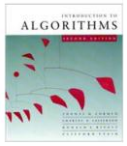
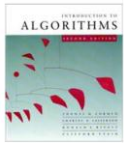


Graph Algorithms



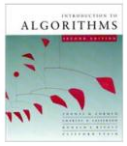
Graph representation

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

- May be either directed or undirected.
- Two common ways to represent for algorithms:

1. Adjacency lists.
2. Adjacency matrix.

When expressing the running time of an algorithm, it's often in terms of both $|V|$ and $|E|$.
Example: $O(V + E)$.



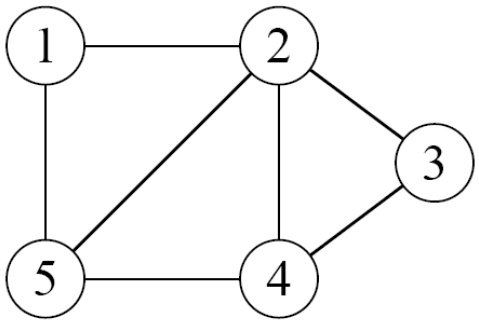
Graph representation

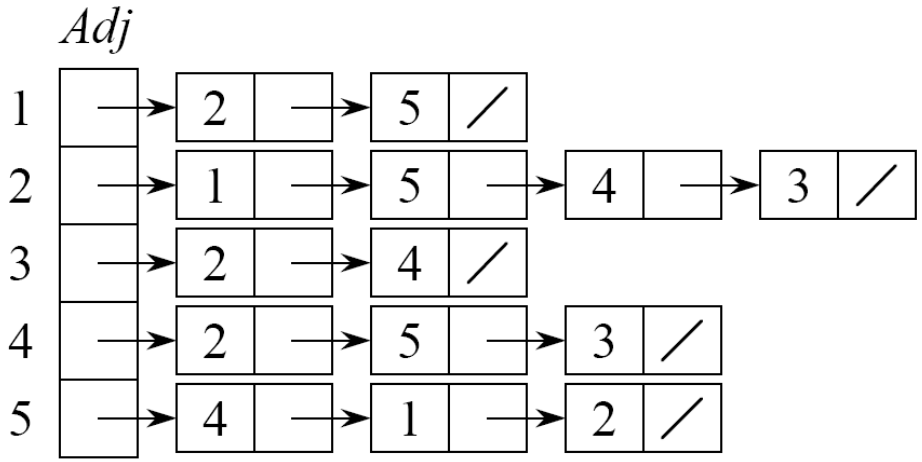
Adjacency lists

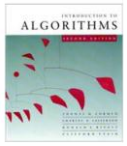
Array *Adj* of $|V|$ lists, one per vertex.

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

Example: For an undirected graph:







Graph representation

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

Weight: $w : E \to \mathbf{R}$

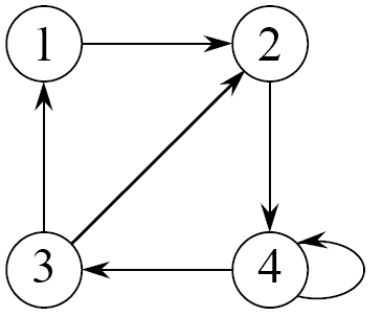
We'll use weights later on for spanning trees and shortest paths.

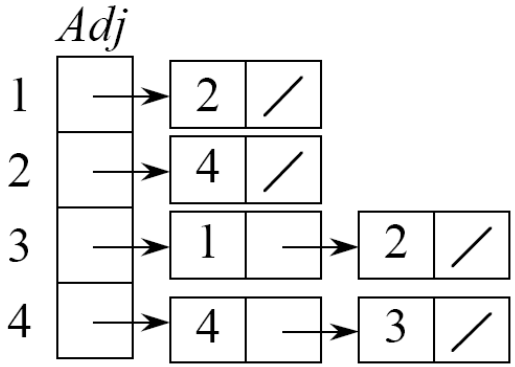
Space: $(V + E)$.

Time: to list all vertices adjacent to u: $\Theta(\text{degree}(u))$.

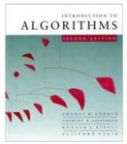
Time: to determine if $(u, v) \in E$: $O(\text{degree}(u))$.

Example: For a directed graph:





Same asymptotic space and time.



Graph representation

Adjacent matrix

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

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

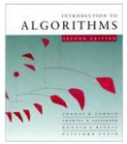
	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

	1	2	3	4
1	0	1	0	0
2	0	0	0	1
3	1	1	0	0
4	0	0	1	1

Space: $\Theta(V^2)$.

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

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



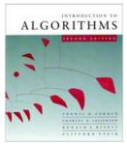
Breadth-first search

BFS is one of the simplest algorithms for searching a graph and the archetype for many important graph algorithms.

- Prim's minimum-spanning tree algorithm
- Shortest-paths algorithm.

BFS expands the frontier between discovered and undiscovered vertices uniformly across the breath of the frontier.

That is, the algorithm discovers all vertices at distance k from s before discovering any vertices at distance k + 1.

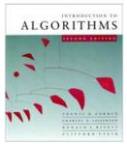


Breadth-first search

Input: Graph $G = (V, E)$, either directed or undirected,
and ***source vertex*** $s \in V$.

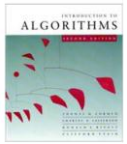
Output: $d[v]$ = distance (smallest # of edges) from s to v,
for all $v \in V$.
$\pi[v] = u$ such that (u, v) is last edge on path from s *to* v.

- u is v's ***predecessor***.
- set of edges $\{(\pi[v], v) : v = s\}$ forms a tree.



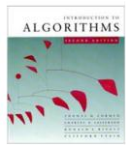
Breadth-first search

- BFS colors each vertex white, gray, or black.
- All vertices start out white and may later become gray and then black.
- A vertex is discovered the first time it is encountered during the search, at which time it becomes nonwhite.
- If $(u, v) \in E$ and vertex u is black, then vertex v is either gray or black; that is, all vertices adjacent to black vertices have been discovered.
- Gray vertices may have some adjacent white vertices; they represent the frontier between discovered and undiscovered vertices.

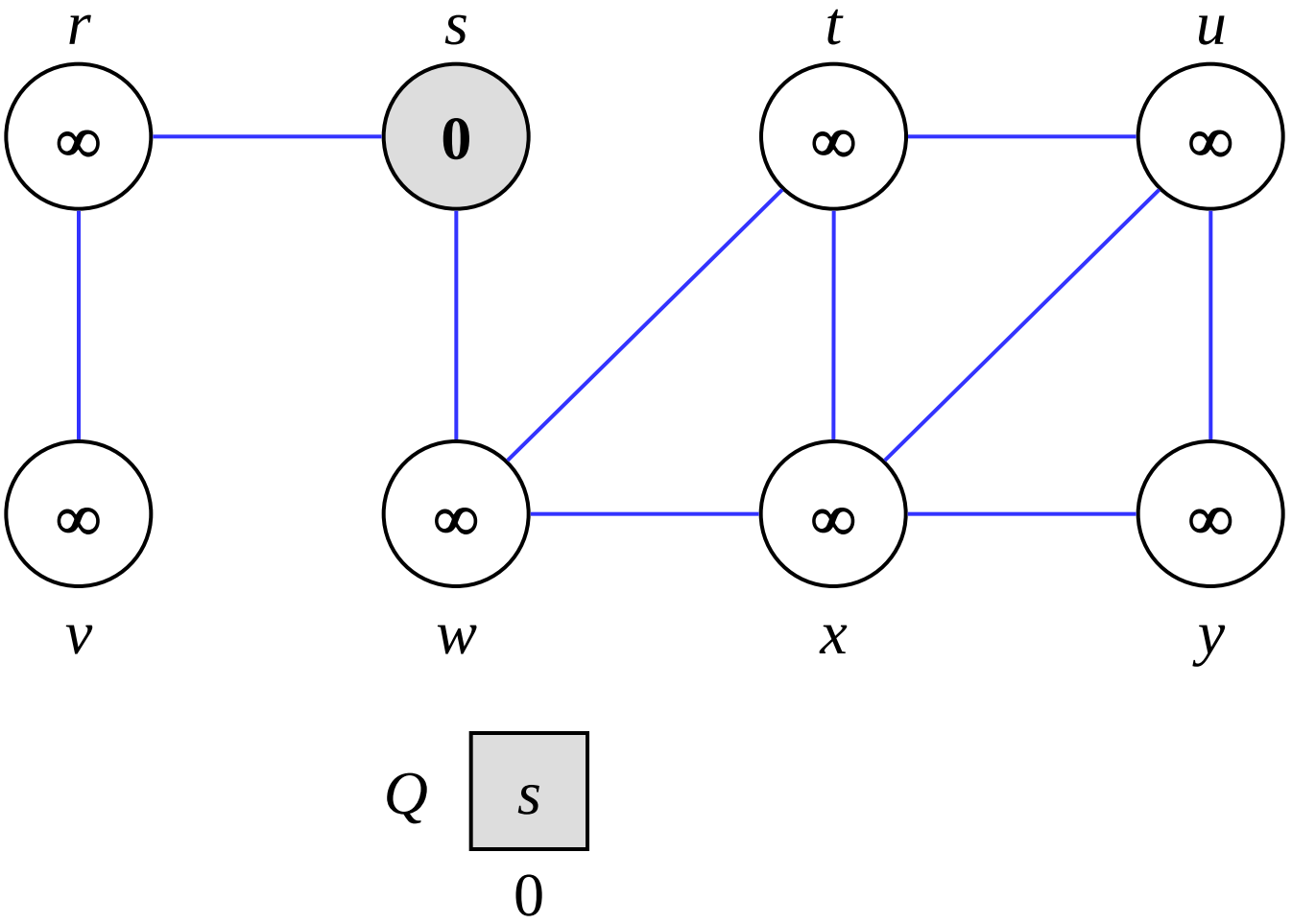


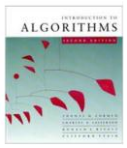
Breadth-first search

```
BFS(G, s)
for each vertex u ∈ V[G] – {s}
    do color[u] ← WHITE
        d[u] ← ∞
        π[u] ← NIL
color[s] ← GRAY
d[s] ← 0
π[s] ← NIL
Q ← Ø
ENQUEUE(Q, s)
while Q ≠ Ø
    do u ← DEQUEUE(Q)
        for each v ∈ Adj[u]
            do if color[v] = WHITE
                then color[v] ← GRAY
                        d[v] ← d[u] + 1
                        π[v] ← u
                        ENQUEUE(Q, v)
        color[u] ← BLACK
```

