COMP251: Elementary graph algorithms

Jérôme Waldispühl
School of Computer Science
McGill University

Based on (Cormen *et al.*, 2002)

Based on slides from D. Plaisted (UNC)

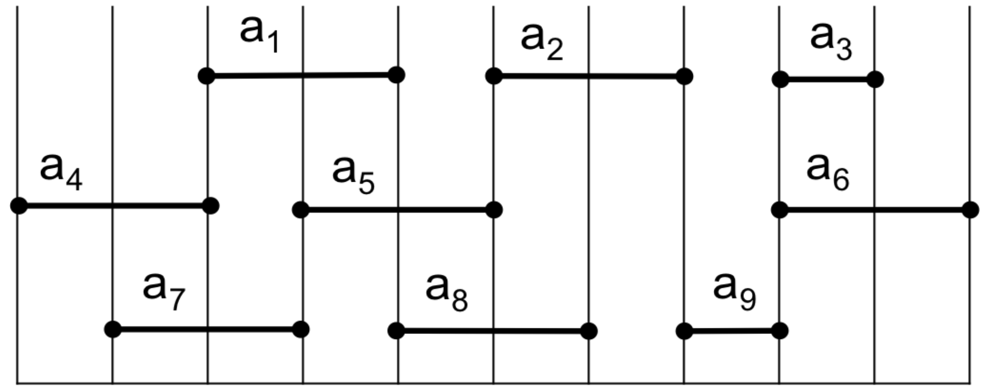
The greedy choice is a property that enable us to make a locally optimal choice at each step of the algorithm. Which of the following assertions are true?

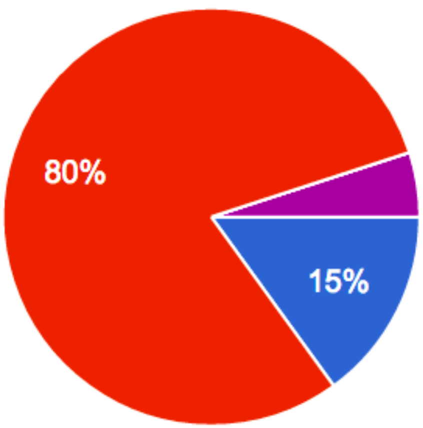
- It always guarantees to return an optimal solution for any problem where it can be applied. ✓
- The algorithm is usually fast. ✓
- It requires to define optimal sub-structures. ✓

It always guarantees to return an optimal solution for any problem where it is applied.	7	35%
The algorithm is usually fast.	7	35%
It requires to define optimal sub-structures.	18	90%

Consider the scheduling problem represented in the figure above. What will be the solution returned by the greedy algorithm introduced in class?

- a7, a5, a2, a9, a3
- a4, a1, a8, a9, a3 ✓
- a4, a5, a6
- a4, a5, a2, a9, a6
- None of these solutions





a7, a5, a2, a9, a3	3	15%
a4, a1, a8, a9, a3	16	80%
a4, a5, a6	0	0%
a4, a5, a2, a9, a6	0	0%
None of these solutions	1	5%

Elements of Greedy Algorithms

No general way to tell if a greedy algorithm is optimal, but two key ingredients are:

- Greedy-choice Property

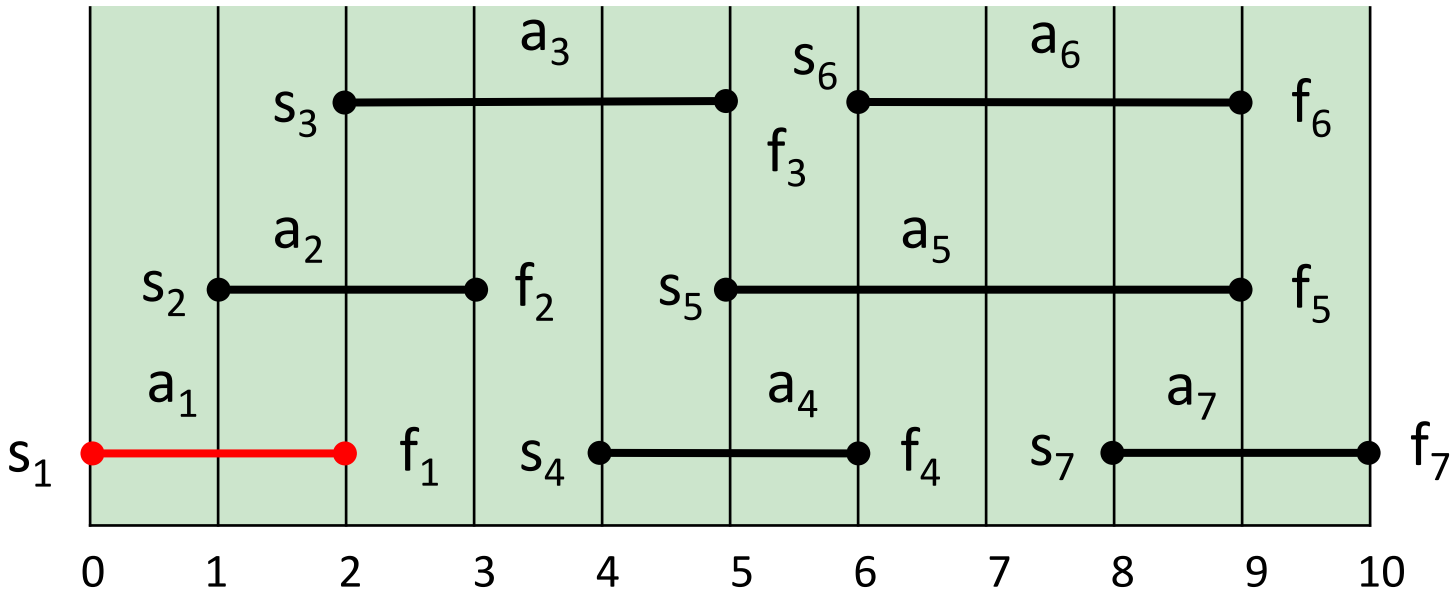
  (an optimal solution can be found at by making a locally optimal choice)

- Optimal Substructure.

Activity-selection Problem

i	1	2	3	4	5	6	7
s_i	0	1	2	4	5	6	8
f_i	2	3	5	6	9	9	10

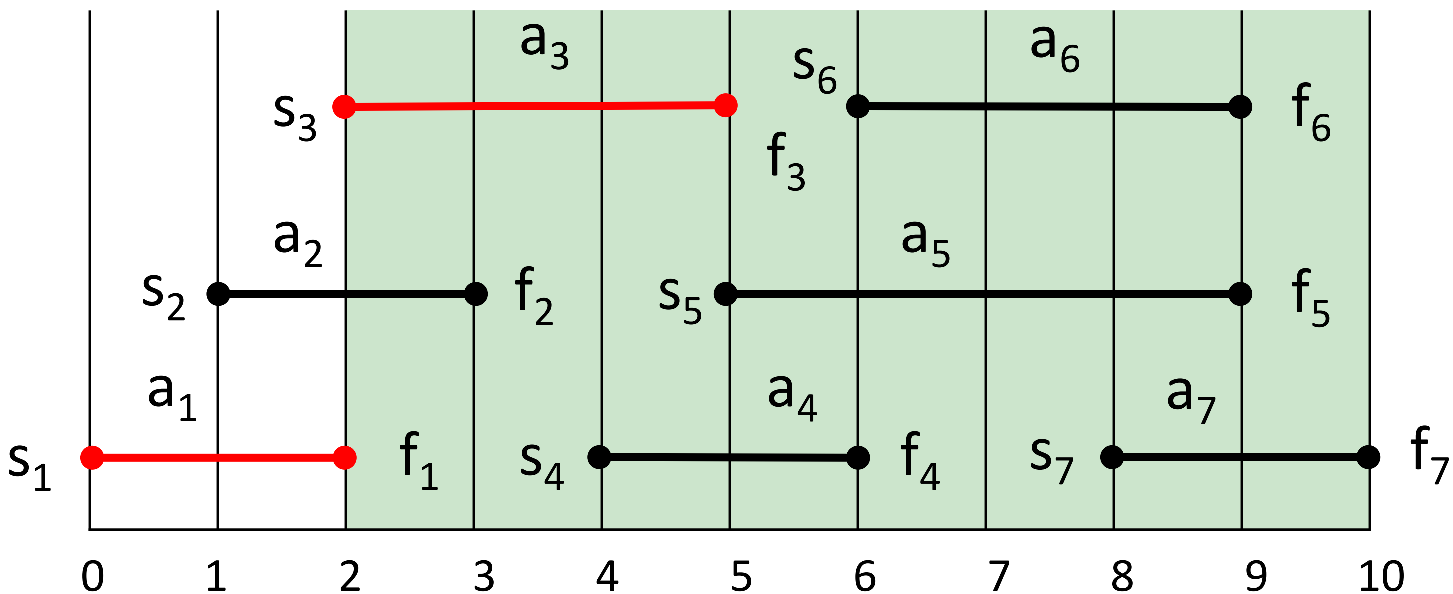
Activities sorted by finishing time.



