CSEN 703 - Analysis and Design of Algorithms

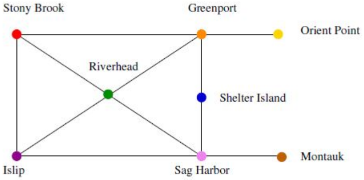
Lecture 8 - Graph Algorithms I

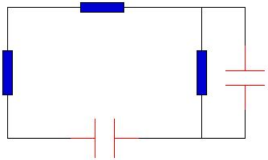
Dr. Nourhan Ehab
nourhan.ehab@guc.edu.eg

Department of Computer Science and Engineering
Faculty of Media Engineering and Technology

Motivation

- Graphs are one of the unifying themes of computer science.
- A graph $G = (V, E)$ consists of a set of vertices V together with a set E of edges.
- Graphs can be used to model any relationship from modelling roads, friendships, and electric circuits.





Motivation

- The key to solving many algorithmic problems is to think of them in terms of graphs.

Motivation

- The key to solving many algorithmic problems is to think of them in terms of graphs.
- The key to using graph algorithms effectively in applications lies in correctly modelling your problem so you can take advantage of existing algorithms.

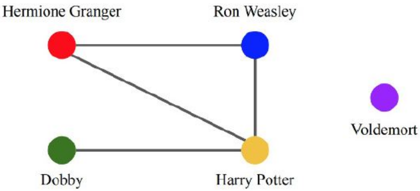
Outline

1. Graphs Modelling and Representation
2. Graph Traversal Algorithms
3. Traversal Applications
4. Recap

Flavours of Graphs

1. Undirected vs. Directed.
2. Weighted vs. Unweighted.
3. Sparse vs. Dense.
4. Embedded vs. Topological.
5. Implicit vs. Explicit.

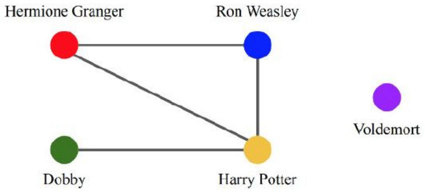
Example - Social Networks



Questions to ask:

1. If I am your friend, does that mean you are my friend?

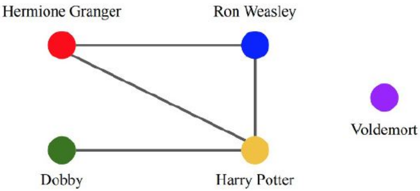
Example - Social Networks



Questions to ask:

1. If I am your friend, does that mean you are my friend?
2. How close a friend are you?

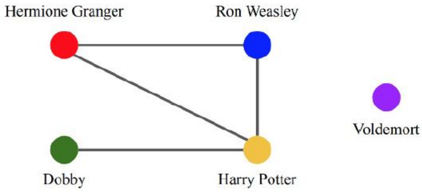
Example - Social Networks



Questions to ask:

1. If I am your friend, does that mean you are my friend?
2. How close a friend are you?
3. Who has the most friends?

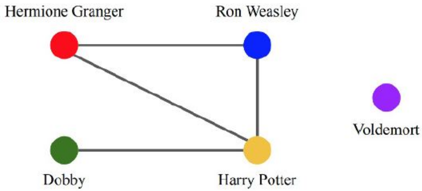
Example - Social Networks



Questions to ask:

1. If I am your friend, does that mean you are my friend?
2. How close a friend are you?
3. Who has the most friends?
4. Do my friends live near me?

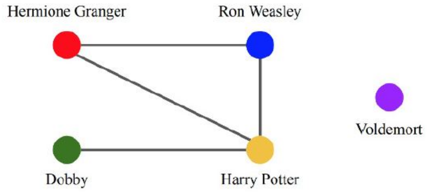
Example - Social Networks



Questions to ask:

1. If I am your friend, does that mean you are my friend?
2. How close a friend are you?
3. Who has the most friends?
4. Do my friends live near me?
5. Oh, you also know her?

Example - Social Networks



Questions to ask:

1. If I am your friend, does that mean you are my friend?
2. How close a friend are you?
3. Who has the most friends?
4. Do my friends live near me?
5. Oh, you also know her?

Graphs Modelling

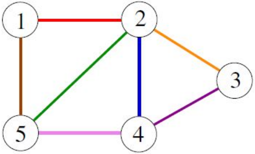
Key Takeaway

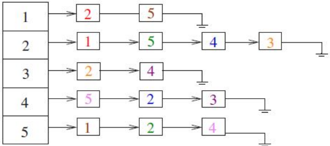
Graphs can be used to model a wide variety of structures and relationships. Graph-theoretic terminology gives us a language to talk about them.

