CS101 Algorithms and Data Structures

Graph Traversal

Textbook Ch 22.2/3/5

Outline

- Graph traversal
  - Breadth-first
  - Depth-first
- Applications
  - Connectedness
  - Unweighted path length
  - Identifying bipartite graphs

Graph Traversal

Traversals of a graph

- A means of visiting all the vertices in a graph
- Also called *searches*

Similar to tree traversal, we have breadth-first and depth-first traversals on graphs

- Breadth-first requires a queue
- Depth-first requires a stack

Graph Traversal

Different from tree traversal: there may be multiple paths between two vertices.

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
- We may use a hash table or a bit array
- Requiring $\Theta(|V|)$ memory

The time complexity of graph traversal cannot be better than and should not be worse than $\Theta(|V| + |E|)$

- Connected graphs simplify this to $\Theta(|E|)$
- Worst case: $\Theta(|V|^{2})$

Breadth-first traversal

Breadth-first traversal on a graph:

- 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

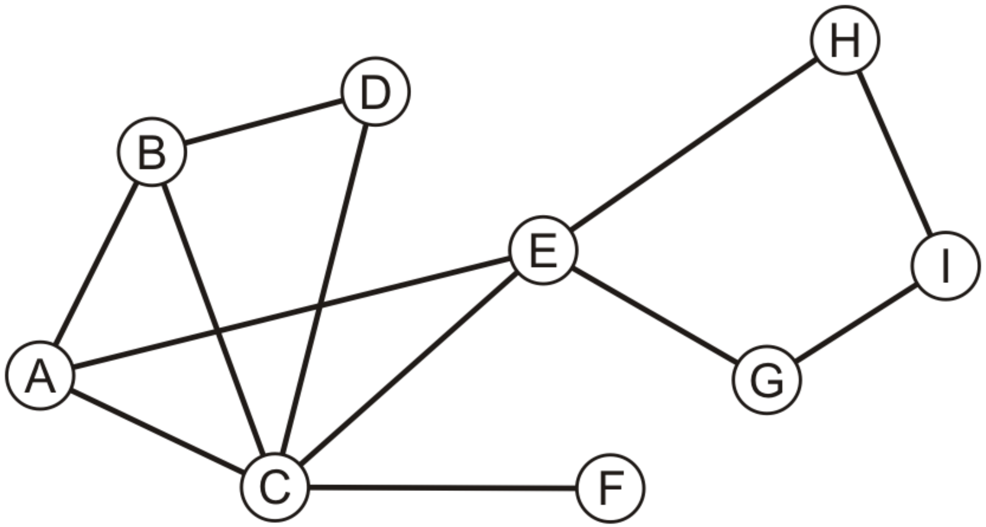
This continues until the queue is empty

- If there are no unvisited vertices, the graph is connected

The size of the queue is $\mathrm{O}(|V|)$

Example

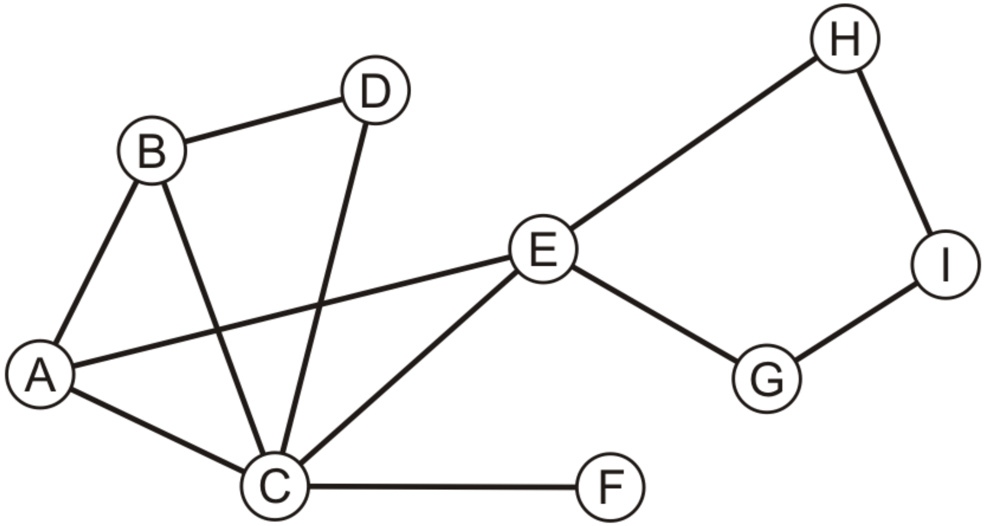
Consider this graph



Example

Performing a breadth-first traversal

- Push the first vertex onto the queue



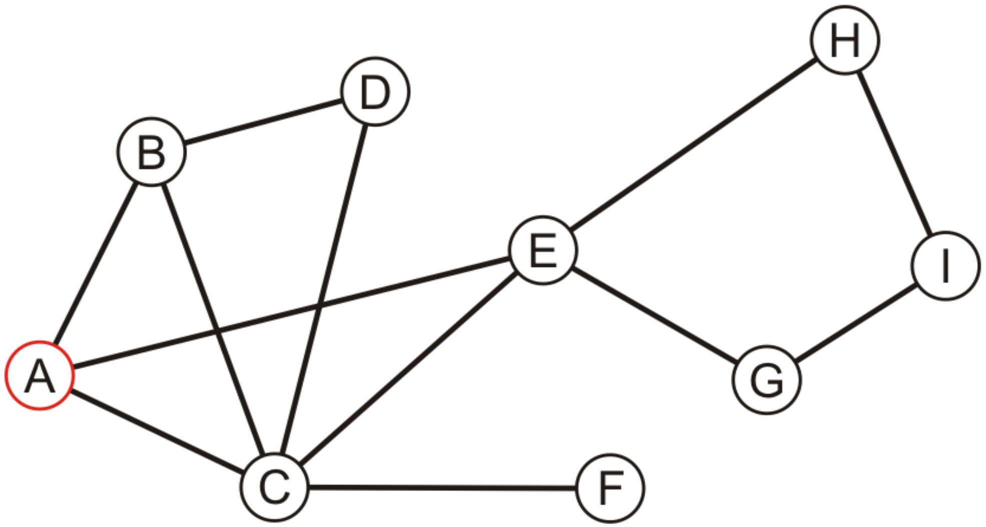
A					

Example

Performing a breadth-first traversal

– Pop A and push B, C and E

A



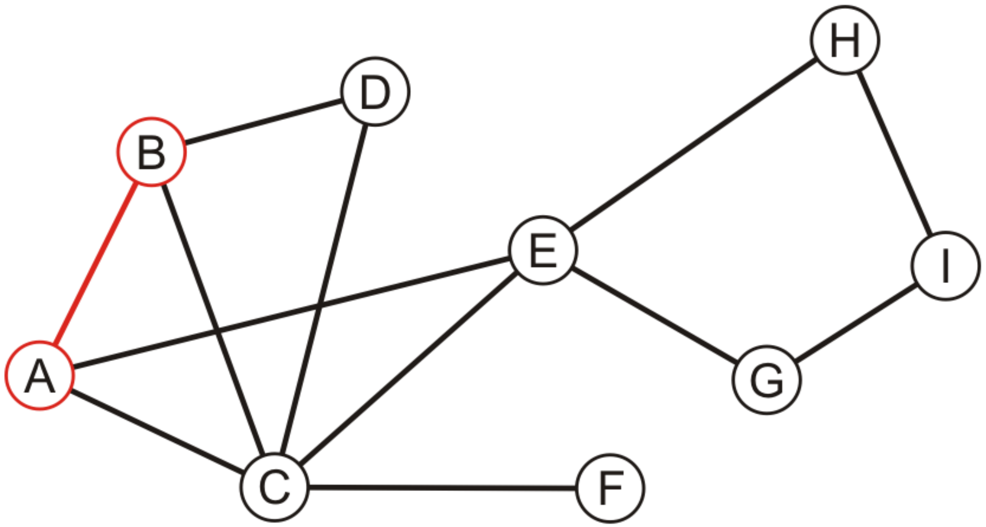
B	C	E			

Example

Performing a breadth-first traversal:

– Pop B and push D

A, B



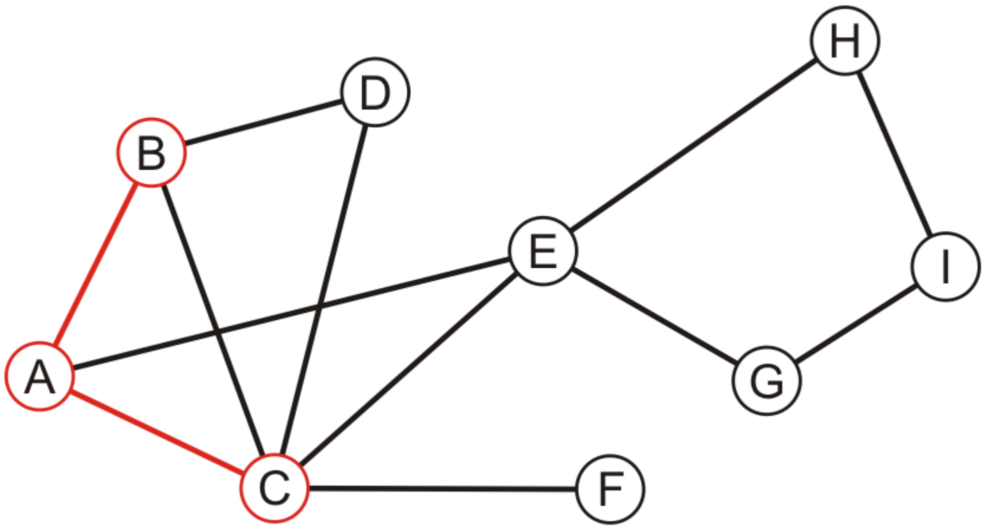
C	E	D			

Example

Performing a breadth-first traversal:

– Pop C and push F

A, B, C



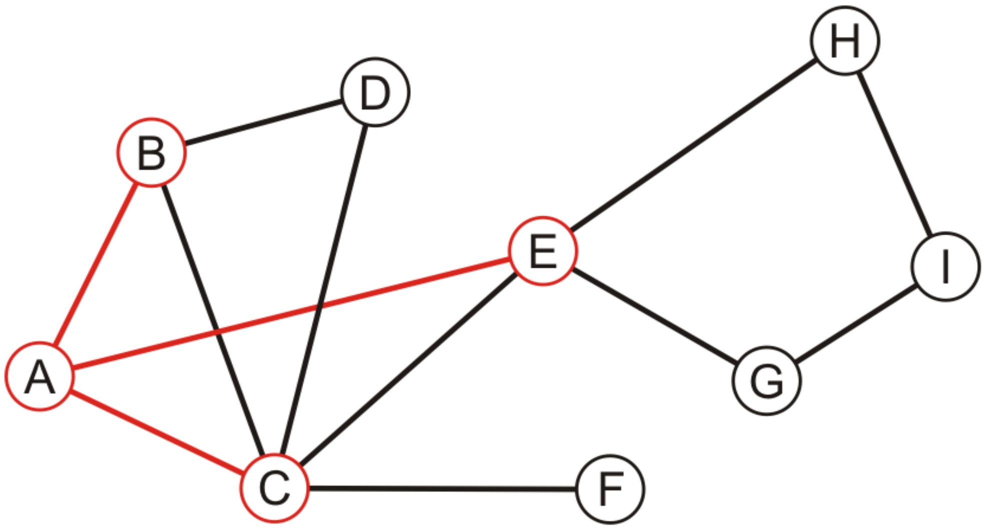
E	D	F			

Example

Performing a breadth-first traversal:

- Pop E and push G and H

A, B, C, E



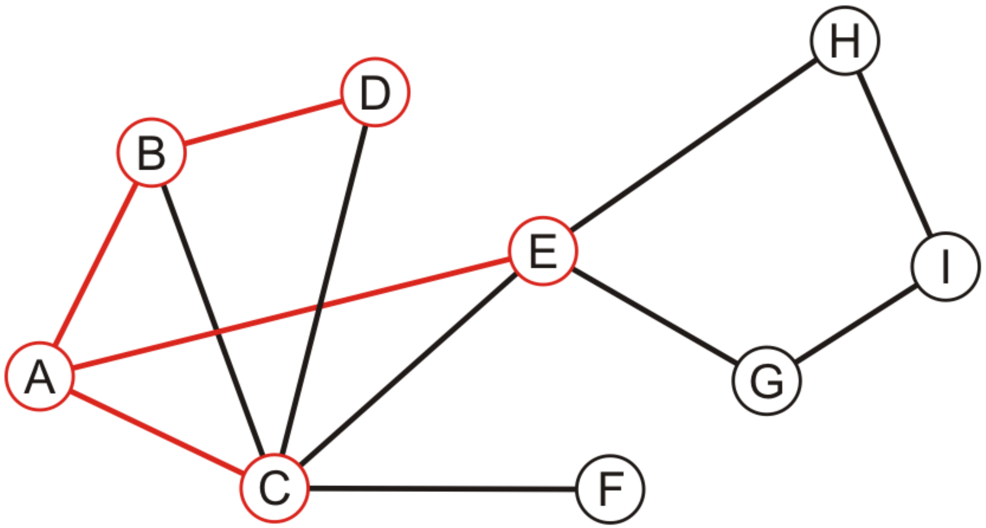
D	F	G	H		

Example

Performing a breadth-first traversal:

– Pop D

A, B, C, E, D



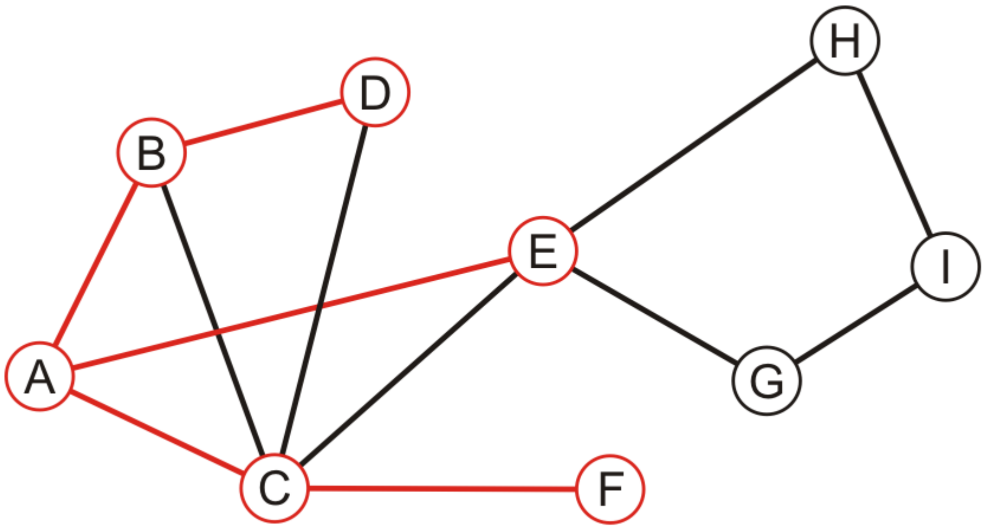
F	G	H			

Example

Performing a breadth-first traversal:

– Pop F

A, B, C, E, D, F



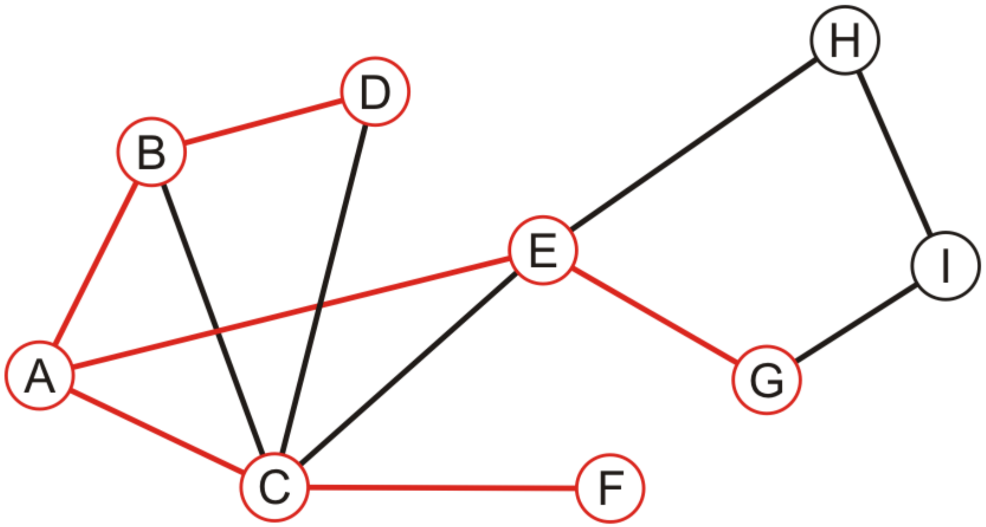
G	H				

Example

Performing a breadth-first traversal:

– Pop G and push I

A, B, C, E, D, F, G



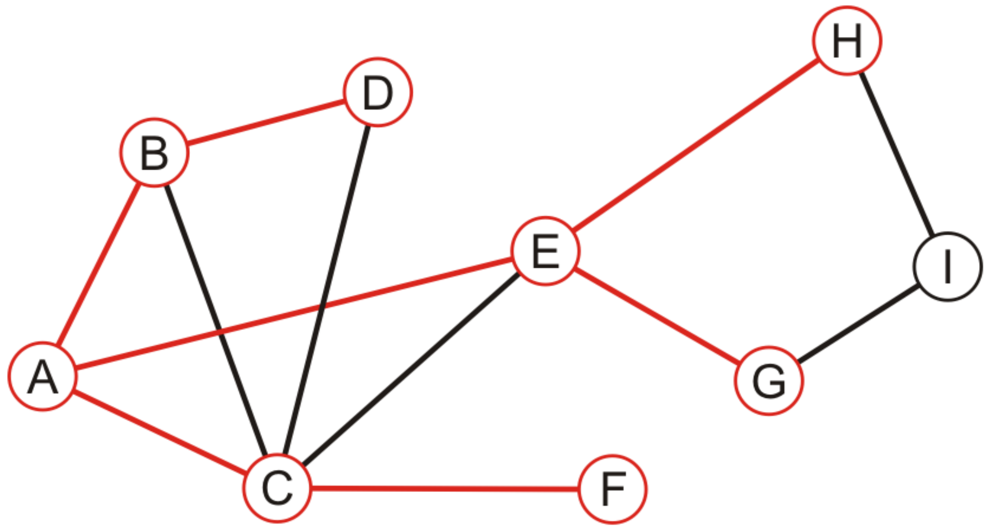
H	I				

Example

Performing a breadth-first traversal:

– Pop H

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



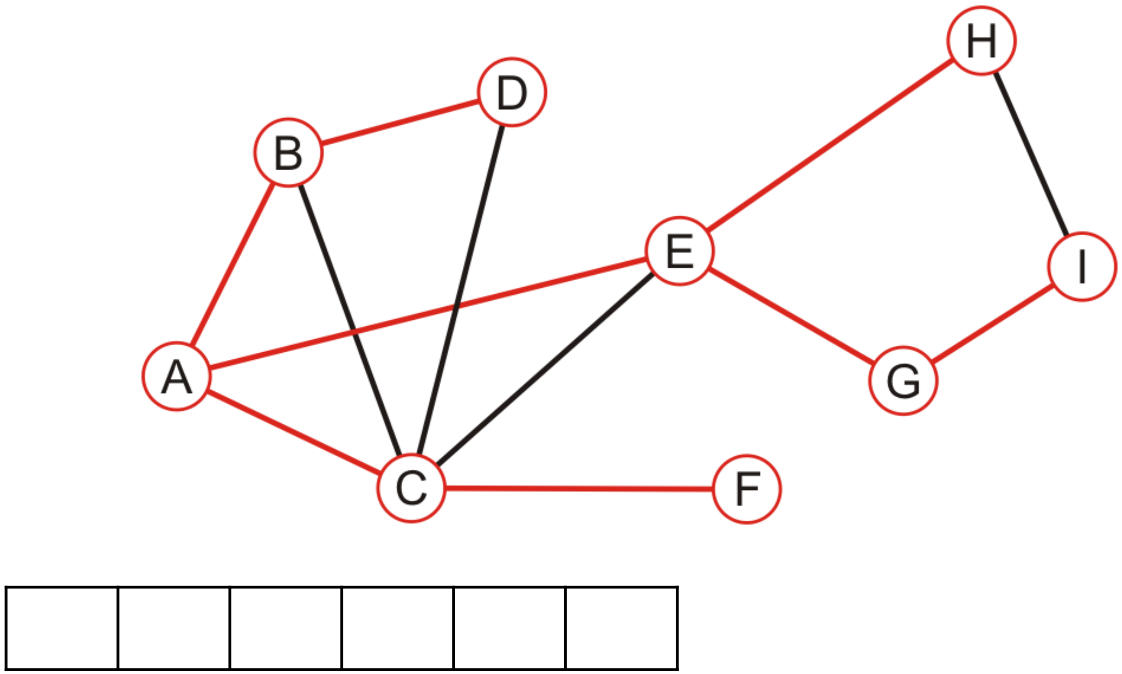
I					

Example

Performing a breadth-first traversal:

– Pop I

A, B, C, E, D, F, G, H, I

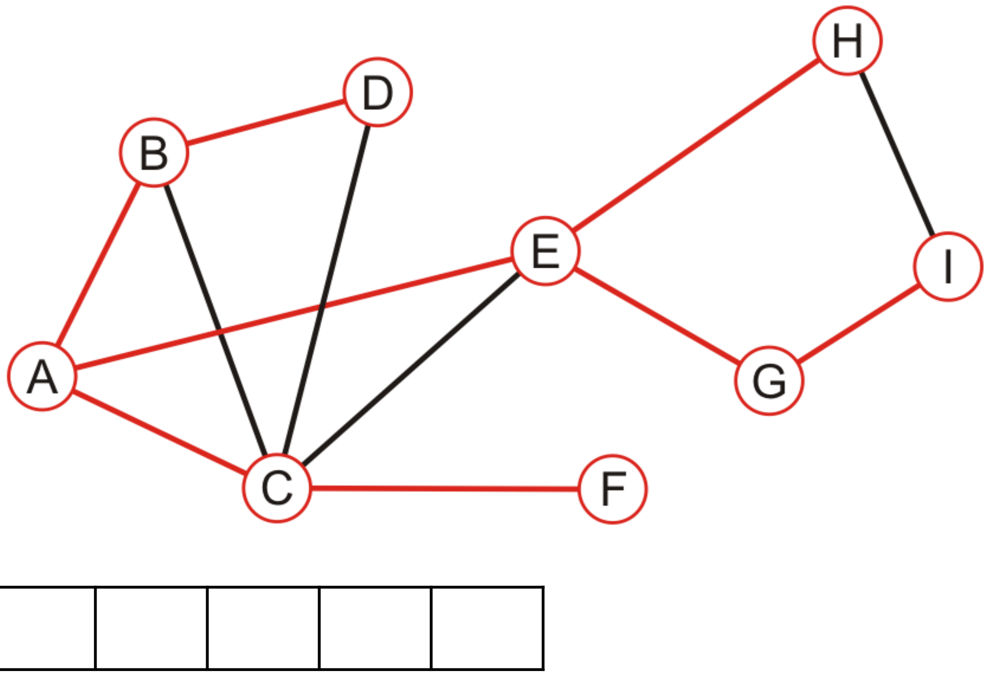


Example

Performing a breadth-first traversal:

- The queue is empty: we are finished

A, B, C, E, D, F, G, H, I



Depth-first traversal

Depth-first traversal on a graph:

- Choose any vertex, mark it as visited
- 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

Two implementations:

- Recursive
- Use a stack

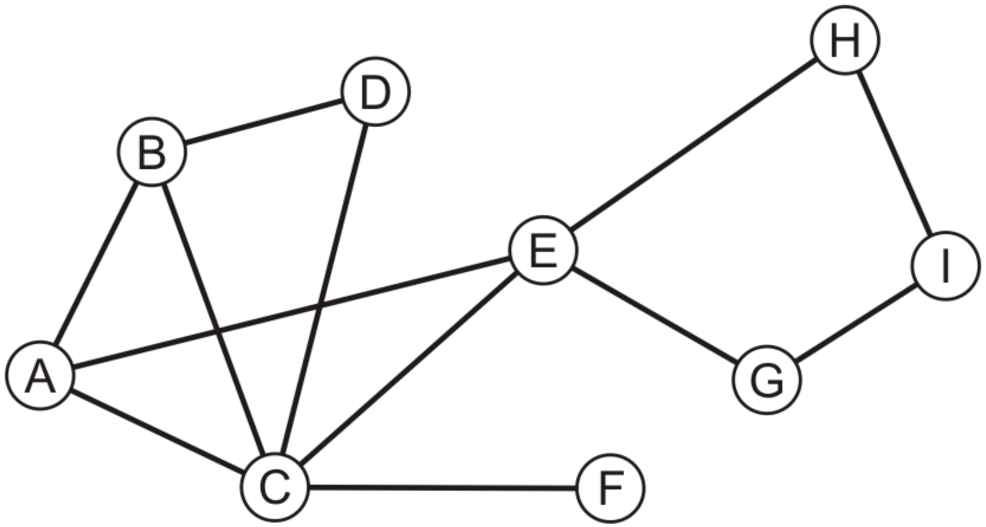
Depth-first traversal

Use a stack:

- Choose any vertex
  - Mark it as visited
  - Place it onto an empty stack
- While the stack is not empty:
  - If the vertex on the top of the stack has an unvisited adjacent vertex v,
    - Mark v as visited
    - Place v onto the top of the stack
  - Otherwise, pop the top of the stack