Activity-selection Problem

i	1	2	3	4	5	6	7
s_i	0	1	2	4	5	6	8
f_i	2	3	5	6	9	9	10

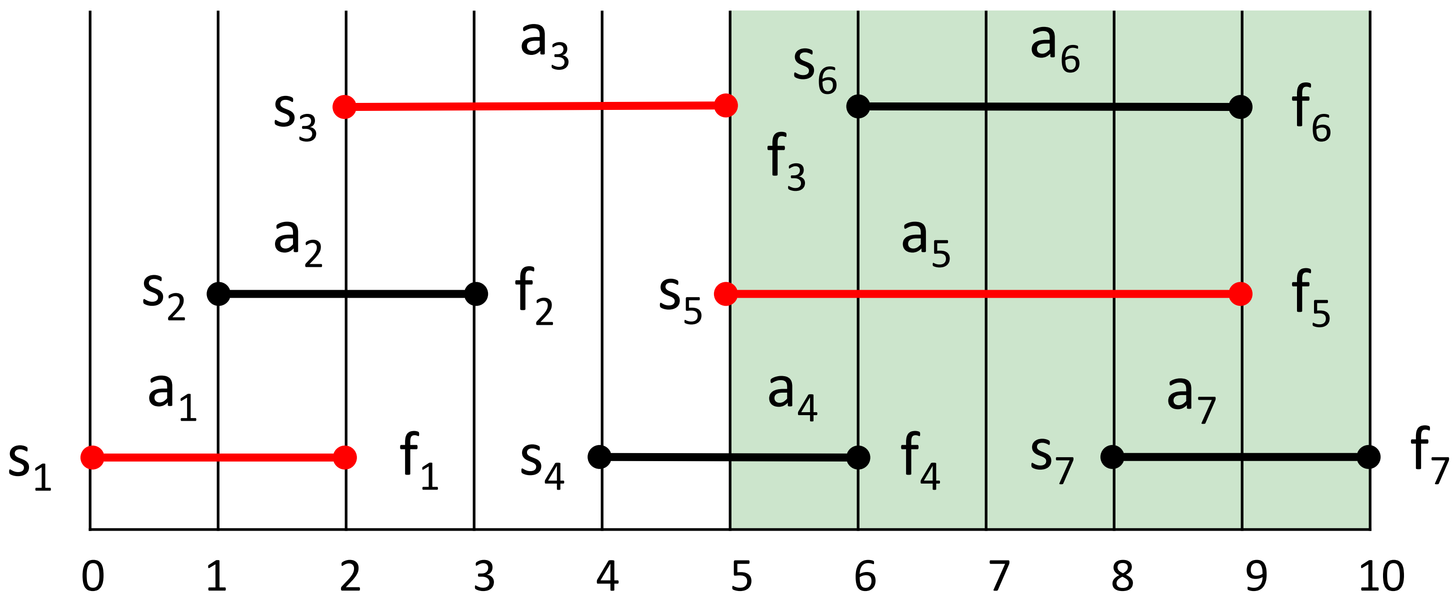
Activities sorted by finishing time.



Activity-selection Problem

i	1	2	3	4	5	6	7
s_i	0	1	2	4	5	6	8
f_i	2	3	5	6	9	9	10

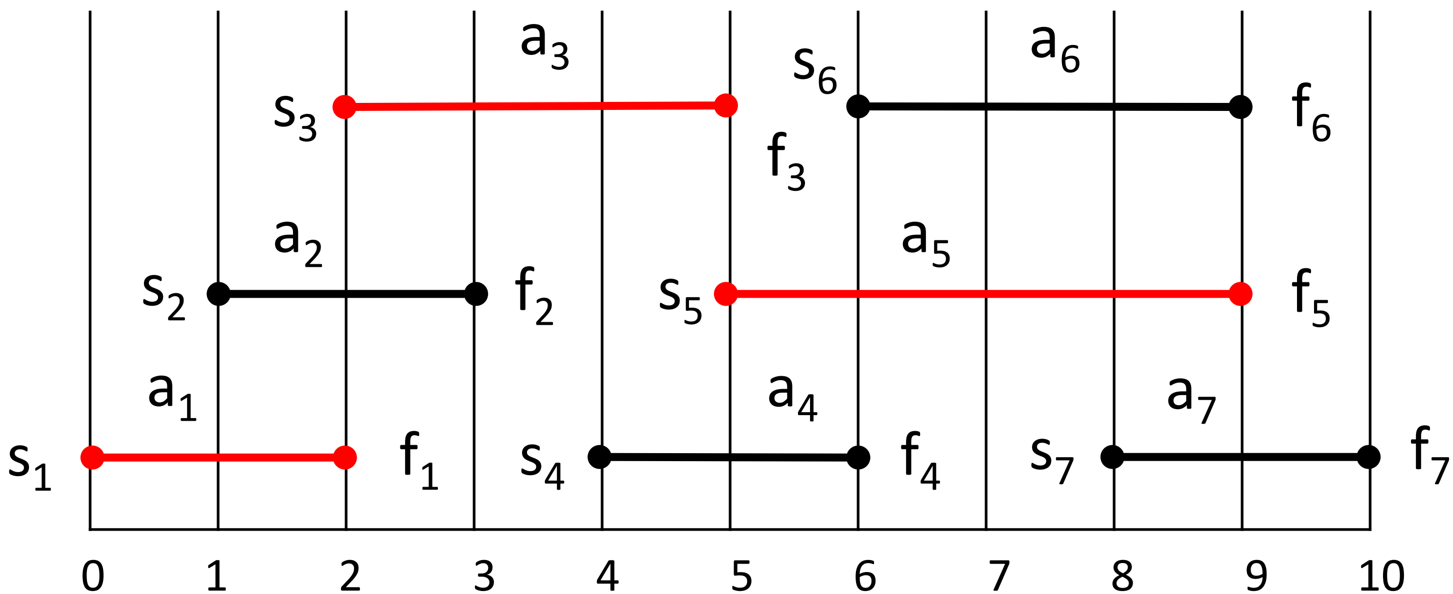
Activities sorted by finishing time.



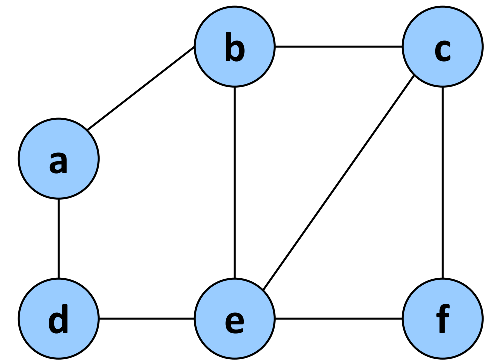
Activity-selection Problem

i	1	2	3	4	5	6	7
s_i	0	1	2	4	5	6	8
f_i	2	3	5	6	9	9	10

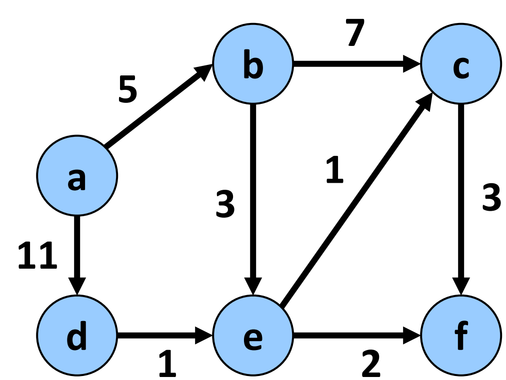
Activities sorted by finishing time.



Graphs



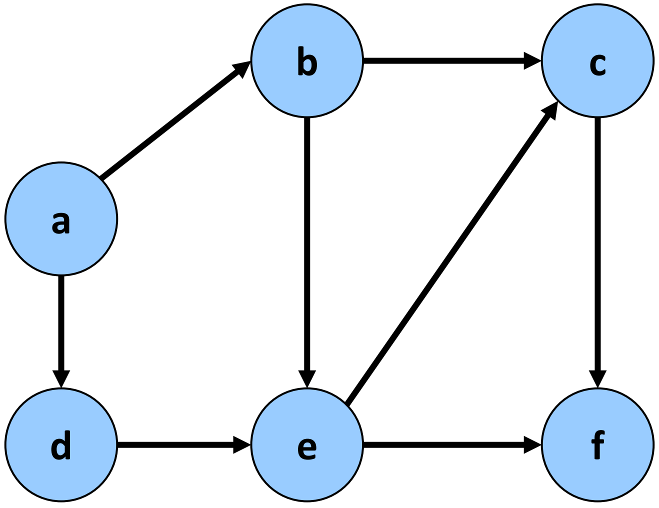
- *Graph G* = (*V*, *E*)
  - V = set of vertices
  - E = set of edges $\subseteq (V \times V)$
- Types of graphs
  - Undirected: edge $(u, v) = (v, u)$; for all v, $(v, v) \notin E$ (No self loops.)
  - Directed: (u, v) is edge from u to v, denoted as $u \to v$. Self loops are allowed.
  - Weighted: each edge has an associated weight, given by a weight function $w : E \to \mathbf{R}$.
  - Dense: $|E| \approx |V|^2$.
  - Sparse: $|E| << |V|^2$.
- $|E| = O(|V|^2)$



Properties

- If $(u, v) \in E$, then vertex v is adjacent to vertex u.
- Adjacency relationship is:
  - Symmetric if G is undirected.
  - Not necessarily so if G is directed.
- If G is connected:
  - There is a path between every pair of vertices.
  - $|E| \geq |V| - 1$.
  - Furthermore, if $|E| = |V| - 1$, then G is a tree.

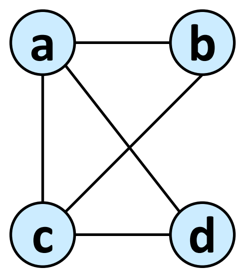
Vocabulary

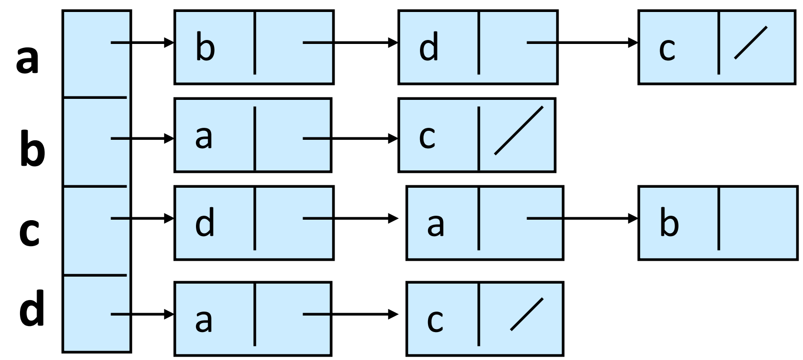


- Ingoing edges of u: $\{ (v,u) \in E \}$ (e.g. in(e) = $\{ (b,e), (d,e) \}$)
- Outgoing edges of u: $\{ (u,v) \in E \}$ (e.g. out(d) = $\{ (d,e) \}$)
- In-degree(u): $|$ in(u) $|$
- Out-degree(u): $|$ out(u) $|$

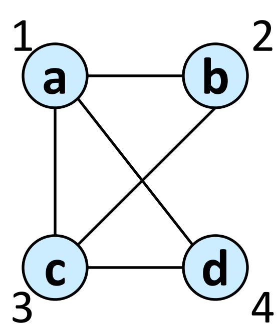
Representation of Graphs

- Two standard ways.
  - Adjacency Lists.





