Graph Traversal

COMP9312_23T2

UNSW SYDNEY

Outline

- BFS
- DFS
- Connectivity
- Topological sort

Breath-first and depth-first traversals

Strategies

Traversals of graphs are also called searches

Applications of BFS

- Shortest Path
- …

Applications of DFS

- Strongly connected component
- Topological Order
- …

A quick view:

https://seanperfecto.github.io/BFS-DFS-Pathfinder/

Breadth-first traversal

Consider implementing a 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 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

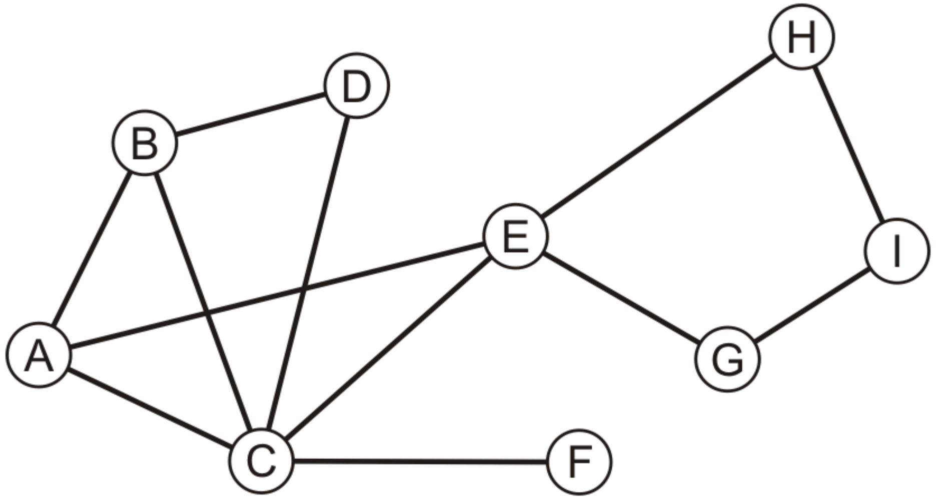
Breadth-first traversal

An implementation can use a queue

```
void Graph::breadth_first_traversal( Vertex first ) {
    bool<Vertex> visited(|V|, false);
    visited[first] = true;
    queue<Vertex> q;
    q.push( first );

    while ( !q.empty() ) {
        Vertex v = q.front();
        q.pop();
        print the vertex v;
        for ( Vertex w : v->adjacent_vertices() ) {
            if ( ! visited[w] ) {
                visited[w] = true;
                q.push( w );
            }
        }
    }
}
```

Example

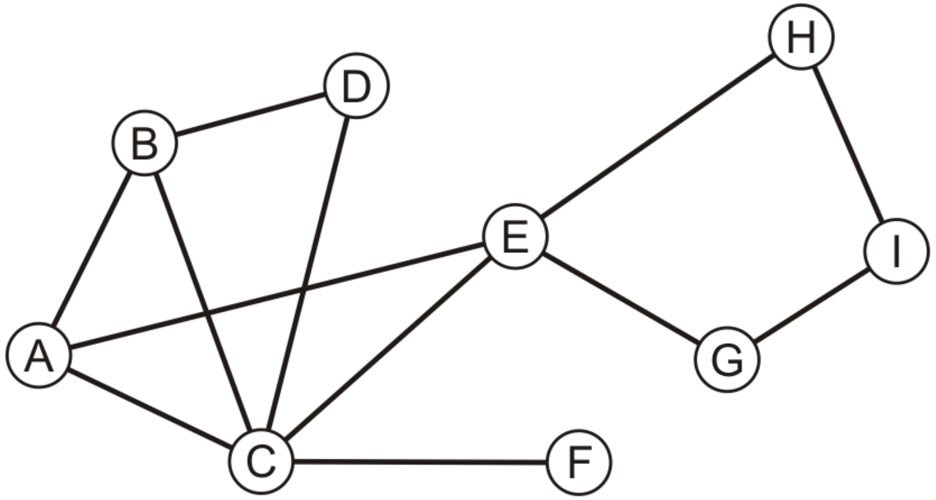
Consider this graph



Example

Performing a breadth-first traversal

- Push the first vertex onto the queue



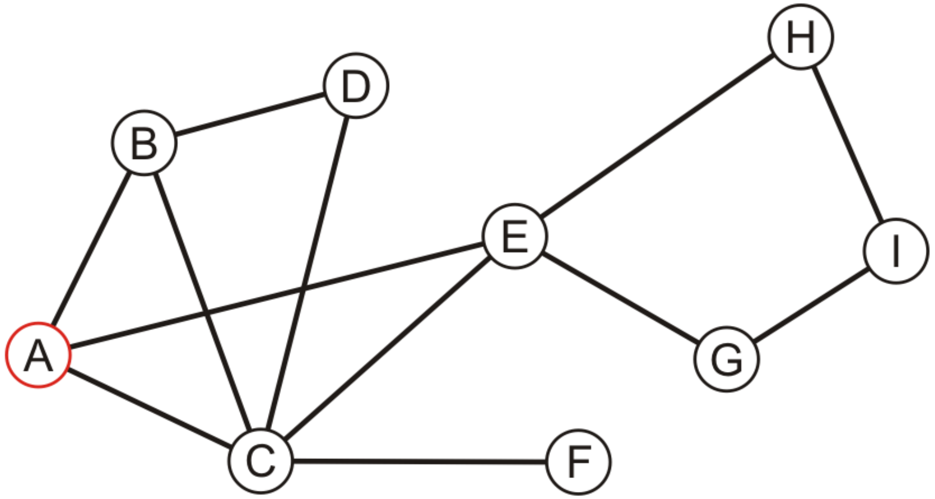
A					

Example

Performing a breadth-first traversal

- Pop A and push B, C and E

A



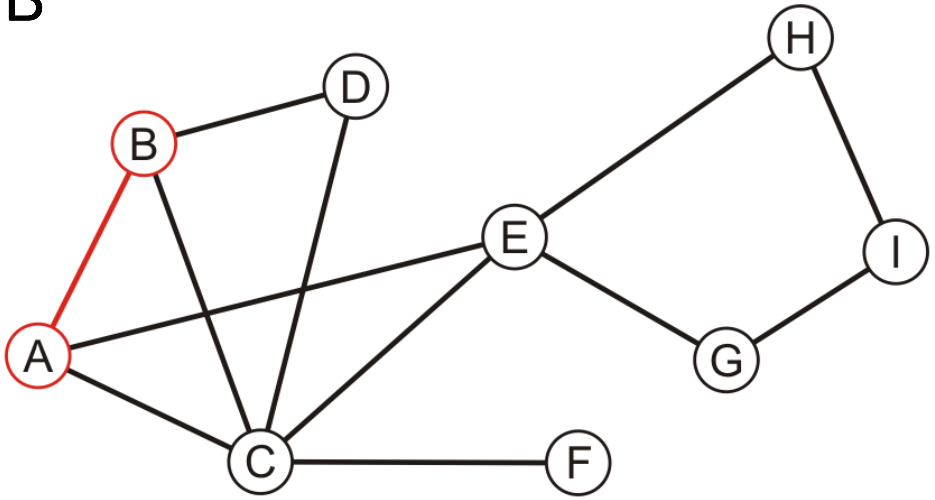
B	C	E			

Example

Performing a breadth-first traversal:

- Pop B and push D

A, B



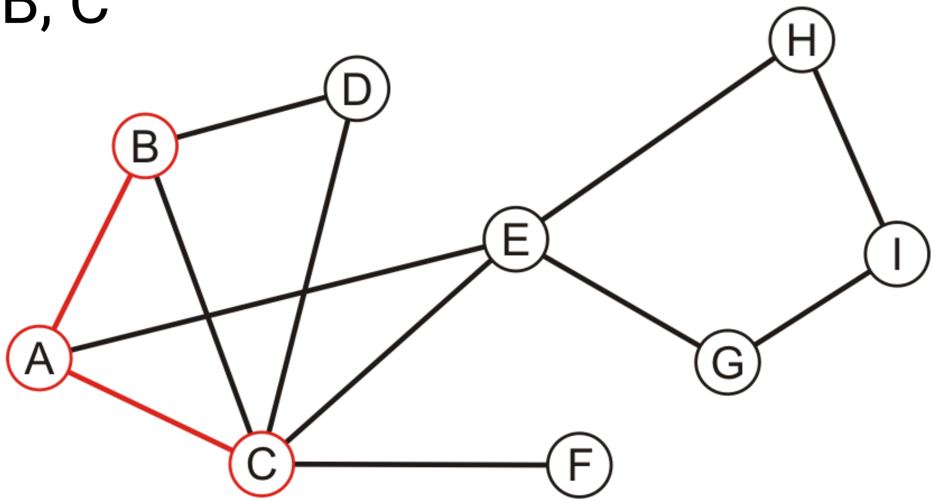
C	E	D			

Example

Performing a breadth-first traversal:

- Pop C and push F

A, B, C



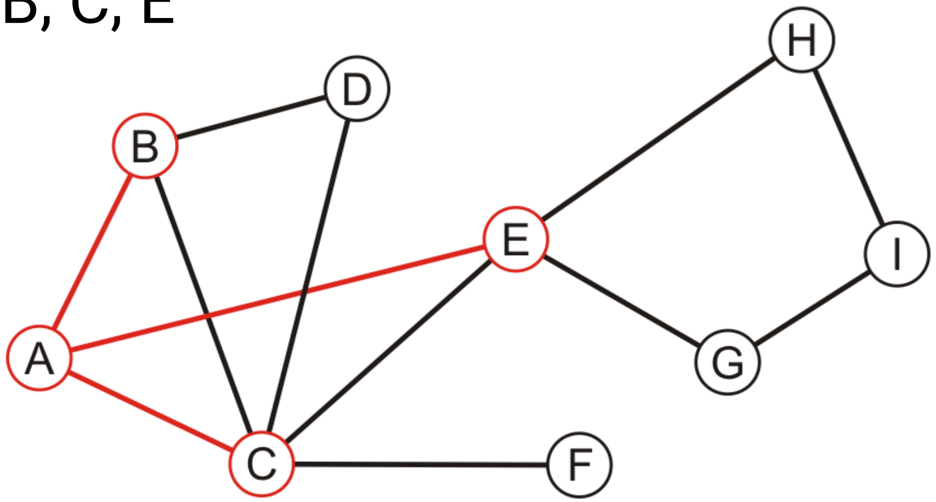
E	D	F			

Example

Performing a breadth-first traversal:

- Pop E and push G and H

A, B, C, E



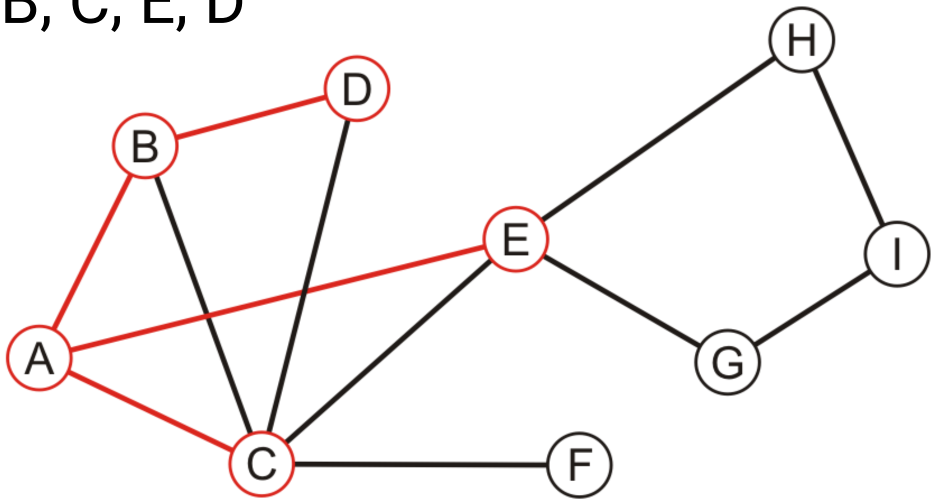
D	F	G	H		

Example

Performing a breadth-first traversal:

- Pop D

A, B, C, E, D



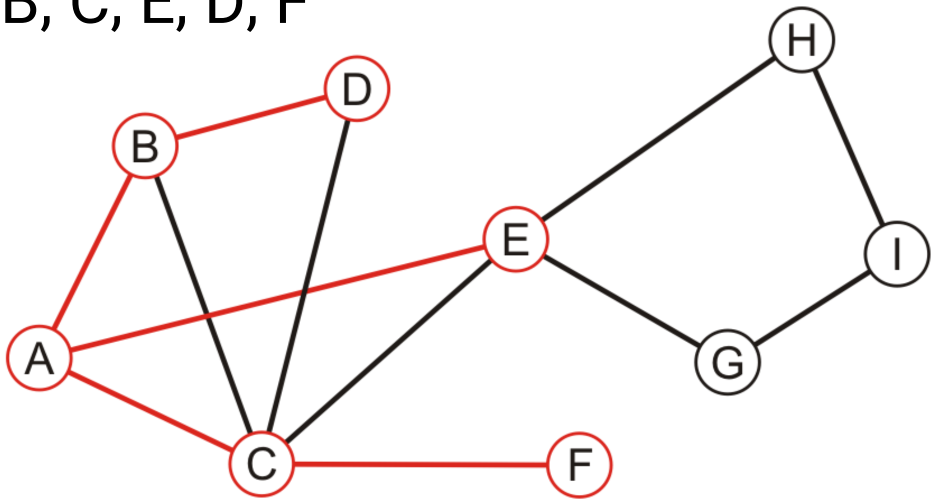
F	G	H			

Example

Performing a breadth-first traversal:

- Pop F

A, B, C, E, D, F



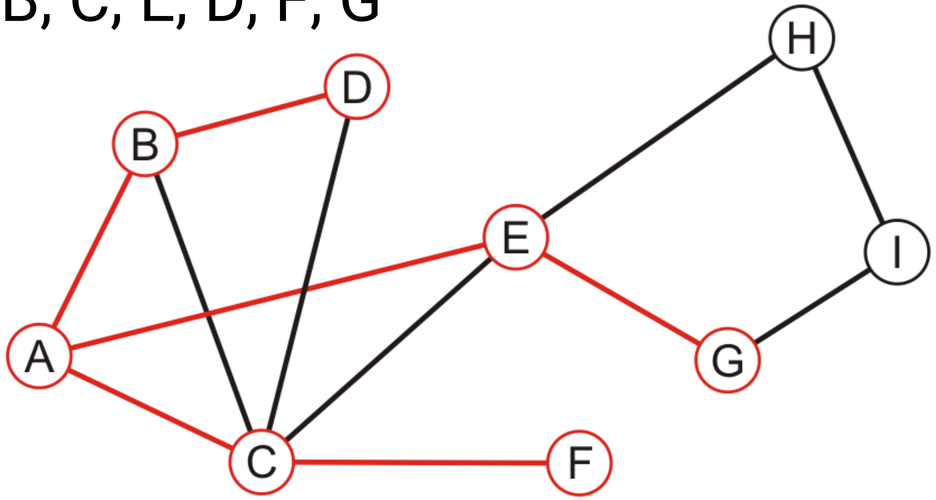
G	H				

Example

Performing a breadth-first traversal:

- Pop G and push I

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



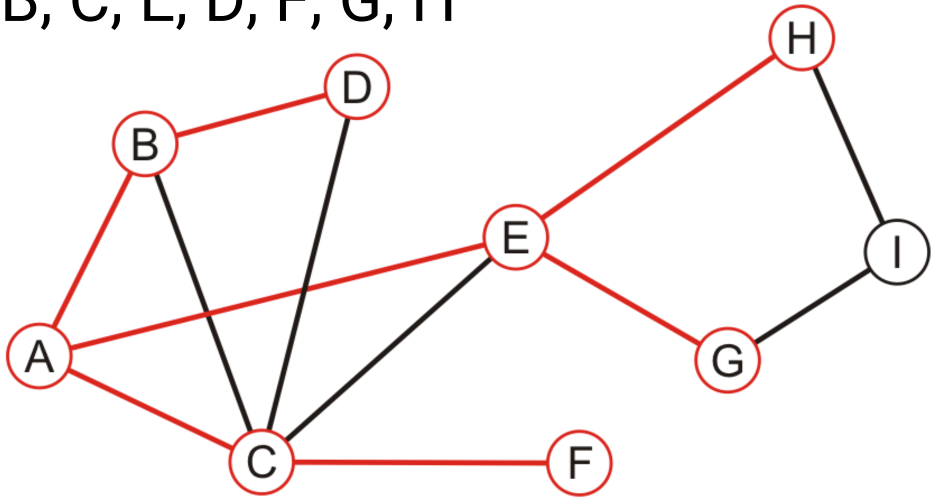
H	I				

Example

Performing a breadth-first traversal:

- Pop H

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



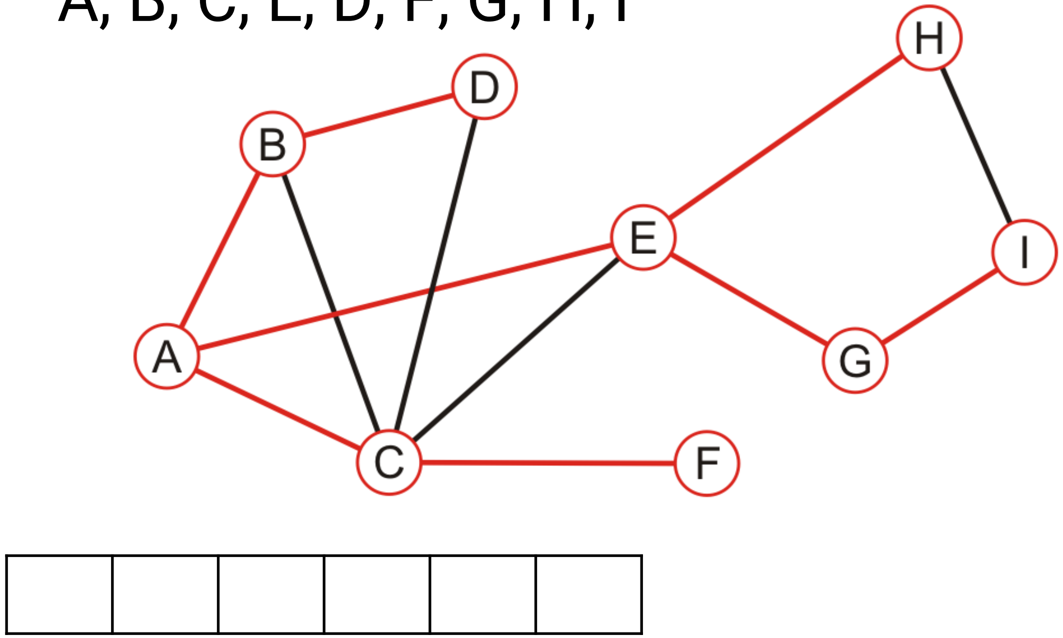
I					

Example

Performing a breadth-first traversal:

- Pop I, The queue is empty: we are finished

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



BFS

Coding practice~

Number of layers in BFS tree: the longest shortest distance

Depth-First Traversal

Consider implementing a 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 last vertex that has not had all of its adjacent vertices visited and continue from there
- Continue until no visited vertices have unvisited adjacent vertices

Two implementations:

- Recursive approach (a statement in a function calls itself repeatedly)
- Iterative approach (a loop repeatedly executes until the controlling condition becomes false)

Recursive depth-first traversal

A recursive implementation uses the call stack for memory:

```
# DFS recursive
visited = [False] * n
def DFS_recursive(u):
    print(u)
    visited[u] = True
    for i in range(offset[u],offset[u+1]):
        nbr_of_u = csr_edges[i]
        if visited[nbr_of_u]: continue
        DFS_recursive(nbr_of_u)
```

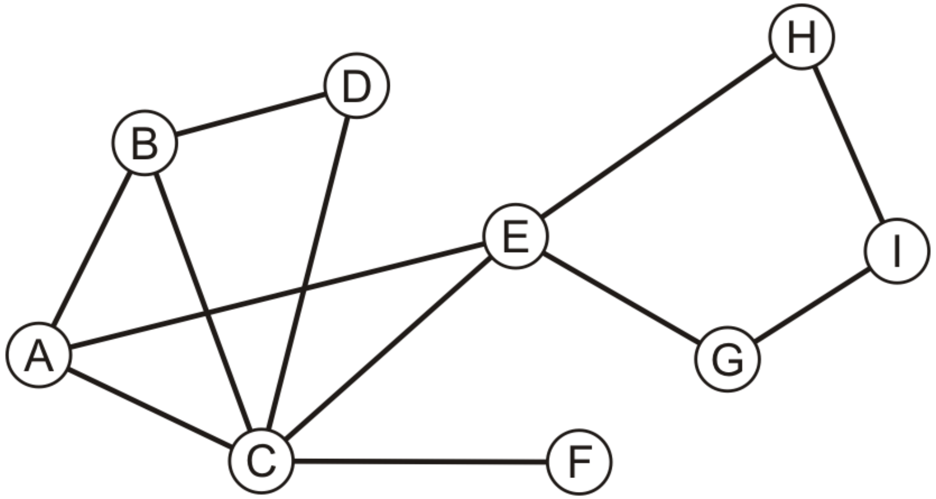
Iterative depth-first traversal

An iterative implementation can use a stack

```python
# DFS iterative
def DFS_iterative(u):
    visited = [False] * n
    stack = []
    stack.append(u)

    while (len(stack)):
        s = stack.pop()
        if(visited[u]):
            continue;

        visited[u] = True
        for i in range(offset[s],offset[s+1]):
            nbr_of_s = csr_edges[i]
            if visited[nbr_of_s]: continue
            stack.append(nbr_of_s)
```

Example

Perform a recursive depth-first traversal on this same graph

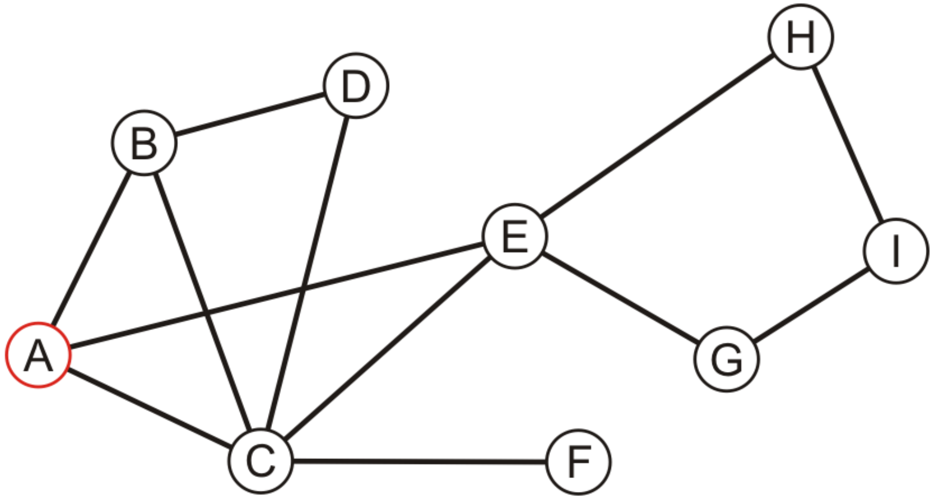


Example

Performing a recursive depth-first traversal:

