Discussion Week 8

Graph

Outline

- Objective
  - Understand the concepts of graph, which serve as the basic for later study

Outline

- Checklist of graph concepts
  - Undirected & directed graph
  - Edge & vertex, their relations
  - Weighted graph
  - Degree(undirected) & in/out degree (directed), source and sink
  - Sub-graph
  - Path
  - Connectedness
  - Graph & tree

Outline

- Later study
  - Minimum Spanning Tree:

    weighted graph + sub-graph + …
  - Topological Sorts

    weighted directed graph + DAG + source + …
  - Shortest Path

    weighted graph + path + …

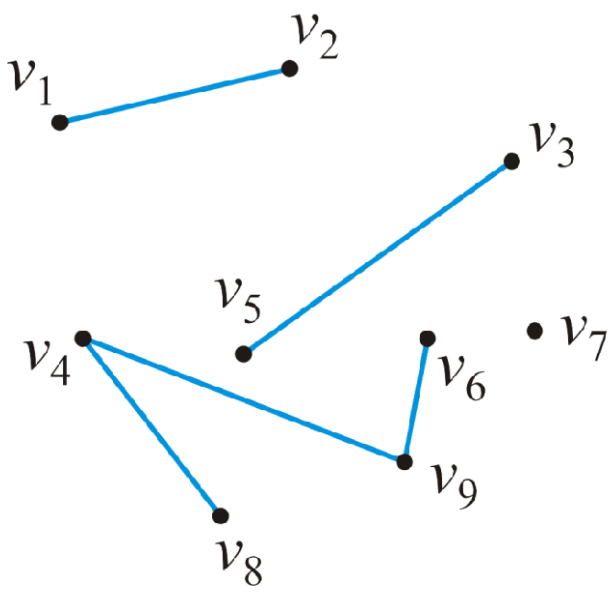
  … … …

Recap

- Undirected & directed graph

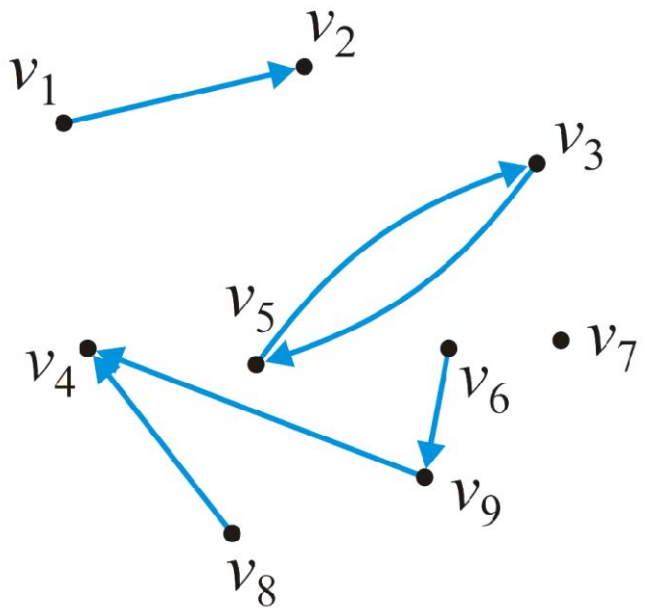
Unordered (v_j, v_k) represents edge connecting v_j and v_k

(v_j, v_k) = (v_k, v_j)



(v_j, v_k) represents connection from v_j to v_k

(v_j, v_k) is different from (v_k, v_j)



Recap

- Edge & vertex

In undirected graph, the maximum number of edges is

$$|E| \le \binom{|V|}{2} = \frac{|V|(|V|-1)}{2} = \mathrm{O}(|V|^2)$$

In directed graph, the maximum number of directed edges is

$$|E| \le 2\binom{|V|}{2} = 2\frac{|V|(|V|-1)}{2} = |V|(|V|-1) = \mathrm{O}(|V|^2)$$

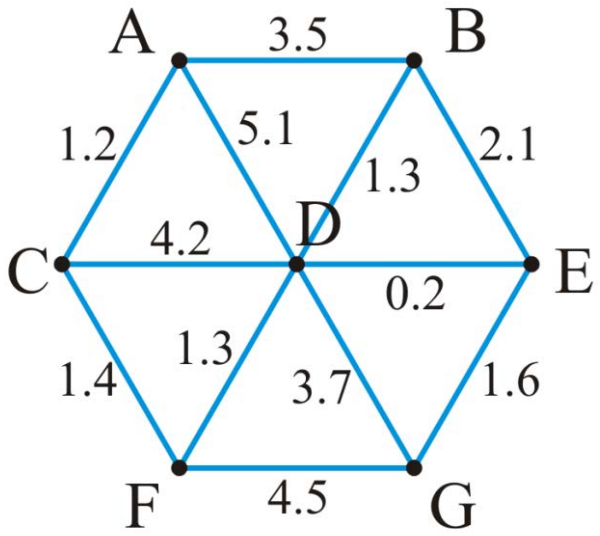
Recap

- Weighted graph

In weighted graph, each weight has a weight.

Each weight could represent distance, energy consumption, cost, etc.

An unweighted graph can be seen as a graph with uniform weight, e.g. 1.

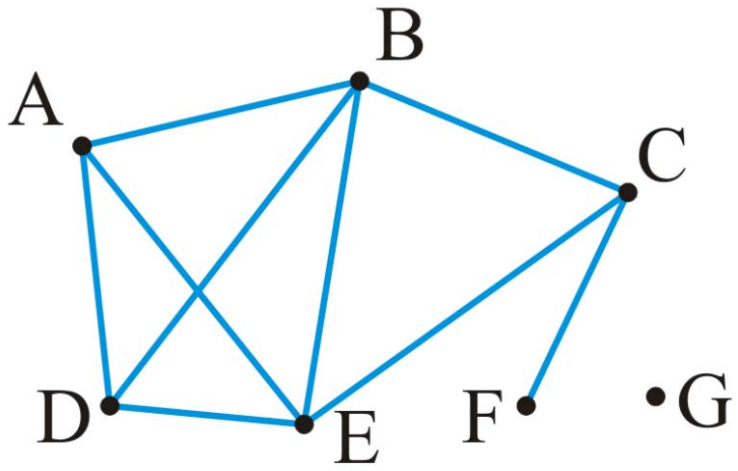


Recap

- Degree

degree of a vertex: number of adjacent vertices (neighbors)

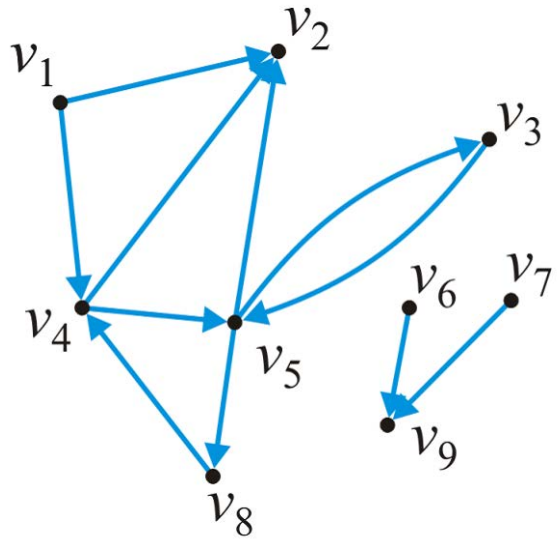
degree(A) = 3
degree(B) = 4
degree(C) = 3
… … …
degree(G) = 0



in-degree: the number of inward edges to the vertex

out-degree: the number of outward edges from the vertex

$in\_deg(v_1) = 0$
$out\_deg(v_1) = 2$
$in\_deg(v_2) = 3$
$out\_deg(v_2) = 0$
… … …

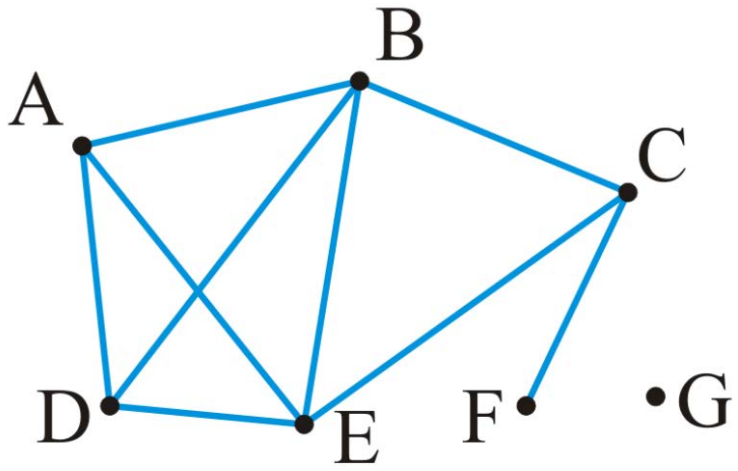


Recap

- Degree

degree of a vertex: number of adjacent vertices (neighbors)