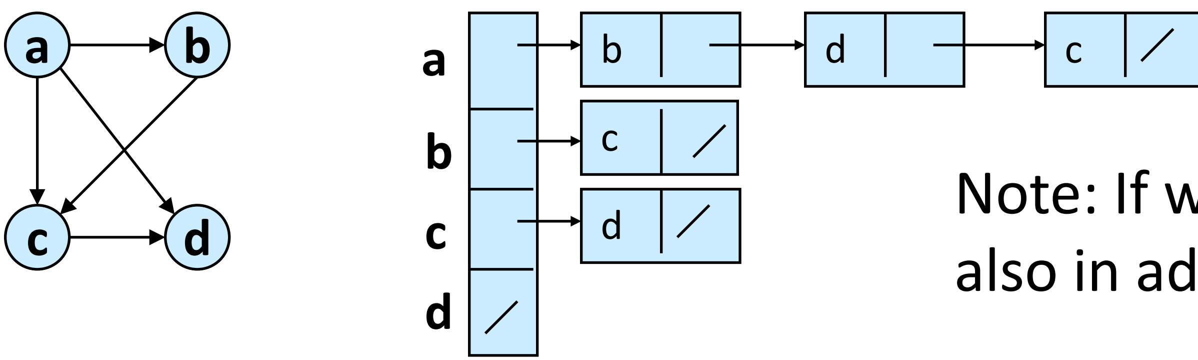
  - Adjacency Matrix.



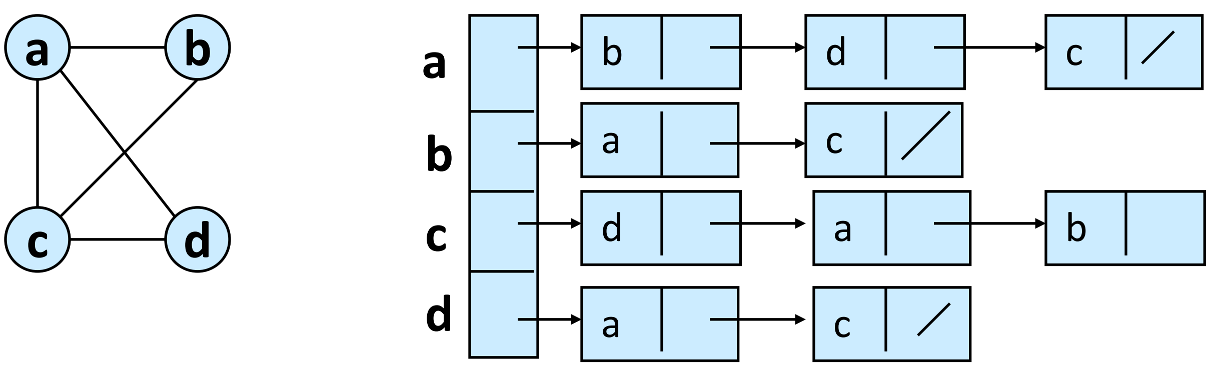
	1	2	3	4
1	0	1	1	1
2	1	0	1	0
3	1	1	0	1
4	1	0	1	0

Adjacency Lists

- Consists of an array *Adj* of $|V|$ lists.
- One list per vertex.
- For $u \in V$, $Adj[u]$ consists of all vertices adjacent to u.



Note: If weighted, store weights also in adjacency lists.



Storage Requirement

- For directed graphs:
  - Sum of lengths of all adj. lists is

    $$\sum_{v \in V} \text{out-degree}(v) = |E|$$

    No. of edges leaving *v*

  - Total storage: $\Theta(V+E)$

- For undirected graphs:
  - Sum of lengths of all adj. lists is

    $$\sum_{v \in V} \text{degree}(v) = 2|E|$$

    No. of edges incident on *v*. Edge (u,v) is incident on vertices *u* and *v*.

  - Total storage: $\Theta(V+E)$

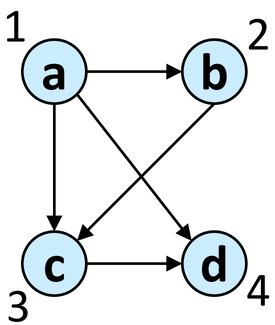
Pros and Cons: adj list

- Pros
  - Space-efficient, when a graph is sparse.
  - Can be modified to support many graph variants.
- Cons
  - Determining if an edge $(u,v) \in E$ is not efficient.
    - Have to search in u's adjacency list. $\Theta(\text{degree}(u))$ time.
    - $\Theta(V)$ in the worst case.

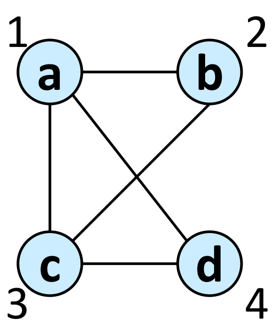
Adjacency Matrix

- $|V| \times |V|$ matrix A.
- Number vertices from 1 to $|V|$ in some arbitrary manner.
- A is then given by:

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



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



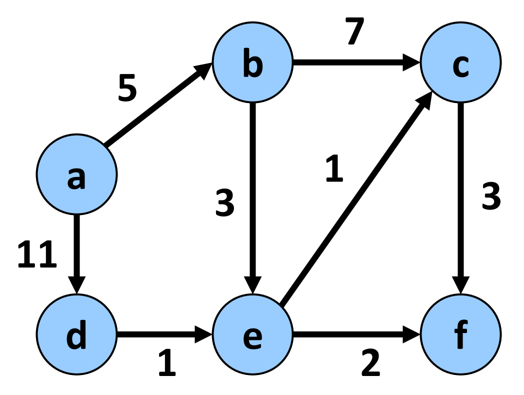
	1	2	3	4
1	0	1	1	1
2	1	0	1	0
3	1	1	0	1
4	1	0	1	0

$A = A^{\mathrm{T}}$ for undirected graphs.

Space and Time

- **Space:** $\Theta(V^2)$.
  - Not memory efficient for large sparse graphs.
- **Time:** to list all vertices adjacent to u: $\Theta(V)$.
- **Time:** to determine if $(u, v) \in E$: $\Theta(1)$.
- Can store weights instead of bits for weighted graph.

	a	b	c	d	e	f
a	0	5	0	11	0	0
b	0	0	7	0	3	0
c	0	0	0	0	0	3
d	0	0	0	0	1	0
e	0	0	1	0	0	2
f	0	0	0	0	0	0



Graph-searching Algorithms (COMP250)

- Searching a graph:
  - Systematically follow the edges of a graph to visit the vertices of the graph.
- Used to discover the structure of a graph.
- Standard graph-searching algorithms.
  - Breadth-first Search (BFS).
  - Depth-first Search (DFS).

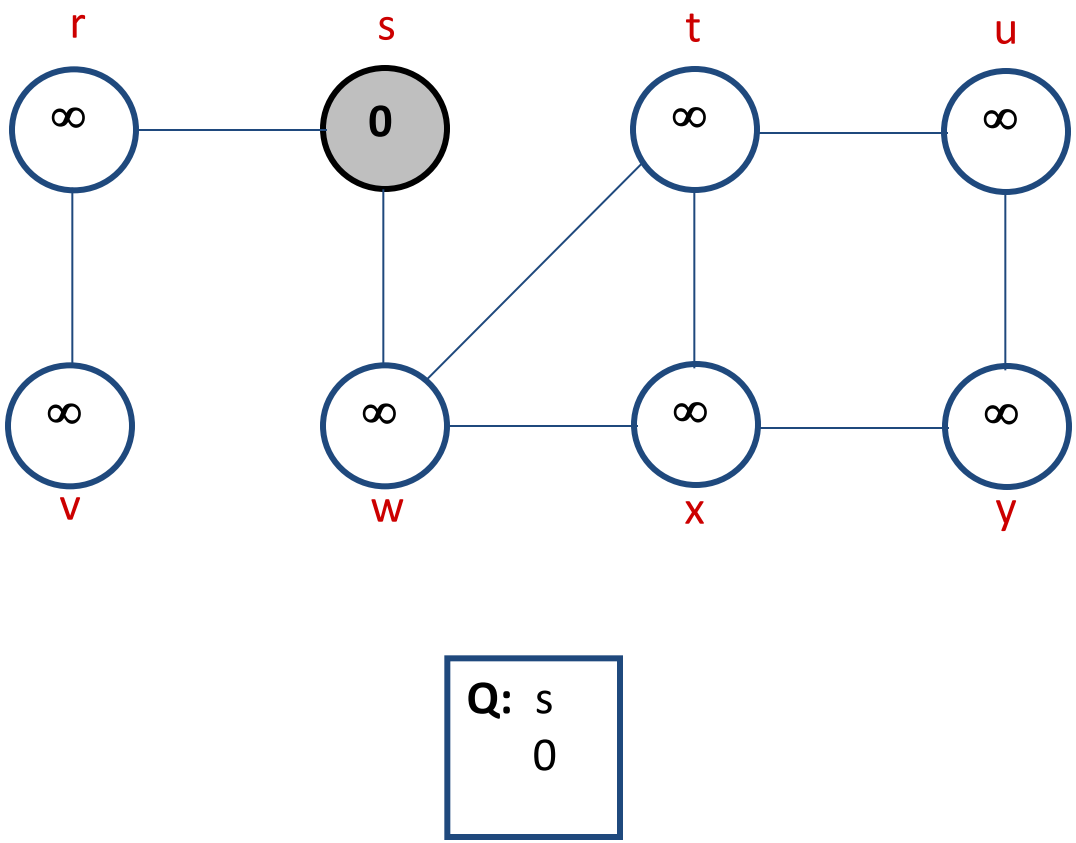
Breadth-first Search

- Expands the frontier between discovered and undiscovered vertices uniformly across the breadth of the frontier.
  - A vertex is "discovered" the first time it is encountered during the search.
  - A vertex is "finished" if all vertices adjacent to it have been discovered.
- Colors the vertices to keep track of progress.
  - White – Undiscovered.
  - Gray – Discovered but not finished.
  - Black – Finished.
    - Colors are required only to reason about the algorithm. Can be implemented without colors.

