Faculty of Information Technology,
Monash University

FIT2004: Algorithms and Data Structures

Week 4: Introduction to Graphs

Overview

Divide and conquer (W 1-3) | Greedy algorithms (W 4-5) | Dynamic programming (W 6-7) | Network flow (W 8-9) | Data structures (W 10-11)

- Today's lecture
  - Introduction to Graphs
  - Graph Traversal Algorithms
    - The idea
    - Breadth-First Search (BFS)
    - Depth-First Search (DFS)
    - Applications

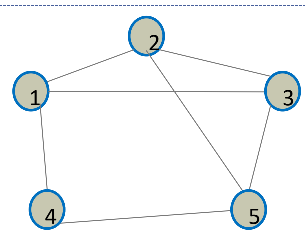
Outline

1. Introduction to Graphs
2. Graph Traversal Algorithms
   A. The idea
   B. Breadth-First Search (BFS)
   C. Depth-First Search (DFS)
   D. Applications

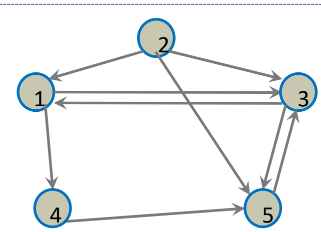
Graphs

- A graph is simply a way of encoding pairwise relationships among a set of objects.
- Each object is called a vertex or node.
- Each edge of the graph “connects” two nodes.

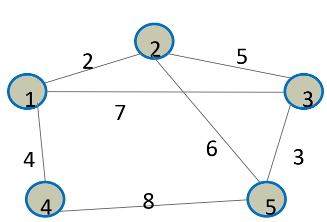
Graph - Examples



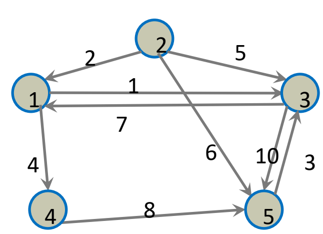
Undirected Graph



Directed Graph



Undirected Weighted Graph



Directed Weighted Graph

Uses of Graphs

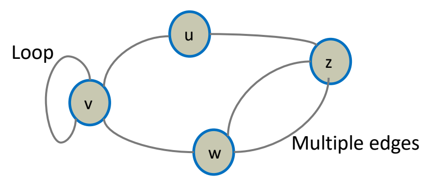
- Graphs are extremely useful for modelling. For example:
  - **Transportation networks**: map of routes of an airline, rail network, …
  - **Communication networks**: the connections between different Internet service providers, wireless ad-hoc networks, …
  - **Information networks**: the connections between different webpages using links (Google PageRank algorithm for determining the relative importance of each website), …
  - **Social networks**: the persons could be the nodes and the edges represent friendship (Facebook), or the nodes represent companies and people, and the edges financial relationships between them, etc. Properties of the graphs representing social networks are often used to find influencers, target ads,…
  - **Dependency networks**: prerequisites in a course map
  - …

Graphs – Formal notations

- A graph G = (V, E) is defined using a set of vertices V and a set of edges E.
- An edge e is represented as e = (u, v) where u and v are two vertices
- For undirected graphs, **(u, v) = (v, u)** because there is no sense of direction. For a directed graph, (u, v) represents an edge **from** u **to** v and (u, v) ≠ (v, u).
- We will slightly abuse notation and use V (instead of |V|) for the number of vertices and E (instead of |E|) for the number of edges when what is meant is clear from the context.

Graphs – Formal notations

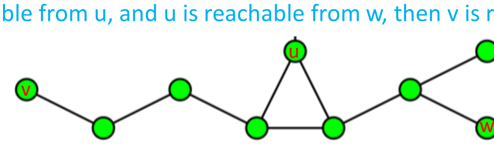
- A weighted graph is represented as G = (V, E) and each edge (u, v) has an associated weight w.
- A graph is called a **simple graph** if it does not have loops AND does not contain multiple edges between same pair of vertices.



- **In this unit, we focus on simple graphs with a finite number of vertices.**

Graphs – Connected Components

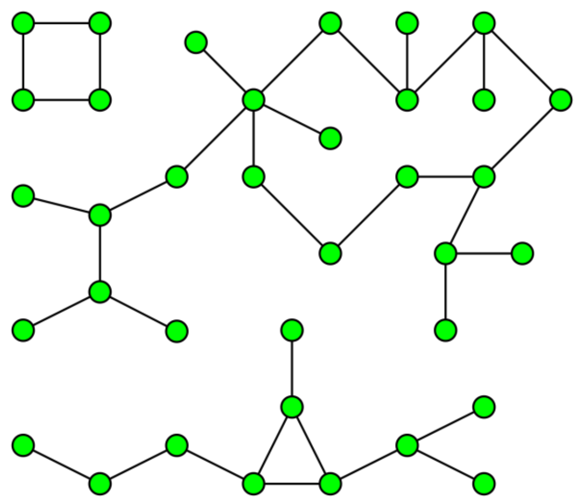
- A vertex v is **reachable** from u if there is a path in the graph that starts in u and ends v.
- In an undirected graph, reachability is an equivalence relation:
  - Reflexive: each u node is reachable from itself.
  - Symmetric: if v is reachable from u, then u is reachable from v.
  - Transitive: if v is reachable from u, and u is reachable from w, then v is reachable from w.



- The set of **vertices reachable from u** defines the **connected component** of G containing u.
- For any two nodes u and v, their connected components are either identical or disjoint.

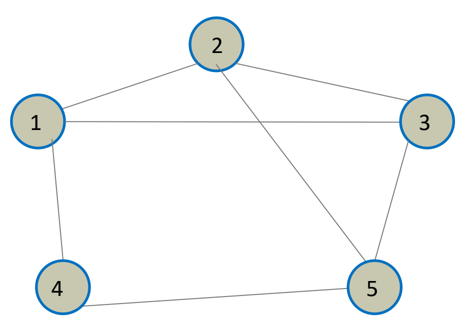
Graphs – Connected Components

- An undirected graph with 3 connected components:



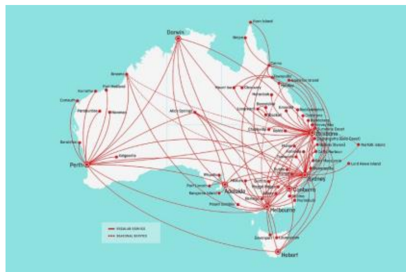
Graphs – Connected Components

- An undirected graph is **connected** if all vertices are part of a single connected component.
- In other words, **for any pair of vertices u and v, there is a path between them.**



Graphs – Connected Components

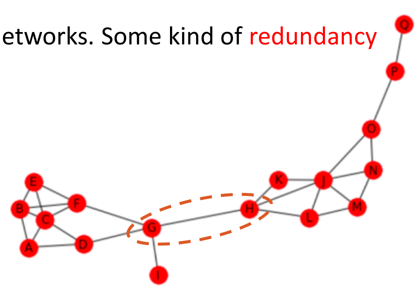
- Why is that an important concept in practice?
  - Example: Airlines normally want their air routes to form a connected graph.



  - Getting a connected graph is often an important consideration when designing communication and transportation networks.
- Hubs: Often it is not viable to **have pairwise connections between all nodes**; but one still wants to have paths, without many intermediary nodes, between every pair of nodes.

Graphs – Connected Components

- In this graph the edge (g, h) is quite critical as any problem in the network that eliminates this edge would break the connected graph into "large" disjoint connected components.
- This can be quite bad in networks. Some kind of redundancy is often desirable.



- **Adding edges that join distinct connected components** can sometimes also have bad consequences. E.g., quarantine measures often try to avoid a disease from reaching a disease-free connected component of a social network.

Some Graph Properties

Let G be a graph.

- The minimum number of edges in a **connected undirected** graph
  - ???
- The maximum number of edges in an **undirected** graph
  - ???

Some Graph Properties

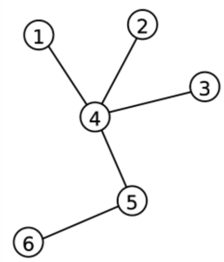
Let G be a graph.

- The minimum number of edges in a connected undirected graph
  - $V-1 = O(V)$
- The maximum number edges in an undirected graph
  - $V(V - 1)/2 = O(V^2)$

- A graph is called **sparse** if $E \ll V^2$ (<< means significantly smaller than)
- A graph is called **dense** if $E \approx V^2$

Tree

- Let G=(V, E) be an undirected graph. G is a tree if it satisfies any of the following equivalent conditions:
  - G is connected and acyclic (i.e., contains no cycles).
  - G is connected and has V-1 edges.
  - G is acyclic and has V-1 edges.
  - G is acyclic, but a cycle is formed if any edge is added to G.
  - G is connected, but would become disconnected if any single edge is removed from G.



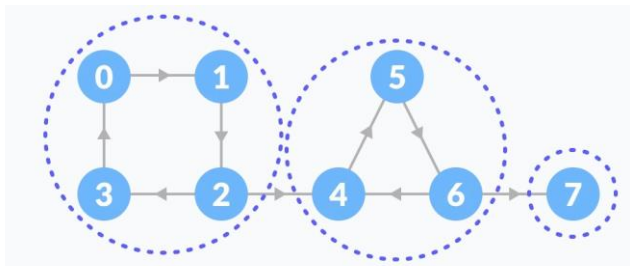
- In other words, if any of the above conditions is satisfied for an undirected graph G, then G is a tree (and all the other conditions will also hold).

Graphs – Connected Components

- In directed graphs, reachability is reflexive and transitive, but not guaranteed to be symmetric (i.e., possibly there could be a path from u to v, but no path from v to u).
- Vertices u and v are called **mutually reachable** if there are paths from u to v and from v to u.
- Mutual reachability is an equivalence relation and decomposes the graph into **strongly-connected components** (for any two vertices u and v, their strong components are either identical or disjoint).

Graphs – Connected Components

A directed graph with 3 strongly-connected components:



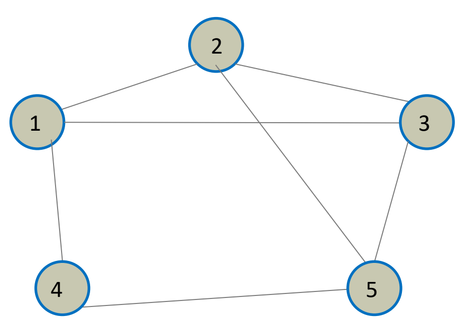
- A directed graph is **strongly connected** if for every pair of vertices u and v of G, there are paths from u to v and from v to u.
- I.e., the graph only has one strongly-connected component.

Representing Graphs

Adjacency Matrix (Undirected Graph):

Create a V x V matrix M and store T (true) for M[i][j] if there exists an edge between i-th and j-th vertex. Otherwise, store F (false).

	1	2	3	4	5
1	F	T	T	T	F
2	T	F	T	F	T
3	T	T	F	F	T
4	T	F	F	F	T
5	F	T	T	T	F

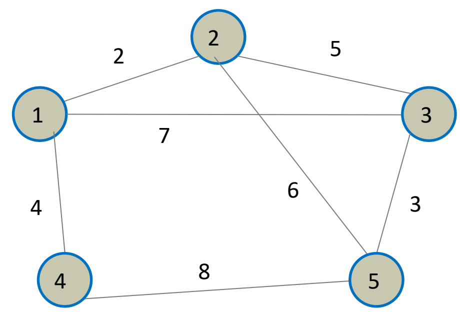


Representing Graphs

Adjacency Matrix (Undirected Weighted Graph):

Create a V x V matrix M and store **weight** at M[i][j] only if there exists an edge **between** i-th and j-th vertex.

	1	2	3	4	5
1		2	7	4	
2	2		5		6
3	7	5			3
4	4				8
5		6	3	8	



Representing Graphs

Adjacency Matrix (Directed Weighted Graph):

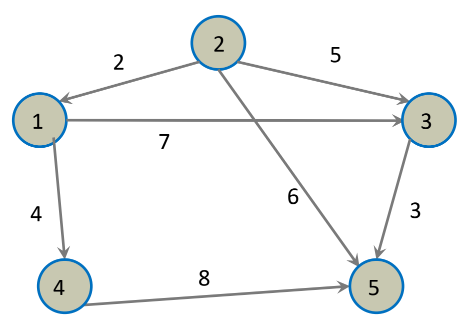
Create a V x V matrix M and store weight at M[i][j] only if there exists an edge **from** i-th **to** j-th vertex.

