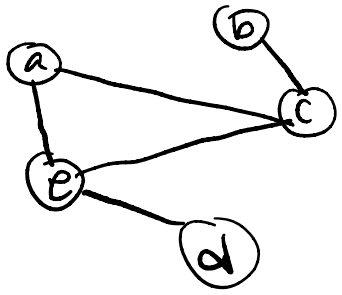
Graphs (also called Networks) represents connections

A (simple) **undirected graph** $G(V, E)$ consists of:

- a set V of vertices (alternatively called "nodes")
- a set E of edges consisting of **2-element subsets** of V

$V = \{a, b, c, d, e\}$
$E = \{\{a, c\}, \{b, c\}, \{e, a\}, \{d, e\}, \{e, c\}\}$

∴ order doesn't matter $\{a, c\} = \{c, a\}$

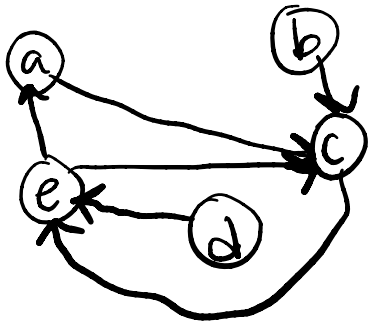


A **directed graph** $G(V, E)$ consists of:

- a set V of vertices (same as the undirected graph)
- a set E of edges consisting of **pairs** of elements from V (i.e. $E \subseteq V \times V$)

$V = \{a, b, c, d, e\}$
$E = \{(a, c), (b, c), (e, a), (d, e), (e, c), (c, e)\}$

∴ order does matter $(a, c) \neq (c, a)$



$|V| = n$
$n(n-1) = n^2$

In either case, we have $|E| \in O(|V|^2)$.

⇒ # edges is $O(|V|^2)$ (upper bound by #vertices squared)

Why Graphs?

- Graphs are useful when we have a bunch of things and we're interested in the relationships between pairs of those things:
  - **Example graphs:** Facebook's friend network, Google maps
  - **Applications:** navigation, AI, 3D graphics, games, flow networks, finite state machines, and many more!