- Visit the first node

A

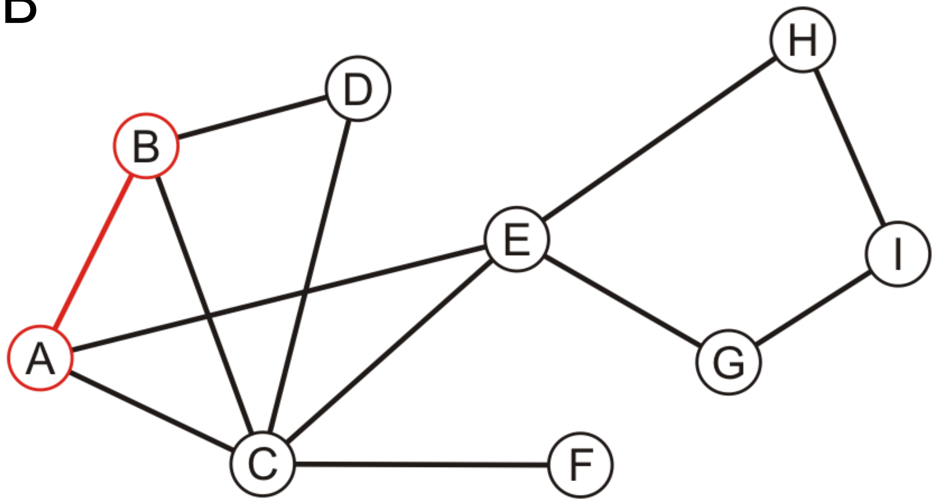


Example

Performing a recursive depth-first traversal:

- A has an unvisited neighbor

A, B

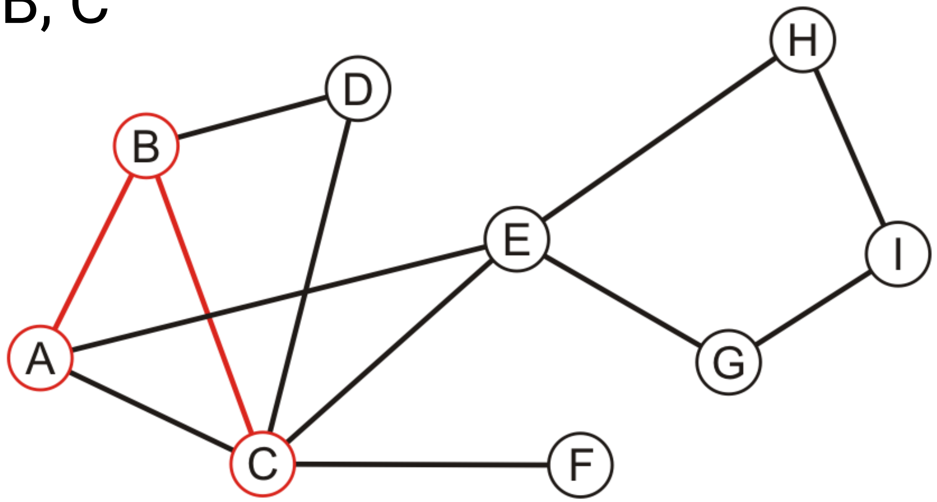


Example

Performing a recursive depth-first traversal:

- B has an unvisited neighbor

A, B, C

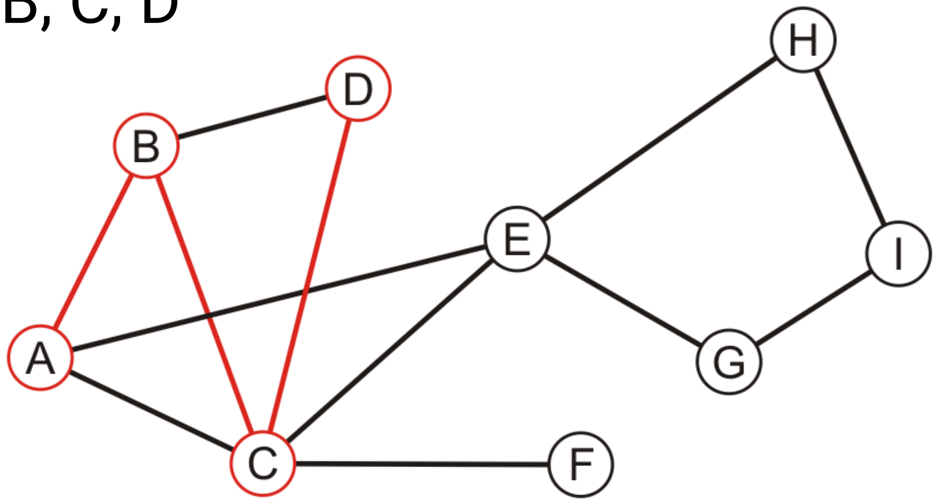


Example

Performing a recursive depth-first traversal:

- C has an unvisited neighbor

A, B, C, D

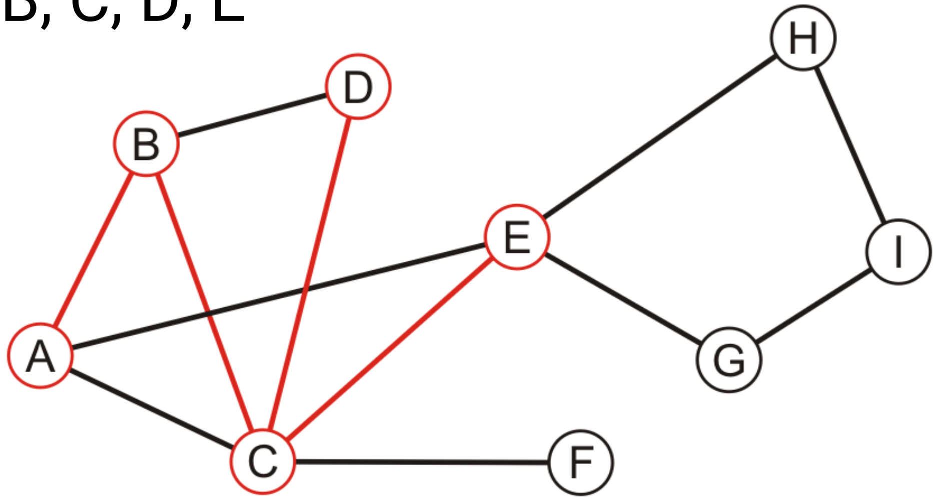


Example

Performing a recursive depth-first traversal:

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

A, B, C, D, E

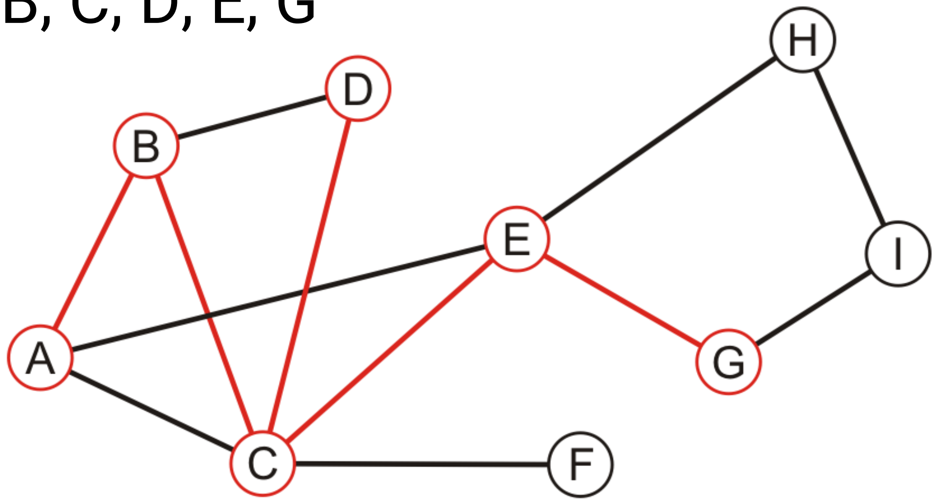


Example

Performing a recursive depth-first traversal:

- E has an unvisited neighbor

A, B, C, D, E, G

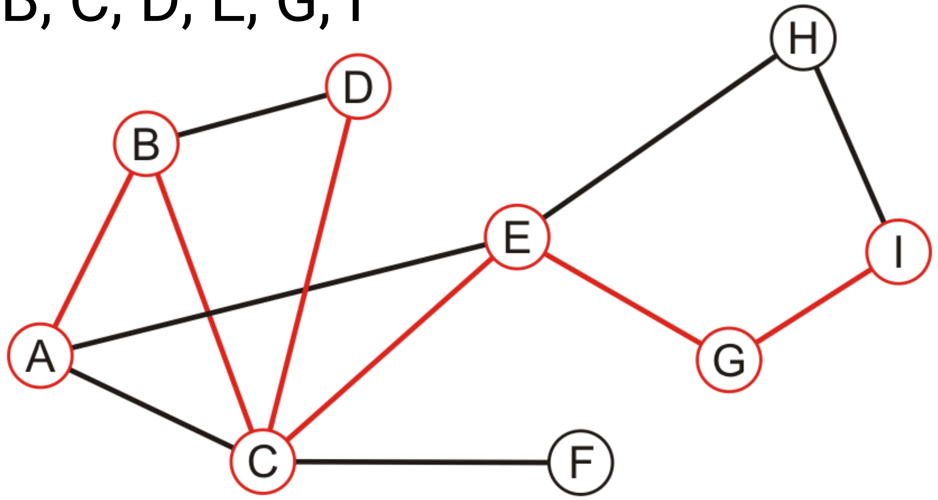


Example

Performing a recursive depth-first traversal:

- G has an unvisited neighbor

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

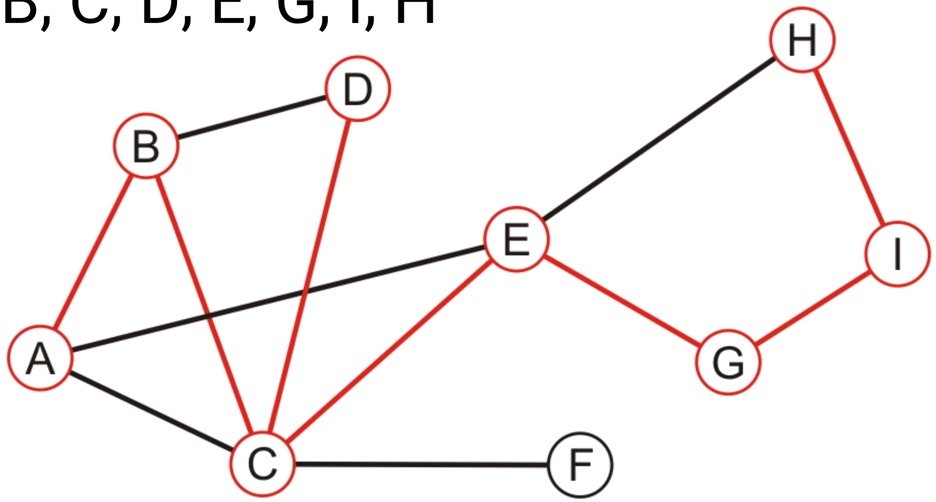


Example

Performing a recursive depth-first traversal:

- I has an unvisited neighbor

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

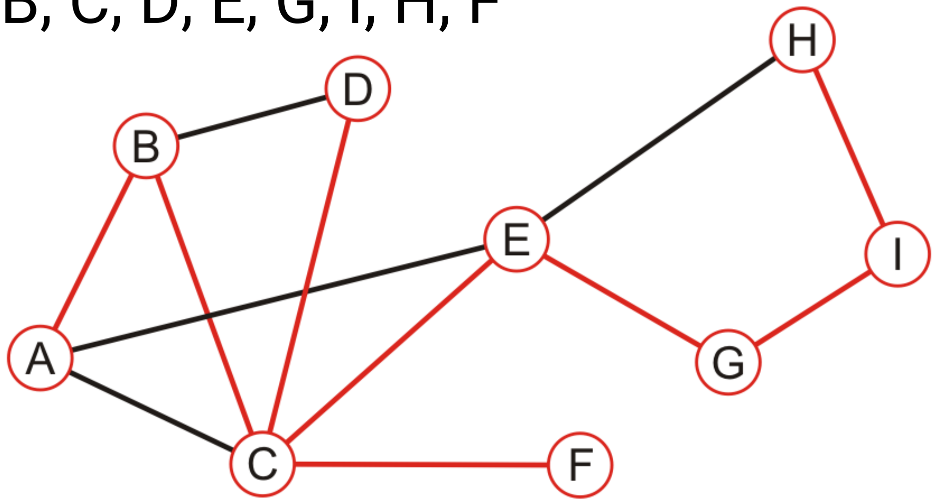


Example

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

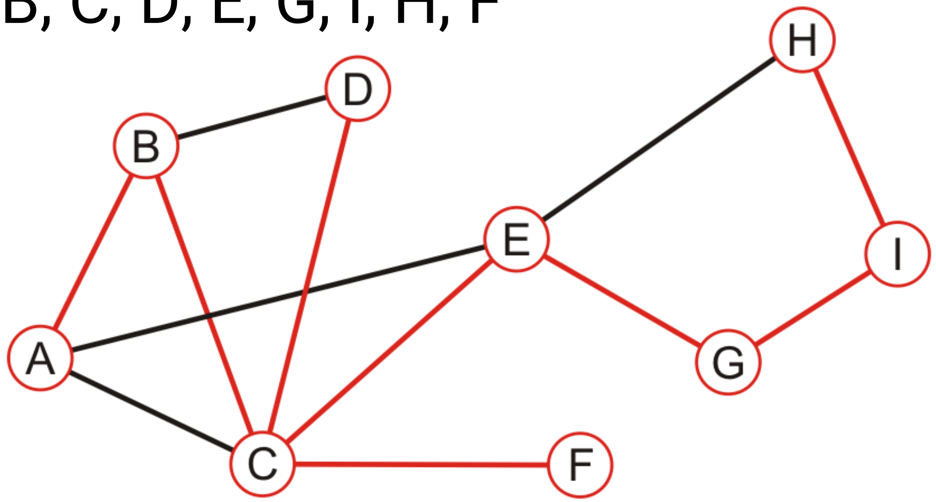


Example

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

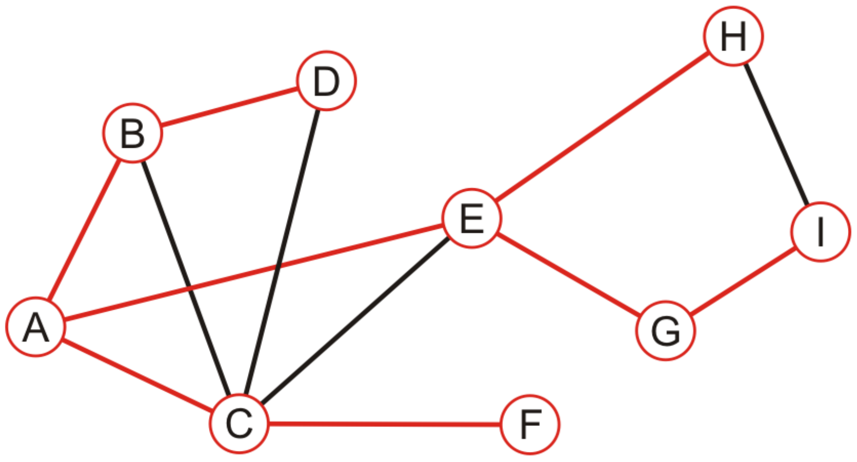


Comparing BFS and DFS

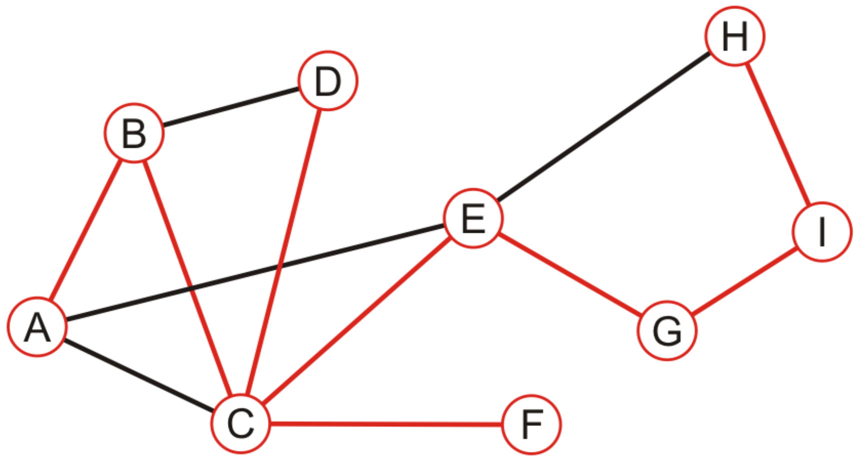
The order can differ greatly

- An iterative depth-first traversal may also be different again

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

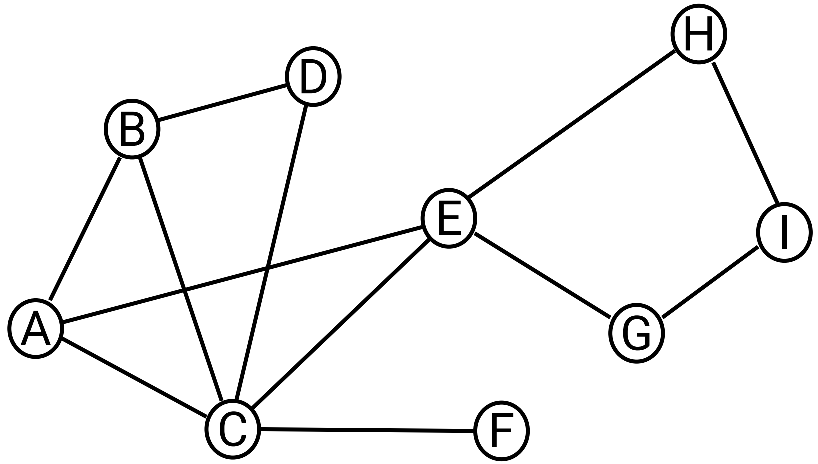


Recursive DFS: A, B, C, D, E, G, I, H, F



Quick Quiz

Can you show the result of iterative depth-first traversal?

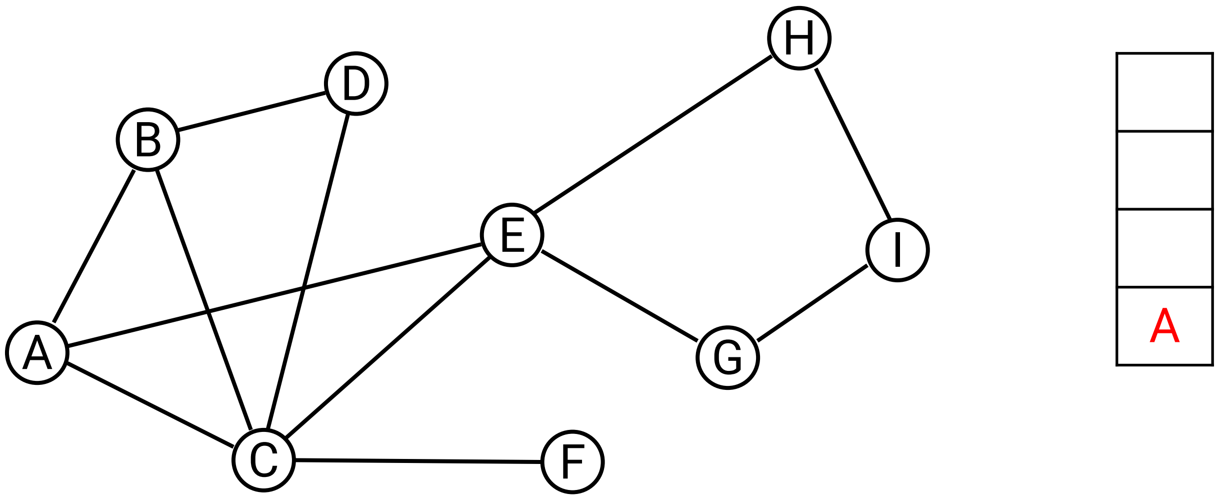



```
# DFS iterative
def DFS_iterative(u):
    visited = [False] * n
    stack = []
    stack.append(u)

    while (len(stack)):
        s = stack.pop()
        if(visited[u]):
            continue;

        visited[u] = True
        for i in range(offset[s],offset[s+1]):
            nbr_of_s = csr_edges[i]
            if visited[nbr_of_s]: continue
            stack.append(nbr_of_s)
```

Example

Performing an iterative depth-first traversal:

- Push the first vertex onto the stack

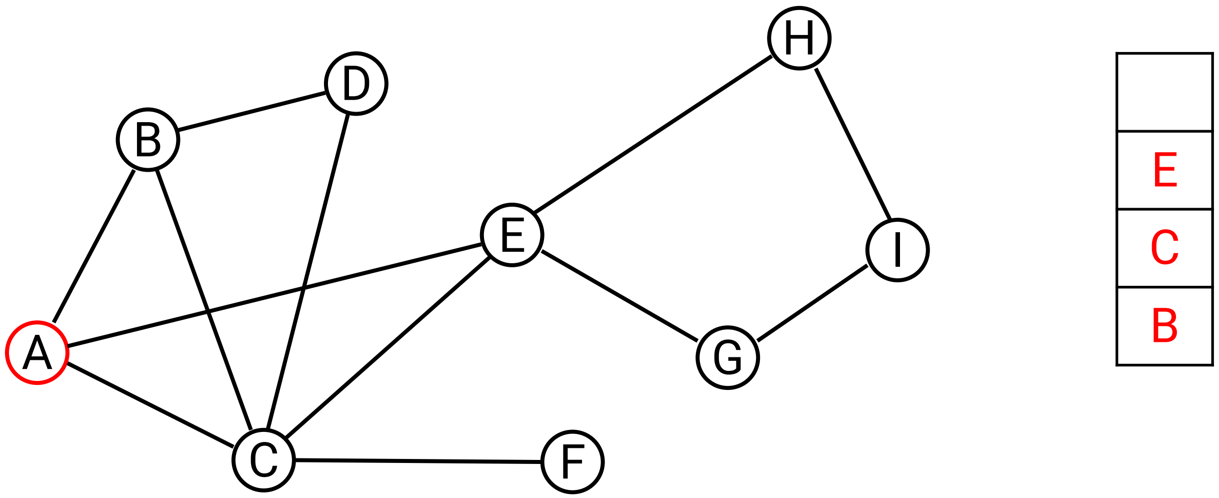


Example

Performing an iterative depth-first traversal:

- Pop A and push B, C and E

A

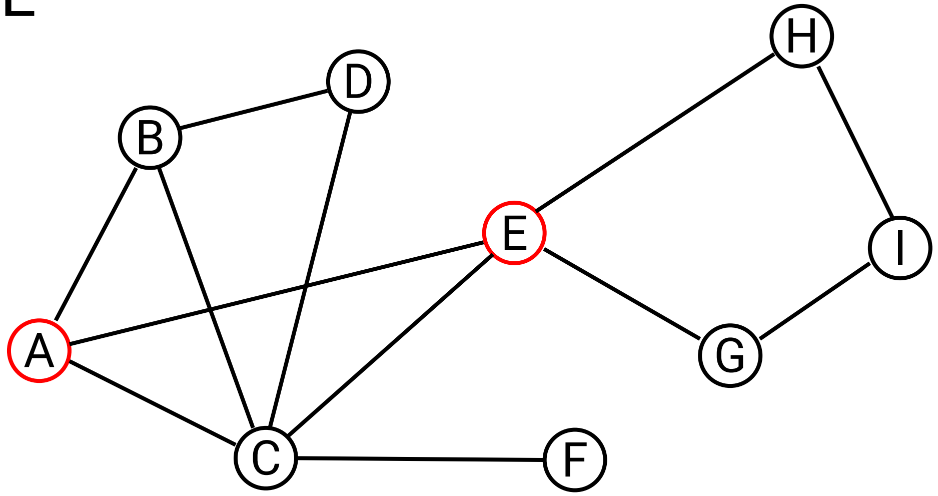


Example

Performing an iterative depth-first traversal:

- Pop E and push C, G, and H

A, E



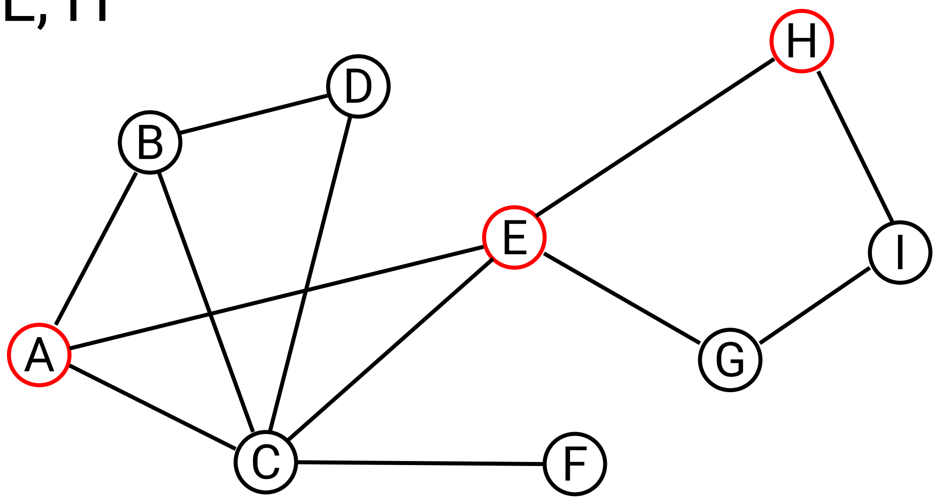
H
G
C
C
B

Example

Performing an iterative depth-first traversal:

- Pop H, and push I

A, E, H



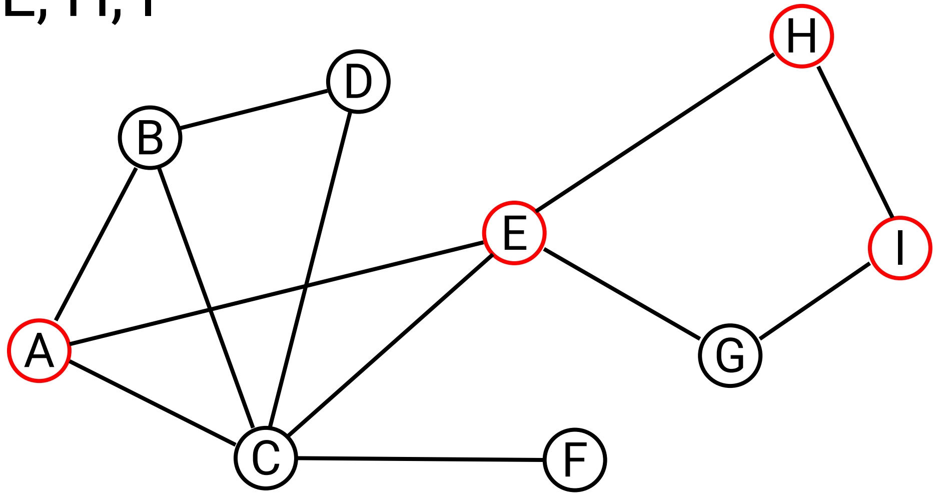
I
G
C
C
B

Example

Performing an iterative depth-first traversal:

- Pop I and push G

A, E, H, I



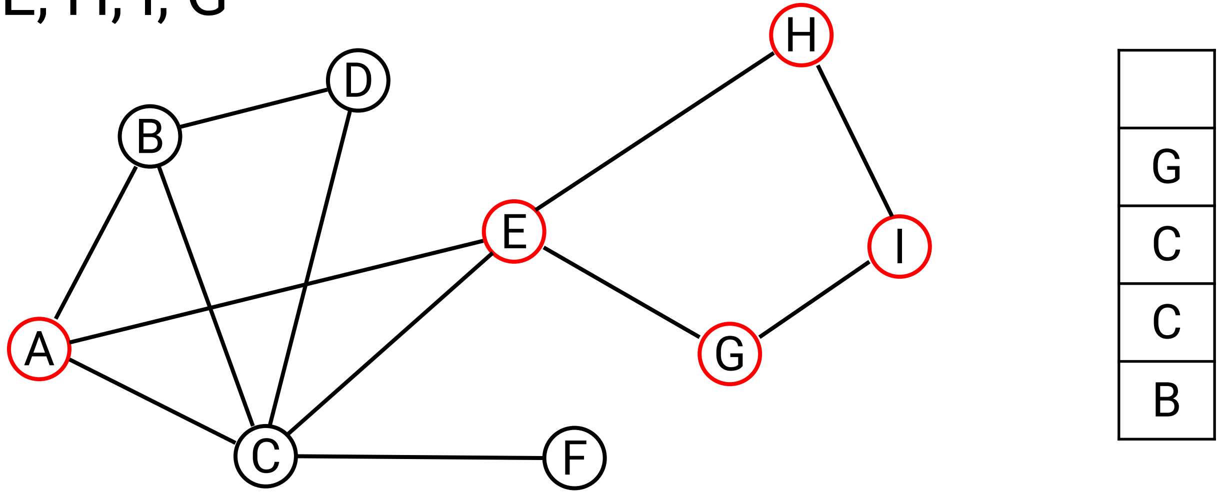
G
G
C
C
B

Example

Performing an iterative depth-first traversal:

- Pop G

A, E, H, I, G

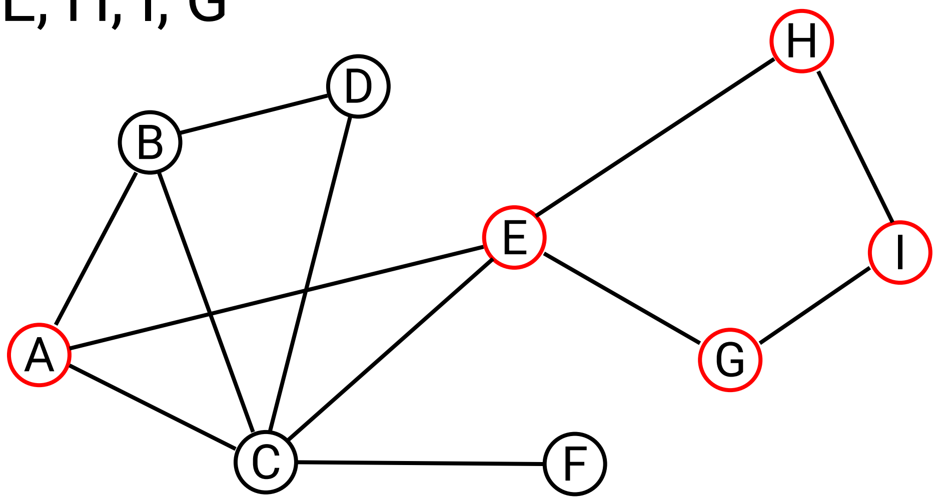


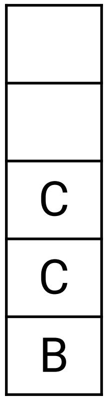
Example

Performing an iterative depth-first traversal:

- Pop G again, and skip G since it is visited

A, E, H, I, G



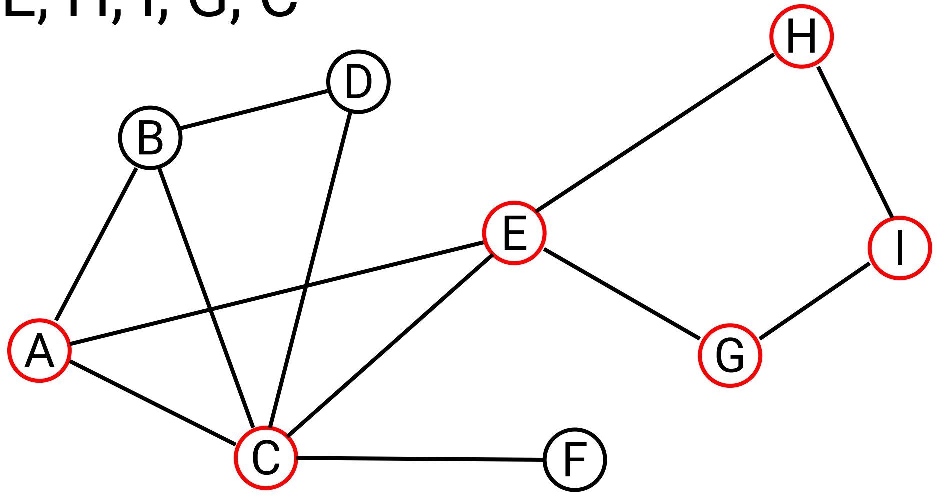


Example

Performing an iterative depth-first traversal:

- Pop C, and add B, D, F

A, E, H, I, G, C



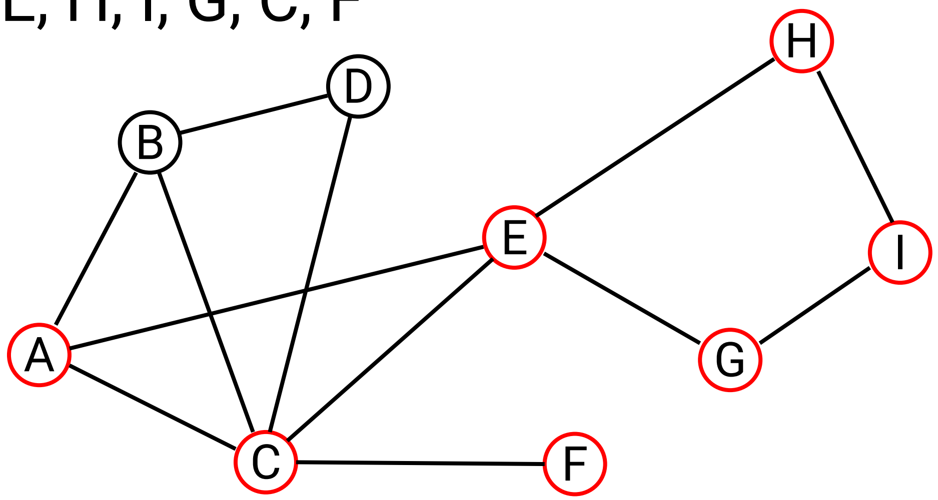
F
D
B
C
B

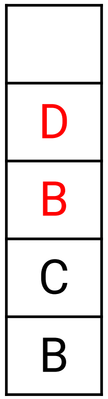
Example

Performing an iterative depth-first traversal:

- Pop F

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



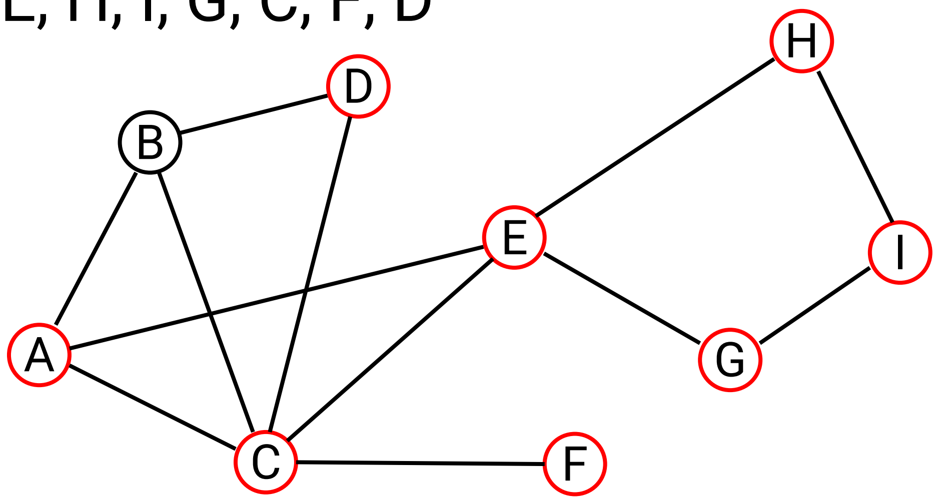


Example

Performing an iterative depth-first traversal:

- Pop D and add B

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



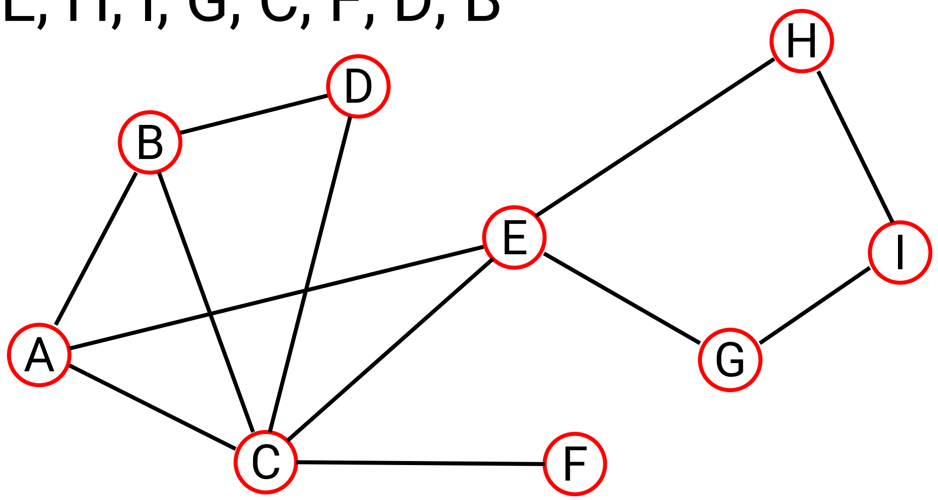
B
B
C
B

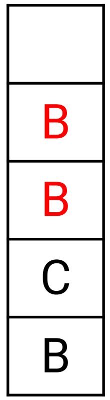
Example

Performing an iterative depth-first traversal:

- Pop B

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





Pop and skip all remining vertices in the stack since they are already visited

Complexity Analysis

We have to track which vertices have been visited requiring $O(|V|)$ memory

The time complexity cannot be better than and should not be worse than $O(|V| + |E|)$

Connected graphs simplify this to $O(|E|)$ – Why?

DFS: Recursive VS stack-based

Which one is better?

Coding practice~

Summary

This topic covered graph traversals

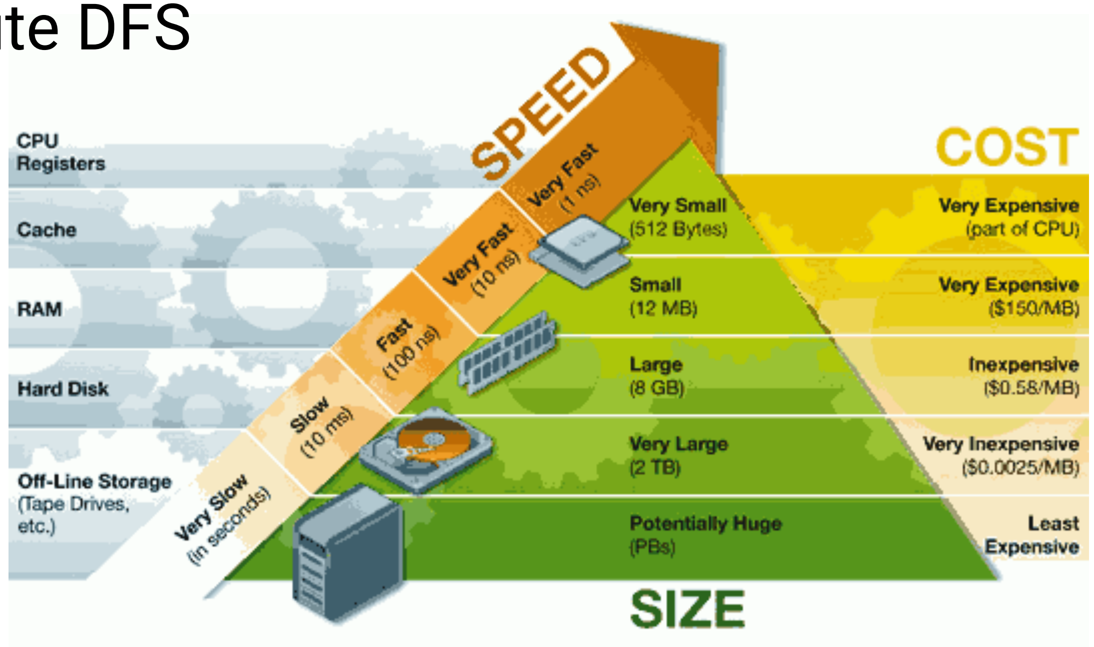
- Considered breadth-first and depth-first traversals
- Depth-first traversals can recursive or iterative
- Considered an example with both implementations
- They are also called *searches*

Recent Research on DFS/BFS

Optional

External Memory Algorithms

If there is no enough memory to store the whole graph, how to compute DFS



https://dl.acm.org/doi/10.1145/2723372.2723740

Recent Research on DFS/BFS

Optional

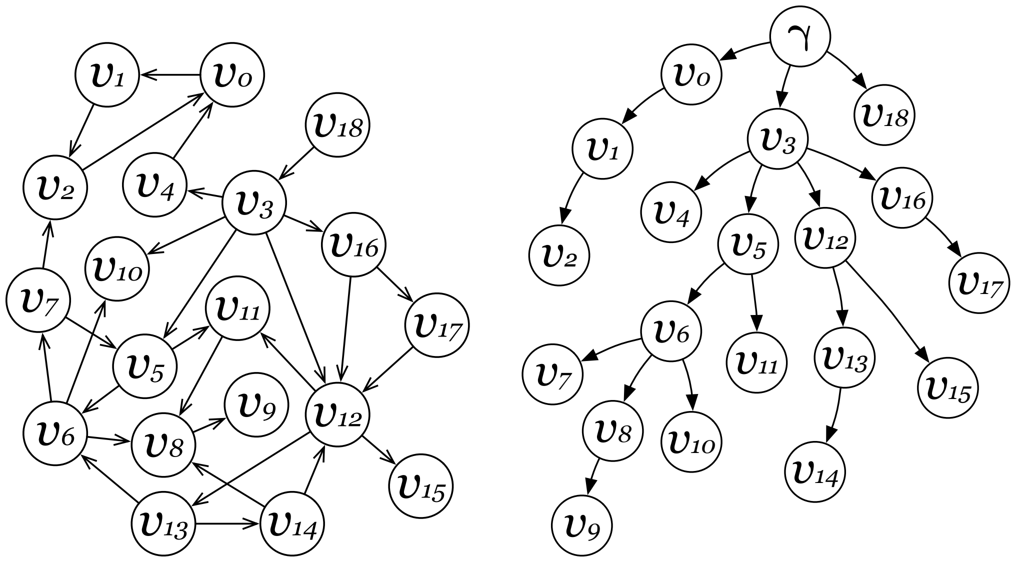
Dynamic Graphs

When graph updates (new edge inserts or old edge removes)

Compute DFS from scratch

VS

Update DFS tree



(a) The graph G (b) A DFS-Tree $\mathcal{T}$ of G

https://dl.acm.org/doi/10.14778/3364324.3364329

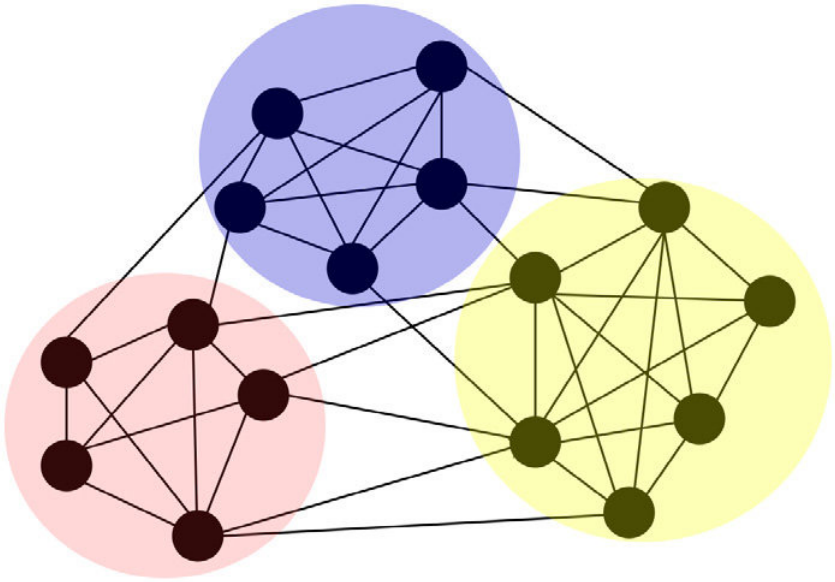
Recent Research on DFS/BFS

Optional

Distributed Algorithms

The information (neighbors) of different vertices locate in different machines.

