CMP_SC 3050: Graphs

Rohit Chadha

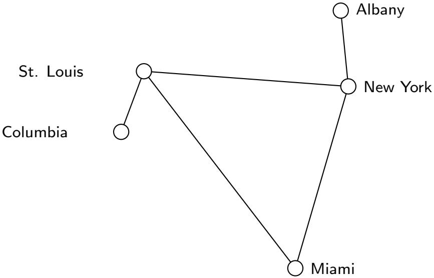
September 16, 2014

Graphs

A graph is a way of encoding pairwise relationships amongst a set of objects

- The objects are often called vertices
- The pairwise relationships are called edges
- If the relationship is symmetric, we get undirected graphs
- If the relationship is asymmetric, we get directed graphs

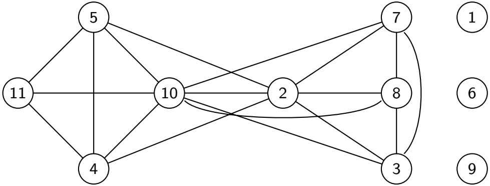
Transportation Networks



Scheme: Points denote cities, warehouses, ports, airfields, etc. A line between x and y denotes the ability to move goods, people, etc. from x to y.

Goal: Design network so that traffic can move efficiently, reliably …

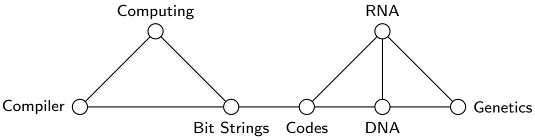
Social Networks



Scheme: Points denote individuals who interact. A line between two points denotes friendship relation between the individuals

Goal: Study the dynamics of interaction

Information Retrieval



Scheme: Points denote "descriptors" or "index terms." Lines denote similarity between descriptors

Goal: Can be used to classify similar documents together, retrieve similar documents ...

Undirected Graphs

A (undirected) graph G is a pair of sets (V, E) where

1. V is a set of vertices or nodes and
2. E is a set of unordered pairs of vertices called edges.
   An edge is a 2-element set $\{u, v\}$ where $u, v \in V$

Undirected Graphs

A (undirected) graph G is a pair of sets (V, E) where

1. V is a set of vertices or nodes and
2. E is a set of unordered pairs of vertices called edges.
   An edge is a 2-element set $\{u, v\}$ where $u, v \in V$

The edge $\{u, v\}$ is the same as the edge $\{v, u\}$ for undirected graphs

Undirected Graphs

A (undirected) graph G is a pair of sets (V, E) where

1. V is a set of vertices or nodes and
2. E is a set of unordered pairs of vertices called edges.
   An edge is a 2-element set $\{u, v\}$ where $u, v \in V$

The edge $\{u, v\}$ is the same as the edge $\{v, u\}$ for undirected graphs

From now on, we will write (u, v) instead of $\{u, v\}$

Undirected Graphs

A (undirected) graph G is a pair of sets (V, E) where

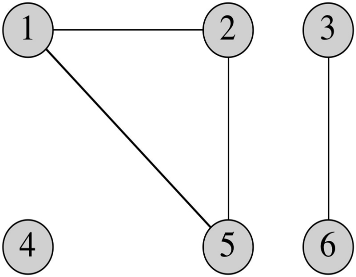
1. V is a set of vertices or nodes and
2. E is a set of unordered pairs of vertices called edges.
   An edge is a 2-element set $\{u, v\}$ where $u, v \in V$

The edge $\{u, v\}$ is the same as the edge $\{v, u\}$ for undirected graphs

From now on, we will write (u, v) instead of $\{u, v\}$

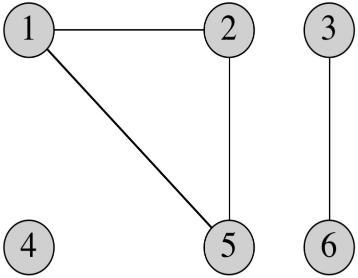
The number of vertices shall be denoted by $|V|$ and the number of edges by $|E|$

Example



Example

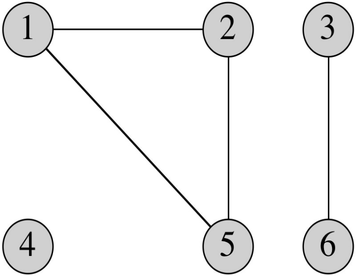
$$
\begin{aligned}
V &= \{1, 2, 3, 4, 5, 6\} \\
E &= \{(1, 2), (2, 5), (1, 5), (3, 6)\}
\end{aligned}
$$



Some terminology

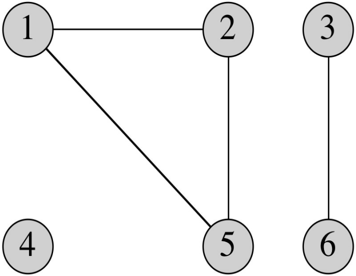
- The end points of an edge (u, v) are the vertices u and v
- A self-loop is an edge both of whose endpoints are the same
- A simple undirected graph is an undirected graph without loops. Unless otherwise stated, we will take undirected graph to mean a simple undirected graph
- An edge (u, v) is said to be incident on vertices u and v
- The degree of a vertex is the number of edges incident on it
- A vertex v is said to be adjacent to vertex u if there is an edge (u, v) in the graph

Example



Example

$$
\begin{aligned}
degree(1) &= 2 \\
degree(2) &= 2 \\
degree(3) &= 1 \\
degree(4) &= 0 \\
degree(5) &= 2 \\
degree(6) &= 1
\end{aligned}
$$



Paths

A path P in a graph G is a sequence of vertices $v_1, v_2, \ldots, v_k$ of G such that for each $i = 1, 2, \ldots, k$, the pair (v_i, v_{i+1}) is an edge of G

Paths

A path P in a graph G is a sequence of vertices $v_1, v_2, \ldots, v_k$ of G such that for each $i = 1, 2, \ldots, k$, the pair (v_i, v_{i+1}) is an edge of G

- The length of path P is $k - 1$

Paths

A path P in a graph G is a sequence of vertices $v_1, v_2, \ldots, v_k$ of G such that for each $i = 1, 2, \ldots, k$, the pair (v_i, v_{i+1}) is an edge of G