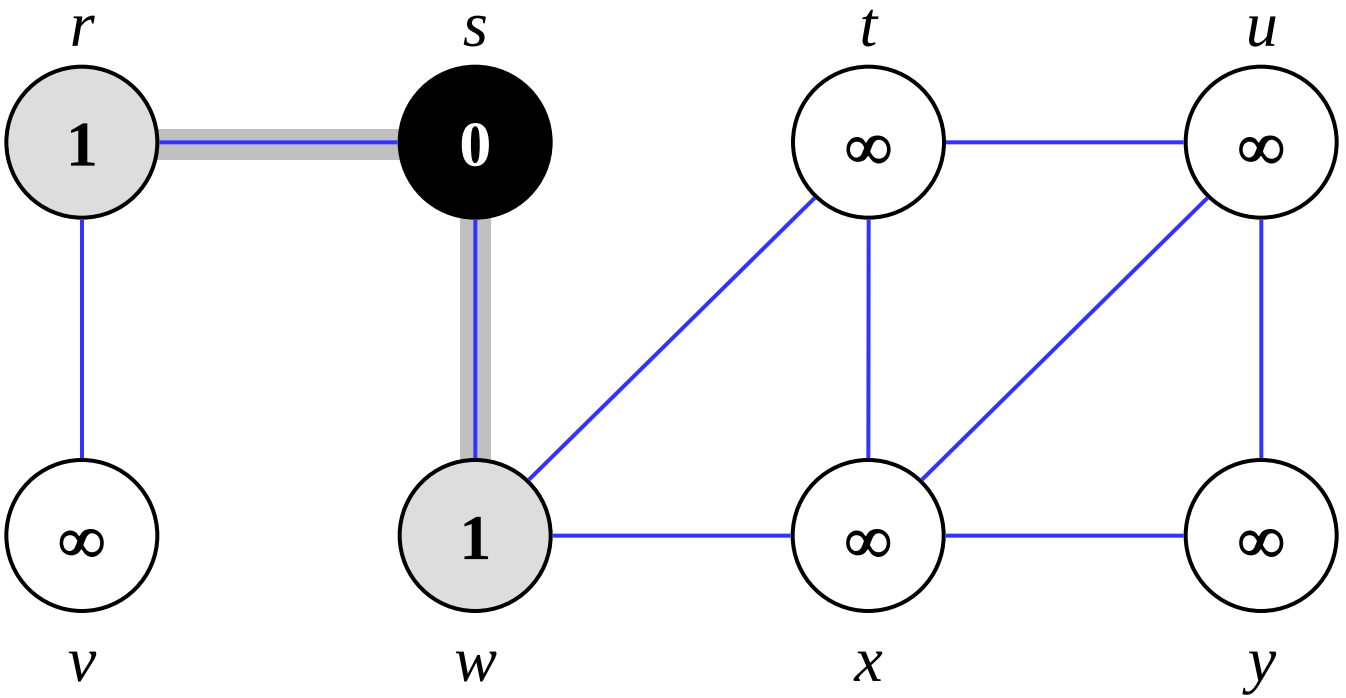


Breadth-first search



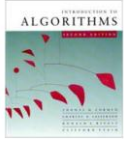


Breadth-first search

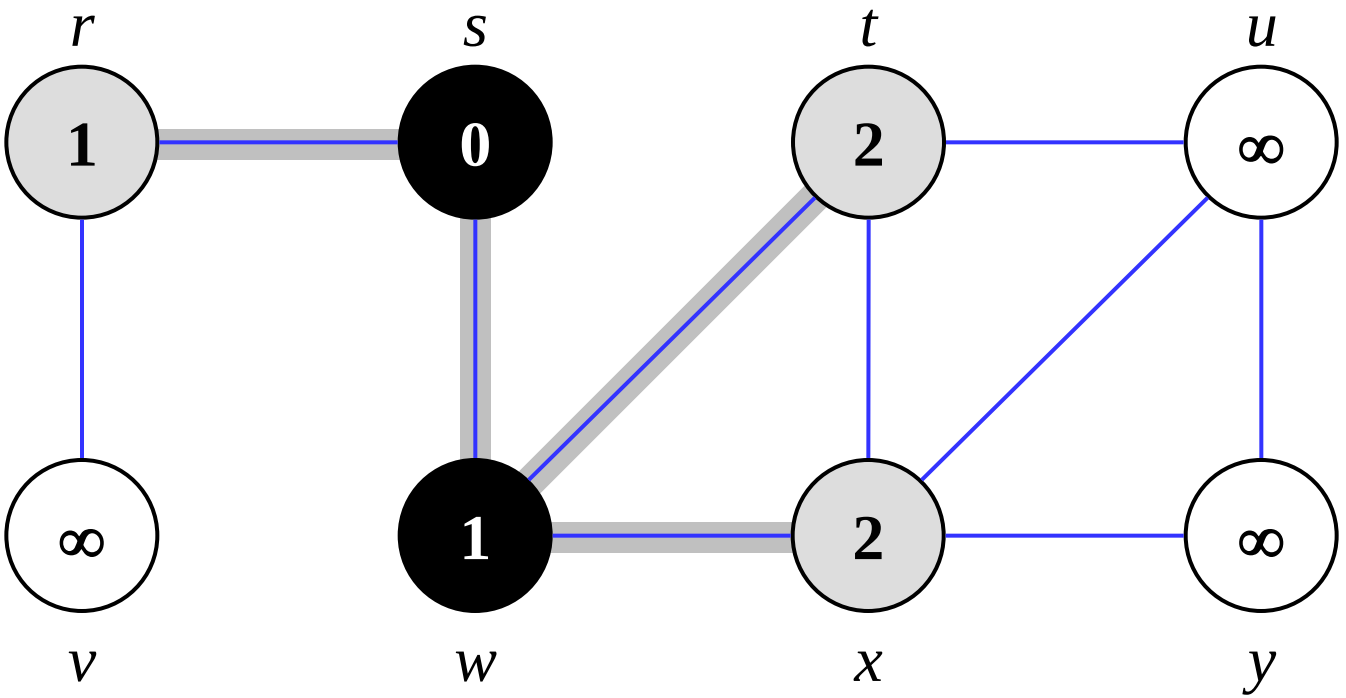


Q | w | r |

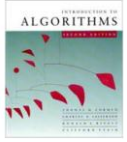
1 1



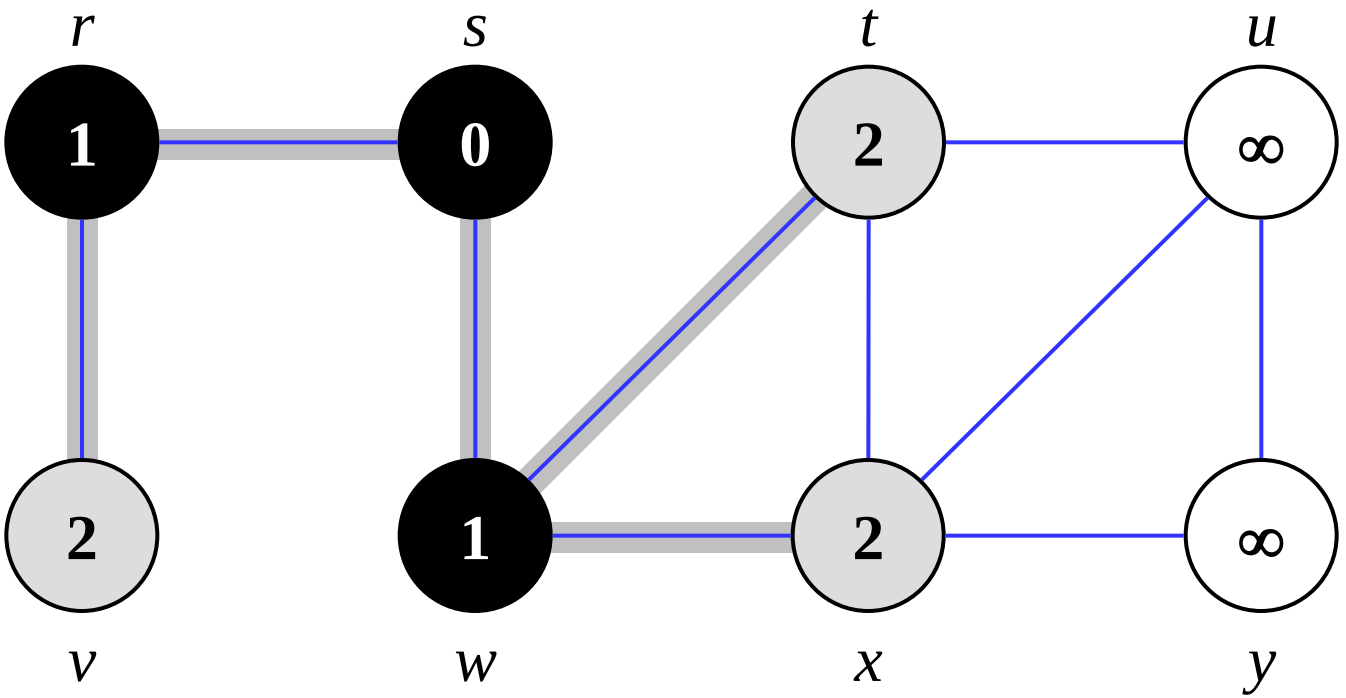
Breadth-first search



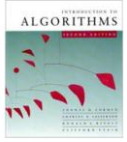
Q | r | t | x |
1 2 2



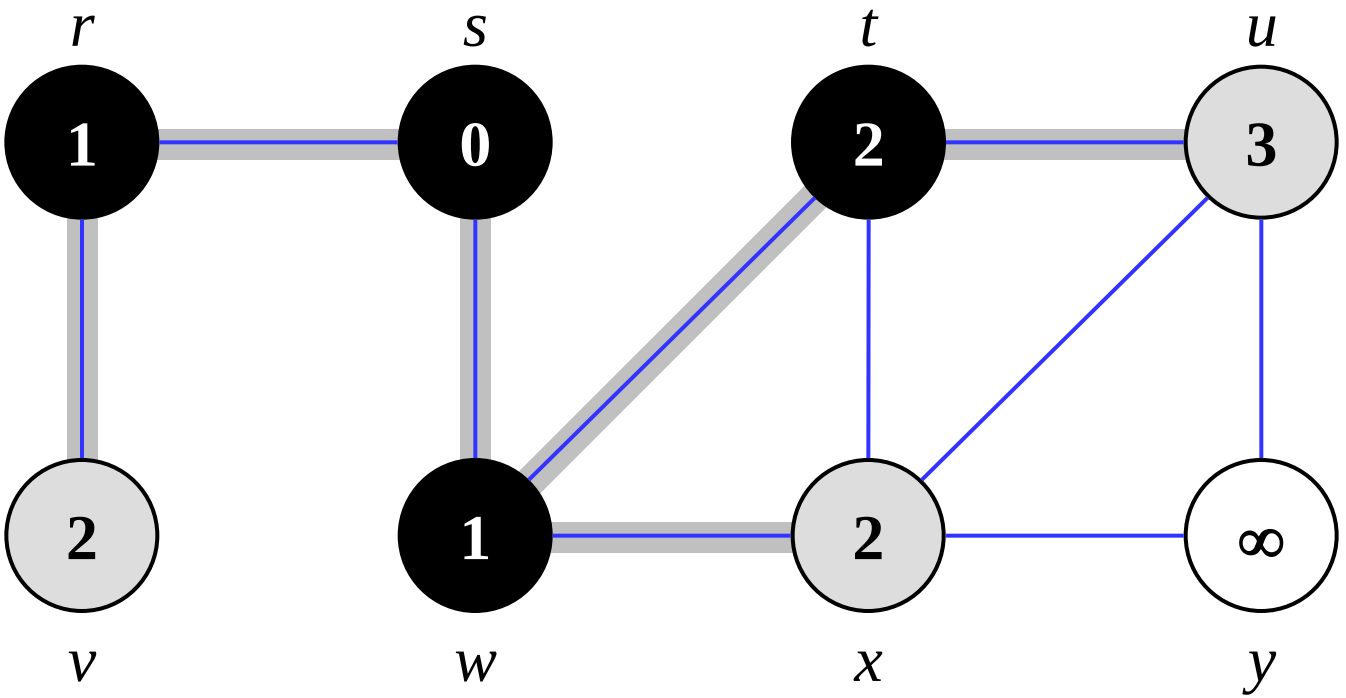
Breadth-first search



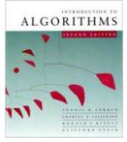
Q | t | x | v |
2 2 2



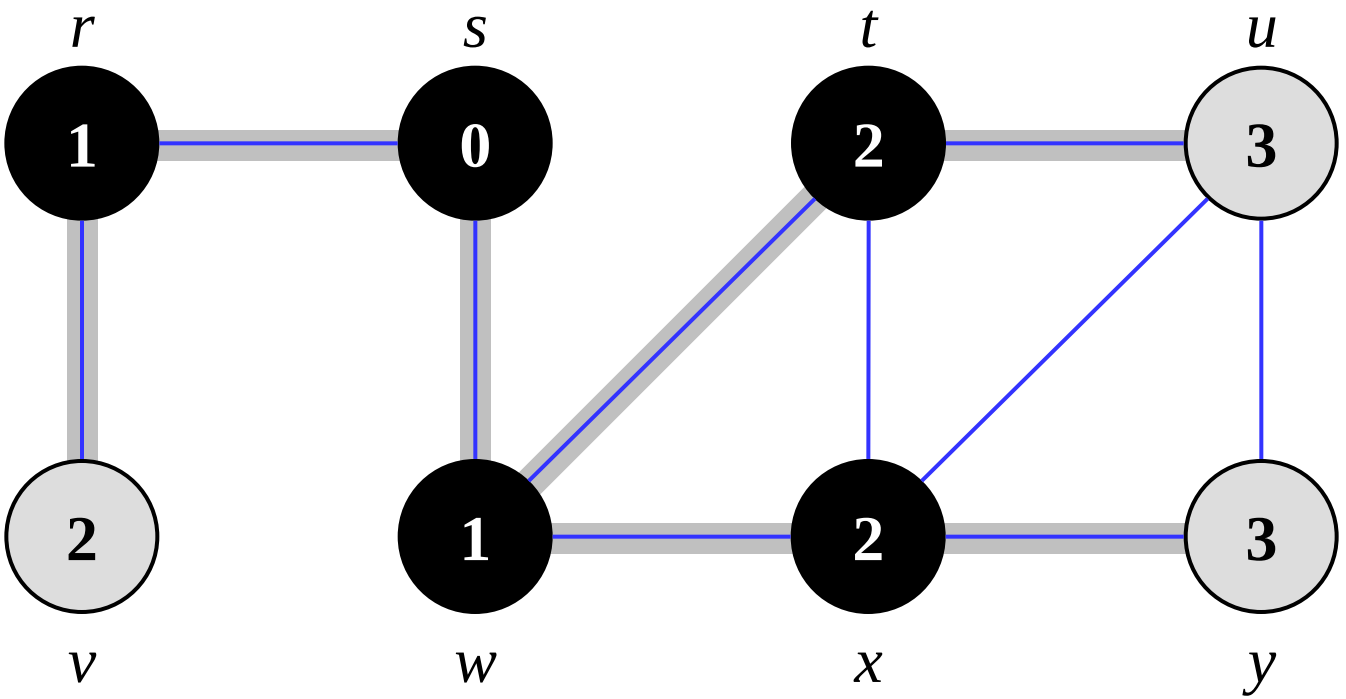
Breadth-first search



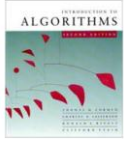
Q | x | v | u |
2 2 3



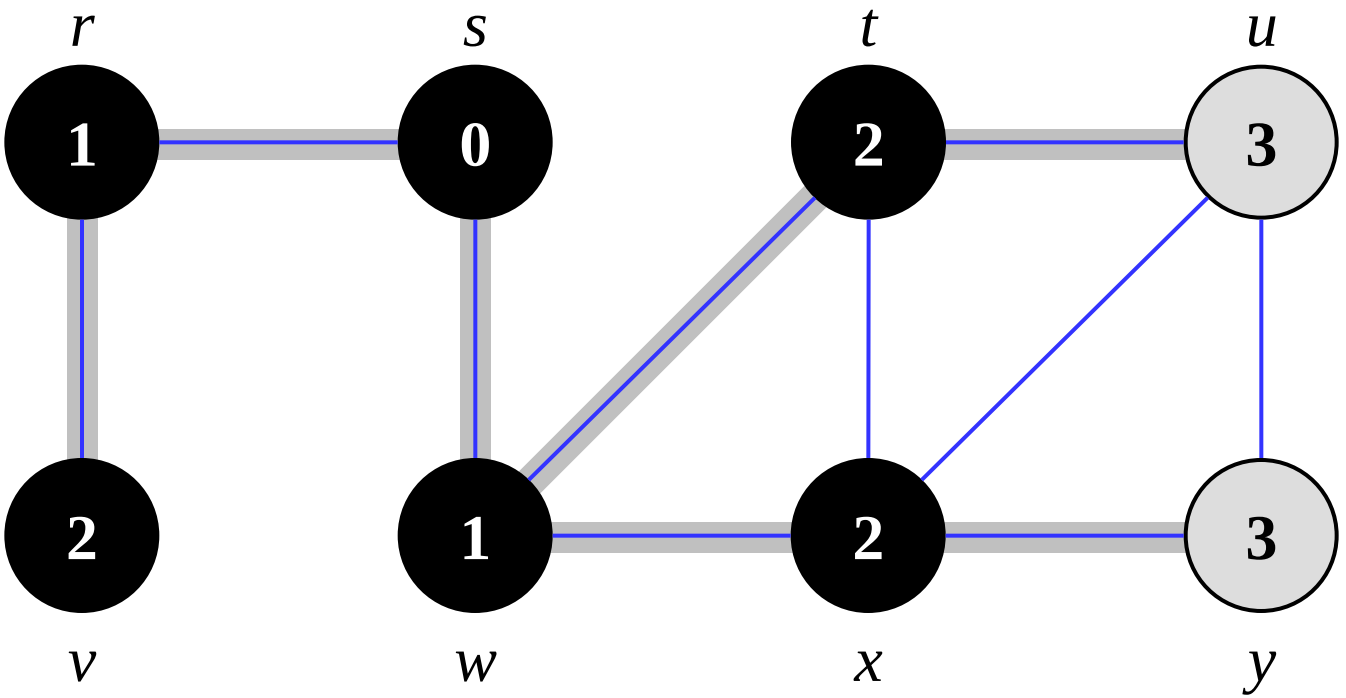
Breadth-first search



Q | v | u | y |
2 3 3

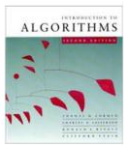


Breadth-first search

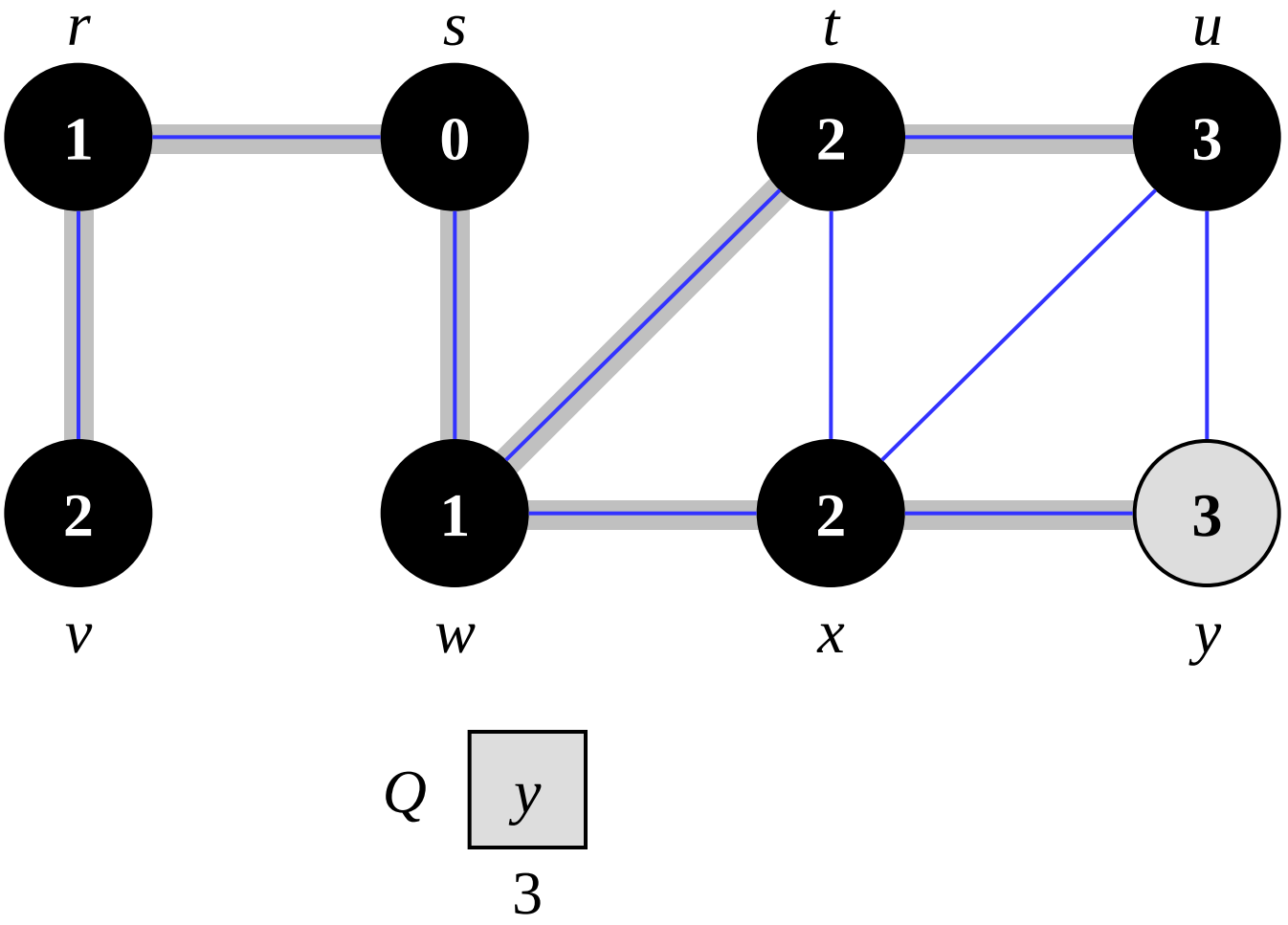


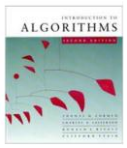
Q | u | y |

3 3

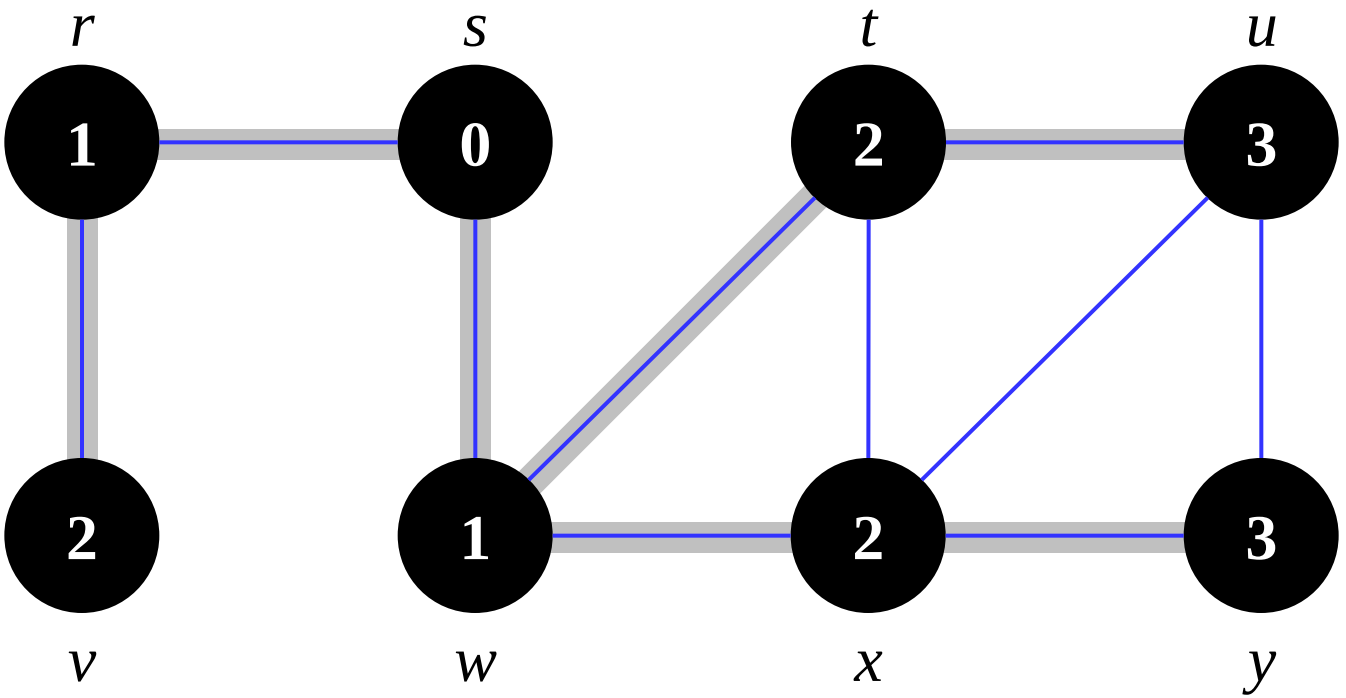


Breadth-first search

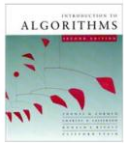




Breadth-first search



$Q \quad \varnothing$



Breadth-first search

Q consists of vertices with d values.

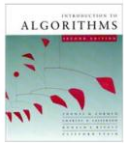
$i\ i\ i\ \ldots\ i\ \ i{+}1\ \ i{+}1\ \ldots\ i{+}1$

- Only 1 or 2 values.
- If 2, differ by 1 and all smallest are first.

Each vertex gets a finite d value monotonically increasing over time.

Time = $O(V + E)$.

- $O(V)$ because every vertex enqueued at most once (since only white vertices are enqueued).
- $O(E)$ because every vertex dequeued at most once and we examine (u, v) only when u is dequeued. Therefore, every edge examined at most once if directed, at most twice if undirected.



Depth-first search

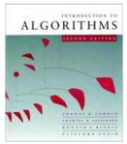
Input: $G = (V, E)$, directed or undirected.

Output: 2 ***timestamps*** on each vertex:
. $d[v]$ = ***discovery time***
. $f[v]$ = ***finishing time***

These will be useful for other algorithms later on.
