, consider the vertices in order of decreasing *u.f* (as computed above)

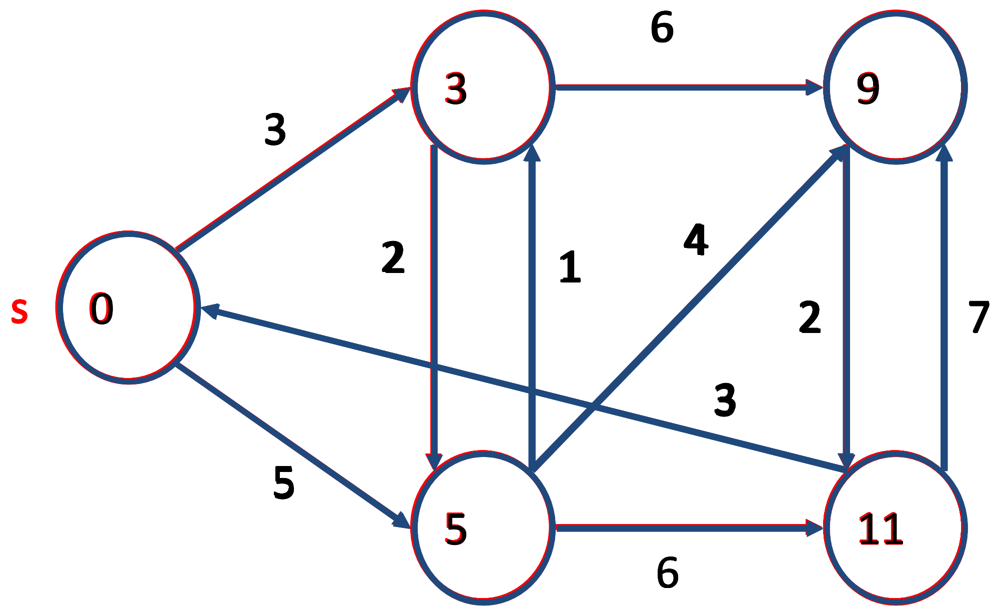
output vertices of each tree from previous DFS(G^T) call as a separate strongly connected component

Runtime: $\Theta(V+E)$

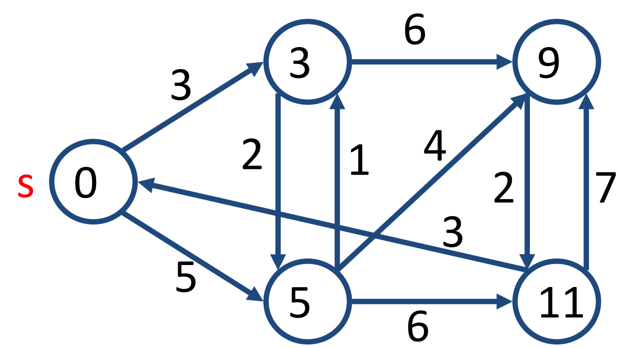
Single-Source Shortest Paths

Single-Source Shortest Paths

Given a weighted, directed graph, find the shortest paths from a given source vertex s to other vertices.

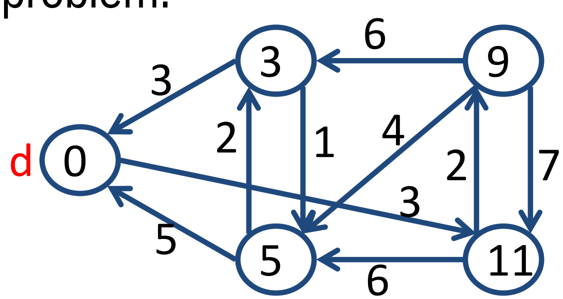


SSSP Variants



Single-destination shortest-path problem

By reversing the direction of each edge, we can reduce this problem to a single-source problem.



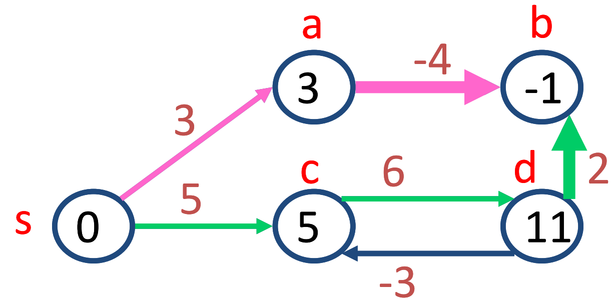
Single-pair shortest-path problem

If the single-source problem is solved, we can solve this problem also. There are no asymptotically faster algorithms.

All-pairs shortest-path problem

Can be solved by running a single source algorithm once for each source vertex. However, other faster approaches exist.

Single-Source Shortest Paths



Optimal substructure of a shortest path:

A shortest path between 2 vertices contains other shortest paths within it.