Distributed DFS algorithm is hard.



Connectivity

Connectivity

We will use graph traversals to determine:

- Whether one vertex is connected to another
- The connected sub-graphs of a graph

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

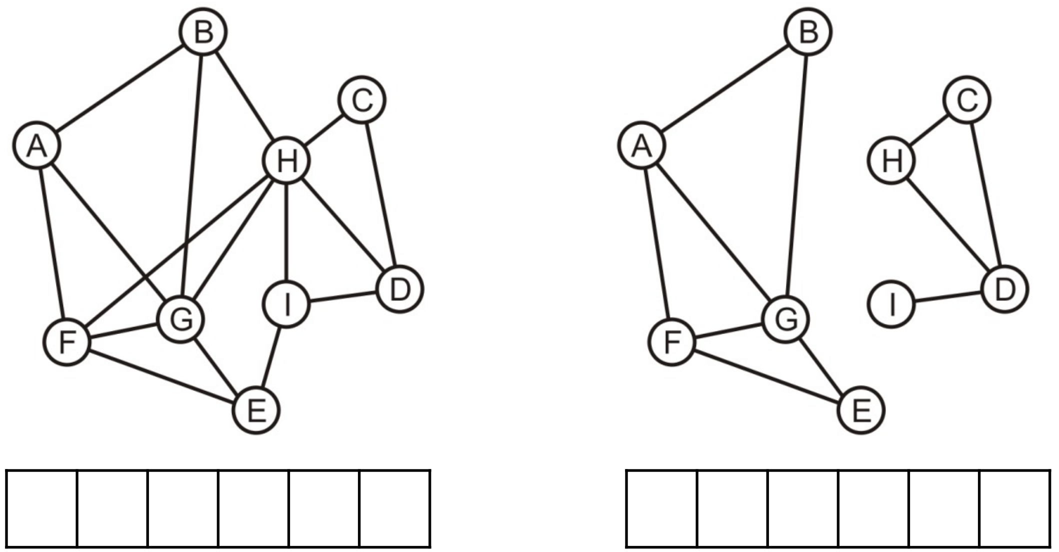
- v_j is connected to v_k if there is a path from v_j to v_k

Strategy:

- Perform a breadth-first traversal starting at v_j
- While looping, if the vertex v_k ever found to be adjacent to the front of the queue, return true
- If the loop ends, return false

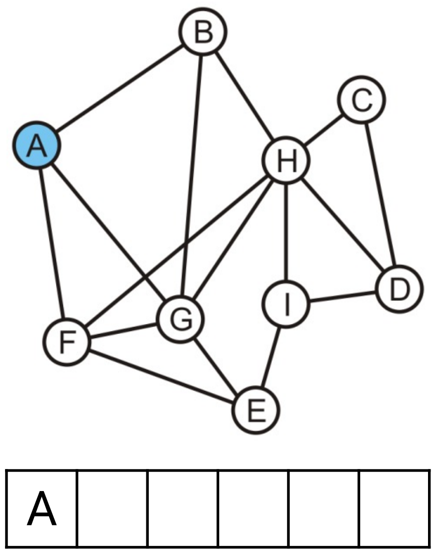
Determining Connections

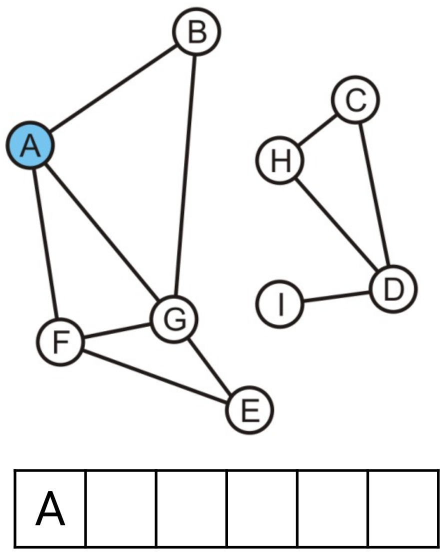
Is A connected to D?



Determining Connections

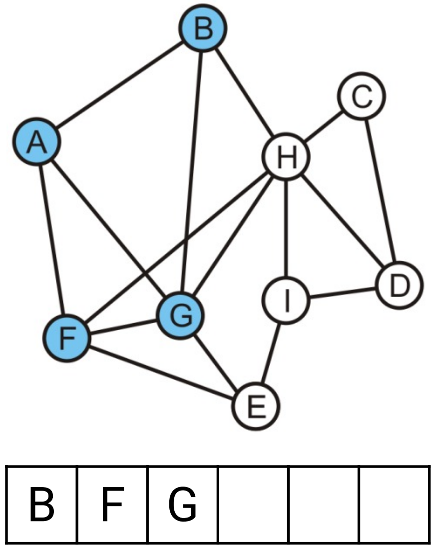
Vertex A is marked as visited and pushed onto the queue

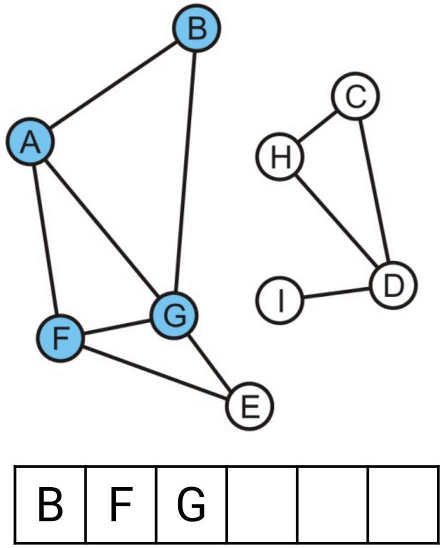




Determining Connections

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

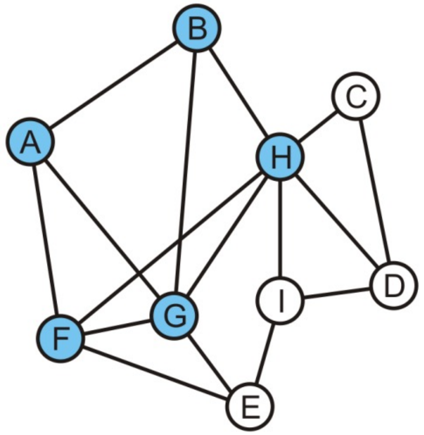




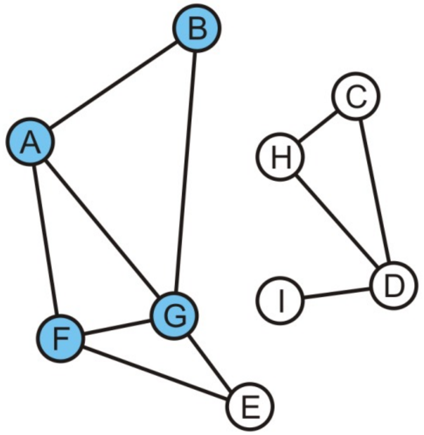
Determining Connections

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

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



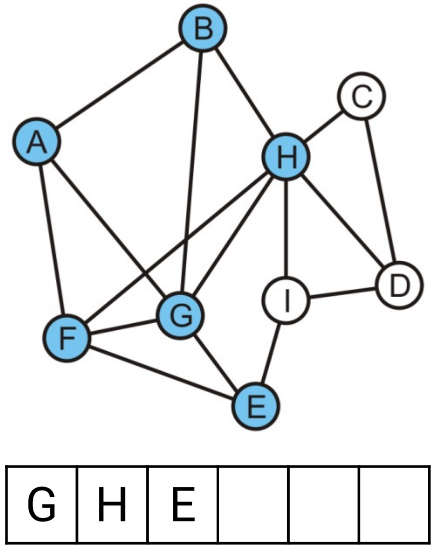
F	G	H			

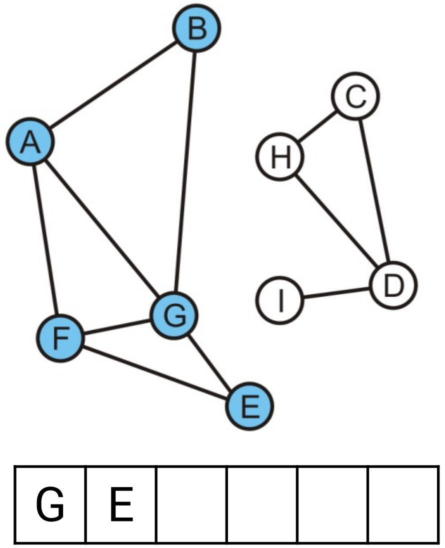


F	G				

Determining Connections

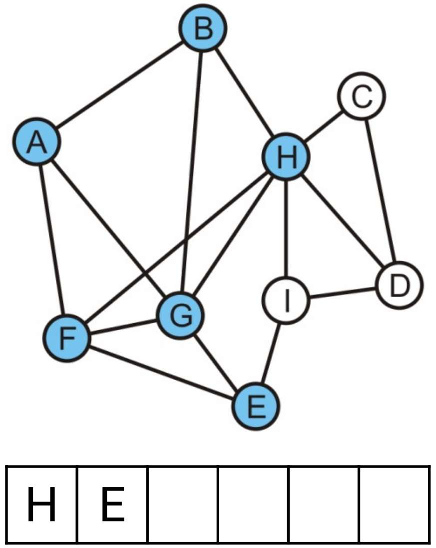
Popping F results in the pushing of E

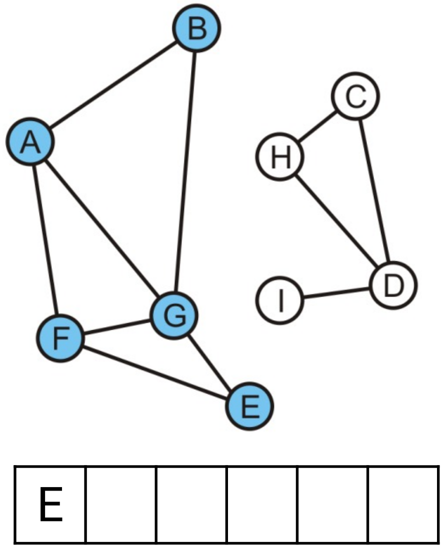




Determining Connections

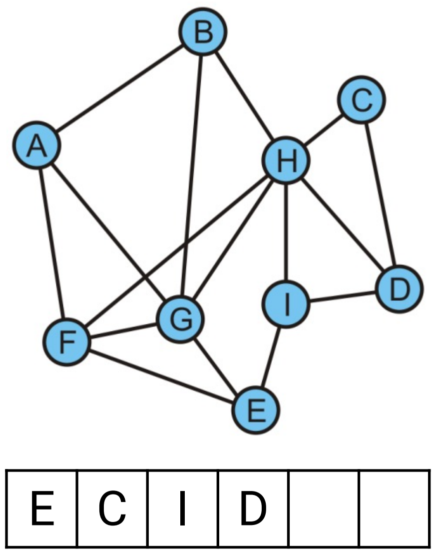
In either graph, G has no adjacent vertices that are unvisited

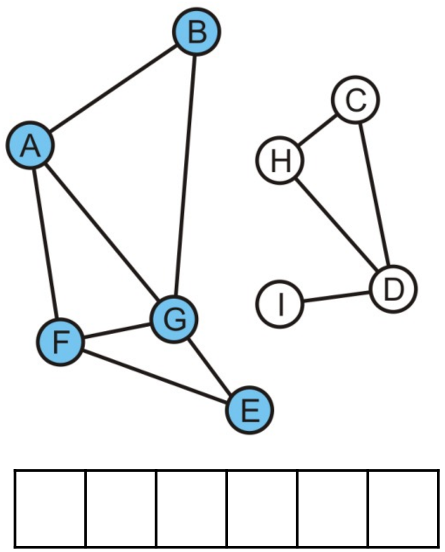




Determining Connections

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





In the left graph, A is connected to D, since D is in the queue

The queue on the right is empty. We determine A is not connected to D

Connectivity

Coding practice~

Any better idea?

- Bidirectional search

Connected Components

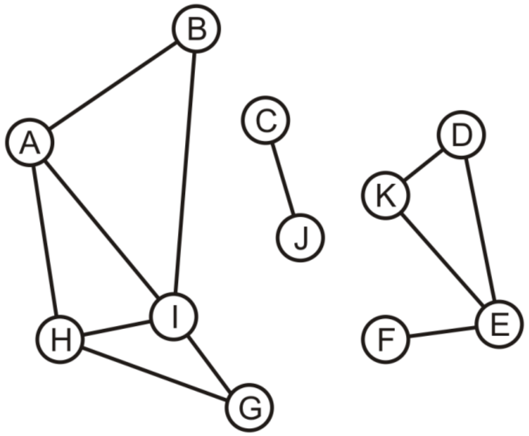
If we continued the traversal, we would find all vertices that are connected to A

Suppose we want to find the connected components of the graph

- While there are unvisited vertices:
    - 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

Connected Components

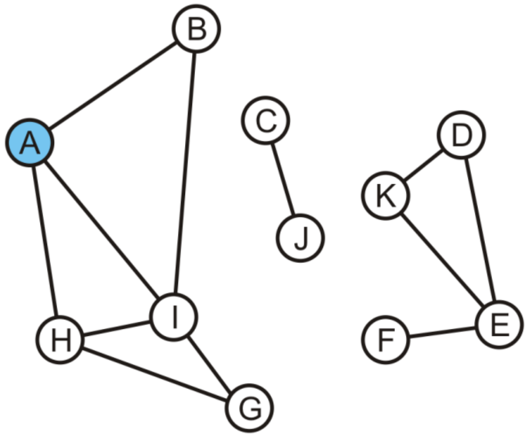
Here we start with a set of singletons



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

Connected Components

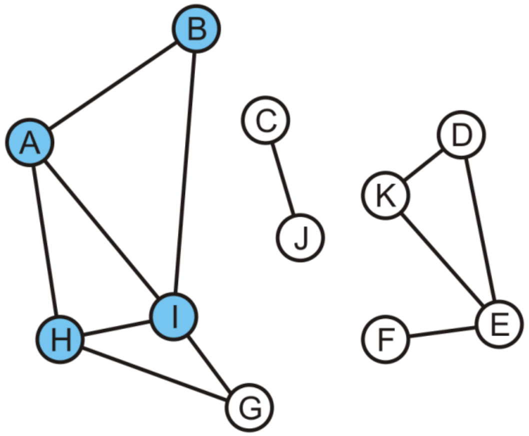
The vertex A is unvisited, so we start with it



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}

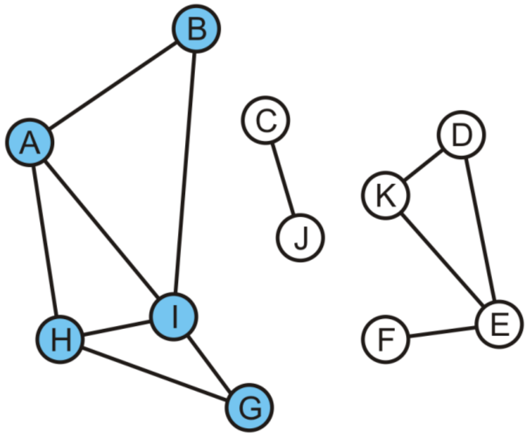


A	B	C	D	E	F	G	H	I	J	K
A	**A**	**C**	**D**	**E**	**F**	**G**	**A**	**A**	**J**	**K**

Connected Components

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

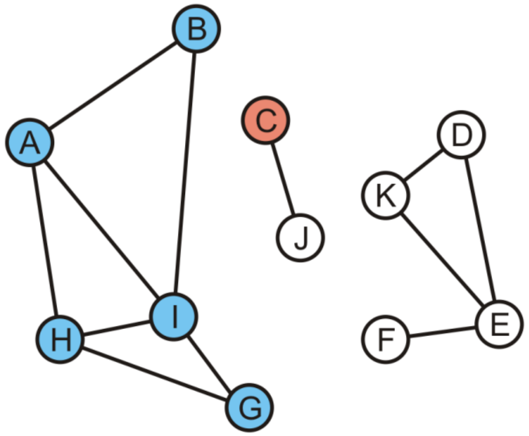
- The traversal is finished



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}

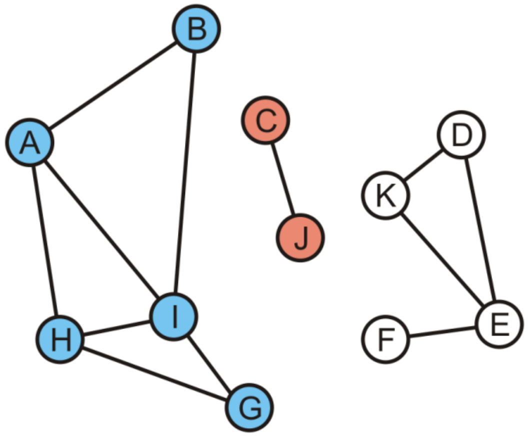


A	B	C	D	E	F	G	H	I	J	K
A	**A**	**C**	**D**	**E**	**F**	**A**	**A**	**A**	**J**	**K**

Connected Components

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

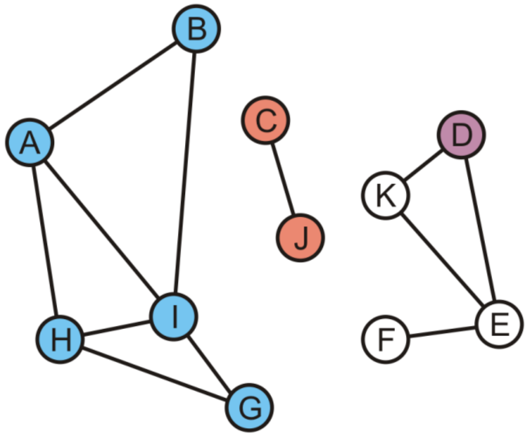
- This traversal is finished



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

Connected Components

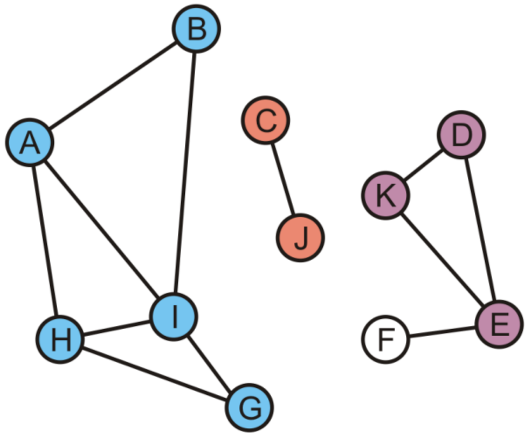
We start again with the set {D}



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}

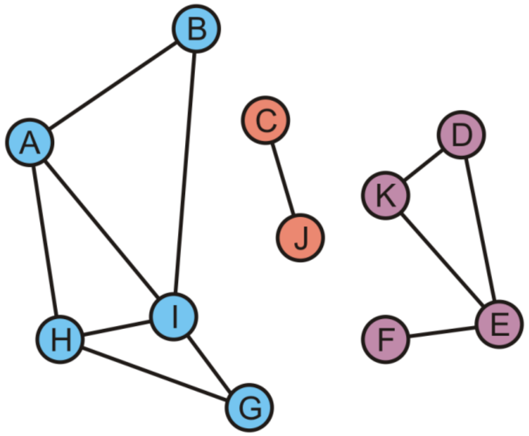


A	B	C	D	E	F	G	H	I	J	K
A	**A**	**C**	**D**	**D**	**F**	**A**	**A**	**A**	**C**	**D**

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}

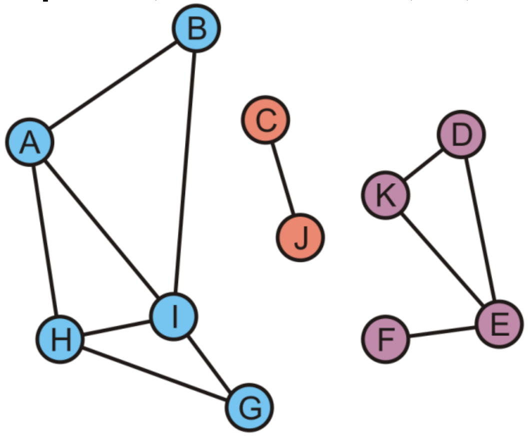


A	B	C	D	E	F	G	H	I	J	K
A	A	C	D	D	D	A	A	A	C	D

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

The time complexity to find an unvisited vertex: $O(|V|)$

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

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

The solution will use $O(|V|)$ additional memory

Coding practice~

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

Tracking Unvisited Vertices

Suppose we visit D

- D is in entry 3

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

Tracking Unvisited Vertices

The actual algorithm is exceptionally fast:

- The initialization is $\mathrm{O}(|V|)$
- Determining if the vertex v_k is visited is fast: $\mathrm{O}(1)$
- Marking vertex v_k as having been visited is also fast: $\mathrm{O}(1)$
- Returning a vertex that is unvisited is also fast: $\mathrm{O}(1)$

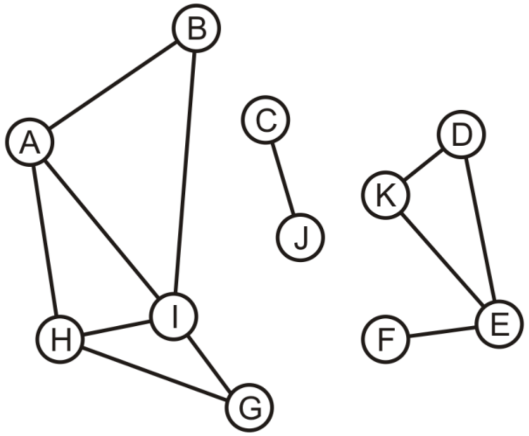
- The idea/structure is for any scenario that needs to remove an item from a list (without any order limitation).
- The other option: doubly linked list