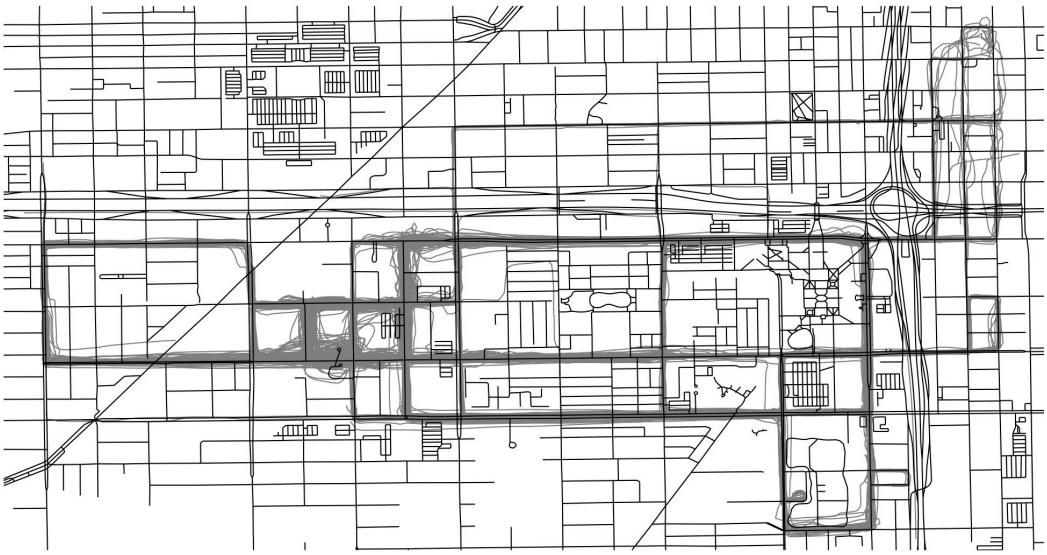


Fig. 1. Chicago road network and GPS traj. from `http://mapconstruction.org/`

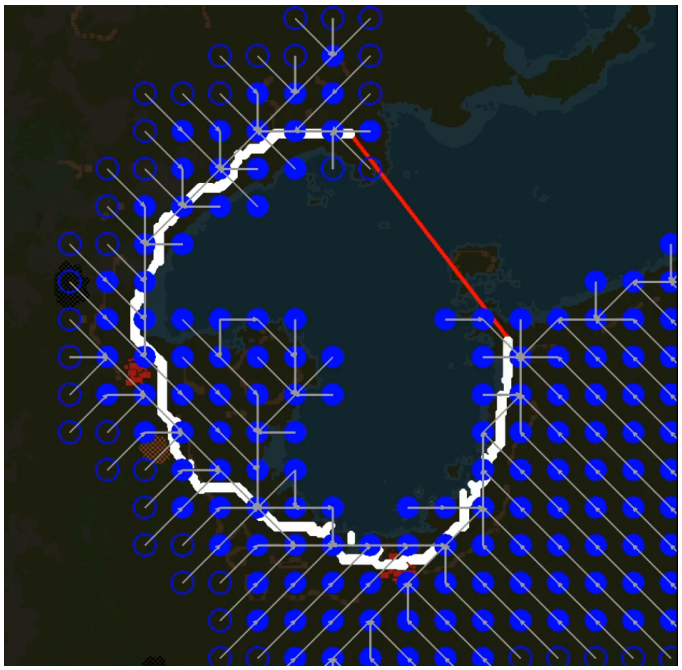


Fig. 2. Path finding in Factorio `https://factorio.com/blog/post/fff-317`

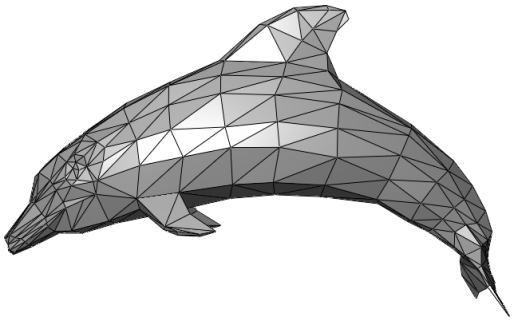


Fig. 3. Mesh of a dolphin
By en:User:Chrschn, Public Domain

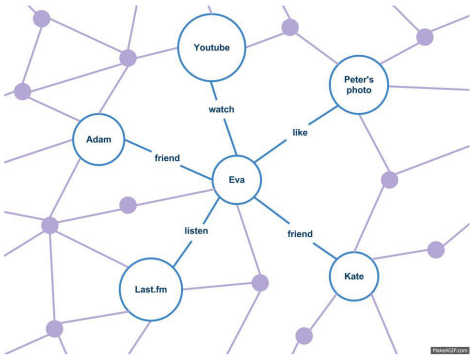


Fig. 4. Graphs of a social network.
By Festys, Own work, CC BY-SA 3.0

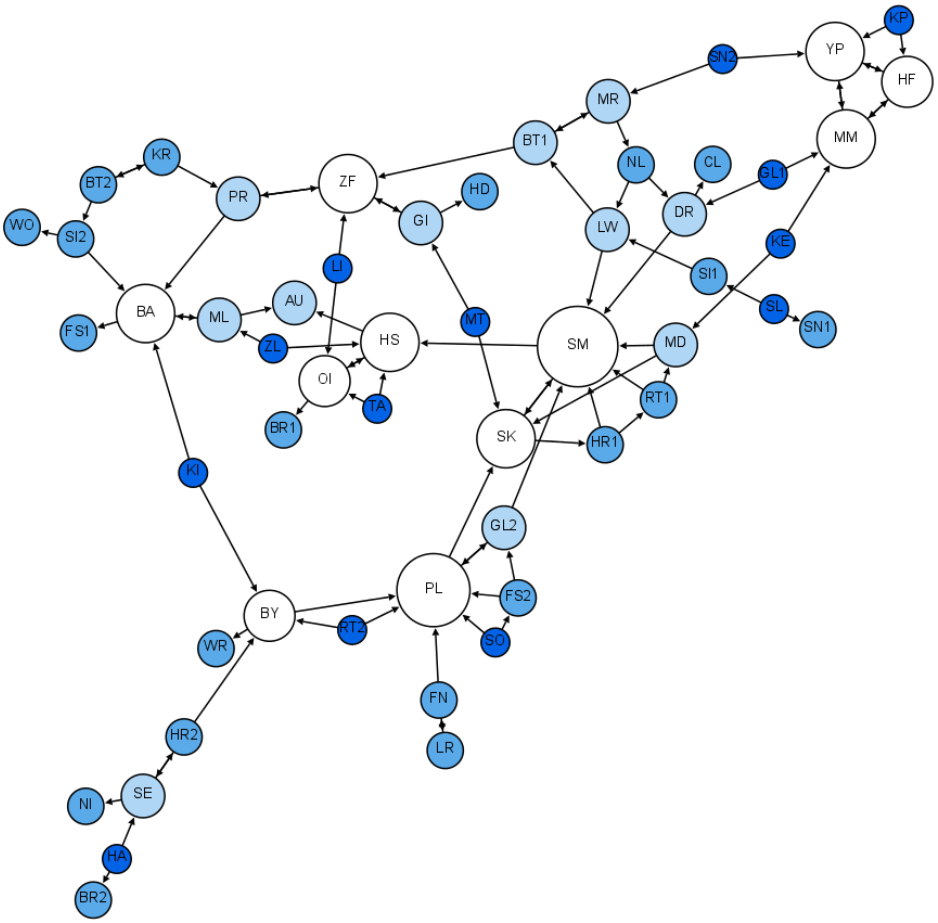


Fig. 5. Sociogram of 8th grade students' seating preferences
By Martin Grandjean, Own work, CC BY-SA 4.0

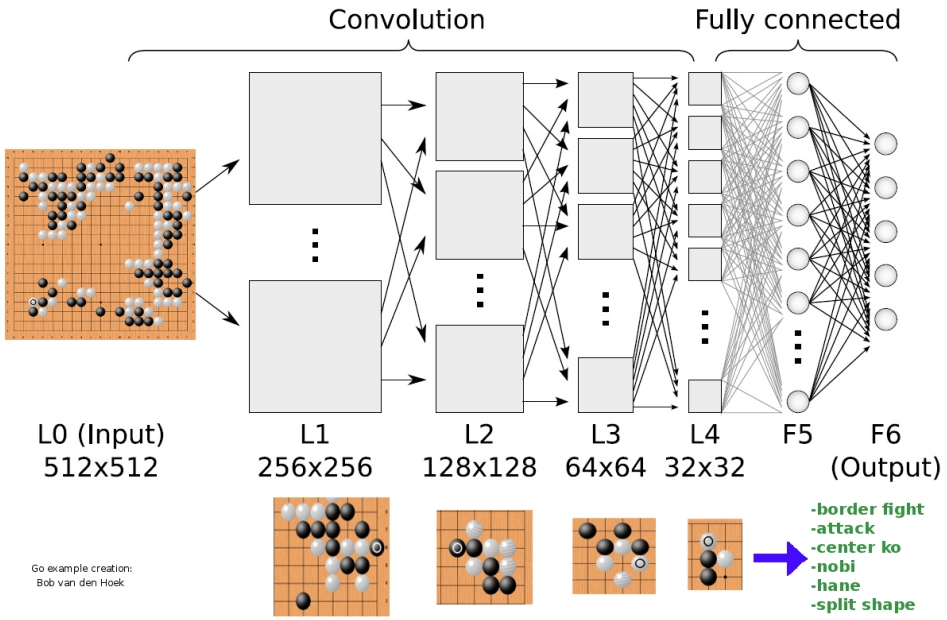


Fig. 6. AlphaGo Neural Network. Link to full article.

Bounds on the number of edges

Undirected graph:

$0 \leq |E| \leq$ number of 2-element subsets of V

$= \binom{|V|}{2} = \frac{(|V|)!}{2!(|V|-2)!} = \frac{(|V|)(|V|-1)}{2}$

choose 2 from V

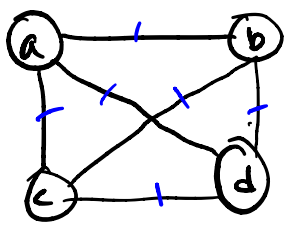
$\in O(|V|^2)$

Directed graph:

$0 \leq |E| \leq |V \times V| = |V| \cdot |V| = |V|^2$

Example: (complete undirected graph of 4 vertices)

every possible edge is connected



6 edges

$\Rightarrow \frac{4(4-1)}{2} = \frac{4(3)}{2} = \frac{12}{2} = 6$

$|V| = 4$ and $|E| = 6$

Directed graph # edges $= |V|^2 = 4^2 = 16$

Degree of a vertex:

$deg(v)$ =number of vertices adjacent to v (i.e., number of edges involving v)

in-degree(v) = $deg^-(v)$ =# of "incoming" edges to v
out-degree(v) = $deg^+(v)$ =# of "outgoing" edges from v