Graph Data Structures

- Choosing the correct data structure to represent your graph affects the performance of algorithms operating on the graphs.
- Two choices: adjacency matrix and adjacency list.

	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





Space and Time Requirements

Space and Time Requirements

- An adjacency matrix requires a space of $\Theta(V^2)$.

Space and Time Requirements

- An adjacency matrix requires a space of $\Theta(V^2)$.
- If the graph is weighted, store the weight instead of 0,1.

Space and Time Requirements

- An adjacency matrix requires a space of $\Theta(V^2)$.
- If the graph is weighted, store the weight instead of 0,1.
- Time for basic operations:
  - List all elements adjacent to v:

Space and Time Requirements

- An adjacency matrix requires a space of $\Theta(V^2)$.
- If the graph is weighted, store the weight instead of 0,1.
- Time for basic operations:
  - List all elements adjacent to v: $\Theta(V)$

Space and Time Requirements

- An adjacency matrix requires a space of $\Theta(V^2)$.
- If the graph is weighted, store the weight instead of 0,1.
- Time for basic operations:
  - List all elements adjacent to v: $\Theta(V)$
  - Determine if u, v are connected:

Space and Time Requirements

- An adjacency matrix requires a space of $\Theta(V^2)$.
- If the graph is weighted, store the weight instead of 0,1.
- Time for basic operations:
  - List all elements adjacent to v: $\Theta(V)$
  - Determine if u, v are connected: $\Theta(1)$

Space and Time Requirements

- An adjacency matrix requires a space of $\Theta(V^2)$.
- If the graph is weighted, store the weight instead of 0,1.
- Time for basic operations:
  - List all elements adjacent to v: $\Theta(V)$
  - Determine if u, v are connected: $\Theta(1)$
- An adjacency list requires a space of $\Theta(V + E)$.

Space and Time Requirements

- An adjacency matrix requires a space of $\Theta(V^2)$.
- If the graph is weighted, store the weight instead of 0,1.
- Time for basic operations:
  - List all elements adjacent to v: $\Theta(V)$
  - Determine if u, v are connected: $\Theta(1)$
- An adjacency list requires a space of $\Theta(V + E)$.
- If the graph is weighted, store the weights in the list as well.

Space and Time Requirements

- An adjacency matrix requires a space of $\Theta(V^2)$.
- If the graph is weighted, store the weight instead of 0,1.
- Time for basic operations:
  - List all elements adjacent to v: $\Theta(V)$
  - Determine if u, v are connected: $\Theta(1)$
- An adjacency list requires a space of $\Theta(V + E)$.
- If the graph is weighted, store the weights in the list as well.
- Time for basic operations:
  - List all elements adjacent to v:

Space and Time Requirements

- An adjacency matrix requires a space of $\Theta(V^2)$.
- If the graph is weighted, store the weight instead of 0,1.
- Time for basic operations:
  - List all elements adjacent to v: $\Theta(V)$
  - Determine if u, v are connected: $\Theta(1)$
- An adjacency list requires a space of $\Theta(V + E)$.
- If the graph is weighted, store the weights in the list as well.
- Time for basic operations:
  - List all elements adjacent to v: $\Theta(degree(v))$

Space and Time Requirements

- An adjacency matrix requires a space of $\Theta(V^2)$.
- If the graph is weighted, store the weight instead of 0,1.
- Time for basic operations:
  - List all elements adjacent to v: $\Theta(V)$
  - Determine if u, v are connected: $\Theta(1)$
- An adjacency list requires a space of $\Theta(V + E)$.
- If the graph is weighted, store the weights in the list as well.
- Time for basic operations:
  - List all elements adjacent to v: $\Theta(degree(v))$
  - Determine if u, v are connected:

Space and Time Requirements

- An adjacency matrix requires a space of $\Theta(V^2)$.
- If the graph is weighted, store the weight instead of 0,1.
- Time for basic operations:
  - List all elements adjacent to v: $\Theta(V)$
  - Determine if u, v are connected: $\Theta(1)$
- An adjacency list requires a space of $\Theta(V + E)$.
- If the graph is weighted, store the weights in the list as well.
- Time for basic operations:
  - List all elements adjacent to v: $\Theta(degree(v))$
  - Determine if u, v are connected: $O(degree(u))$

Space and Time Requirements

- An adjacency matrix requires a space of $\Theta(V^2)$.
- If the graph is weighted, store the weight instead of 0,1.
- Time for basic operations:
  - List all elements adjacent to v: $\Theta(V)$
  - Determine if u, v are connected: $\Theta(1)$
- An adjacency list requires a space of $\Theta(V + E)$.
- If the graph is weighted, store the weights in the list as well.
- Time for basic operations:
  - List all elements adjacent to v: $\Theta(degree(v))$
  - Determine if u, v are connected: $O(degree(u))$

Adjacency Matrix or List?