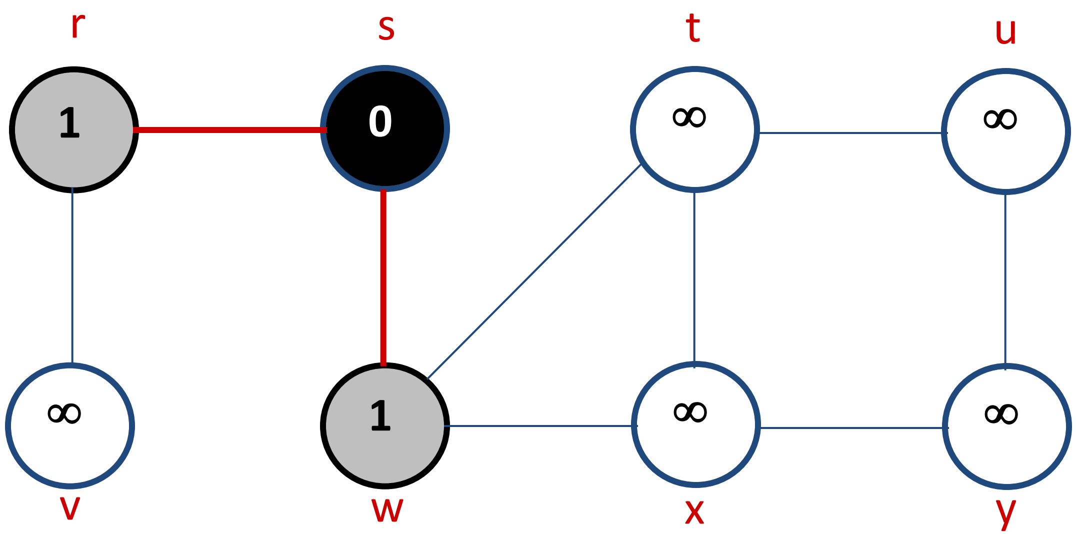
Breadth-first Search

- **Input:** Graph $G = (V, E)$, either directed or undirected, and ***source vertex*** $s \in V$.
- **Output:**
  - $d[v]$ = distance (smallest # of edges, or shortest path) from s to v, for all $v \in V$. $d[v] = \infty$ if v is not reachable from s.
  - $\pi[v] = u$ such that (u, v) is last edge on shortest path $s \rightsquigarrow v$.
    - u is v's predecessor.
  - Builds breadth-first tree with root s that contains all reachable vertices.

Example (BFS)

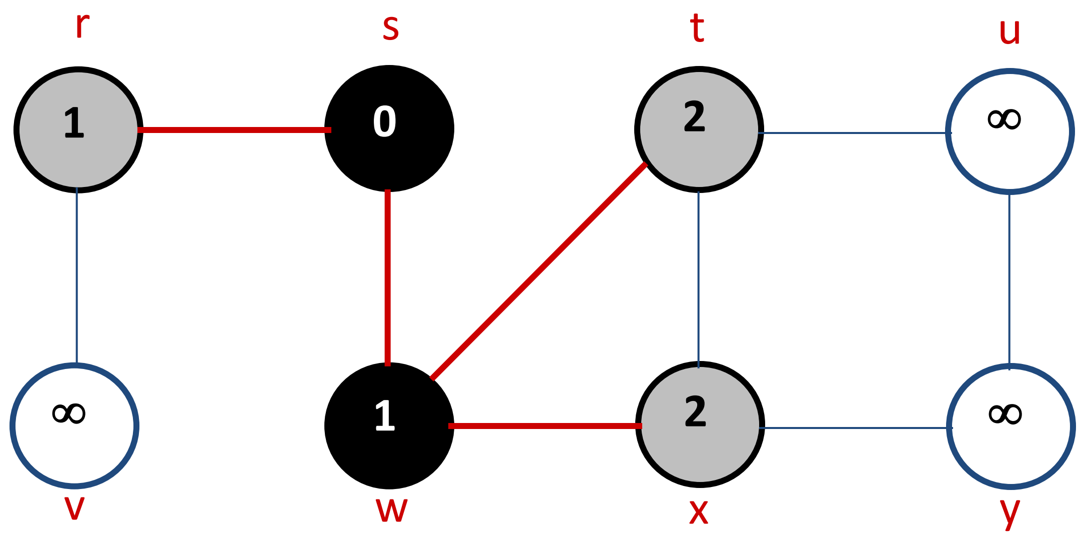


Example (BFS)



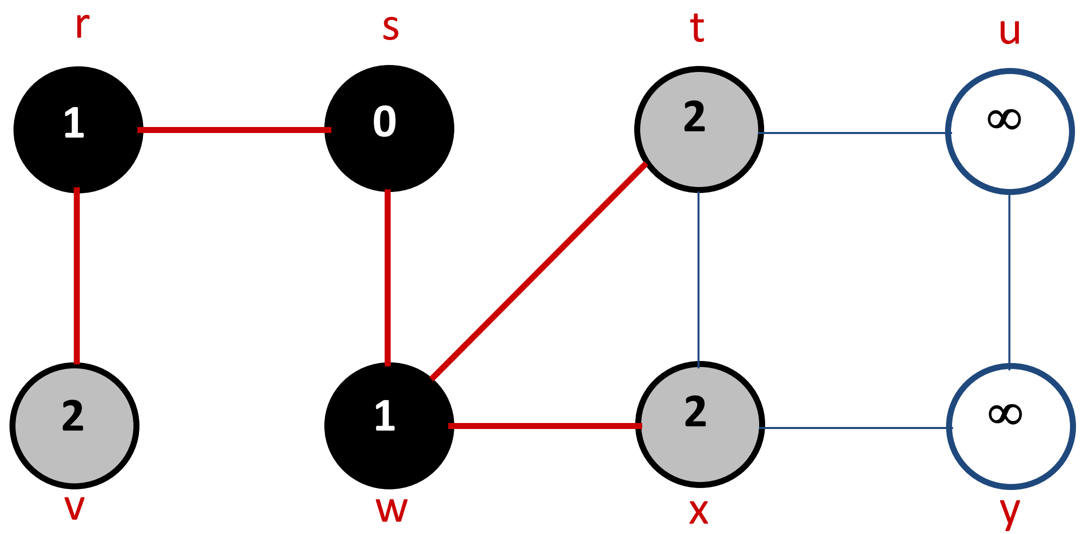
Q: w r
1 1

Example (BFS)



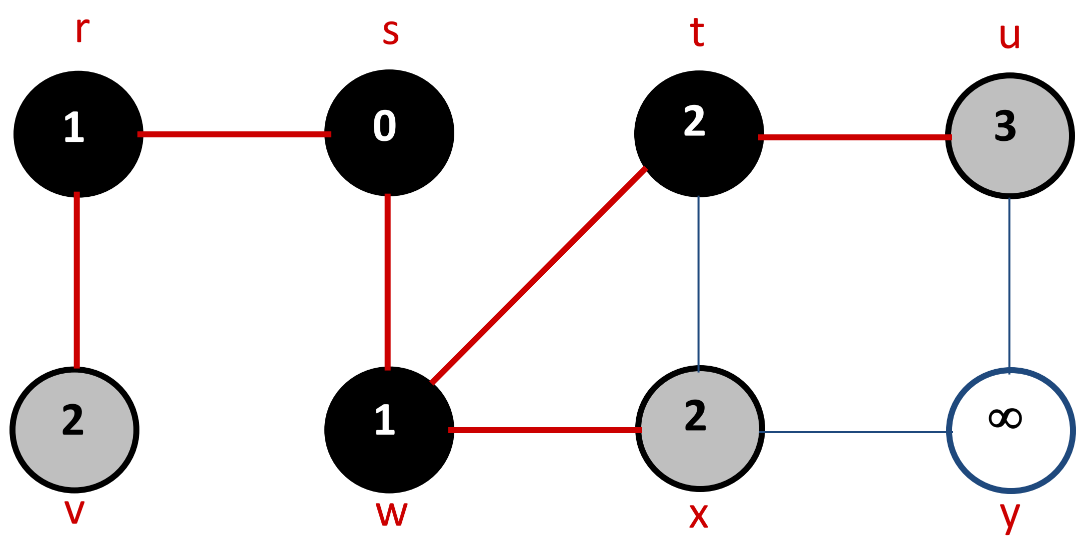
Q:	r	t	x
	1	2	2

Example (BFS)



Q: t x v
2 2 2

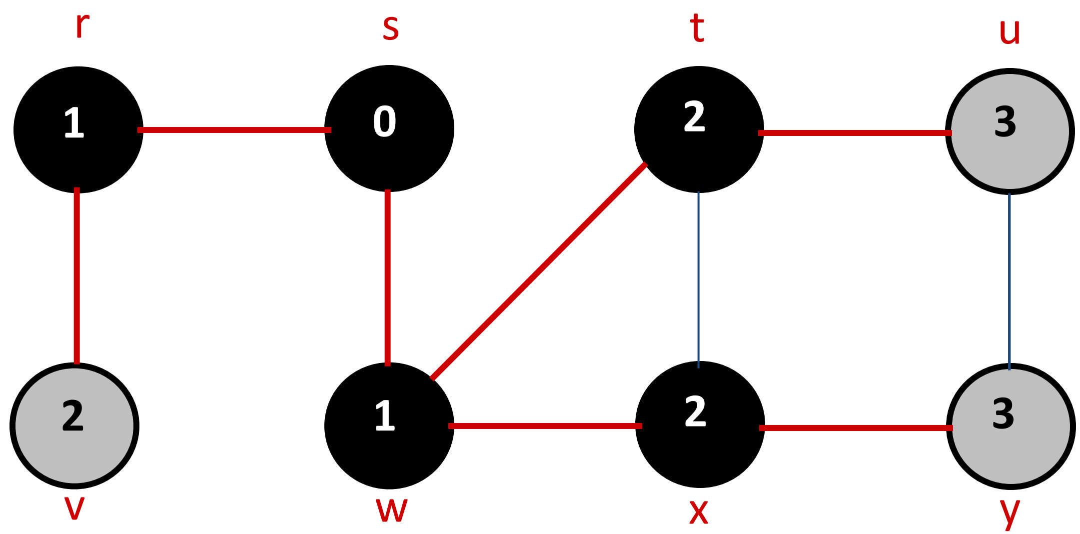
Example (BFS)