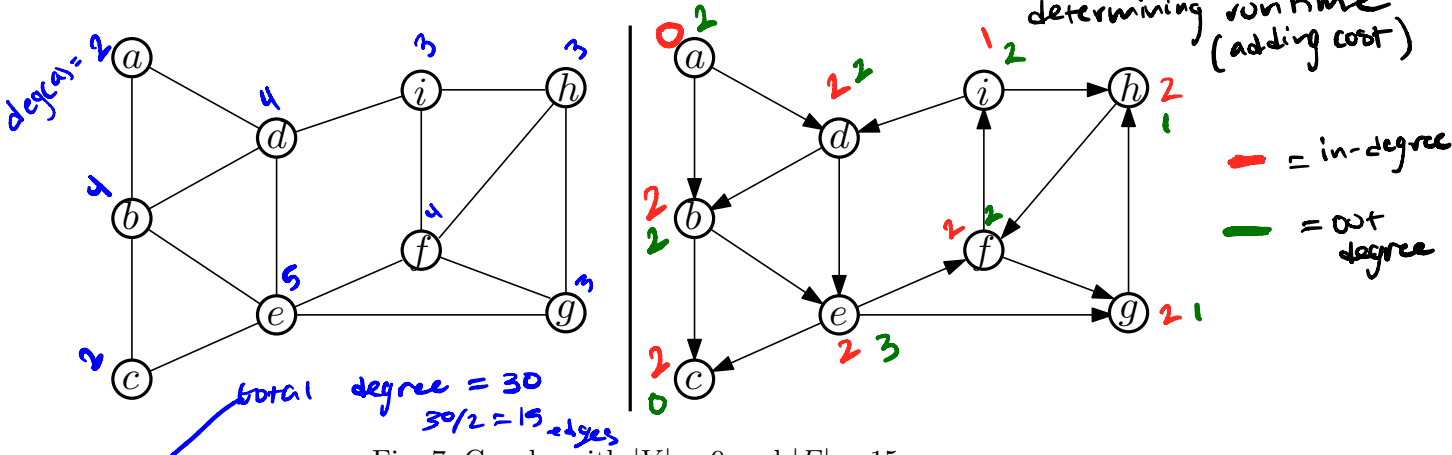


Fig. 7. Graphs with $|V| = 9$ and $|E| = 15$

Handshaking Lemma:

Idea: Every edge is counted twice

For undirected graphs:

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

For directed graphs:

$$\sum_{v \in V} deg^-(v) = |E|$$

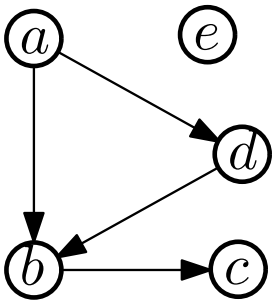
$$\sum_{v \in V} deg^+(v) = |E|$$

Adjacency-matrix representation

The adjacency matrix of a graph $G = (V, E)$, where $V = 1, 2, \ldots, n$, is the matrix $A[1, \ldots, n, 1, \ldots, n]$ given by

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

"Is there a connection from a → b ?"



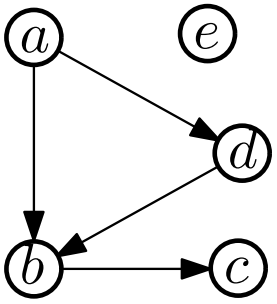
1.E

(from) \ (to)	a	b	c	d	e
a	0	1	0	1	0
b	0	0	1	0	0
c	0	0	0	0	0
d	0	1	0	0	0
e	0	0	0	0	0

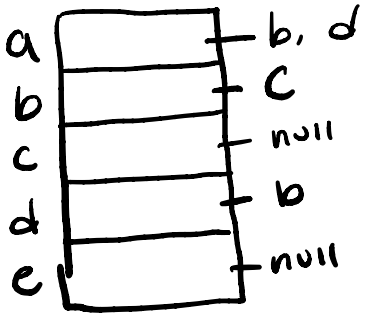
- $\Theta(|V|^2)$ storage (dense representation)
- O(1) to check connections

Adjacency-list representation

The adjacency list of a vertex $v \in V$ is the list $Adj[v]$ of vertices adjacent to v.



1.E



- For undirected graphs, $|Adj[v]| = deg(v)$
- For directed graphs, $|Adj[v]| = deg^+(v)$ — Out degrees (out going edges)
- Thus, the adjacency-list representation uses $\Theta(|V|+|E|)$ storage (sparse representation)
- => $O(deg^+(v))$ to check connections

Finding a path of length, $\leq k$, from a to b)

Algorithm 1 bool findKPath(int k, node a, node b)

```
1: if a == b then
2:     return true;
3: else if k == 0 then
4:     return false;
5: end if
6: for c = 0 to n − 1) do
7:     if adjacent(a,c) AND findKPath(k − 1, c, b) then
8:         return TRUE;
9:     end if
10: end for
11: return FALSE;
```

path found from a → b

We went nowhere, thus no path exists

loop through all vertices

Analysis

=> 2 recursive calls within a for loop that runs n times

=> n vertices e/a could make K recursive calls

=> $O(n^K)$

Is this an efficient algorithm?

=> No, this is not too efficient of an algorithm

Graph Traversal

Let $G = (V, E)$ be a (directed or undirected) graph, given in adjacency list representation.

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

A graph traversal visits every vertex:

- Breadth-first search (BFS)
- Depth-first search (DFS)

ways to traverse graph

Breadth-First Search (BFS)

// Setup BFS & call method that performs BFS

Algorithm 2 void BFS(graph $G = (V, E)$)

```
1: Mark all vertices in Mark all vertices in G as "unvisited" // time = 0
2: Initialize empty queue Q
3: for each vertex v ∈ V do
4:     if v is unvisited then
5:         visit v // time++
6:         Q.enqueue(v)
7:         BFS_iter(G)
8:     end if
9: end for
```

- Checks next unvisited vertex

Algorithm 3 void BFS_iter(graph $G = (V, E)$)

```
1: while Q is non-empty do
2:     v = Q.dequeue
3:     for each w adjacent to v do
4:         if w is unvisited then
5:             visit w // time++
6:             Add edge (v, w) to T
7:             Q.enqueue(w)
8:         end if
9:     end for
10: end while
```