Adjacency lists are be more efficient for sparse graphs, while matrices are be more efficient for dense graphs.

Adjacency Matrix or List?

Adjacency lists are be more efficient for sparse graphs, while matrices are be more efficient for dense graphs.

Key Takeaway

Since most problems are represented with sparse graphs, adjacency lists are the right data structure for most applications.

Outline

Graph Traversal

- The most fundamental graph problem is to visit all vertices and edges in a systemic way.

Graph Traversal

- The most fundamental graph problem is to visit all vertices and edges in a systemic way.
- Two types of traversals:
  1. Breadth-First Search (BFS).
  2. Depth-First Search (DFS).

Breadth-First Search

- Discover all vertices at depth k before the vertices at depth $k+1$.

Breadth-First Search

- Discover all vertices at depth k before the vertices at depth $k+1$.
- Input: A graph $G = (V, E)$ and a source vertex $s \in V$.

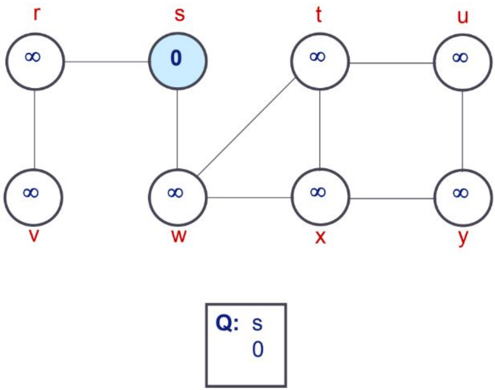
Breadth-First Search

- Discover all vertices at depth k before the vertices at depth $k+1$.
- Input: A graph $G = (V, E)$ and a source vertex $s \in V$.
- Output:
  1. $d[v]$ = number of edges from s to v, for all $v \in V$. $d[v] = \infty$ if v is not reachable from s.
  2. $\pi[v] = u$ such that (u, v) is last edge on the path $s \longrightarrow v$ (u is v's predecessor).
- BFS builds a tree with root s that contains all reachable vertices.

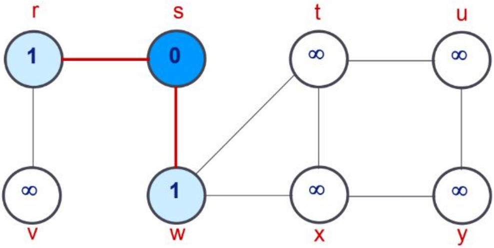
Breadth-First Search

- Discover all vertices at depth k before the vertices at depth $k+1$.
- Input: A graph $G = (V, E)$ and a source vertex $s \in V$.
- Output:
  1. $d[v]$ = number of edges from s to v, for all $v \in V$. $d[v] = \infty$ if v is not reachable from s.
  2. $\pi[v] = u$ such that (u, v) is last edge on the path $s \longrightarrow v$ (u is v's predecessor).
- BFS builds a tree with root s that contains all reachable vertices.
- If the graph is unweighted, the BFS tree can be used to find out the shortest path from s to all other vertices in the graph.