Can also compute $\pi[v]$.

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.



Depth-first search

As DFS progresses, every vertex has a ***color***:

. WHITE = undiscovered

. GRAY = discovered, but not finished (not done exploring from it)

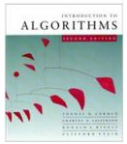
. BLACK = finished (have found everything reachable from it)

Discovery and finish times:

. Unique integers from 1 to $2|V|$. (start and finish time for each vertex)

. For all v, $d[v] < f[v]$.

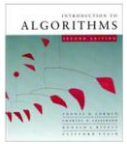
In other words, $1 \le d[v] < f[v] \le 2|V|$.



Depth-first search

```
DFS(V, E)
for each u ∈ V
    do color[u] ← WHITE
time ← 0
for each u ∈ V
    do if color[u] = WHITE
        then DFS-VISIT(u)

DFS-VISIT(u)
color[u] ← GRAY          ▷ discover u
time ← time + 1
d[u] ← time
for each v ∈ Adj[u]      ▷ explore (u, v)
    do if color[v] = WHITE
        then DFS-VISIT(v)
color[u] ← BLACK
time ← time + 1
f[u] ← time              ▷ finish u
```



Depth-first search

Classification of edges

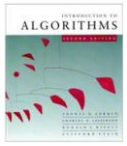
1. Tree edges are edges in the depth-first forest.
   Edge *(u, v)* is a tree edge if *v* was first discovered by exploring edge *(u, v)*.

2. Back edge are those edges *(u, v)* connecting a vertex *u* to an ancestor *v* in depth-first tree.
   Self-loops, which may occur in directed graphs, are considered to be back edges.

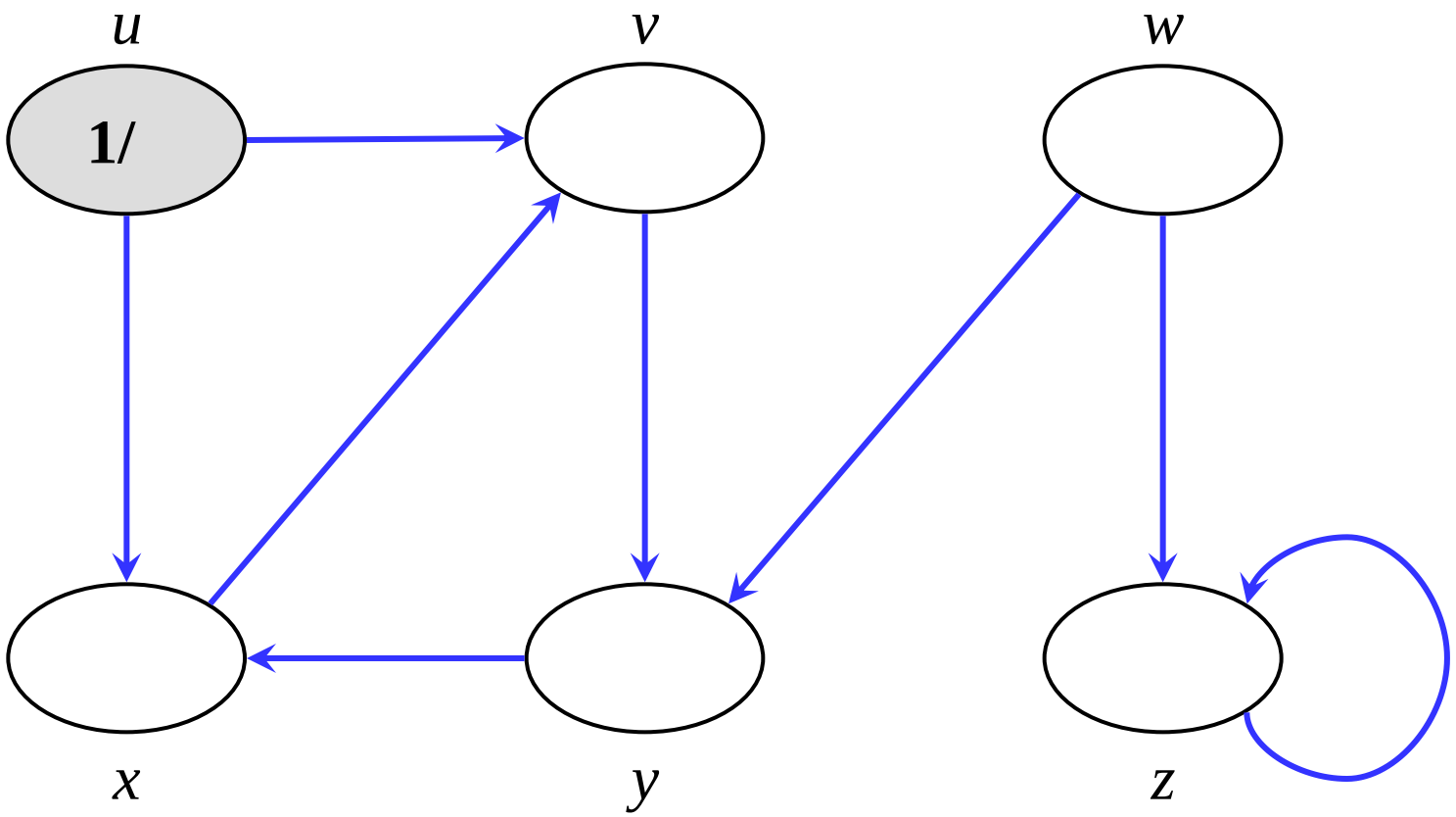
3. Forward edges are those edges *(u, v)*, where *v* is a descendant of *u*, but not a tree edge.

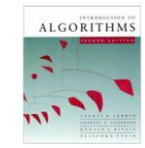
4. Cross edges are all other edges. They can go between vertices in the same depth-first tree, as long as one vertex is not an ancestor of the other, or they can go between vertices in different depth-first trees.

◦ Only tree edges form the edges in the depth-first trees, or depth-first forest.