Q: x v u

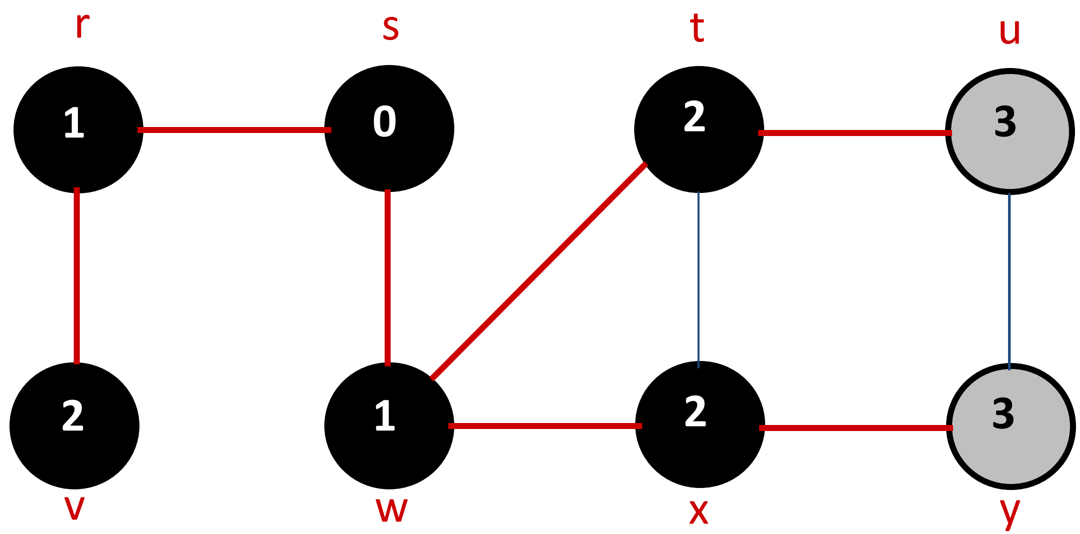
2 2 3

Example (BFS)



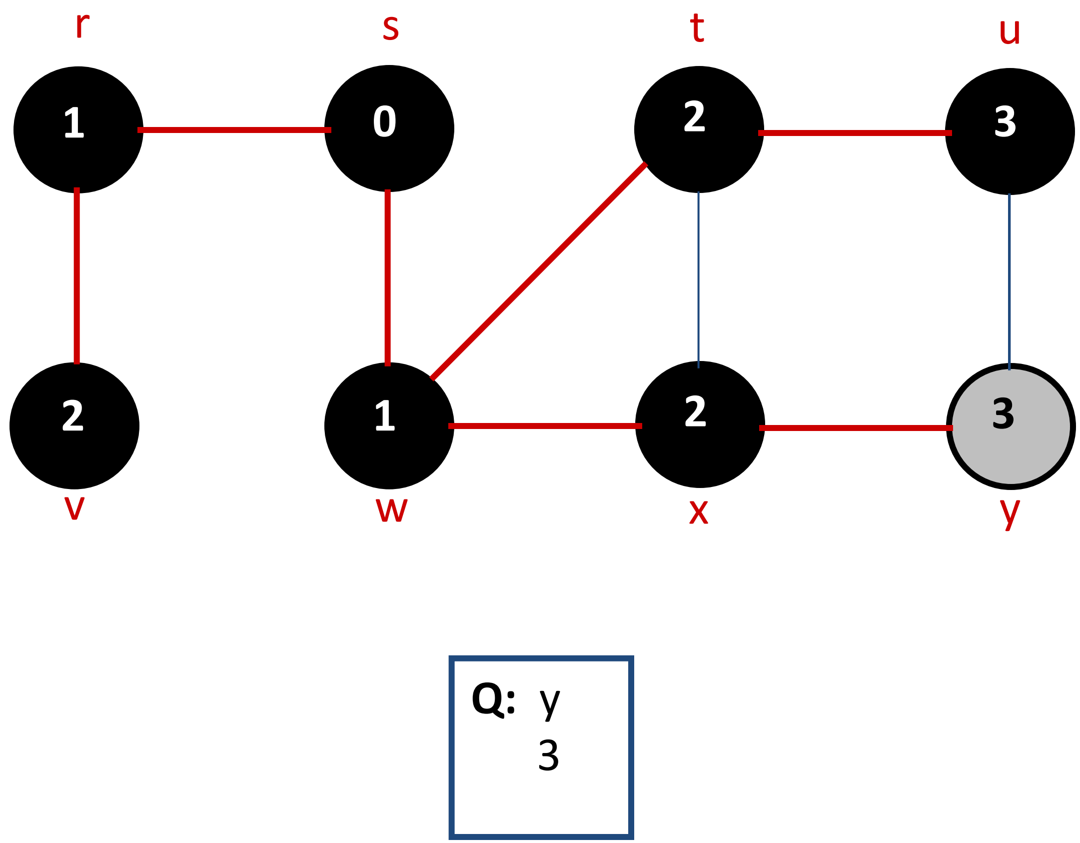
Q: v u y
2 3 3

Example (BFS)

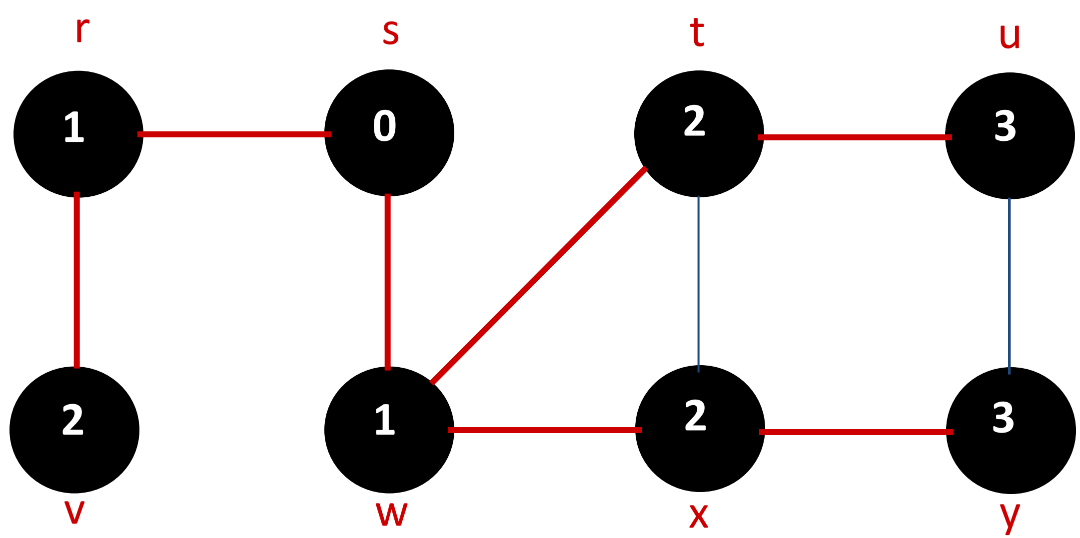


Q: u y
3 3

Example (BFS)

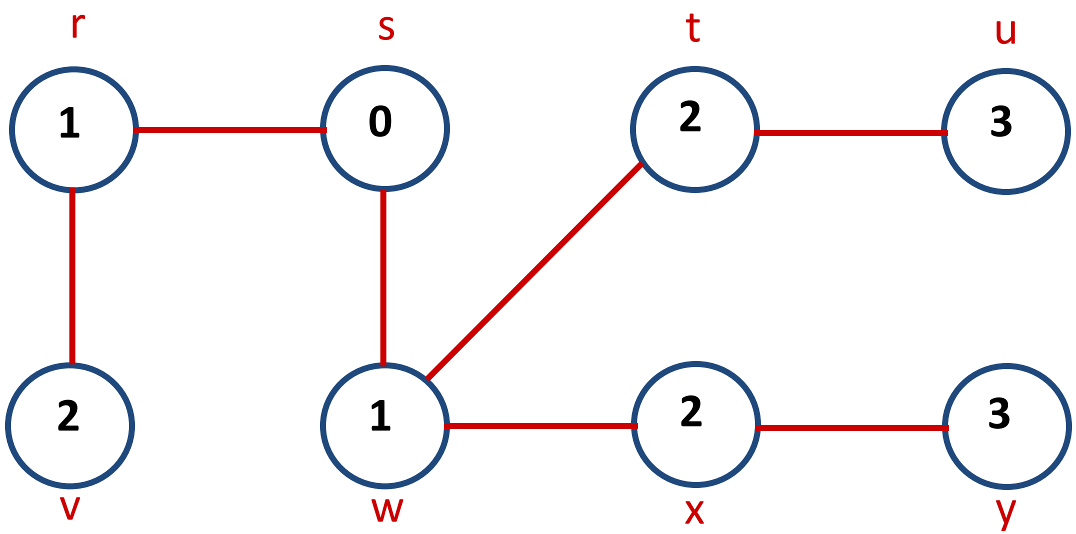


Example (BFS)



Q: $\varnothing$

Example (BFS)



BF Tree

Analysis of BFS

- Initialization takes $O(V)$.
- Traversal Loop
  - After initialization, each vertex is enqueued and dequeued at most once, and each operation takes $O(1)$. So, total time for queuing is $O(V)$.
  - The adjacency list of each vertex is scanned at most once. The sum of lengths of all adjacency lists is $\Theta(E)$.
- Summing up over all vertices => total running time of BFS is $O(V+E)$, linear in the size of the adjacency list representation of graph.

Depth-first Search (DFS)

- Explore edges out of the most recently discovered vertex *v*.
- When all edges of *v* have been explored, backtrack to explore other edges leaving the vertex from which *v* was discovered (its *predecessor*).
- "Search as deep as possible first."
- Continue until all vertices reachable from the original source are discovered.
- If any undiscovered vertices remain, then one of them is chosen as a new source and search is repeated from that source.

Depth-first Search

- **Input:** $G = (V, E)$, directed or undirected. No source vertex given.
- **Output:**
  - 2 **timestamps** on each vertex. Integers between 1 and $2|V|$.
    - $d[v]$ = ***discovery time*** (v turns from white to gray)
    - $f[v]$ = ***finishing time*** (v turns from gray to black)
  - $\pi[v]$: predecessor of $v = u$, such that v was discovered during the scan of u's adjacency list.
- Uses the same coloring scheme for vertices as BFS.

Pseudo-code

DFS(*G*)

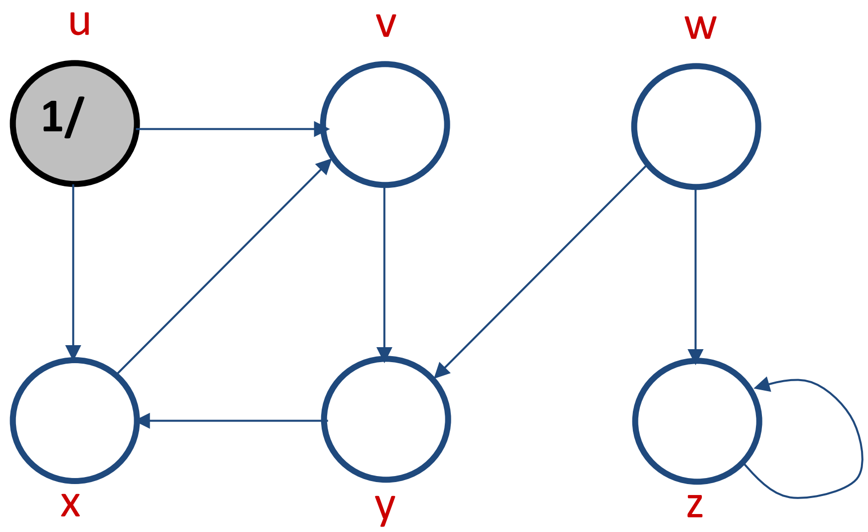
```
1. for each vertex u ∈ V[G]
2.     do color[u] ← white
3.         π[u] ← NIL
4. time ← 0
5. for each vertex u ∈ V[G]
6.     do if color[u] = white
7.           then DFS-Visit(u)
```

Uses a global timestamp ***time***.