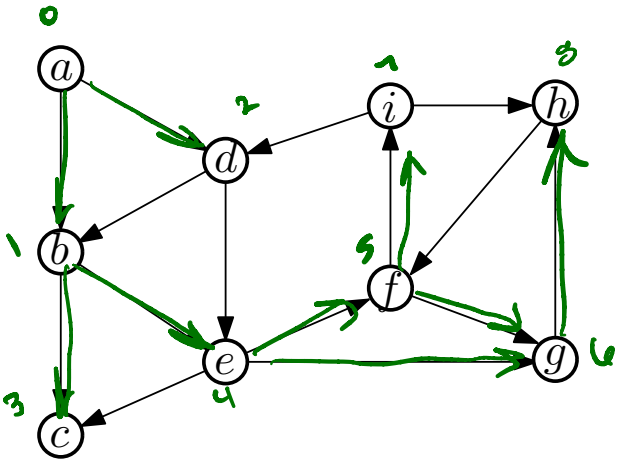
- mark node w/ unique time to show order we arrived
- creates a forest / Tree in graph

// where to start?
=> go alphabetically

queue: FIFO
=> add to rear
remove from front

● = visited
↳ upon adding to queue node is visited

Breadth-First Search Example (1)



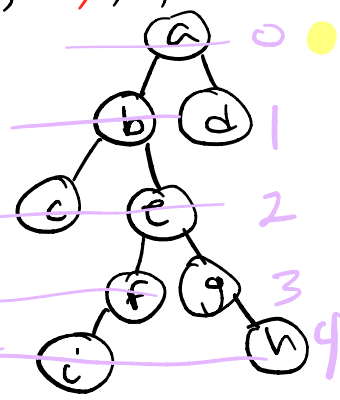
Time : 0, 1, 2, 3, 4, 5, 6, 7, 8

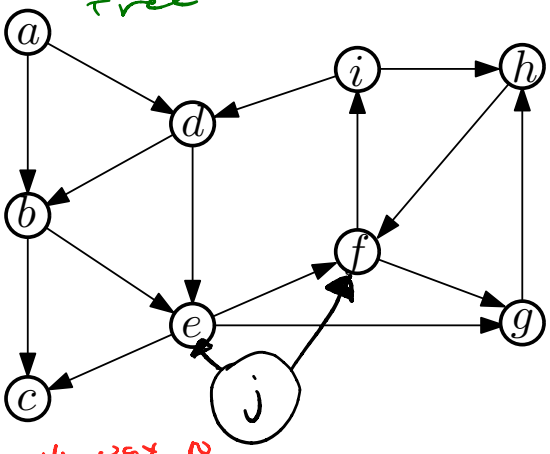
Q : ~~a~~, ~~b~~, ~~d~~, ~~e~~, ~~c~~, ~~f~~, ~~g~~, ~~i~~, ~~h~~

/ = dequeue

↳ highlighted edges form a tree

∴ If node visited already we just continue on





Time :

Q :

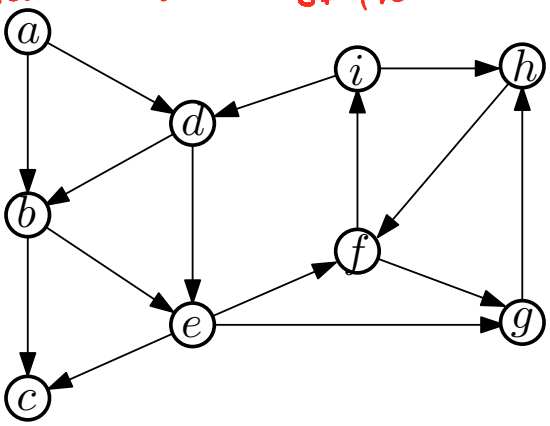
↳ sweep outwards getting links relative to a

= Minimum distance from a to certain node

↳ aka: link distance

=> No way to reach j, we would restart search starting @ j

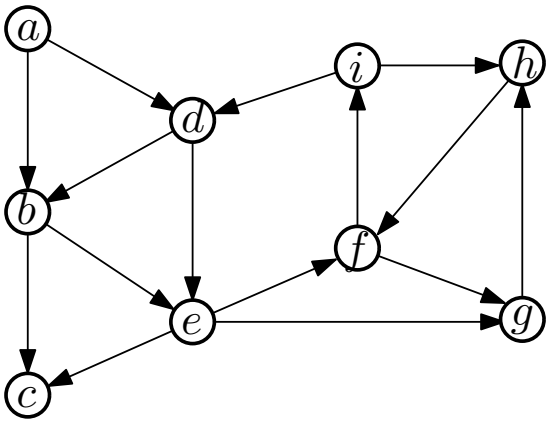
=> leads to forest of trees



Time :

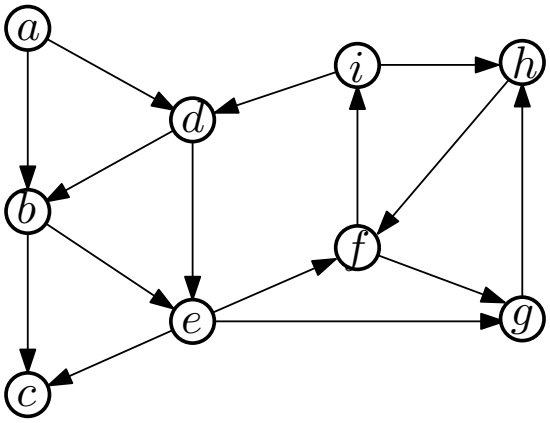
Q :

Breadth-First Search Example (2)



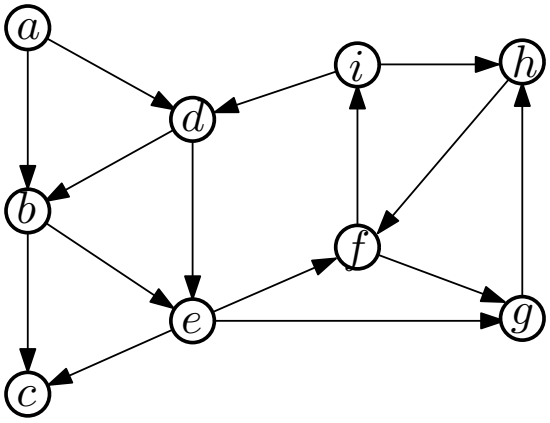
Time :

Q :



Time :

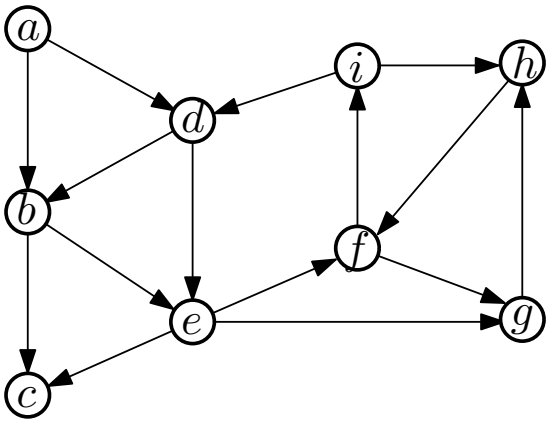
Q :