degree(A) = 3
degree(B) = 4
degree(C) = 3
… … …
degree(G) = 0

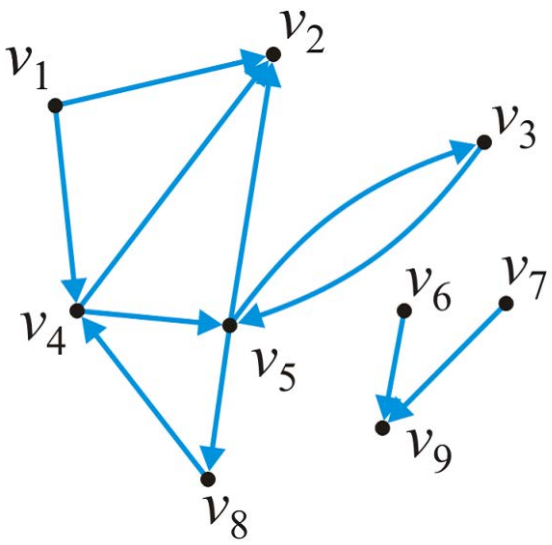


Sources: vertices with an in-degree of zero

Sinks: vertices with an out-degree of zero

Sources: v_1 , v_6 , v_7

Sinks: v_2 , v_9

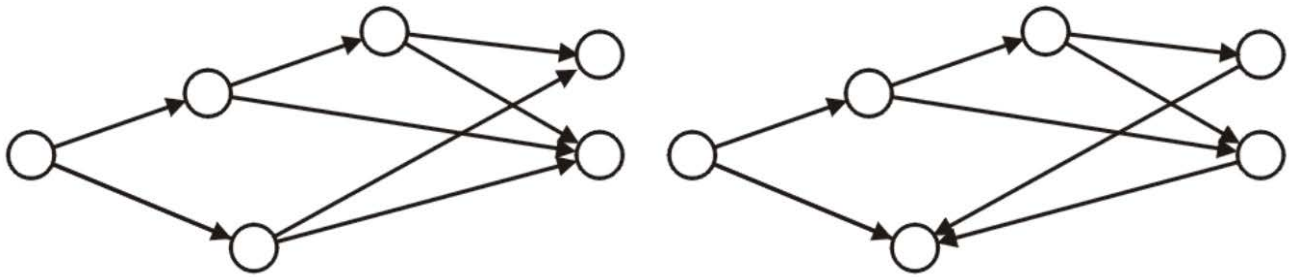


Recap

- DAG (Directed acyclic graphs)

DAG has no cycle.

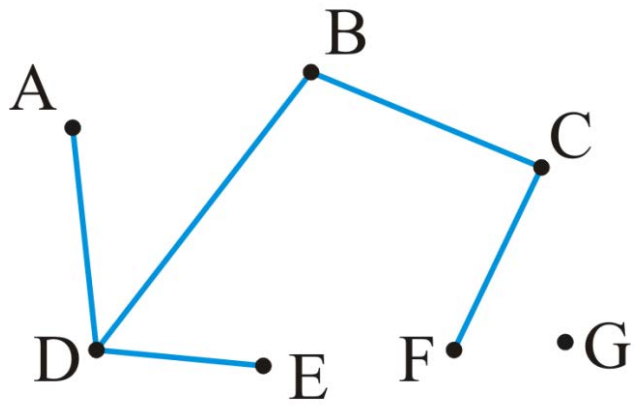
DAG has at least one source and at least one sink.

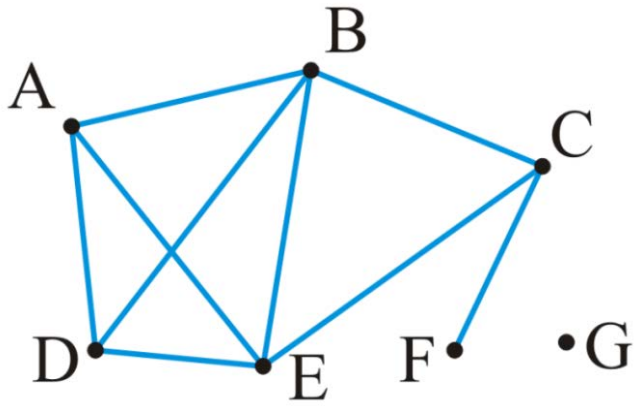


Recap

- Sub-graph

1. V': a subset of original vertices.
2. E': a subset of original edges that connect the subset of the vertices (V') in the original graph





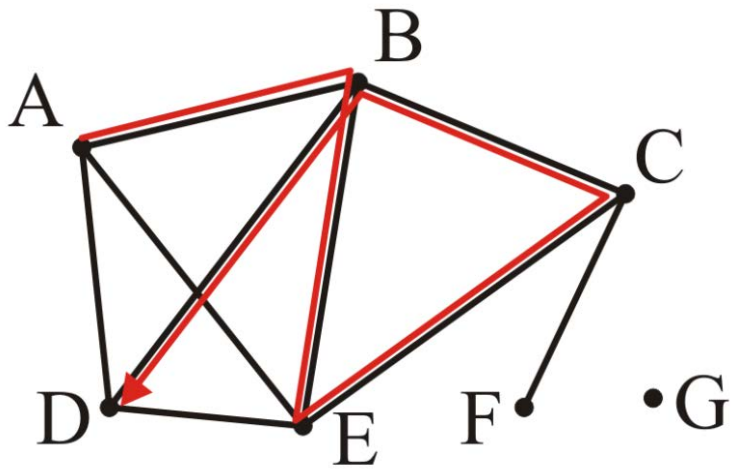
Recap

- Path

A path in an undirected graph is an ordered sequence of vertices (v_0, v_1, v_2, ..., v_k) where $\{v_{j-1}, v_j\}$ is an edge for j = 1, ..., k.

Represents a path from v_0 to v_k.

(A, B, C, E, B, D)

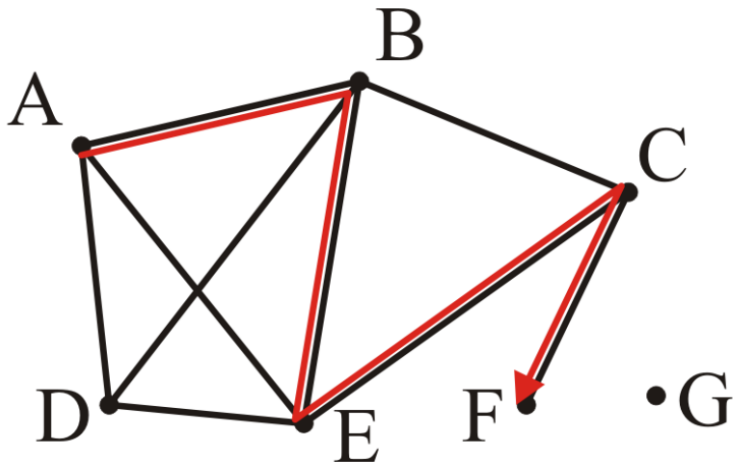


Recap

- Simple path

A simple path has no repetitions of vertices (other than perhaps the first and last vertices)

(A, B, E, C, F)



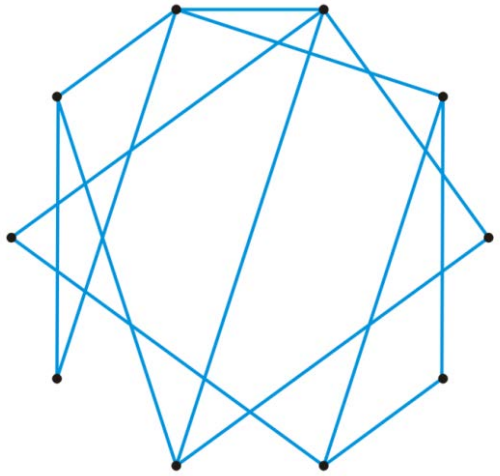
(A, B, E, C, F)

Recap

- Connectedness

Unweighted graph

Two vertices v_i , v_j are said to be connected if there exists a path from v_i to v_j

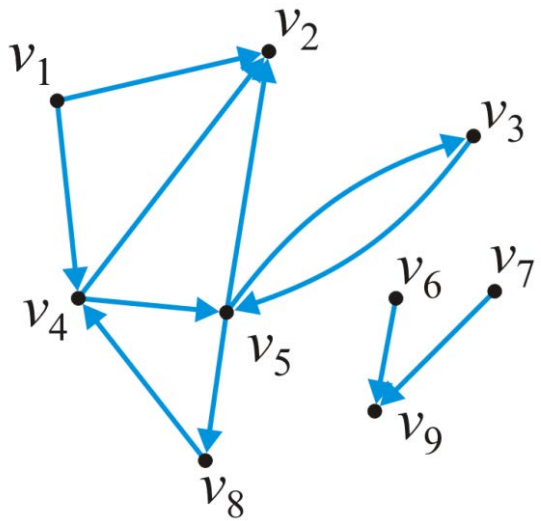


Weighted graph

- strongly connected: there exists a directed path between any two vertices
- weakly connected: there exists a path between any two vertices that ignores the direction

sub-graph $\{v_3, v_4, v_5, v_8\}$ is strongly connected

sub-graph $\{v_1, v_2, v_3, v_4, v_5, v_8\}$ is strongly connected



Recap

- Graph & tree

A graph is a tree if it is connected and there is a unique path between any two vertices.

i.e. No cycle in the graph

-> convert to a tree:

1. Choosing any vertex to be the root
2. Defining its neighboring vertices as its children

and then recursively defining:

3. All neighboring vertices other than that one designated its parent to be its children

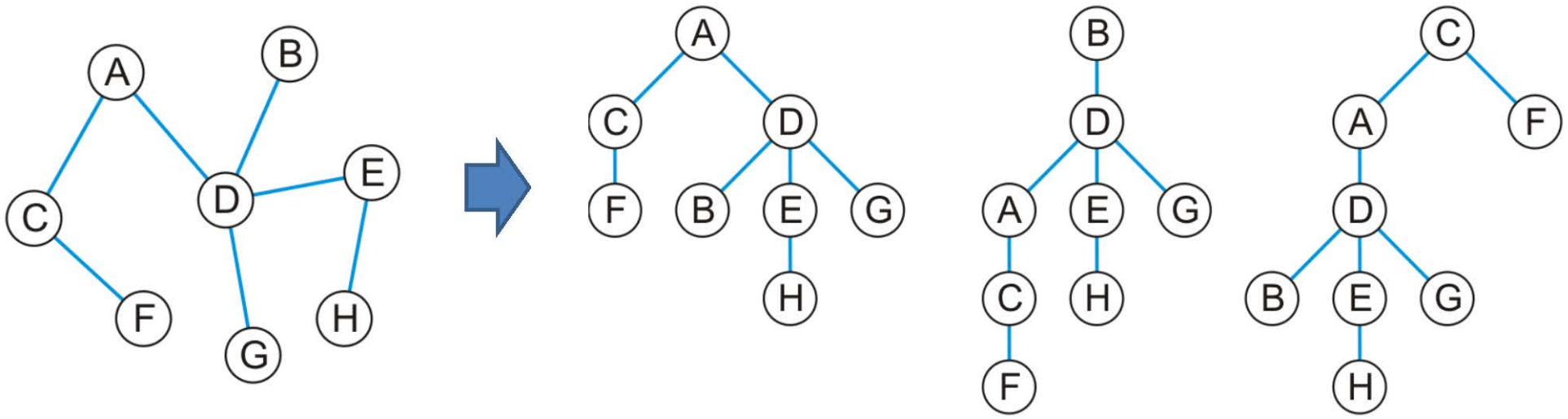
Recap

- Graph & tree

1. Choosing any vertex to be the root
2. Defining its neighboring vertices as its children

and then recursively defining:

3. All neighboring vertices other than that one designated its parent to be its children



Graph Representation

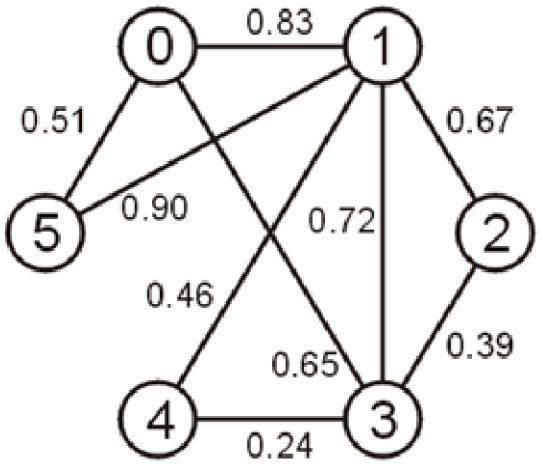
Adjacency Matrix

Undirected graph:

If the vertices v_i and v_j are connected with weight w, then set a_{ij} = w and a_{ji} = w

The matrix is symmetric

	0	1	2	3	4	5
0		0.83		0.65		0.51
1	0.83		0.67	0.72	0.46	0.90
2		0.67		0.39		
3	0.65	0.72	0.39		0.24	
4		0.46		0.24		
5	0.51	0.90				



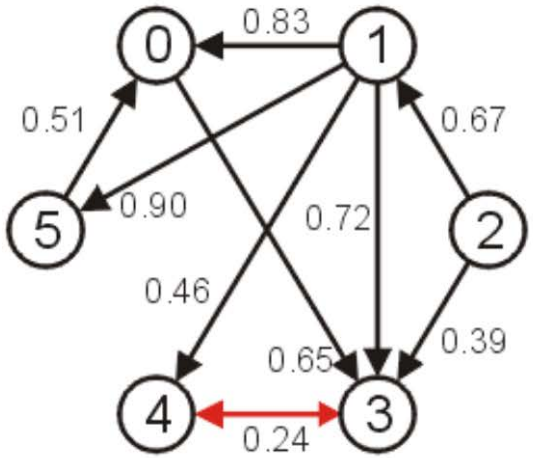
Adjacency Matrix

directed graph:

If the edge (v_i, v_j) has weight w, then set $a_{ij} = w$

If the graph was directed, then the matrix would not necessarily be symmetric

	0	1	2	3	4	5
0				0.65		
1	0.83			0.72	0.46	0.90
2		0.67		0.39		
3					0.24	
4				0.24		
5	0.51					



Adjacency Matrix

Space:

A graph of n vertices may have up to

$$\binom{n}{2} = \frac{n(n-1)}{2} = \mathbf{O}(n^2)$$

edges, which is terrible when n is large and the matrix is sparse

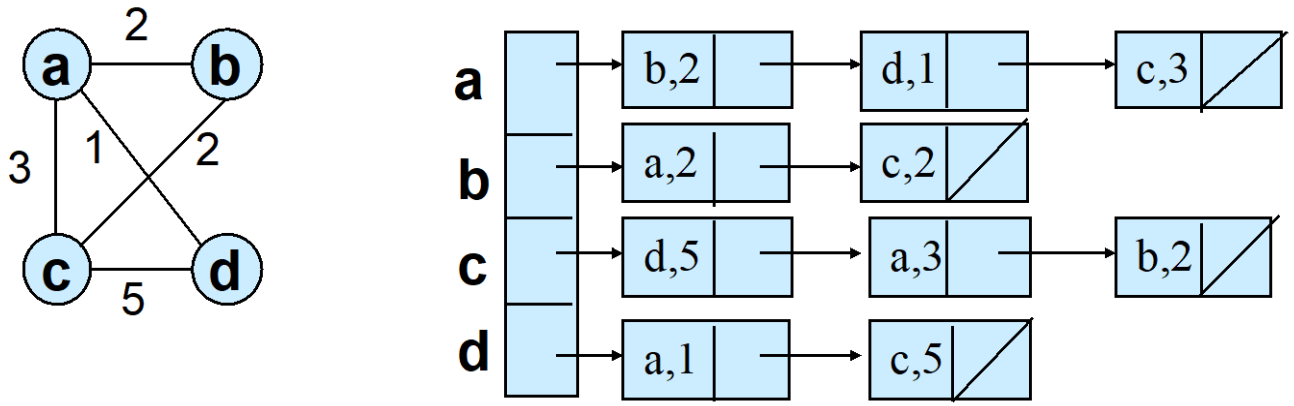
Look up time:

O(1) once you know index i, j

Adjacency List

Undirected graph: use an array of linked lists to store edges

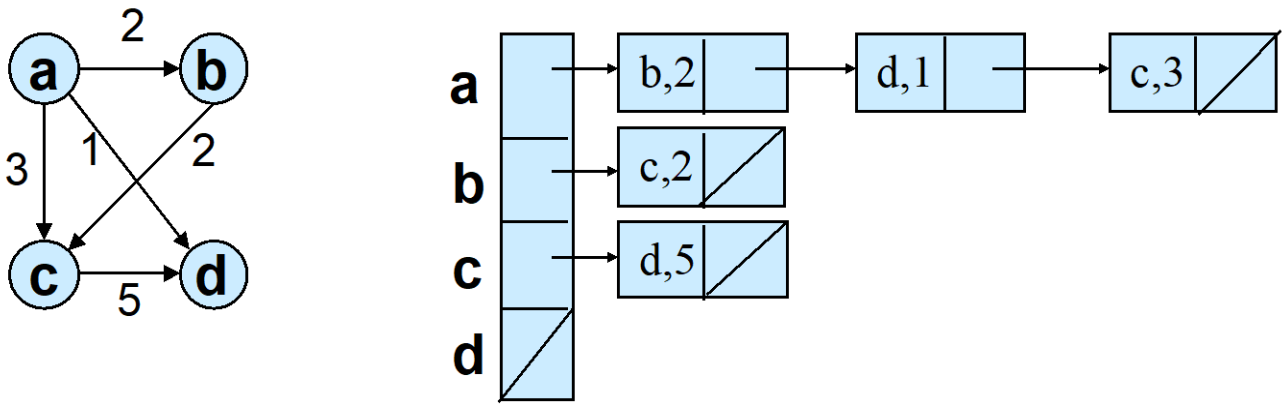
- Each vertex has a linked list that stores all the edges connected to the vertex.
- Each node in a linked list must store two items of information: the connecting vertex and the weight.



Adjacency List

Undirected graph: use an array of linked lists to store edges

- Each vertex has a linked list that stores all the edges originated from the vertex
- Each node in a linked list stores two items of information: the vertex that the edge connects to, the weight



Adjacency List

Space:

$O(|E|) = O(V^2)$, which is good when $|E| << |V|^2$

Look up time:

For a certain vertex,

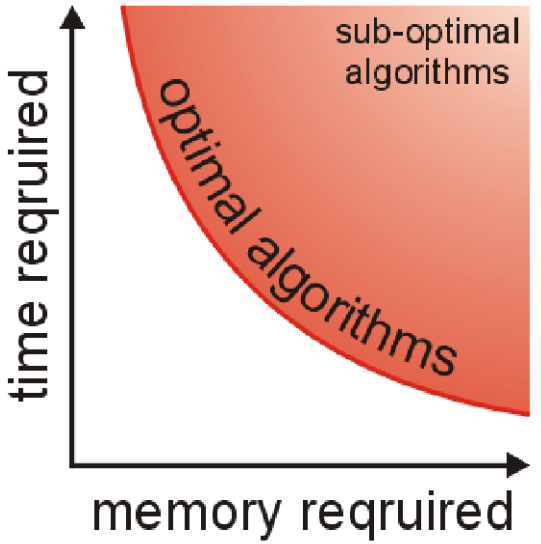
$O(|\{ \{v_i, v_j\} \mid \{v_i, v_j\} \text{ is in } E \}|)$, which is terrible when the vertex has large out-degree

Graph Representation

It’ s a trade-off between time and space.