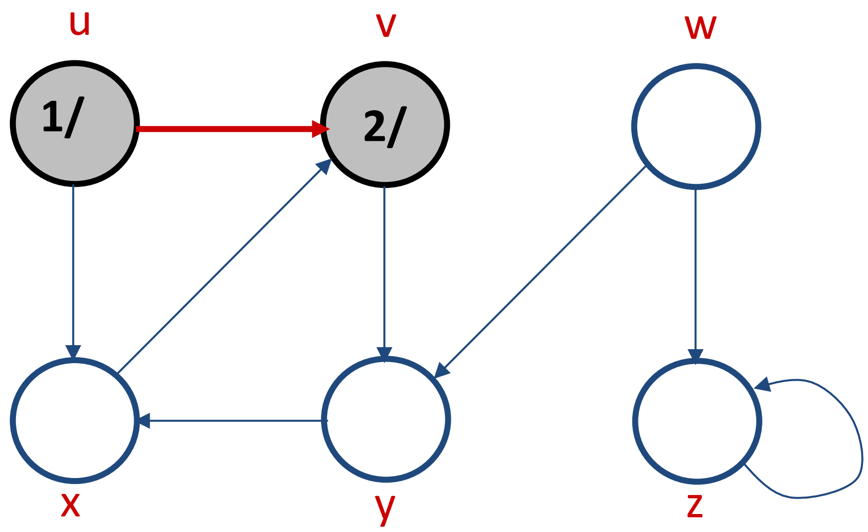
DFS-Visit(*u*)

```
1.    color[u] ← GRAY   ∇ White vertex u has
      been discovered
2.    time ← time + 1
3.    d[u] ← time
4.    for each v ∈ Adj[u]
5.        do if color[v] = WHITE
6.             then π[v] ← u
7.                  DFS-Visit(v)
8.    color[u] ← BLACK    ∇ Blacken u; it is
      finished.
9.    f[u] ← time ← time + 1
```

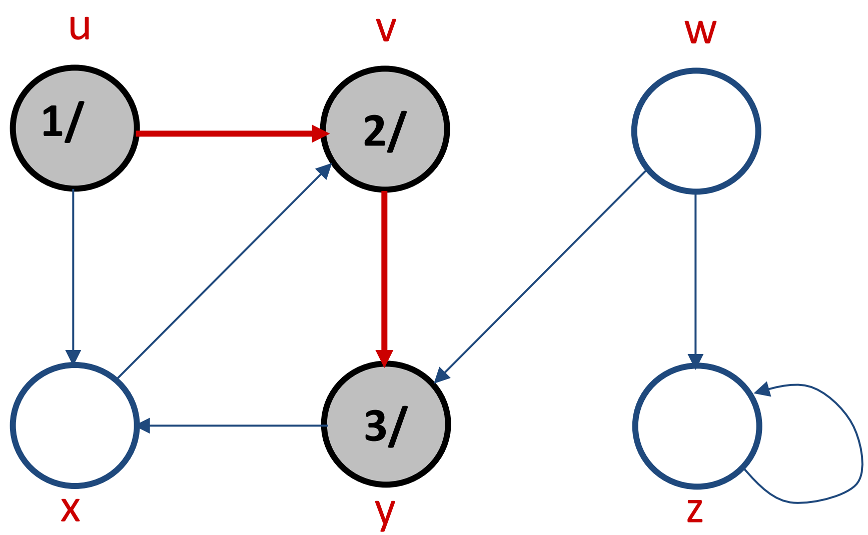
Example (DFS)



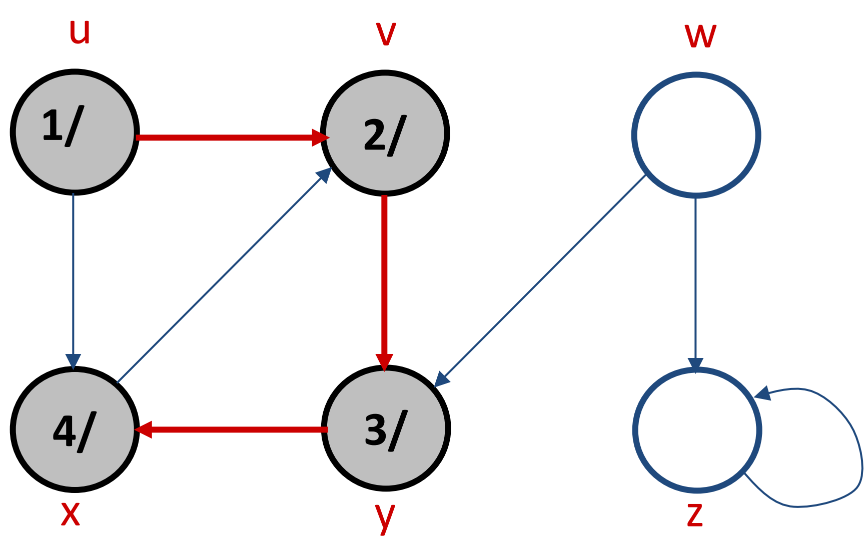
Example (DFS)



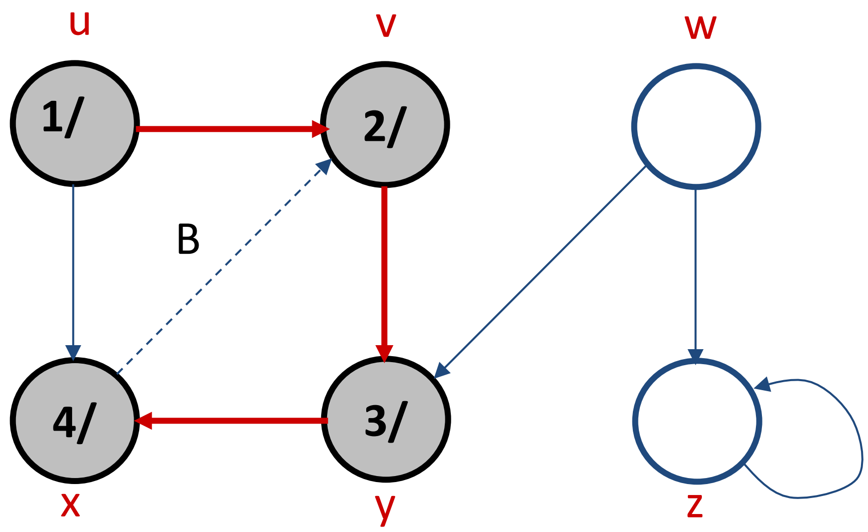
Example (DFS)



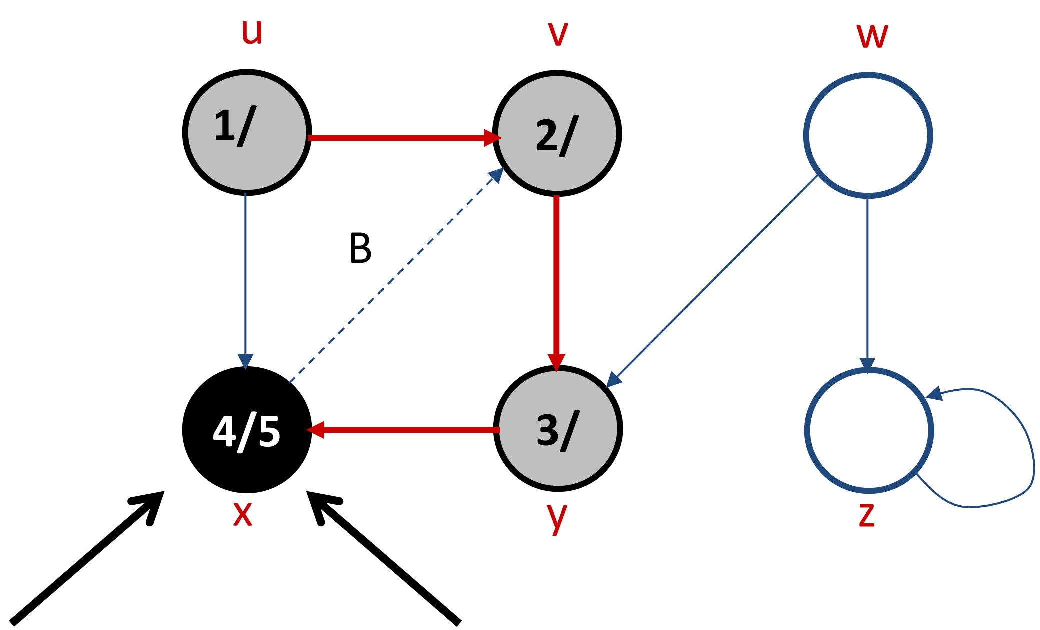
Example (DFS)



Example (DFS)



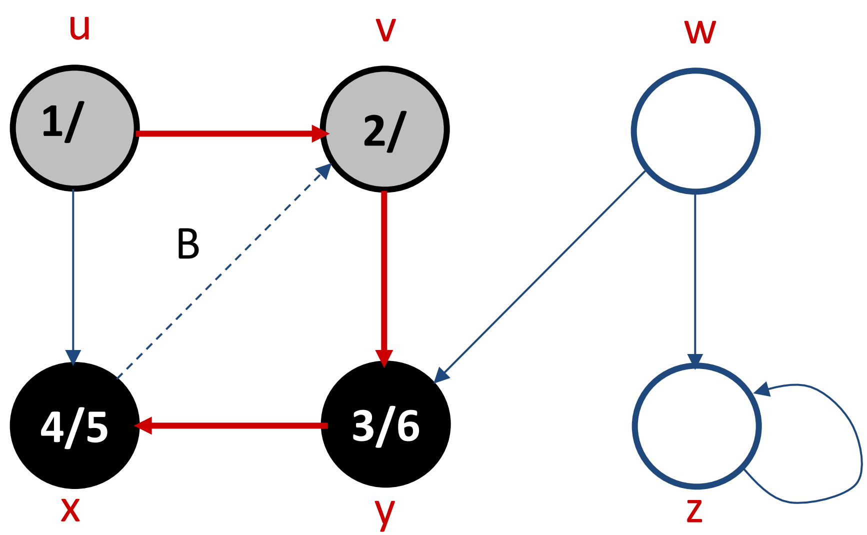
Example (DFS)