Time :

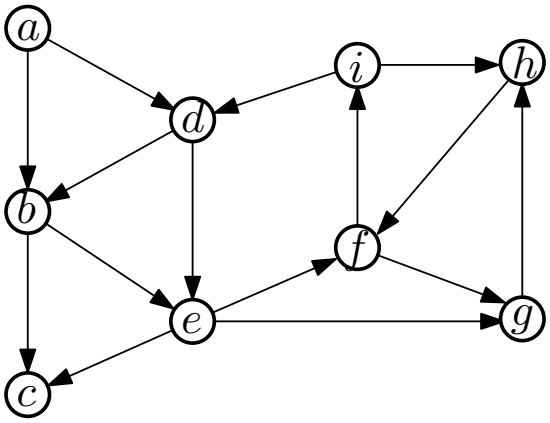
Q :

Breadth-First Search Example (3)



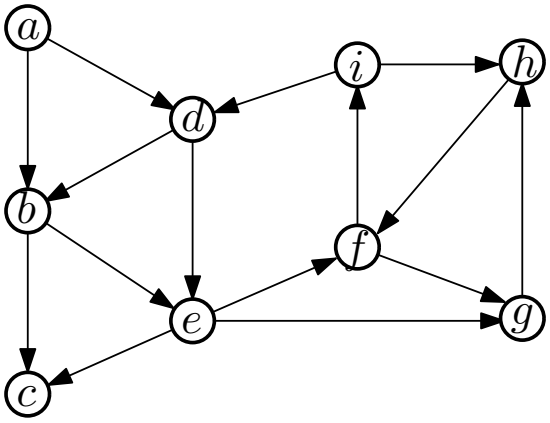
Time :

Q :



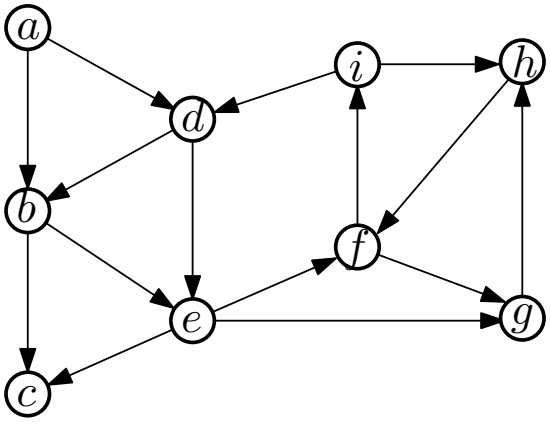
Time :

Q :



Time :

Q :



Time :

Q :

every vertice in queue

Runtime: $O(|V| + |E|)$

every edge traversed at least once

Breadth-First Search Runtime

Algorithm 4 void BFS(graph $G = (V, E)$)

```
1: Mark all vertices in Mark all vertices in G as "unvisited" // time = 0
2: Initialize empty queue Q
3: for each vertex v ∈ V do
4:     if v is unvisited then
5:         visit v // time++
6:         Q.enqueue(v)
7:         BFS_iter(G)
8:     end if
9: end for
```

n times

Every node in queue exactly once

Algorithm 5 void BFS_iter(graph $G = (V, E)$)

```
1: while Q is non-empty do
2:     v = Q.dequeue
3:     for each w adjacent to v do
4:         if w is unvisited then
5:             visit w // time++
6:             Add edge (v, w) to T
7:             Q.enqueue(w)
8:         end if
9:     end for
10: end while
```

$deg^+(v)$ = # outgoing connections from v

O(1)

∴ for loop runtime $= O(deg^+(v))$

contributes to runtime

every node goes into queue at least once

$$\Rightarrow \sum_{v \in V} deg^+(v) = |E|$$

sum for every vertex of cost

edges (handshaking lemma)

Depth-First Search (DFS)

Algorithm 6 void DFS(graph $G = (V, E)$)

```
1: Mark all vertices in Mark all vertices in G as "unvisited" // time = 0
2: for each vertex v ∈ V do
3:     if v is unvisited then
4:         DFS_rec(G, v)
5:     end if
6: end for
```

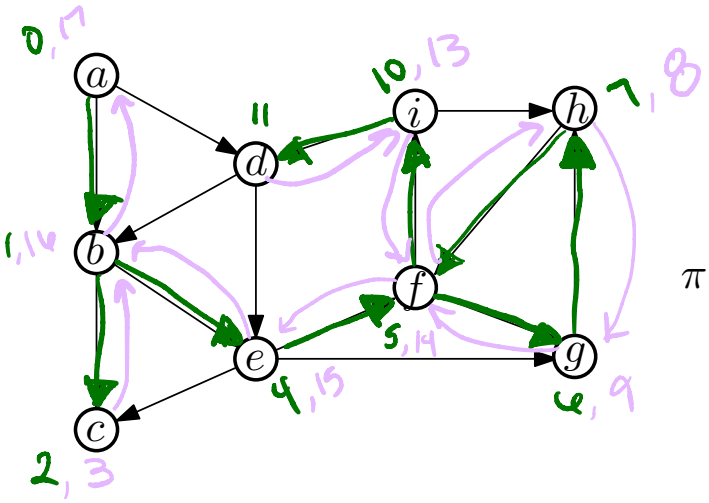
Algorithm 7 void DFS_rec(graph $G = (V, E)$, vertex v)

```
1: mark v as "visited" // d[v] = time      time++
2: for each w adjacent to v do
3:     if w is unvisited then
4:         Add edge (v, w) to T
5:         DFS_rec(G, w)
6:     end if
7: end for
8: mark v as "finished" // f[v] = time      time++
```

∴ Stack (LIFO)
add to top
remove from top

• Discovered time
• Finished time

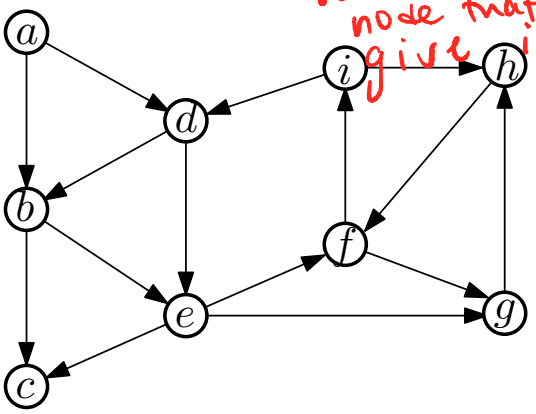
Depth-First Search Example (1)