Space Complexity: $O(V^2)$ regardless of the number of edges

Time Complexity of checking if an edge exits: O(1)

Time Complexity of retrieving all neigbhbors (adjacent vertices) of a given vertex:
O(V) regardless of the number of neighbors (unless additional pointers are stored)

	1	2	3	4	5
1			7	4	
2	2		5		6
3					3
4					8
5					

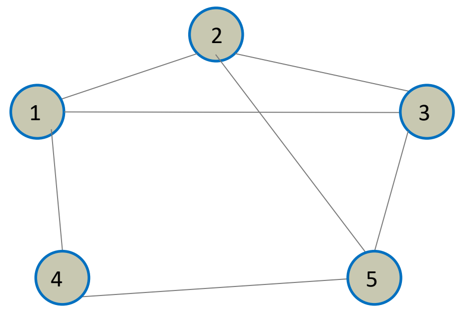


Representing Graphs

Adjacency List (Undirected Graph):

Create an array of size V. At each V[i], store the list of vertices adjacent to the i-th vertex.

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



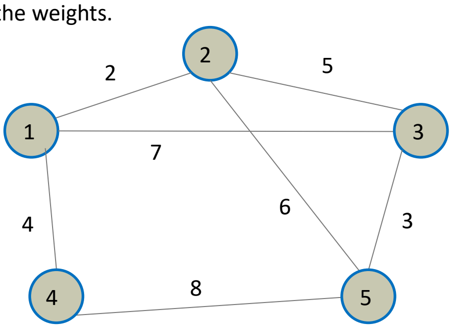
Representing Graphs

Adjacency List (Undirected Weighted Graph):

Create an array of size V. At each V[i], store the list of vertices adjacent to the i-th vertex **along with the weights**.

The numbers in parentheses correspond to the weights.

1	2 (2)	3 (7)	4 (4)
2	1 (2)	3 (5)	5 (6)
3	1 (7)	2 (5)	5 (3)
4	1 (4)	5 (8)	
5	2 (6)	3 (3)	4 (8)



Representing Graphs

Adjacency List (Directed Weighted Graph):

Create an array of size V. At each V[i], store the list of vertices adjacent to the i-th vertex **along with the weights**.

Space Complexity:

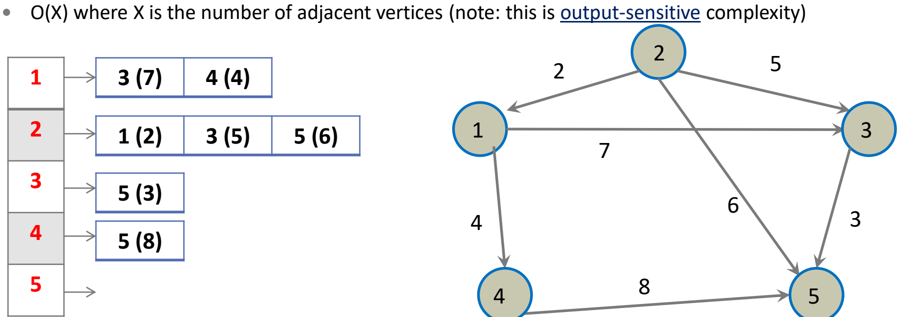
- O(V + E)

Time complexity of checking if a particular edge exists:

- O(log V) assuming each adjacency list is a sorted array on vertex IDs

Time complexity of retrieving all adjacent vertices of a given vertex:

- O(X) where X is the number of adjacent vertices (note: this is output-sensitive complexity)



Outline

1. Introduction to Graphs
2. Graph Traversal Algorithms
   A. The idea
   B. Breadth-First Search (BFS)
   C. Depth-First Search (DFS)
   D. Applications

Graph Traversal

Graph traversal algorithms traverse (visit) all nodes of a graph.

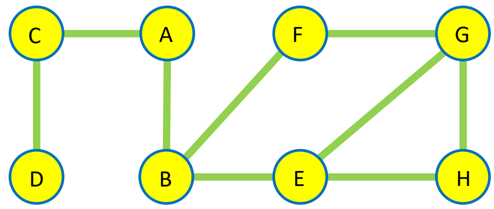
They are very important in the design of numerous algorithms.

We will **look into two algorithms** that traverse a connected component from a graph starting from a source vertex:

- Breadth-First Search (BFS)
- Depth-First Search (DFS)

Both of them visit the vertices exactly once.

They visit vertices in different orders.

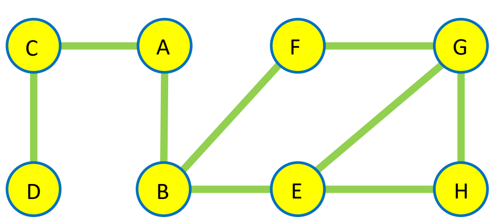


If a graph has more than one connected component, they can be repeatedly called (on unvisited nodes) until all graph nodes are marked as visited.

Each one has properties that makes it useful for certain kinds of graph problems.

Graph Traversal - BFS

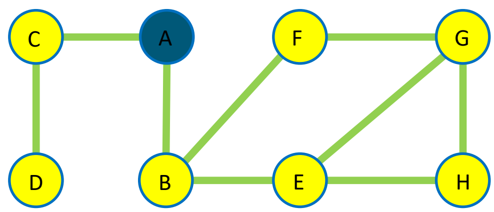
- Breadth-First Search (BFS)
  - Traverses the graph uniformly from the source vertex
  - i.e., all vertices that are **k edges away** from the source vertex are visited before all vertices that are **k+1 edges away** from source
  - In the graph below, if A is the source, then one possible BFS order is:
  - A, C, B, D, E, F, G, H



Konrad Zuse, computer science pioneer

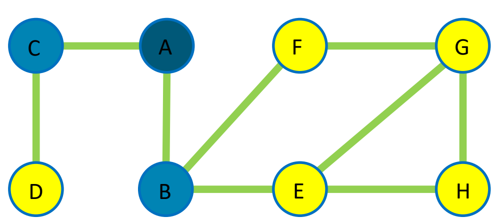
Graph Traversal - BFS

- Breadth-First Search (BFS)
  - Traverses the graph uniformly from the source vertex
  - i.e., all vertices that are **k edges away** from the source vertex are visited before all vertices that are **k+1 edges away** from source
  - In the graph below, if A is the source, then one possible BFS order is:
  - A, C, B, D, E, F, G, H



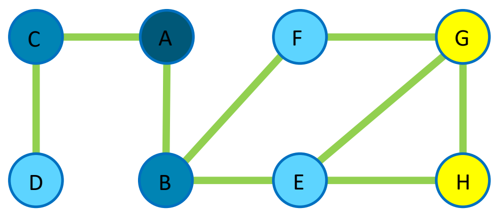
Graph Traversal - BFS

- Breadth-First Search (BFS)
  - Traverses the graph uniformly from the source vertex
  - i.e., all vertices that are **k edges away** from the source vertex are visited before all vertices that are **k+1 edges away** from source
  - In the graph below, if A is the source, then one possible BFS order is:
  - A, C, B, D, E, F, G, H



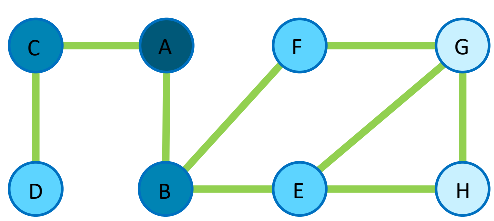
Graph Traversal - BFS

- Breadth-First Search (BFS)
  - Traverses the graph uniformly from the source vertex
  - i.e., all vertices that are **k edges away** from the source vertex are visited before all vertices that are **k+1 edges away** from source
  - In the graph below, if A is the source, then one possible BFS order is:
  - A, C, B, D, E, F, G, H



Graph Traversal - BFS

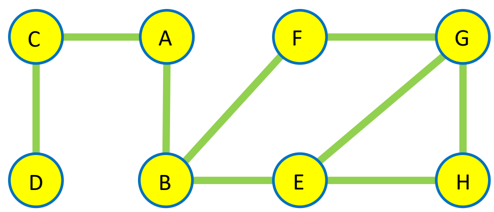
- Breadth-First Search (BFS)
  - Traverses the graph uniformly from the source vertex
  - i.e., all vertices that are **k edges away** from the source vertex are visited before all vertices that are **k+1 edges away** from source
  - In the graph below, if A is the source, then one possible BFS order is:
  - A, C, B, D, E, F, G, H



Graph Traversal - DFS

- Depth-First Search (DFS)
  - A version of DFS was investigated by the 19th century French mathematician Charles Pierre Trémaux to solve mazes.
  - Traverses the graph as deeply as possible before backtracking and traversing other nodes
  - In the graph below, one possible DFS order is: A, B, F, G, H, E, C, D

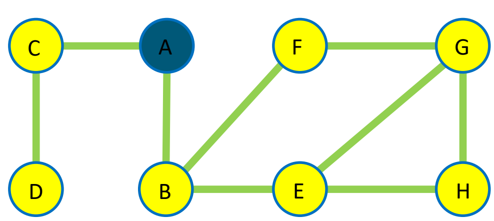
Is A, B, E, H, F, G, C, D a possible DFS order?



Graph Traversal - DFS

- Depth-First Search (DFS)
  - Traverses the graph as deeply as possible before backtracking and traversing other nodes
  - In the tree, one possible DFS order is: A, B, F, G, H, E, C, D

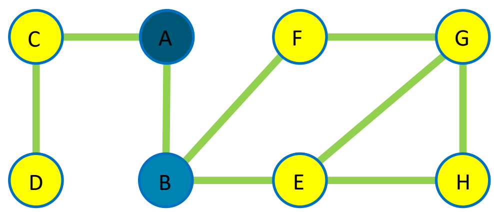
Is A, B, E, H, F, G, C, D as possible DFS order?
No!



Graph Traversal - DFS

- Depth-First Search (DFS)
  - Traverses the graph as deeply as possible before backtracking and traversing other nodes
  - In the tree, one possible DFS order is: A, B, F, G, H, E, C, D

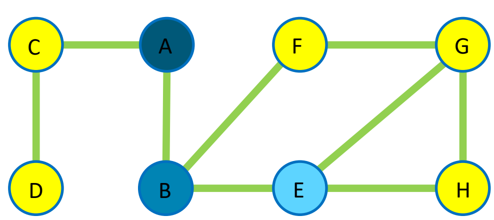
Is A, B, E, H, F, G, C, D as possible DFS order?
No!



Graph Traversal - DFS

- Depth-First Search (DFS)
  - Traverses the graph as deeply as possible before backtracking and traversing other nodes
  - In the tree, one possible DFS order is: A, B, F, G, H, E, C, D

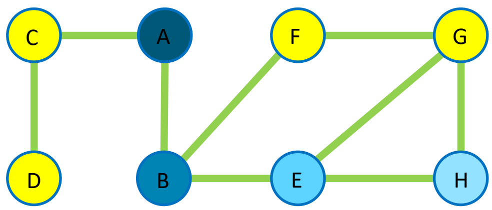
Is A, B, E, H, F, G, C, D as possible DFS order?
No!



Graph Traversal - DFS

- Depth-First Search (DFS)
  - Traverses the graph as deeply as possible before backtracking and traversing other nodes
  - In the tree, one possible DFS order is: A, B, F, G, H, E, C, D

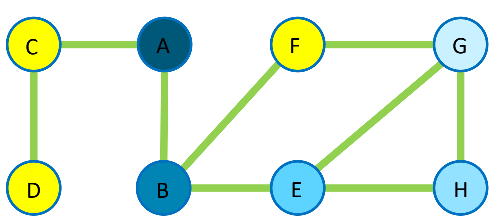
Is A, B, E, H, F, G, C, D as possible DFS order?
No!



Graph Traversal - DFS

- Depth-First Search (DFS)
  - Traverses the graph as deeply as possible before backtracking and traversing other nodes
  - In the tree, one possible DFS order is: A, B, F, G, H, E, C, D

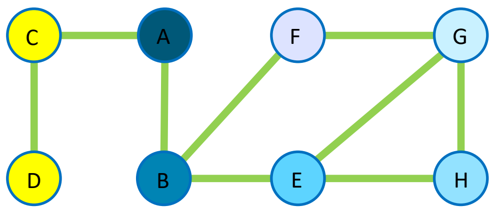
Is A, B, E, H, F, G, C, D as possible DFS order?
No!