Example

Perform a recursive depth-first traversal on this same graph

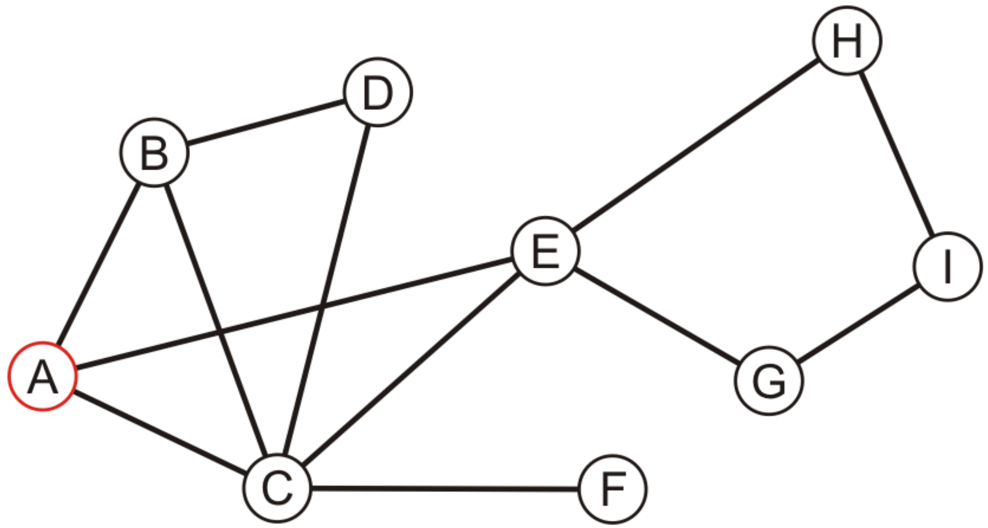


Example

Performing a recursive depth-first traversal:

- Visit the first node

A

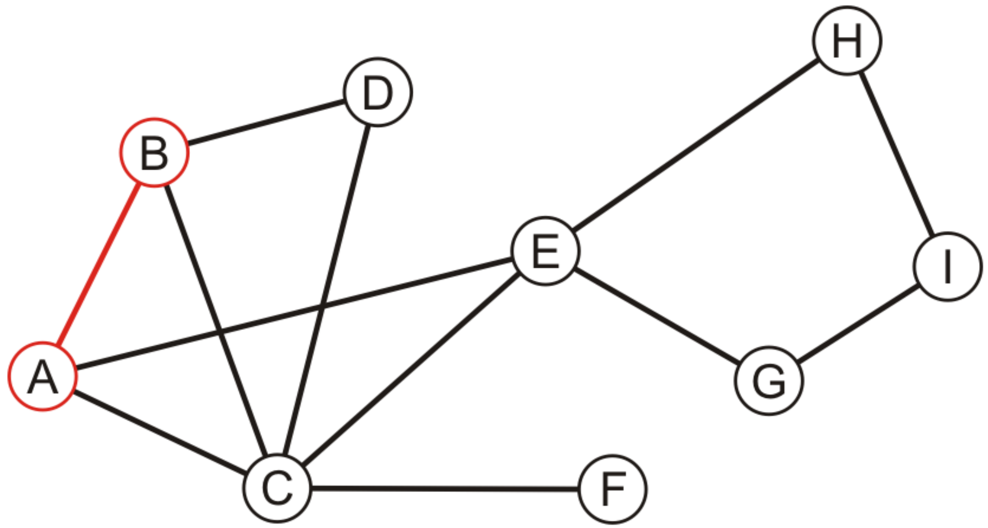


Example

Performing a recursive depth-first traversal:

– A has an unvisited neighbor

A, B

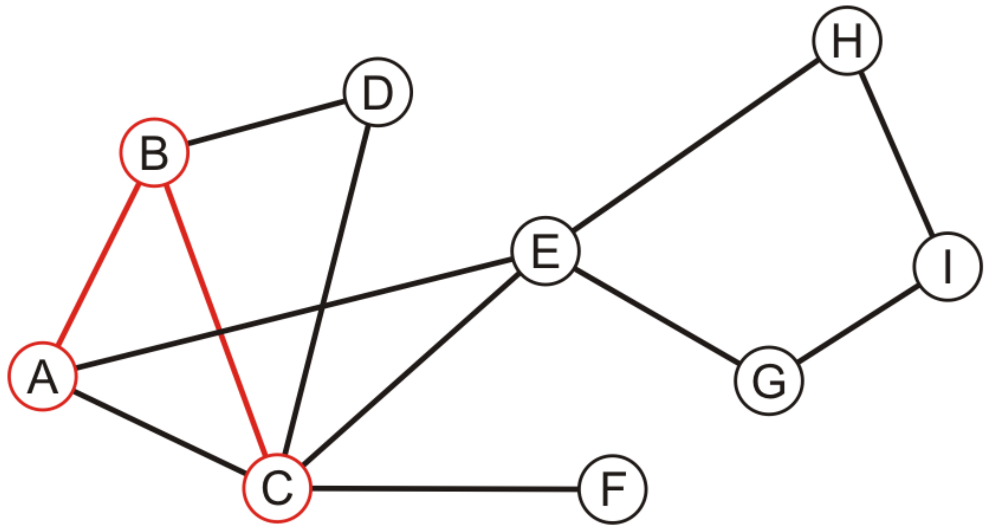


Example

Performing a recursive depth-first traversal:

– B has an unvisited neighbor

A, B, C

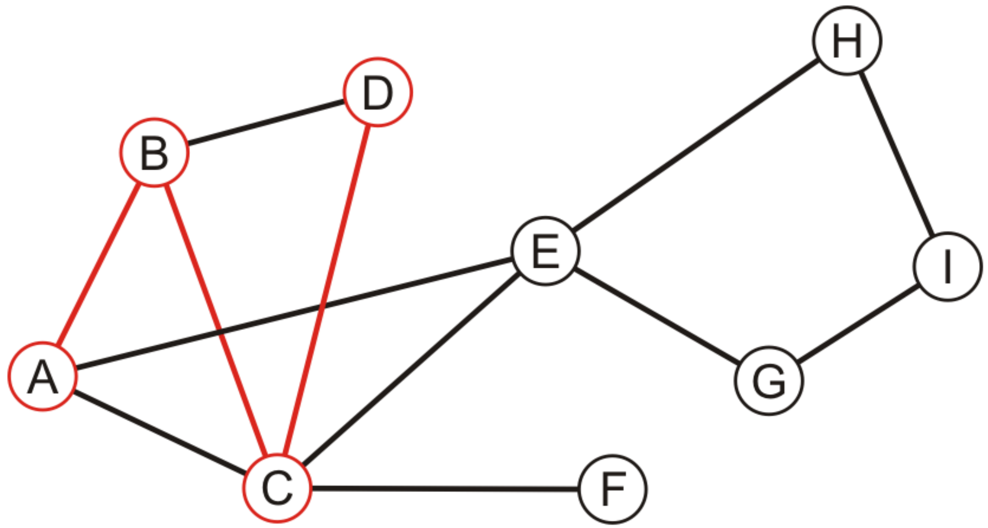


Example

Performing a recursive depth-first traversal:

– C has an unvisited neighbor

A, B, C, D

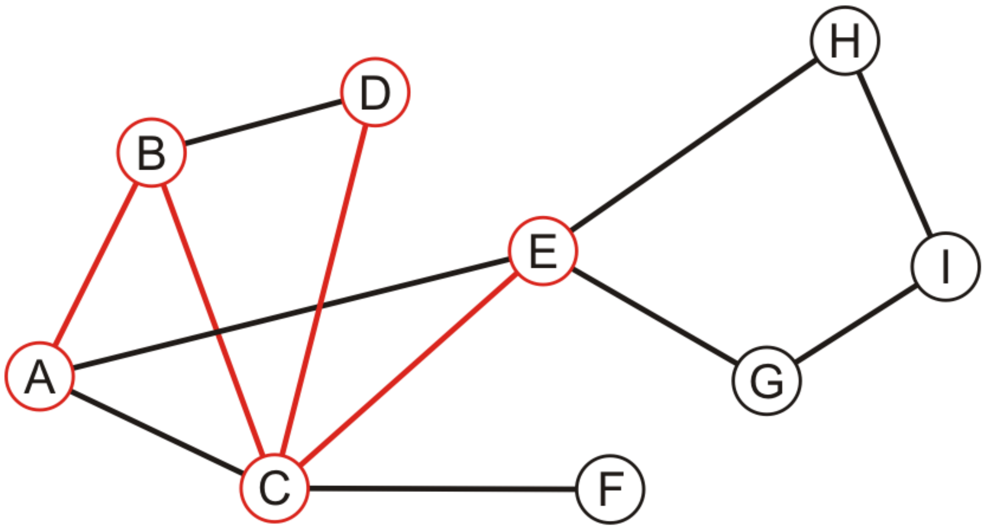


Example

Performing a recursive depth-first traversal:

– D has no unvisited neighbors, so we return to C

A, B, C, D, E

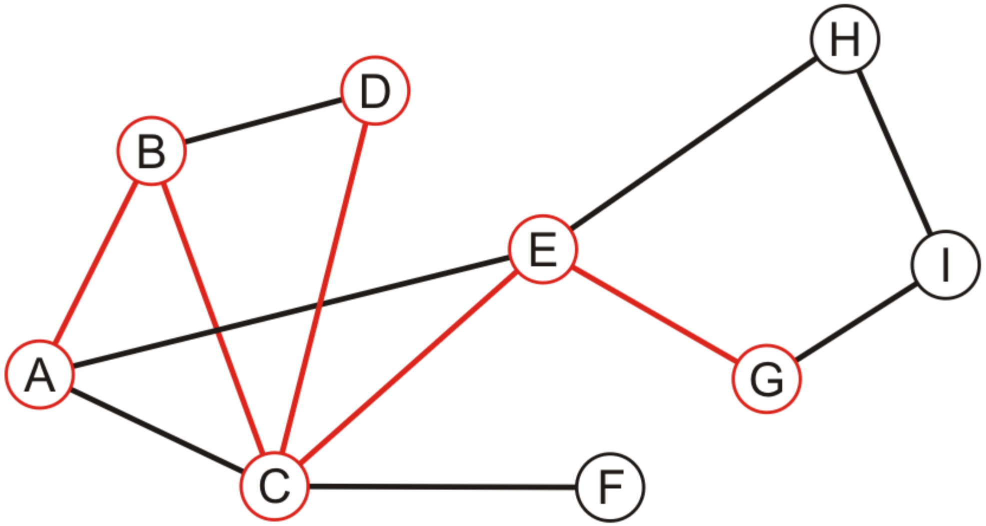


Example

Performing a recursive depth-first traversal:

– E has an unvisited neighbor

A, B, C, D, E, G

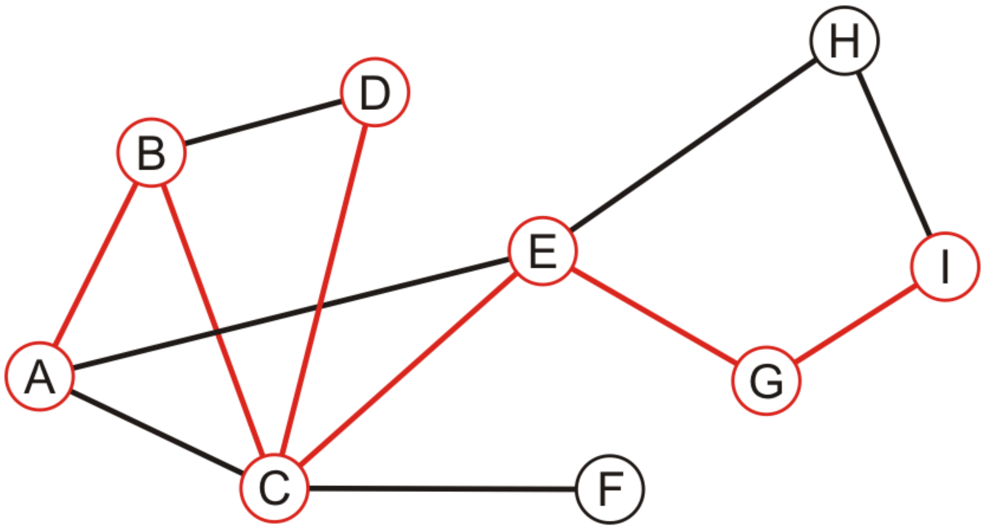


Example

Performing a recursive depth-first traversal:

– G has an unvisited neighbor

A, B, C, D, E, G, I

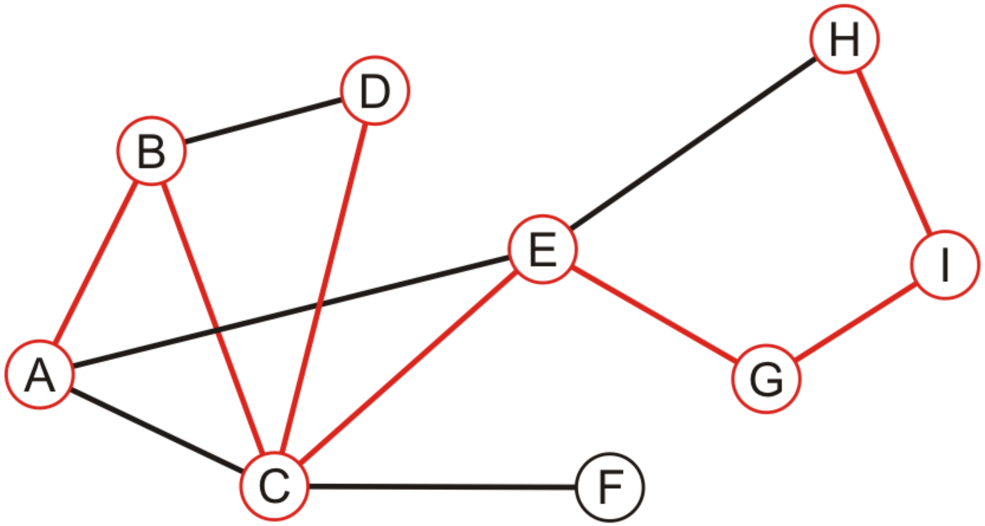


Example

Performing a recursive depth-first traversal:

– I has an unvisited neighbor

A, B, C, D, E, G, I, H

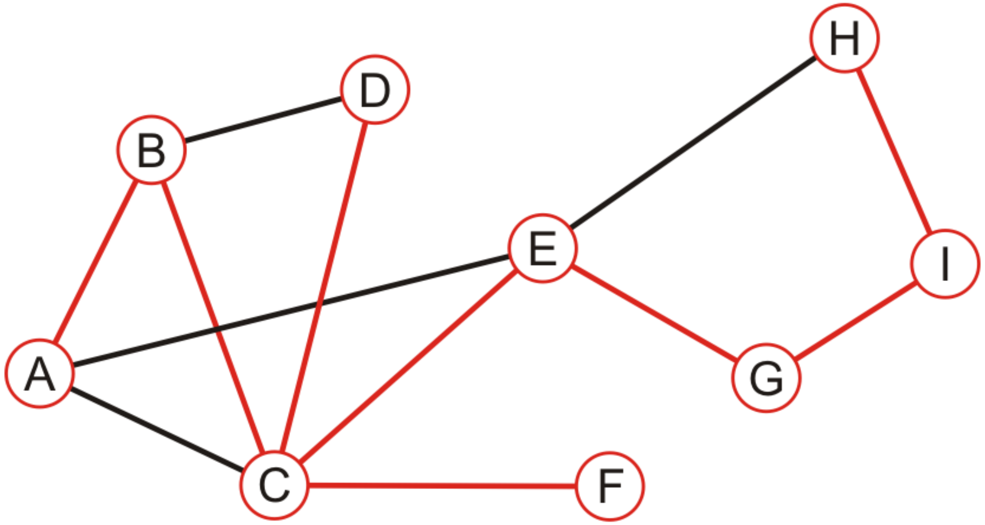


Example

Performing a recursive depth-first traversal:

- We recurse back to C which has an unvisited neighbour

A, B, C, D, E, G, I, H, F

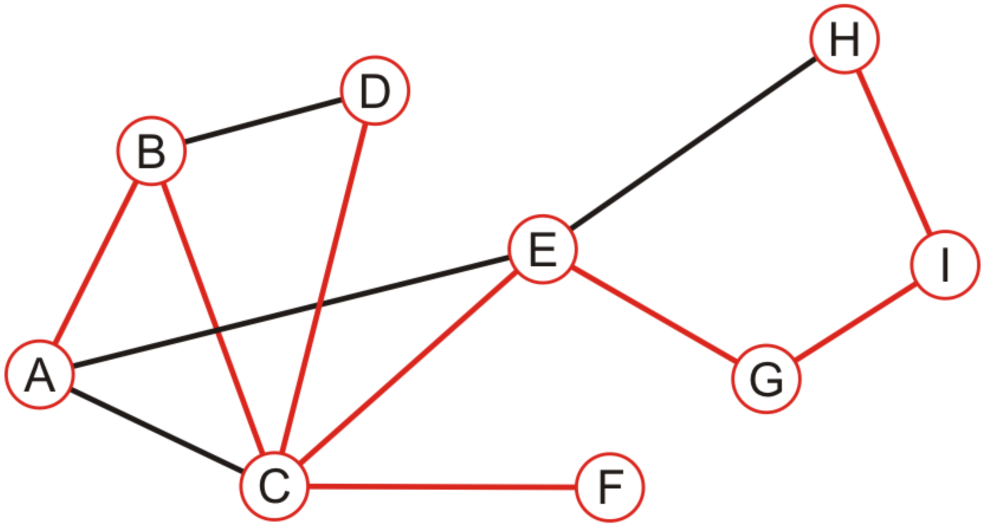


Example

Performing a recursive depth-first traversal:

- We recurse finding that no other nodes have unvisited neighbours

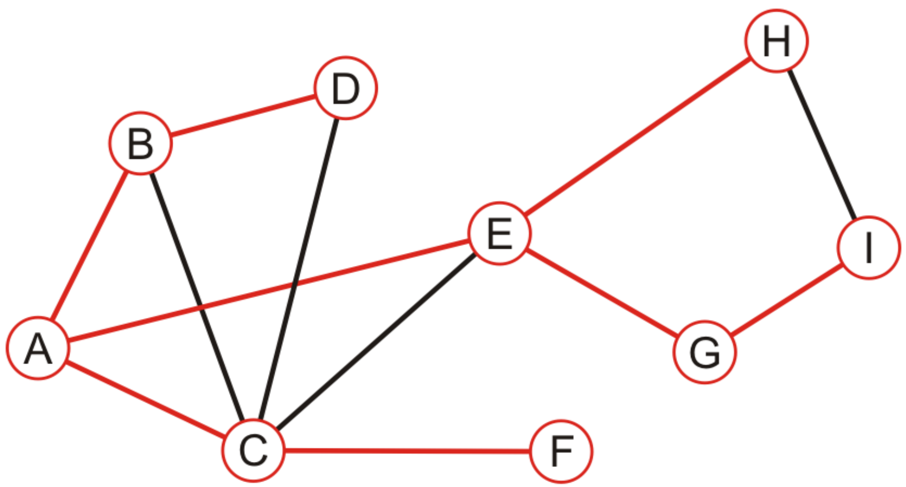
A, B, C, D, E, G, I, H, F



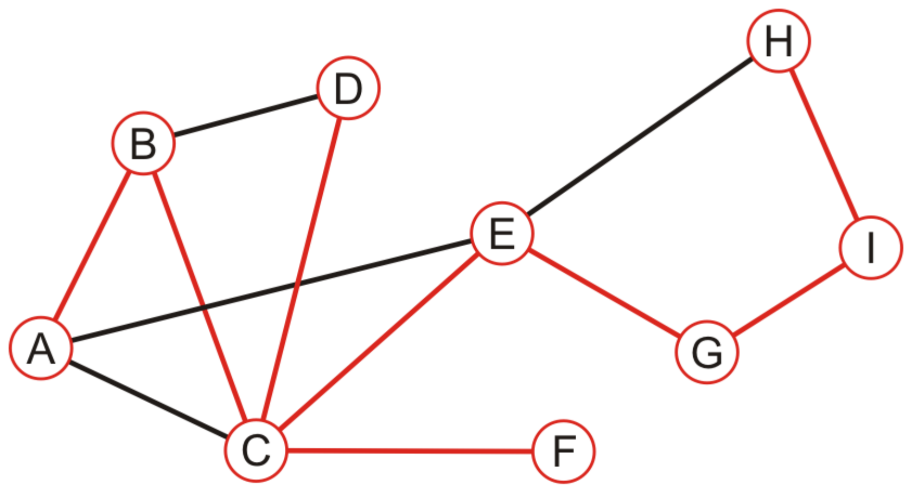
Comparison

The order in which vertices can differ greatly
- An iterative depth-first traversal may also be different again

A, B, C, E, D, F, G, H, I



A, B, C, D, E, G, I, H, F



Outline

- Graph traversal
  - Breadth-first
  - Depth-first
- Applications
  - Connectedness
  - Unweighted path length
  - Identifying bipartite graphs

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

Connected

Consider implementing a breadth-first traversal on an undirected graph:

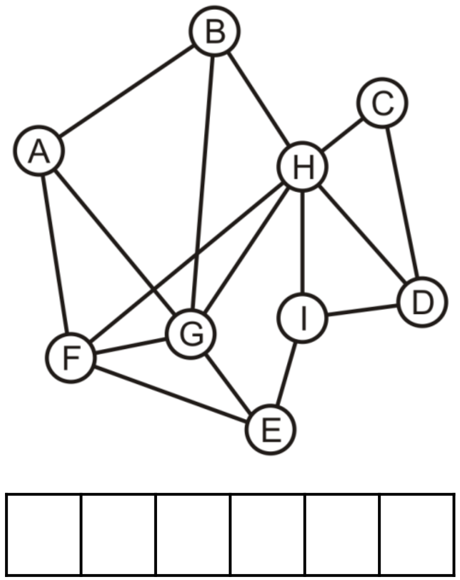
- Choose any vertex, mark it as visited and push it onto queue
- While the queue is not empty:
  - Pop to top vertex v from the queue
  - For each vertex adjacent to v that has not been visited:
    - Mark it visited, and
    - Push it onto the queue

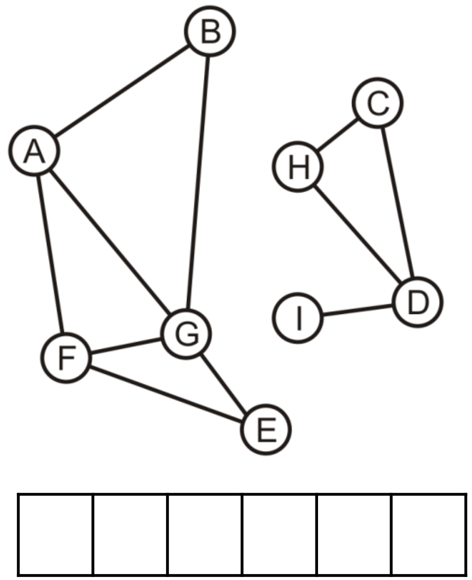
This continues until the queue is empty

- Note: if there are no unvisited vertices, the graph is connected,

Determining Connections

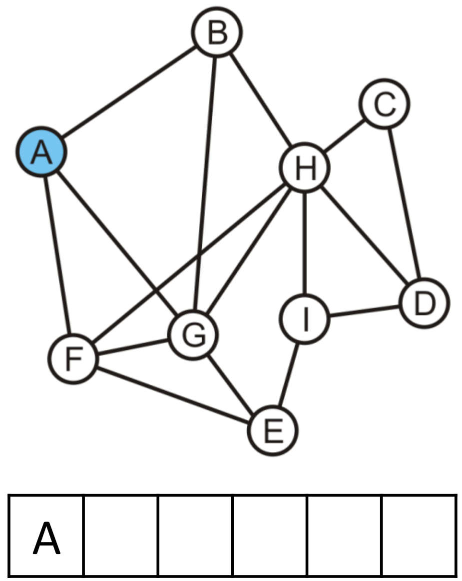
Is A connected to D?

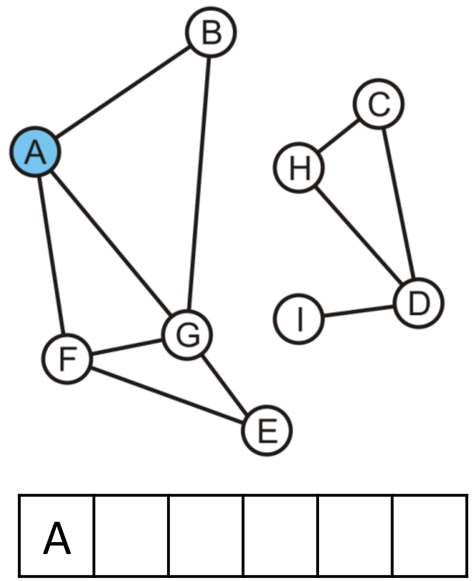




Determining Connections

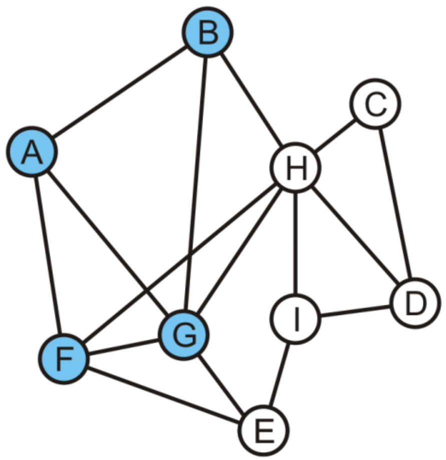
Vertex A is marked as visited and pushed onto the queue