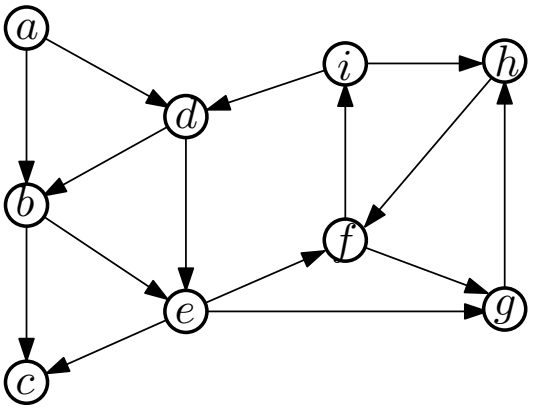
π : a b c d e f g h i

=> go back to b now

∴ if you reach node that can't go anywhere give it a finished time
=> look @ node c as example

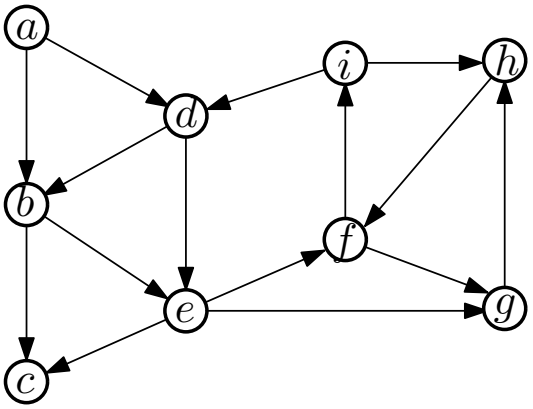


π : a b c d e f g h i

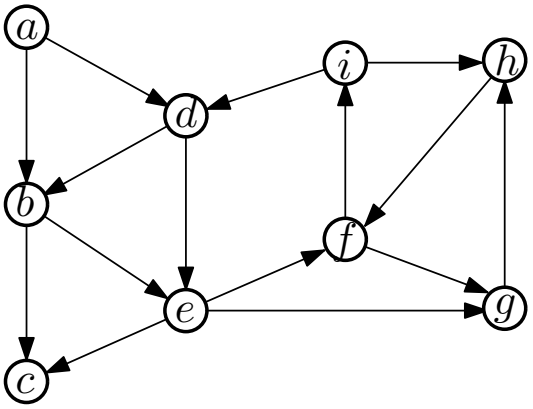


π : a b c d e f g h i

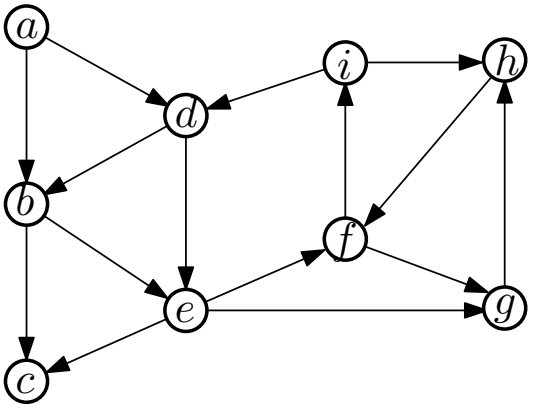
Depth-First Search Example (2)



π : a b c d e f g h i

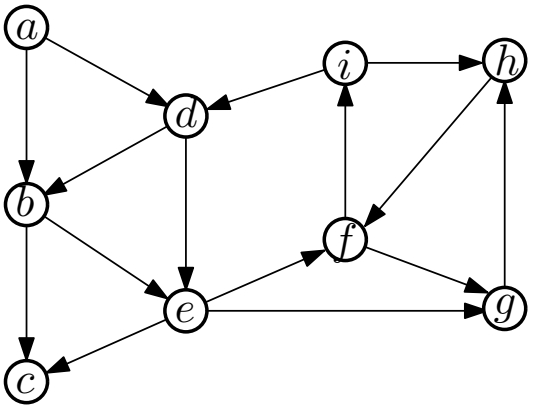


π : a b c d e f g h i

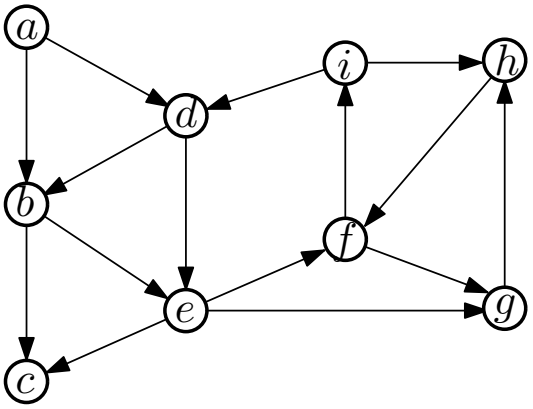


π : a b c d e f g h i

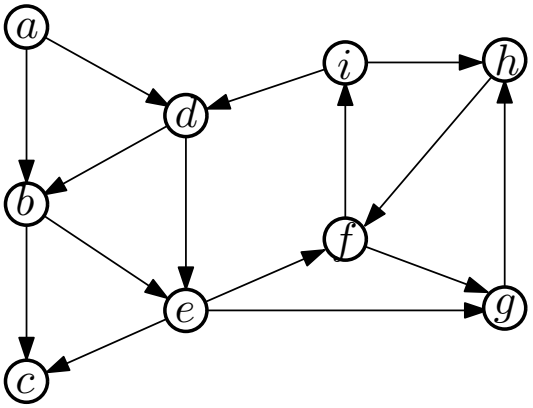
Depth-First Search Example (3)