Edge weight & Path weight :

Edge weight: eg. $w(c,d) = 6$

Path weight: eg. For a path p=<s,c,d>, $w(p) = w(s,c) + w(c,d) = 11$

Shortest-path weight:

Define shortest-path weight for a path **p** from **u** to **v** as:

$$\delta(u,v) = \begin{cases} \min\{ w(p) : u \overset{p}{\rightsquigarrow} v \} & \text{if there is a path from u to v} \\ \infty & \text{otherwise} \end{cases}$$

Single-Source Shortest Paths

Negative-weight edges

eg. w(a,b) = -4

Negative-weight path

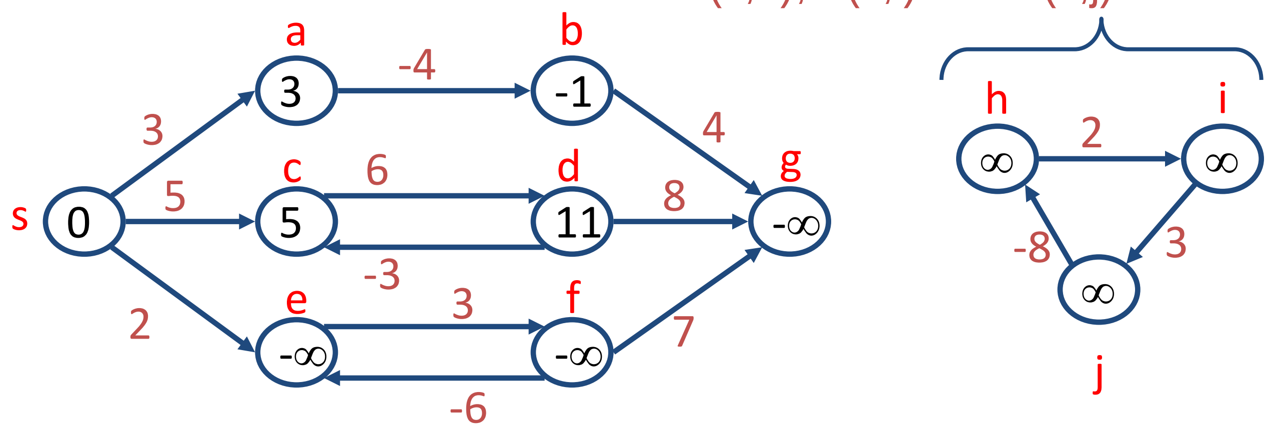
eg. <s,a,b>: -1

Negative-weight cycle

eg. <e,f,e>: -3

h, i, and j are not reachable from s

=> $\delta(s,h)$, $\delta(s,i)$ and $\delta(s,j)$ are ∞



> If there is no negative weight cycle reachable from the source vertex s, then for all v in V, the shortest-path weight $\delta(s,v)$ remains well defined.

A well defined shortest path has no cycle. Prove:

1. A shortest path should not contain non-negative weight cycle.
   [otherwise reducing the cycle would give a more optimal path]
2. A well defined shortest path should not contain negative weight cycle

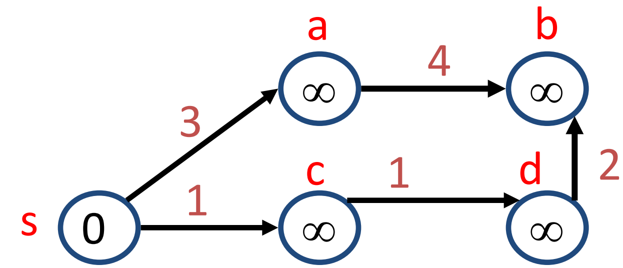
=> A well defined shortest path has no cycle, and has at most |V|-1 edges.

Single-Source Shortest Paths

A general function for single-source shortest paths algorithms:

```
INITIALIZE-SINGLE-SOURCE()
For each vertex v in V
        v.d = ∞
        v.pred = NIL
s.d = 0
```

$\Theta(V)$



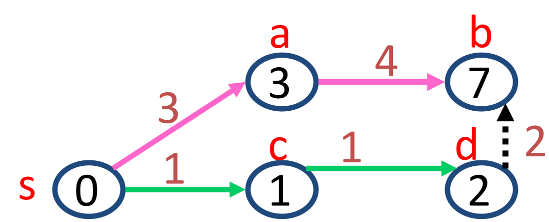
Where v.d is the upper bound on the weight of a shortest path from source vertex s to v.

A general technique for single-source shortest paths algorithms:

Relaxation

"Relaxing an edge (d,b)" :

Testing whether we can improve the shortest path to b found so far by going through d, if so, update b.d and b.pred.




```
RELAX(u,v)
if v.d > u.d + w(u,v)
        v.d = u.d + w(u,v)
        v.pred = u
```

Single-Source Shortest Paths

Three solutions to the problem:

Bellman-Ford algorithm

- By relaxing the whole set of edges |V|-1 times

Algorithm for directed acyclic graphs (DAG)

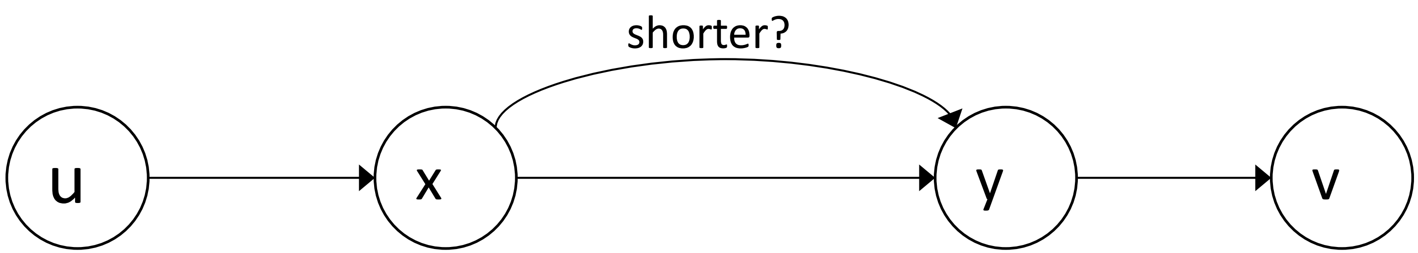
- By topological sorting the vertices first, then relax the edges of the sorted vertices one by one.