Graph Traversal - DFS

- Depth-First Search (DFS)
  - Traverses the graph as deeply as possible before backtracking and traversing other nodes
  - In the tree, one possible DFS order is: A, B, F, G, H, E, C, D

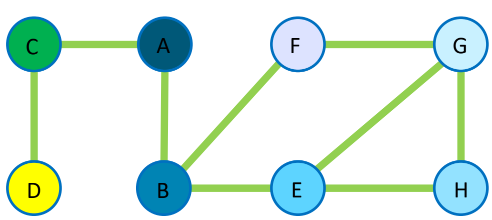
Is A, B, E, H, F, G, C, D as possible DFS order?
No!



Graph Traversal - DFS

- Depth-First Search (DFS)
  - Traverses the graph as deeply as possible before backtracking and traversing other nodes
  - In the tree, one possible DFS order is: A, B, F, G, H, E, C, D

Is A, B, E, H, F, G, C, D as possible DFS order?
No!

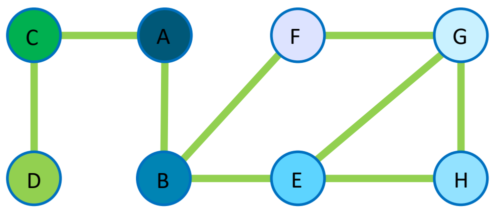


FIT2004: Seminar 4 - Introduction to Graphs

Graph Traversal - DFS

- Depth-First Search (DFS)
  - Traverses the graph as deeply as possible before backtracking and traversing other nodes
  - In the tree, one possible DFS order is: A, B, F, G, H, E, C, D

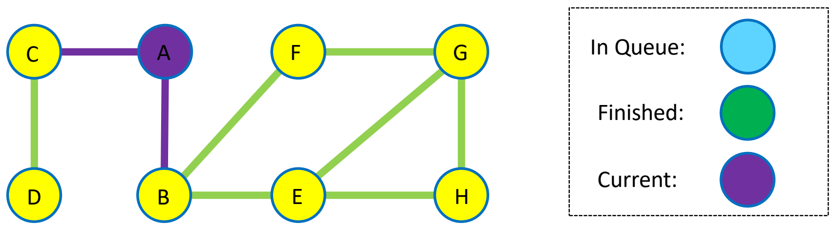
Is A, B, E, H, F, G, C, D as possible DFS order?
No!



Outline

1. Introduction to Graphs
2. Graph Traversal Algorithms
   A. The idea
   B. Breadth-First Search (BFS)
   C. Depth-First Search (DFS)
   D. Applications

Breadth-First Search (BFS)

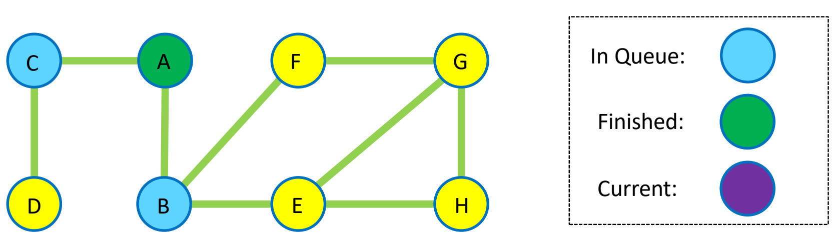


Current: A

Queue:

Finished:

Breadth-First Search (BFS)



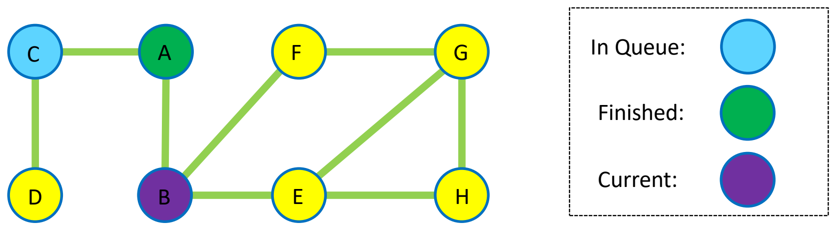
Current:

Queue: B | C

Finished: A

FIT2004: Seminar 4 - Introduction to Graphs

Breadth-First Search (BFS)

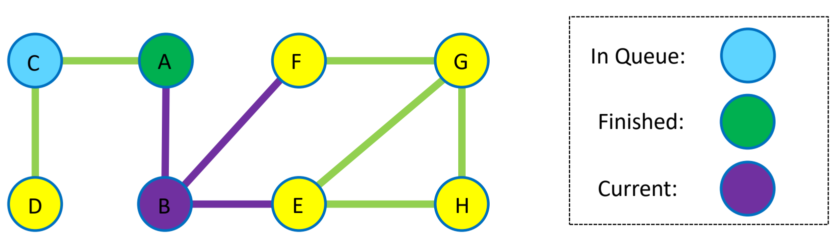


Current: B

Queue: C

Finished: A

Breadth-First Search (BFS)



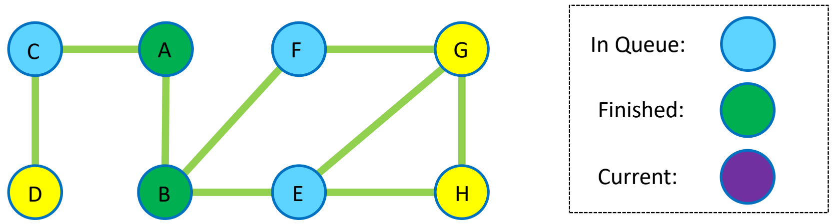
Current: B

Queue: C

Finished: A

FIT2004: Seminar 4 - Introduction to Graphs

Breadth-First Search (BFS)

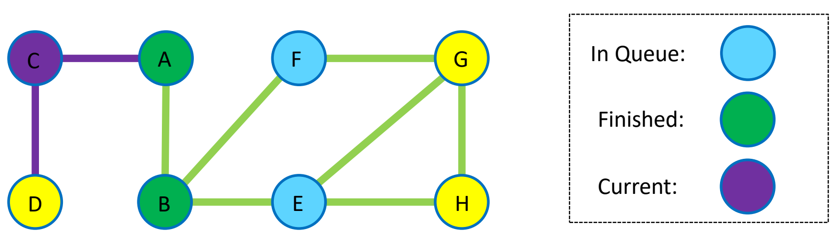


Current:

Queue: | C | E | F |

Finished: | A | B |

Breadth-First Search (BFS)

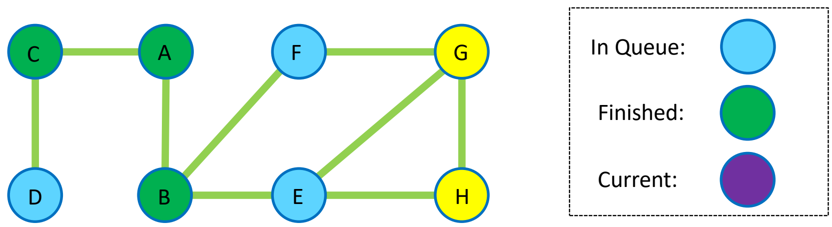


Current: | C |

Queue: | E | F |

Finished: | A | B |

Breadth-First Search (BFS)

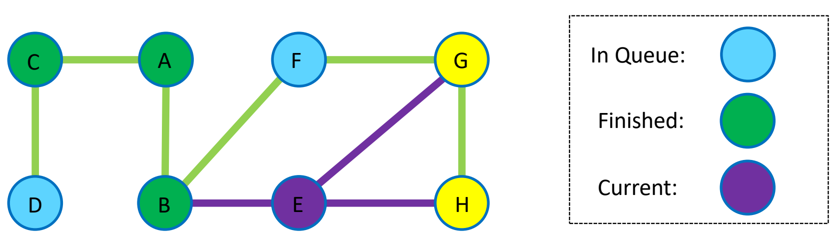


Current:

Queue:			
	E	F	D

Finished:			
	A	B	C

Breadth-First Search (BFS)

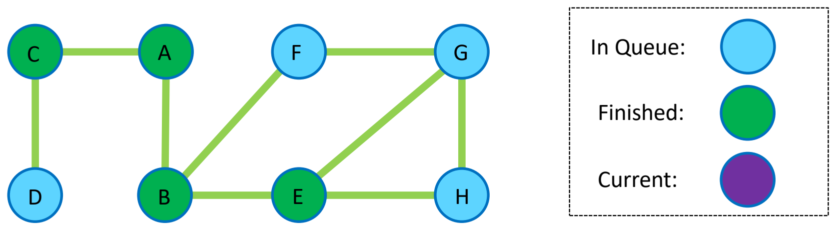


Current:	
	E

Queue:		
	F	D

Finished:			
	A	B	C

Breadth-First Search (BFS)



Current:

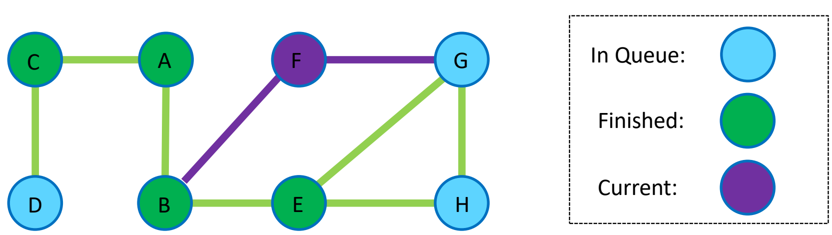
Queue:

F	D	G	H

Finished:

A	B	C	E

Breadth-First Search (BFS)



Current:

F

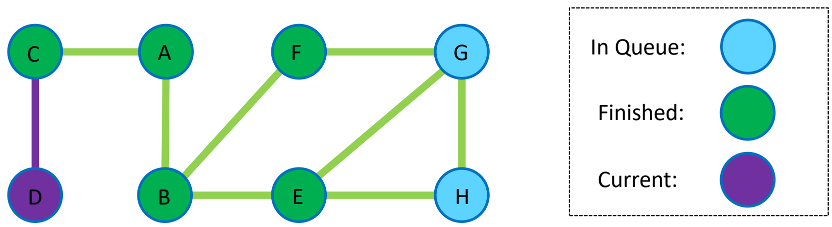
Queue:

D	G	H

Finished:

A	B	C	E

Breadth-First Search (BFS)

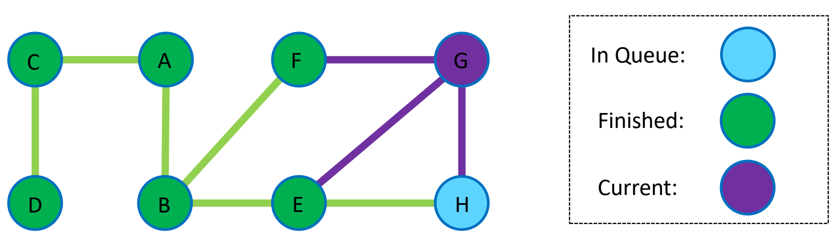


Current: D

Queue: G | H

Finished: A | B | C | E | F

Breadth-First Search (BFS)

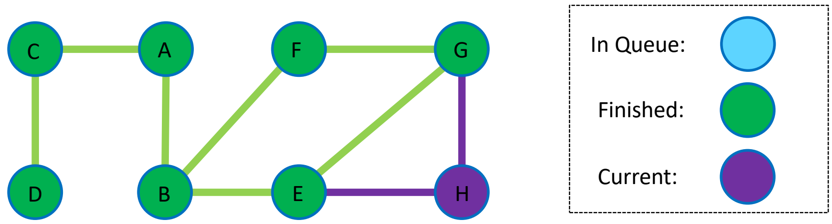


Current: G

Queue: H

Finished: A | B | C | E | F | D

Breadth-First Search (BFS)



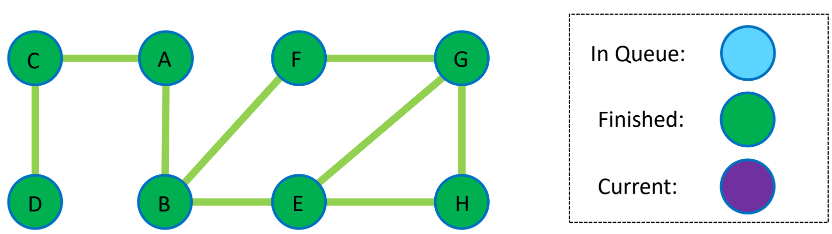
Current: H

Queue:

Finished:

A	B	C	E	F	D	G

Breadth-First Search (BFS)



Current:

Queue:

Finished:

A	B	C	E	F	D	G	H

Breadth-First Search (BFS)

- Initialize some data structure "**queue**" and some data structure "**visited**", both empty of vertices
- Put an initial vertex in **queue**
- Mark the initial vertex as visited
- While **queue** is not empty
  - Get the first vertex, **u**, from **queue**
  - For each edge (**u**,**v**)
    - If **v** is not **visited**
      - Add **v** to **visited**
      - Add **v** at the end of **queue**

What is the time complexity of BFS?