π : a b c d e f g h i



π : a b c d e f g h i

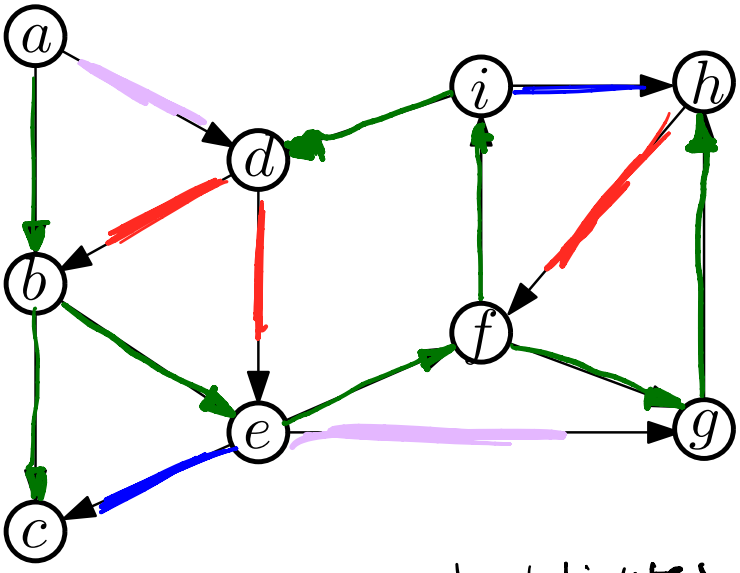


π : a b c d e f g h i

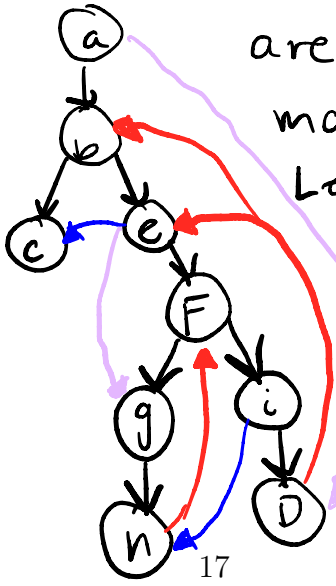
Depth-First Edge Classification (Application)

Edge $u \to v$ is a:

- **Tree edge**, if it is part of the depth-first forest.
- **Back edge**, if u connects to an ancestor v in a depth-first tree. It holds $d(u) > d(v)$ and $f(u) < f(v)$.
- **Forward edge**, if u connects to a descendant v in a depth-first tree. It holds $d(u) < d(v)$.
- **Cross edge**, if it is any other edge. It holds $d(u) > d(v)$ and $f(u) > f(v)$.



=>



- highlighted green edges are "Tree Edges", they make up the tree on Left

- "Cross Edge"
  - going to a sibling node
  - true if not on same level
  - I.E (I to h)

- "Forward edge"
  - From ancestor node to a descendant node (a to D) & (E to G)

- "Back Edge"
  - going back to a nodes great-grandparent (D to E) & (D to b)

Depth-First Search Runtime

Make sure all nodes visited

Algorithm 8 void DFS(graph $G = (V, E)$)

1: Mark all vertices in Mark all vertices in G as "unvisited" // $time = 0$
2: **for** each vertex $v \in V$ **do**
3: **if** v is unvisited **then**
4: DFS_iter(G, v)
5: **end if**
6: **end for**

Algorithm 9 void DFS_rec(graph $G = (V, E)$, vertex v)

1: mark v as "visited" // $d[v] = time$ $time++$ $-O(1)$
2: **for** each w adjacent to v **do** $-O(\deg^+(v)) = |E| = \#$ time loop runs
3: **if** w is unvisited **then**
4: Add edge (v, w) to T $-O(1)$
5: DFS_rec(G, w)
6: **end if** — marks node w as visited = visits each node exactly one time
7: **end for**
8: mark v as "finished" // $f[v] = time$ $time++$ $-O(1)$

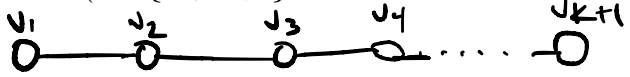
So 1 recursive call per node...

$$\Rightarrow \sum_{v \in V} \deg^+(v) = |E|$$

$\Rightarrow$ runtime: $O(|V| + |E|)$

$\Rightarrow$ This uses recursion to utilize call stack, but it may be better to create your own stack DS to do DFS

A **path** of length k in $G = (V, E)$ is a sequence of vertices $v_1, v_2, \ldots, v_{k+1}$ such that $(v_i, v_{i+1}) \in E$ (or $\{v_i, v_{i+1}\} \in E$ if G is undirected)for all $1 \leq i \leq k$.



A path is **simple** iff all vertices on the path are distinct.

A path $v_1, v_2, \ldots, v_{k-1}$ forms a **cycle** if $v_1 = v_{k-1}$ and $k \geq 2$.

A cycle is **simple** iff all vertices, except v_1 and v_{k-1}, are distinct.

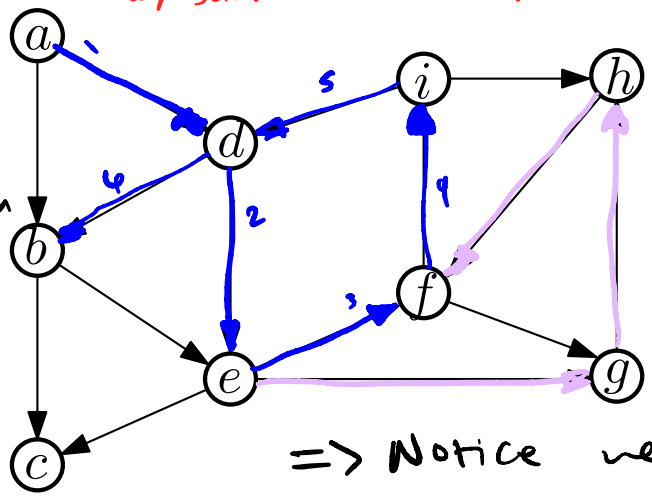
path sequence

If cycle starts / ends w/ same node = simple

∴ e, g, h, f

=> this is a "simple path" as we don't go through vertice twice

length = 3 Edges



∴ a, d, e, F, i, d, b

1 2 3 4 5 6

length = 6 Edges

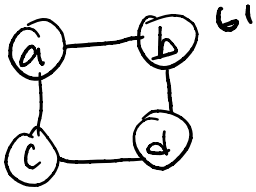
Not a "simple path" because of reason below

=> Notice we went through d twice to get to destination (b)

An undirected graph is **connected** if for every pair $u, v \in V$ there is a path from u to v. (Note that there is then a path from v to u)

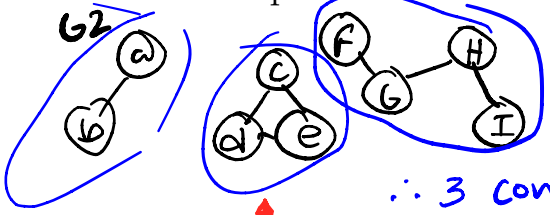
vertices

example:



G1

connected



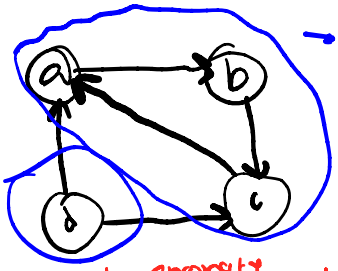
G2

∴ 3 connected components

Not connected

A directed graph is **strongly connected** if for every pair $u, v \in V$ there is a path from u to v and there is a path from v to u.

example:



→ 2 connected components

Not strongly connected

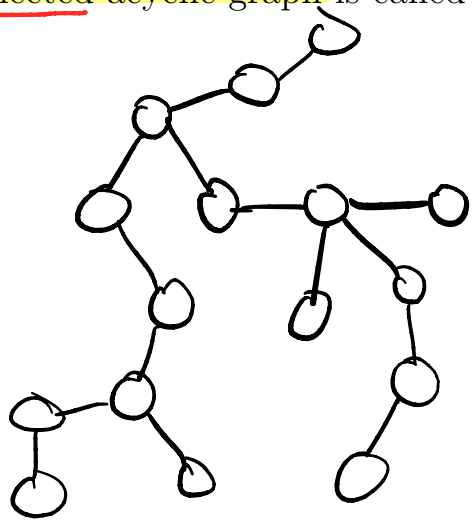
∴ d cannot be reached

No simple cycles in graph (no loops allowed)

An undirected connected acyclic graph is called a **tree**.