Dijkstra’s algorithm

- Handle non-negative edges only. Grow the solution by checking vertices one by one, starting from the one nearest to the source vertex.

A Fact About Shortest Paths – Optimal Substructure

- **Theorem:** If p is a shortest path from u to v, then any subpath of p is also a shortest path.
- **Proof:** Consider a subpath of p from x to y. If there were a shorter path from x to y, then there would be a shorter path from u to v.



Shortest-Paths Idea

- $\delta(u,v) \equiv$ length of the shortest path from *u* to *v*.
- All SSSP algorithms maintain a field d[*u*] for every vertex *u*. d[*u*] will be an estimate of $\delta(s,u)$. As the algorithm progresses, we will refine d[*u*] until, at termination, d[*u*] = $\delta(s,u)$. Whenever we discover a new shortest path to *u*, we update d[*u*].
- In fact, d[*u*] will always be an *overestimate* of $\delta(s,u)$:

$$d[u] \geq \delta(s,u)$$

- We'll use $\pi[u]$ to point to the parent (or predecessor) of *u* on the shortest path from *s* to *u*. We update $\pi[u]$ when we update d[*u*].

SSSP Subroutine

RELAX(u, v, w)

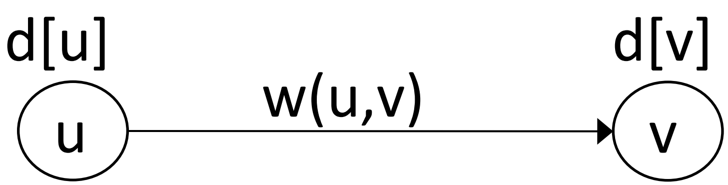
$\triangleright$ *(Maybe) improve our estimate of the distance to v*

$\triangleright$ *by considering a path along the edge (u, v).*

if $v.d > u.d + w(u,v)$ then

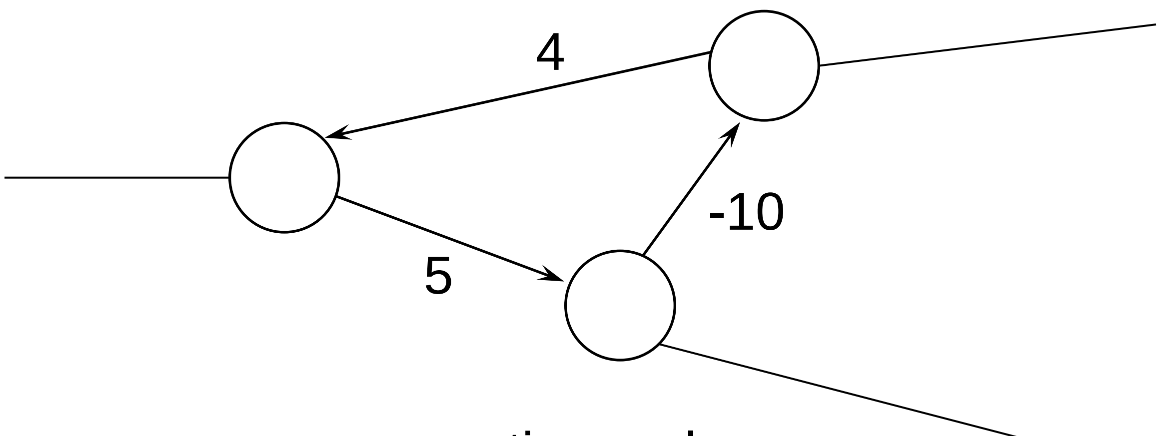
$v.d \leftarrow u.d + w(u, v)$ $\triangleright$ *actually, DECREASE-KEY*

$v.\pi \leftarrow u$ $\triangleright$ *remember predecessor on path*



The Bellman-Ford Algorithm

- Handles negative edge weights
- Detects negative cycles
- Is slower than Dijkstra



a negative cycle

Bellman-Ford: Idea

- Repeatedly update d for all pairs of vertices connected by an edge.
- **Theorem:** If u and v are two vertices with an edge from u to v, and $s \Rightarrow u \rightarrow v$ is a shortest path, and $u.d = \delta(s,u)$,

  then $u.d+w(u,v)$ is the length of a shortest path to v.
- **Proof:** Since $s \Rightarrow u \rightarrow v$ is a shortest path, its length is $\delta(s,u) + w(u,v) = u.d + w(u,v)$. ■

Why Bellman-Ford Works

- On the first pass, we find δ (s,u) for all vertices whose shortest paths have one edge.
- On the second pass, the d[*u*] values computed for the one-edge-away vertices are correct (= δ (s,u)), so they are used to compute the correct d values for vertices whose shortest paths have two edges.
- Since no shortest path can have more than |V[G]|-1 edges, after that many passes all d values are correct.
- Note: all vertices not reachable from s will have their original values of infinity. (Same, by the way, for Dijkstra).