For a data structure/algorithm:

- Improving the run time usually requires more memory
- Reducing the required memory usually requires more run time



Graph Traversal

Outline

- Objective
  - Understand the algorithm of BFS, DFS in graph.
  - Understand their classical applications:

    Determining Connections

    Determining Distances (in an unweighted graph)

    Bipartite Graphs

Outline

- Graph traversal
  - The only difference with tree traversal using BFS and DFS:
    - To avoid visiting a vertex for multiple times, we have to track which vertices have already been visited.
    - We may have an indicator variable in each vertex or with other approaches.

BFS

- Basics
  - Queue
  - Size of queue: $O(|V|)$
  - Time complexity: $\Theta(|V| + |E|)$

BFS

- Algorithm:
  - – Choose any vertex, mark it as visited and push it onto queue
    - – While the queue is not empty:
      - Pop the top vertex v from the queue
      - For each vertex adjacent to v that has not been visited:
        - – Mark it visited, and
        - – Push it onto the queue

DFS

- Basics
  - Stack or recursive
  - Size of stack: O(|V|)
  - Time complexity: Θ(|V| + |E|)

DFS

- Algorithm:
  - – Choose any vertex, mark it as visited and push it onto queue
    - – From that vertex:
      - If there is another adjacent vertex not yet visited, go to it
      - Otherwise, go back to the previous vertex
    - – Continue until no visited vertices have unvisited adjacent vertices

Connected

First, let us determine whether one vertex is connected to another

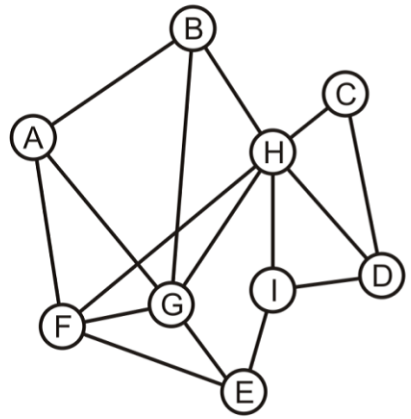
- v_j is connected to v_k if there is a path from the first to the second

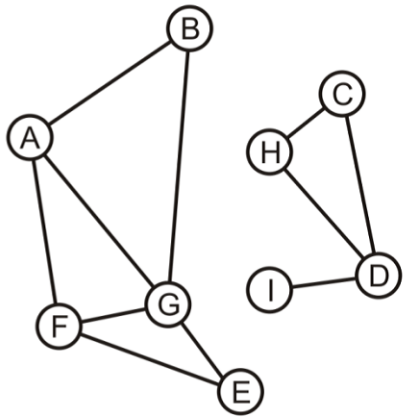
Strategy:

- Perform a breadth-first traversal starting at v_j
- If the vertex v_k is ever found during the traversal, return true
- Otherwise, return false

Determining Connections

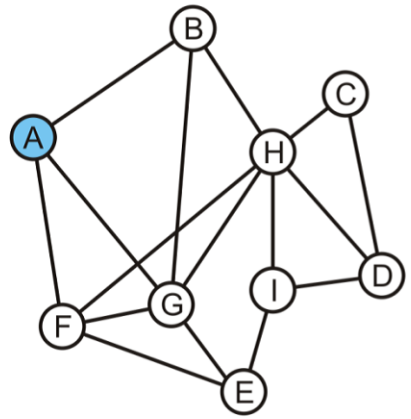
Is A connected to D?

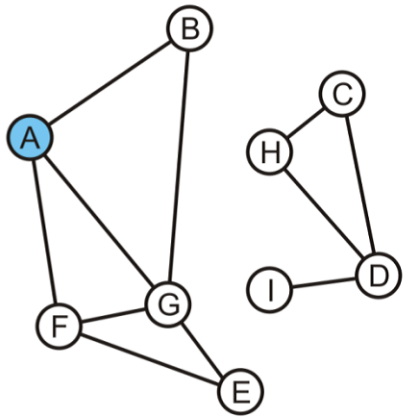




Determining Connections

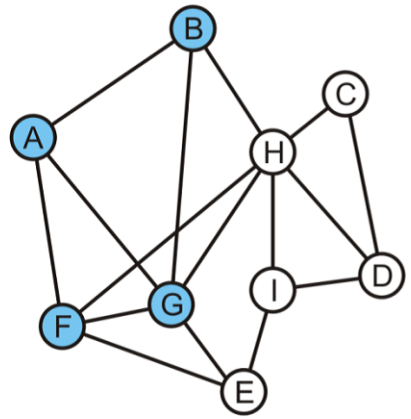
Vertex A is marked as visited and pushed onto the queue