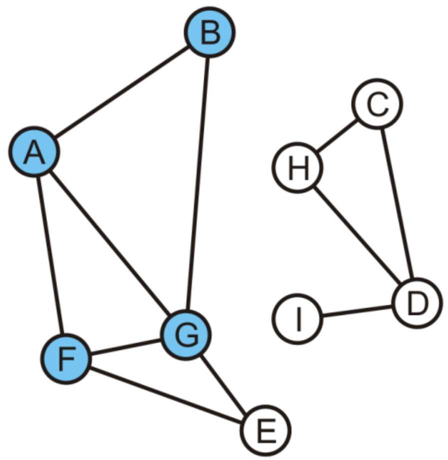


Determining Connections

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



B	F	G			

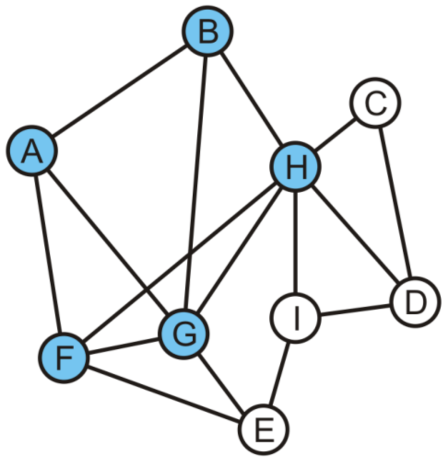


B	F	G			

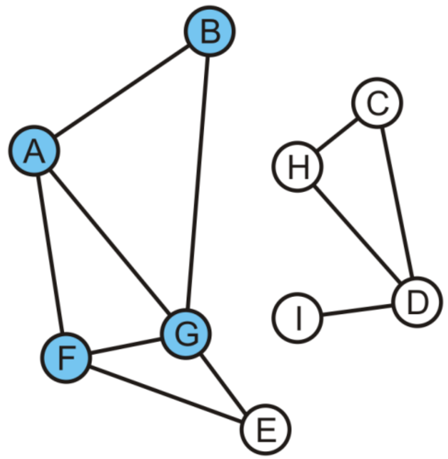
Determining Connections

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

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



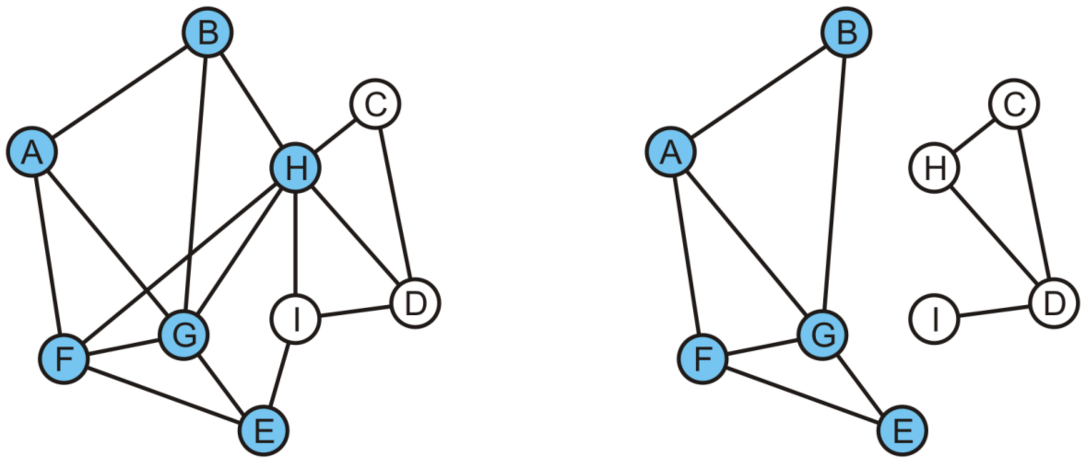
F	G	H			



F	G				

Determining Connections

Popping F results in the pushing of E

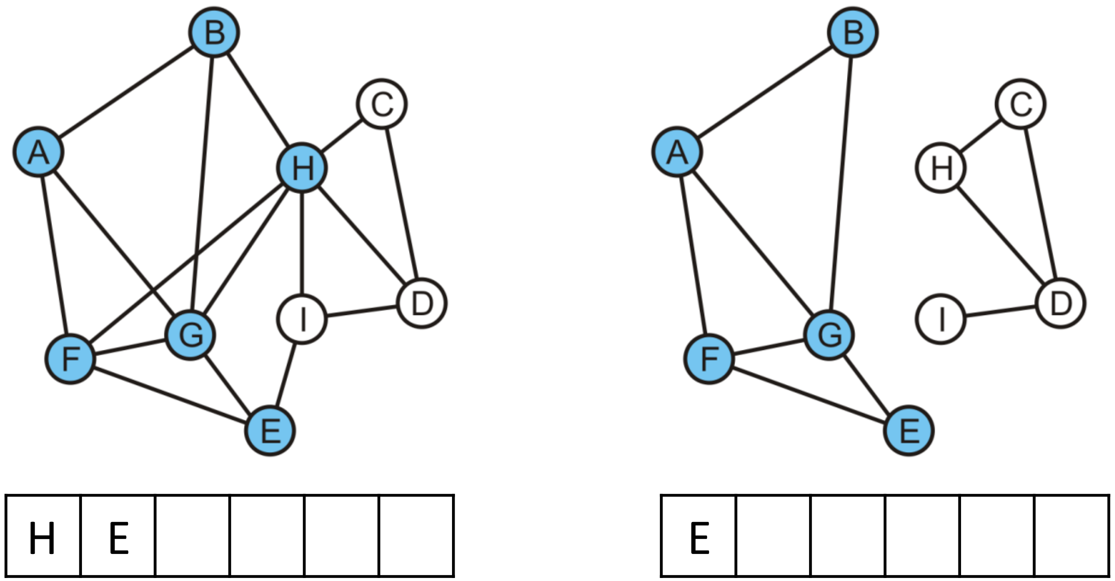


G	H	E			

G	E				

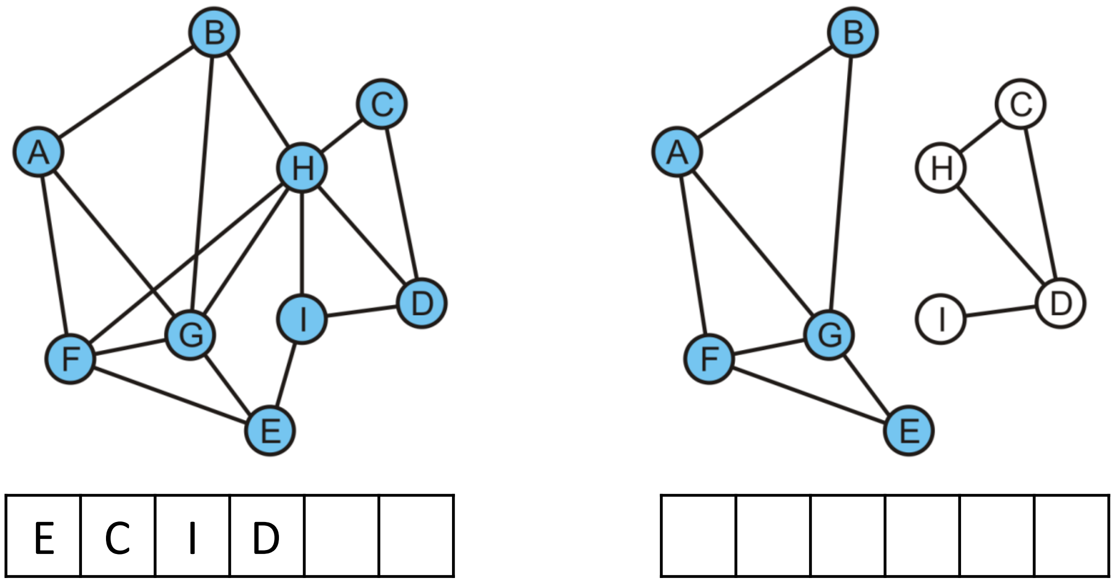
Determining Connections

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



Determining Connections

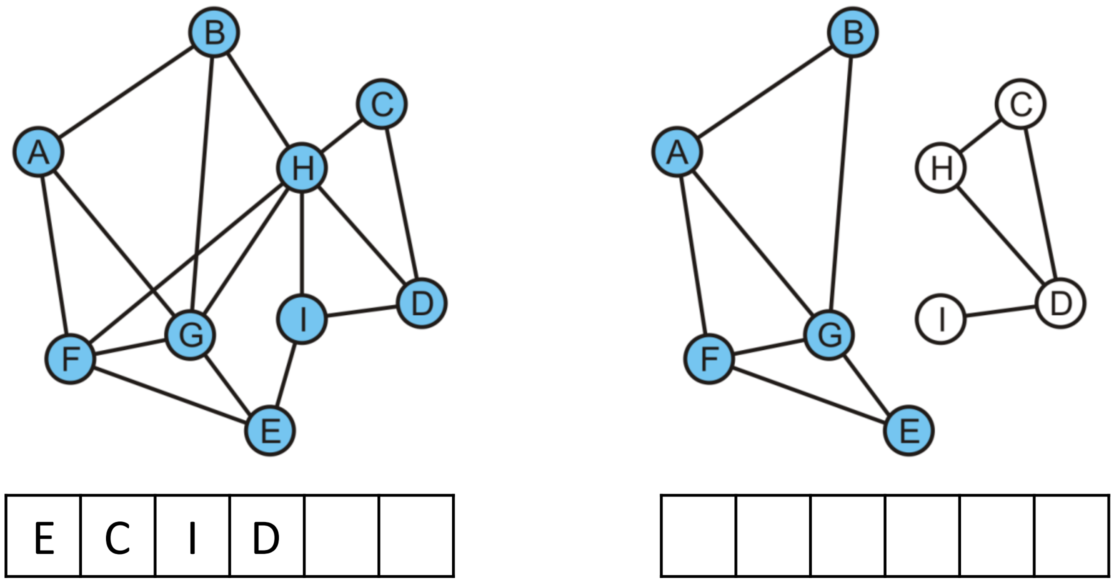
Popping H on the left graph results in C, I, D being pushed



Determining Connections

On the left, D is now visited

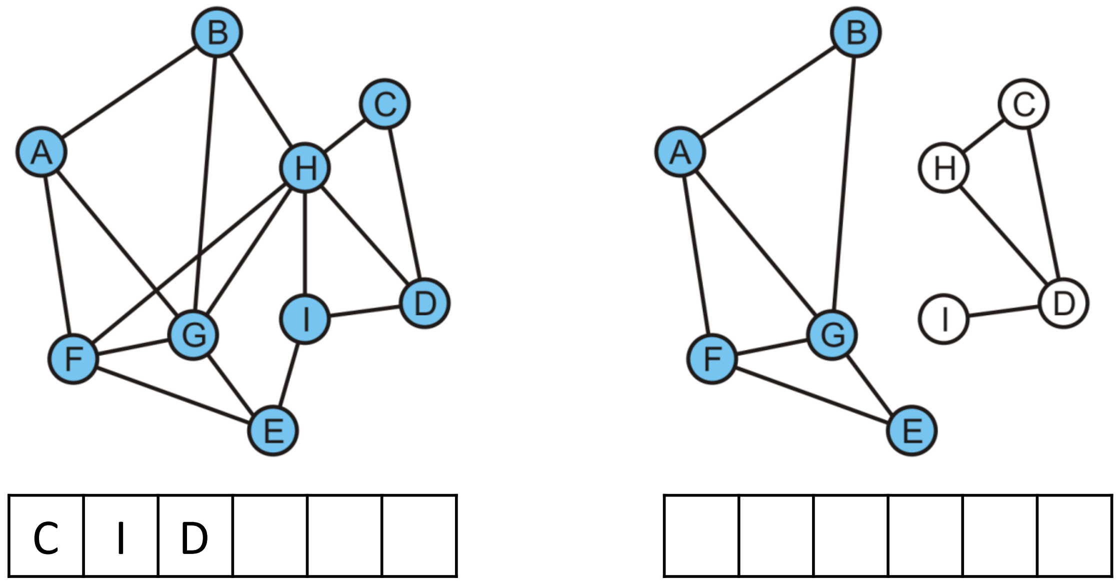
- We determine A is connected to D



Determining Connections

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

- We determine A is not connected to D



Connected Components

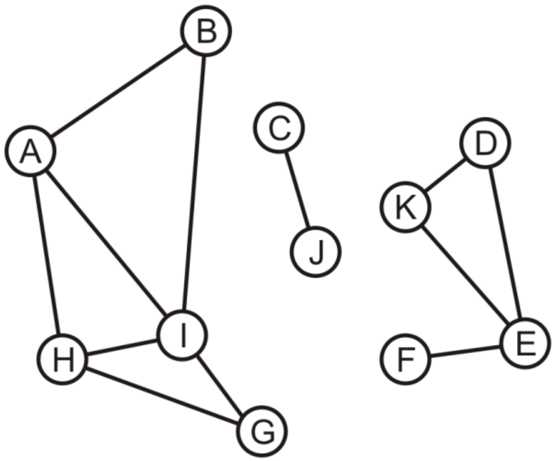
Suppose we want to partition the vertices into connected sub-graphs

- 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

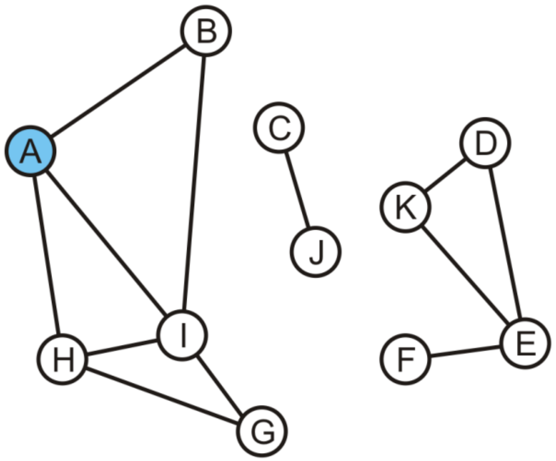
Here we start with a set of singletons



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

Connected Components

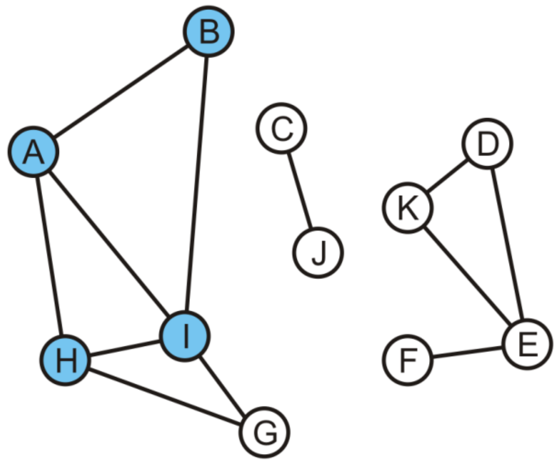
The vertex A is unvisited, so we start with it