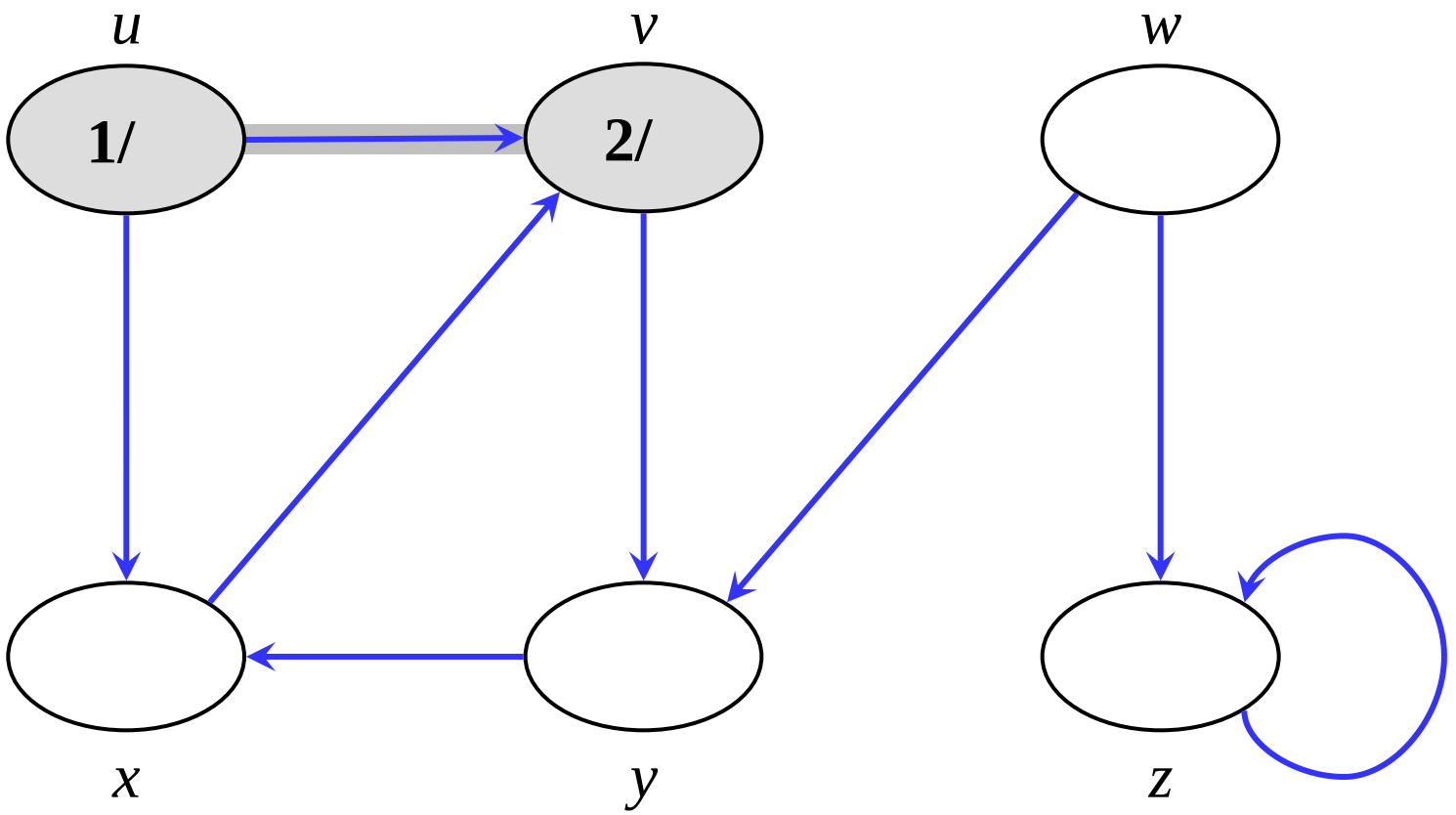


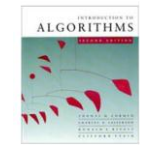
Depth-first search



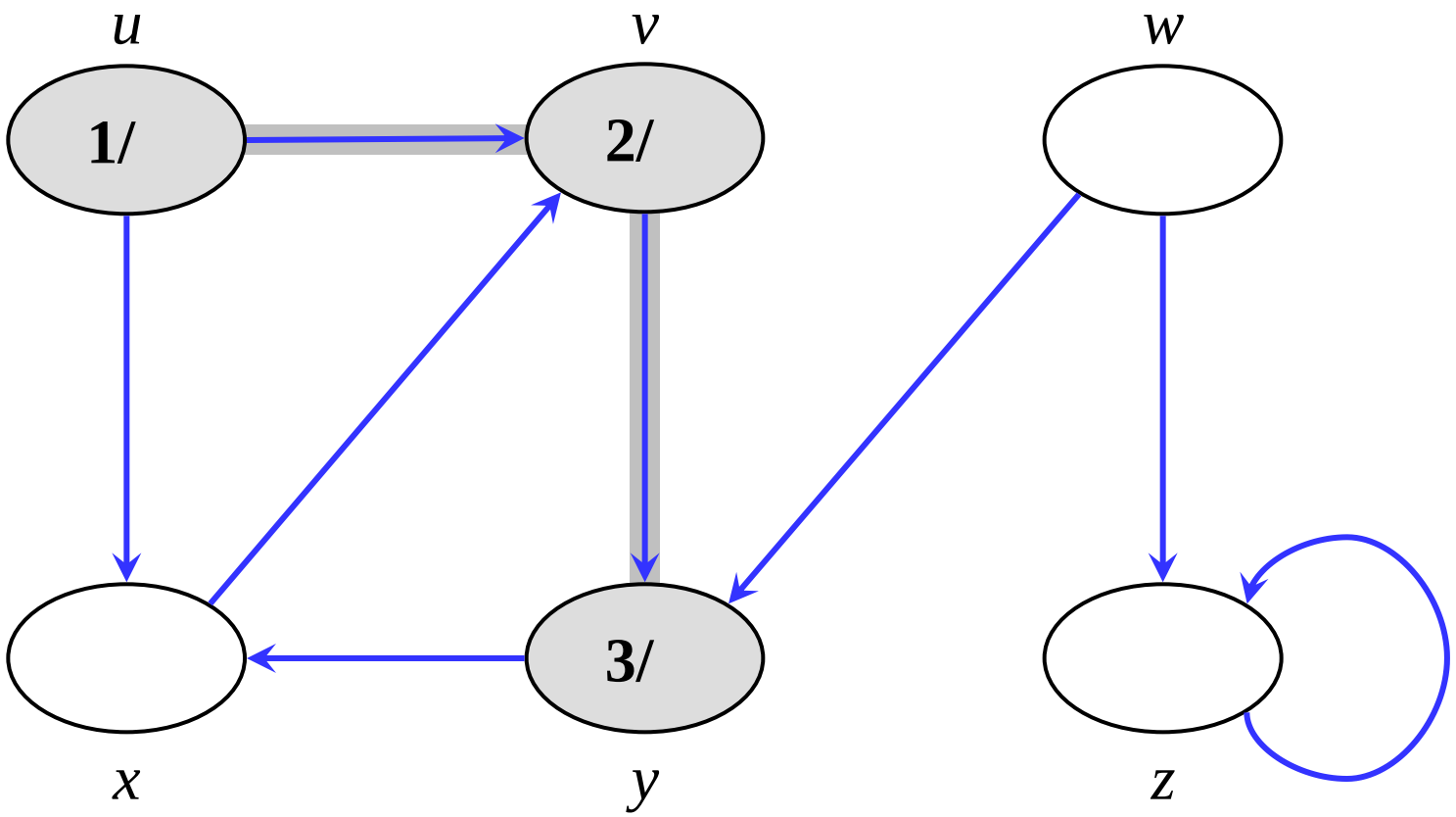


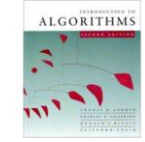
Depth-first search



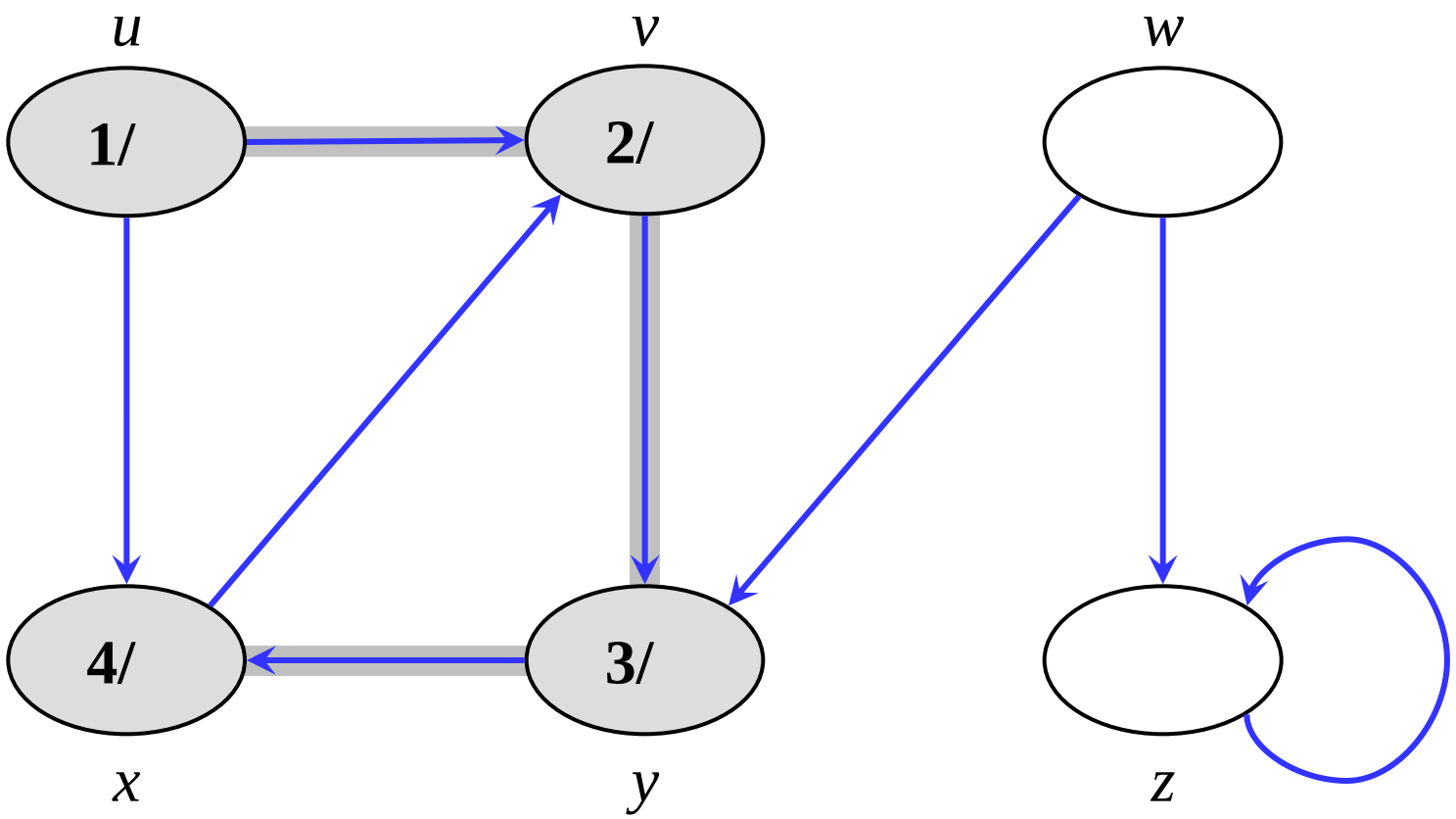


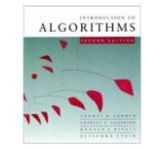
Depth-first search



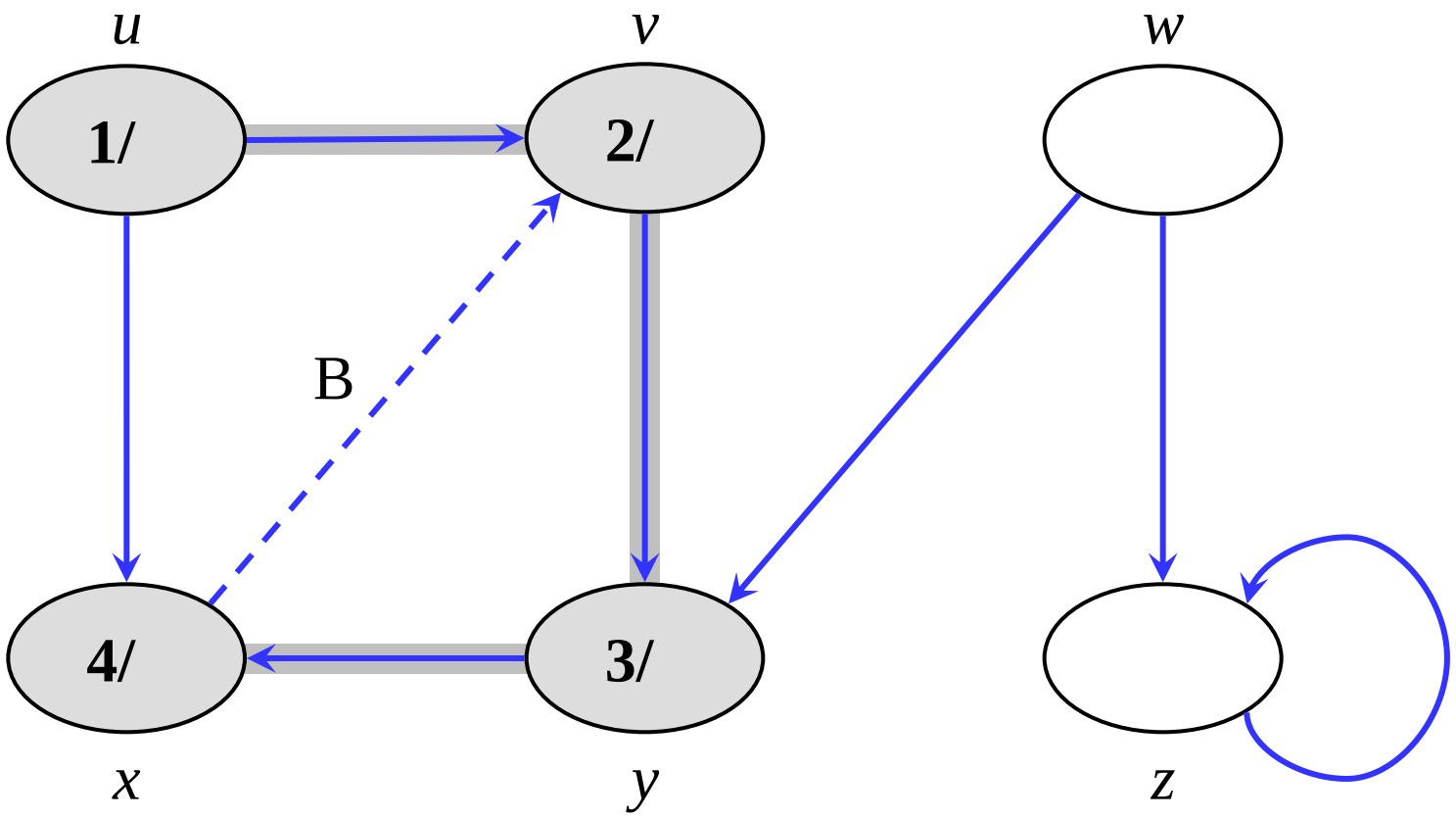


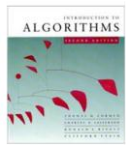
Depth-first search



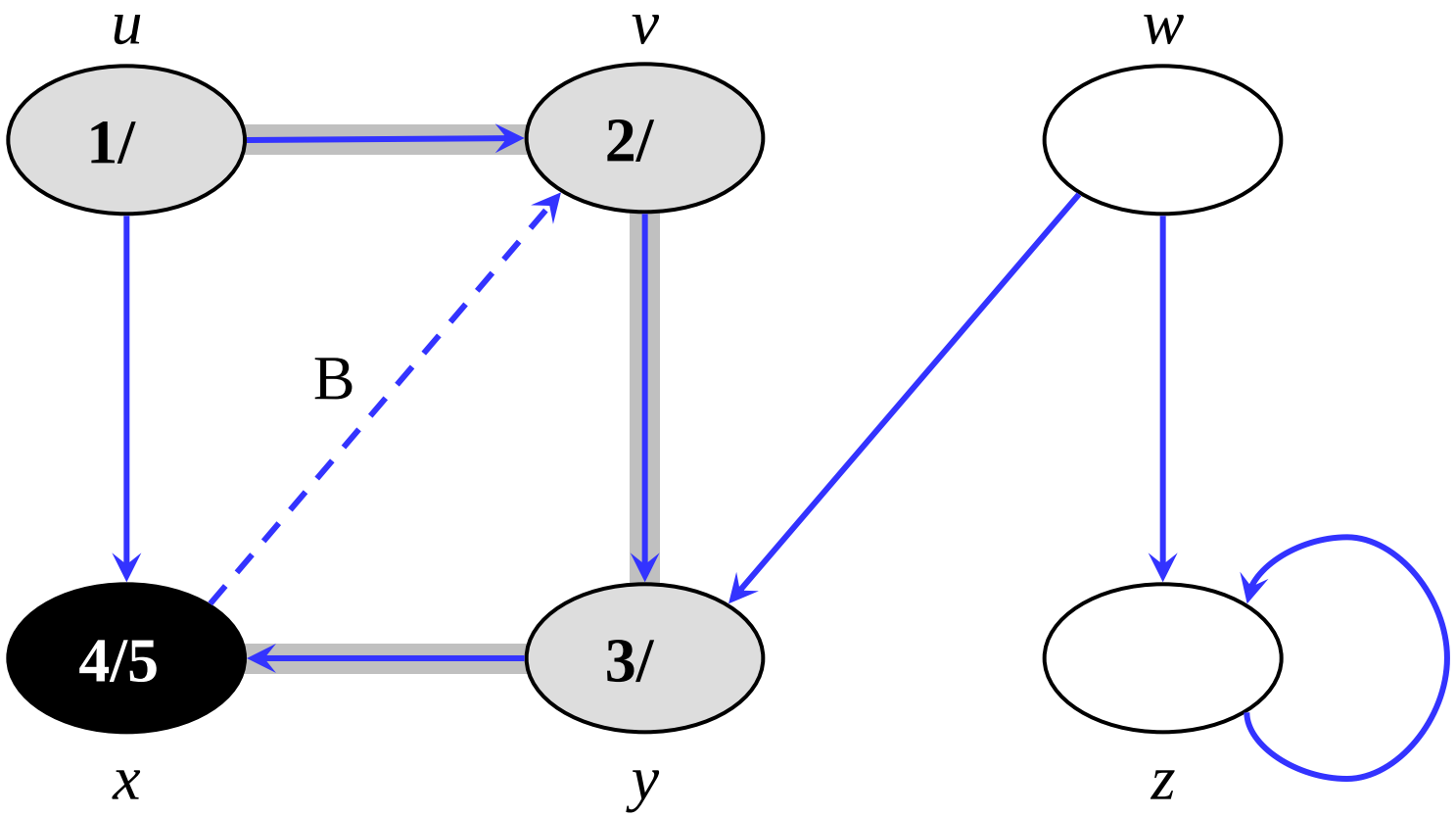


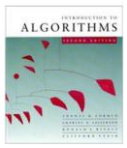
Depth-first search



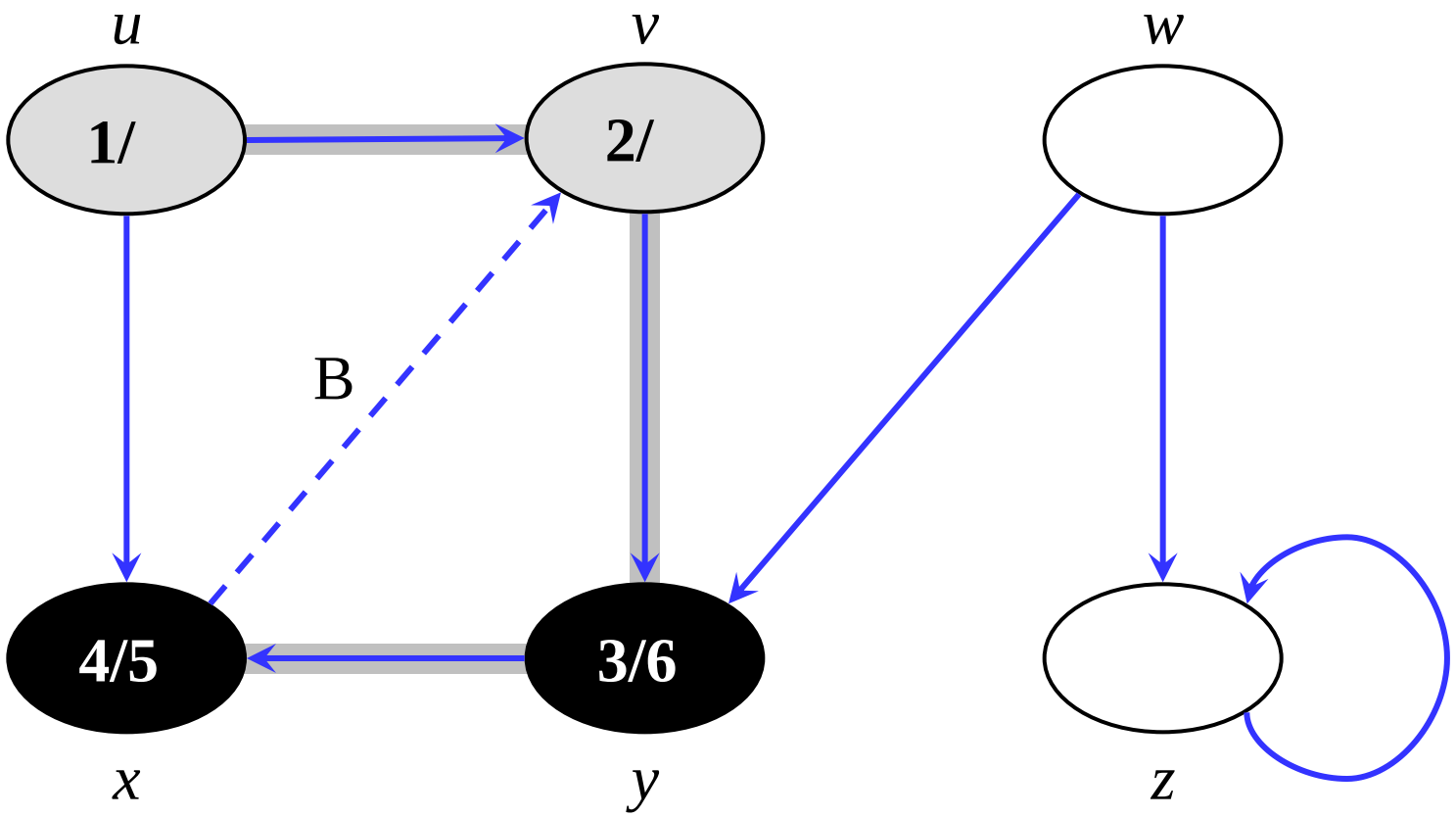


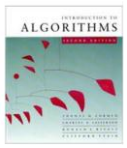
Depth-first search



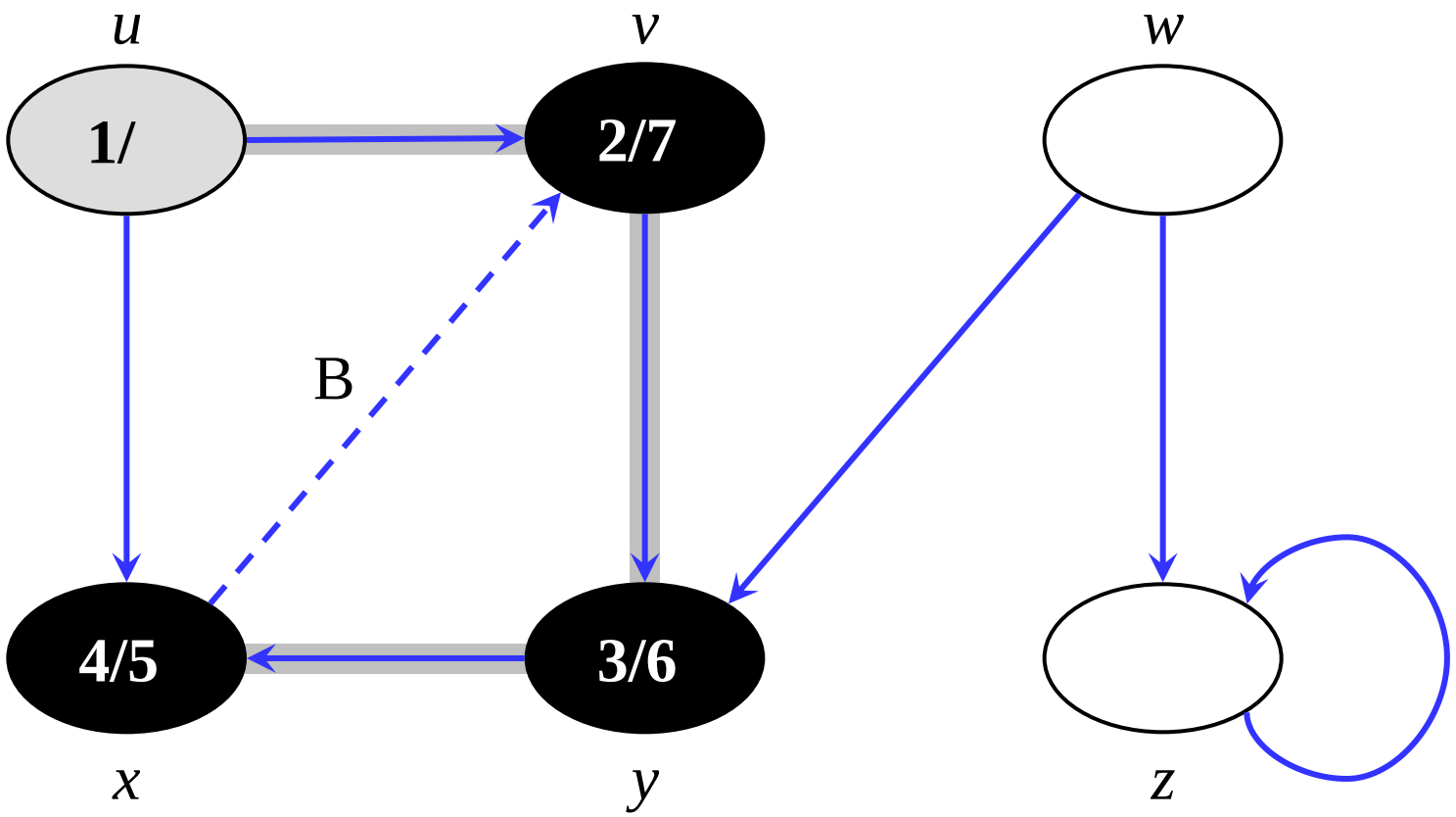