BFS Example

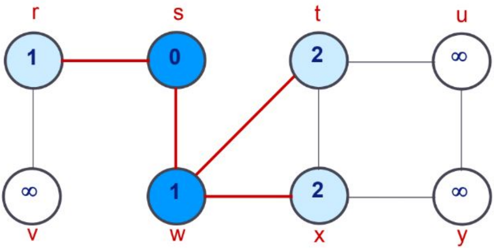


BFS Example



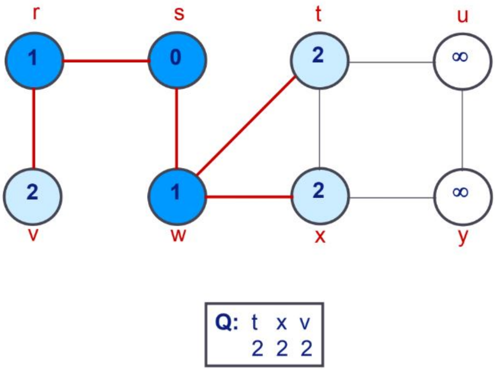
Q:	w	r
	1	1

BFS Example

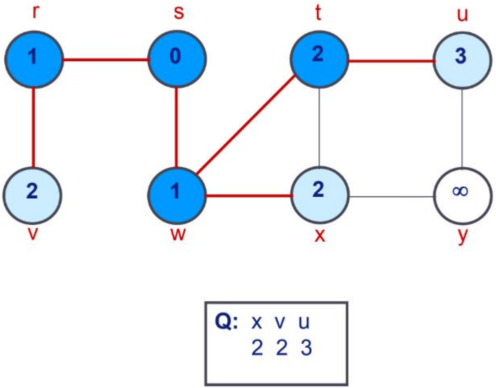


Q:	r	t	x
	1	2	2

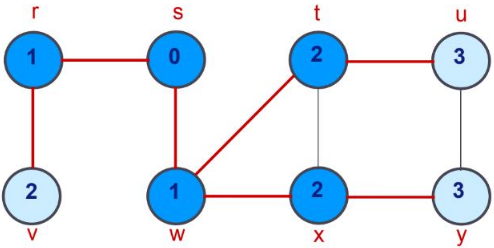
BFS Example



BFS Example

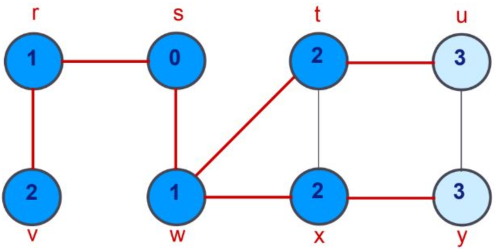


BFS Example



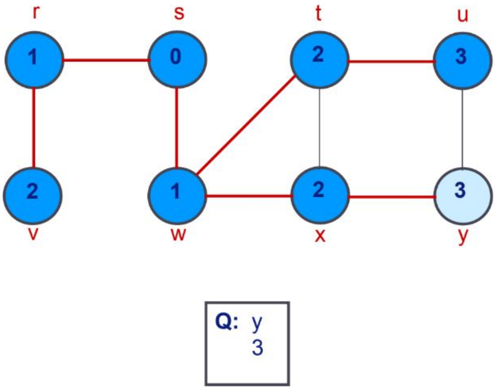
Q: v u y
2 3 3

BFS Example

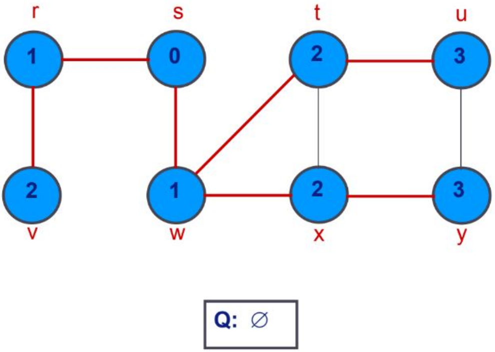


Q:	u	y
	3	3

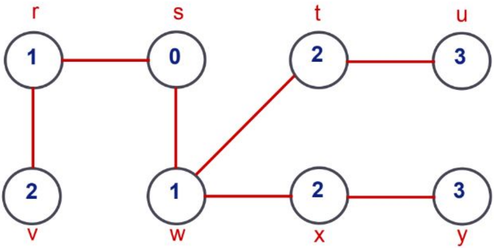
BFS Example



BFS Example



BFS Example



BF Tree

BFS Algorithm

```
BFS(G,s)
for each vertex u in V[G] – {s}
        do color[u] ← white
           d[u] ← ∞
           π[u] ← nil
color[s] ← gray
d[s] ← 0
π[s] ← nil
Q ← Φ
enqueue(Q,s)
while Q ≠ Φ
        do u ← dequeue(Q)
               for each v in 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
```

white: undiscovered
gray: discovered
black: finished

Q: a queue of discovered vertices
color[v]: color of v
d[v]: distance from s to v
π[u]: predecessor of v

BFS Analysis

- Initialization takes $O(V)$.

BFS Analysis

- 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 $O(E)$.

BFS Analysis

- 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 $O(E)$.
- Summing up over all vertices $\Rightarrow$ total running time of BFS is $O(V + E)$, linear in the size of the adjacency list representation of the graph.

Depth-First Search

- Expand deeper vertices first. When you hit a dead-end, backtrack.

Depth-First Search

- Expand deeper vertices first. When you hit a dead-end, backtrack.
- Input: A graph $G = (V, E)$ (no need for source vertex here).