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}

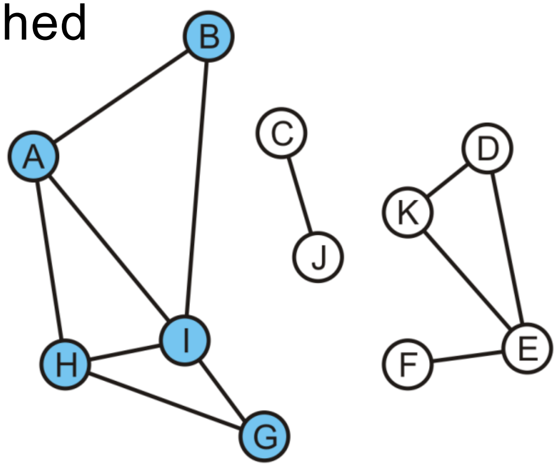


A	B	C	D	E	F	G	H	I	J	K
A	**A**	**C**	**D**	**E**	**F**	**G**	**A**	**A**	**J**	**K**

Connected Components

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

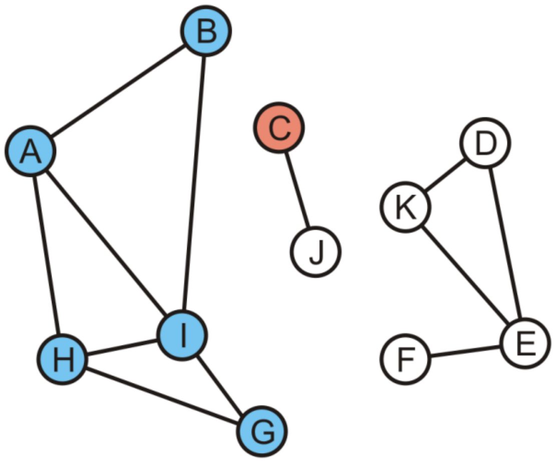
- The traversal is finished



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

Connected Components

Start another traversal with C: this defines a new set $\{C\}$

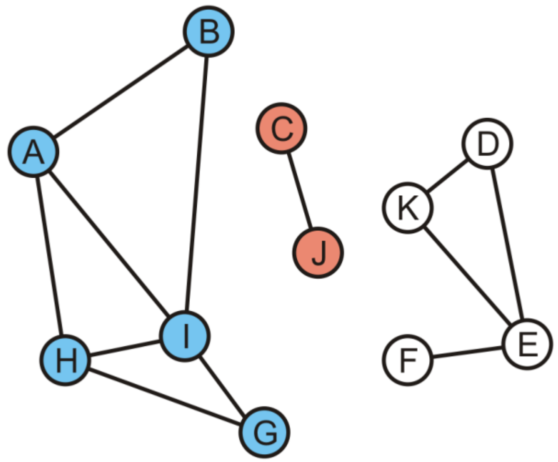


A	B	C	D	E	F	G	H	I	J	K
A	**A**	**C**	**D**	**E**	**F**	**A**	**A**	**A**	**J**	**K**

Connected Components

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

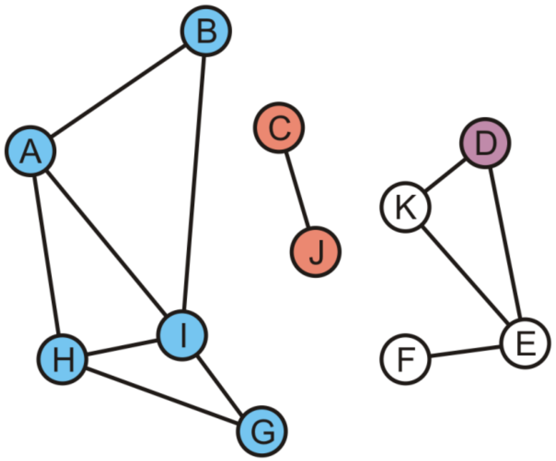
– This traversal is finished



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

Connected Components

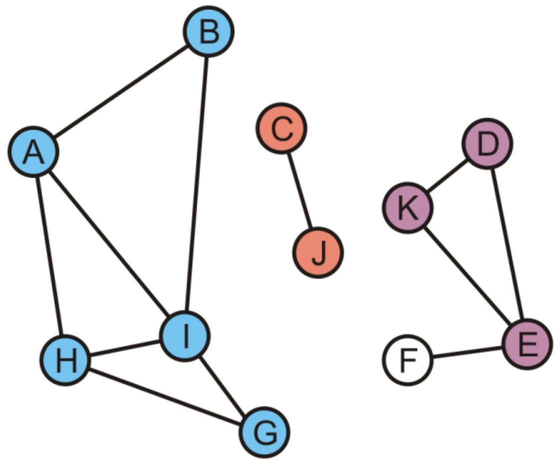
We start again with the set $\{D\}$



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 $\{D, E, K\}$

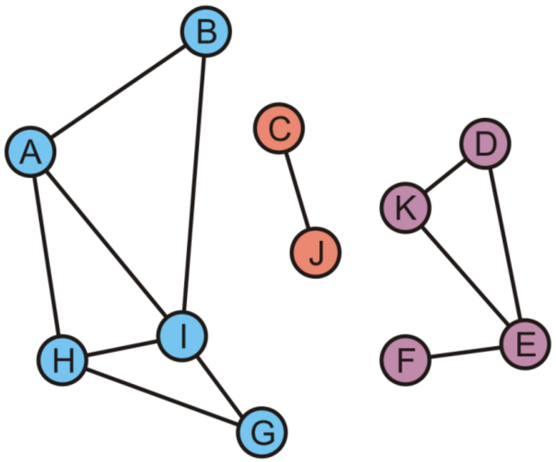


A	B	C	D	E	F	G	H	I	J	K
A	**A**	**C**	**D**	**D**	**F**	**A**	**A**	**A**	**C**	**D**

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}

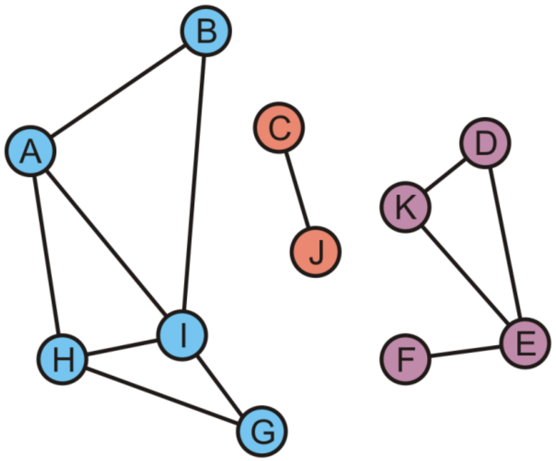


A	B	C	D	E	F	G	H	I	J	K
A	A	C	D	D	D	A	A	A	C	D

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**

Tracking Unvisited Vertices

How do you implement a set of unvisited vertices so as to:

- Find an unvisited vertex in $\Theta(1)$ time?
- Remove a vertex that has been visited from this list in $\Theta(1)$ time?

Bad solution

- We can simply flag vertices as visited, but this would require $O(|V|)$ time to find an unvisited vertex

Good solutions

- A hash table of unvisited vertices
- Or, an array of unvisited vertices, and we store for each vertex its position in the array

Tracking Unvisited Vertices

Create two arrays:

- One array, `unvisited`, will contain the unvisited vertices
- The other, `loc_in_unvisited`, will contain the location of vertex v_i in the first array

0	1	2	3	4	5	6	7	8	9	10
A	B	C	D	E	F	G	H	I	J	K

A	B	C	D	E	F	G	H	I	J	K
0	1	2	3	4	5	6	7	8	9	10

- Or, instead of a second array, we may add a member variable in the vertex class

Tracking Unvisited Vertices

Suppose we visit D
- D is in entry 3
- How shall we delete D in the first array?

0	1	2	3	4	5	6	7	8	9	10
A	B	C	D	E	F	G	H	I	J	K

A	B	C	**D**	E	F	G	H	I	J	K
0	1	2	**3**	4	5	6	7	8	9	10

Tracking Unvisited Vertices

Suppose we visit D

- D is in entry 3
- Copy the last unvisited vertex into this location and update the location array for this value

0	1	2	3	4	5	6	7	8	9	10
A	B	C	**K**	E	F	G	H	I	J	

A	B	C	D	E	F	G	H	I	J	**K**
0	1	2	3	4	5	6	7	8	9	**3**

Tracking Unvisited Vertices

Suppose we visit G

– G is in entry 6

0	1	2	3	4	5	6	7	8	9	10
A	B	C	K	E	F	G	H	I	J	

A	B	C	D	E	F	**G**	H	I	J	K
0	1	2	3	4	5	**6**	7	8	9	3

Tracking Unvisited Vertices

Suppose we visit G

- G is in entry 6
- Copy the last unvisited vertex into this location and update the location array for this value

0	1	2	3	4	5	6	7	8	9	10
A	B	C	K	E	F	J	H	I		

A	B	C	D	E	F	G	H	I	J	K
0	1	2	3	4	5	6	7	8	6	3

Tracking Unvisited Vertices

Suppose we now visit K

– K is in entry 3

0	1	2	3	4	5	6	7	8	9	10
A	B	C	K	E	F	J	H	I		

A	B	C	D	E	F	G	H	I	J	K
0	1	2	3	4	5	6	7	8	6	3

Tracking Unvisited Vertices

Suppose we now visit K

– K is in entry 3

– Copy the last unvisited vertex into this location and update the location array for this value

0	1	2	3	4	5	6	7	8	9	10
A	B	C	I	E	F	J	H			

A	B	C	D	E	F	G	H	I	J	K
0	1	2	3	4	5	6	7	**3**	6	3

Tracking Unvisited Vertices

If we want to find an unvisited vertex, we simply return the last entry of the first array and return it

0	1	2	3	4	5	6	7	8	9	10
A	B	C	I	E	F	J	H			

A	B	C	D	E	F	G	H	I	J	K
0	1	2	3	4	5	6	7	3	6	3

Tracking Unvisited Vertices

In this case, an unvisited vertex is H

- Removing it is trivial: just decrement the count of unvisited vertices

0	1	2	3	4	5	6	7	8	9	10
A	B	C	I	E	F	J				

A	B	C	D	E	F	G	H	I	J	K
0	1	2	3	4	5	6	7	3	6	3

Outline

- Graph traversal
  - Breadth-first
  - Depth-first
- Applications
  - Connectedness
  - Unweighted path length
  - Identifying bipartite graphs

Determining Distances

Problem: in an unweighted graph, find the distances from one vertex v to all the other vertices

– Distance: the length of the shortest path between two vertices

Method:

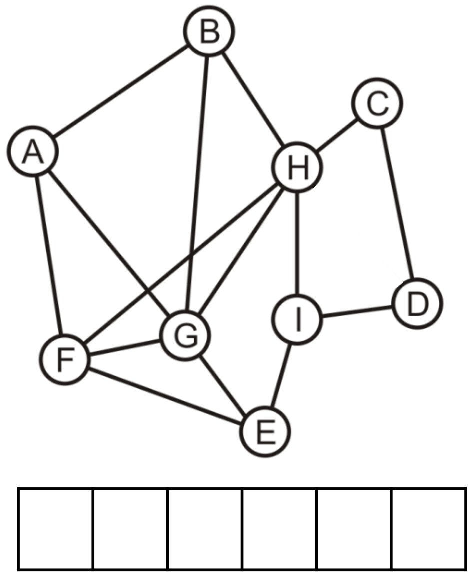
– Use a breadth-first traversal
– Vertices are added in *layers*
– The starting vertex v is defined to be in the zeroth layer, L_0
– While the k^{th} layer is not empty, all unvisited vertices adjacent to vertices in L_k are added to the $(k + 1)^{\text{st}}$ layer