- The length of path P is $k - 1$
- P is said to be a path from v_1 to v_2 and is said to contain the vertices $v_1, v_2, \ldots, v_k$ and the edges $(v_1, v_2), (v_2, v_3) \ldots (v_{k-1}, v_k)$

Paths

A path P in a graph G is a sequence of vertices $v_1, v_2, \ldots, v_k$ of G such that for each $i = 1, 2, \ldots, k$, the pair (v_i, v_{i+1}) is an edge of G

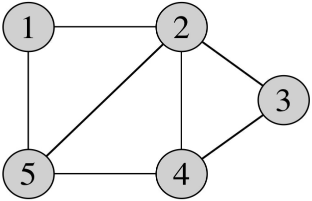
- The length of path P is $k - 1$
- P is said to be a path from v_1 to v_2 and is said to contain the vertices $v_1, v_2, \ldots, v_k$ and the edges $(v_1, v_2), (v_2, v_3) \ldots (v_{k-1}, v_k)$
- P is said to be simple if all the vertices are distinct

Paths

A path P in a graph G is a sequence of vertices $v_1, v_2, \ldots, v_k$ of G such that for each $i = 1, 2, \ldots, k$, the pair (v_i, v_{i+1}) is an edge of G

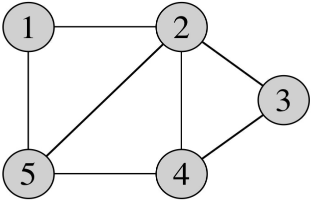
- The length of path P is $k - 1$
- P is said to be a path from v_1 to v_2 and is said to contain the vertices $v_1, v_2, \ldots, v_k$ and the edges $(v_1, v_2), (v_2, v_3) \ldots (v_{k-1}, v_k)$
- P is said to be simple if all the vertices are distinct
- The distance of a vertex v from the vertex u is the length of the shortest path from u to v

Example



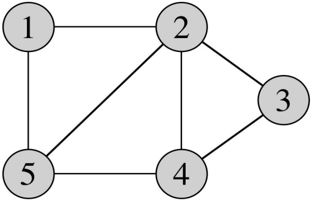
Example

- $1, 2, 3, 4, 2, 5$ is a path of length 5 from 1 to 5 (but this is not simple)



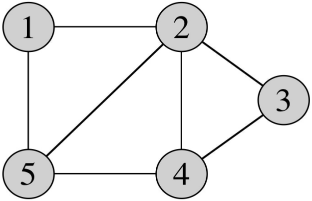
Example

- $1, 2, 3, 4, 2, 5$ is a path of length 5 from 1 to 5 (but this is not simple)
- $1, 2, 5$ is a simple path of length 2 from 1 to 5



Example

- $1, 2, 3, 4, 2, 5$ is a path of length 5 from 1 to 5 (but this is not simple)
- $1, 2, 5$ is a simple path of length 2 from 1 to 5
- Distance of 5 from 1 is 1



Cycles

A cycle C in a graph G is a path $v_1, v_2, \ldots, v_k$ if

1. $v_1 = v_k$ and
2. all edges in C are distinct

Cycles

A cycle C in a graph G is a path $v_1, v_2, \ldots, v_k$ if

1. $v_1 = v_k$ and
2. all edges in C are distinct

- C is said to be simple if $v_2, \ldots, v_k$ are all distinct

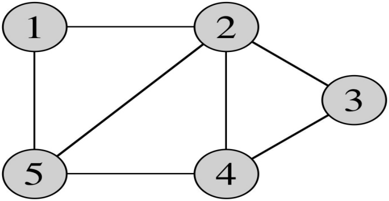
Cycles

A cycle C in a graph G is a path $v_1, v_2, \ldots, v_k$ if

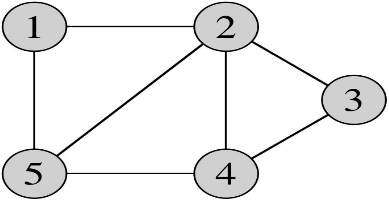
1. $v_1 = v_k$ and
2. all edges in C are distinct

- C is said to be simple if $v_2, \ldots, v_k$ are all distinct
- A graph with no cycles is said to be acyclic

Example

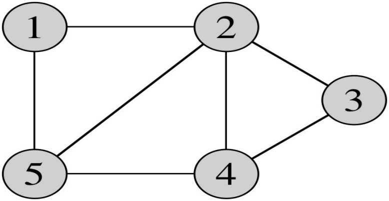


Example



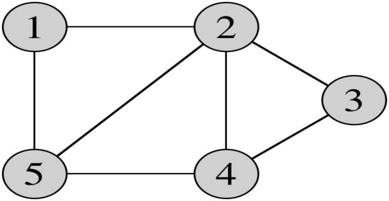
- 1, 2, 3, 4, 2, 5, 1 is a cycle (but not simple)

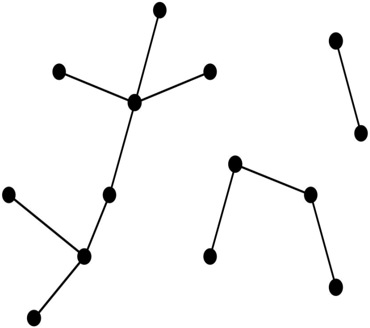
Example



- $1, 2, 3, 4, 2, 5, 1$ is a cycle (but not simple)
- $1, 2, 5, 1$ is a simple cycle

Example





- $1, 2, 3, 4, 2, 5, 1$ is a cycle (but not simple)
- $1, 2, 5, 1$ is a simple cycle

An acyclic graph

Connectivity

A vertex v is said to be reachable from u if there is a path from u to v

Connectivity

A vertex v is said to be reachable from u if there is a path from u to v

Fact: For undirected graphs, v is reachable from u iff u is reachable from v

Connectivity

A vertex v is said to be reachable from u if there is a path from u to v

Fact: For undirected graphs, v is reachable from u iff u is reachable from v

The graph G is said to be connected if every vertex is reachable from every other vertex

Connected components

The connected component of vertex u is the set $con(u)$ where

$$con(u) = \{v \mid v \text{ is reachable from } u\}$$

Connected components

The connected component of vertex u is the set $con(u)$ where

$$con(u) = \{v \mid v \text{ is reachable from } u\}$$

The connected components of a graph G is the set

$$\{con(u) \mid u \text{ is a vertex of } G\}$$

Connected components

The connected component of vertex u is the set $con(u)$ where

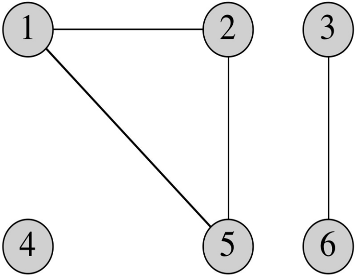
$$con(u) = \{v \mid v \text{ is reachable from } u\}$$