Breadth-First Search (BFS)

- Initialize some data structure "**queue**" and some data structure "**visited**", both empty of vertices → $O(?)$
- Put an initial vertex in **queue** → $O(1)$
- Mark the initial vertex as visited → $O(?)$
- While **queue** is not empty → $O(1)$
  - Get the first vertex, **u**, from **queue** → $O(V) total$
  - For each edge (**u**,**v**) → $O(E)$ $total$
    - If **v** is not **visited** → $O(?)$
      - Add **v** to **visited** → $O(?)$
      - Add **v** at the end of **queue** → $O(V) total$

Assuming adjacency list representation.

Time Complexity:

- We look at every edge twice
- For each edge, we do a lookup on visited (with some complexity)
- We insert vertices to visited at most O(V) times
- O(V*insert to visited + E*lookup on visited)

Breadth-First Search (BFS)

- Initialize some data structure "**queue**" and some data structure "**visited**", both empty of vertices → $O(?)$
- Put an initial vertex in **queue** → $O(1)$
- Mark the initial vertex as visited → $O(?)$
- While **queue** is not empty → $O(1)$
  - Get the first vertex, **u**, from **queue** → $O(V) total$
  - For each edge (**u**,**v**) → $O(E)$ total
    - If **v** is not **visited** → $O(?)$
      - Add **v** to **visited** → $O(?)$
      - Add **v** at the end of **queue** → $O(V) total$

Assuming adjacency list representation.

Time Complexity:

- O(V*insert to visited + E*lookup on visited)
- Visited is just a bit list, indexed by vertex ID
- Lookup and insert are both O(1)

Breadth-First Search (BFS)

- Initialize some data structure "**queue**" and some data structure "**visited**", both empty of vertices
- Put an initial vertex in **queue**
- Mark the initial vertex as visited
- While **queue** is not empty
  - Get the first vertex, **u**, from **queue**
  - For each edge (**u**,**v**)
    - If **v** is not **visited**
      - Add **v** to visited
      - add **v** at the end of **queue**

Assuming adjacency list representation.

Time Complexity:

- O(V * 1 + E * 1) = O(V+E)

Space Complexity:

- O(V+E)

Breadth-First Search (BFS)

Algorithm 55 Generic breadth-first search

```
function BFS(G = (V, E), s)
    visited[1..n] = false
    visited[s] = true
    queue = Queue()
    queue.push(s)
    while queue is not empty do
        u = queue.pop()
        for each vertex v adjacent to u do
            if not visited[v] then
                visited[v] = true
                queue.push(v)
```

Assuming adjacency list representation.

Time Complexity:

- O(V+E)

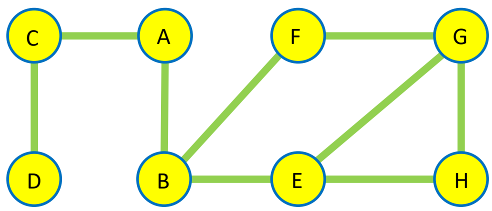
Space Complexity:

- O(V+E)

Outline

1. Introduction to Graphs
2. Graph Traversal Algorithms
   A. The idea
   B. Breadth-First Search (BFS)
   C. Depth-First Search (DFS)
   D. Applications

Depth-First Search (DFS)



Visited:

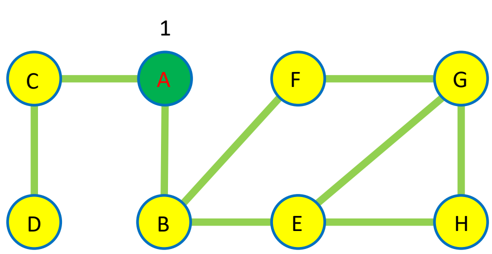
Current: **A**

Visited is indexed by vertex ID. Normally, the IDs are integers from from 0 to V-1 (to allow O(1) lookup), but letters are used here for ease of understanding

Visited:

A	B	C	D	E	F	G	H
0	0	0	0	0	0	0	0

Depth-First Search (DFS)



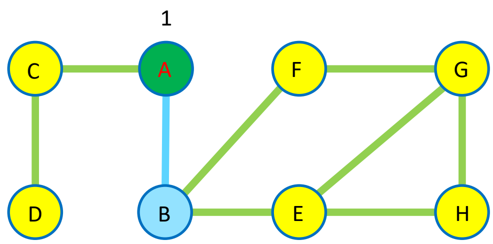
Visited:

Current: **A**

Visited:

A	B	C	D	E	F	G	H
1	0	0	0	0	0	0	0

Depth-First Search (DFS)



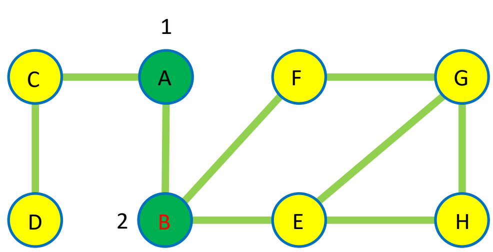
Visited:

Current: **A**

Visited:

A	B	C	D	E	F	G	H
1	0	0	0	0	0	0	0

Depth-First Search (DFS)



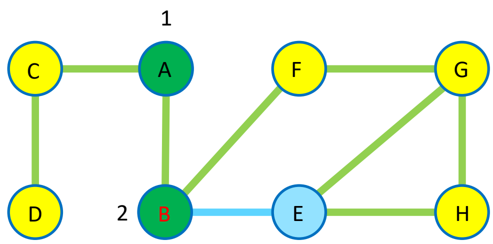
Visited:

Current: **B**

Visited:

A	B	C	D	E	F	G	H
1	1	0	0	0	0	0	0

Depth-First Search (DFS)

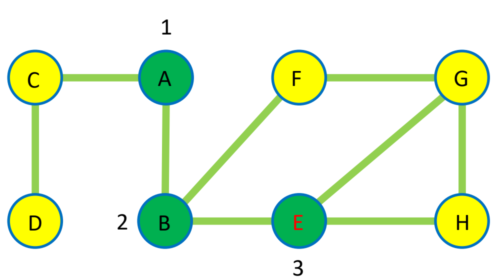


Current: **B**

Visited:

A	B	C	D	E	F	G	H
1	1	0	0	0	0	0	0

Depth-First Search (DFS)

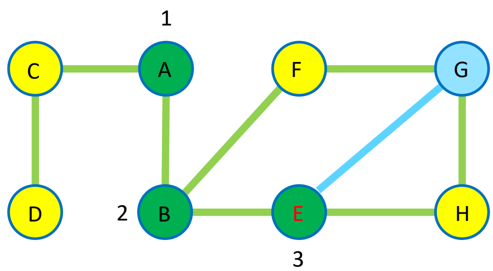


Current: **E**

Visited:

A	B	C	D	E	F	G	H
1	1	0	0	1	0	0	0

Depth-First Search (DFS)



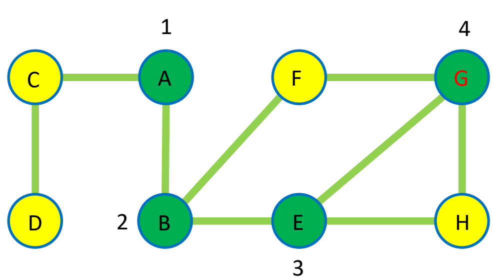
Visited:

Current: **E**

Visited:

A	B	C	D	E	F	G	H
1	1	0	0	1	0	0	0

Depth-First Search (DFS)



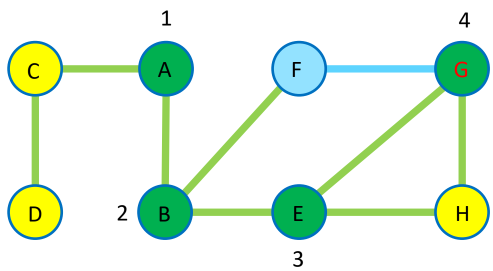
Visited:

Current: **G**

Visited:

A	B	C	D	E	F	G	H
1	1	0	0	1	0	1	0

Depth-First Search (DFS)



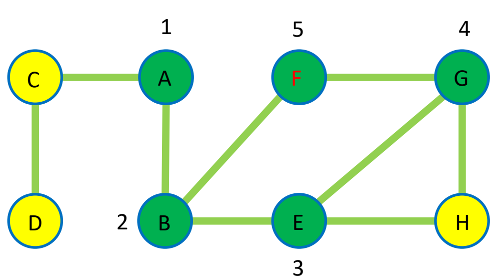
Visited:

Current: **G**

Visited:

A	B	C	D	E	F	G	H
1	1	0	0	1	0	1	0

Depth-First Search (DFS)



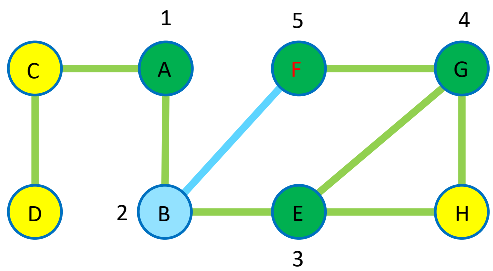
Visited:

Current: **F**

Visited:

A	B	C	D	E	F	G	H
1	1	0	0	1	1	1	0

Depth-First Search (DFS)



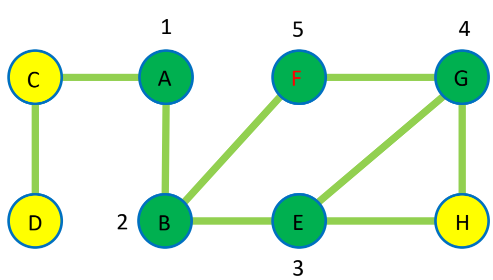
Visited:

Current: **F**

Visited:

A	B	C	D	E	F	G	H
1	1	0	0	1	1	1	0

Depth-First Search (DFS)



Visited:

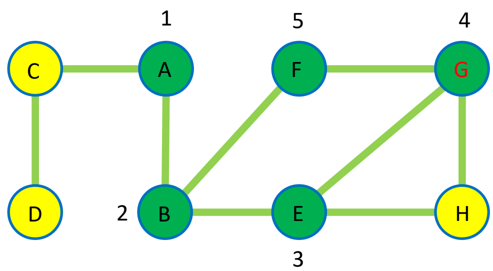
- F is a dead end
- Go back to the last active node (G)

Current: **F**

Visited:

A	B	C	D	E	F	G	H
1	1	0	0	1	1	1	0

Depth-First Search (DFS)



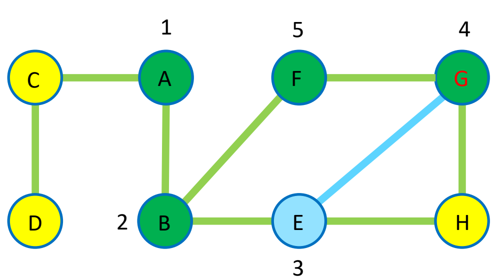
Visited:

Current: G

Visited:

A	B	C	D	E	F	G	H
1	1	0	0	1	1	1	0

Depth-First Search (DFS)



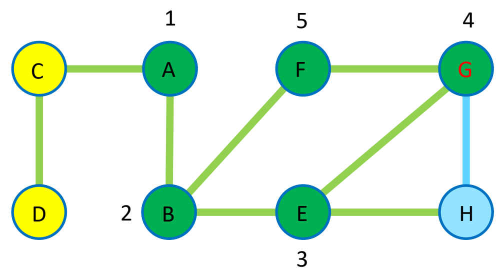
Visited:

Current: G

Visited:

A	B	C	D	E	F	G	H
1	1	0	0	1	1	1	0

Depth-First Search (DFS)



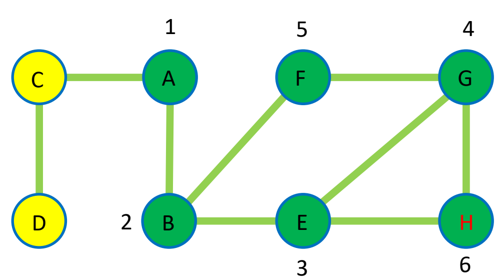
Visited:

Current: **G**

Visited:

A	B	C	D	E	F	G	H
1	1	0	0	1	1	1	0

Depth-First Search (DFS)



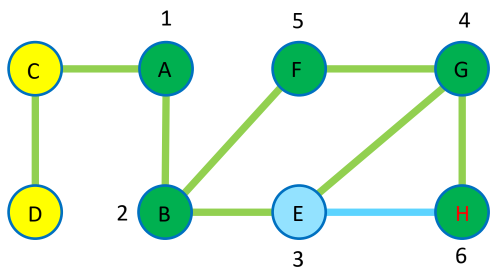
Visited:

Current: **H**

Visited:

A	B	C	D	E	F	G	H
1	1	0	0	1	1	1	1

Depth-First Search (DFS)



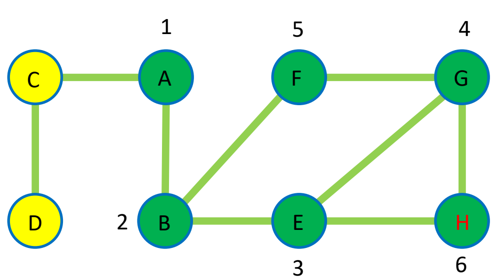
Visited:

Current: **H**

Visited:

A	B	C	D	E	F	G	H
1	1	0	0	1	1	1	1

Depth-First Search (DFS)



Visited:

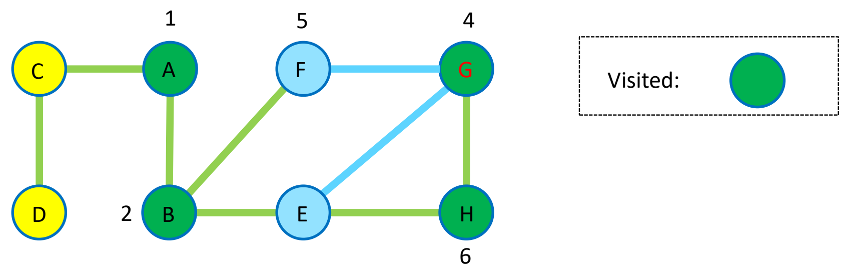
- H is a dead end
- Speeding up visualisation…

Current: **H**

Visited:

A	B	C	D	E	F	G	H
1	1	0	0	1	1	1	1

Depth-First Search (DFS)

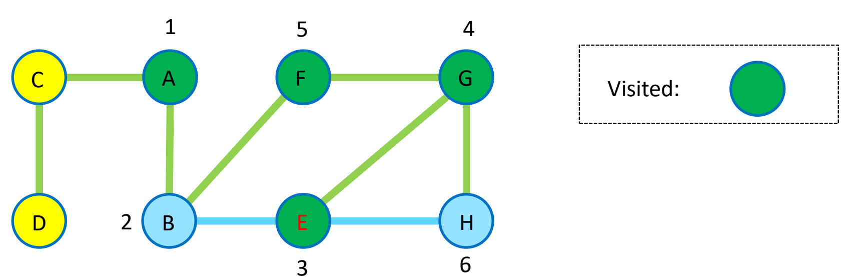


Current: **G**

Visited:

A	B	C	D	E	F	G	H
1	1	0	0	1	1	1	1

Depth-First Search (DFS)

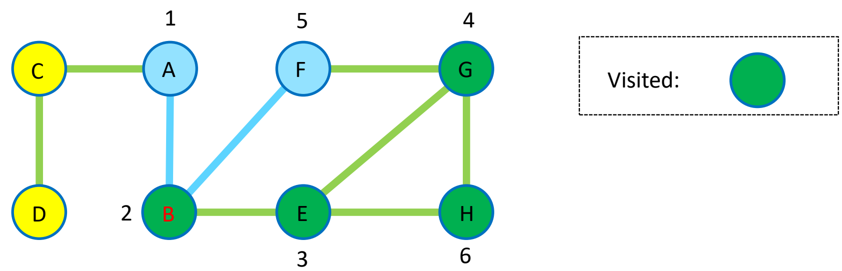


Current: **E**

Visited:

A	B	C	D	E	F	G	H
1	1	0	0	1	1	1	1

Depth-First Search (DFS)

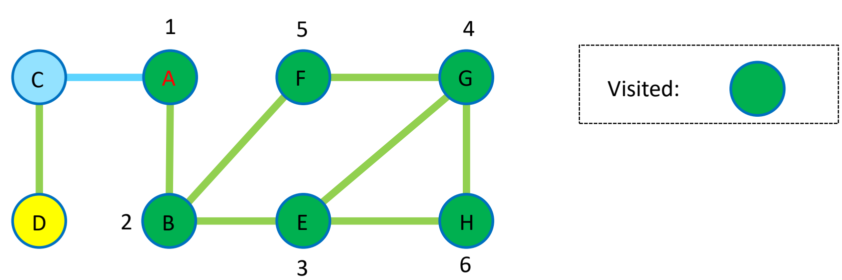


Current: **B**

Visited:

A	B	C	D	E	F	G	H
1	1	0	0	1	1	1	1

Depth-First Search (DFS)



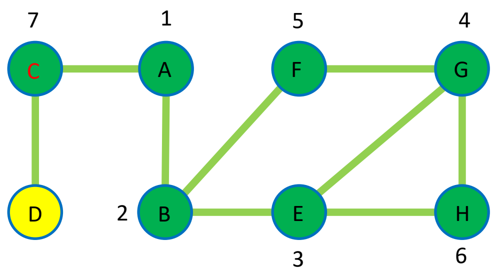
Current: **A**

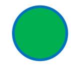
Visited:

A	B	C	D	E	F	G	H
1	1	0	0	1	1	1	1

FIT2004: Seminar 4 - Introduction to Graphs

Depth-First Search (DFS)



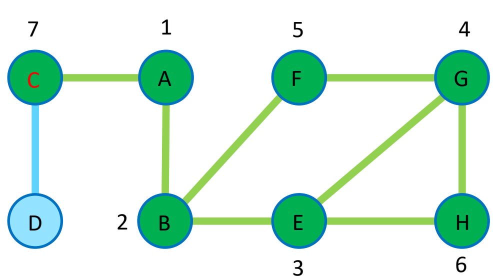
Visited: 

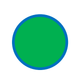
Current: **C**

Visited:

A	B	C	D	E	F	G	H
1	1	1	0	1	1	1	1

Depth-First Search (DFS)



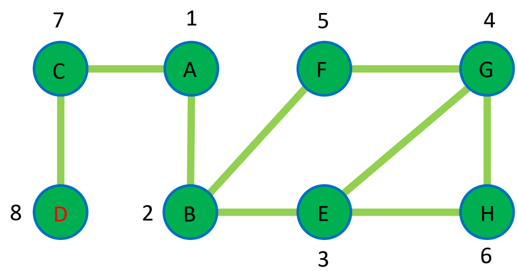
Visited: 

Current: **C**

Visited:

A	B	C	D	E	F	G	H
1	1	1	0	1	1	1	1

Depth-First Search (DFS)



Visited:

Current: **D**

Visited:

A	B	C	D	E	F	G	H
1	1	1	1	1	1	1	1

Depth-First Search (DFS)

Algorithm 52 Generic depth-first search

```
// Driver function that calls DFS until everything has been visited
function TRAVERSE(G = (V, E))
    visited[1..n] = false
    for each vertex u = 1 to n do
        if not visited[u] then
            DFS(u)

function DFS(u)
    visited[u] = true
    for each vertex v adjacent to u do
        if not visited[v] then
            DFS(v)
```

Assuming adjacency list representation.

Time Complexity:

- Each vertex visited at most once
- Each edge accessed at most twice (once when u is visited once when v is visited)
- Total cost: O(V+E)

Space Complexity:

- O(V+E)

Outline

1. Introduction to Graphs
2. Graph Traversal Algorithms
   A. The idea
   B. Breadth-First Search (BFS)
   C. Depth-First Search (DFS)
   D. Applications

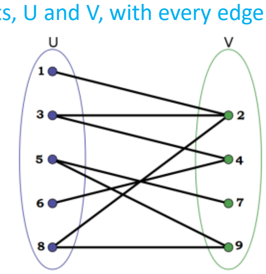
Applications of DFS and BFS

The algorithms we saw can also be applied on directed graphs.

BFS and DFS have a wide variety of applications:

- Reachability
- Finding all connected components
- Testing a graph for bipartiteness
  - A graph is bipartite when we can divide it into two sets, U and V, with every edge having one vertex in set U and the other in set V
- Finding cycles
- **Shortest paths on unweighted graphs**
- **Topological sort**
- …

More details are given in unit notes and applied classes.



Example of bipartite graph

Shortest Path Problem

Length of a path:

For unweighted graphs, the length of a path is the number of edges along the path.

For weighted graphs, the length of a path is the sum of weights of the edges along the path.

Shortest Path Problem

Single source, single target:

Given a source vertex *s* and a target vertex *t*, return the shortest path from *s* to *t*.

Single source, all targets:

Given a source vertex *s*, return the shortest paths to every other vertex in the graph.

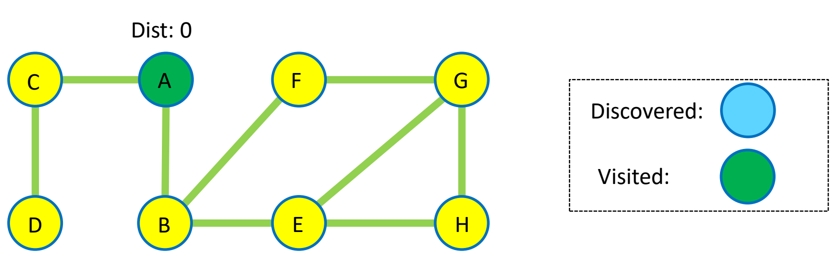
We will focus on single source, all targets problem because the single source, single target problem is subsumed by it.

FIT2004: Seminar 4 - Introduction to Graphs

Shortest Path Algorithms

- Breadth-First Search – (Single source, unweighted graphs) Today
- Dijkstra's Algorithm – (Single Source, weighted graphs with non-negative weights) Week 5
- Bellman-Ford Algorithm – (Single source, weighted graphs including negative weights) Week 7
- Floyd-Warshall Algorithm– (All pairs, weighted graphs including negative weights) Week 7

BFS Shortest Path (now with good data structures)



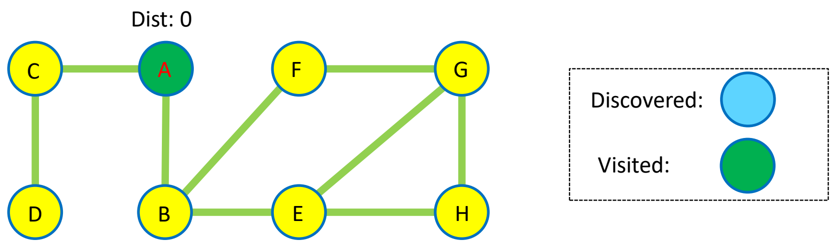
Current:

Queue: A

Visited:

A	B	C	D	E	F	G	H
1	0	0	0	0	0	0	0

BFS Shortest Path (now with good data structures)



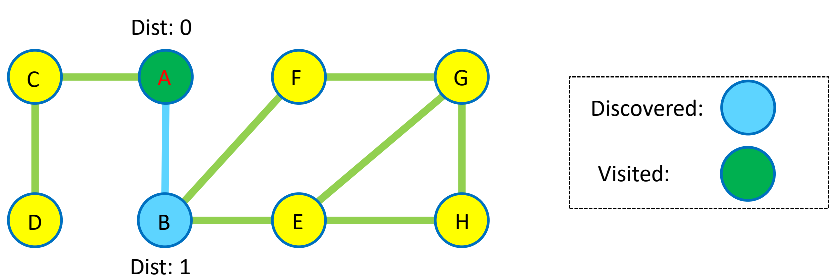
Current: A

Queue:

Visited:

A	B	C	D	E	F	G	H
1	0	0	0	0	0	0	0

BFS Shortest Path (now with good data structures)



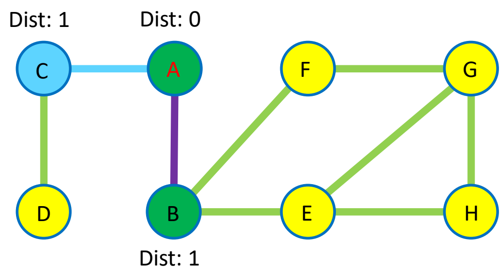
Current: A

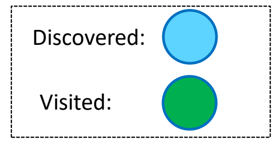
Queue:

Visited:

A	B	C	D	E	F	G	H
1	0	0	0	0	0	0	0

BFS Shortest Path (now with good data structures)





Current: A

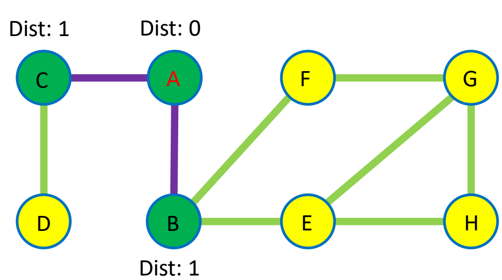
Always keep track of the predecessor nodes.