The distance from v to vertices in L_k is k

Any unvisited vertices are said to have an infinite distance from v

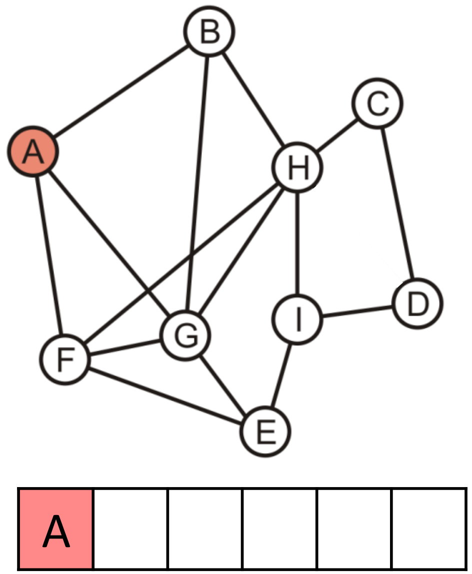
Determining Distances

Consider this graph: find the distance from A to each other vertex



Determining Distances

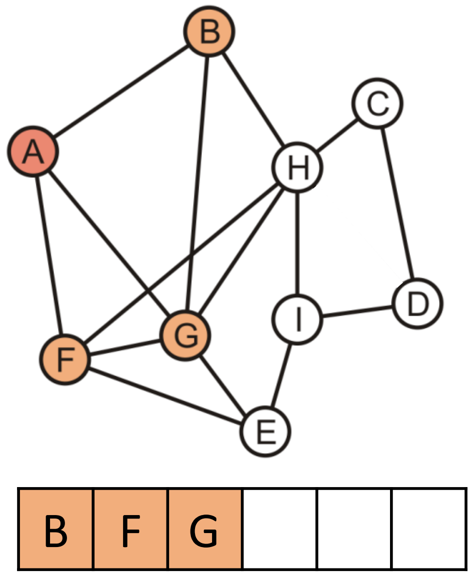
A forms the zeroeth layer, L_0



Determining Distances

The unvisited vertices B, F and G are adjacent to A

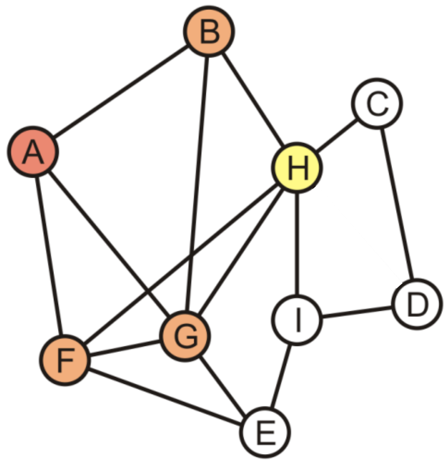
- These form the first layer, L_1



Determining Distances

We now begin popping L_1 vertices: pop B

- H is adjacent to B
- It is tagged L_2

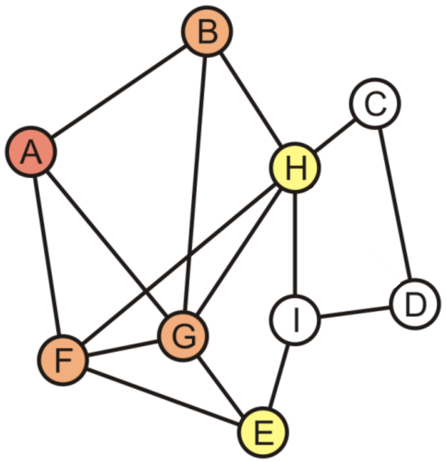


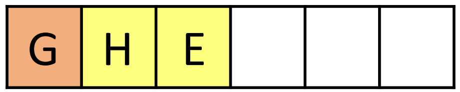
F	G	H			

Determining Distances

Popping F pushes E onto the queue

– It is also tagged L_2

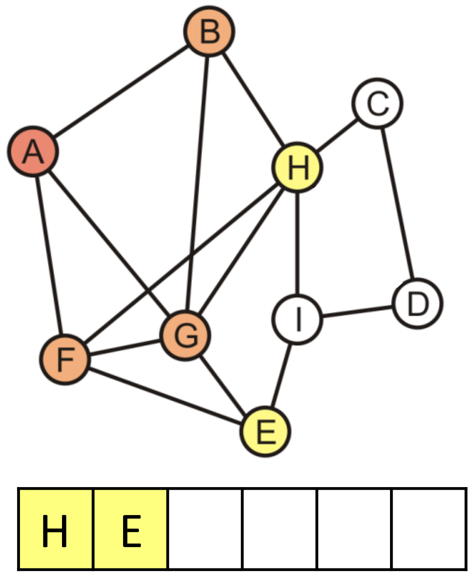




Determining Distances

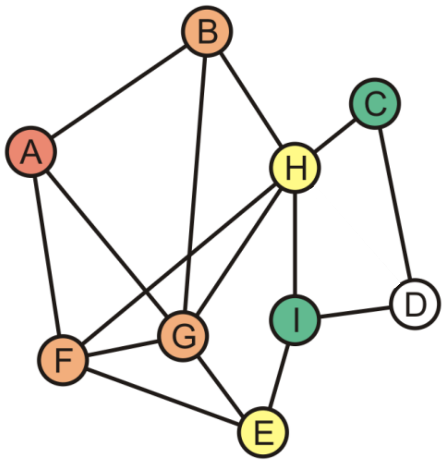
We pop G which has no other unvisited neighbours

– G is the last L_1 vertex; thus H and E form the second layer, L_2



Determining Distances

Popping H in L_2 adds C and I to the third layer L_3

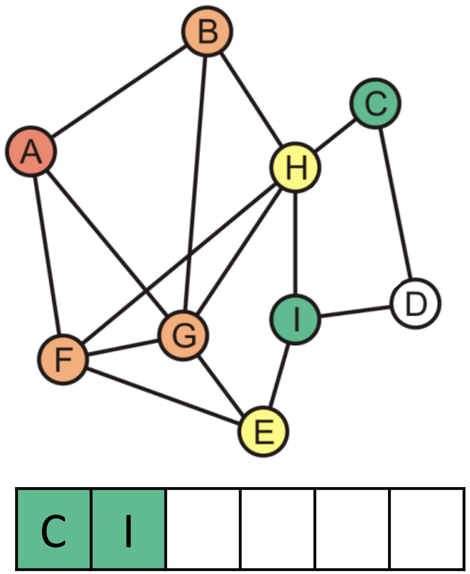


E	C	I			

Determining Distances

E has no more adjacent unvisited vertices

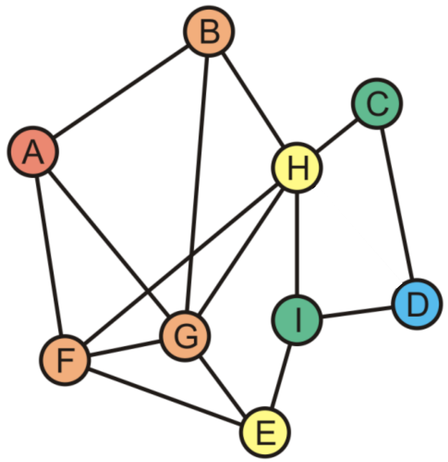
– Thus C and I form the third layer, L_3



Determining Distances

The unvisited vertex D is adjacent to vertices in L_3

- This vertex forms the fourth layer, L_4



- Distance 1: B, F, G
- Distance 2: H, E
- Distance 3: C, I
- Distance 4: D

Determining Distances

Theorem:

- If, in a breadth-first traversal of a graph, two vertices v and w appear in layers L_i and L_j, respectively and $\{v, w\}$ is an edge in the graph, then i and j differ by at most one

Proof:

If $i = j$, we are done

If $i \neq j$, without loss of generality, assume $i < j$

Because $v \in L_i$, w does not appear in any previous layer, and $\{v, w\}$ is an edge in the graph, it follows that $w \in L_{i+1}$

Thus, $j = i + 1$

Therefore, i and j differ by at most one

Outline

- Graph traversal
  - Breadth-first
  - Depth-first
- Applications
  - Connectedness
  - Unweighted path length
  - Identifying bipartite graphs

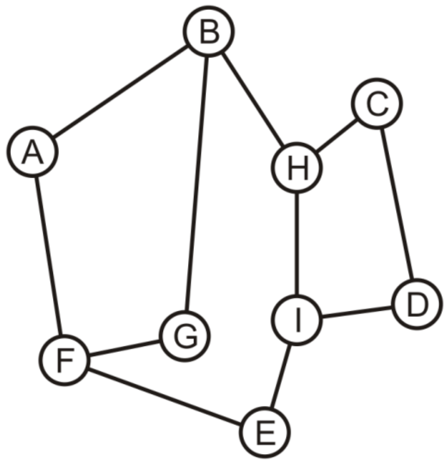
Definition

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

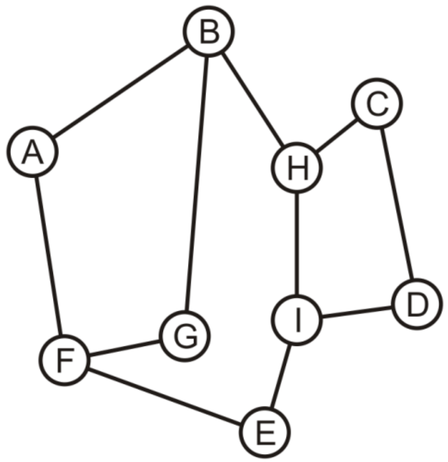
Bipartite Graphs

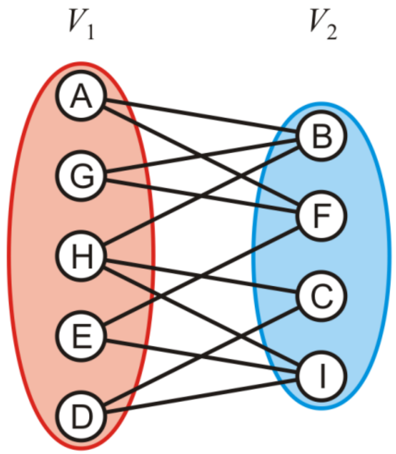
Consider this graph: is it bipartite?



Bipartite Graphs

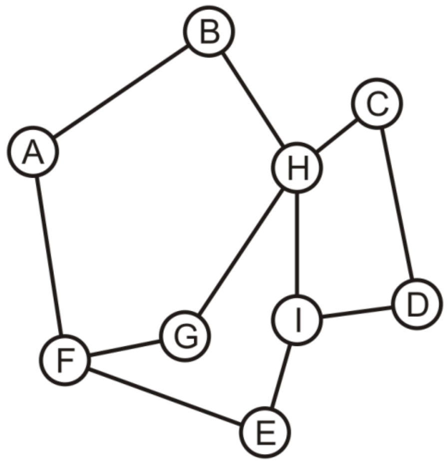
Yes: With a little work, it is possible to determine that we can decompose the vertices into two disjoint sets





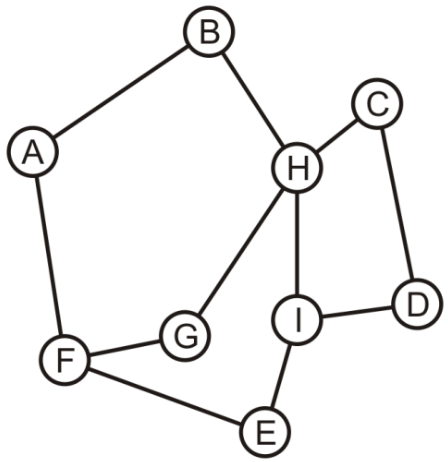
Bipartite Graphs

Is this graph bipartite?



Bipartite Graphs

In this case, it is not a bipartite graph



How can we determine if a graph is bipartite?

Bipartite Graphs

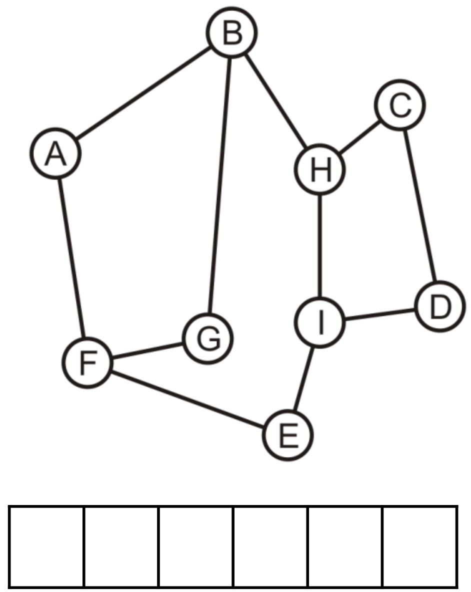
Use a breadth-first traversal for a connected graph:

- 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

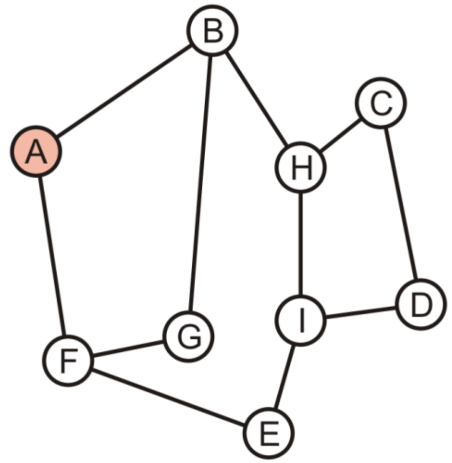
With the first graph, we can start with any vertex