The connected components of a graph G is the set

$$\{con(u) \mid u \text{ is a vertex of } G\}$$

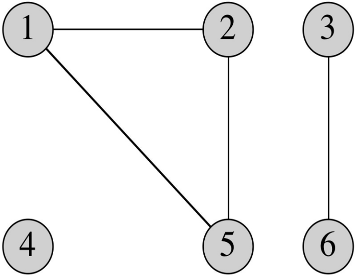
Fact: For undirected graphs, $con(u) = con(v)$ iff v is reachable from u

Example



Example

$$
\begin{aligned}
con(1) &= \{1, 2, 5\} \\
con(2) &= \{1, 2, 5\} \\
con(3) &= \{3, 6\} \\
con(4) &= \{4\} \\
con(5) &= \{1, 2, \} \\
con(6) &= \{3, 6\}
\end{aligned}
$$



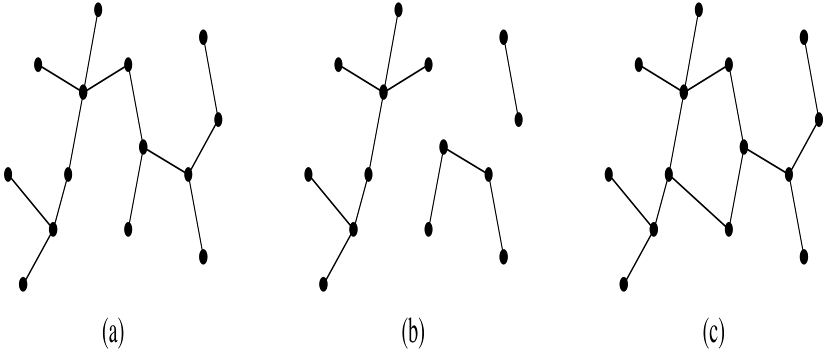
Trees

Trees

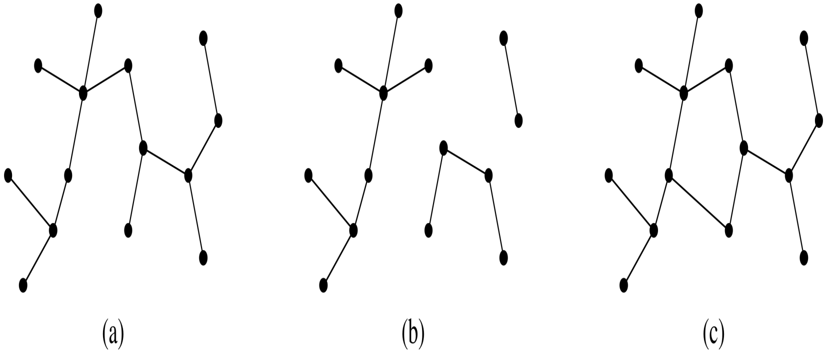
An acyclic graph is called a forest

A tree is a graph that is acyclic and connected

Example

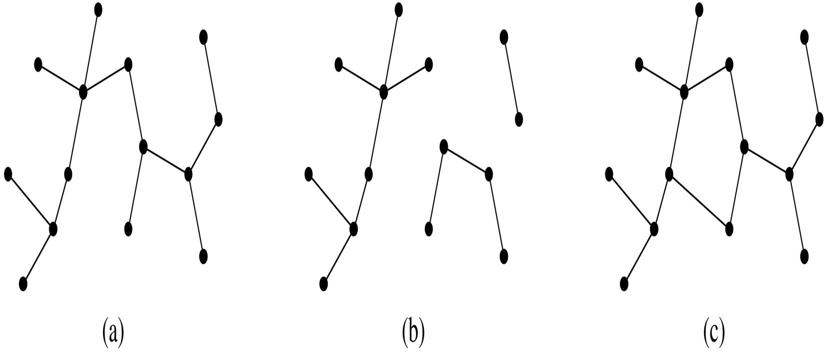


Example



(a) A tree

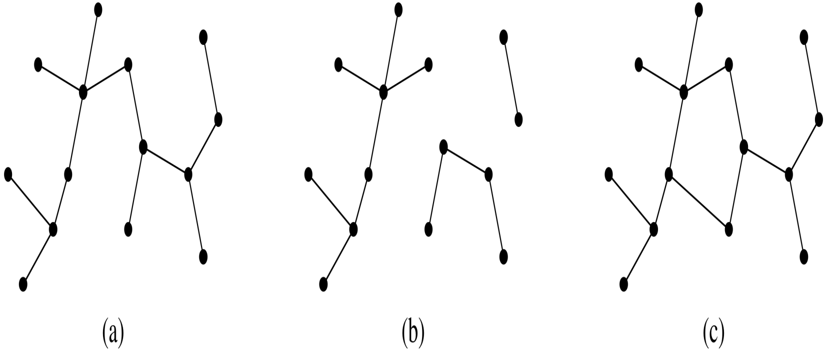
Example



(a) A tree

(b) A forest

Example



(a) A tree

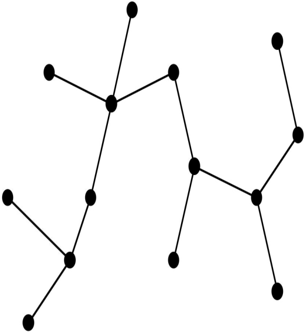
(b) A forest

(c) Neither a tree nor a forest

Properties of trees

The following statements are equivalent

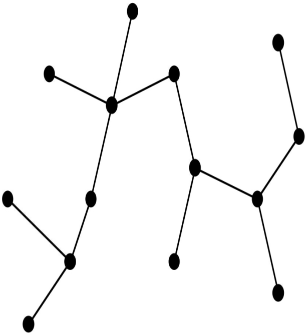
- $T = (V, E)$ is a tree



Properties of trees

The following statements are equivalent

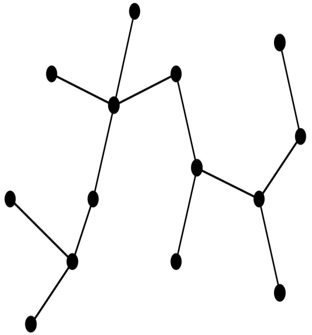
- $T = (V, E)$ is a tree
- For any two vertices u, v of T, there is a unique simple path from u to v