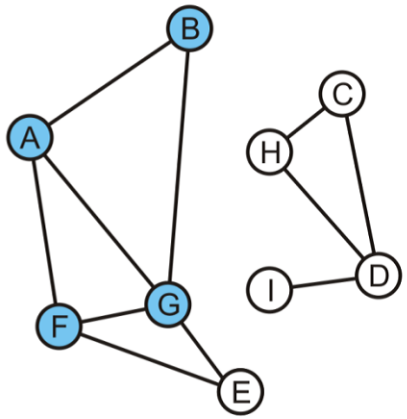


Determining Connections

Pop the head, A, and mark and push B, F and G



B	F	G			

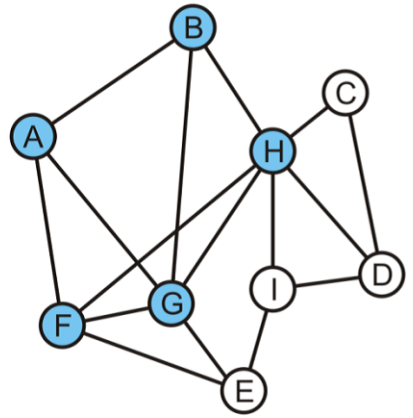


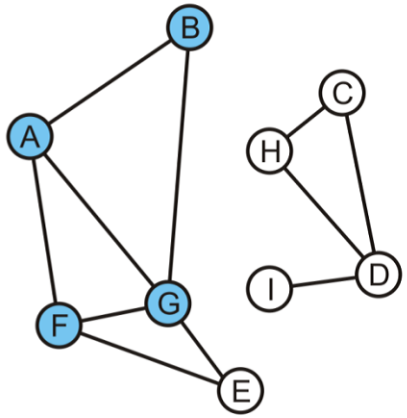
B	F	G			

Determining Connections

Pop B and mark and, in the left graph, mark and push H

- On the right graph, B has no unvisited adjacent vertices



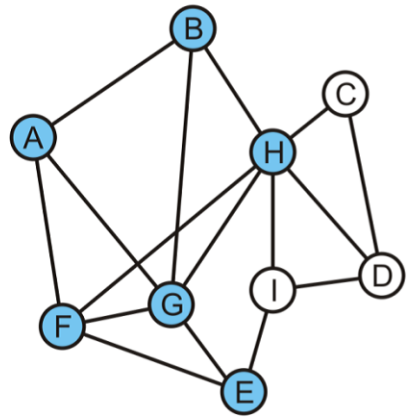


F	G	H			

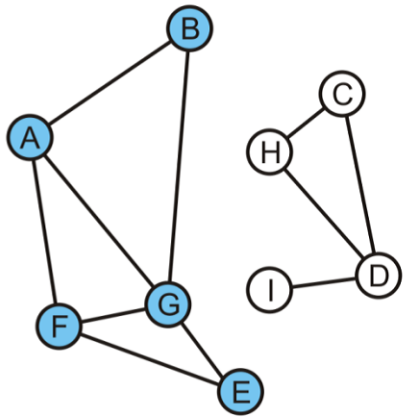
F	G				

Determining Connections

Popping F results in the pushing of E



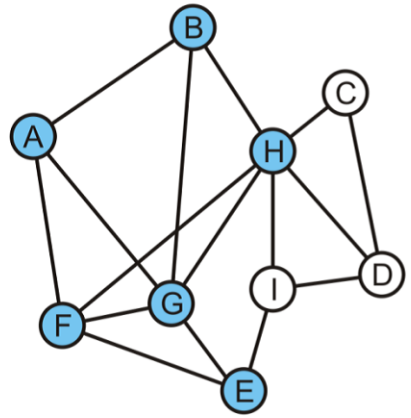
G	H	E			



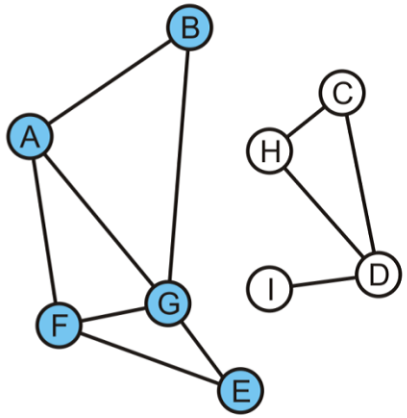
G	E				

Determining Connections

In either graph, G has no adjacent vertices that are unvisited



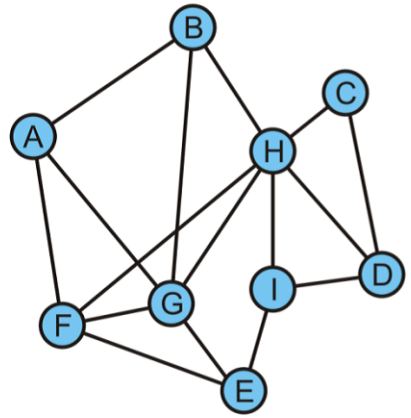
H	E				



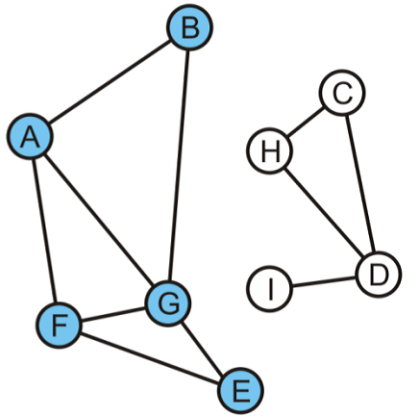
E					

Determining Connections

Popping H on the left graph results in C, I, D being pushed



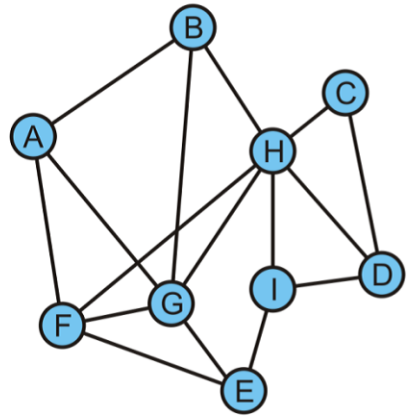
E	C	I	D		

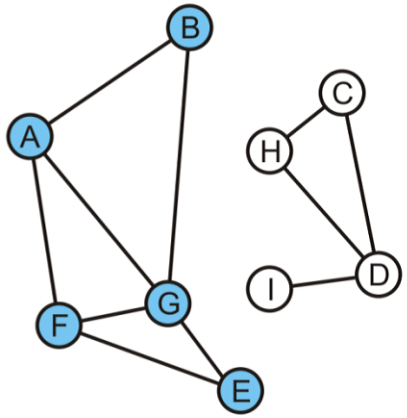


Determining Connections

On the left, D is now visited

- We determine A is connected to D



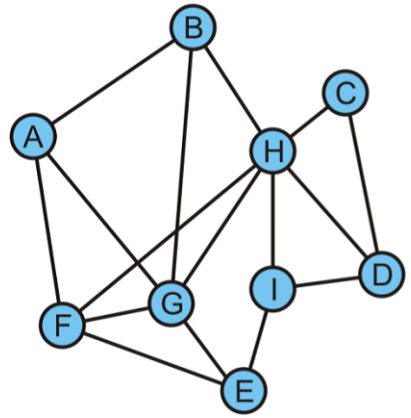


E	C	I	D		

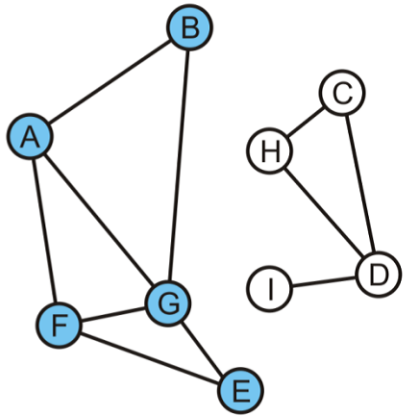
Determining Connections

On the right, the queue is empty and D is not visited

- We determine A is not connected to D



C	I	D			



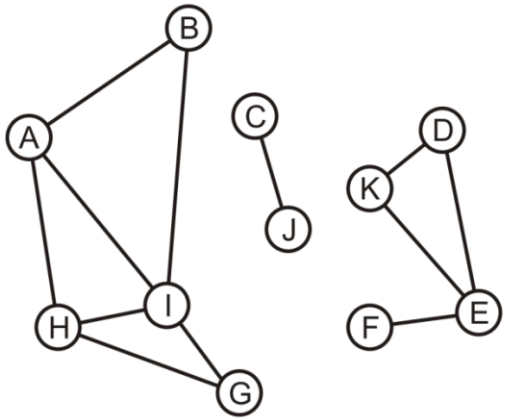
Connected Components

- Suppose we want to partition the vertices into connected sub-graphs
  - While there are unvisited vertices in the tree:
    - Select an unvisited vertex and perform a traversal on that vertex
    - Each vertex that is visited in that traversal is added to the set initially containing the initial unvisited vertex
  - Continue until all vertices are visited

- We would use a disjoint set data structure for maximum efficiency

Connected Components

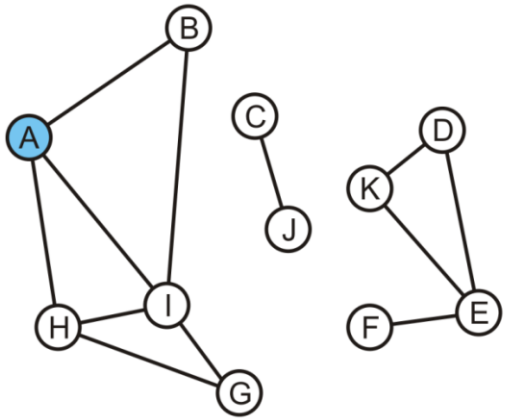
Here we start with a set of singletons



A	B	C	D	E	F	G	H	I	J	K
A	**B**	**C**	**D**	**E**	**F**	**G**	**H**	**I**	**J**	**K**

Connected Components

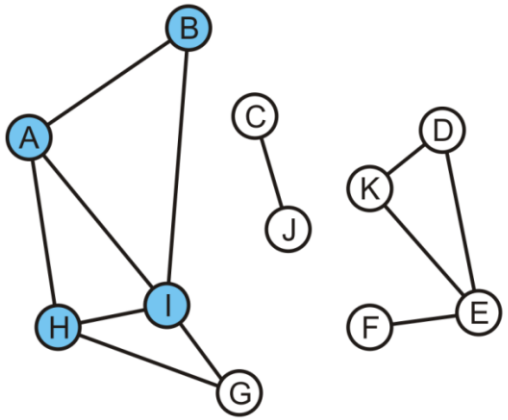
The vertex A is unvisited, so we start with it



A	B	C	D	E	F	G	H	I	J	K
A	**B**	**C**	**D**	**E**	**F**	**G**	**H**	**I**	**J**	**K**

Connected Components

Take the union of with its adjacent vertices: {A, B, H, I}

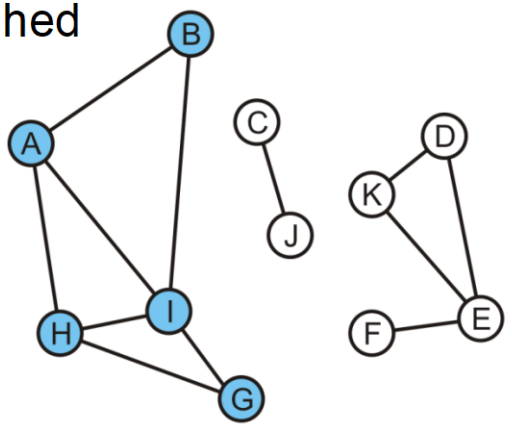


A	B	C	D	E	F	G	H	I	J	K
A	**A**	**C**	**D**	**E**	**F**	**G**	**A**	**A**	**J**	**K**

Connected Components

As the traversal continues, we take the union of the set {G} with the set containing H: {A, B, G, H, I}