- We will use colours to distinguish the two sets



Bipartite Graphs

Push A onto the queue and colour it red

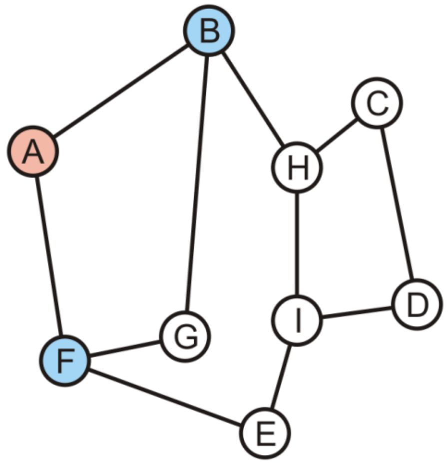


A					

Bipartite Graphs

Pop A and its two neighbours are not marked:

– Mark them as blue and push them onto the queue

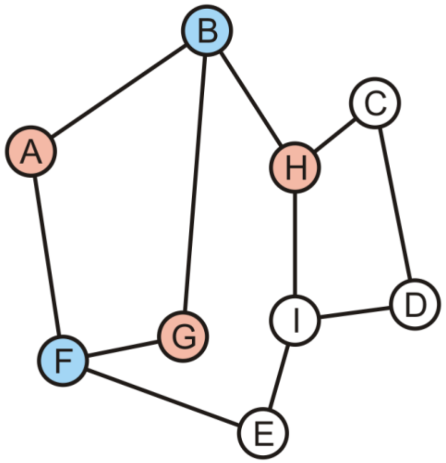


B	**F**				

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

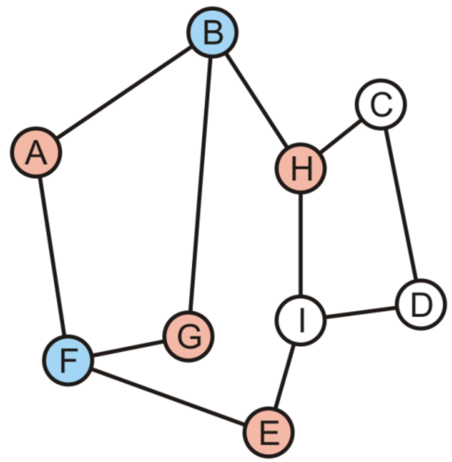


F	G	H			

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

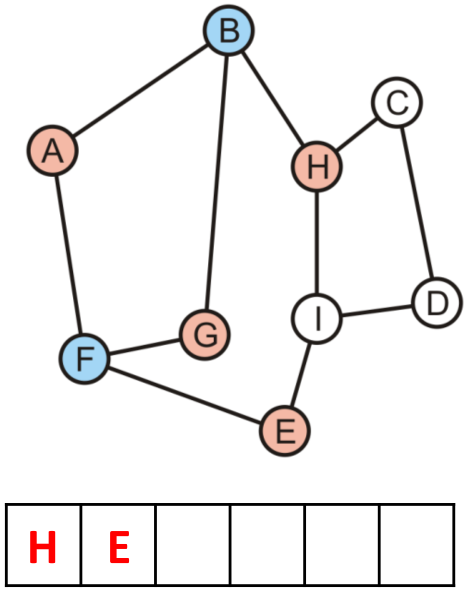


G	H	E			

Bipartite Graphs

Pop G—it is red:

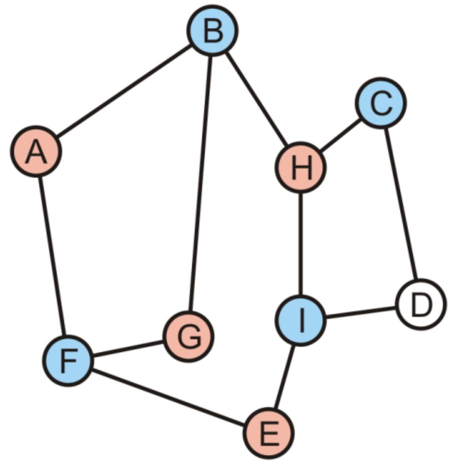
- Its two marked neighbours, B and F, are blue



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

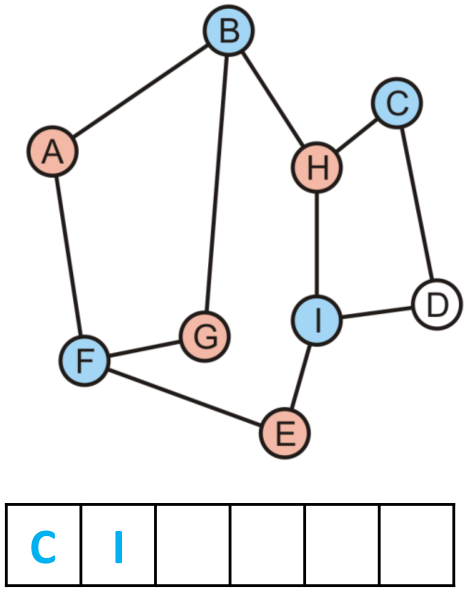


E	C	I			

Bipartite Graphs

Pop E—it is red:

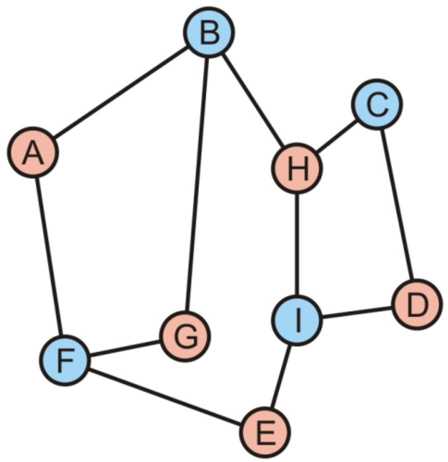
- Its marked neighbours, F and I, are blue



Bipartite Graphs

Pop C—it is blue:

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

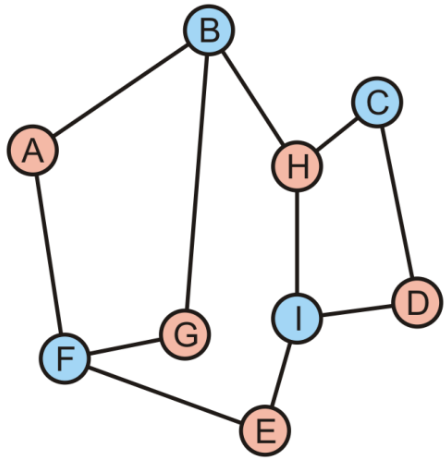


I	D				

Bipartite Graphs

Pop I—it is blue:

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

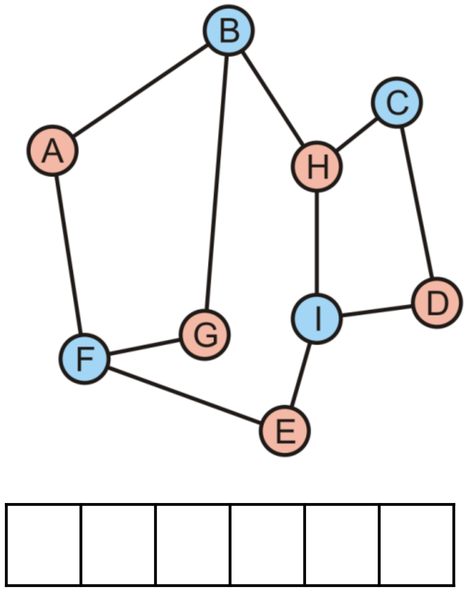


D					

Bipartite Graphs

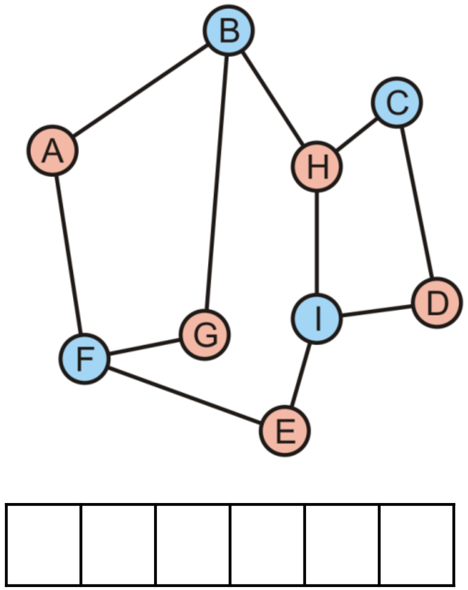
Pop D—it is red:

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



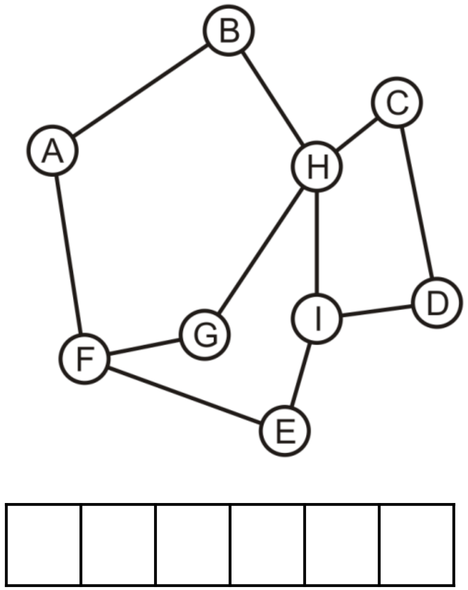
Bipartite Graphs

The queue is empty, the graph is bipartite



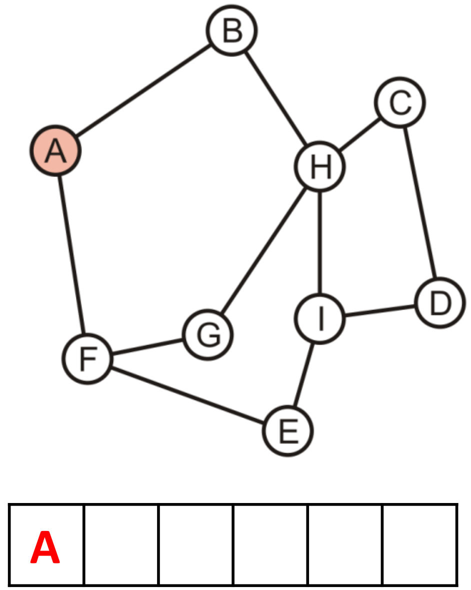
Bipartite Graphs

Consider the other graph which was claimed to be not bipartite



Bipartite Graphs

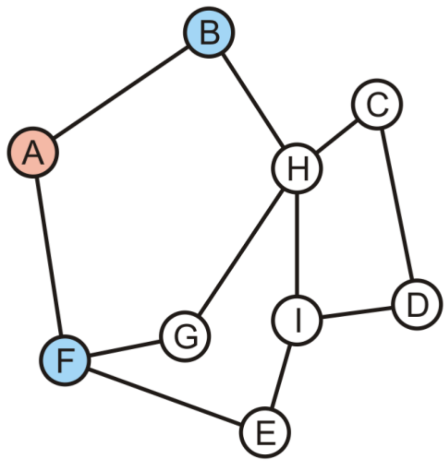
Push A onto the queue and colour it red



Bipartite Graphs

Pop A off the queue:

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

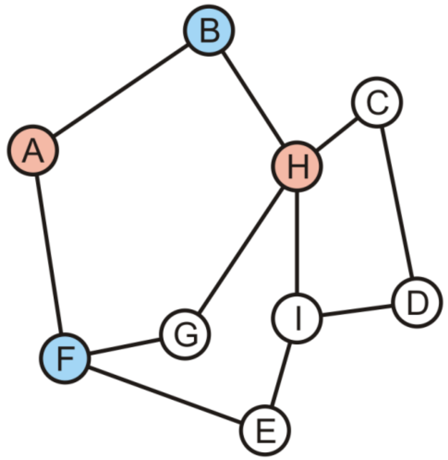


B	F				

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

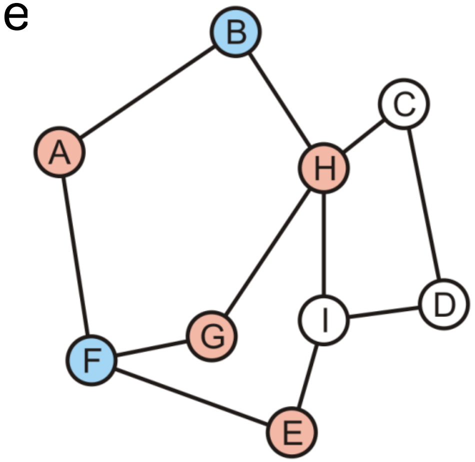


F	H				

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

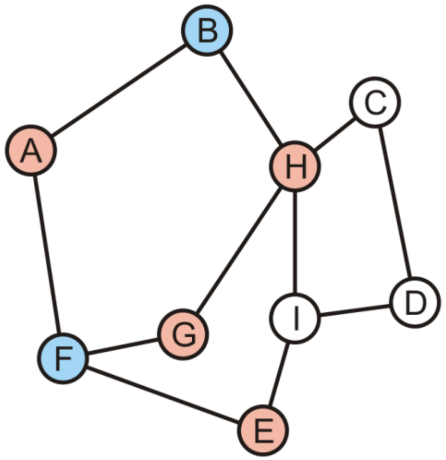


H	E	G			

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

Outline

- Graph traversal
  - Breadth-first: use a queue
  - Depth-first: use recursion or stack
- Applications
  - Connectedness
  - Unweighted path length
  - Identifying bipartite graphs