Bellman-Ford: Algorithm

```
BELLMAN-FORD(G, w, s)
for each vertex v ∈V[G] do //INIT_SINGLE_SOURCE
    v.d ← ∞
    v.π ← NIL
s.d ← 0
for i ← 1 to |V[G]|-1 do  ▷ each iteration is a "pass"
    for each edge (u,v) in E[G] do
        RELAX(u, v, w)
▷ check for negative cycles
for each edge (u,v) in E[G] do
    if v.d > u.d + w(u,v) then
        return FALSE
return TRUE
```

O(V) (lines 1–3)

O(VE) (lines 5–8)

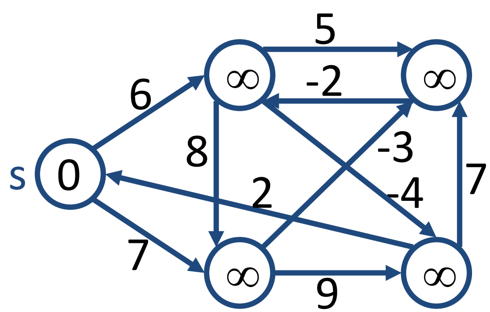
O(E) (lines 9–12)

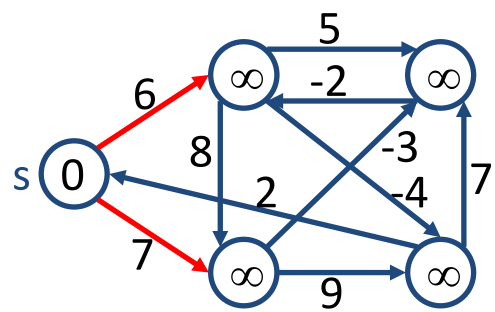
Running time: $\Theta(VE)$

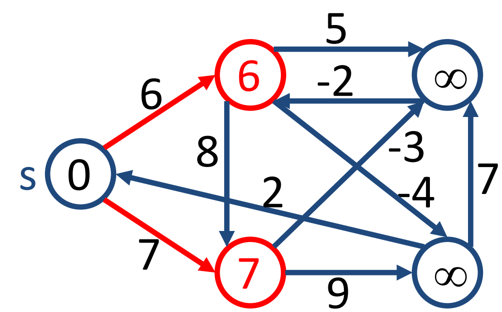
Bellman-Ford Algorithm

Method: Relax the whole set of edges |V|-1 times.

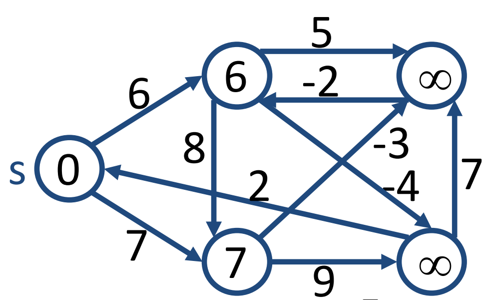
At 1st time:

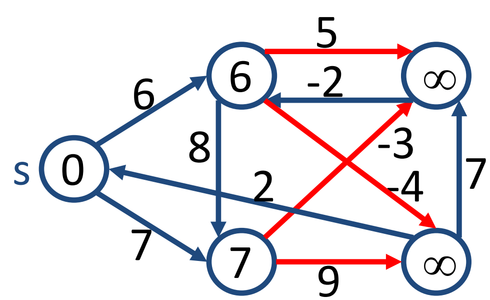


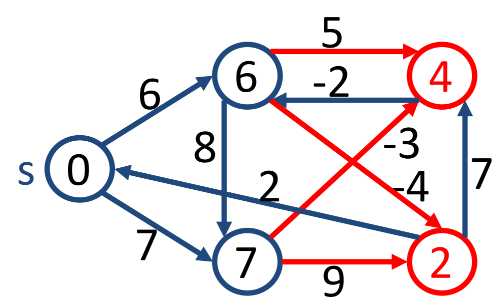




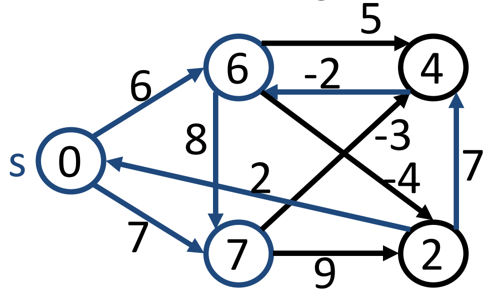
At 2nd time:



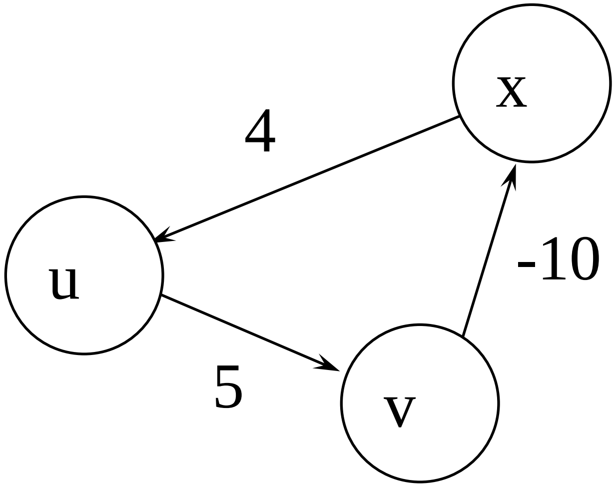




At 3rd , 4th time:



Negative Cycle Detection



- What if there is a negative-weight cycle reachable from s?
- Assume: $u.\mathrm{d} \leq x.\mathrm{d}+4$