Properties of trees

The following statements are equivalent

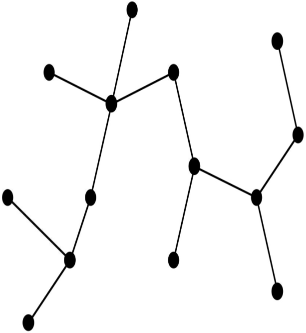
- $T = (V, E)$ is a tree
- For any two vertices u, v of T, there is a unique simple path from u to v
- T is connected, but removing any edge makes T disconnected



Properties of trees

The following statements are equivalent

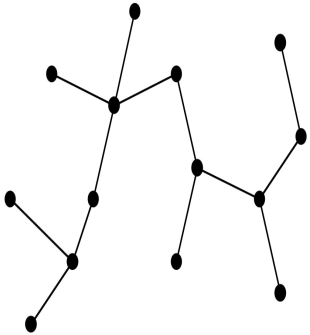
- $T = (V, E)$ is a tree
- For any two vertices u, v of T, there is a unique simple path from u to v
- T is connected, but removing any edge makes T disconnected
- T is connected and $|E| = |V| - 1$



Properties of trees

The following statements are equivalent

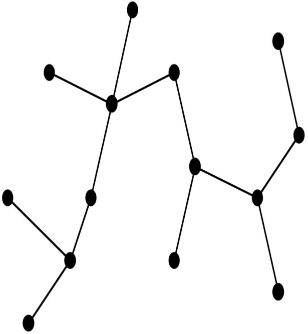
- $T = (V, E)$ is a tree
- For any two vertices u, v of T, there is a unique simple path from u to v
- T is connected, but removing any edge makes T disconnected
- T is connected and $|E| = |V| - 1$
- T is acyclic and $|E| = |V| - 1$



Properties of trees

The following statements are equivalent

- $T = (V, E)$ is a tree
- For any two vertices u, v of T, there is a unique simple path from u to v
- T is connected, but removing any edge makes T disconnected
- T is connected and $|E| = |V| - 1$
- T is acyclic and $|E| = |V| - 1$
- T is acyclic, but if any new edge is added to the graph then the resulting graph is acyclic

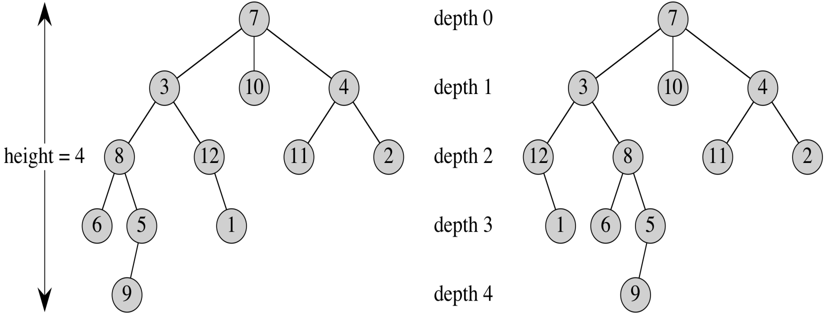


Rooted Trees

A rooted tree is a tree with a designated vertex r as root. In a rooted tree the edges are assumed to be oriented away from the root

- u is said to be parent of v if (u, v) is an edge, and u appears before v in the path from r to v. In such a case, v is said to be a child of u
- u is an ancestor of v if u appears on the path from r to v. In such a case,v is also called a descendent of u
- A vertex with no children is said to be a leaf. A nonleaf vertex is said to be an internal vertex
- The length of a simple path from the root to a vertex u is called the depth of u
- A level of a tree consists of all vertices of a tree at the same depth
- The height of a vertex u is the length of the longest simple downward path from u to a leaf
- The height of a tree is the height of the root

Example


height = 4
7
3
10
4
8
12
11
2
6
5
1
9
depth 0
depth 1
depth 2
depth 3
depth 4
7
3
10
4
12
8
11
2
1
6
5
9

(a)

(b)

How to represent (store) a graph?

Let $G = (V, E)$ be a graph

Adjacency matrix representation

Let $n = |V|$

Assume that the vertices are numbered $1, 2, \ldots, n$

How to represent (store) a graph?

Let $G = (V, E)$ be a graph

Adjacency matrix representation

Let $n = |V|$

Assume that the vertices are numbered $1, 2, \ldots, n$

A graph $G = (V, E)$ with n vertices and m edges can be represented by a $n \times n$ matrix A where

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

How to represent (store) a graph?

Let $G = (V, E)$ be a graph

Adjacency matrix representation

Let $n = |V|$

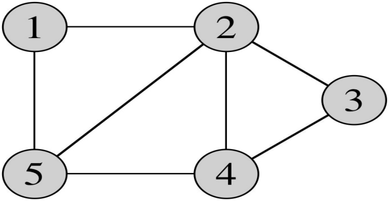
Assume that the vertices are numbered $1, 2, \ldots, n$

A graph $G = (V, E)$ with n vertices and m edges can be represented by a $n \times n$ matrix A where

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

A is the adjacency matrix of G

Example



	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

Adjacency matrix representation

Let $G = (V, E)$ be a graph

Let $n = |V|$

Assume that the vertices are numbered $1, 2, \ldots, n$

A graph $G = (V, E)$ with n vertices and m edges can be represented by a $n \times n$ matrix A where

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

- How much space this representation requires:
- How much time it takes to list all vertices adjacent to vertex i:
- How much time it takes to check if i and j are adjacent:

Adjacency matrix representation

Let $G = (V, E)$ be a graph

Let $n = |V|$

Assume that the vertices are numbered $1, 2, \ldots, n$

A graph $G = (V, E)$ with n vertices and m edges can be represented by a $n \times n$ matrix A where

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

- How much space this representation requires: $\Theta(n^2)$
- How much time it takes to list all vertices adjacent to vertex i:
- How much time it takes to check if i and j are adjacent:

Adjacency matrix representation

Let $G = (V, E)$ be a graph

Let $n = |V|$

Assume that the vertices are numbered $1, 2, \ldots, n$

A graph $G = (V, E)$ with n vertices and m edges can be represented by a $n \times n$ matrix A where

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

- How much space this representation requires: $\Theta(n^2)$
- How much time it takes to list all vertices adjacent to vertex i: $\Theta(n)$
- How much time it takes to check if i and j are adjacent:

Adjacency matrix representation

Let $G = (V, E)$ be a graph

Let $n = |V|$

Assume that the vertices are numbered $1, 2, \dots, n$

A graph $G = (V, E)$ with n vertices and m edges can be represented by a $n \times n$ matrix A where

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

- How much space this representation requires: $\Theta(n^2)$
- How much time it takes to list all vertices adjacent to vertex i: $\Theta(n)$
- How much time it takes to check if i and j are adjacent: $\Theta(1)$

Adjacency list representation

Let $G = (V, E)$ be a graph

Let $n = |V|$ and $m = |E|$

Assume that the vertices are numbered $1, 2, \ldots, n$

Adjacency list representation

Let $G = (V, E)$ be a graph

Let $n = |V|$ and $m = |E|$

Assume that the vertices are numbered $1, 2, \ldots, n$

We can represent G as an array *Adj* of n lists, one per each vertex

Adjacency list representation

Let $G = (V, E)$ be a graph

Let $n = |V|$ and $m = |E|$

Assume that the vertices are numbered $1, 2, \ldots, n$

We can represent G as an array *Adj* of n lists, one per each vertex

Adjacency list representation

Let $G = (V, E)$ be a graph

Let $n = |V|$ and $m = |E|$

Assume that the vertices are numbered $1, 2, \ldots, n$

We can represent G as an array Adj of n lists, one per each vertex

For each $i \in V$, the list $Adj[i]$ is the list of vertices adjacent to i. This list is the adjacency list of i

Adjacency list representation

Let $G = (V, E)$ be a graph

Let $n = |V|$ and $m = |E|$

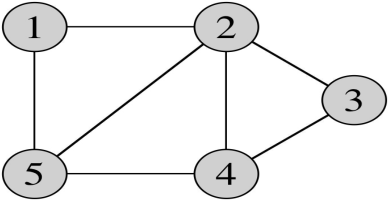
Assume that the vertices are numbered $1, 2, \ldots, n$

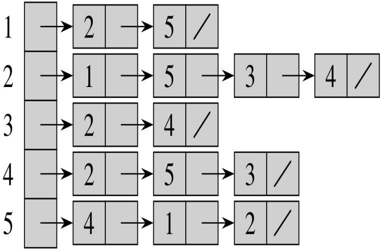
We can represent G as an array Adj of n lists, one per each vertex

For each $i \in V$, the list $Adj[i]$ is the list of vertices adjacent to i. This list is the adjacency list of i

In pseudocode, denote the array as attribute $G.Adj$

Example





Adjacency list representation

Let $G = (V, E)$ be a graph

Let $n = |V|$ and $m = |E|$

Assume that the vertices are numbered $1, 2, \ldots, n$

We can represent G as an array Adj of n lists, one per each vertex

For each $i \in V$, the list $Adj[i]$ is the list of vertices adjacent to i. This list is the adjacency list of i

- How much space this representation requires:
- How much time it takes to list all vertices adjacent to vertex i:
- How much time it takes to check if i and j are adjacent:

Adjacency list representation

Let $G = (V, E)$ be a graph

Let $n = |V|$ and $m = |E|$

Assume that the vertices are numbered $1, 2, \ldots, n$

We can represent G as an array Adj of n lists, one per each vertex

For each $i \in V$, the list $Adj[i]$ is the list of vertices adjacent to i. This list is the adjacency list of i

- How much space this representation requires: $\Theta(m + n)$
- How much time it takes to list all vertices adjacent to vertex i:
- How much time it takes to check if i and j are adjacent:

Adjacency list representation

Let $G = (V, E)$ be a graph

Let $n = |V|$ and $m = |E|$

Assume that the vertices are numbered $1, 2, \ldots, n$

We can represent G as an array Adj of n lists, one per each vertex

For each $i \in V$, the list $Adj[i]$ is the list of vertices adjacent to i. This list is the adjacency list of i

- How much space this representation requires: $\Theta(m + n)$
- How much time it takes to list all vertices adjacent to vertex i: $\Theta(degree(i))$
- How much time it takes to check if i and j are adjacent:

Adjacency list representation

Let $G = (V, E)$ be a graph

Let $n = |V|$ and $m = |E|$

Assume that the vertices are numbered $1, 2, \ldots, n$

We can represent G as an array *Adj* of n lists, one per each vertex

For each $i \in V$, the list $Adj[i]$ is the list of vertices adjacent to i. This list is the adjacency list of i

- How much space this representation requires: $\Theta(m + n)$
- How much time it takes to list all vertices adjacent to vertex i: $\Theta(degree(i))$
- How much time it takes to check if i and j are adjacent: $\Theta(degree(i))$

Which representation to use?

- $m + n < n^2 + n$

Which representation to use?

- $m + n < n^2 + n$
- So if the graphs is sparse, that is contain very few edges then the adjacency list representation is preferred otherwise we prefer the adjacency matrix representation

Which representation to use?

- $m + n < n^2 + n$
- So if the graphs is sparse, that is contain very few edges then the adjacency list representation is preferred otherwise we prefer the adjacency matrix representation
- We will use adjacency list representation for most cases. This is the default representation

Which representation to use?

- $m + n < n^2 + n$
- So if the graphs is sparse, that is contain very few edges then the adjacency list representation is preferred otherwise we prefer the adjacency matrix representation
- We will use adjacency list representation for most cases. This is the default representation
- Sometimes, however, graph algorithms become easier when using adjacency matrix representation (and we will be clear when we use this representation)

Representing graph attributes

Graph algorithms usually need to maintain attributes for vertices and/or edges.

- Denote attribute a of vertex v by $v.a$
- Denote attribute f of edge u by $u.f$

Implementing graph attributes

- No one best way to implement. Depends on programming language, algorithm etc..
- If representing the graph with adjacency lists, can represent vertex attributes in additional arrays that parallel the Adj array, e.g.,
  - If n is the number of vertices which are numbered $1, 2, \ldots n$ in Adj then store the attribute a in another array $a[1\,..\,n]$ with $a.i$ storing the value of attribute a for the vertex i

Fundamental graph algorithms

1. Given graph G and vertices s and t, is t reachable from s?
2. Given graph G and vertex s, compute $con(s)$.
3. Given graph G, compute the connected components of G.