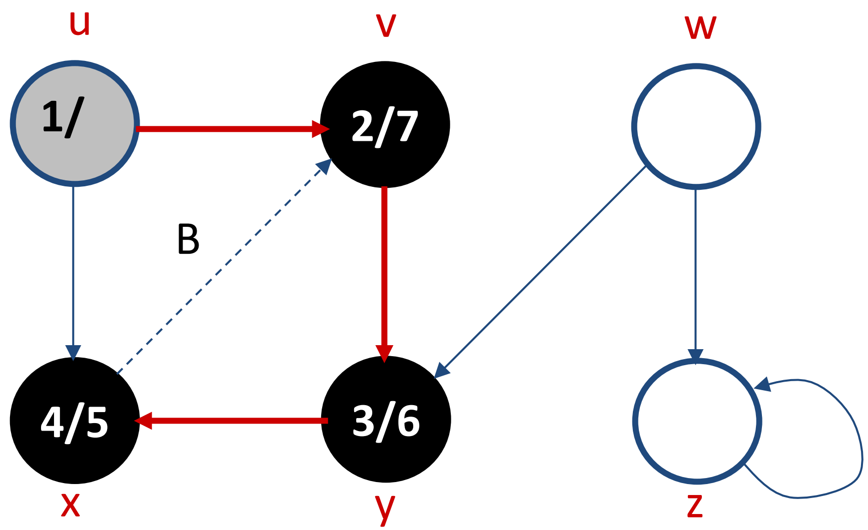
Starting time
d(x)

Finishing time
f(x)

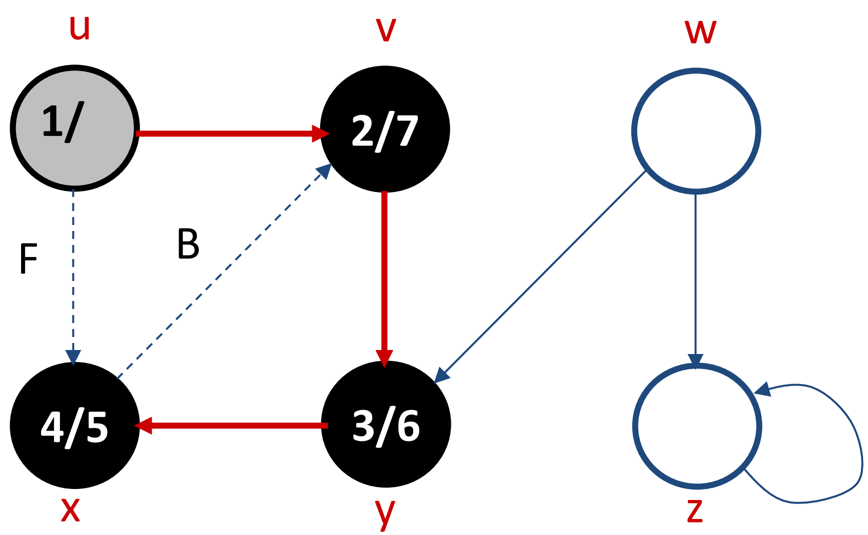
Example (DFS)



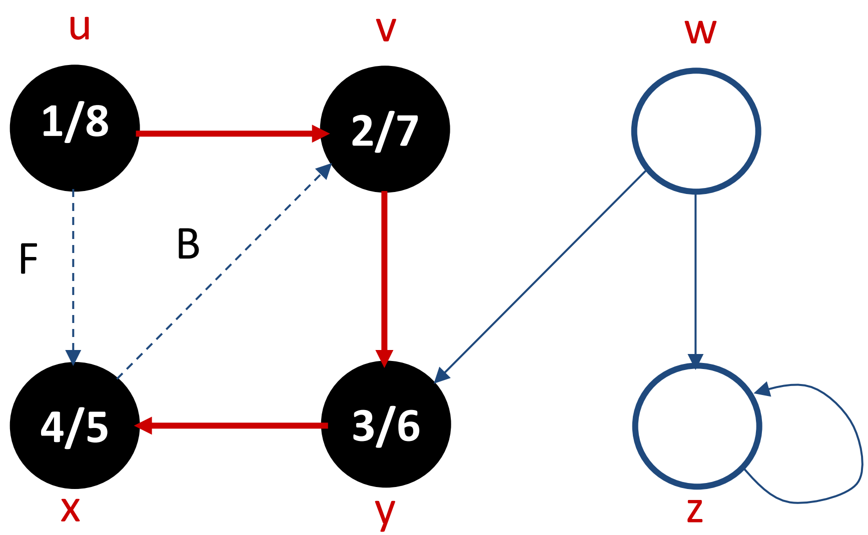
Example (DFS)



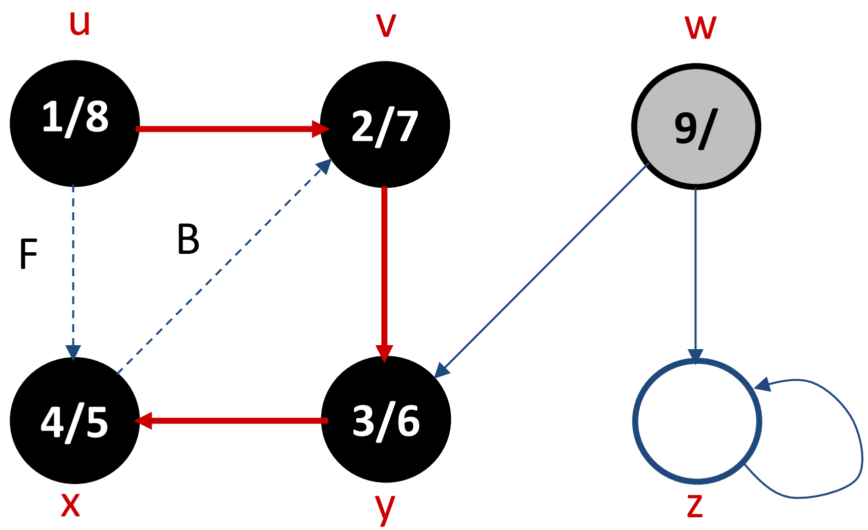
Example (DFS)



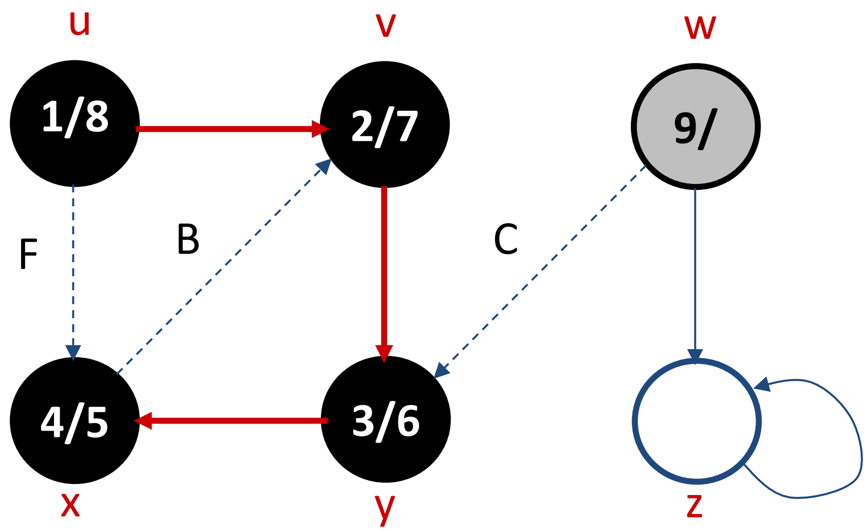
Example (DFS)



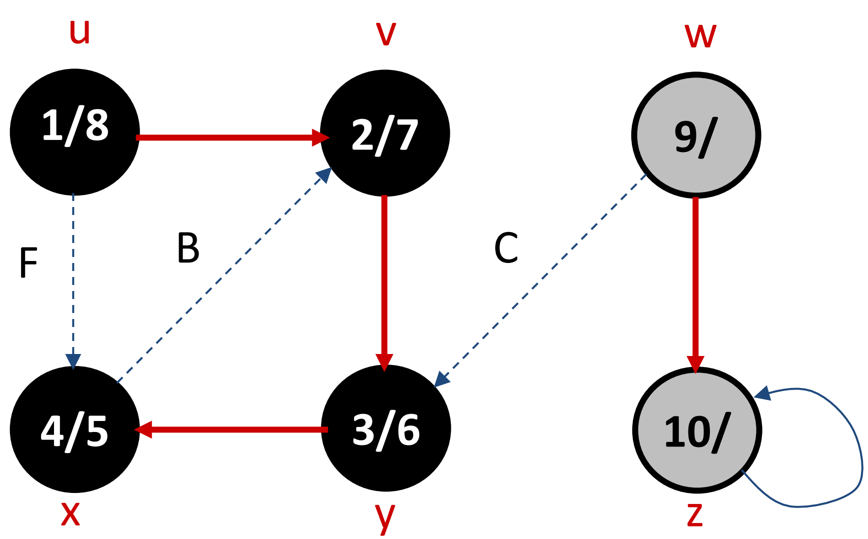
Example (DFS)



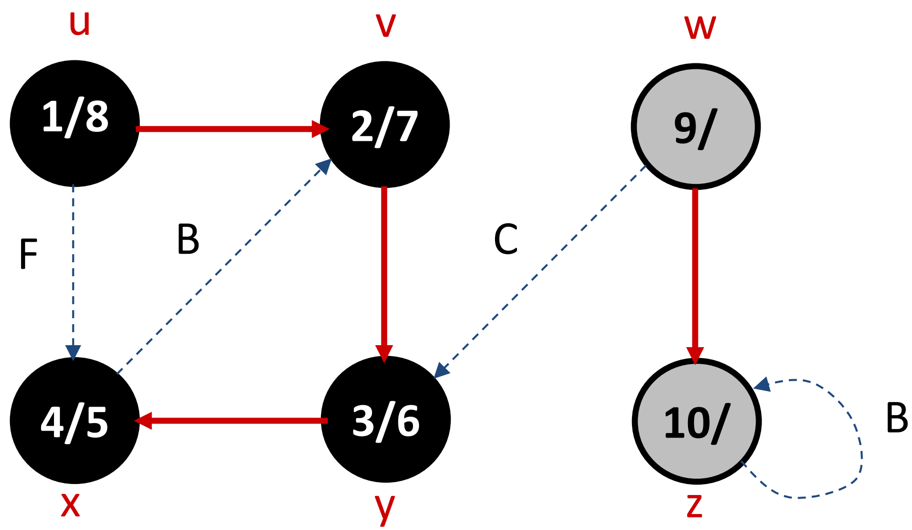
Example (DFS)