  $v.\mathrm{d} \leq u.\mathrm{d}+5$

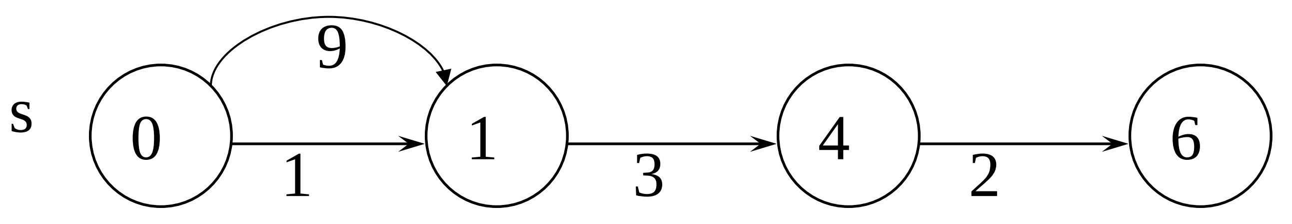
  $x.\mathrm{d} \leq v.\mathrm{d}-10$
- Adding:

  $u.\mathrm{d}+v.\mathrm{d}+x.\mathrm{d} \leq x.\mathrm{d}+u.\mathrm{d}+v.\mathrm{d}-1$
- Because it's a cycle, vertices on left are same as those on right. Thus we get $0 \leq -1$; a contradiction.
  So for at least one edge (u,v),

  $v.\mathrm{d} > u.\mathrm{d} + \mathrm{w}(u,v)$
- This is exactly what Bellman-Ford checks for.

SSSP in a DAG

- Recall: a *DAG* is a *d*irected *a*cyclic *g*raph.
- If we update the edges in topologically sorted order, we correctly compute the shortest paths.
- Reason: the only paths to a vertex come from vertices before it in the topological sort.



SSSP in a DAG Theorem

- **Theorem:** For any vertex *u* in a DAG, if all the vertices before *u* in a topological sort of the DAG have been updated, then $u.\mathrm{d} = \delta(s,u)$.
- **Proof:** By induction on the position of a vertex in the topological sort.
- Base case: $s.\mathrm{d}$ is initialized to 0.
- Inductive case: Assume all vertices before *u* have been updated, and for all such vertices *v*, $v.\mathrm{d}=\delta(s,v)$. (continued)

Proof, Continued

- Some edge (v,u) where v is before u, must be on the shortest path to u, since there are no other paths to u.
- When v was updated, we set u.d to

  $$v.\mathrm{d}+\mathrm{w}(v,u)$$

  $$= \delta(s,v) + \mathrm{w}(v,u)$$

  $$= \delta(s,u) \blacksquare$$

SSSP-DAG Algorithm

	DAG-SHORTEST-PATHS(G,w,s)
$\boxed{\Theta(V+E)}$	topologically sort the vertices of G
$\boxed{\Theta(V)}$	initialize d and π as in previous algorithms
$\boxed{\Theta(E)}$	for each vertex u in topological sort order do
	for each vertex v in Adj[u] do
	RELAX(u, v, w)

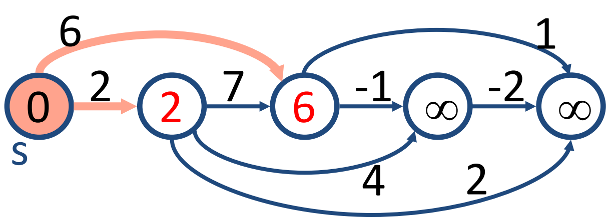
Running time: $\theta(V+E)$, same as topological sort

Algorithm for directed acyclic graphs (DAG)

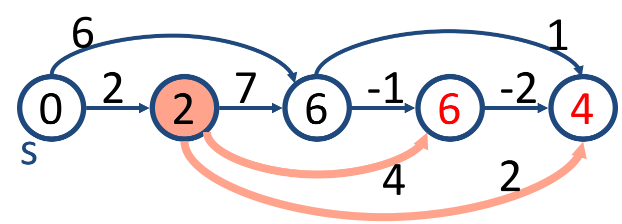
Single-Source Shortest Paths

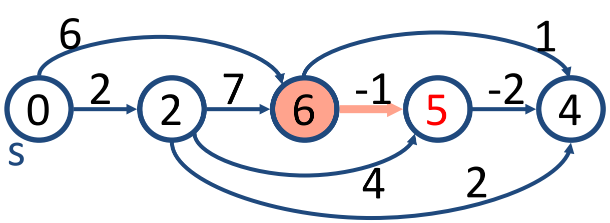
DAG-Shortest-Path

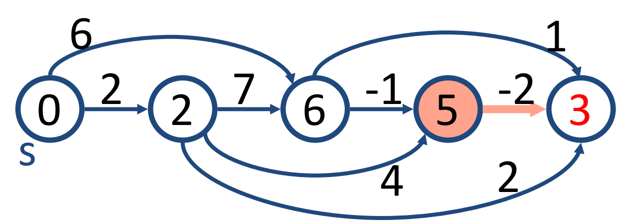
Method: By topological sorting the vertices first, then relax the edges of the sorted vertices one by one.