A first attempt at search

Input: A graph $G = (V, E)$ and a source vertex $s \in V$
Output: The connected component of s, $con(s)$

A first attempt at search

Input: A graph $G = (V, E)$ and a source vertex $s \in V$
Output: The connected component of s, $con(s)$

$\textsc{Search}(V, E, s)$

$con(s) = \{s\}$
while there is an edge (u, v) such that $u \in con(s)$ and $v \notin con(s)$
Add v to $con(s)$

A first attempt at search

Input: A graph $G = (V, E)$ and a source vertex $s \in V$
Output: The connected component of s, $con(s)$

$\textsc{Search}(V, E, s)$
  $con(s) = \{s\}$
  while there is an edge (u, v) such that $u \in con(s)$ and $v \notin con(s)$
    Add v to $con(s)$

What is missing in this algorithm?

A first attempt at search

Input: A graph $G = (V, E)$ and a source vertex $s \in V$
Output: The connected component of s, $con(s)$

$\textsc{Search}(V, E, s)$

  $con(s) = \{s\}$
  while there is an edge (u, v) such that $u \in con(s)$ and $v \notin con(s)$
    Add v to $con(s)$

What is missing in this algorithm?
The order in which edges are considered is left unspecified

Breadth First Search (BFS)

Key idea: Processes the vertices in the graph in the order of their shortest distance from the vertex s (the start vertex)

- Send a wave out from s
- First hits all vertices 1 edge from s
- From there, hits all vertices 2 edges from s

Breadth First Search (BFS)

Key idea: Processes the vertices in the graph in the order of their shortest distance from the vertex s (the start vertex)

- Send a wave out from s
- First hits all vertices 1 edge from s
- From there, hits all vertices 2 edges from s
- Use a queue Q to maintain the wavefront
  - $v \in Q$ if and only if wave has hit v but has not come out of v

BFS algorithm in pseudocode

Input: A graph $G = (V, E)$ and a source vertex $s \in V$

Output: For each $v \in V$, $v.d$ is the distance of v from u

BFS algorithm in pseudocode

Input: A graph $G = (V, E)$ and a source vertex $s \in V$

Output: For each $v \in V$, $v.d$ is the distance of v from u

$\mathrm{BFS}(V, E, s)$

```
// Distance of s from s is 0
s.d = 0
// Initialize other nodes as unreachable
for each u ∈ V \ {s}
    u.d = ∞
// Queue gets initialized
Q = ∅
ENQUEUE(Q, s)
// Process the vertices in the queue
while Q ≠ ∅
    u = DEQUEUE(Q)
    for each v ∈ G.Adj[u]
        if v.d == ∞
            v.d = u.d + 1
            ENQUEUE(Q, v)
```

BFS algorithm in pseudocode

Input: A graph $G = (V, E)$ and a source vertex $s \in V$

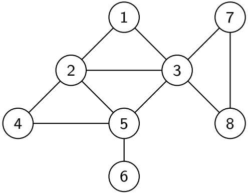
Output: For each $v \in V$, $v.d$ is the distance of v from u

How do we get $con(s)$?

$\text{BFS}(V, E, s)$

```
// Distance of s from s is 0
s.d = 0
// Initialize other nodes as unreachable
for each u ∈ V \ {s}
    u.d = ∞
// Queue gets initialized
Q = ∅
ENQUEUE(Q, s)
// Process the vertices in the queue
while Q ≠ ∅
    u = DEQUEUE(Q)
    for each v ∈ G.Adj[u]
        if v.d == ∞
            v.d = u.d + 1
            ENQUEUE(Q, v)
```

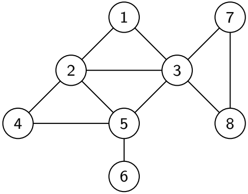
BFS: An Example

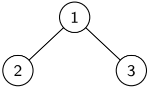


The queue at the beginning of each operation of the **while** loop

1. [1]

BFS: An Example





The queue at the beginning of each operation of the **while** loop

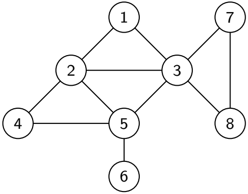
1. [1]
2. [2,3]

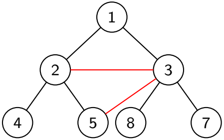
BFS: An Example

The queue at the beginning of each operation of the **while** loop

1. [1]
2. [2,3]
3. [3,4,5]

BFS: An Example





The queue at the beginning of each operation of the **while** loop

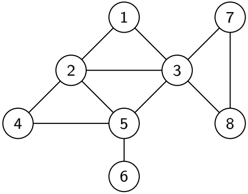
1. [1]
2. [2,3]
3. [3,4,5]
4. [4,5,7,8]

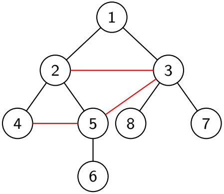
BFS: An Example

The queue at the beginning of each operation of the **while** loop

1. [1]
2. [2,3]
3. [3,4,5]
4. [4,5,7,8]
5. [5,7,8]

BFS: An Example

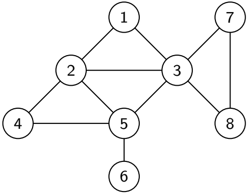


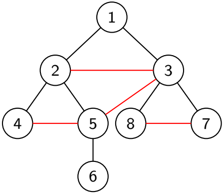


The queue at the beginning of each operation of the **while** loop

1.	[1]	4.	[4,5,7,8]
2.	[2,3]	5.	[5,7,8]
3.	[3,4,5]	6.	[7,8,6]

BFS: An Example

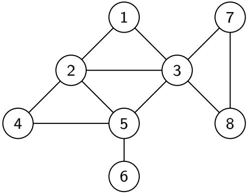


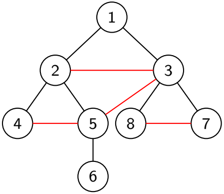


The queue at the beginning of each operation of the **while** loop

1. [1]
2. [2,3]
3. [3,4,5]
4. [4,5,7,8]
5. [5,7,8]
6. [7,8,6]
7. [8,6]