Compute connected components with new data structure

We start with two arrays

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

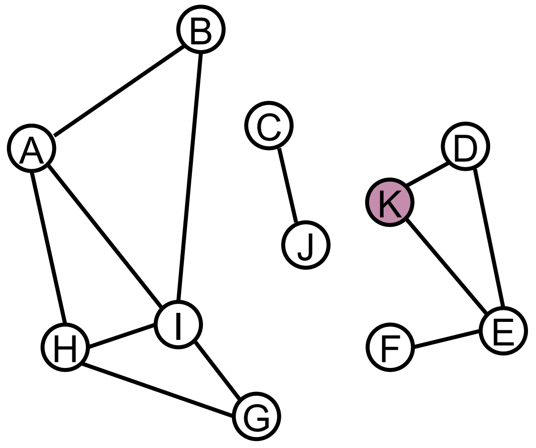


Compute connected components with new data structure

The first unvisited vertex is K
– Remove K

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

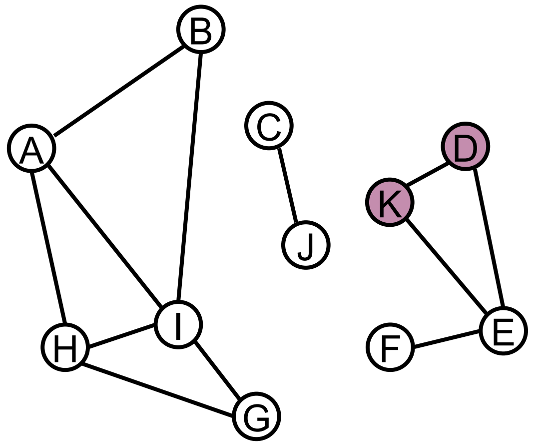


Compute connected components with new data structure

– Visit D through the edge (K, D)
– Copy J into location 3 and update the location array

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

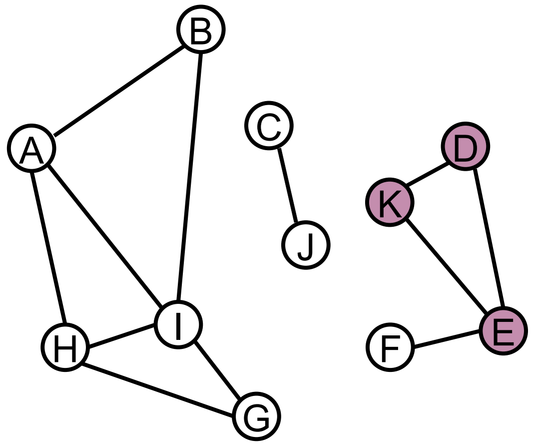


Compute connected components with new data structure

- Visit E through the edge (K, E)
- Copy I into location 4 and update the location array

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

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

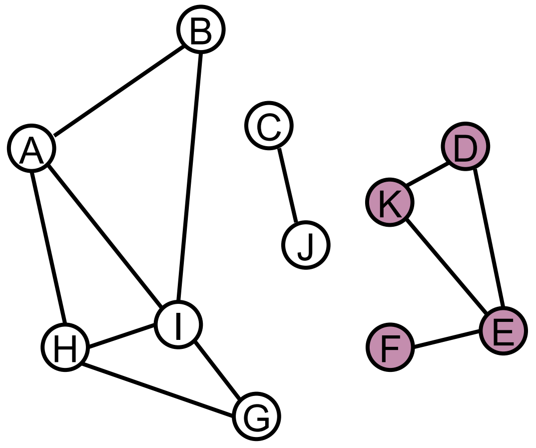


Compute connected components with new data structure

- Visit F through the edge (E, F)
- Copy H into location 5 and update the location array

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

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

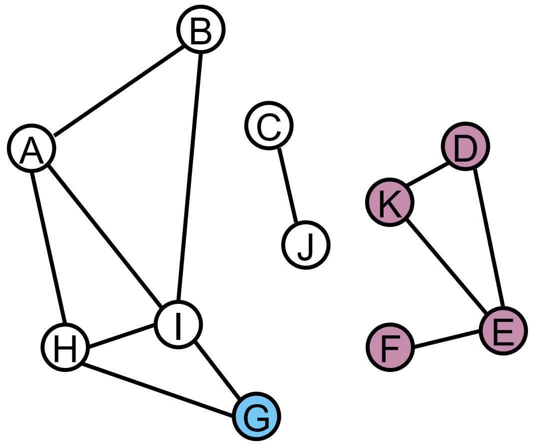


Compute connected components with new data structure

- BFS Queue is empty, one component {D, E, F, K} is found.
- Then, we visit G
- Remove G

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

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

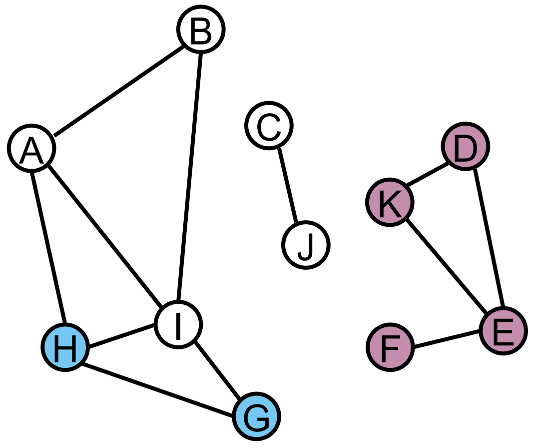


Compute connected components with new data structure

- Visit H through (G, H)
- Remove H

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

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

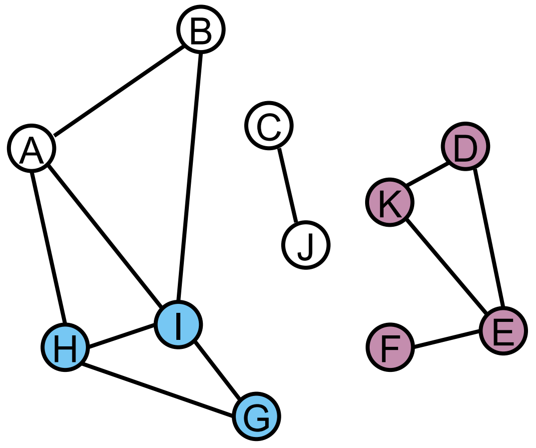


Compute connected components with new data structure

- Visit I
- Remove I

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

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

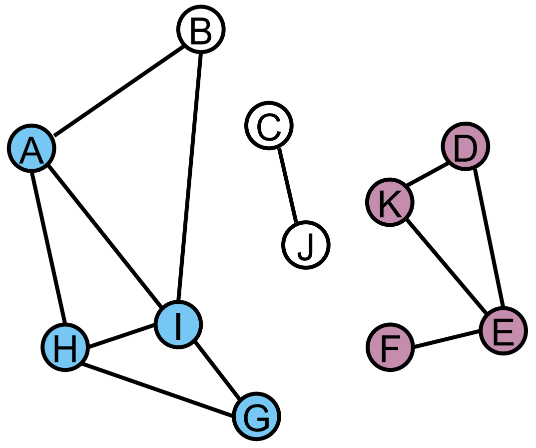


Compute connected components with new data structure

– Visit A
– Copy J into location 0 and update the location array

0	1	2	3	4	5	6	7	8	9	10
J	B	C	J							

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

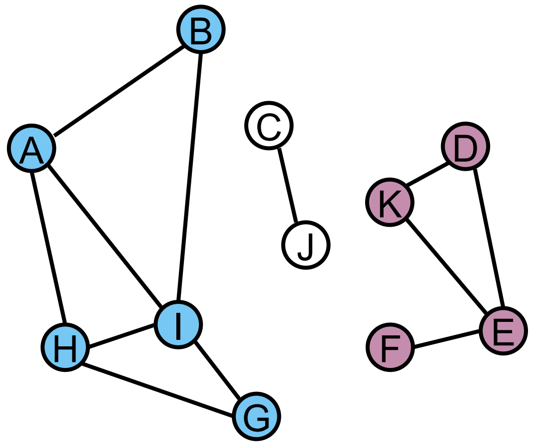


Compute connected components with new data structure

- Visit B
- Copy C into location 1 and update the location array

0	1	2	3	4	5	6	7	8	9	10
J	C	C								

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

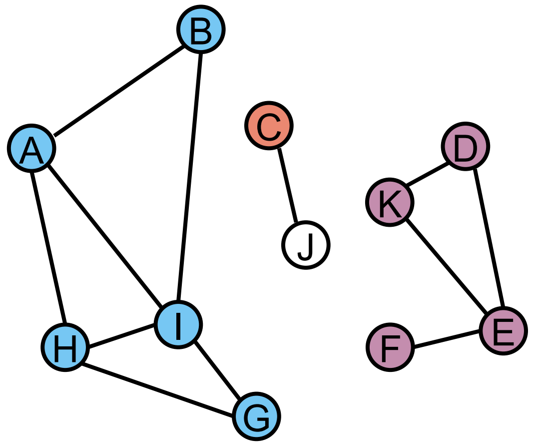


Compute connected components with new data structure

– Visit C
– Remove C

0	1	2	3	4	5	6	7	8	9	10
J	C									

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

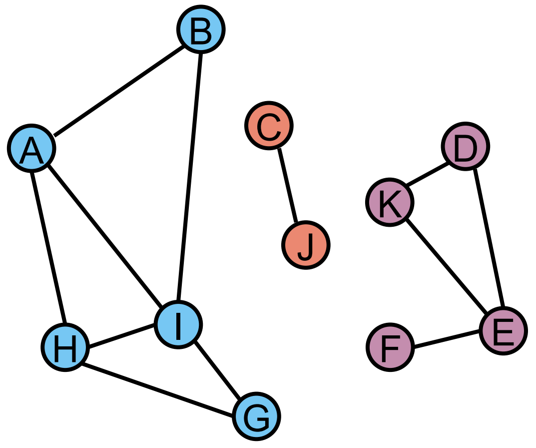


Compute connected components with new data structure

- Visit J
- Remove J

0	1	2	3	4	5	6	7	8	9	10

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

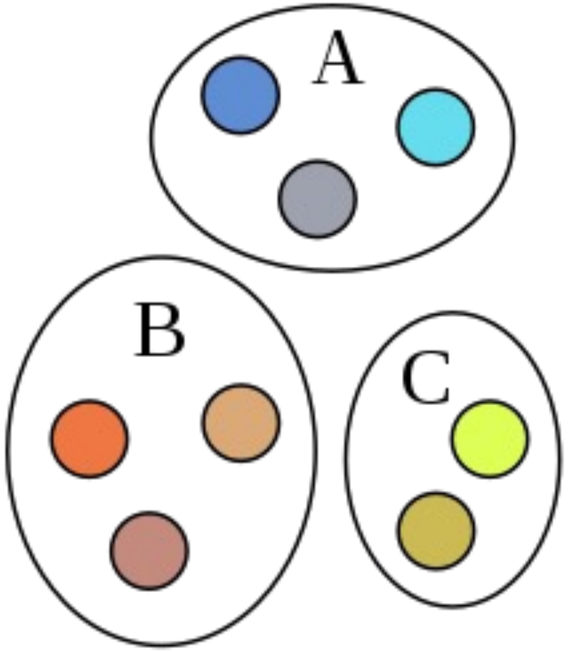


Connected Component Detection

Coding practice~

Any other easier way to implement?

Disjoint set data structure



- Consider n elements, named 1, 2, …, n
- The disjoint set is a collection of sets of elements
- Each element is in exactly one set
  - sets are disjoint
  - to start, each set contains one element
- SetName = find (elementName)
  - returns the name of the set that contains the given element
- union (SetName1, SetName2)
  - union two sets together into a new set

How to quickly perform **union** and **find** operations?

Disjoint set data structure

Attempt 1: Quick Find

- Array implementation. elements are 1, …, N
- SetName[i] = name of the set containing element i
- Pseudo code:

```
Initialize(int N)
  SetName = new int [N+1];
  for (int e=1; e<=N; e++)
    SetName[e] = e;

Union(int i, int j)
  for (int k=1; k<=N; k++)
    if (SetName[k] == j)
      SetName[k] = i;

int Find(int e)
  return SetName[e];
```

Time Complexity Analysis:

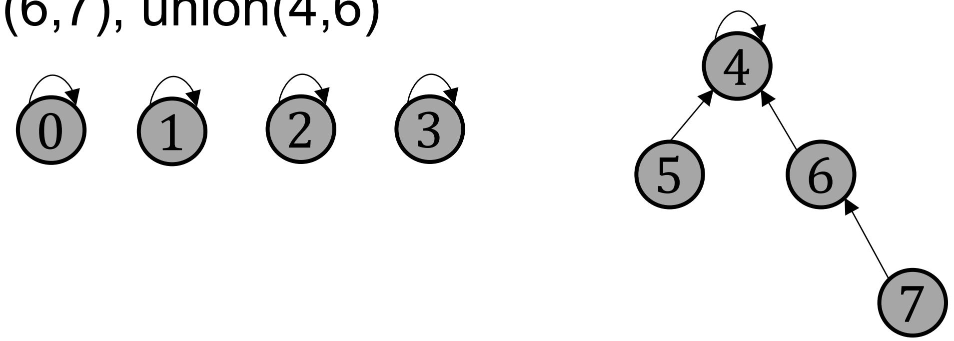
Find : $O(1)$, Union : O(n)

Note: we usually use n to denote the number of vertices (i.e., |V|) and use m to denote the number of edges (i.e., |E|).

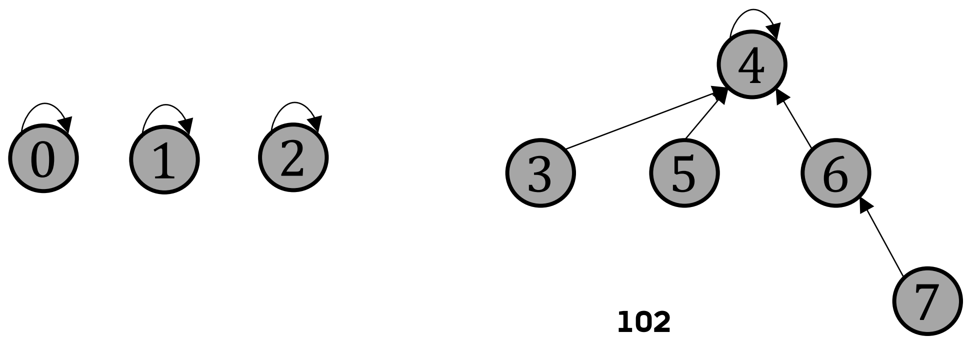
Disjoint Set data structure

Attempt 2: Smart Union: Union by Size

- union(u, v): make smaller tree's root point to bigger one's root
- That is, make v's root point to u's if v's tree is smaller.
- Union(4,5), union(6,7), union(4,6)



Now perform union(3, 4). Smaller tree made the child node.



Disjoint Set data structure

```
Initialize(int N)
  setsize = new int[N+1];
  parent = new int [N+1];
  for (int e=1; e <= N; e++)
    parent[e] = 0;
    setsize[e] = 1;

int Find(int e)
  while (parent[e] != 0)
    e = parent[e];
  return e;

Union(int i, int j)
   i = find(i);
   j = find(j)'
  if setsize[i] < setsize[j]
  then
    setsize[j] += setsize[i];
    parent[i] = j;
  else
    setsize[i] += setsize[j];
    parent[j] = i ;
```

Union by Size:
link smaller tree to larger one

Lemma: After n union ops, the tree height is at most log(n).

Disjoint Set data structure

Time Complexity:

- Find(u) takes time proportional to u’s depth in its tree.

- When union(u, v) performed, the depth of u only increases if its root becomes the child of v’s root. That only happens if v’s tree is larger than u’s tree.

- If u’s depth grows by 1, its (new) treeSize is > 2 * oldTreeSize
  Each increment in depth doubles the size of u’s tree.
  After n union operations, size is at most n, so depth at most log(n).

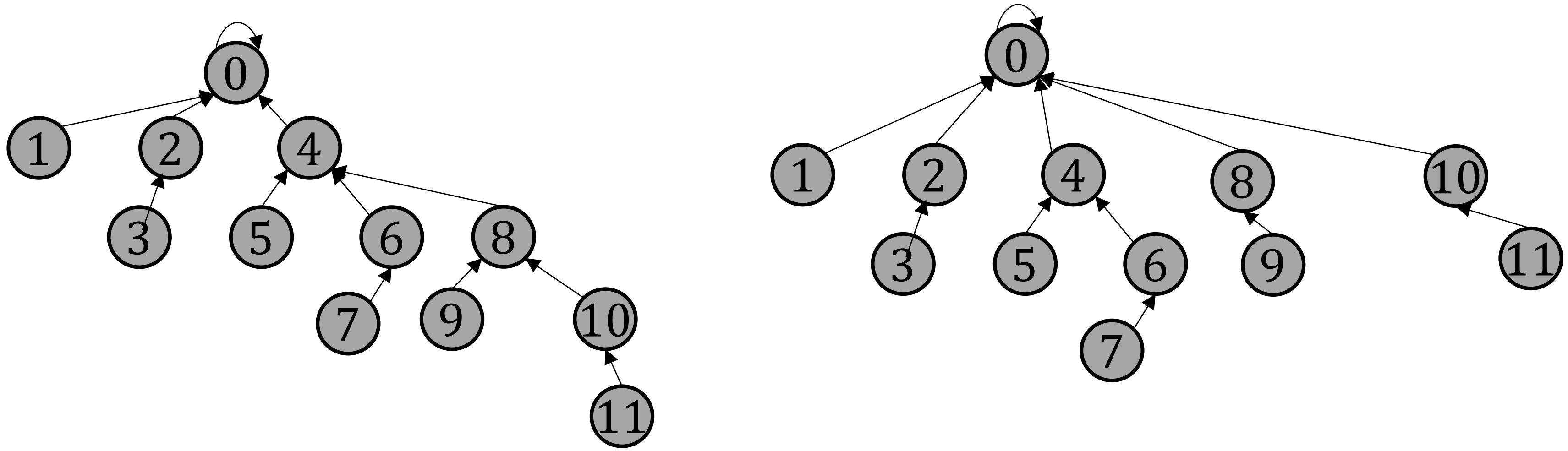
- Theorem: With Union-By-Size, we can do find in O(log n) time and union in O(log(n)) time.

Disjoint Set data structure

- The Ultimate Union-Find: Path compression

```
int Find(int e)
  if (parent[e] == 0)
    return e
  else
    parent[e] = Find(parent[e])
    return parent[e]
```

- While performing Find, direct all nodes on the path to the root.
- Example: Find(10)



Disjoint Set data structure

- The Ultimate Union-Find: Path compression

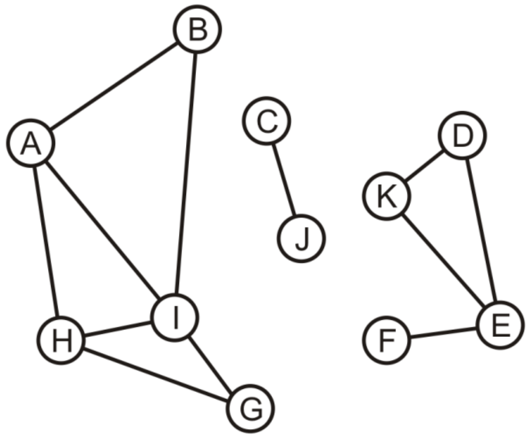
```
int Find(int e)
  if (parent[e] == 0)
    return e
  else
    parent[e] = Find(parent[e])
    return parent[e]
```

- Any single find can still be O(log(n)),
  but later finds on the same path are faster

- Union, Find: "almost linear" total time
- Amortized O(1) time for each Union or Find.

Check Connected Components by Disjoint Sets

We would like to find the connected components by using Disjoint Sets (Union Find).



List all edges in this graph (in alphabetical order):

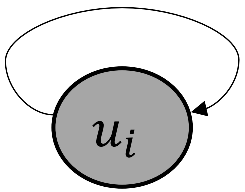
{A,B}, {A,H}, {A,I}, {B,I}, {C,J}, {D,E}, {D,K}, {E,F},
{E,K}, {G,H}, {G,I}, {H,I}

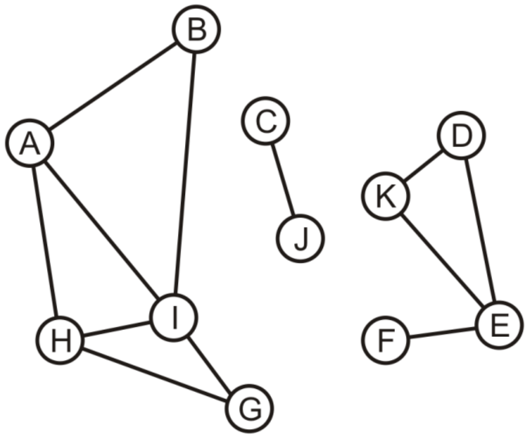
Check Connected Components by Disjoint Sets

Going through the example again with disjoint sets. We start with eleven singletons.

{A}, {B}, {C}, {D}, {E}, {F}, {G}, {H}, {I}, {J}, {K}

Initialization:
Direct all nodes on the path to the root. For each vertex $u_i \in [A, L]$, each vertex direct to themselves.



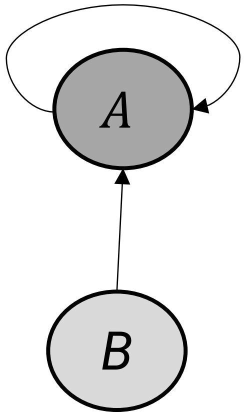


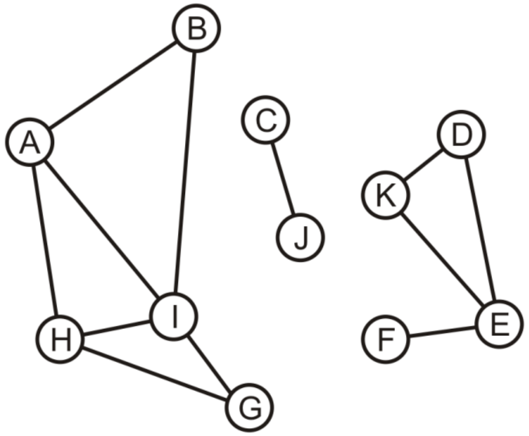
{A, B}
{A, H}
{A, I}
{B, I}
{C, J}
{D, E}
{D, K}
{E, F}
{E, K}
{G, H}
{G, I}
{H, I}

Check Connected Components by Disjoint Sets

We start by adding edge {A, B}

{A, B}, {C}, {D}, {E}, {F}, {G}, {H}, {I}, {J}, {K}





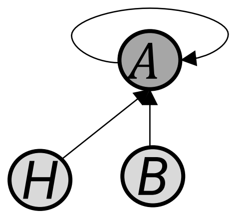
{A, B}
{A, H}
{A, I}
{B, I}
{C, J}
{D, E}
{D, K}
{E, F}
{E, K}
{G, H}
{G, I}
{H, I}

Check Connected Components by Disjoint Sets

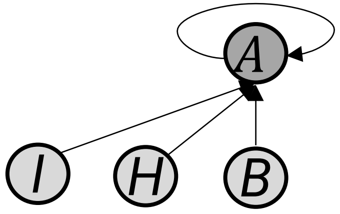
We add edge {A, H}, {A, I}

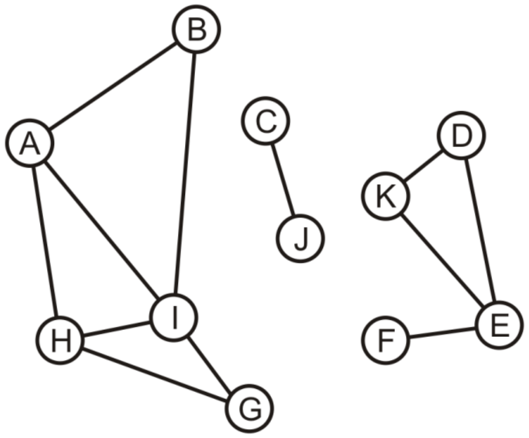
{A, B, H, I}, {C}, {D}, {E}, {F}, {G}, {J}, {K}

Add {A,H}: According to the rule of Union by Size, make smaller tree H point to bigger one's root A.