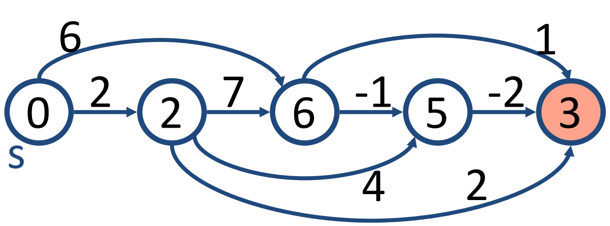


3



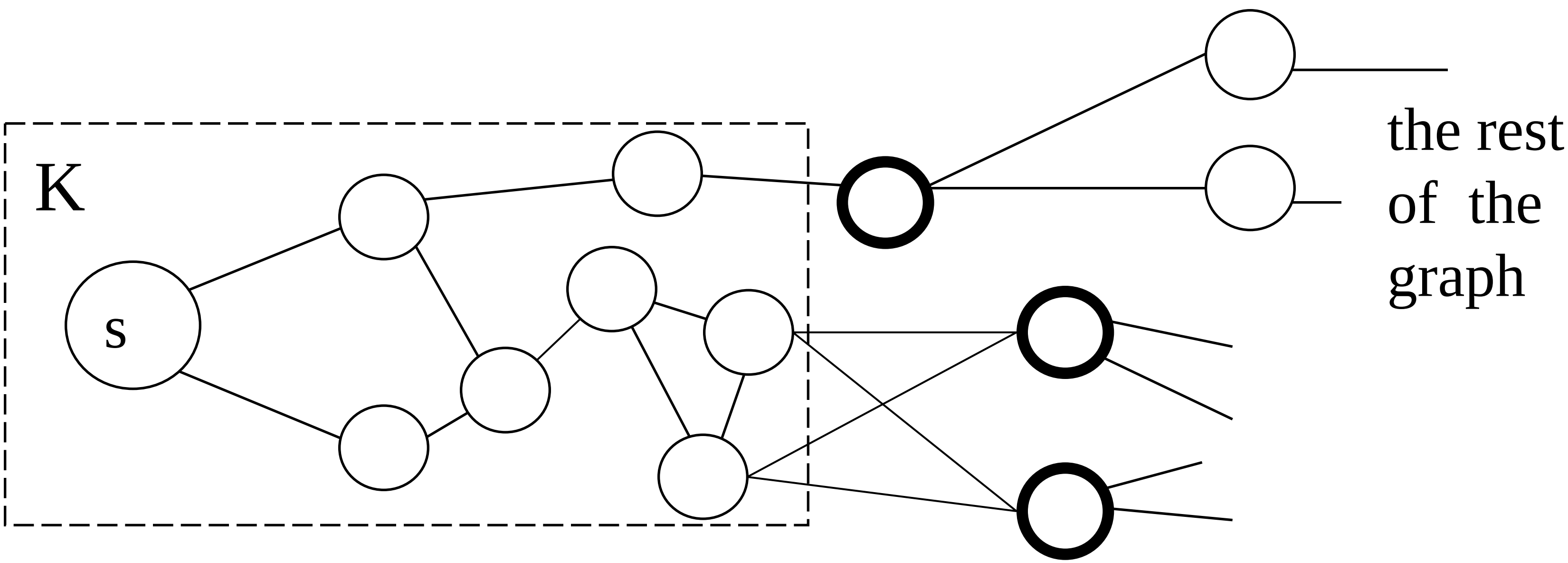






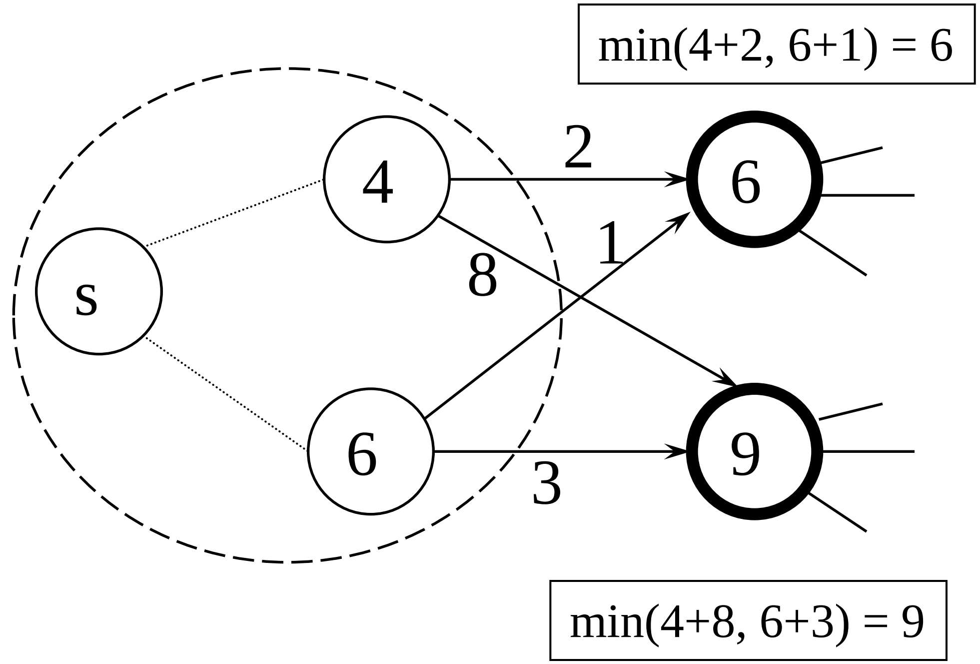
Dijkstra's Algorithm

- Assume that all edge weights are ≥ 0.
- Idea: say we have a set *K* containing all vertices whose shortest paths from s are known (i.e. $u.d = d(s,u)$ for all *u* in *K*).
- Now look at the "frontier" of *K*—all vertices adjacent to a vertex in *K*.