Depth-First Search

- Expand deeper vertices first. When you hit a dead-end, backtrack.
- Input: A graph $G = (V, E)$ (no need for source vertex here).
- Output:
  1. $d[v]$ = discovery time for v, the time v turns gray.
  2. $f[v]$ = finishing time for v, the time v turns black.
  3. $\pi[v] = u$ where u is v's predecessor (v discovered during the scan of the adjacency list of u) .

Depth-First Search

GUC German University in Cairo

- Expand deeper vertices first. When you hit a dead-end, backtrack.
- Input: A graph $G = (V, E)$ (no need for source vertex here).
- Output:
  1. $d[v]$ = discovery time for v, the time v turns gray.
  2. $f[v]$ = finishing time for v, the time v turns black.
  3. $\pi[v] = u$ where u is v's predecessor (v discovered during the scan of the adjacency list of u) .
- DFS builds a forest of trees.

DFS Algorithm

DFS(G)

1. **for** each vertex $u \in V[G]$
2. **do** $color[u] \leftarrow$ white
3. $\pi[u] \leftarrow$ NIL
4. $time \leftarrow 0$
5. **for** each vertex $u \in V[G]$
6. **do if** $color[u]$ = white
7. **then** DFS-Visit(u)

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

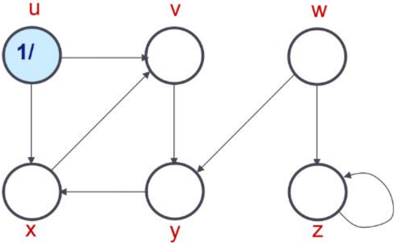
DFS-Visit(u)

1. $color[u] \leftarrow$ GRAY
2. $time \leftarrow time + 1$
3. $d[u] \leftarrow time$
4. **for** each $v \in Adj[u]$
5. **do if** $color[v]$ = WHITE
6. **then** $\pi[v] \leftarrow u$
7. DFS-Visit(v)
8. $color[u] \leftarrow$ BLACK
9. $f[u] \leftarrow time \leftarrow time + 1$

DFS Example

DFS-Visit(*u*)

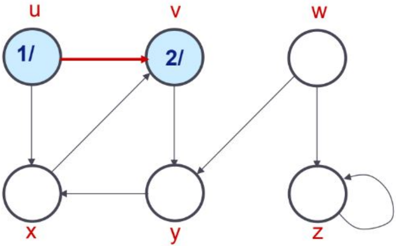
```
color[u] ← GRAY
time ← time + 1
d[u] ← time
for each v ∈ Adj[u]
    do if color[v] = WHITE
        then π[v] ← u
            DFS-Visit(v)
color[u] ← BLACK
f[u] ← time ← time + 1
```



DFS Example

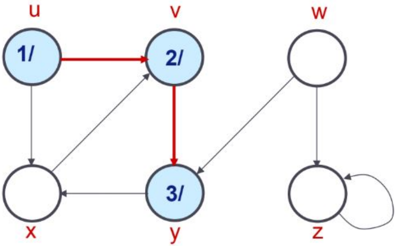
DFS-Visit(u)

1. $color[u] \leftarrow$ GRAY
2. $time \leftarrow time + 1$
3. $d[u] \leftarrow time$
4. **for each** $v \in Adj[u]$
5. **do if** $color[v] =$ WHITE
6. **then** $\pi[v] \leftarrow u$
7. DFS-Visit(v)
8. $color[u] \leftarrow$ BLACK
9. $f[u] \leftarrow time \leftarrow time + 1$



DFS Example

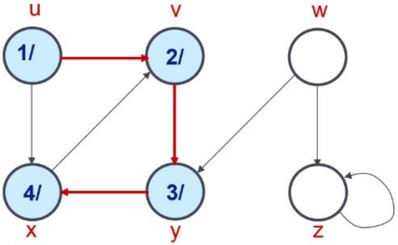
```
DFS-Visit(u)
    color[u] ← GRAY
    time ← time + 1
    d[u] ← time
    for each v ∈ Adj[u]
        do if color[v] = WHITE
            then π[v] ← u
                DFS-Visit(v)
    color[u] ← BLACK
    f[u] ← time ← time + 1
```



DFS Example

DFS-Visit(u)