– The traversal is finished

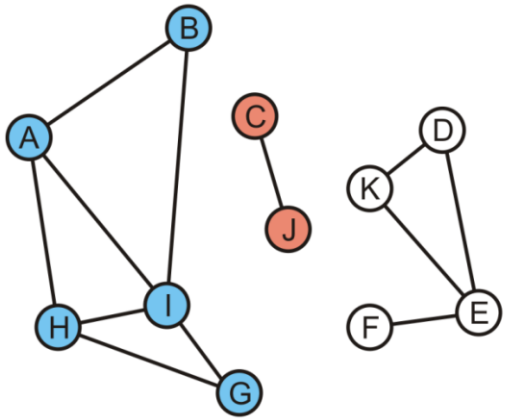


A	B	C	D	E	F	G	H	I	J	K
A	**A**	**C**	**D**	**E**	**F**	**A**	**A**	**A**	**J**	**K**

Connected Components

We take the union of {C} and its adjacent vertex J: {C, J}

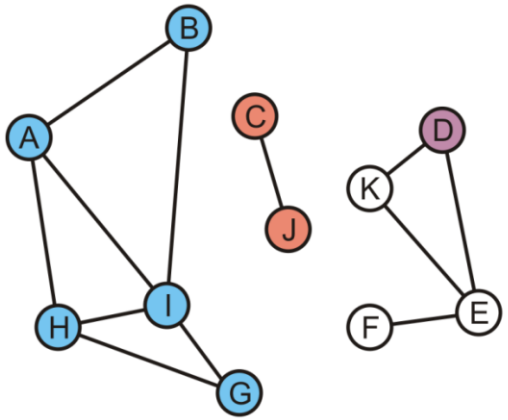
– This traversal is finished



A	B	C	D	E	F	G	H	I	J	K
A	A	C	D	E	F	A	A	A	C	K

Connected Components

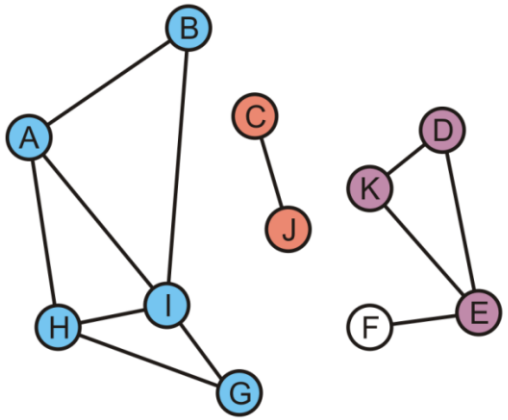
We start again with the set {D}



A	B	C	D	E	F	G	H	I	J	K
A	A	C	D	E	F	A	A	A	C	K

Connected Components

K and E are adjacent to D, so take the unions creating {D, E, K}

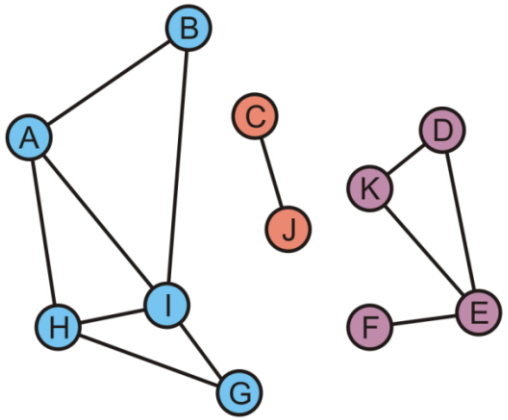


A	B	C	D	E	F	G	H	I	J	K
A	A	C	D	D	F	A	A	A	C	D

Connected Components

Finally, during this last traversal we find that F is adjacent to E

- Take the union of {F} with the set containing E: {D, E, F, K}

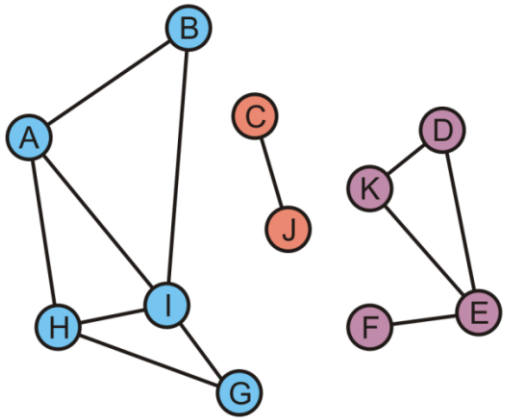


A	B	C	D	E	F	G	H	I	J	K
A	A	C	D	D	D	A	A	A	C	D

Connected Components

All vertices are visited, so we are done

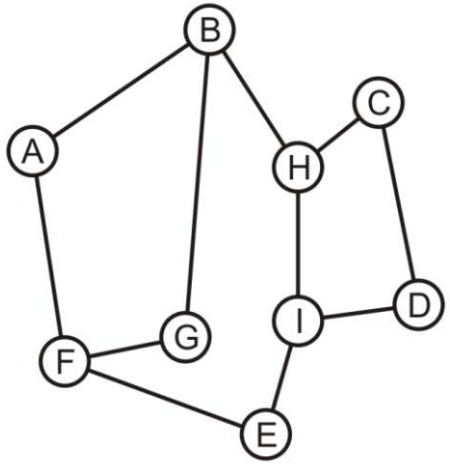
- There are three connected sub-graphs {A, B, G, H, I}, {C, J}, {D, E, F, K}

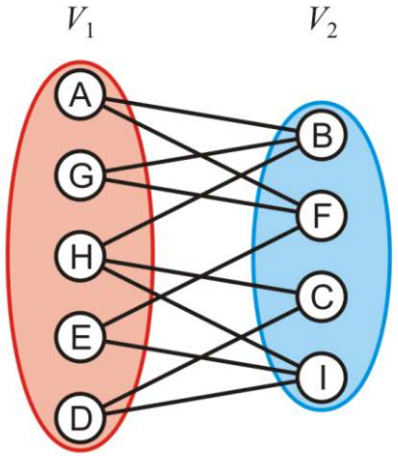


A	B	C	D	E	F	G	H	I	J	K
A	A	C	D	D	D	A	A	A	C	D

Bipartite graph

- Definition
  - A bipartite graph is a graph where the vertices V can be divided into two disjoint sets V_1 and V_2 such that every edge has one vertex in V_1 and the other in V_2





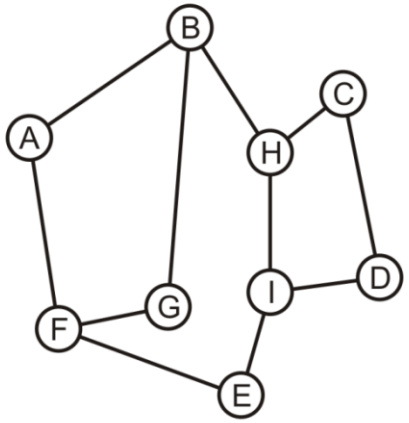
Bipartite Graphs

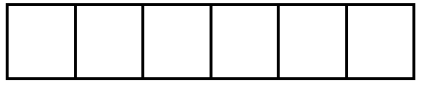
- Use a breadth-first traversal for a connected graph:
  - Choose a vertex, mark it belonging to V_1 and push it onto a queue
  - While the queue is not empty, pop the front vertex v and
    - Any adjacent vertices that are already marked must belong to the set not containing v, otherwise, the graph is not bipartite (we are done);
    - Any unmarked adjacent vertices are marked as belonging to the other set and they are pushed onto the queue
  - If the queue is empty, the graph is bipartite

Bipartite Graphs

With the first graph, we can start with any vertex

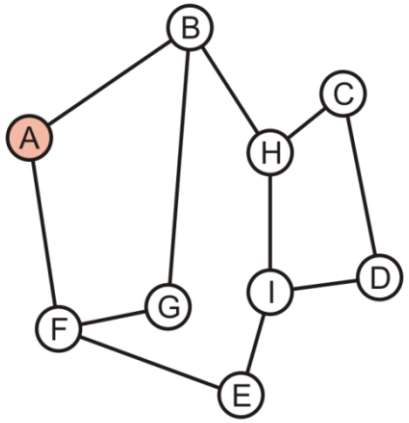
- We will use colours to distinguish the two sets

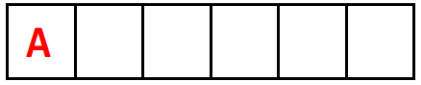




Bipartite Graphs

Push A onto the queue and colour it red

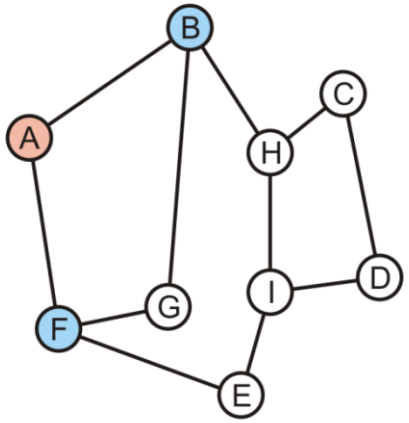


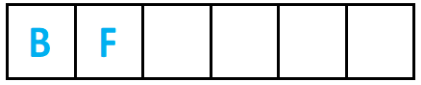


Bipartite Graphs

Pop A and its two neighbours are not marked:

- Mark them as blue and push them onto the queue

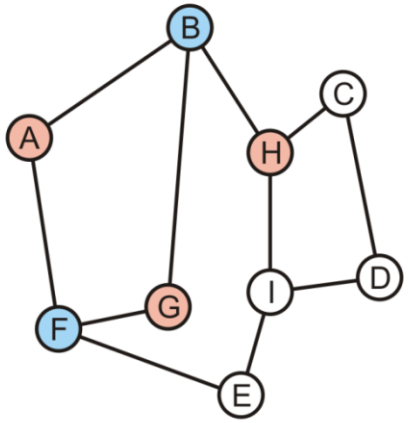




Bipartite Graphs

Pop B—it is blue:

- Its one marked neighbour, A, is red
- Its other neighbours G and H are not marked: mark them red and push them onto the queue

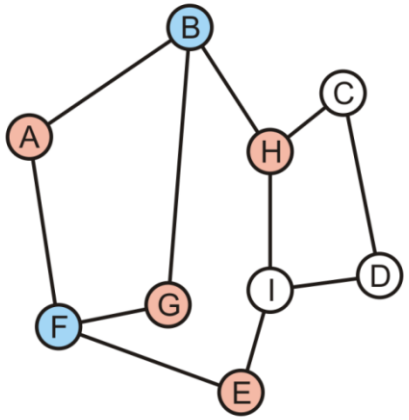


F	G	H			

Bipartite Graphs

Pop F—it is blue:

- Its two marked neighbours, A and G, are red
- Its neighbour E is not marked: mark it red and pus it onto the queue

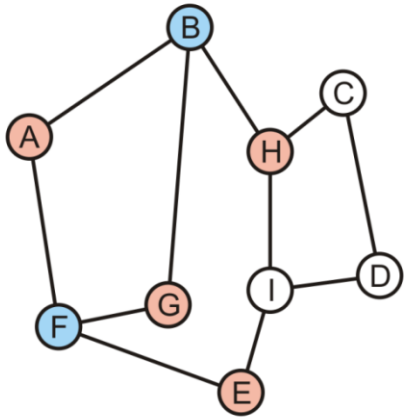


G	H	E			

Bipartite Graphs

Pop G—it is red:

– Its two marked neighbours, B and F, are blue

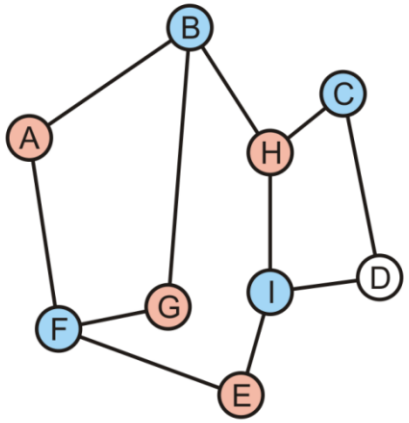


H	E				

Bipartite Graphs

Pop H—it is red:

- Its marked neighbours, B, is blue
- It has two unmarked neighbours, C and I; mark them blue and push them onto the queue