Dijkstra's: Theorem

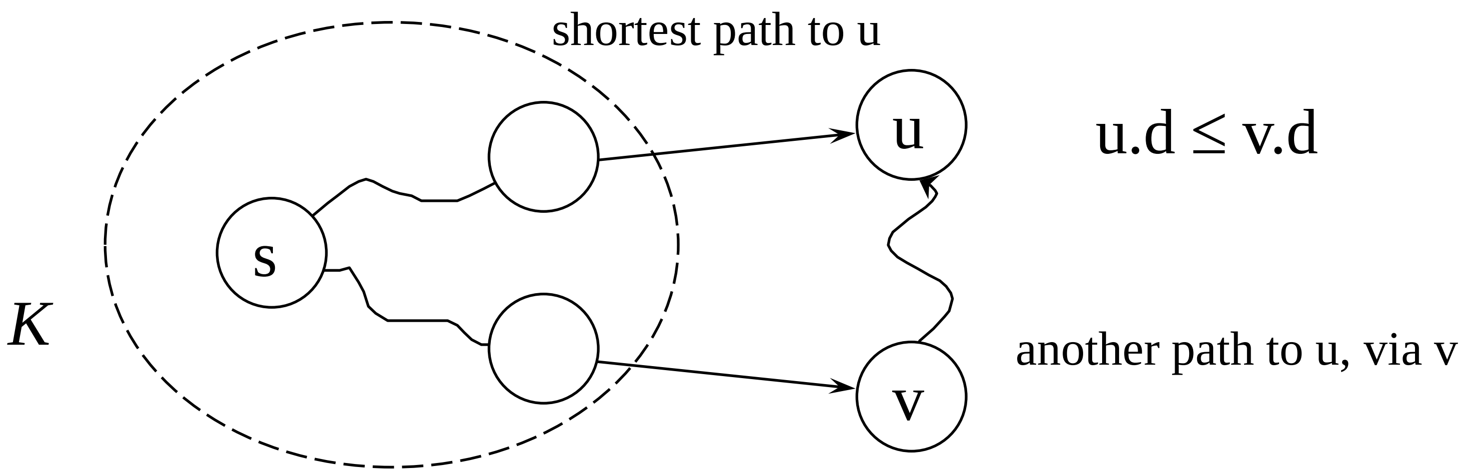
- At each frontier vertex *u*, update *u.d* to be the minimum from all edges from *K.*
- Now pick the frontier vertex *u* with the smallest value of *u.d*.
- Claim: $u.d = \delta(s,u)$



Dijkstra's: Proof

- By construction, *u*.d is the length of the shortest path to *u* going through only vertices in *K*.
- Another path to *u* must leave *K* and go to *v* on the frontier.
- But the length of this path is at least *v*.d, (assuming non-negative edge weights), which is $\geq$ *u*.d. ■

Proof Explained



- Why is the path through *v* at least *v*.d in length?
- We know the shortest paths to every vertex in *K*.
- We've set *v*.d to the shortest distance from *s* to *v* via *K*.
- The additional edges from *v* to *u* cannot decrease the path length.

Dijkstra's Algorithm, Rough Draft

$K \leftarrow \{s\}$

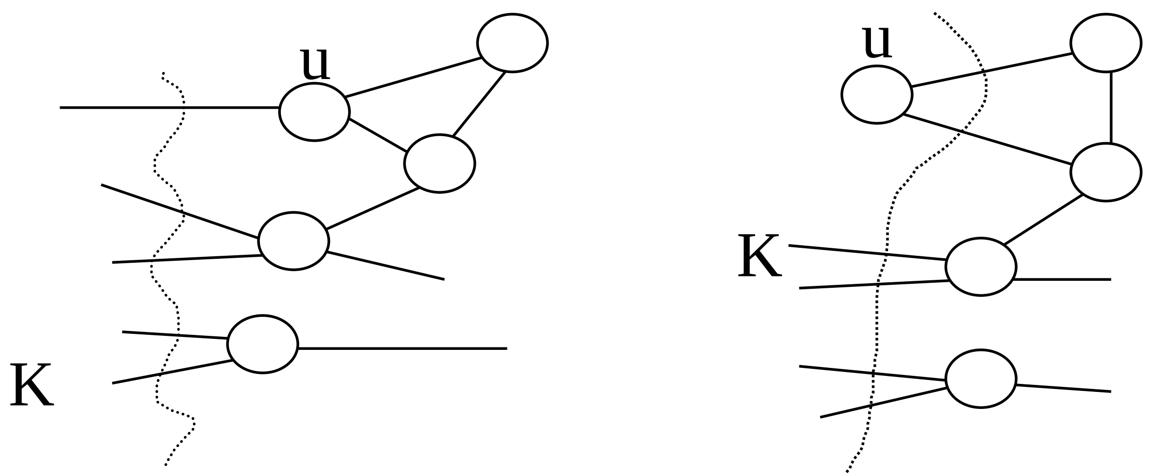
Update d for frontier of K

$u \leftarrow$ vertex with minimum d on frontier

$\triangleright$ we now know $u.d = \delta(s,u)$

$K \leftarrow K \bigcup \{u\}$

repeat until all vertices are in K.