BFS: An Example

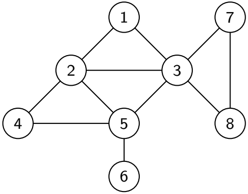


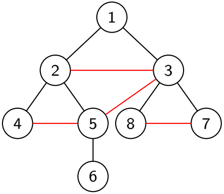


The queue at the beginning of each operation of the **while** loop

1. [1]
2. [2,3]
3. [3,4,5]
4. [4,5,7,8]
5. [5,7,8]
6. [7,8,6]
7. [8,6]
8. [6]

BFS: An Example





The queue at the beginning of each operation of the **while** loop

1. [1]
2. [2,3]
3. [3,4,5]
4. [4,5,7,8]
5. [5,7,8]
6. [7,8,6]
7. [8,6]
8. [6]
9. []

BFS: An Example

The queue at the beginning of each operation of the **while** loop

1.	[1]	4.	[4,5,7,8]	7.	[8,6]
2.	[2,3]	5.	[5,7,8]	8.	[6]
3.	[3,4,5]	6.	[7,8,6]	9.	[]

Breadth First Search Tree is the tree with the black edges as the set of edges

Properties of BFS

Properties of BFS

- The BFS search tree contains exactly the set of vertices $con(s)$

Properties of BFS

- The BFS search tree contains exactly the set of vertices $con(s)$
- If $u.d < v.d$ then u is visited before v

Properties of BFS

- The BFS search tree contains exactly the set of vertices $con(s)$
- If $u.d < v.d$ then u is visited before v
- For every vertex u, $u.d$ is the length of shortest path from s to u

Properties of BFS

- The BFS search tree contains exactly the set of vertices $con(s)$
- If $u.d < v.d$ then u is visited before v
- For every vertex u, $u.d$ is the length of shortest path from s to u
- For every edge $e = (u, v)$, either e is an edge in the BFS search tree or $u.d$ and $v.d$ differ by at most 1

Properties of BFS

- The BFS search tree contains exactly the set of vertices $con(s)$
- If $u.d < v.d$ then u is visited before v
- For every vertex u, $u.d$ is the length of shortest path from s to u
- For every edge $e = (u, v)$, either e is an edge in the BFS search tree or $u.d$ and $v.d$ differ by at most 1
- Running time is $O(|V| + |E|)$

Properties of BFS

- The BFS search tree contains exactly the set of vertices $con(s)$
- If $u.d < v.d$ then u is visited before v
- For every vertex u, $u.d$ is the length of shortest path from s to u
- For every edge $e = (u, v)$, either e is an edge in the BFS search tree or $u.d$ and $v.d$ differ by at most 1
- Running time is $O(|V| + |E|)$
  - $O(|V|)$ because every vertex enqueued at most once

Properties of BFS

- The BFS search tree contains exactly the set of vertices $con(s)$
- If $u.d < v.d$ then u is visited before v
- For every vertex u, $u.d$ is the length of shortest path from s to u
- For every edge $e = (u, v)$, either e is an edge in the BFS search tree or $u.d$ and $v.d$ differ by at most 1
- Running time is $O(|V| + |E|)$
  - $O(|V|)$ because every vertex enqueued at most once
  - $O(|E|)$ because every edge examined at most twice

Properties of BFS

- The BFS search tree contains exactly the set of vertices $con(s)$
- If $u.d < v.d$ then u is visited before v
- For every vertex u, $u.d$ is the length of shortest path from s to u
- For every edge $e = (u, v)$, either e is an edge in the BFS search tree or $u.d$ and $v.d$ differ by at most 1
- Running time is $O(|V| + |E|)$
  - $O(|V|)$ because every vertex enqueued at most once
  - $O(|E|)$ because every edge examined at most twice

Properties of BFS

- The BFS search tree contains exactly the set of vertices $con(s)$
- If $u.d < v.d$ then u is visited before v
- For every vertex u, $u.d$ is the length of shortest path from s to u
- For every edge $e = (u, v)$, either e is an edge in the BFS search tree or $u.d$ and $v.d$ differ by at most 1
- Running time is $O(|V| + |E|)$
  - $O(|V|)$ because every vertex enqueued at most once
  - $O(|E|)$ because every edge examined at most twice
- It requires extra $O(|V|)$ working space

How to solve fundamental graph algorithms with BFS

1. Given graph G and vertices s and t, is t reachable from s?

2. Given graph G and vertex s, compute $con(s)$.

3. Given graph G, compute the connected components of G.

How to solve fundamental graph algorithms with BFS

1. Given graph G and vertices s and t, is t reachable from s?

   Answer: t is reachable from s if $t.d$ is not ∞

2. Given graph G and vertex s, compute $con(s)$.

   Answer: $con(s)$ is just the set of all vertices t reachable from s

3. Given graph G, compute the connected components of G.

   Answer:

   - Compute the connected component of a vertex numbered 1.
   - Then pick the (smallest numbered) vertex not reachable from 1 and compute its connected component.
   - Keep going..

Depth first search (DFS)

Key idea:
Search deeper in the graph whenever possible
Start exploring the vertices in the graph from s

1. Check the most recently discovered vertex v
2. Pick a vertex adjacent to v which has as yet not been discovered and start exploring this vertex
3. Once all such vertices are explored then backtrack

Use color-coding

As DFS progresses, every vertex has a color

WHITE: undiscovered

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

BLACK: finished (have found everything reachable from it)

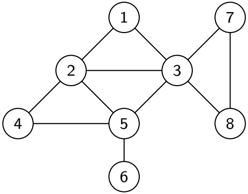
Pseudocode for DFS

Input: A graph $G = (V, E)$ and a source vertex $s \in V$
Output: For each $v \in V$, $v.color$ is BLACK if v is reachable from s and is WHITE otherwise

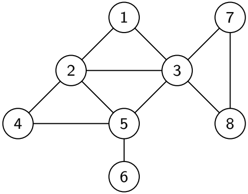
```
DFS(V, E, s)
    for each u ∈ V
        u.color = WHITE
    DFS_RECUR(V, E, s)
```

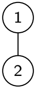
```
DFS_RECUR(V, E, u)
    // Discover u
    u.color = GRAY
    for each v ∈ G.Adj[u]
    // Explore (u, v)
        if v.color == WHITE
            DFS_RECUR(V, E,v)
    // Finish u
    u.color = BLACK
```