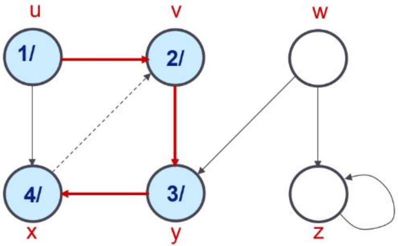
1. $color[u] \leftarrow$ GRAY
2. $time \leftarrow time + 1$
3. $d[u] \leftarrow time$
4. **for** each $v \in Adj[u]$
5. **do if** $color[v] =$ WHITE
6. **then** $\pi[v] \leftarrow u$
7. DFS-Visit(v)
8. $color[u] \leftarrow$ BLACK
9. $f[u] \leftarrow time \leftarrow time + 1$



DFS Example

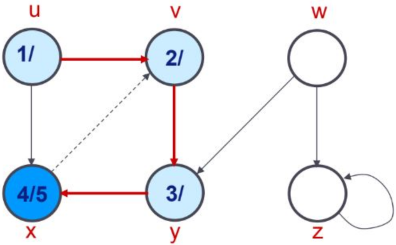
DFS-Visit(u)

1. $color[u] \leftarrow$ GRAY
2. $time \leftarrow time + 1$
3. $d[u] \leftarrow time$
4. **for** each $v \in Adj[u]$
5. **do if** $color[v] =$ WHITE
6. **then** $\pi[v] \leftarrow u$
7. DFS-Visit(v)
8. $color[u] \leftarrow$ BLACK
9. $f[u] \leftarrow time \leftarrow time + 1$



DFS Example

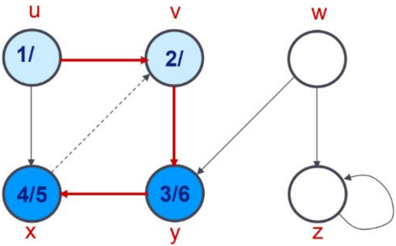
```
DFS-Visit(u)
    color[u] ← GRAY
    time ← time + 1
    d[u] ← time
    for each v ∈ Adj[u]
        do if color[v] = WHITE
            then π[v] ← u
                DFS-Visit(v)
    color[u] ← BLACK
    f[u] ← time ← time + 1
```



DFS Example

DFS-Visit(u)

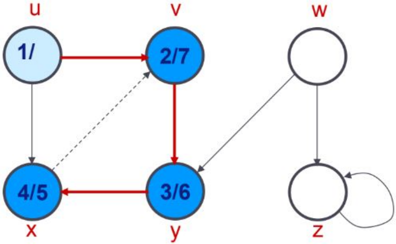
1. $color[u] \leftarrow$ GRAY
2. $time \leftarrow time + 1$
3. $d[u] \leftarrow time$
4. **for** each $v \in Adj[u]$
5. **do if** $color[v] =$ WHITE
6. **then** $\pi[v] \leftarrow u$
7. DFS-Visit(v)
8. $color[u] \leftarrow$ BLACK
9. $f[u] \leftarrow time \leftarrow time + 1$



DFS Example

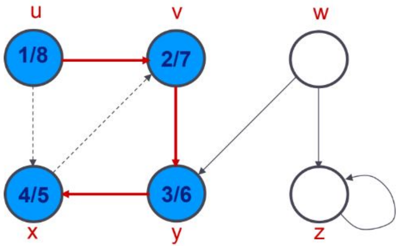
DFS-Visit(u)

1. $color[u] \leftarrow$ GRAY
2. $time \leftarrow time + 1$
3. $d[u] \leftarrow time$
4. **for each** $v \in Adj[u]$
5. **do if** $color[v] =$ WHITE
6. **then** $\pi[v] \leftarrow u$
7. DFS-Visit(v)
8. $color[u] \leftarrow$ BLACK
9. $f[u] \leftarrow time \leftarrow time + 1$



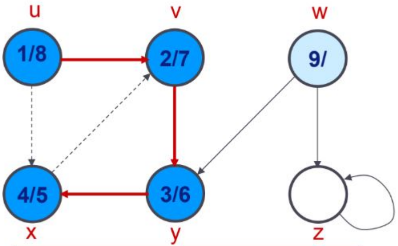
DFS Example

```
DFS-Visit(u)
    color[u] ← GRAY
    time ← time + 1
    d[u] ← time
    for each v ∈ Adj[u]
        do if color[v] = WHITE
            then π[v] ← u
                DFS-Visit(v)
    color[u] ← BLACK
    f[u] ← time ← time + 1
```



DFS Example

```
DFS-Visit(u)
    color[u] ← GRAY
    time ← time + 1
    d[u] ← time
    for each v ∈ Adj[u]
        do if color[v] = WHITE
            then π[v] ← u
                DFS-Visit(v)
    color[u] ← BLACK
    f[u] ← time ← time + 1
```