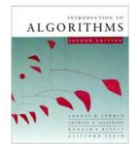
Depth-first search



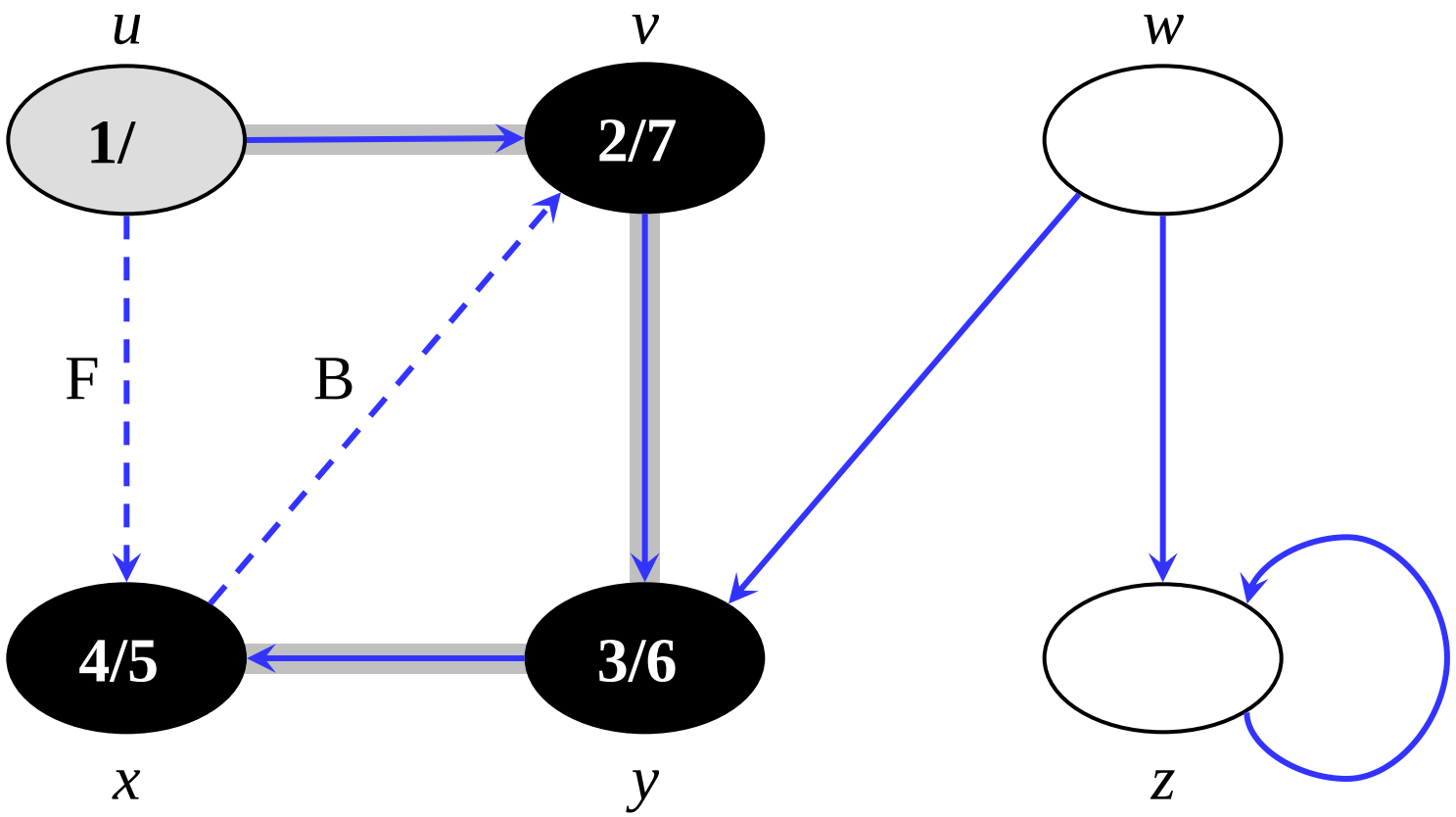


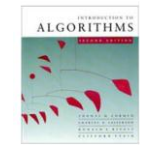
Depth-first search



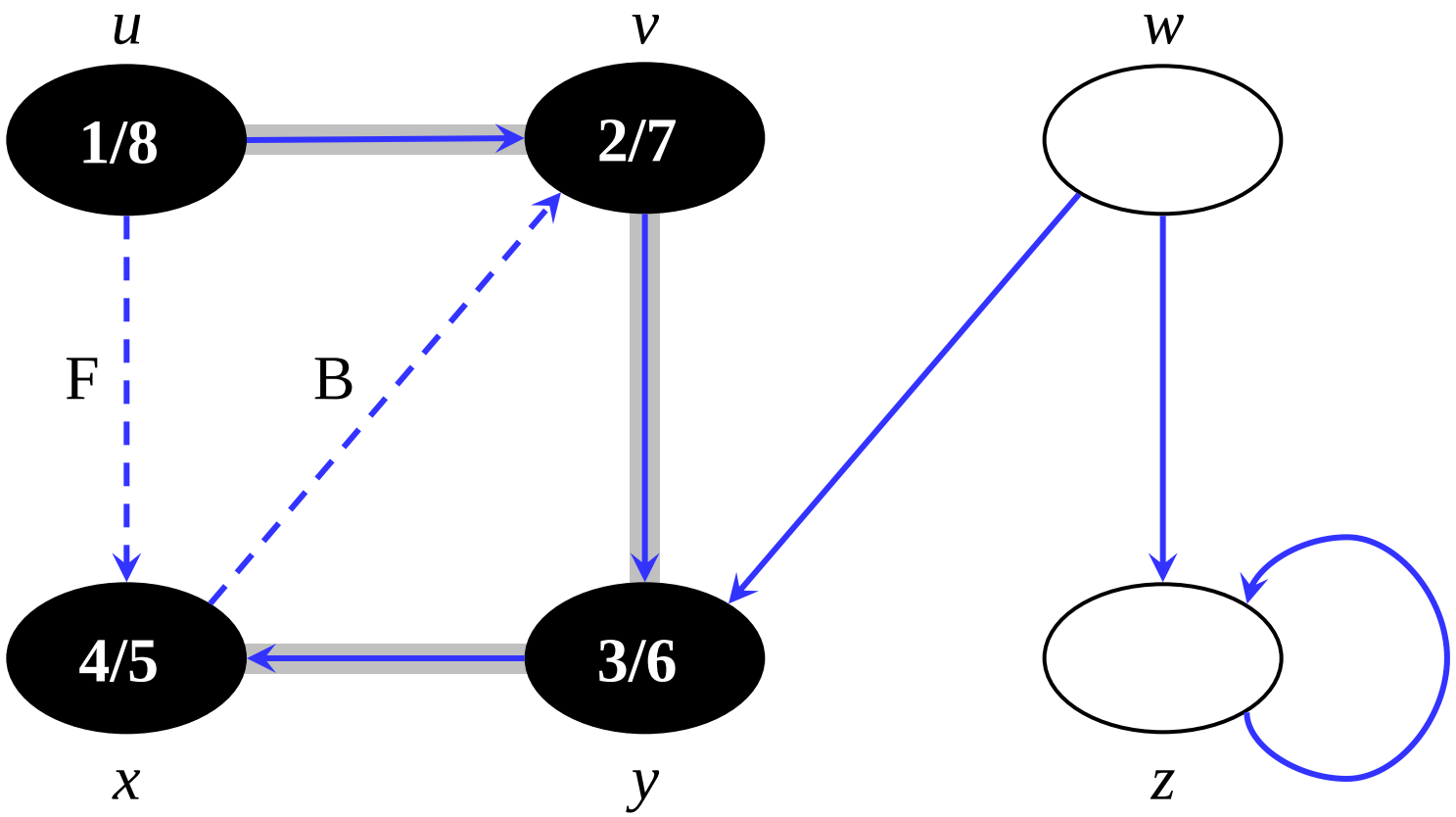


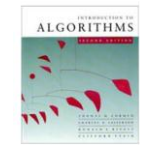
Depth-first search



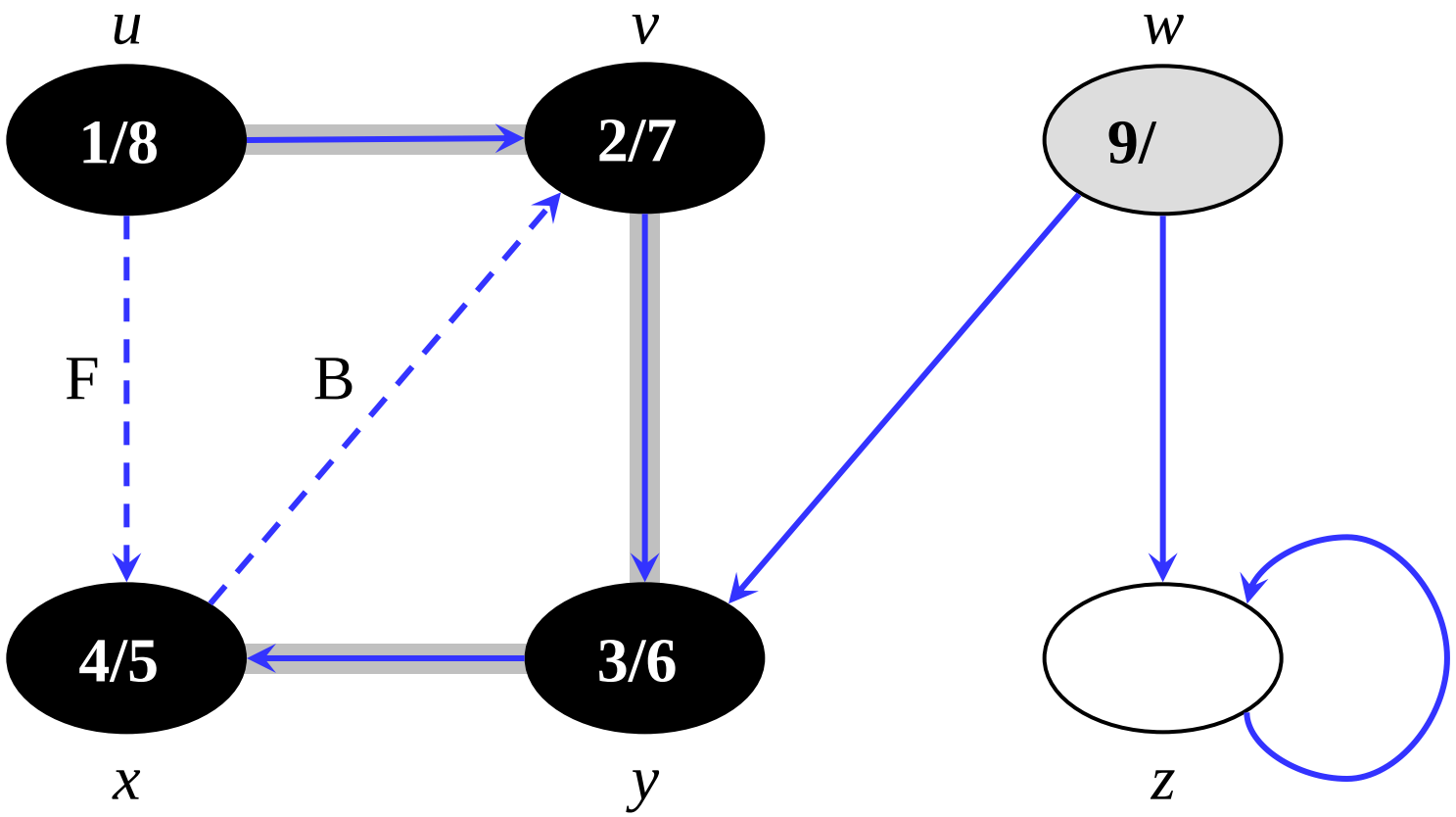


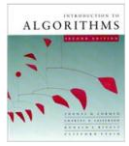
Depth-first search



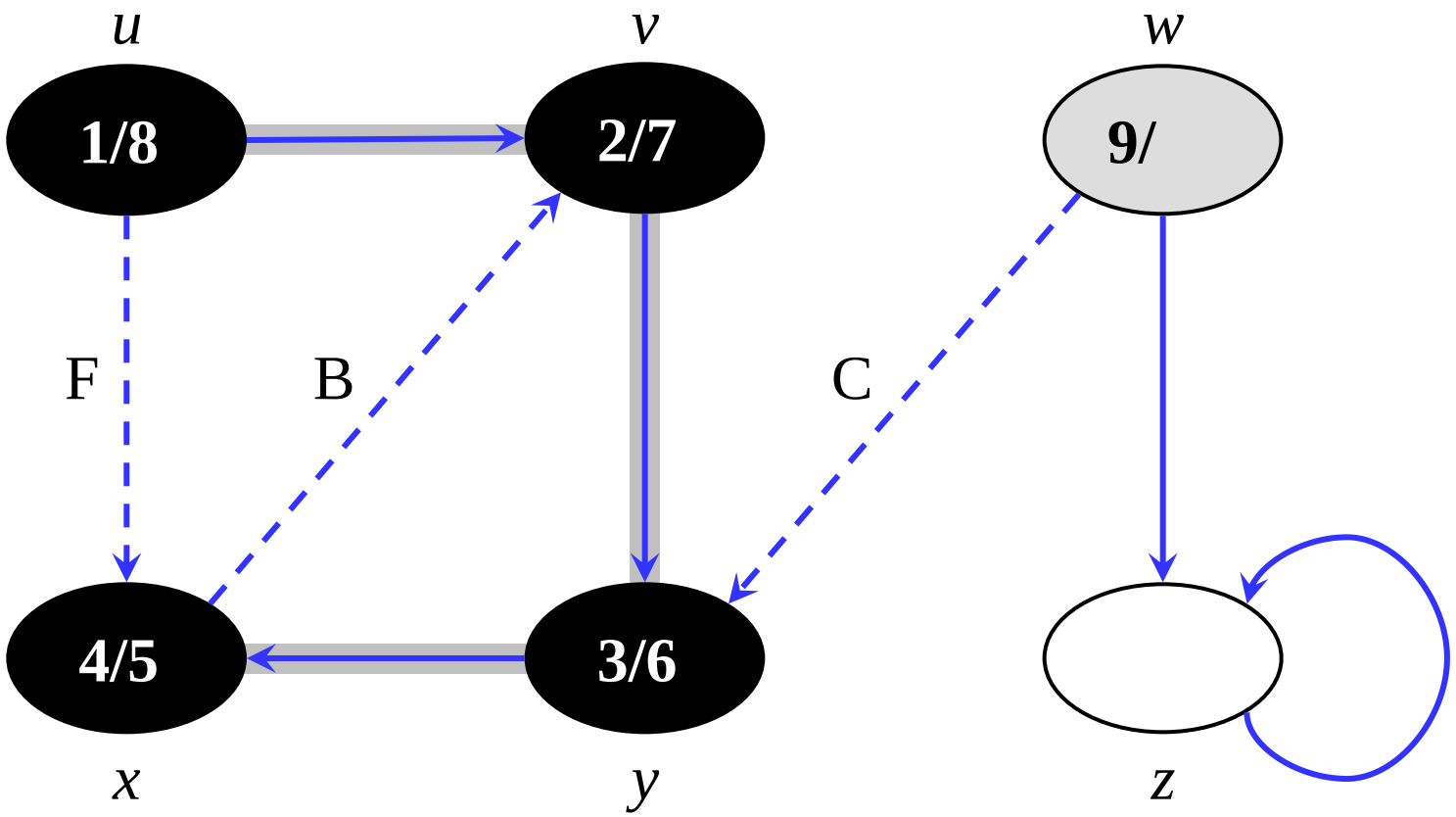


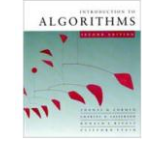
Depth-first search



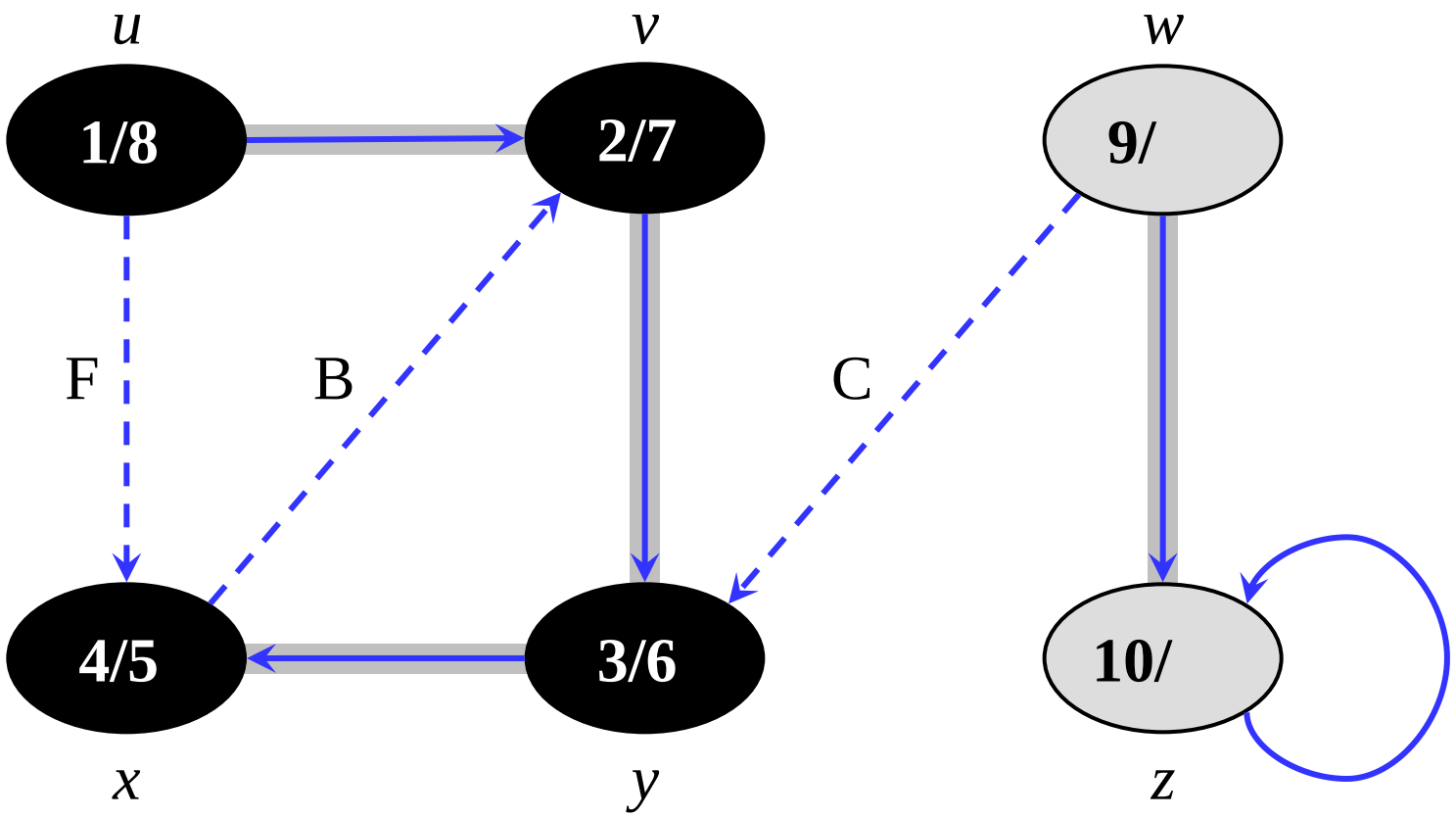


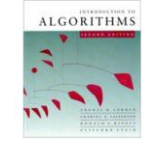
Depth-first search



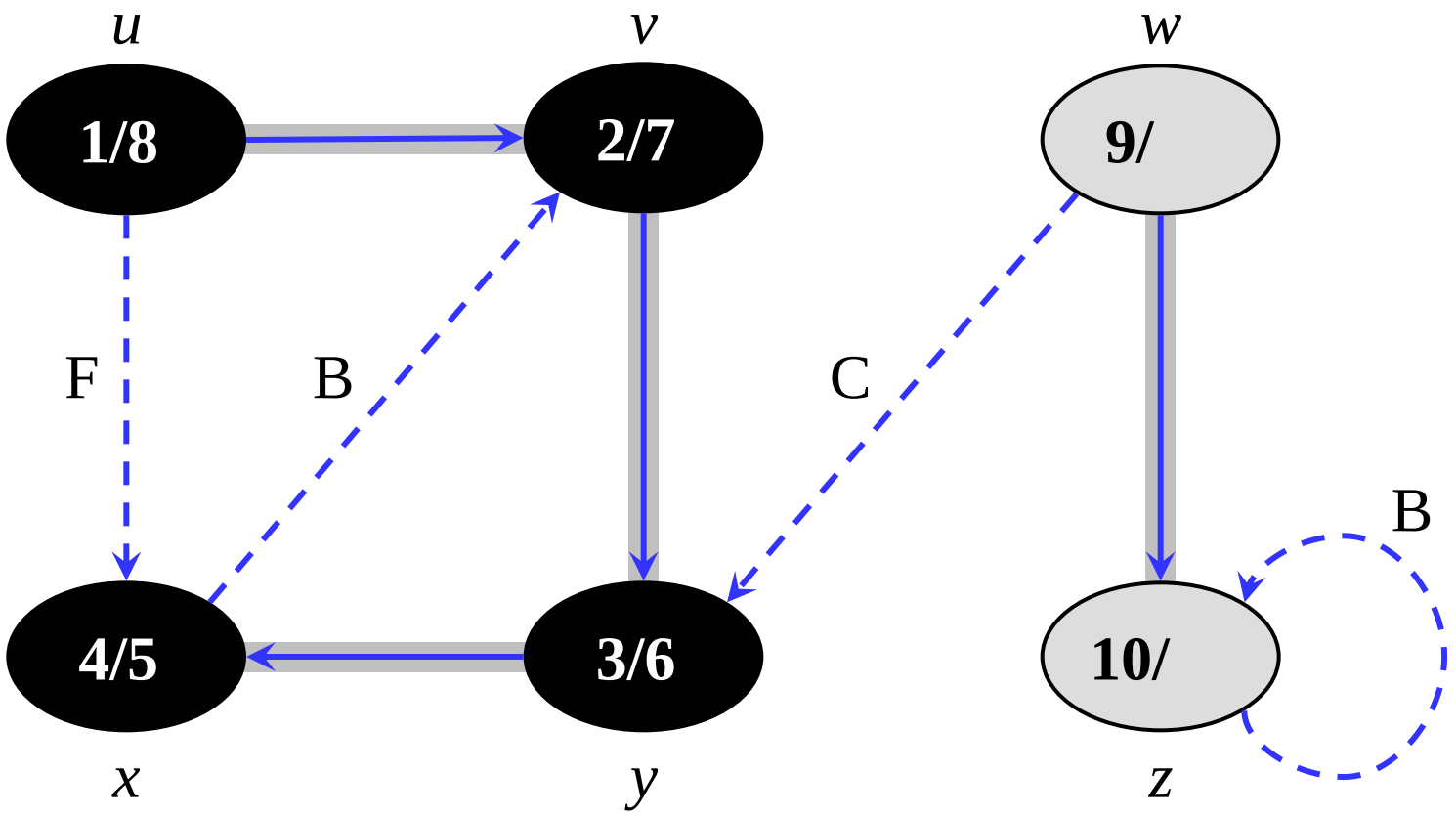