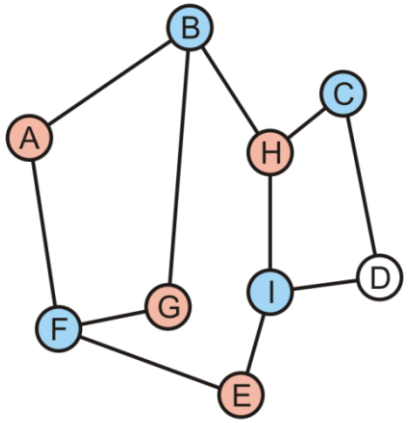


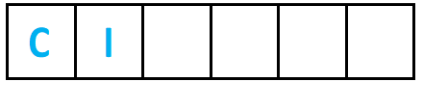
E	C	I			

Bipartite Graphs

Pop E—it is red:

- Its marked neighbours, F and I, are blue

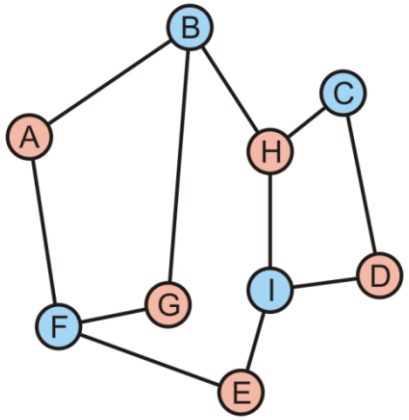




Bipartite Graphs

Pop C—it is blue:

- Its marked neighbour, H, is red
- Mark D as red and push it onto the queue

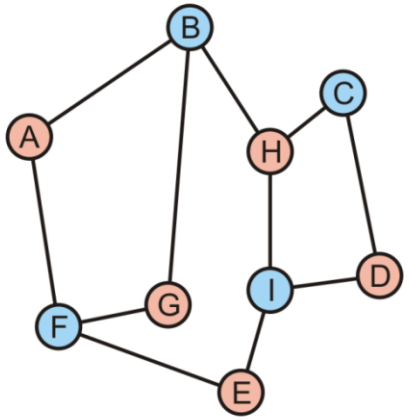


I	D				

Bipartite Graphs

Pop I—it is blue:

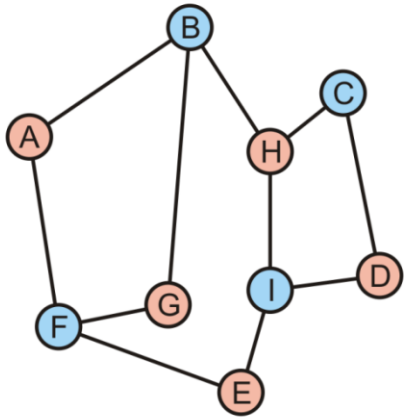
- Its marked neighbours, H, D and E, are all red



Bipartite Graphs

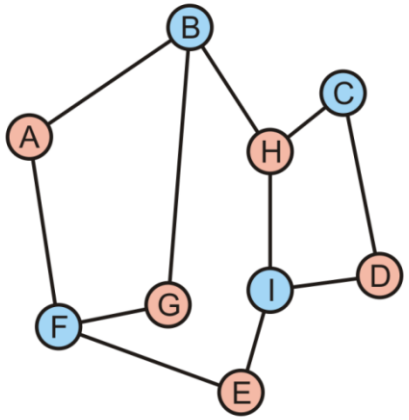
Pop D—it is red:

- Its marked neighbours, C and I, are both blue



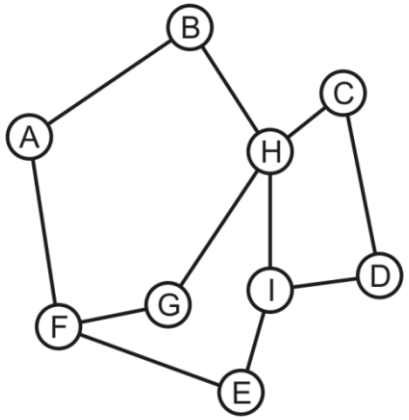
Bipartite Graphs

The queue is empty, the graph is bipartite



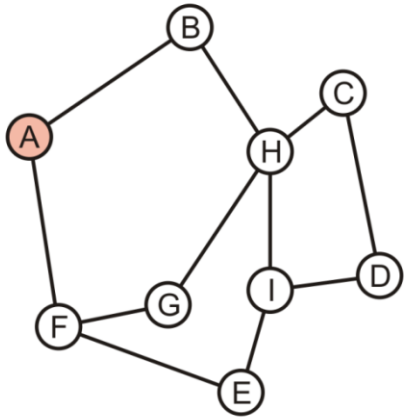
Bipartite Graphs

Consider the other graph which was claimed to be not bipartite



Bipartite Graphs

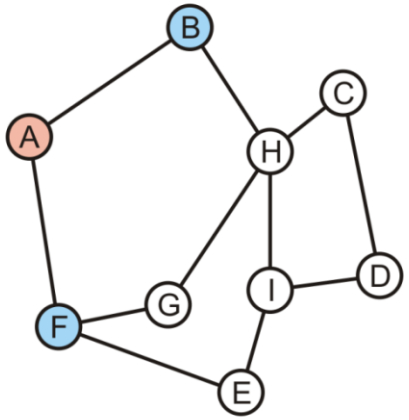
Push A onto the queue and colour it red



Bipartite Graphs

Pop A off the queue:

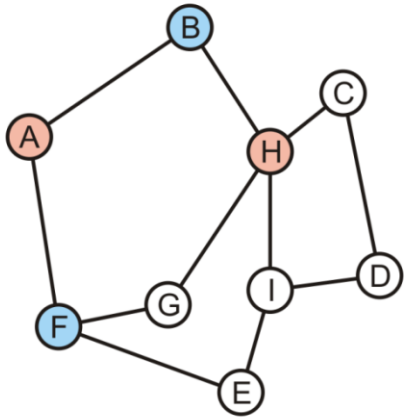
– Its neighbours are unmarked: colour them blue and push them onto the queue



Bipartite Graphs

Pop B off the queue:

- Its one neighbour, A, is red
- The other neighbour, H, is unmarked: colour it red and push it onto the queue

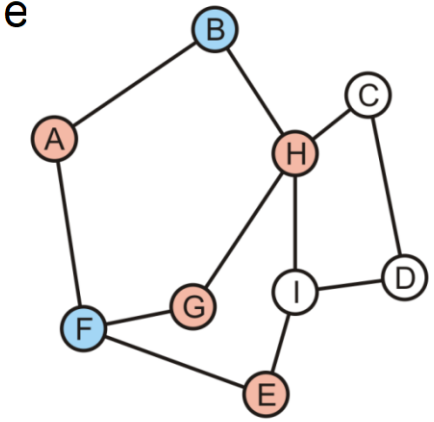


F	H				

Bipartite Graphs

Pop F off the queue:

- Its one neighbour, A, is red
- The other neighbours, E and G, are unmarked: colour them red and push it onto the queue

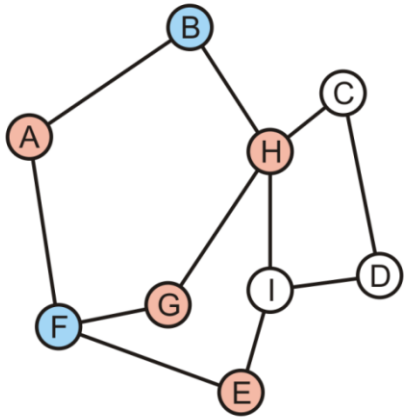


H	E	G			

Bipartite Graphs

Pop H off the queue—it is red:

- Its one neighbour, G, is already red
- The graph is not bipartite



E	G				

Bipartite Graphs

- Definition
  - Cycles that contains either an even number or an odd number of vertices are said to be even cycles and odd cycles, respectively

- Theorem
  - A graph is bipartite if and only if it does not contain any odd cycles

Disjoint Set

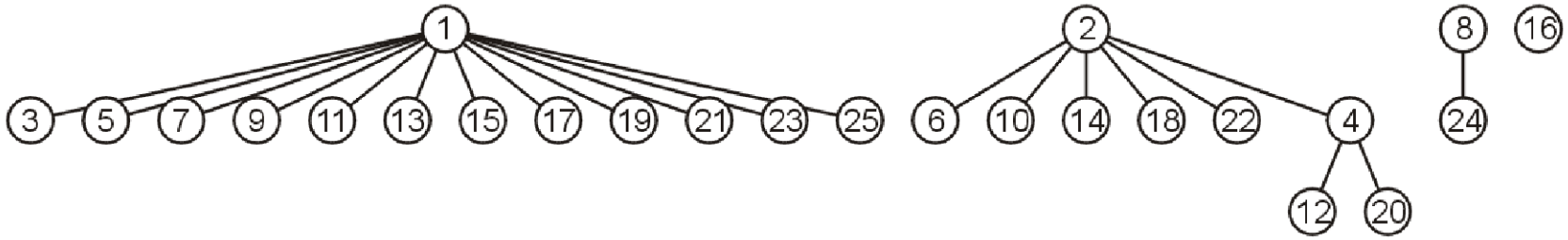
Outline

- Objective
  - Definition of a Disjoint Set
  - The basic operations: find(), union()
  - Height
    - Worst: O(log n)
    - Best: O(1)
  - Optimizations:
    - Union by size
    - Path compression in find()

Another way to represent Root

But it is not a tree!