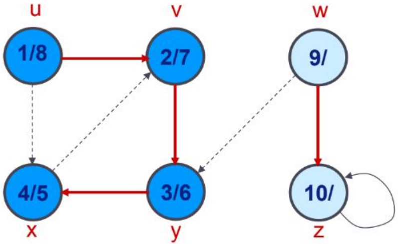



```
DFS(G)
for each vertex u ∈ V[G]
    do if color[u] = white
        then DFS-Visit(u)
```

DFS Example

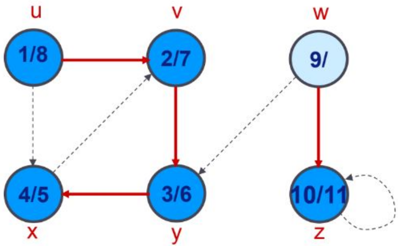
DFS-Visit(u)

```
color[u] ← GRAY
time ← time + 1
d[u] ← time
for each v ∈ Adj[u]
    do if color[v] = WHITE
        then π[v] ← u
            DFS-Visit(v)
color[u] ← BLACK
f[u] ← time ← time + 1
```



DFS Example

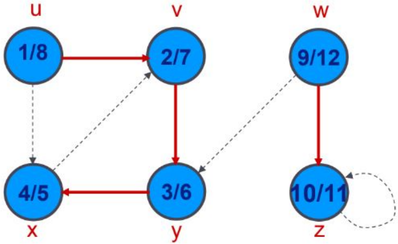
```
DFS-Visit(u)
    color[u] ← GRAY
    time ← time + 1
    d[u] ← time
    for each v ∈ Adj[u]
        do if color[v] = WHITE
            then π[v] ← u
                DFS-Visit(v)
    color[u] ← BLACK
    f[u] ← time ← time + 1
```



DFS Example

DFS-Visit(u)

```
color[u] ← GRAY
time ← time + 1
d[u] ← time
for each v ∈ Adj[u]
        do if color[v] = WHITE
                then π[v] ← u
                        DFS-Visit(v)
color[u] ← BLACK
f[u] ← time ← time + 1
```



DFS Analysis

- Loops on lines 1-3 5-7 take $O(V)$ time, excluding time to execute DFS-Visit.

DFS Analysis

- Loops on lines 1-3 5-7 take $O(V)$ time, excluding time to execute DFS-Visit.
- DFS-Visit is called once for each vertex $v \in V$.
  Lines 4-7 of DFS-Visit is executed $|Adj[v]|$ times. The total cost of executing DFS-Visit is $\Sigma_{v \in V}|Adj[v]| = O(E)$.

DFS Analysis

- Loops on lines 1-3 5-7 take $O(V)$ time, excluding time to execute DFS-Visit.
- DFS-Visit is called once for each vertex $v \in V$.
  Lines 4-7 of DFS-Visit is executed $|Adj[v]|$ times. The total cost of executing DFS-Visit is $\Sigma_{v \in V} |Adj[v]| = O(E)$.
- Total running time of DFS is $O(V + E)$. Still a linear in the size of the graph.

Outline

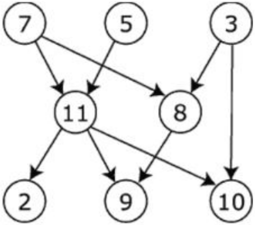
Applications of DFS

- We will look into three important applications of DFS.
  1. Topological Sorting.
  2. Strongly Connected Components.
  3. Minimum Spanning Trees. $\Rightarrow$ Next Lecture!

Topological Sorting

- Topological sorting takes a directed acyclic graph (DAG) as an input and returns a total order that respects the directions of the edges.
- Each vertex comes before all vertices to which it has edges. Every DAG has at least one topological sort, and may have many.

Output:
7,5,3,11,8,2,10,9
7,5,11,2,3,10,8,9
3,7,8,5,11,10,9,2

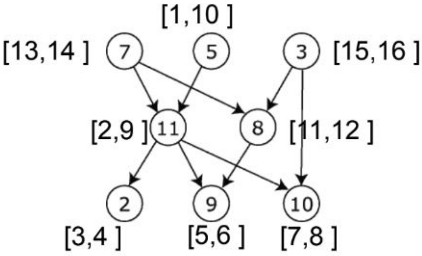


Topological Sorting Algorithm

TOPOLOGICAL-SORT(G)