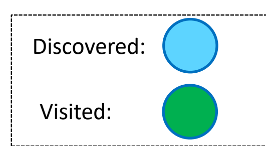
Queue: B

Visited:

A	B	C	D	E	F	G	H
1	1	0	0	0	0	0	0

BFS Shortest Path (now with good data structures)





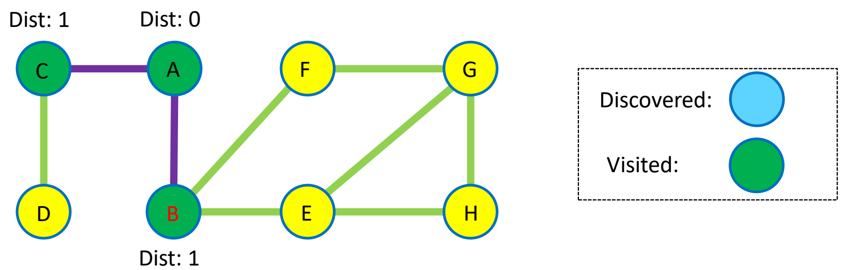
Current: A

Queue: B | C

Visited:

A	B	C	D	E	F	G	H
1	1	1	0	0	0	0	0

BFS Shortest Path (now with good data structures)



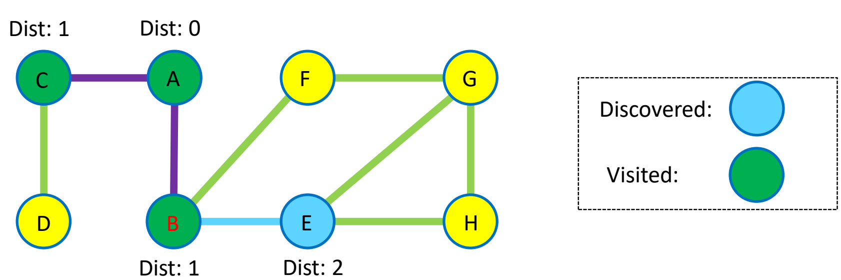
Current: B

Queue: C

Visited:

A	B	C	D	E	F	G	H
1	1	1	0	0	0	0	0

BFS Shortest Path (now with good data structures)



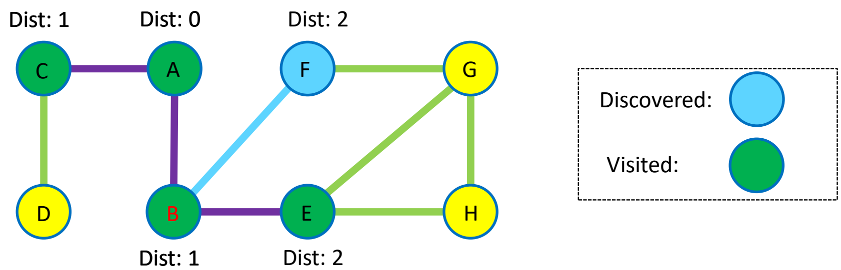
Current: B

Queue: C

Visited:

A	B	C	D	E	F	G	H
1	1	1	0	0	0	0	0

BFS Shortest Path (now with good data structures)



Current: B

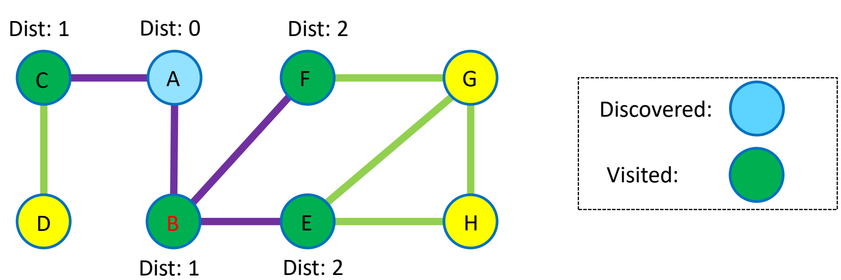
Queue:

C	E

Visited:

A	B	C	D	E	F	G	H
1	1	1	0	1	0	0	0

BFS Shortest Path (now with good data structures)



Current: B

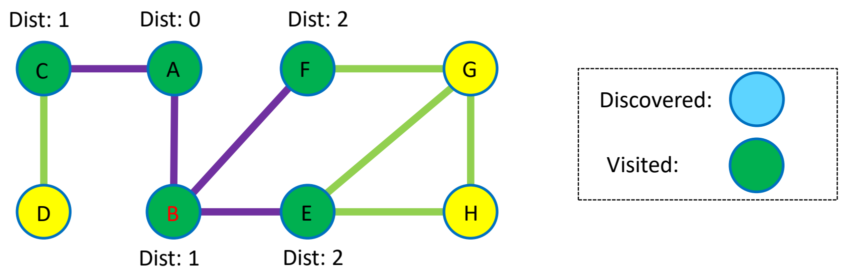
Queue:

C	E	F

Visited:

A	B	C	D	E	F	G	H
1	1	1	0	1	1	0	0

BFS Shortest Path (now with good data structures)



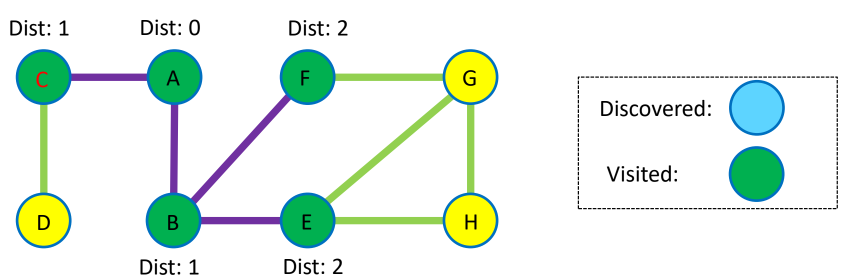
Current: B

Queue: C | E | F

Visited:

A	B	C	D	E	F	G	H
1	1	1	0	1	1	0	0

BFS Shortest Path (now with good data structures)



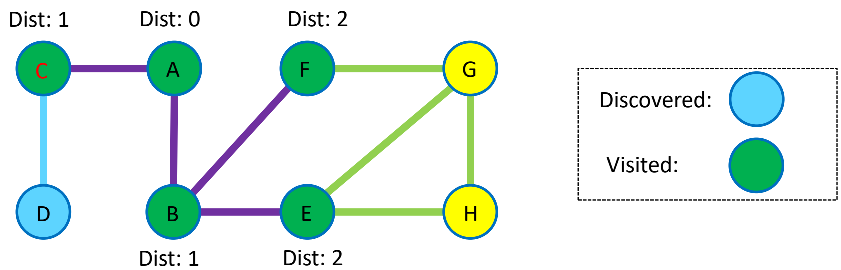
Current: C

Queue: E | F

Visited:

A	B	C	D	E	F	G	H
1	1	1	0	1	1	0	0

BFS Shortest Path (now with good data structures)



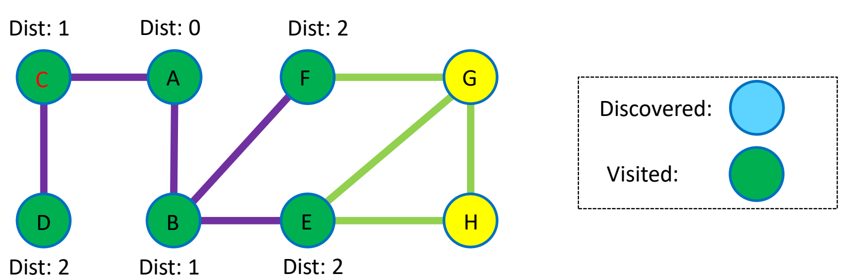
Current: C

Queue: E F

Visited:

A	B	C	D	E	F	G	H
1	1	1	0	1	1	0	0

BFS Shortest Path (now with good data structures)



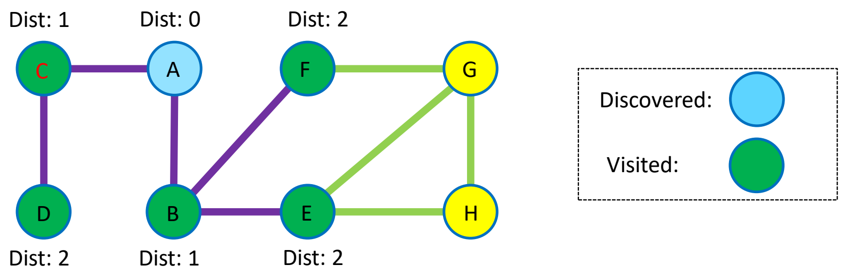
Current: C

Queue: E F D

Visited:

A	B	C	D	E	F	G	H
1	1	1	1	1	1	0	0

BFS Shortest Path (now with good data structures)



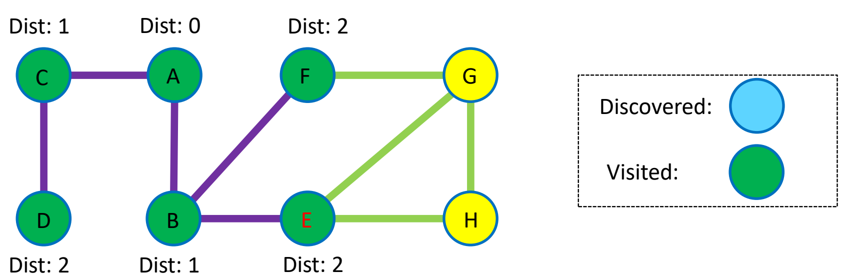
Current: C

Queue: E | F | D

Visited:

A	B	C	D	E	F	G	H
1	1	1	1	1	1	0	0

BFS Shortest Path (now with good data structures)



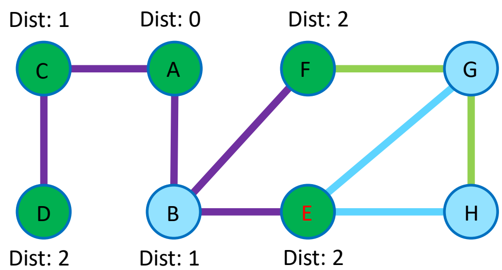
Current: E

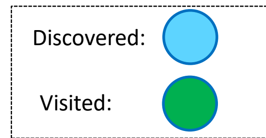
Queue: F | D

Visited:

A	B	C	D	E	F	G	H
1	1	1	1	1	1	0	0

BFS Shortest Path (now with good data structures)





Current: E

Speeding up visualisation…

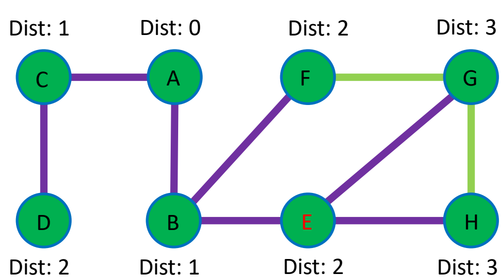
Queue:

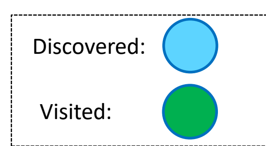
F	D

Visited:

A	B	C	D	E	F	G	H
1	1	1	1	1	1	0	0

BFS Shortest Path (now with good data structures)





Current: E

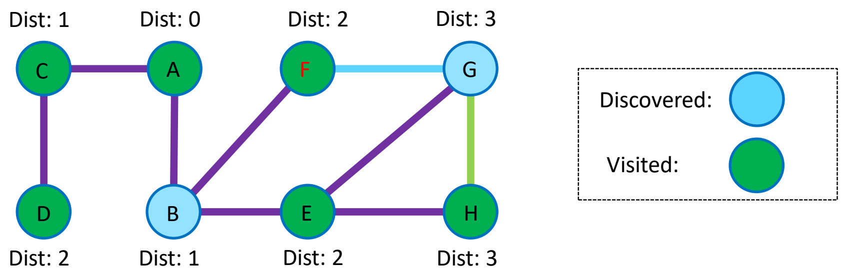
Queue:

F	D	G	H

Visited:

A	B	C	D	E	F	G	H
1	1	1	1	1	1	1	1

BFS Shortest Path (now with good data structures)



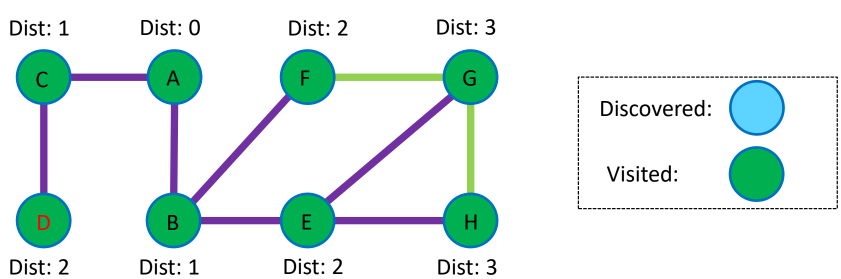
Current: F

Queue: D | G | H

Visited:

A	B	C	D	E	F	G	H
1	1	1	1	1	1	1	1

BFS Shortest Path (now with good data structures)



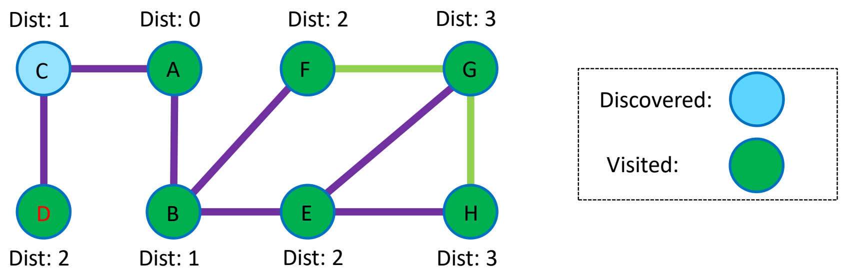
Current: D

Queue: G | H

Visited:

A	B	C	D	E	F	G	H
1	1	1	1	1	1	1	1

BFS Shortest Path (now with good data structures)



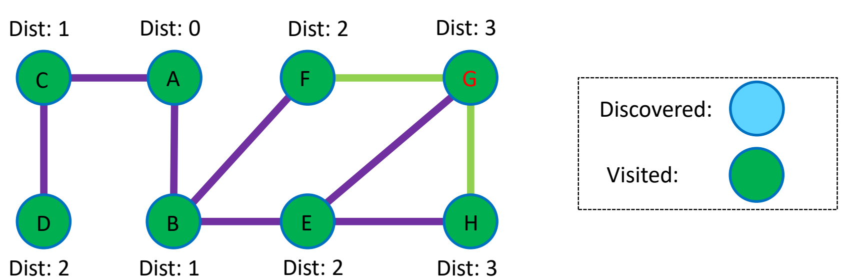
Current: D

Queue: G | H

Visited:

A	B	C	D	E	F	G	H
1	1	1	1	1	1	1	1

BFS Shortest Path (now with good data structures)



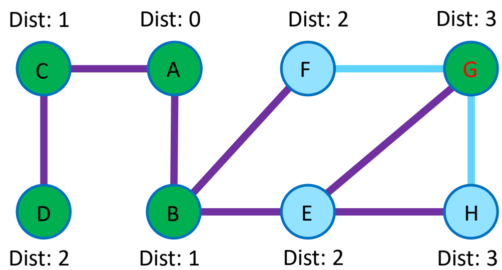
Current: G

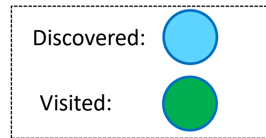
Queue: H

Visited:

A	B	C	D	E	F	G	H
1	1	1	1	1	1	1	1

BFS Shortest Path (now with good data structures)





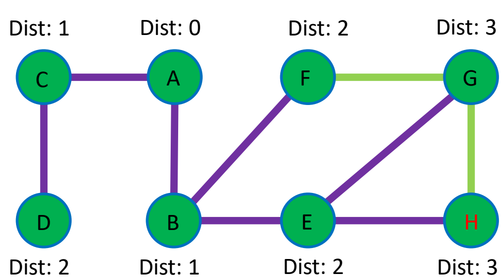
Current: G

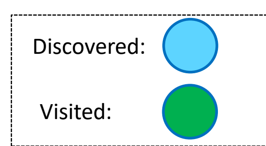
Queue: H

Visited:

A	B	C	D	E	F	G	H
1	1	1	1	1	1	1	1

BFS Shortest Path (now with good data structures)





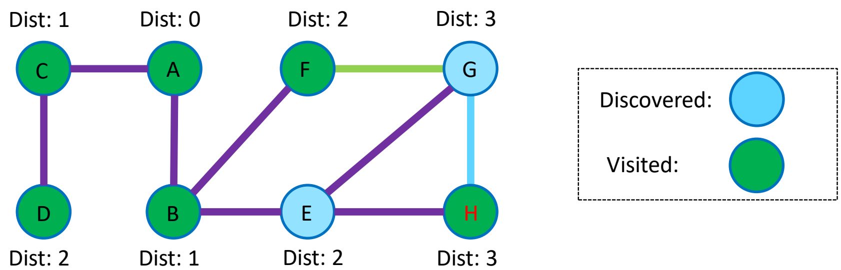
Current: H

Queue:

Visited:

A	B	C	D	E	F	G	H
1	1	1	1	1	1	1	1

BFS Shortest Path (now with good data structures)



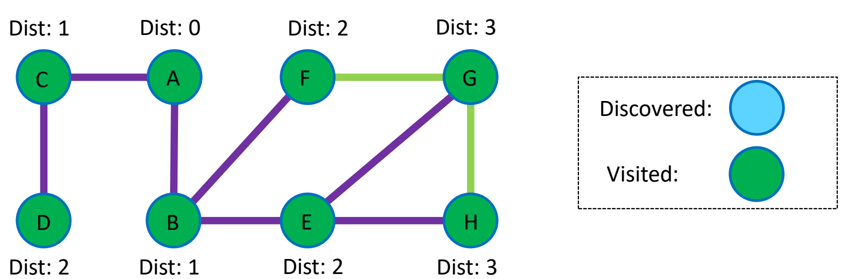
Current: H

Queue:

Visited:

A	B	C	D	E	F	G	H
1	1	1	1	1	1	1	1

BFS Shortest Path (now with good data structures)



Current:

Queue:

Visited:

A	B	C	D	E	F	G	H
1	1	1	1	1	1	1	1

Unweighted Shortest Paths

Algorithm 56 Single-source shortest paths in an unweighted graph

```
function BFS(G = (V, E), s)
    dist[1..n] = ∞
    pred[1..n] = null
    queue = Queue()
    queue.push(s)
    dist[s] = 0
    while queue is not empty do
        u = queue.pop()
        for each vertex v adjacent to u do
            if dist[v] = ∞ then
                dist[v] = dist[u] + 1
                pred[v] = u
                queue.push(v)
```

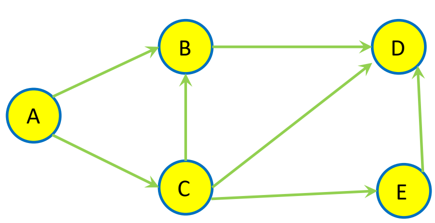
- Note that distances are stored in an O(1) lookup structure.
- Distances are set by lookup at the distance of the current vertex and adding 1.
- Path from s to v can be found by backtracking from v to s using the array pred.
- Complexity is the same as regular BFS, O(V+E).

Directed Acyclic Graph (DAG)

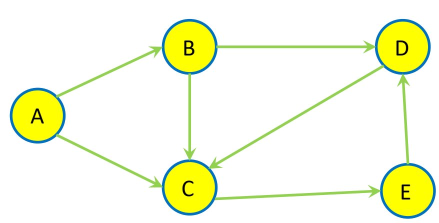
A Directed Acyclic Graph (DAG) is

- Directed
- Acyclic – has no cycles
- Graph

Which of the two graphs is a DAG?



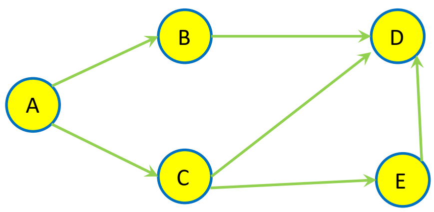
Graph 1

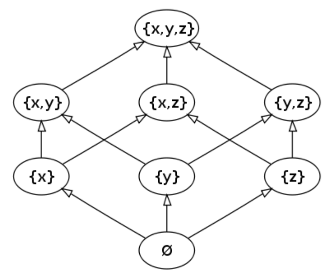


Graph 2

DAG: Examples

- Sub-tasks of a project and which "must finish before"
  - A → B means task A must finish before task B
  - so, DAGs useful in project management
- Relationships between subjects for your degree -- "is prerequisite for"
  - A→B means subject A must be completed before enrolling in subject B
- People genealogy – "is an ancestor of"
  - A → B means A is an ancestor of B
- Power sets and "is a subset of"
  - A → B means A is a subset of B





Source: wikipedia

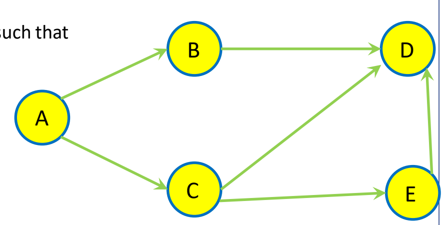
Topological Sort of a DAG

Order of vertices in a DAG

- A < B if A→B.
  - Note that if A → B and B→D, we have A < B and B < D which implies that A < D (i.e., transitivity).
- Some vertices may be incomparable (e.g., B and C are incomparable), i.e. A< B and A < C but we do not know whether C < B or B < C.

A topological order

- is a permutation of the vertices in the original DAG such that
- for **every** directed edge u→v of the DAG
  - u appears before v in the permutation



Example: A, B, C, E, D

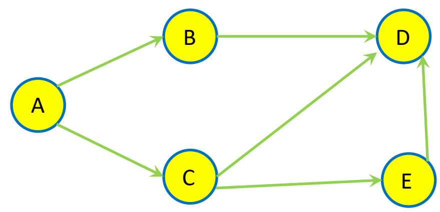
- Topological sort of a DAG of "is prerequisite of" example gives an ordering of the subjects for studying your degree, one at a time, while obeying prerequisite rules.

Topological Sort of a DAG

- A DAG can have many valid topological sorts, e.g., let u and v be two incomparable vertices, u may appear before or after v.

Which of these is NOT a valid topological ordering of the DAG?

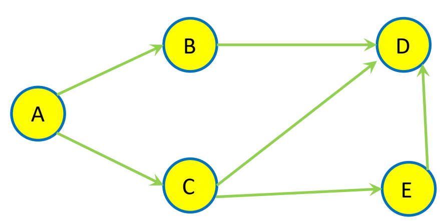
1. A, B, C, E, D
2. A, C, B, E, D
3. A, C, E, B, D
4. A, B, E, C, D



Depth First Search (DFS)

Assume we call DFS(A), which of the following is NOT a possible order in which vertices are marked visited?