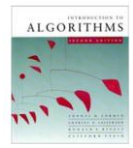
Depth-first search



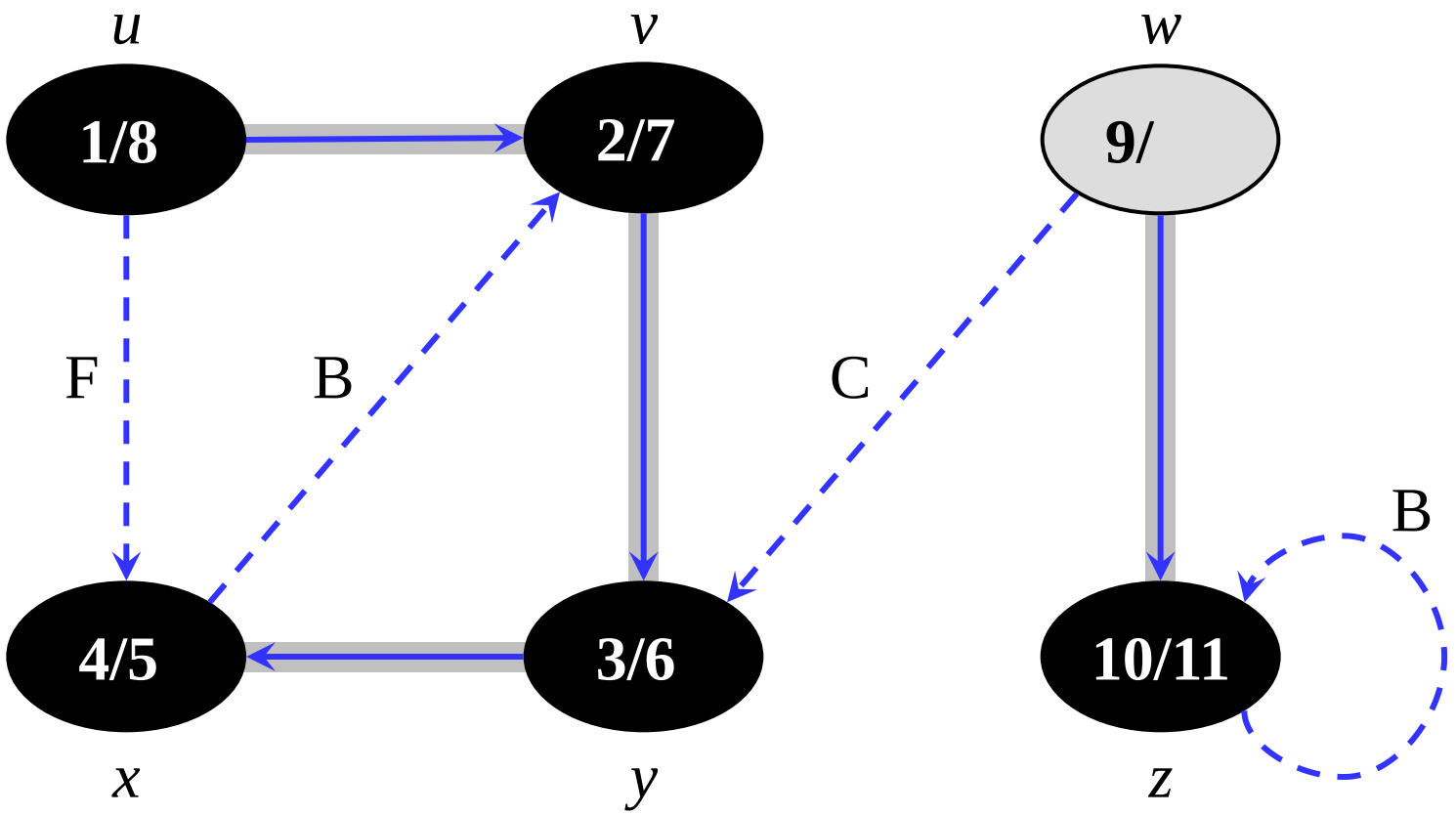


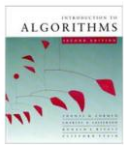
Depth-first search



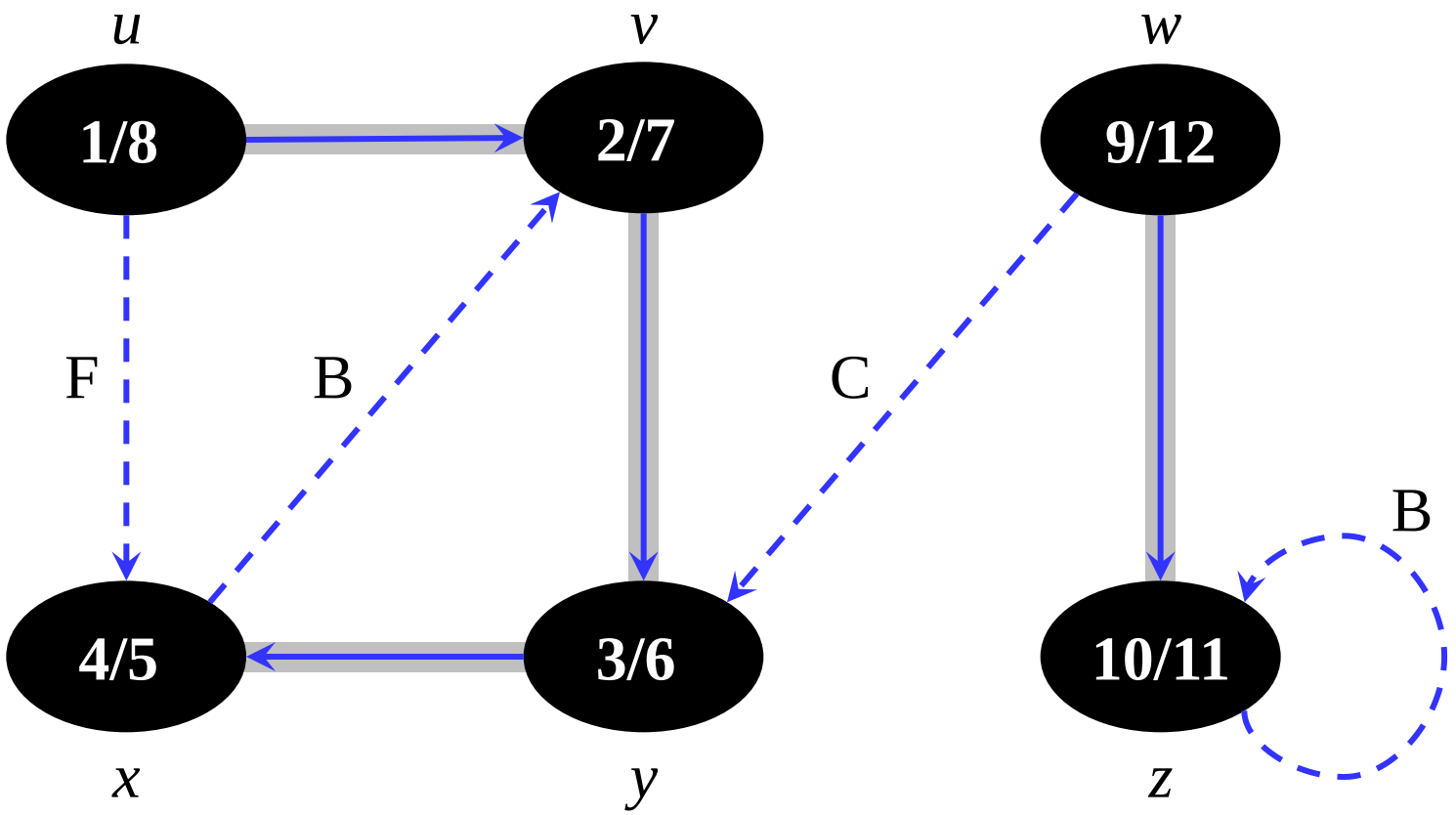


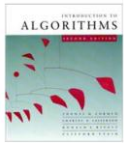
Depth-first search





Depth-first search

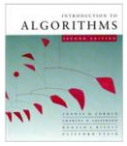


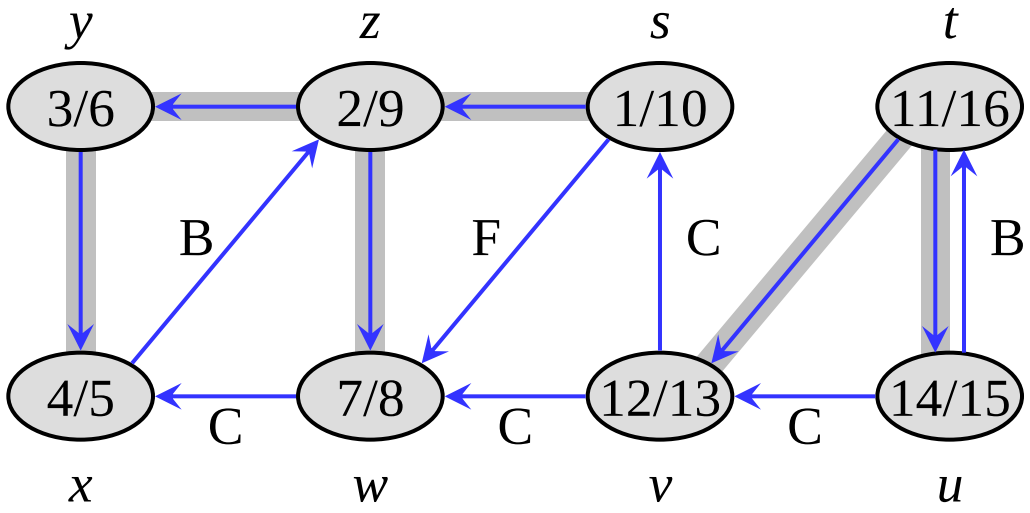


Depth-first search

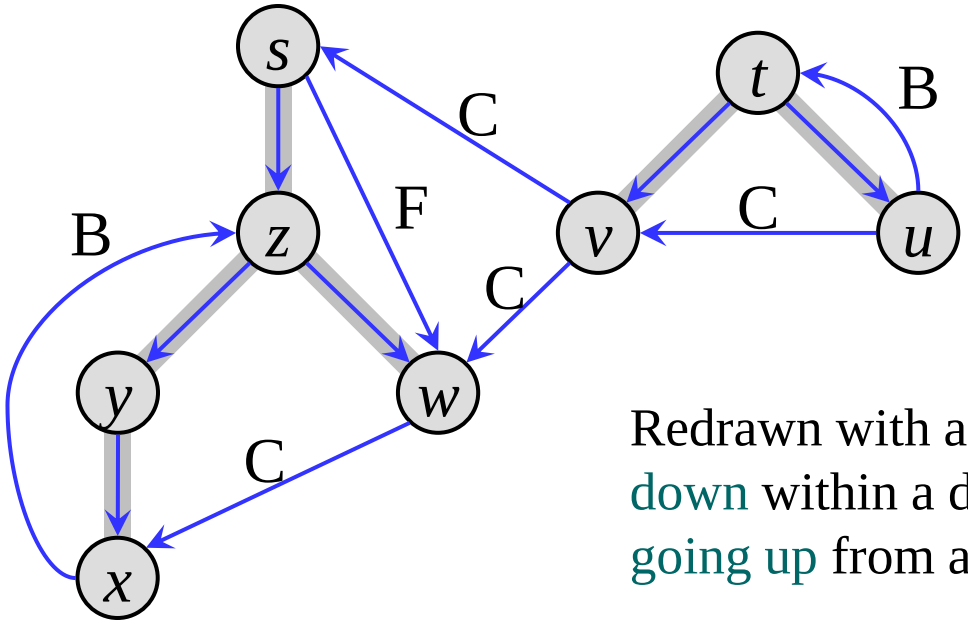
Time = $O(V + E)$.
. Similar to BFS analysis.

DFS forms a ***depth-first forest*** comprised of > 1 ***depth-first trees***.
Each tree is made of edges (u, v) such that u is gray and v is white when (u, v) is explored.