Add {A,I}:





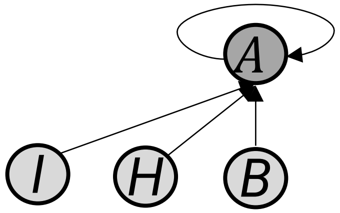
{A, B}
{A, H}
{A, I}
{B, I}
{C, J}
{D, E}
{D, K}
{E, F}
{E, K}
{G, H}
{G, I}
{H, I}

Check Connected Components by Disjoint Sets

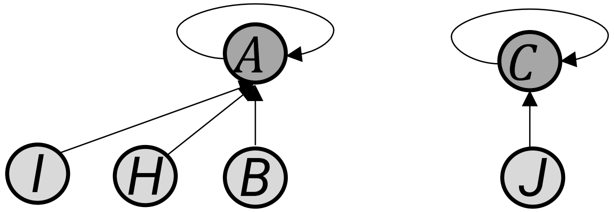
We add edge {B, I}, {C, J}

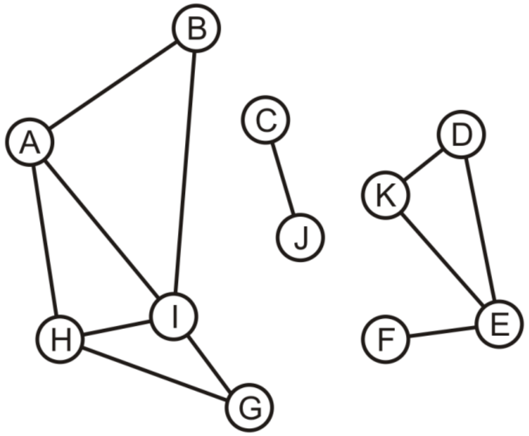
{A, B, H, I}, {C, J}, {D}, {E}, {F}, {G}, {K}

Add {B,I}: B and I are already in the tree, and they all point to the root. Thus, nothing will be changed.



Add {C,J}:





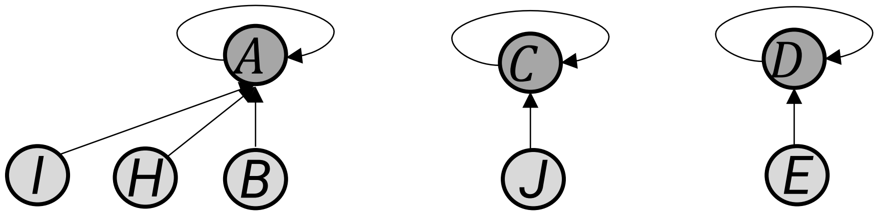
{A, B}
{A, H}
{A, I}
{B, I}
{C, J}
{D, E}
{D, K}
{E, F}
{E, K}
{G, H}
{G, I}
{H, I}

Check Connected Components by Disjoint Sets

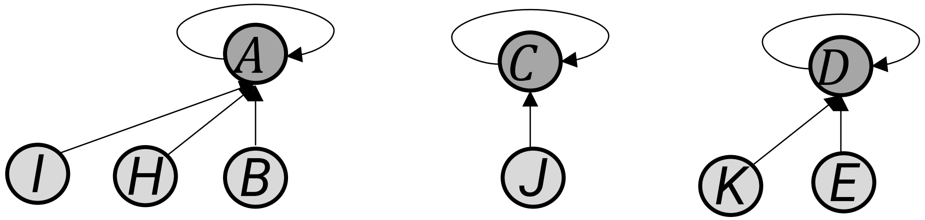
We add edge {D, E}, {D, K}

{A, B, H, I}, {C, J}, {D, E, K}, {F}, {G}

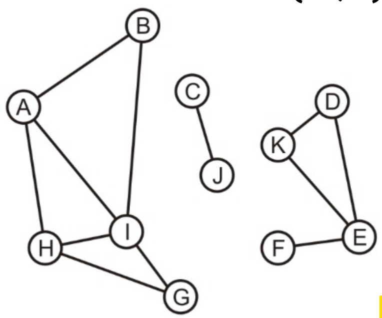
Add {D,E}:



Add {D,K}:



{A, B}
{A, H}
{A, I}
{B, I}
{C, J}
{D, E}
{D, K}
{E, F}
{E, K}
{G, H}
{G, I}
{H, I}

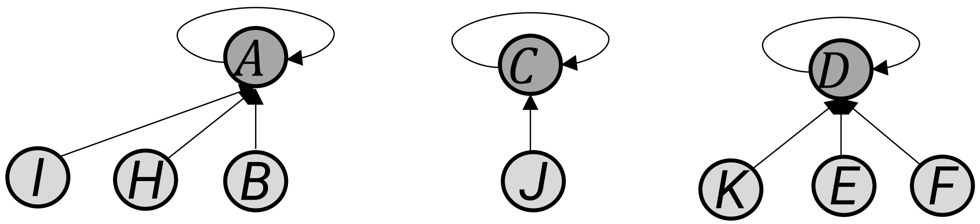


Check Connected Components by Disjoint Sets

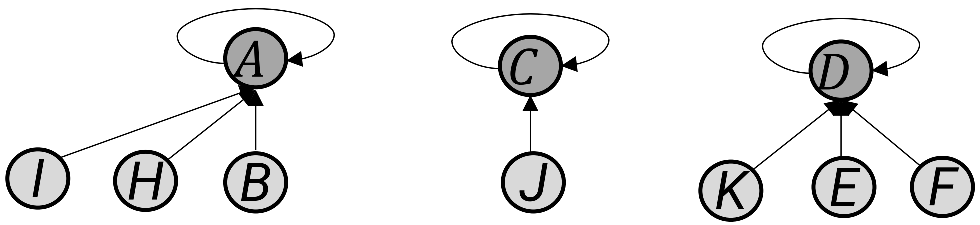
We add edge {E, F}, {E, K}

{A, B, H, I}, {C, J}, {D, E, F, K}, {G}

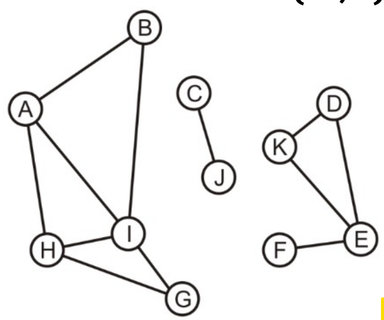
Add {E,F}:



Add {E,K}: E, K are already pointed to root D.
Thus, there is nothing change of adding {E,K}



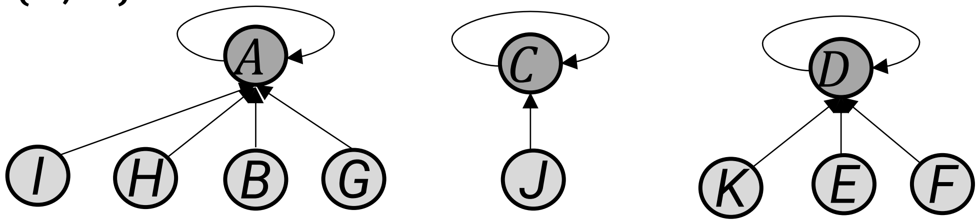
{A, B}
{A, H}
{A, I}
{B, I}
{C, J}
{D, E}
{D, K}
{E, F}
{E, K}
{G, H}
{G, I}
{H, I}



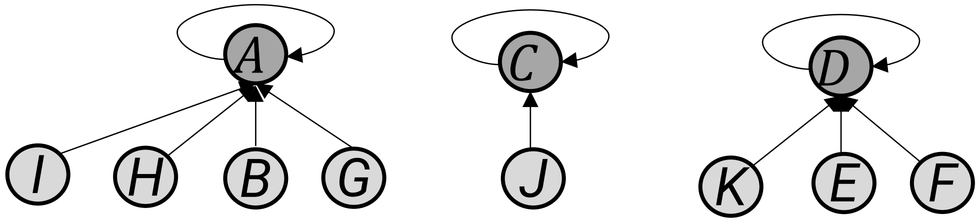
Check Connected Components by Disjoint Sets

We add edge {G, H}, {G, I}, {H, I}

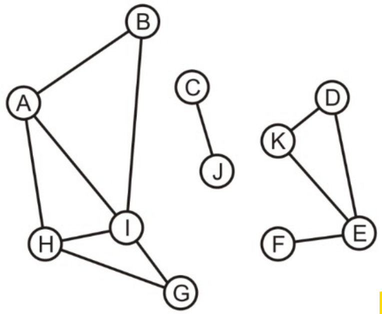
Add {G, H}:



Add {G, I}, {H, I}: G, H, I are already pointed to root A.
Thus, there is nothing change of adding {G, I}, {H, I}.



{A, B}
{A, H}
{A, I}
{B, I}
{C, J}
{D, E}
{D, K}
{E, F}
{E, K}
→ {G, H}
→ {G, I}
→ {H, I}



At last we get result: {A, B, G, H, I}, {C, J}, {D, E, F, K}

Check Connected Components by Disjoint Sets

Build the index: $O(|V|+|E|)$

Space: $O(|V|)$

Good for incremental connected components maintenance~

Coding practice~

Topological Sort

Topological Sort

In this topic, we will discuss:

- Motivations
- The definition of a directed acyclic graph (DAG)
- Describe a topological sort and applications
- Describe the algorithm
- Do a run-time and memory analysis of the algorithm

Motivation

Given a set of tasks with dependencies,

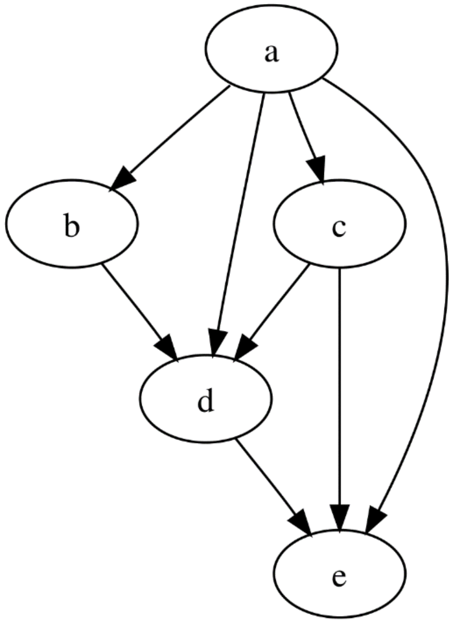
is there an order in which we can complete the tasks?

Dependencies form a partial ordering

A partial ordering on a number of objects can
be represented as a directed acyclic graph (DAG)

Directed acyclic graph (DAG)

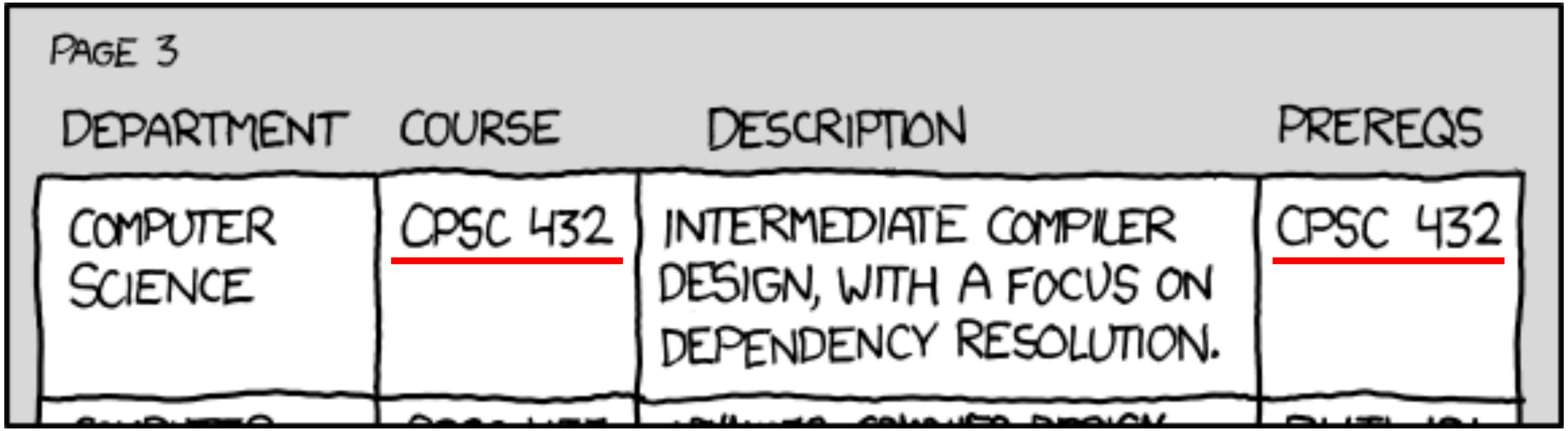
- A **directed acyclic graph** (**DAG**) is a directed graph with no directed cycles.



https://en.wikipedia.org/wiki/Directed_acyclic_graph

Motivation

Cycles in dependencies can cause issues...



http://xkcd.com/754/

Another example: the precedence graph in database transaction management

Restriction of paths in DAGs

Observation:

In a DAG, given two different vertices v_j and v_k,
there cannot **both** be a path from v_j to v_k and a path from v_k to v_j.

Definition:

A topological sorting of the vertices in a DAG is an ordering

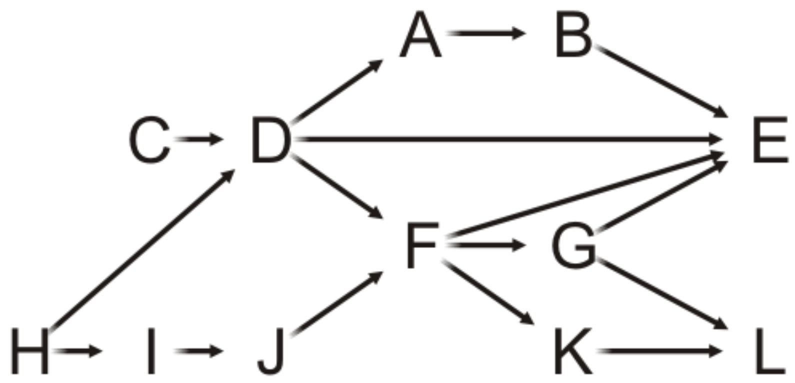
$$v_1, v_2, v_3, \ldots, v_{|V|}$$

such that if there is a path from v_j to v_k , v_j appears before v_k.

Definition of topological sorting

Given this DAG, a topological sort is

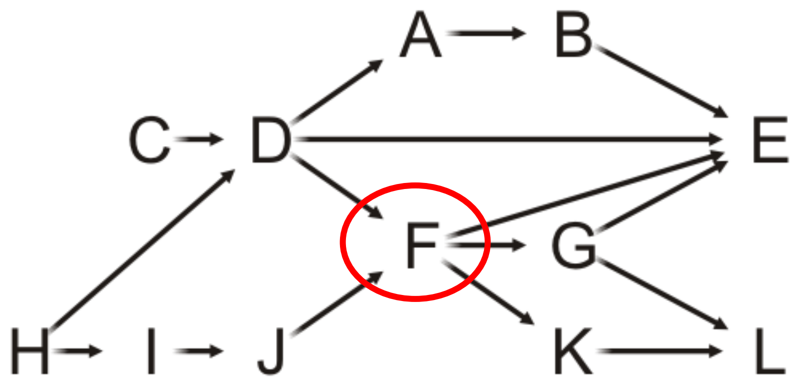
H, C, I, D, J, A, F, B, G, K, E, L



Example

There are paths from H, C, I, D and J to F, so all these must come before F in a topological sort

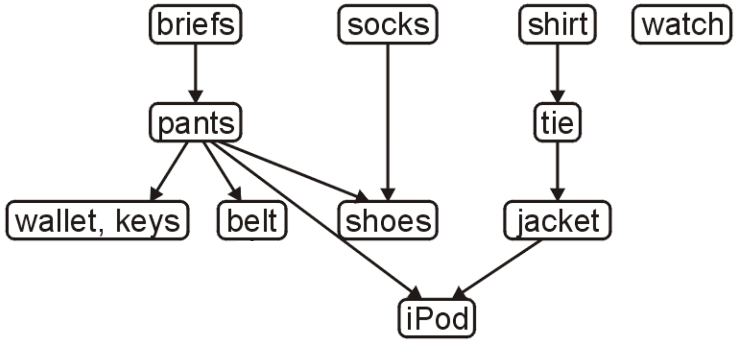
H, **C**, **I**, **D**, **J**, A, **F**, B, G, K, E, L



Clearly, this sorting need not be unique

Applications

The following is a task graph for getting dressed:



One topological sort is:

briefs, pants, wallet, keys, belt, socks, shoes, shirt, tie, jacket, iPod, watch

Another topological sort is:

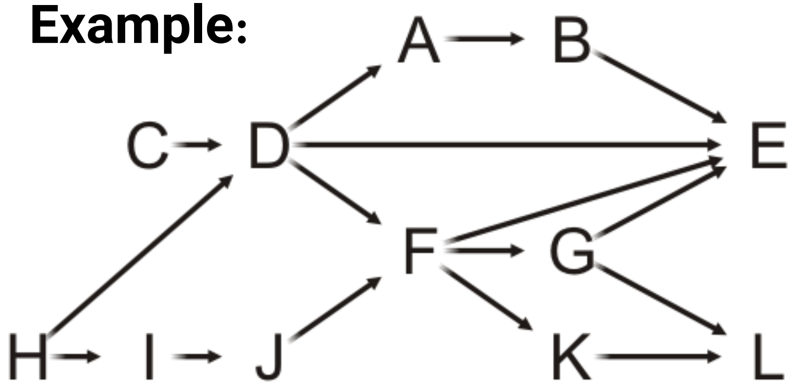
briefs, socks, pants, shirt, belt, tie, jacket, wallet, keys, iPod, watch, shoes

Topological Sort

Idea:

- Given a DAG V, make a copy W and iterate:
  - Find a vertex v in W with in-degree zero (i.e., the source vertex)
  - Let v be the next vertex in the topological sort
  - Continue iterating with the vertex-induced sub-graph $W \setminus \{v\}$

Example:



Possible solutions:

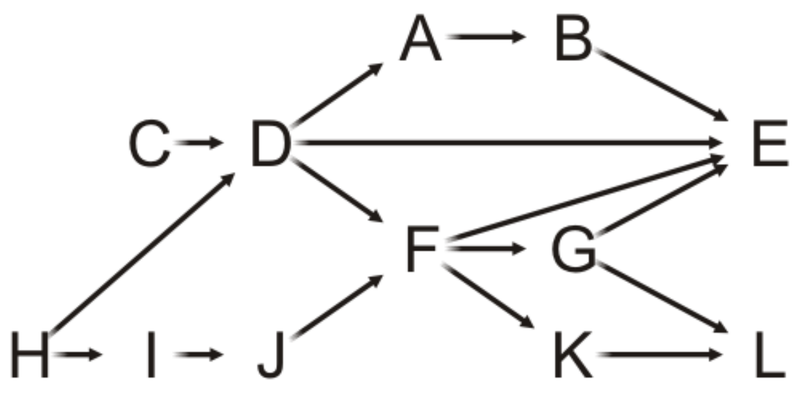
C, H, D, A, B, I, J, F, G, E, K, L

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

Analysis

What are the tools **necessary** for a topological sort?

- We must know and be able to update the in-degrees of each of the vertices
- We could do this with a table of the in-degrees of each of the vertices
- This requires $O(|V|)$ memory



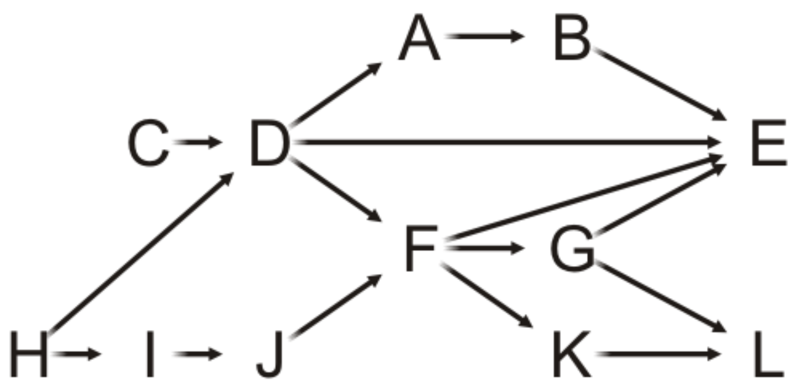
A	1
B	1
C	0
D	2
E	4
F	2
G	1
H	0
I	1
J	1
K	1
L	2

Analysis

We must iterate at least $|V|$ times, so the run-time must be $\Omega(|V|)$

We need to find vertices with in-degree zero

- We **could loop through the array** with each iteration
- The run time would be $O(|V|^2)$

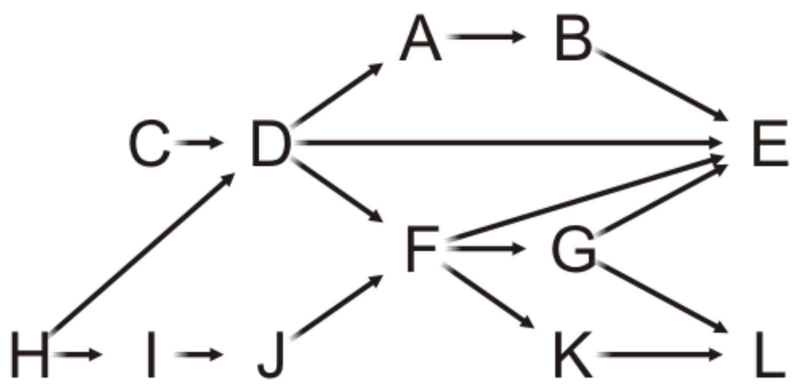


A	1
B	1
C	0
D	2
E	4
F	2
G	1
H	0
I	1
J	1
K	1
L	2

Analysis

How did we do with BFS and DFS?