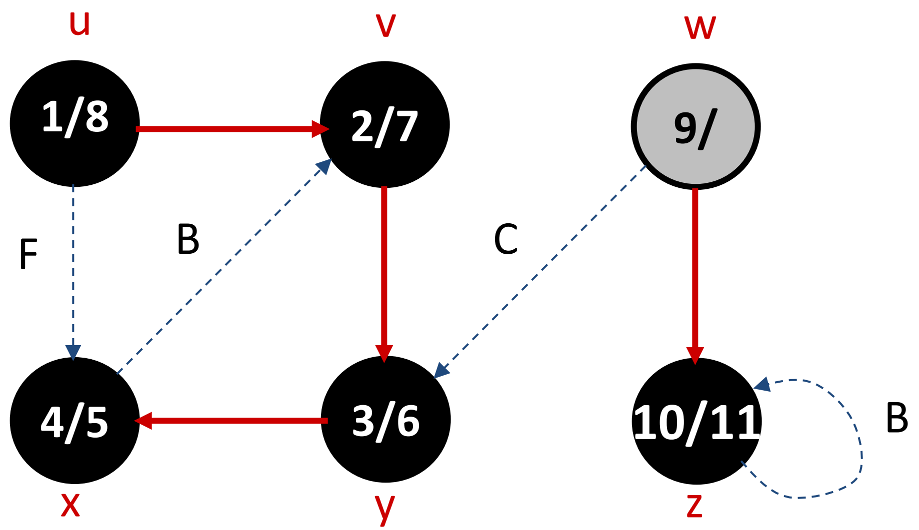
Example (DFS)



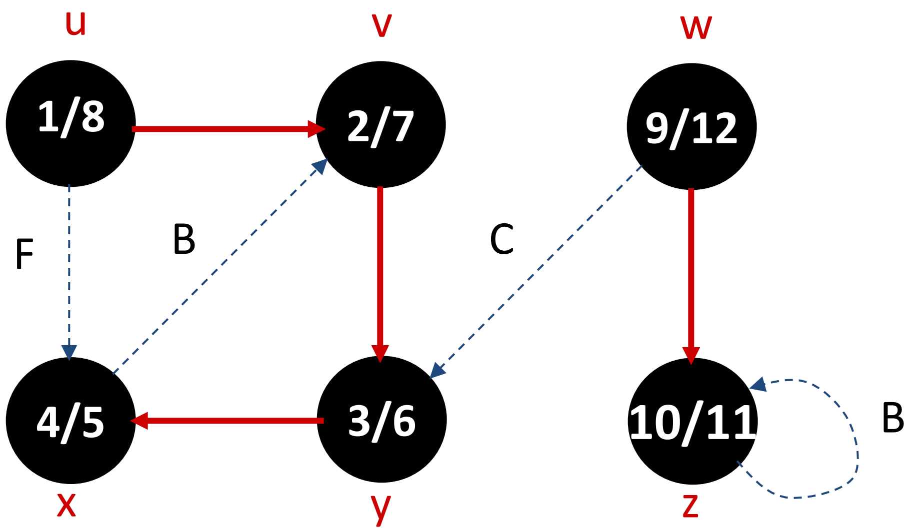
Example (DFS)



Example (DFS)



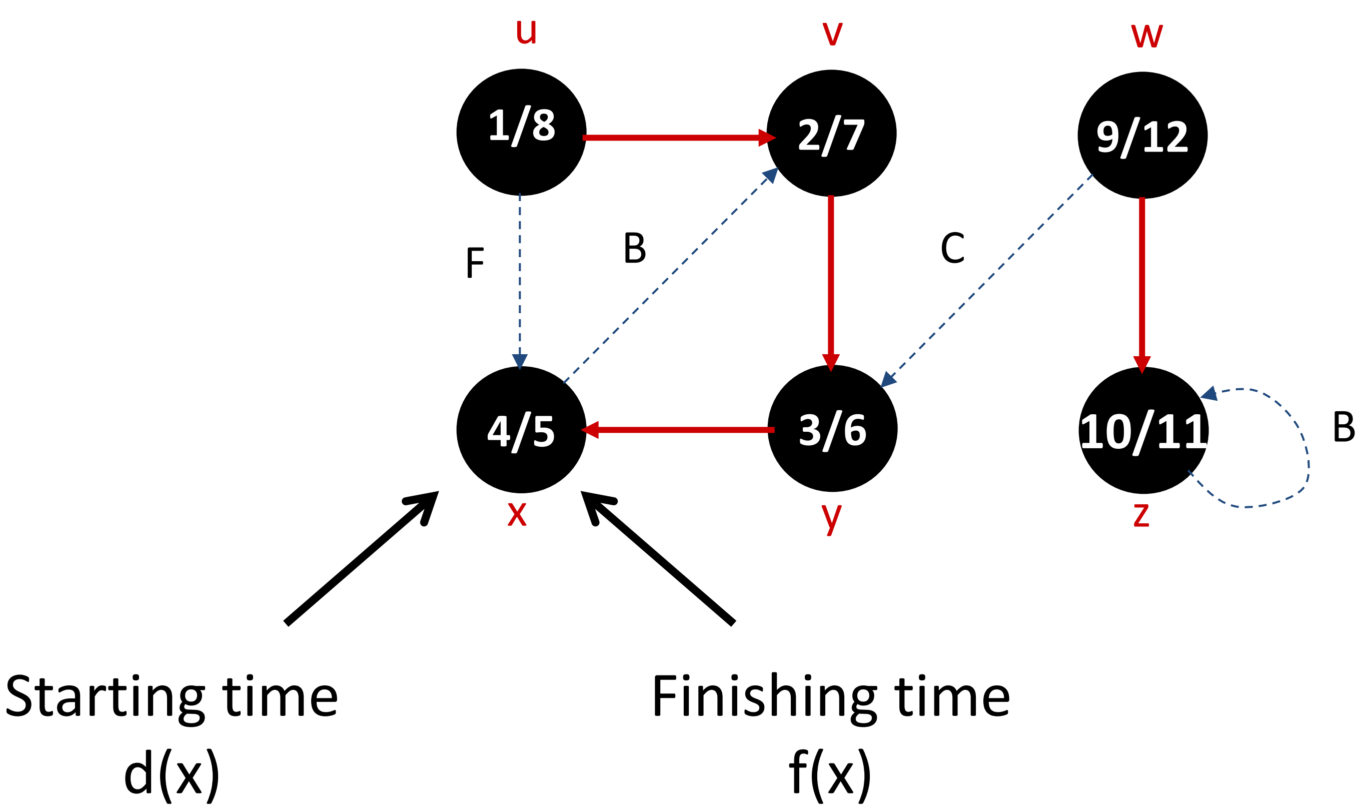
Example (DFS)



Analysis of DFS

- Loops on lines 1-2 & 5-7 take $\Theta(V)$ time, excluding time to execute DFS-Visit.
- DFS-Visit is called once for each white vertex $v \in V$ when it's painted gray the first time. Lines 3-6 of DFS-Visit is executed $|\text{Adj}[v]|$ times. The total cost of executing DFS-Visit is $\sum_{v \in V} |\text{Adj}[v]| = \Theta(E)$
- Total running time of DFS is $\Theta(V+E)$.

Example (DFS)


u
v
w
1/8
2/7
9/12
F
B
C
4/5
3/6
10/11
B
x
y
z
Starting time
d(x)
Finishing time
f(x)

Parenthesis Theorem

Theorem 1:

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

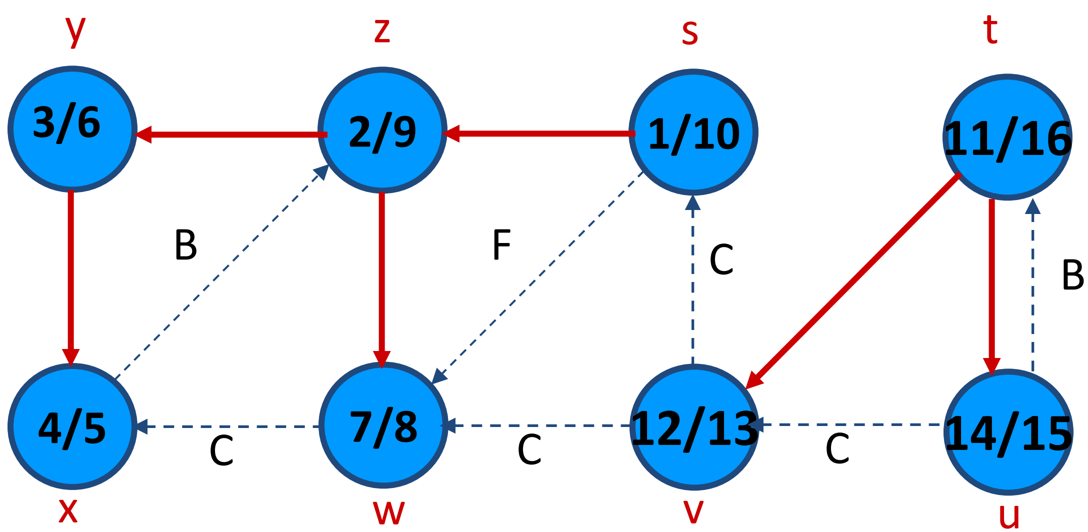
1. $d[u] < f[u] < d[v] < f[v]$ or $d[v] < f[v] < d[u] < f[u]$ and neither u nor v is a descendant of the other.
2. $d[u] < d[v] < f[v] < f[u]$ and v is a descendant of u.
3. $d[v] < d[u] < f[u] < f[v]$ and u is a descendant of v.

- 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 $d[u] < d[v] < f[v] < f[u]$.

Example (Parenthesis Theorem)



(s (z (y (x x) y) (w w) z) s) (t (v v) (u u) t)