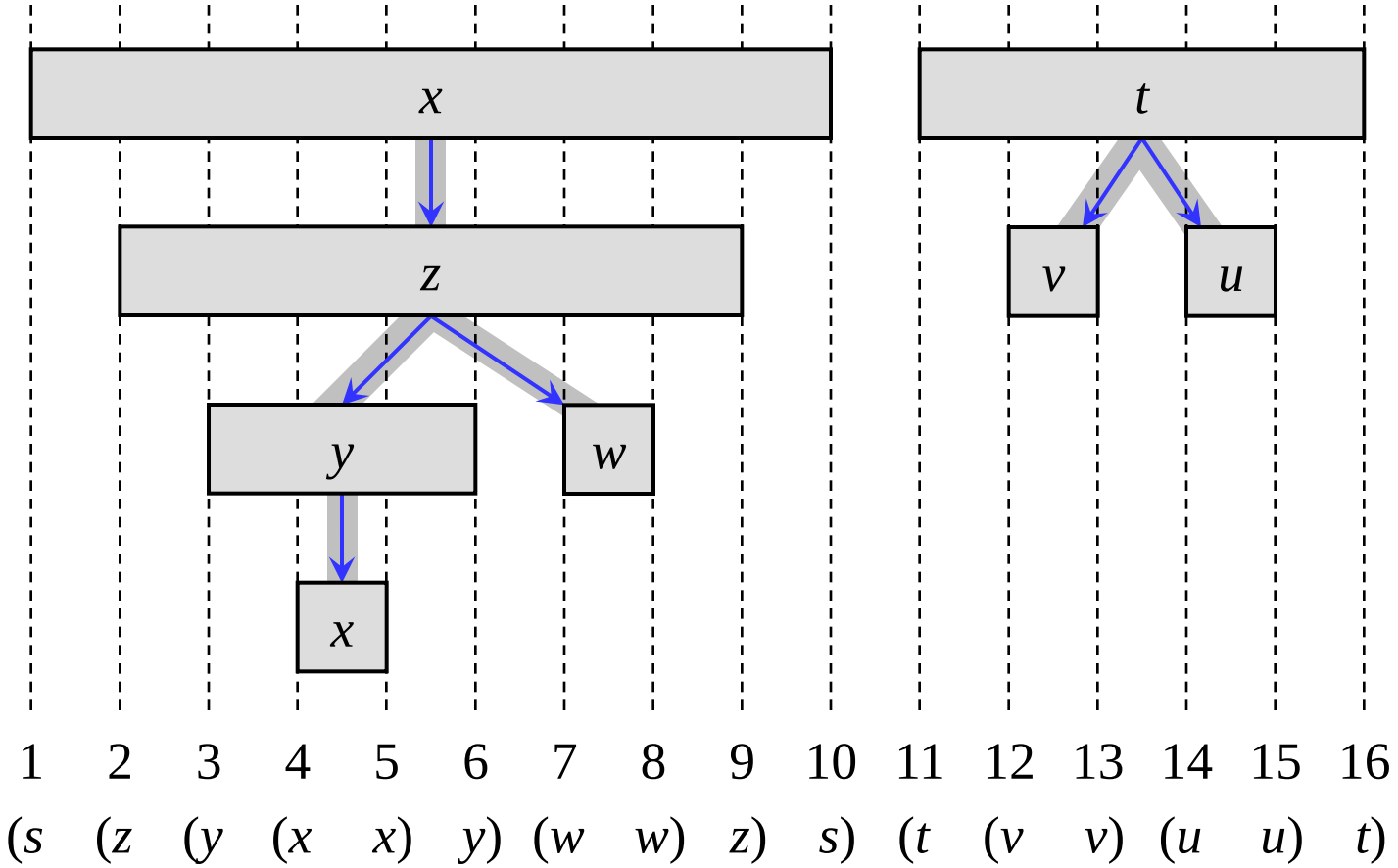




Results of DFS of a directed graph.

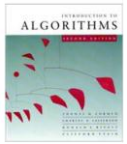


Redrawn with all tree and forward edges going down within a depth-first tree and all back edges going up from a descendant to an ancestor.



Intervals for the discovery time and finishing time of each vertex.

If two intervals overlap, then one is nested within the other, and the vertex corresponding to the smaller interval is a descendant of the vertex corresponding to the later.



Depth-first search

Theorem (Parenthesis theorem) ***[Refer to the interval graph.]***

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.