A Refinement

- Note: we don't really need to keep track of the frontier.
- When we add a new vertex u to K, just update vertices adjacent to u.

Dijkstra's Algorithm

```
DIJKSTRA(G, w, s) ▹ Graph, weights, start vertex
      for each vertex v in V[G] do
          v.d ← ∞
          v.π ← NIL
     s.d ← 0
     Q ← BUILD-PRIORITY-QUEUE(V[G])
      ▹ Q is V[G] - K
     while Q is not empty do
          u = EXTRACT-MIN(Q)
          for each vertex v in Adj[u]
               RELAX(u, v, w)     // DECREASE_KEY
```

Running Time of Dijkstra

- Initialization: $\theta(V)$
- Building priority queue: $\theta(V)$
- “while” loop done |V| times
- |V| calls of EXTRACT-MIN
- Inner “edge” loop done |E| times
- At most |E| calls of DECREASE-KEY
- Total time:

$$\Theta(V + V \times T_{\text{EXTRACT-MIN}} + E \times T_{\text{DECREASE-KEY}})$$

Dijkstra Running Time (cont.)

$\Theta(V + V \times T_{\text{EXTRACT-MIN}} + E \times T_{\text{DECREASE-KEY}})$

- 1. Priority queue is an array.
  EXTRACT-MIN in $\Theta(n)$ time, DECREASE-KEY in $\Theta(1)$
  Total time: $\Theta(V + VV + E) = \Theta(V^2)$
- 2. ("Modified Dijkstra")
  Priority queue is a binary (standard) heap.
  EXTRACT-MIN in $\Theta(\lg n)$ time, also DECREASE-KEY
  Total time: $\Theta(V\lg V + E\lg V)$
- 3. Priority queue is Fibonacci heap. (Of theoretical interest only.)
  EXTRACT-MIN in $\Theta(\lg n)$,
  DECREASE-KEY in $\Theta(1)$ (amortized)
  Total time: $\Theta(V\lg V+E)$

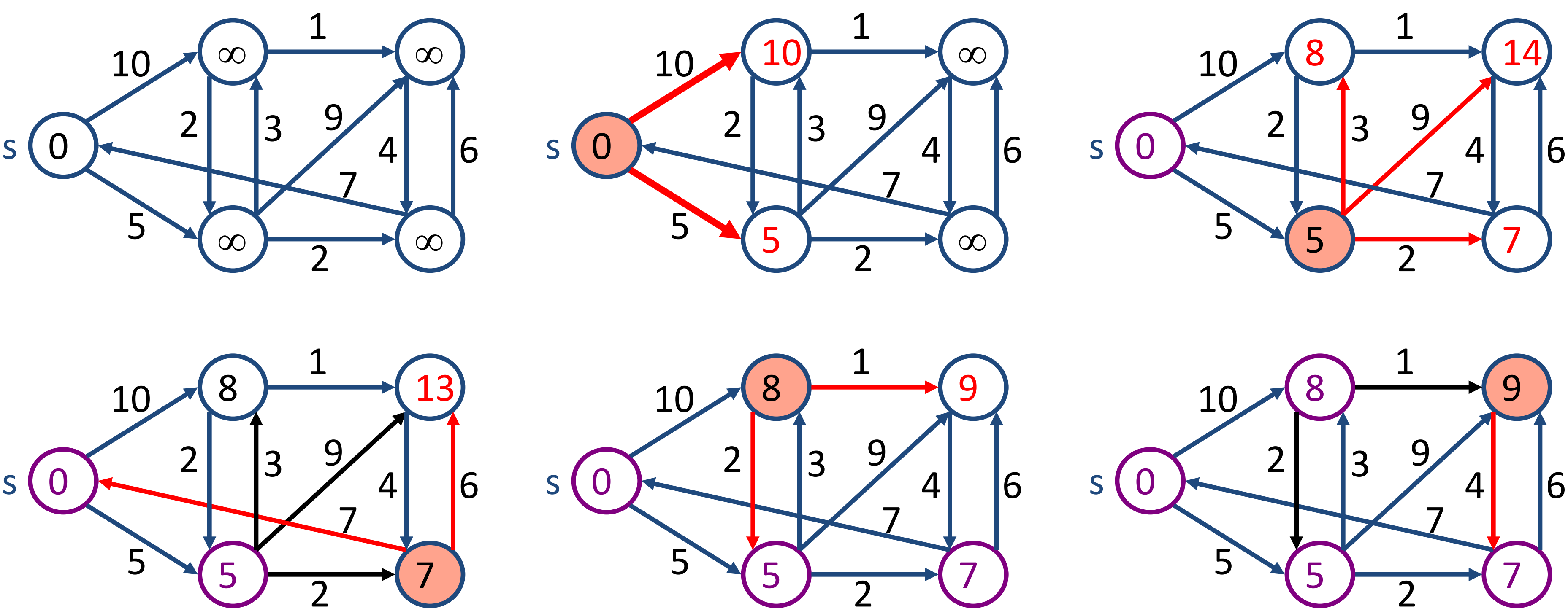
Dijkstra’s Algorithm Example

Single-Source Shortest Paths

Dijkstra’s Algorithm

Handle non-negative edges only.

Method: Grow the solution by checking vertices one by one, starting from the one nearest to the source vertex.



Reading Assignments

- Reading assignment for next class:
  - Chapter 25.1-25.2

- Announcement: Exam 1 is on Tues, Feb. 18
  - Will cover everything up through dynamic programming