DFS: An Example

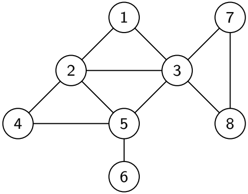


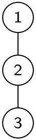
DFS: An Example



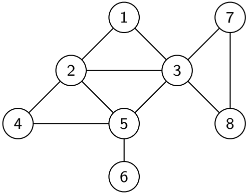


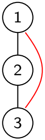
DFS: An Example



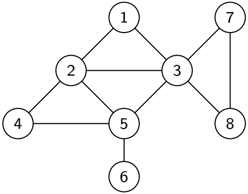


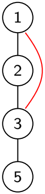
DFS: An Example



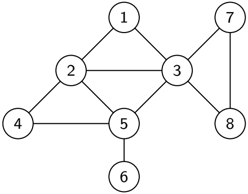


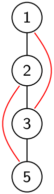
DFS: An Example



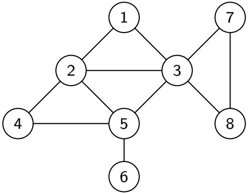


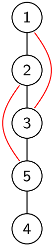
DFS: An Example



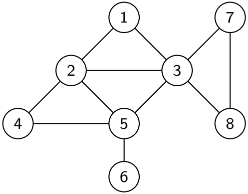


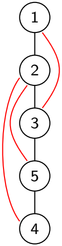
DFS: An Example



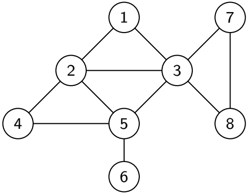


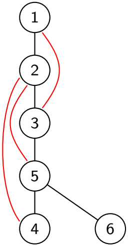
DFS: An Example



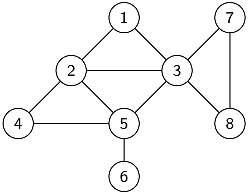


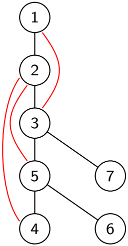
DFS: An Example



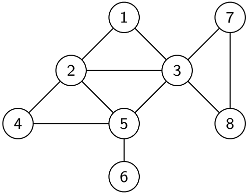


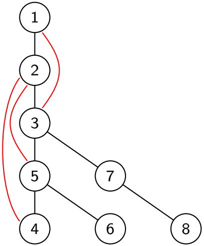
DFS: An Example



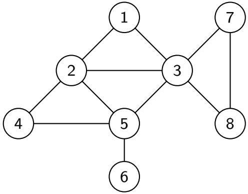


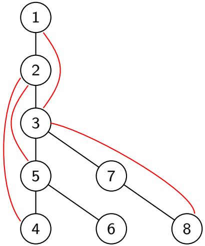
DFS: An Example





DFS: An Example





DFS: An Example

Depth First Search Tree is the set of black edges.

DFS and BFS Trees: An Example

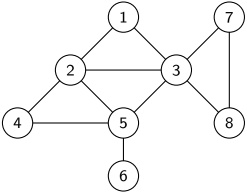


Figure : Graph G

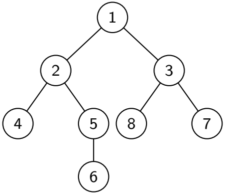


Figure : BFS Tree starting from 1

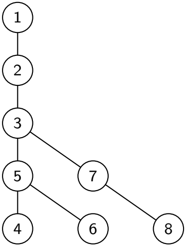


Figure : DFS Tree starting from 1

DFS tree with visit times

It will be useful to timestamp the vertices with the times during which they are visited

- $v.d$ = time at which v is discovered by the DFS algorithm
- $v.f$ = time at which processing of v finishes

DFS tree with visit times

It will be useful to timestamp the vertices with the times during which they are visited

- $v.d$ = time at which v is discovered by the DFS algorithm
- $v.f$ = time at which processing of v finishes

$\text{DFS}(V, E, s)$
  for each $u \in V$
    $u.color = \text{WHITE}$
  $time = 0$
  $\text{DFS\_VISIT}(V, E, s)$

$\text{DFS\_VISIT}(V, E, u)$
  $time = time + 1$
  $u.d = time$
  $u.color = \text{GRAY}$
  for each $v \in G.Adj[u]$
    if $v.color == \text{WHITE}$
      DFS_VISIT(V, E,v)
  $u.color = \text{BLACK}$
  $time = time + 1$
  $u.f = time$

Visit Times: Example

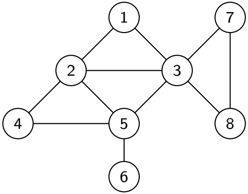


Figure : Graph G

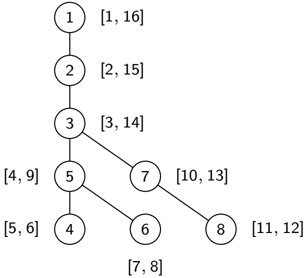
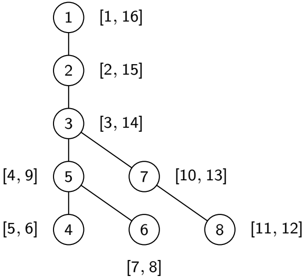


Figure : DFS Tree with visit times

Properties of the DFS tree: Parenthesis Theorem



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

- The intervals $[u.d, u.f]$ and $[v.d, v.f]$ are disjoint and neither of u or v is a descendant of the other in the DFS tree
- The interval $[u.d, u.f]$ contains $[v.d, v.f]$ and v is a descendant of u in the DFS tree
- The interval $[u.d, u.f]$ is contained in $[v.d, v.f]$ and u is a descendant of v in the DFS tree

So $v.d < u.d < v.f < u.f$ cannot happen

Other properties of the DFS

1. v is a descendant of u in DFS tree if and only if at time $u.d$, there is a path in the graph from u to v consisting of only white vertices

Other properties of the DFS

1. v is a descendant of u in DFS tree if and only if at time $u.d$, there is a path in the graph from u to v consisting of only white vertices
2. Running time is $O(|V| + |E|)$

Summary

- Graphs are a good way to model and pairwise relationships amongst a collection of individuals, objects
- We studied basic graph search algorithms BFS and DFS which give different strategies for solving fundamental graph algorithms
- This part was based on Appendix B.4 and Chapters 22.1, 22.2 and 22.3 from the book