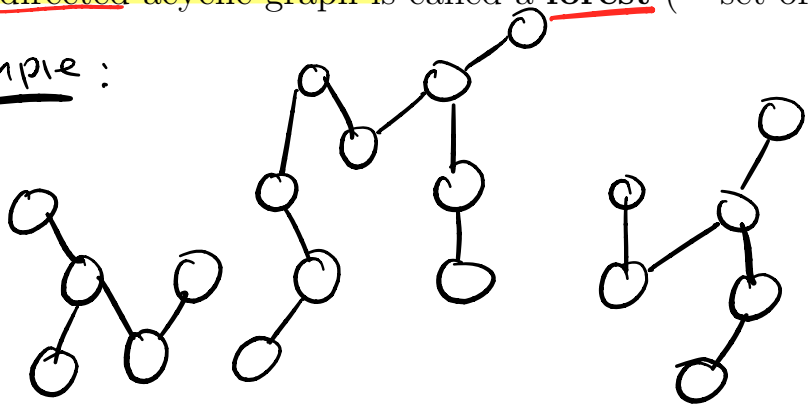
∴ no root & no usual tree restrictions

example:



An undirected acyclic graph is called a **forest** (= set of trees). (union find data struc.)

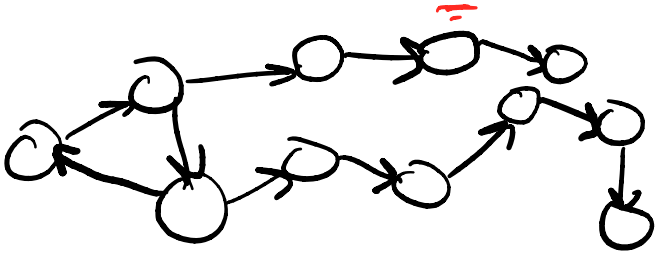
example:



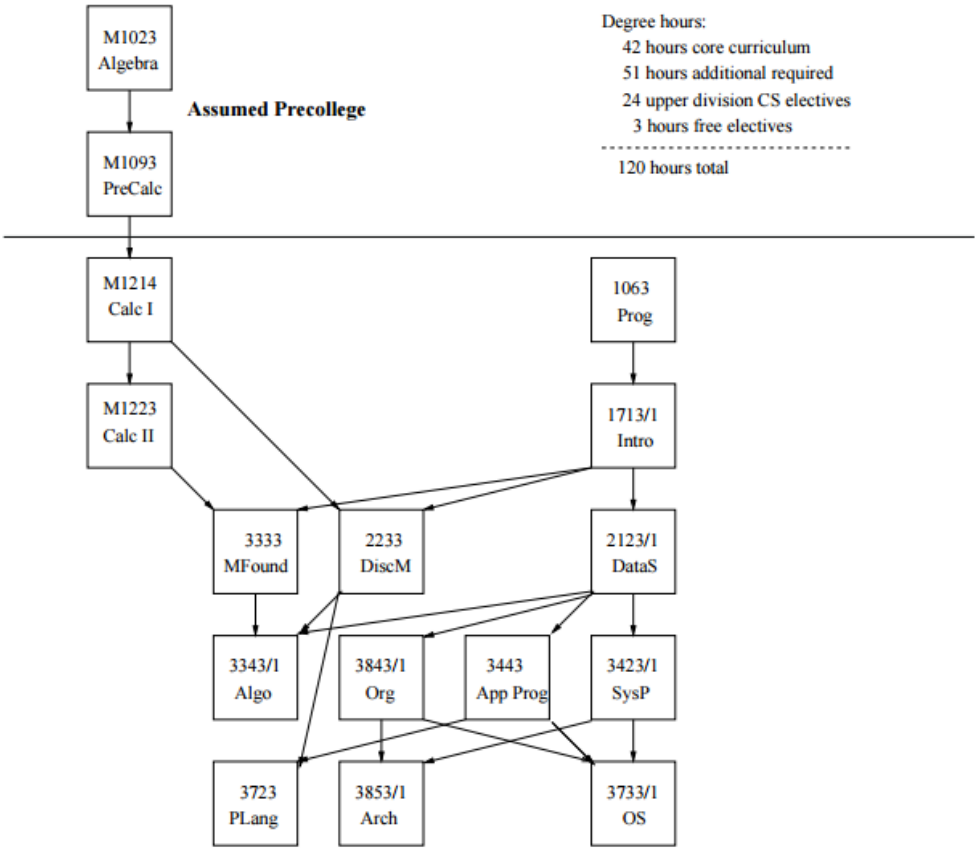
∴ UFDS
- utilize forests
- Application of Forests

"DAG"

An directed acyclic graph is called a **DAG**. // no cycles



DAGs are useful for representing the precedence of events. For example, the CS courses are arranged in a DAG.



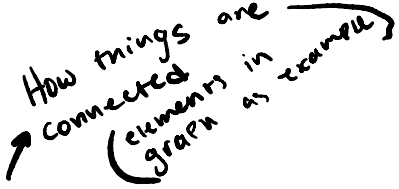
DAG Theorem

Theorem.
A directed graph G is acyclic $\Leftrightarrow$ a depth-first search of G yields no back edges. // "Back Edges" imply a cycle/loop in a graph

Proof sketch:

"$\Rightarrow$":
Suppose there is a back edge (u, v). Then by definition of a back edge there would be a cycle.

"$\Leftarrow$":
Suppose G contains a cycle c. Let v be the first vertex to be discovered in c, and let u be the preceding vertex v in c. v is an ancestor of u in the depth-first forest hence (u, v) is a back edge.



Topological Sort

If the nodes in our DAG represent events which we want to order we can use a topological sort. This gives us a sequence of the events which doesn't violate the dependencies.

We can use a DFS to find such an order. In particular, sort the vertices by their DFS finish time (from largest to smallest).

↳ creates a schedule