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
1	2	1	2	1	2	1	8	1	2	1	4	1	2	1	16	1	2	1	4	1	2	1	8	1



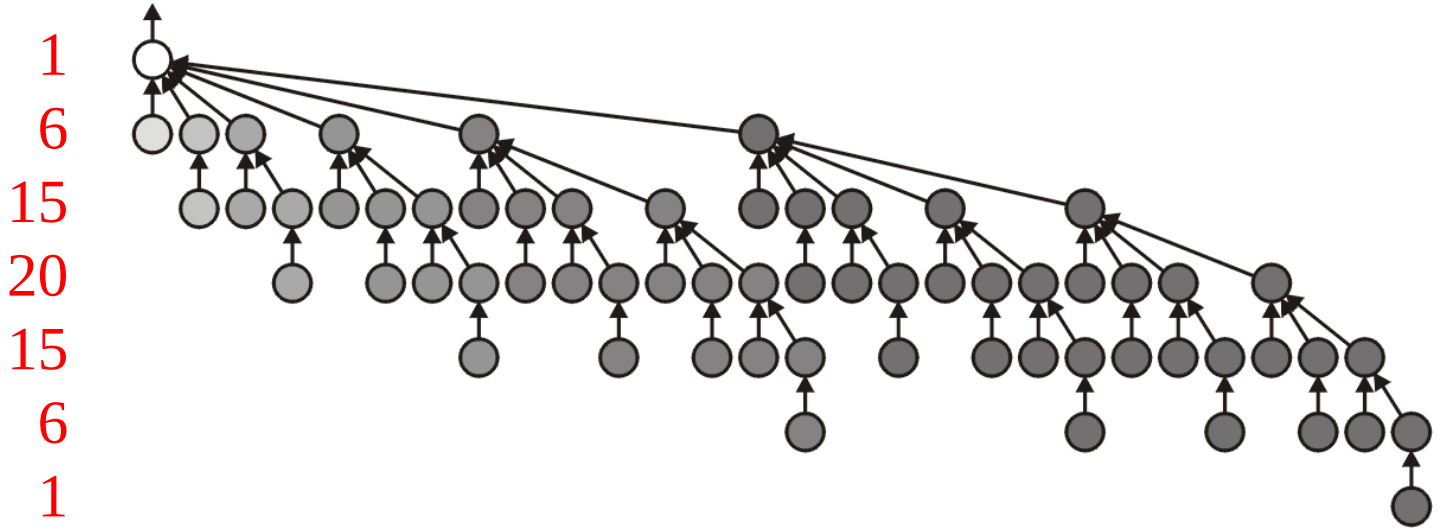
Worst-Case Scenario

- Let us consider creating the worst-case disjoint set
  - The tallest tree with the least number of nodes

- The worst case tree of height h must result from taking union of two worst case trees of height h–1

Worst-Case Scenario

These are *binomial trees*



Worst-Case Scenario

From the construction, it should be clear that this would define Pascal's triangle

- The *binomial* coefficients

$$\binom{n}{m} = \begin{cases} 1 & m = 0 \text{ or } m = n \\ \binom{n-1}{m} + \binom{n-1}{m-1} & 0 < m < n \end{cases}$$

$$= \frac{n!}{m!(n-m)!}$$

1

1 1

1 2 1

1 3 3 1

1 4 6 4 1

1 5 10 10 5 1

1 6 15 20 15 6 1

Worst-Case Scenario

- Thus, suppose we have a worst-case tree of height h
  - The number of nodes is $\sum_{k=0}^{h} \binom{h}{k} = 2^h = n$
  - The sum of node depth is $\sum_{k=0}^{h} k\binom{h}{k} = h2^{h-1}$
  - Therefore, the average depth is $\frac{h2^{h-1}}{2^h} = \frac{h}{2} = \frac{\lg(n)}{2}$
  - The height and average depth of the worst case are O(ln(n))

Optimization 1

- Problem:
  - The height of the tree may grow very large

- To optimize both find and set_union, we must minimize the height of the tree
  - Therefore, point the root of the shorter tree to the root of the taller tree
  - The height of the taller will increase if and only if the trees are equal in height

Union By Size

- Maintain a subtree count for each node, initially 1.
- Link root of smaller tree to root of larger tree (breaking ties arbitrarily).

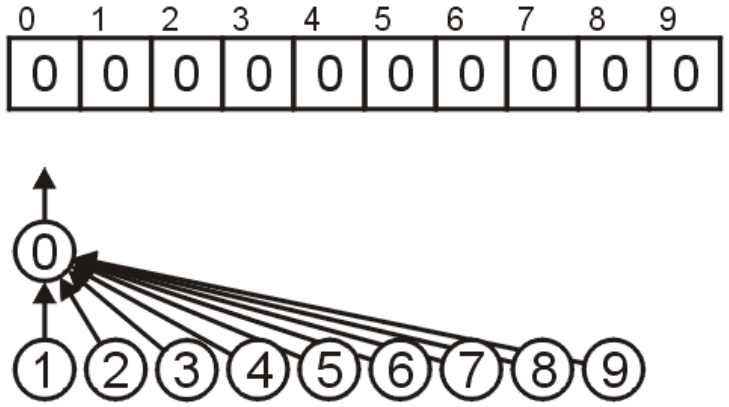
```
MAKE-SET (x)
parent(x) ← x.
size(x) ← 1.
```

```
FIND (x)
WHILE (x ≠ parent(x))
    x ← parent(x).
RETURN x.
```

```
UNION-BY-SIZE (x, y)
r ← FIND (x).
s ← FIND (y).
IF (r = s) RETURN.
ELSE IF (size(r) > size(s))
    parent(s) ← r.
    size(r) ← size(r) + size(s).
ELSE
    parent(r) ← s.
    size(s) ← size(r) + size(s).
```

Best-Case Scenario

- In the best case, all elements point to the same entry with a resulting height of O(1):



Optimization 2: Path Compression

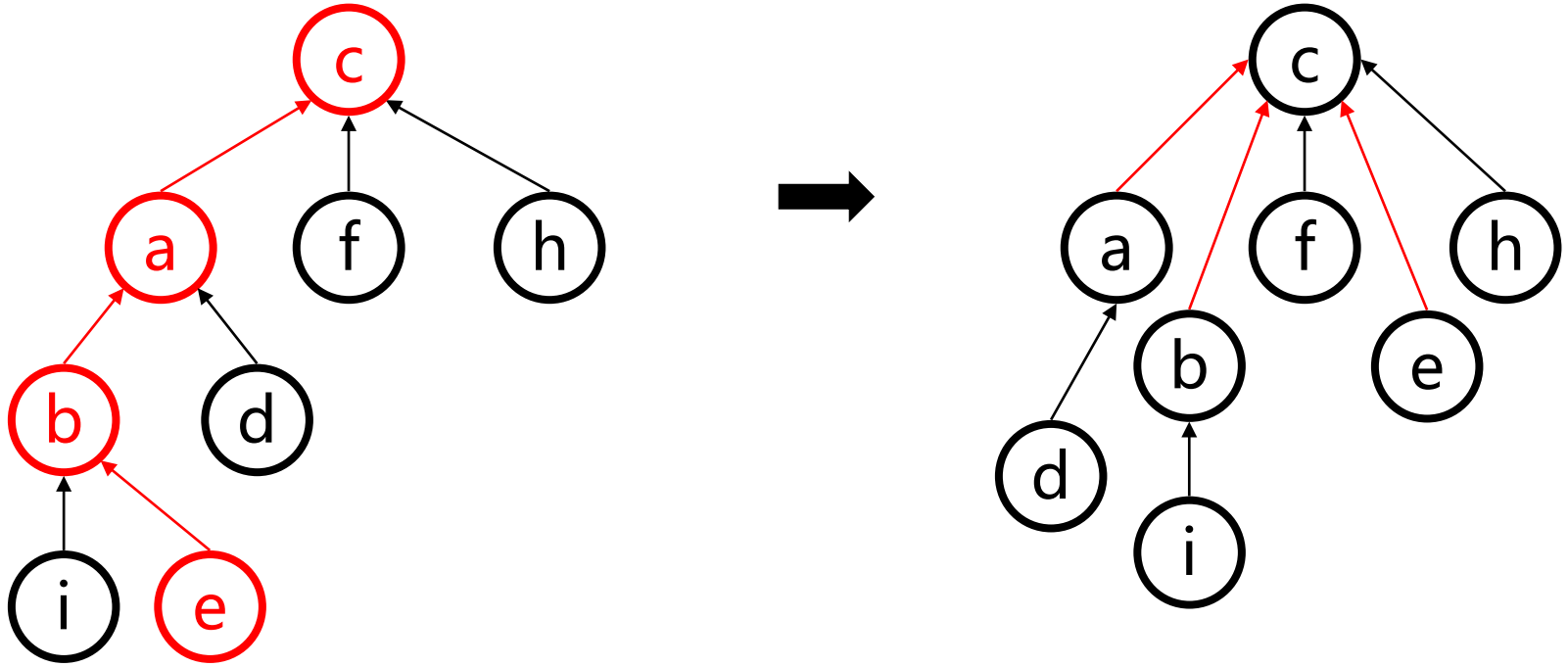
- Another optimization is that, whenever find is called, update the object to point to the root

```
size_t Disjoint_set::find( size_t n ) {
    if ( parent[n] == n ) {
        return n;
    } else {
        parent[n] = find( parent[n] );
        return parent[n];
    }
}
```

- The next call to find(n) is Q(1)
- The cost is O(h) memory

Optimization 2: Path Compression

- find(e)



Time complexity

- With both optimization methods, could it be any better than O(log(n))?

Time complexity

- With both optimization methods, could it be any better than O(log(n))?
- Result in Θ(1) time complexity