1. A, B, D, C, E
2. A, C, E, D, B
3. A, C, D, E, B
4. A, C, E, B, D



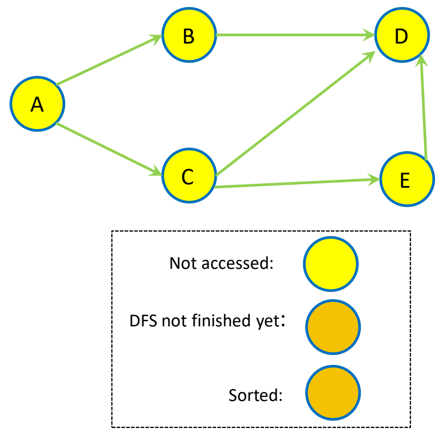
DFS for Topological Sort

Algorithm 72 Topological sorting using DFS

```
function TOPOLOGICAL_SORT(G = (V, E))
    order = empty array
    visited[1..n] = false
    for each vertex v = 1 to n do
        if not visited[v] then
            DFS(v)
    return reverse(order)

function DFS(u)
    visited[u] = true
    for each vertex v adjacent to u do
        if not visited[v] then
            DFS(v)
    order.append(u)
```

*// Add to order **after** visiting descendants*



Sorted:

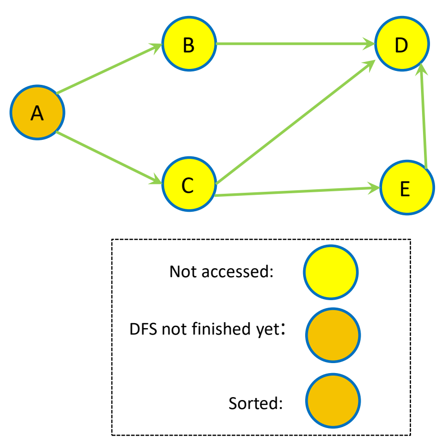
DFS for Topological Sort

Algorithm 72 Topological sorting using DFS

```
function TOPOLOGICAL_SORT(G = (V, E))
    order = empty array
    visited[1..n] = false
    for each vertex v = 1 to n do
        if not visited[v] then
            DFS(v)
    return reverse(order)

function DFS(u)
    visited[u] = true
    for each vertex v adjacent to u do
        if not visited[v] then
            DFS(v)
    order.append(u)
```

*// Add to order **after** visiting descendants*



Sorted:

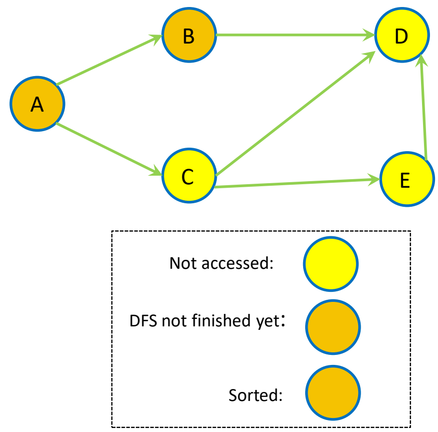
DFS for Topological Sort

Algorithm 72 Topological sorting using DFS

function TOPOLOGICAL_SORT($G = (V, E)$)
    order = empty array
    visited[1..*n*] = **false**
    for each vertex $v = 1$ **to** n **do**
        if not *visited*[v] **then**
            DFS(v)
    return reverse(*order*)

function DFS(u)
    visited[u] = **true**
    for each vertex v adjacent to u **do**
        if not *visited*[v] **then**
            DFS(v)
    order.append(u)

*// Add to order **after** visiting descendants*



Sorted:

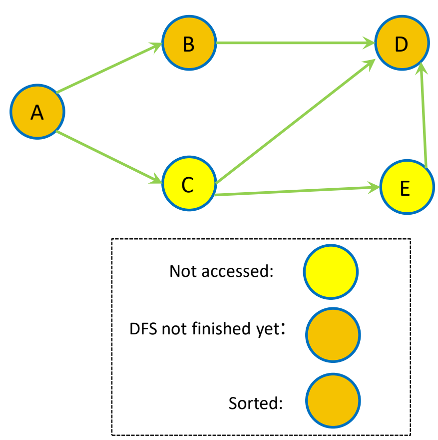
DFS for Topological Sort

Algorithm 72 Topological sorting using DFS

function TOPOLOGICAL_SORT($G = (V, E)$)
    order = empty array
    visited[1..*n*] = **false**
    for each vertex $v = 1$ **to** n **do**
        if not *visited*[v] **then**
            DFS(v)
    return reverse(*order*)

function DFS(u)
    visited[u] = **true**
    for each vertex v adjacent to u **do**
        if not *visited*[v] **then**
            DFS(v)
    order.append(u)

*// Add to order **after** visiting descendants*



Sorted:

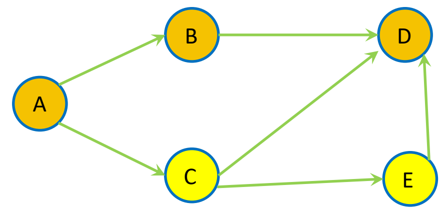
DFS for Topological Sort

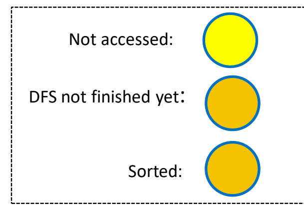
Algorithm 72 Topological sorting using DFS

```
function TOPOLOGICAL_SORT(G = (V, E))
    order = empty array
    visited[1..n] = false
    for each vertex v = 1 to n do
        if not visited[v] then
            DFS(v)
    return reverse(order)

function DFS(u)
    visited[u] = true
    for each vertex v adjacent to u do
        if not visited[v] then
            DFS(v)
    order.append(u)
```

*// Add to order **after** visiting descendants*





Sorted: D

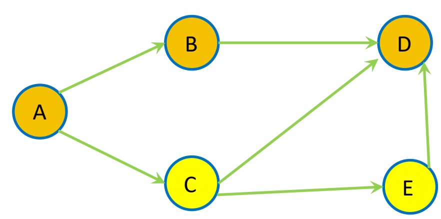
DFS for Topological Sort

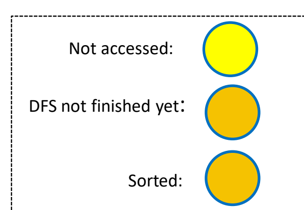
Algorithm 72 Topological sorting using DFS

```
function TOPOLOGICAL_SORT(G = (V, E))
    order = empty array
    visited[1..n] = false
    for each vertex v = 1 to n do
        if not visited[v] then
            DFS(v)
    return reverse(order)

function DFS(u)
    visited[u] = true
    for each vertex v adjacent to u do
        if not visited[v] then
            DFS(v)
    order.append(u)
```

*// Add to order **after** visiting descendants*





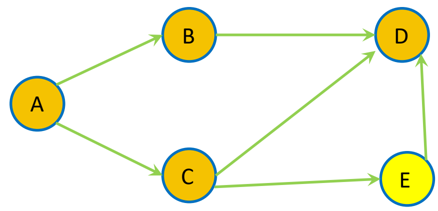
Sorted: B D

DFS for Topological Sort

Algorithm 72 Topological sorting using DFS

```
function TOPOLOGICAL_SORT(G = (V, E))
    order = empty array
    visited[1..n] = false
    for each vertex v = 1 to n do
        if not visited[v] then
            DFS(v)
    return reverse(order)

function DFS(u)
    visited[u] = true
    for each vertex v adjacent to u do
        if not visited[v] then
            DFS(v)
    order.append(u)
```



Not accessed:

DFS not finished yet:

Sorted:

*// Add to order **after** visiting descendants*

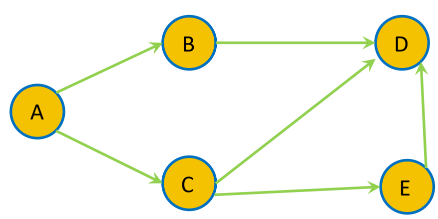
Sorted: B D

DFS for Topological Sort

Algorithm 72 Topological sorting using DFS

```
function TOPOLOGICAL_SORT(G = (V, E))
    order = empty array
    visited[1..n] = false
    for each vertex v = 1 to n do
        if not visited[v] then
            DFS(v)
    return reverse(order)

function DFS(u)
    visited[u] = true
    for each vertex v adjacent to u do
        if not visited[v] then
            DFS(v)
    order.append(u)
```

Not accessed:

DFS not finished yet:

Sorted:

*// Add to order **after** visiting descendants*

Sorted: B D

FIT2004: Seminar 4 - Introduction to Graphs

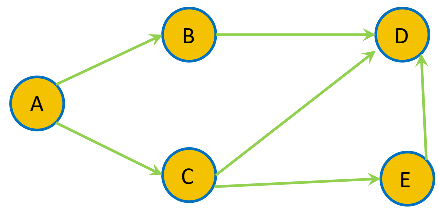
DFS for Topological Sort

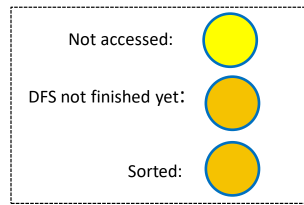
Algorithm 72 Topological sorting using DFS

```
function TOPOLOGICAL_SORT(G = (V, E))
    order = empty array
    visited[1..n] = false
    for each vertex v = 1 to n do
        if not visited[v] then
            DFS(v)
    return reverse(order)

function DFS(u)
    visited[u] = true
    for each vertex v adjacent to u do
        if not visited[v] then
            DFS(v)
    order.append(u)
```

*// Add to order **after** visiting descendants*





Sorted: E B D

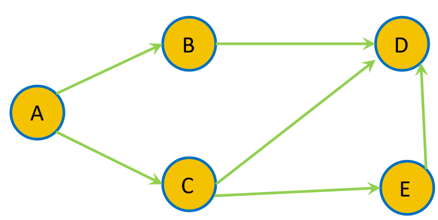
DFS for Topological Sort

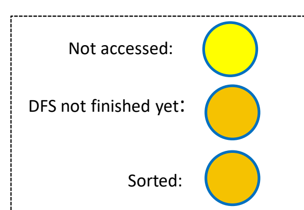
Algorithm 72 Topological sorting using DFS

```
function TOPOLOGICAL_SORT(G = (V, E))
    order = empty array
    visited[1..n] = false
    for each vertex v = 1 to n do
        if not visited[v] then
            DFS(v)
    return reverse(order)

function DFS(u)
    visited[u] = true
    for each vertex v adjacent to u do
        if not visited[v] then
            DFS(v)
    order.append(u)
```

*// Add to order **after** visiting descendants*





Sorted: C E B D

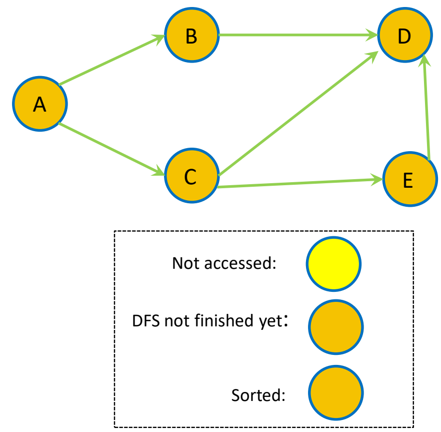
DFS for Topological Sort

Algorithm 72 Topological sorting using DFS

```
function TOPOLOGICAL_SORT(G = (V, E))
    order = empty array
    visited[1..n] = false
    for each vertex v = 1 to n do
        if not visited[v] then
            DFS(v)
    return reverse(order)

function DFS(u)
    visited[u] = true
    for each vertex v adjacent to u do
        if not visited[v] then
            DFS(v)
    order.append(u)
```

*// Add to order **after** visiting descendants*



Sorted:

A	C	E	B	D

Reading

- Course Notes: Chapter 5
- You can also check algorithms' textbooks for contents related to this lecture, e.g.:
  - CLRS: Chapter 22
  - KT: Chapter 3
  - Rou: Chapters 7 and 8

Concluding Remarks

Things to do (this list is not exhaustive)

- Read more about BFS, DFS and their applications
- Implement BFS and DFS
- Read course notes

Coming Up Next

- Greedy (Graph) Algorithms