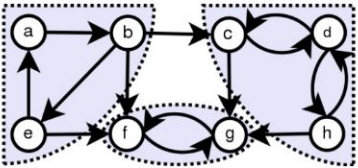
1 call DFS(G) to compute finishing times $v.f$ for each vertex v
2 as each vertex is finished, insert it onto the front of a linked list
3 **return** the linked list of vertices

Output by latest finishing time:
3,7,8,5,11,10,9,2



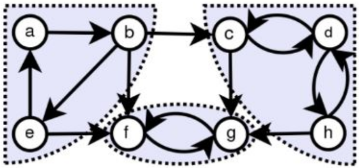
Strongly Connected Components

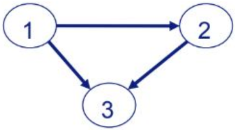
- A directed graph is called strongly connected if for every pair of vertices u and v there is a path from u to v and a path from v to u.
- A strongly connected component (SCC) of a directed graph is a maximal set of vertices $C \subseteq V$ such that for all $u, v \in C$, both $u \longrightarrow v$ and $v \longrightarrow u$.
- The set of all strongly connected components forms a partition of the graph.



Some Terminology

- The transpose of a directed graph G is G_T where G_T has the same vertices as G but with all edges reversed.
- We can compute G_T in $\Theta(V + E)$ if G is represented by an adjacency list.
- A component graph is a meta-graph where each component is represented by only one vertex.





SCC Algorithm

STRONGLY-CONNECTED-COMPONENTS(G)

1. call DFS(G) to compute finishing times $u.f$ for each vertex u
2. compute G^{T}
3. call DFS(G^{T}), but in the main loop of DFS, consider the vertices in order of decreasing $u.f$ (as computed in line 1)
4. output the vertices of each tree in the depth-first forest formed in line 3 as a separate strongly connected component

SCC Algorithm

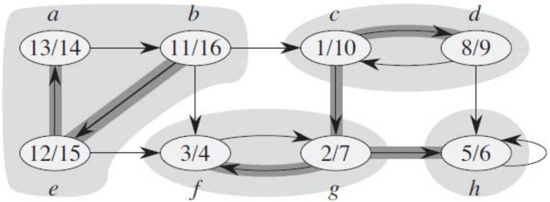
STRONGLY-CONNECTED-COMPONENTS(G)

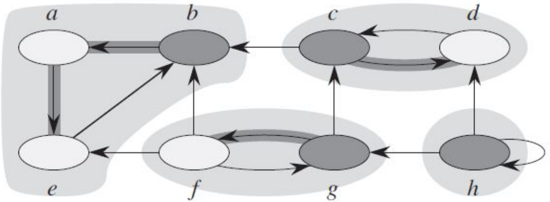
1 call DFS(G) to compute finishing times $u.f$ for each vertex u
2 compute G^{T}
3 call DFS(G^{T}), but in the main loop of DFS, consider the vertices in order of decreasing $u.f$ (as computed in line 1)
4 output the vertices of each tree in the depth-first forest formed in line 3 as a separate strongly connected component

Why does this work?

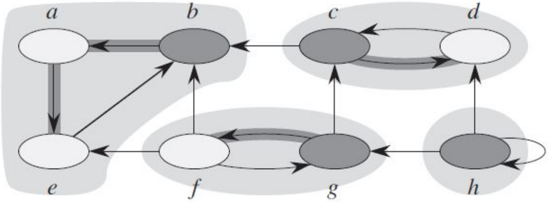
If u and v are in a SCC, then v must be reachable from u (established by the first DFS), and u must be reachable from v (established by the second DFS).

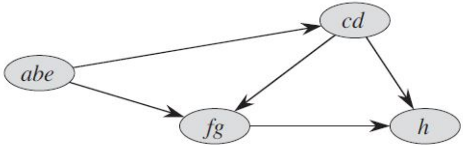
SCC Example





SCC Example





Outline

Points to Take Home

1. Graph Representations.
2. BFS and DFS.
3. Topological Sorting.
4. Strongly Connected Components.
5. Reading Material:
    - Introduction to Algorithms. Chapter 22, Sections 22.1, 22.2, 22.3, 22.4, 22.5.

Next Lecture: Graph Algorithms II!

Due Credits

The presented material is based on:

1. Previous editions of the course at the GUC due to Dr. Wael Aboulsaadat, Dr. Haythem Ismail, Dr. Amr Desouky, and Dr. Carmen Gervet.
2. Stony Brook University's Analysis of Algorithms Course.
3. MIT's Introduction to Algorithms Course.
4. Stanford's Design and Analysis of Algorithms Course.