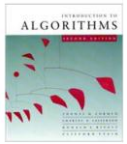
3. $d[v] < d[u] < f[u] < f[v]$ and u is a descendant of v.

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

Like parentheses:

. OK: ( ) [ ] ([ ]) [( )]

. Not OK: ([ )] [( ])



Topological sort

Directed acyclic graph (dag)

A directed graph with no cycles.

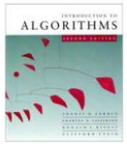
Good for modeling processes and structures that have a ***partial order*:**

. $a > b$ and $b > c \Rightarrow a > c$.

. But may have a and b such that neither $a > b$ nor $b > a$.

, i.e., not comparable between a and b.

Can always make a ***total order*** (either $a > b$ or $b > a$ for all $a = b$) from a partial order. In fact, that's what a topological sort will do.



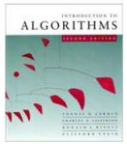
Topological sort

Topological sort of a dag: a linear ordering of vertices such that if $(u, v) \in E$, then u appears somewhere before v. (Not like sorting numbers.)

TOPOLOGICAL-SORT(V, E)
call DFS(V, E) to compute finishing times $f[v]$ for all $v \in V$,
output vertices in order of *decreasing* finish times

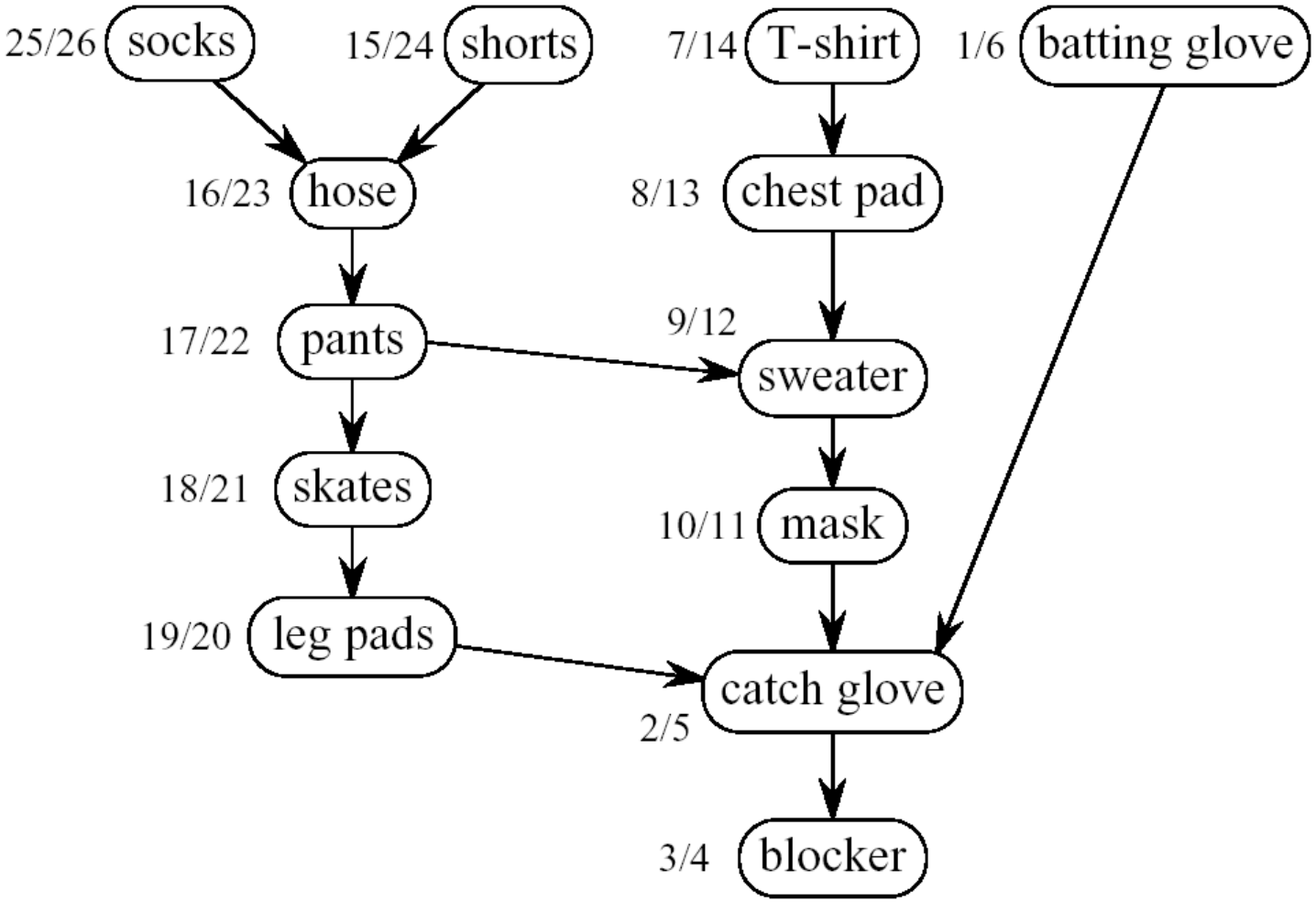
Don't need to sort by finish times.
. Can just output vertices as they're finished and understand that we want the *reverse* of this list.
. Or put them onto the *front* of a linked list as they're finished. When done, the list contains vertices in topologically sorted order.

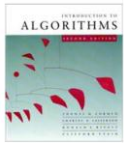
Time: $(V + E)$.



Topological sort

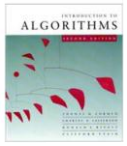
Example: dag of dependencies for putting on goalie equipment:





Topological sort

Order:
26 socks
24 shorts
23 hose
22 pants
21 skates
20 leg pads
14 t-shirt
13 chest pad
12 sweater
11 mask
6 batting glove
5 catch glove
4 blocker



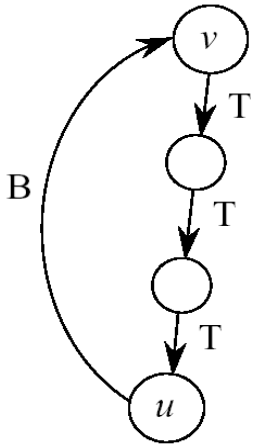
Topological sort

Lemma

A directed graph G is acyclic if and only if a DFS of G yields no back edges.

Proof $\Rightarrow$: Show that back edge $\Rightarrow$ cycle.

Suppose there is a back edge (u, v). Then v is ancestor of u in depth-first forest.



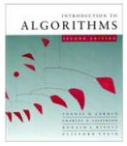
(Tree edge)

Therefore, there is a path $v \rightsquigarrow u$, so $v \rightsquigarrow u \rightarrow v$ is a cycle.

$\Leftarrow$: Show that cycle $\Rightarrow$ back edge.

Suppose G contains cycle c. Let v be the first vertex discovered in c, and let (u, v) be the preceding edge in c. At time $d[v]$, vertices of c form a white path $v \rightsquigarrow u$ (since v is the first vertex discovered in c). By white-path theorem, u is descendant of v in depth-first forest. Therefore, (u, v) is a back edge. ■ (lemma)

If there is a cycle the following always happens: *v* discovered first and *(u, v)* exists.

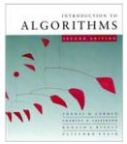


Topological sort

Correctness: Just need to show if $(u, v) \in E$ *(edge of a dag)*,
then the result of topological sort gives $f[v] < f[u]$.

When we explore *(u, v) (within topological sort and within DFS(G))*,
- *u* is gray.

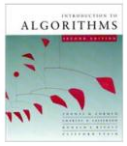
Now consider all the possible colors of v and show f [v] < f [u] for legal colors of dag.
- Is *v* gray, too?
  - *No*, because then *v* would be ancestor of *u*.
    ⇒ *(u, v)* is a back edge.
    ⇒ contradiction of previous lemma (dag has no back edges).
- Is *v* white?
  - Then becomes descendant of *u*.
    By parenthesis theorem, $d[u] < d[v] < f[v] < f[u]$.
- Is *v* black?
  - Then *v* is already finished.
    Since we're exploring *(u, v)*, we have not yet finished *u*.
    Therefore, $f[v] < f[u]$.



Topological sort

Topological-sort(G)

1. Call DFS(G) to compute finishing time f[*v*] 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

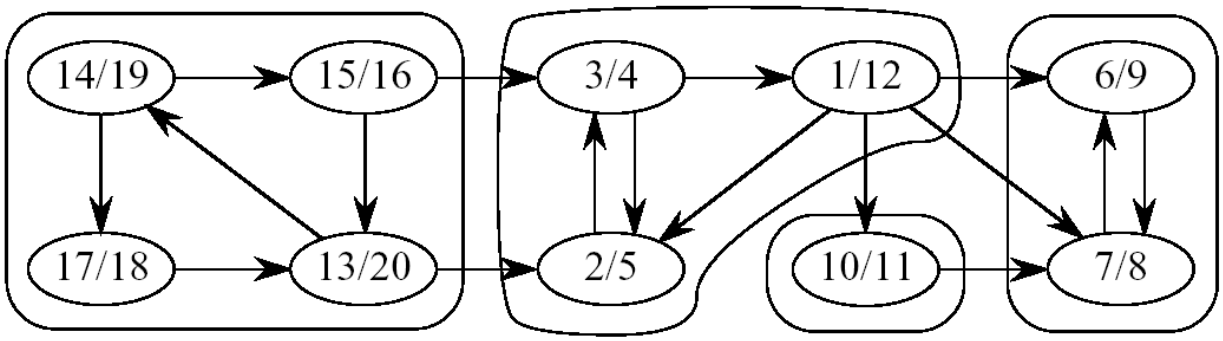


Strongly connected components

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

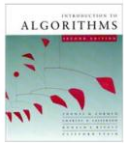
A ***strongly connected component*** (***SCC***) of G is a maximal set of vertices $C \subseteq V$ such that for all $u, v \in C$, both $u \rightsquigarrow v$ and $v \rightsquigarrow u$.

Example: *[Just show SCC's at first. Do DFS a little later.]*



A SCC is a cluster of vertices that are reachable from each other.

Idea: If we reverse the direction of arrows and call DFS with vertices in order of decreasing f [u] each tree vertices corresponds to a SCC.



Strongly connected components

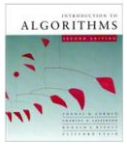
Algorithm uses $G^{\mathrm{T}} =$ ***transpose*** of G.

- $G^{\mathrm{T}} = (V, E^{\mathrm{T}})$, $E^{\mathrm{T}} = \{(u, v) : (v, u) \in E\}$.
- G^{T} is G with all edges reversed.

Can create G^{T} in $\Theta(V + E)$ 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}.)

** Computation for G^T can be done in a manner similar to DFS.*

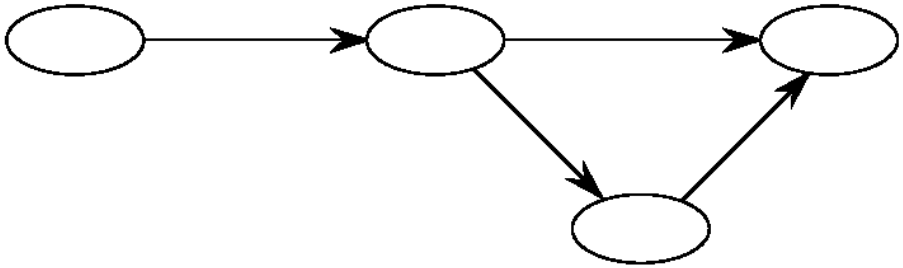


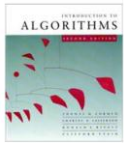
Strongly connected components

Component graph

- $G^{\mathrm{SCC}} = (V^{\mathrm{SCC}}, E^{\mathrm{SCC}})$.
- V^{SCC} has one vertex for each SCC in G.
- E^{SCC} has an edge if there's an edge between the corresponding SCC's in G.

For our example:





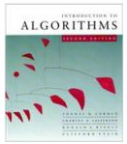
Strongly connected components

Lemma

G^{SCC} is a dag. More formally, let C and C' be distinct SCC's in G, let $u, v \in C$, $u', v' \in C'$, and suppose there is a path $u \rightsquigarrow u'$ in G. Then there cannot also be a path $v' \rightsquigarrow v$ in G.

Proof Suppose there is a path $v' \rightsquigarrow v$ in G. Then there are paths $u \rightsquigarrow u' \rightsquigarrow v'$ and $v' \rightsquigarrow v \rightsquigarrow u$ in G. Therefore, u and v' are reachable from each other, so they are not in separate SCC's. ■ (lemma)

If it does then v, u, v', u' are all in the same SCC.



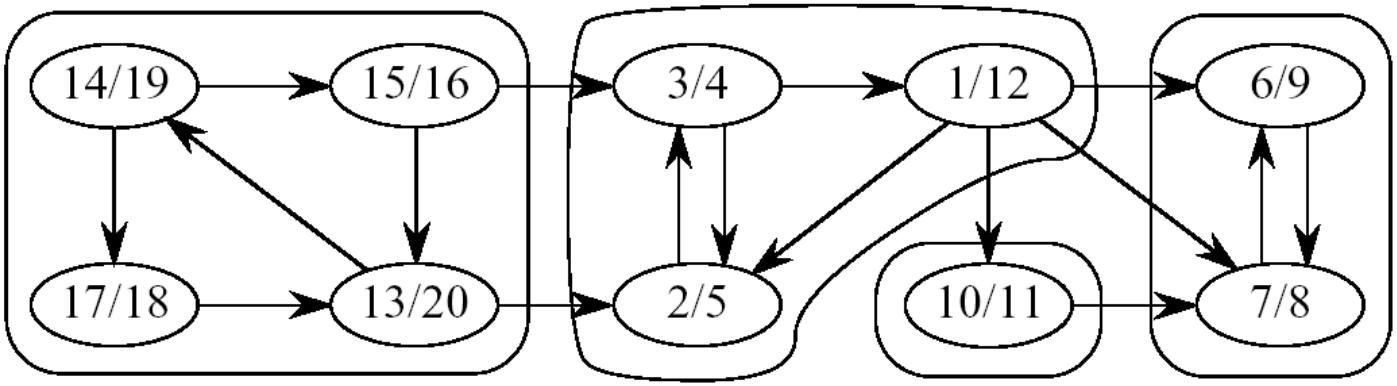
Strongly connected components

SCC(G)

- call DFS(G) to compute finishing times $f[u]$ for all u
- compute G^{T}
- call DFS(G^{T}), but in the main loop, consider vertices in order of decreasing $f[u]$ (as computed in first DFS)
- output the vertices in each tree of the depth-first forest formed in second DFS as a separate SCC

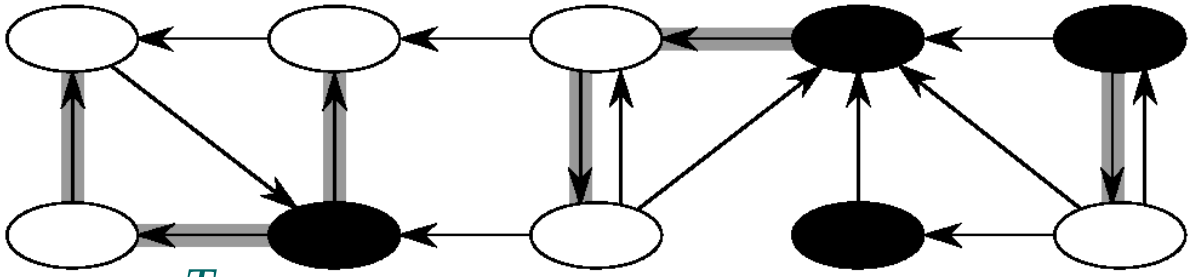
Time: $\Theta(V + E)$

Strongly connected components



Example:

1. Do DFS
2. G^T
3. DFS (roots blackened)



T

in GDS(G^T)