- Use a queue (or other container) to temporarily store those vertices with in-degree zero
- Each time the in-degree of a vertex is decremented to zero, push it onto the queue

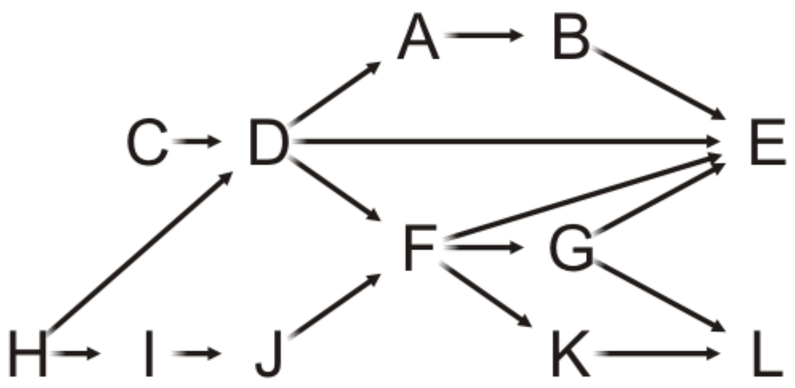


A	1
B	1
C	0
D	2
E	4
F	2
G	1
H	0
I	1
J	1
K	1
L	2

Analysis

What are the run times associated with the queue?

- Initially, we must scan through each of the vertices: $O(|V|)$
- For each vertex, we will have to push onto and pop off the queue once, also $O(|V|)$

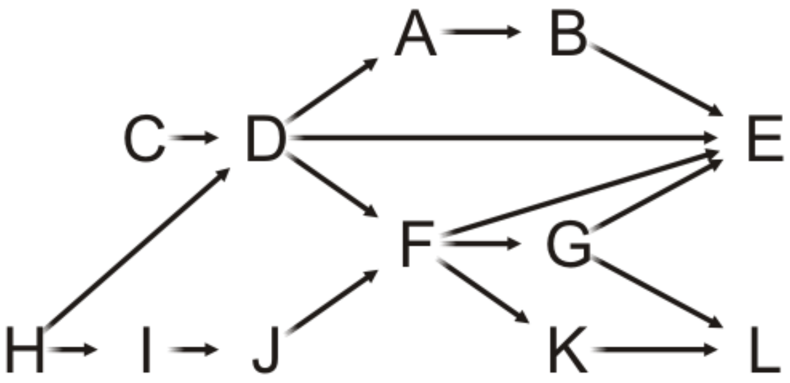


A	1
B	1
C	0
D	2
E	4
F	2
G	1
H	0
I	1
J	1
K	1
L	2

Analysis

Finally, each value in the in-degree table is associated with an edge

- Here, $|E| = 16$
- Each of the in-degrees must be decremented to zero
- The run time of these operations is $O(|E|)$
- If we are using an adjacency matrix: $O(|V|^2)$
- If we are using an adjacency list: $O(|E|)$



A	1
B	+ 1
C	+ 0
D	+ 2
E	+ 4
F	+ 2
G	+ 1
H	+ 0
I	+ 1
J	+ 1
K	+ 1
L	+ 2

16

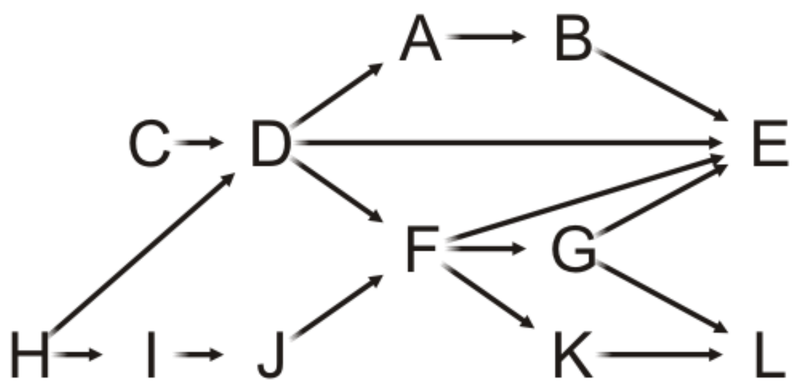
Analysis

Therefore, the run time of a topological sort is:

$O(|V| + |E|)$ if we use an adjacency list

$O(|V|^2)$ if we use an adjacency matrix

and the additional memory requirements is $O(|V|)$



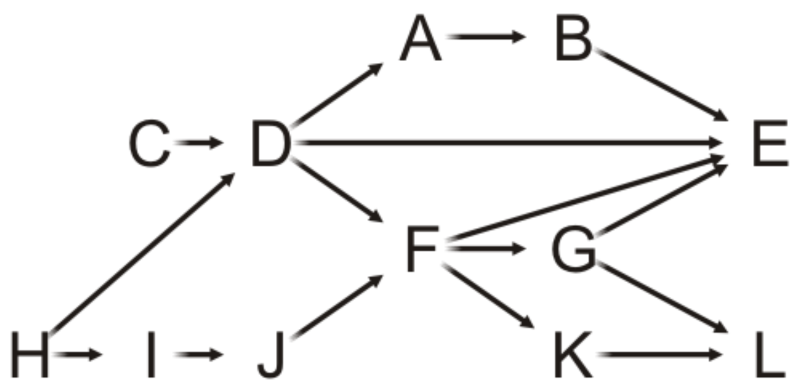
A	1
B	1
C	0
D	2
E	4
F	2
G	1
H	0
I	1
J	1
K	1
L	2

Analysis

What happens if at some step, all remaining vertices have an in-degree greater than zero?

- There must be at least one cycle within that sub-set of vertices

Consequence: we now have an $o(|V| + |E|)$ algorithm for determining if a graph has a cycle



A	1
B	1
C	0
D	2
E	4
F	2
G	1
H	0
I	1
J	1
K	1
L	2

Implementation

Thus, to implement a topological sort:

- Allocate memory for and initialize an array of in-degrees
- Create a queue and initialize it with all vertices that have in-degree zero

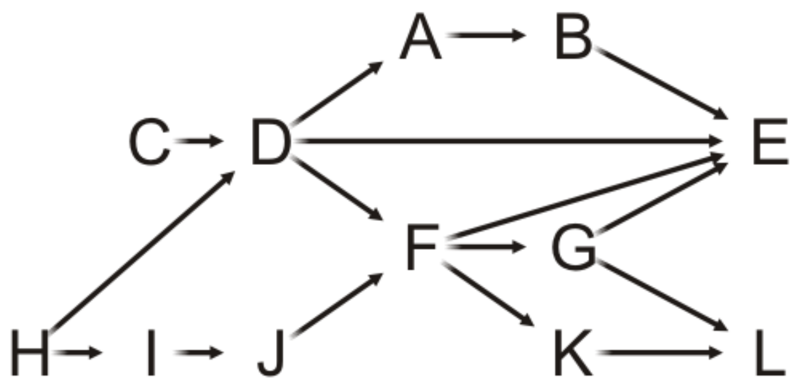
While the queue is not empty:

- Pop a vertex from the queue
- Decrement the in-degree of each neighbor
- Those neighbors whose in-degree was decremented to zero are pushed onto the queue

Example

With the previous example, we initialize:

- The array of in-degrees
- The queue



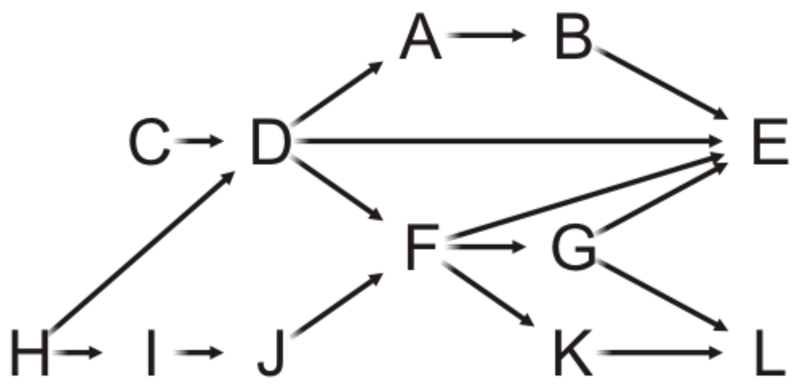
A	1
B	1
C	0
D	2
E	4
F	2
G	1
H	0
I	1
J	1
K	1
L	2

Queue:

The queue is empty

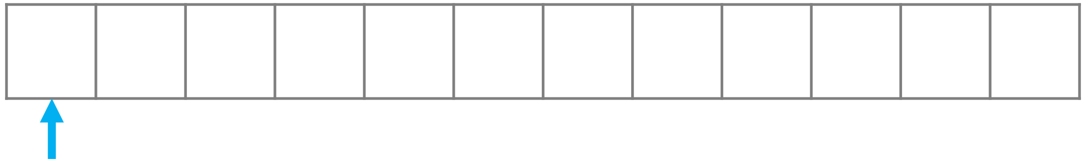
Example

Stepping through the table, push all source vertices into the queue



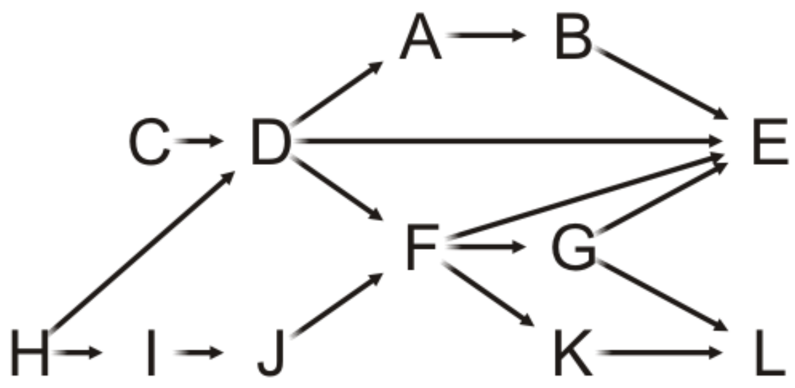
A	1
B	1
C	0
D	2
E	4
F	2
G	1
H	0
I	1
J	1
K	1
L	2

Queue:



Example

Stepping through the table, push all source vertices into the queue



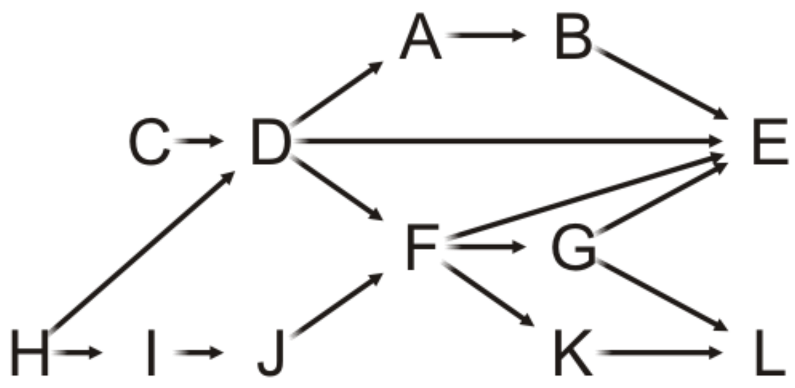
A	1
B	1
C	**0**
D	2
E	4
F	2
G	1
H	**0**
I	1
J	1
K	1
L	2

Queue:

C	**H**										

Example

Pop the front of the queue



A	1
B	1
C	0
D	2
E	4
F	2
G	1
H	0
I	1
J	1
K	1
L	2

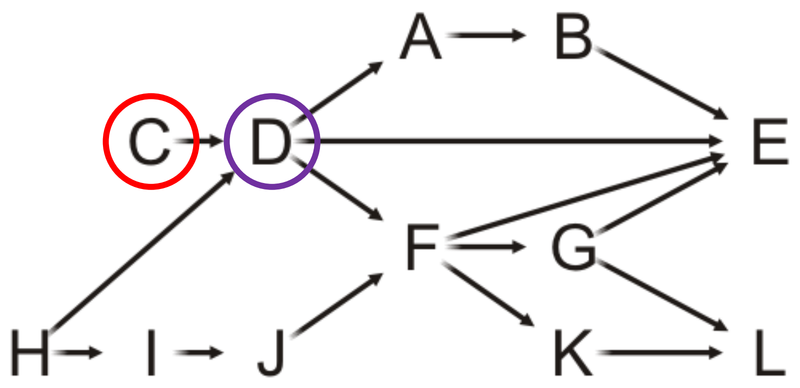
Queue:

C	H										

Example

Pop the front of the queue

- C has one neighbor: D



A	1
B	1
C	**0**
D	**2**
E	4
F	2
G	1
H	0
I	1
J	1
K	1
L	2

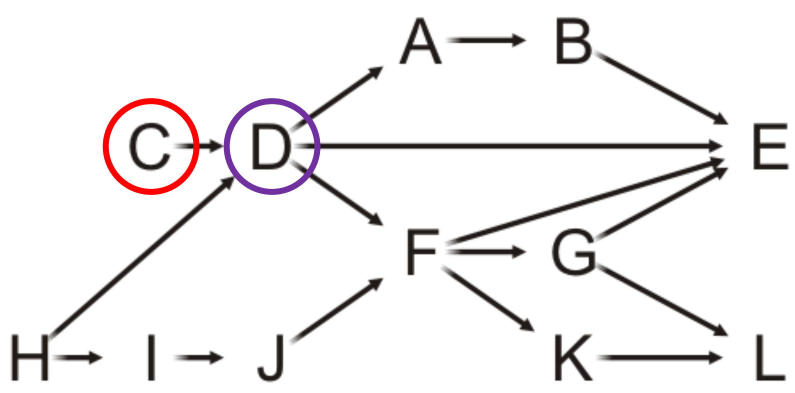
Queue:

C	H										

Example

Pop the front of the queue

- C has one neighbor: D
- Decrement its in-degree



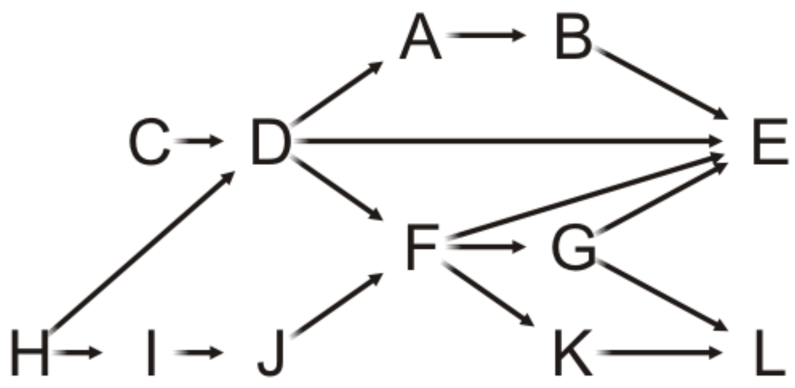
A	1
B	1
C	**0**
D	**1**
E	4
F	2
G	1
H	0
I	1
J	1
K	1
L	2

Queue:

C	H										

Example

Pop the front of the queue



A	1
B	1
C	0
D	1
E	4
F	2
G	1
H	0
I	1
J	1
K	1
L	2

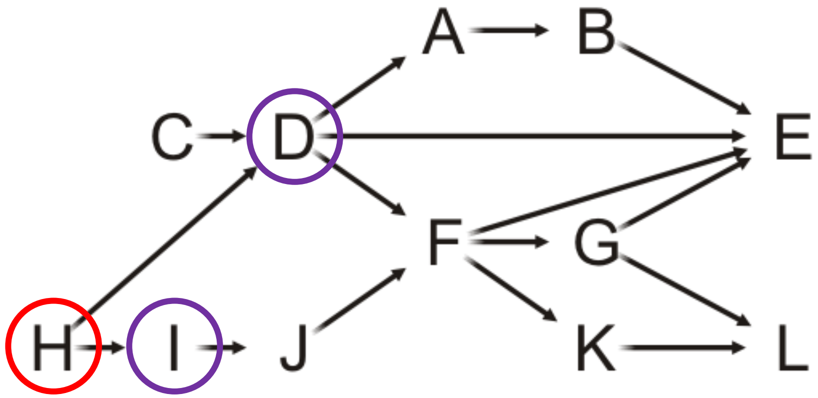
Queue:

C	H										

Example

Pop the front of the queue

- H has two neighbors: D and I



A	1
B	1
C	0
D	**1**
E	4
F	2
G	1
H	**0**
I	**1**
J	1
K	1
L	2

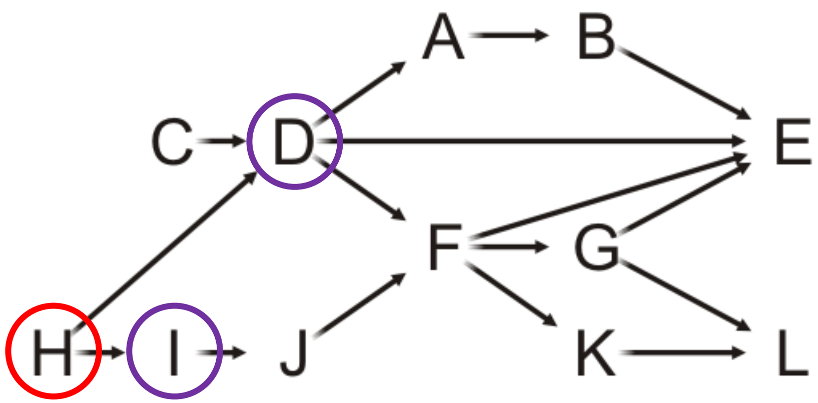
Queue:

C	H										

Example

Pop the front of the queue

- H has two neighbors: D and I
- Decrement their in-degrees



Queue:

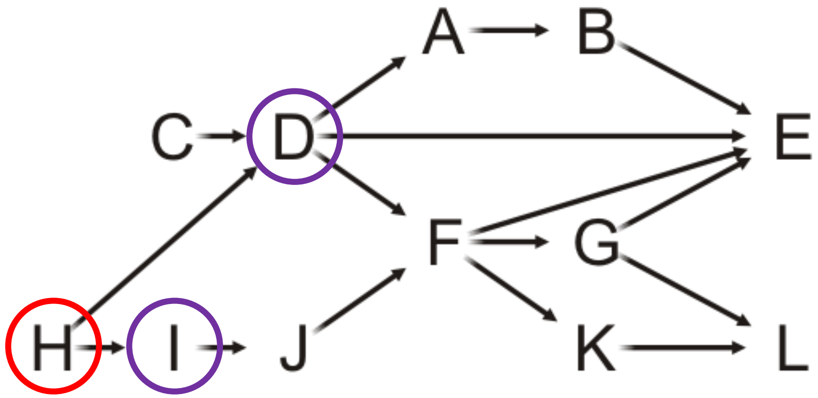
C	H										

A	1
B	1
C	0
D	**0**
E	4
F	2
G	1
H	**0**
I	**0**
J	1
K	1
L	2

Example

Pop the front of the queue

- H has two neighbors: D and I
- Decrement their in-degrees
    - Both are decremented to zero, so push them onto the queue



Queue:

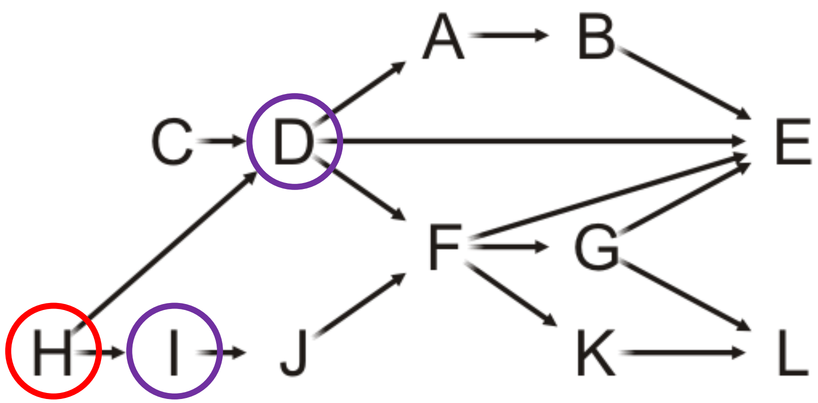
C	H										

A	1
B	1
C	0
D	**0**
E	4
F	2
G	1
H	**0**
I	**0**
J	1
K	1
L	2

Example

Pop the front of the queue

- H has two neighbors: D and I
- Decrement their in-degrees
    - Both are decremented to zero, so push them onto the queue



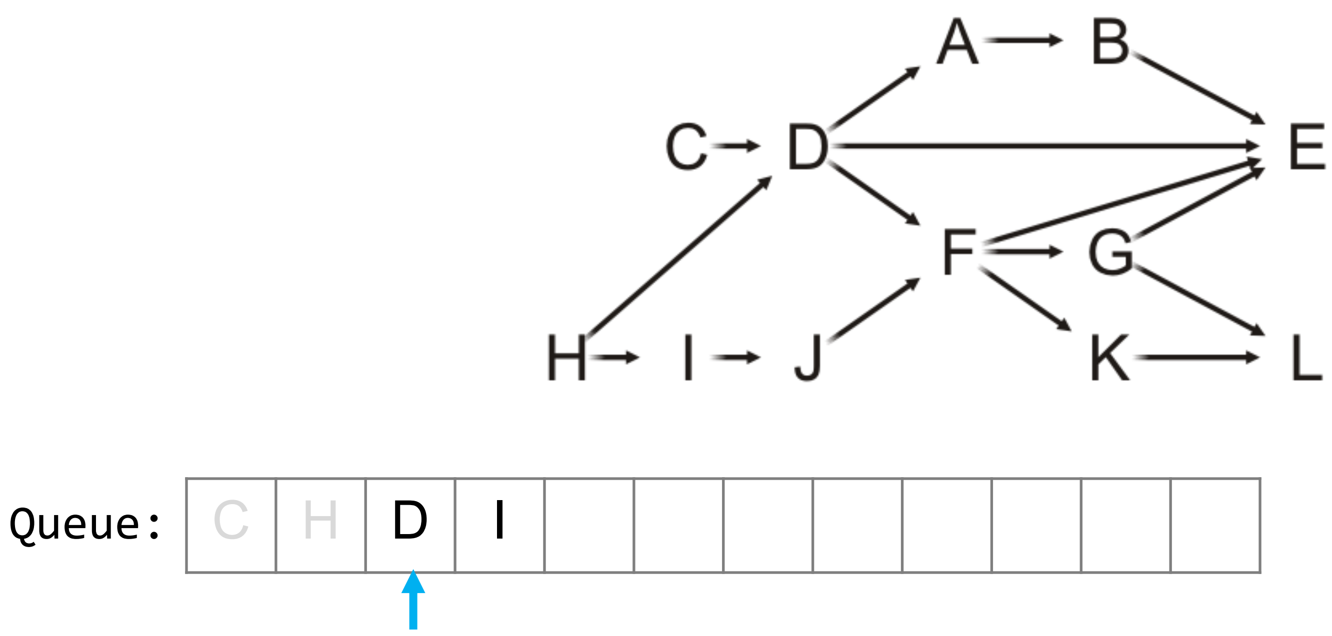
Queue:

C	H	D	I								

A	1
B	1
C	0
D	0
E	4
F	2
G	1
H	0
I	0
J	1
K	1
L	2

Example

Pop the front of the queue

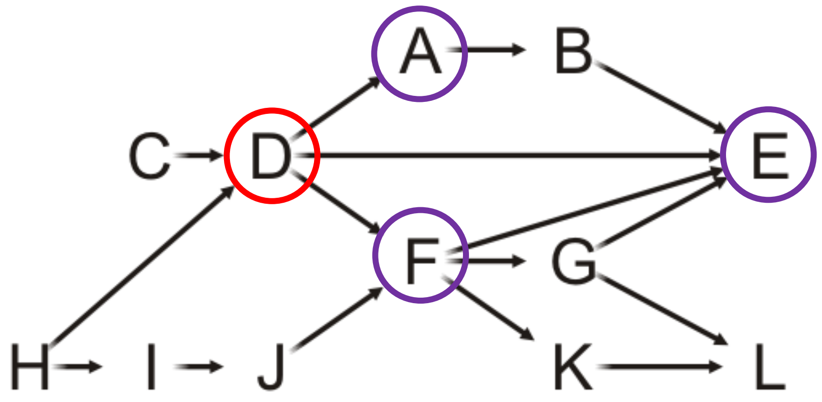


A	1
B	1
C	0
D	0
E	4
F	2
G	1
H	0
I	0
J	1
K	1
L	2

Example

Pop the front of the queue

- D has three neighbors: A, E and F



A	1
B	1
C	0
D	0
E	4
F	2
G	1
H	0
I	0
J	1
K	1
L	2

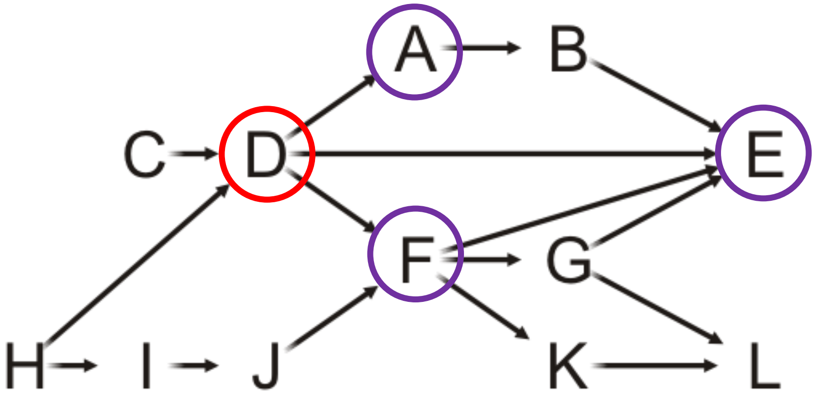
Queue:

C	H	D	I								

Example

Pop the front of the queue

- D has three neighbors: A, E and F
- Decrement their in-degrees



A	0
B	1
C	0
D	0
E	3
F	1
G	1
H	0
I	0
J	1
K	1
L	2

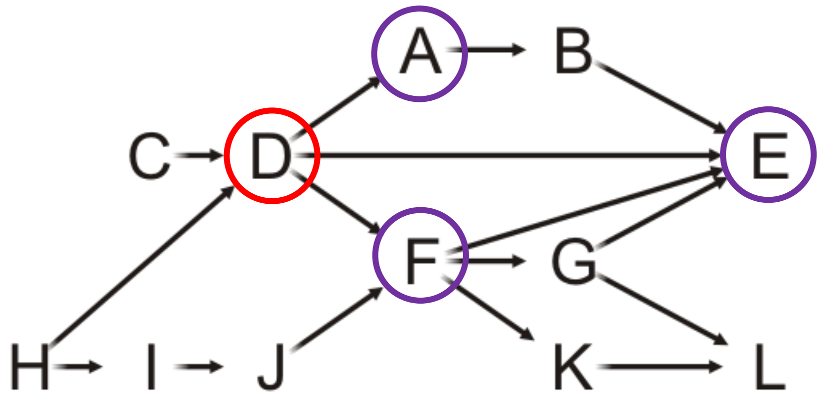
Queue:

C	H	D	I								

Example

Pop the front of the queue

- D has three neighbors: A, E and F
- Decrement their in-degrees
  - A is decremented to zero, so push it onto the queue



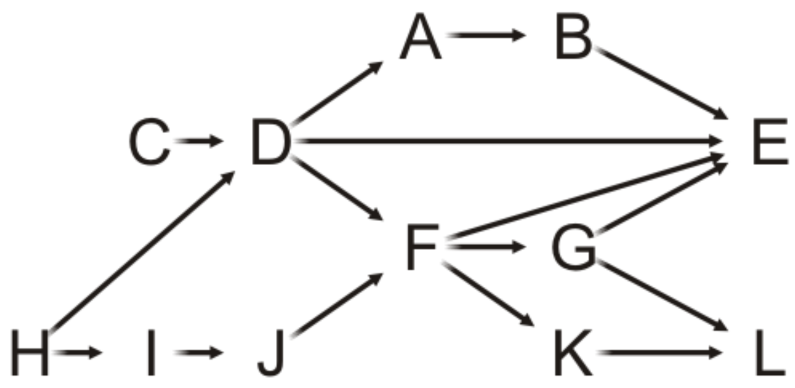
A	0
B	1
C	0
D	0
E	3
F	1
G	1
H	0
I	0
J	1
K	1
L	2

Queue:

C	H	D	I	A							

Example

Pop the front of the queue



A	0
B	1
C	0
D	0
E	3
F	1
G	1
H	0
I	0
J	1
K	1
L	2

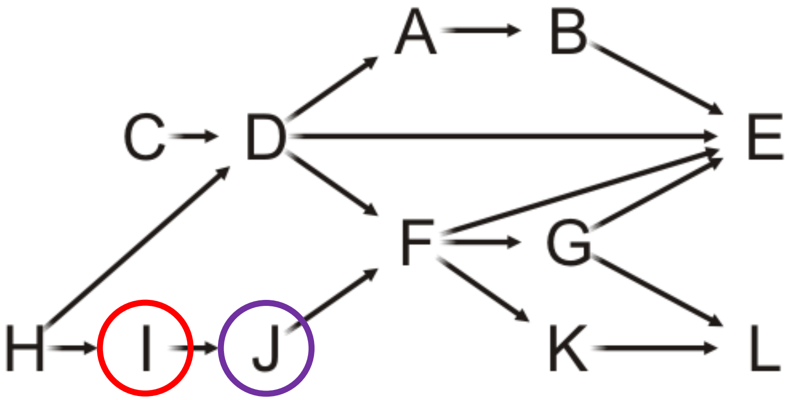
Queue:

C	H	D	I	A							

Example

Pop the front of the queue

- I has one neighbor: J



A	0
B	1
C	0
D	0
E	3
F	1
G	1
H	0
I	**0**
J	**1**
K	1
L	2

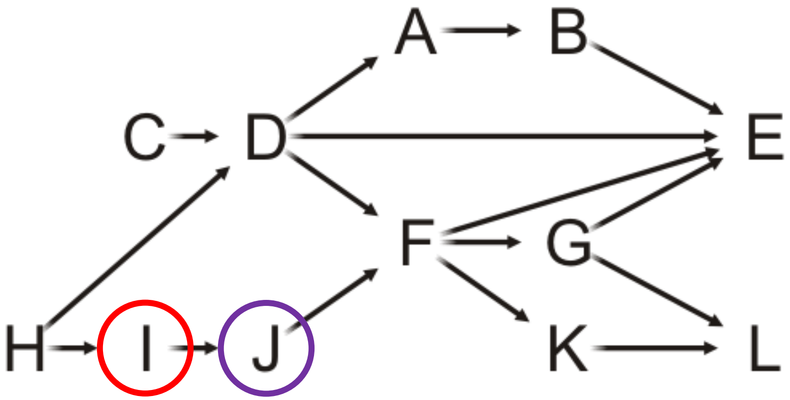
Queue:

C	H	D	I	A							

Example

Pop the front of the queue

- I has one neighbor: J
- Decrement its in-degree



Queue:

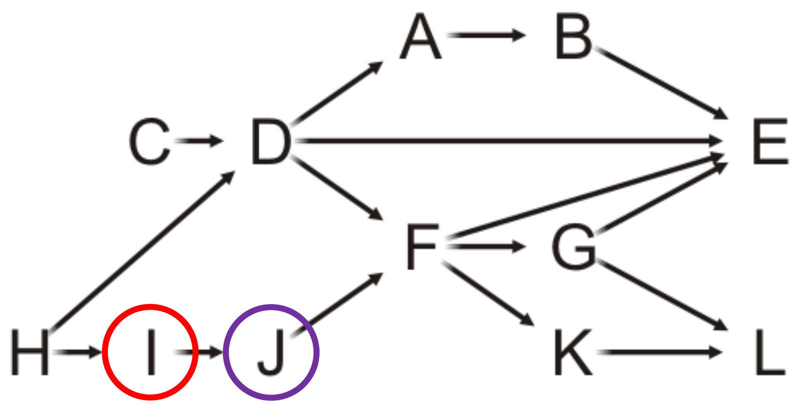
C	H	D	I	A							

A	0
B	1
C	0
D	0
E	3
F	1
G	1
H	0
I	**0**
J	**0**
K	1
L	2

Example

Pop the front of the queue

- I has one neighbor: J
- Decrement its in-degree
  - J is decremented to zero, so push it onto the queue



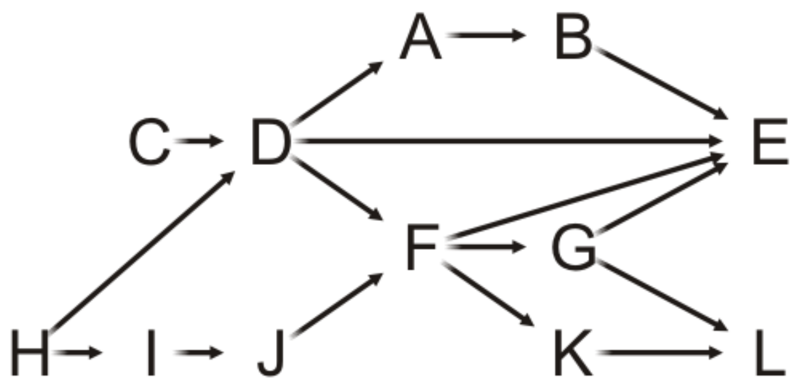
A	0
B	1
C	0
D	0
E	3
F	1
G	1
H	0
I	0
J	0
K	1
L	2

Queue:

C	H	D	I	A	J						

Example

Pop the front of the queue



Queue:

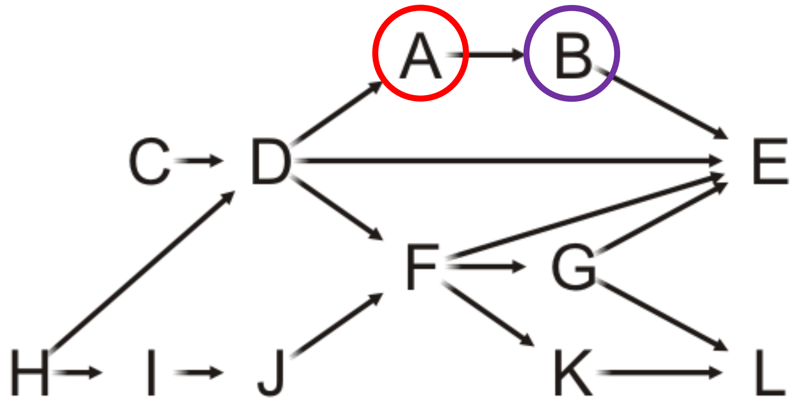
C	H	D	I	A	J						

A	0
B	1
C	0
D	0
E	3
F	1
G	1
H	0
I	0
J	0
K	1
L	2

Example

Pop the front of the queue

- A has one neighbor: B



A	0
B	1
C	0
D	0
E	3
F	1
G	1
H	0
I	0
J	0
K	1
L	2

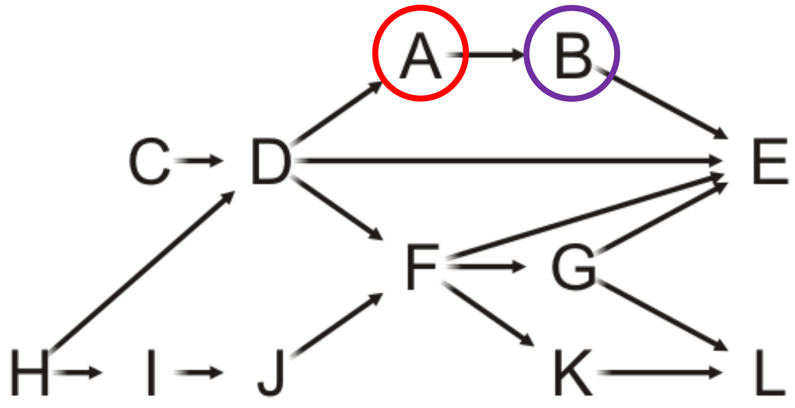
Queue:

C	H	D	I	A	J						

Example

Pop the front of the queue

- A has one neighbor: B
- Decrement its in-degree



Queue:

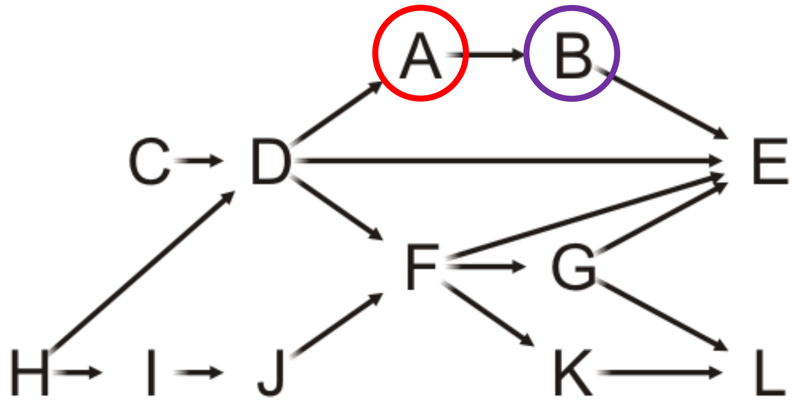
C	H	D	I	A	J						

A	0
B	0
C	0
D	0
E	3
F	1
G	1
H	0
I	0
J	0
K	1
L	2

Example

Pop the front of the queue

- A has one neighbor: B
- Decrement its in-degree
    - B is decremented to zero, so push it onto the queue



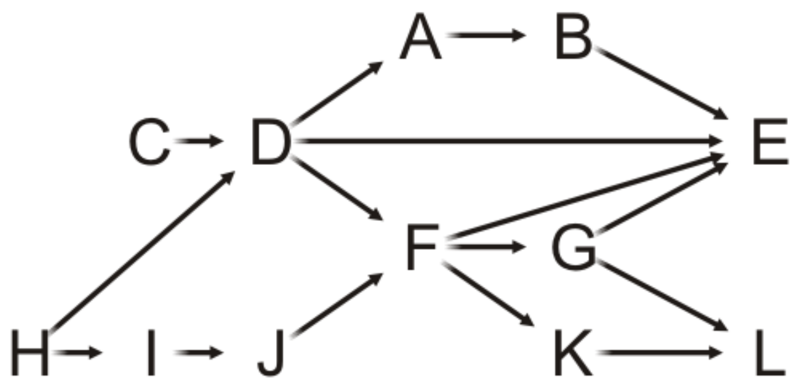
Queue:

C	H	D	I	A	J	B					

A	0
B	0
C	0
D	0
E	3
F	1
G	1
H	0
I	0
J	0
K	1
L	2

Example

Pop the front of the queue



Queue:

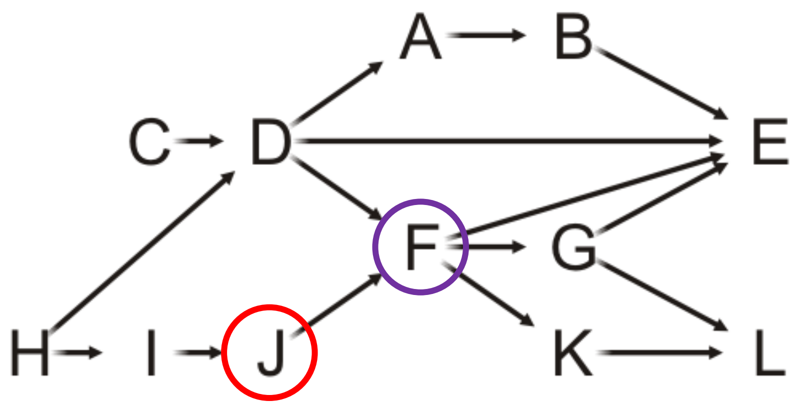
C	H	D	I	A	J	B					

A	0
B	0
C	0
D	0
E	3
F	1
G	1
H	0
I	0
J	0
K	1
L	2

Example

Pop the front of the queue

- J has one neighbor: F



A	0
B	0
C	0
D	0
E	3
F	1
G	1
H	0
I	0
J	0
K	1
L	2

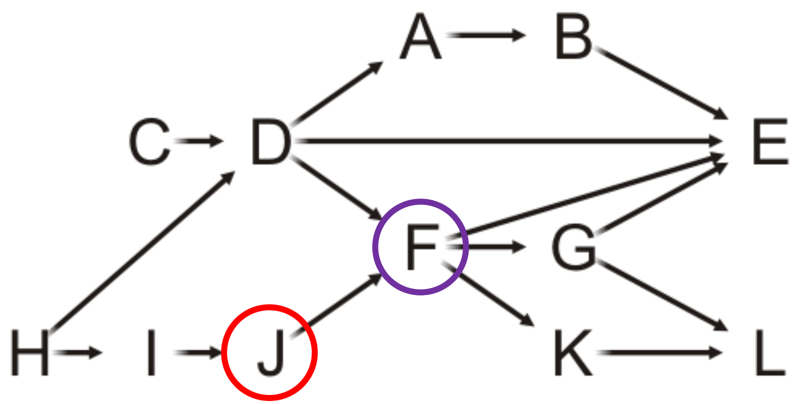
Queue:

C	H	D	I	A	J	B					

Example

Pop the front of the queue

- J has one neighbor: F
- Decrement its in-degree



Queue:

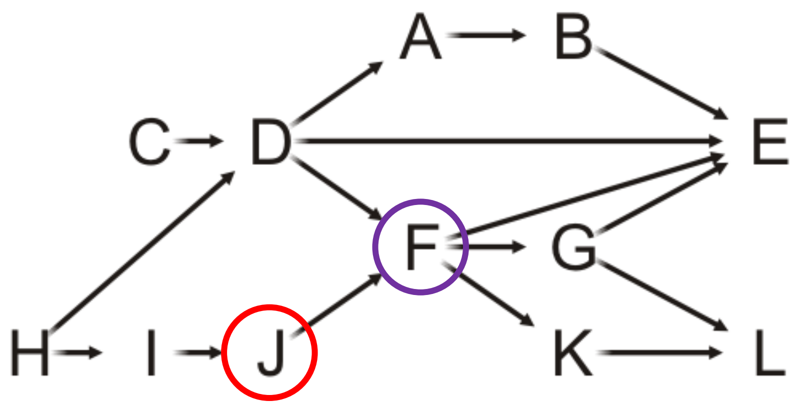
C	H	D	I	A	J	B					

A	0
B	0
C	0
D	0
E	3
F	**0**
G	1
H	0
I	0
J	**0**
K	1
L	2

Example

Pop the front of the queue

- J has one neighbor: F
- Decrement its in-degree
    - F is decremented to zero, so push it onto the queue



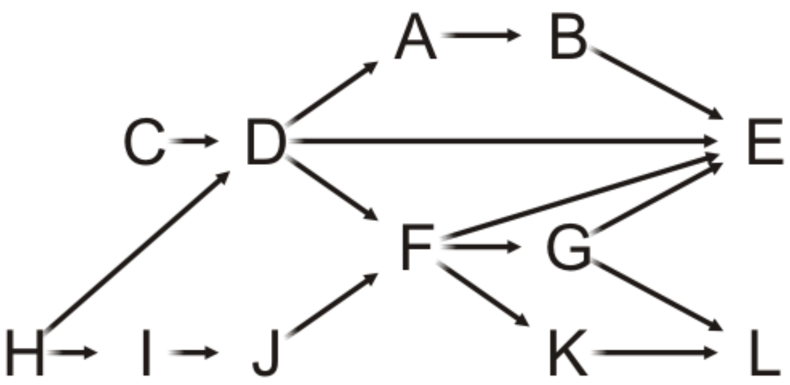
A	0
B	0
C	0
D	0
E	3
F	**0**
G	1
H	0
I	0
J	**0**
K	1
L	2

Queue:

C	H	D	I	A	J	B	F				

Example

Pop the front of the queue



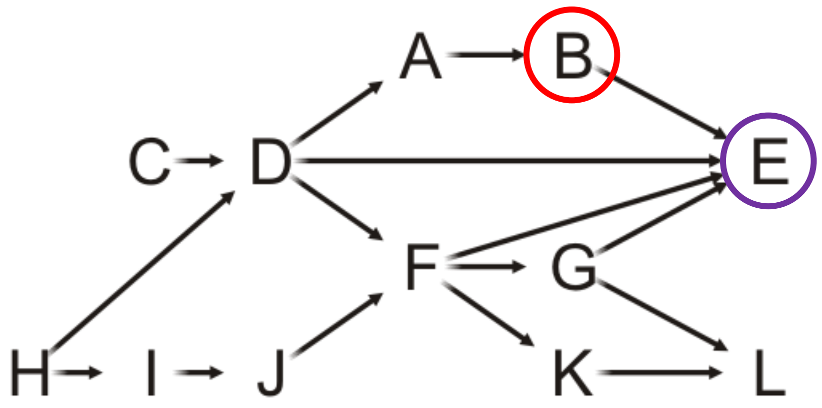
Queue:	C	H	D	I	A	J	B	F				

A	0
B	0
C	0
D	0
E	3
F	0
G	1
H	0
I	0
J	0
K	1
L	2

Example

Pop the front of the queue

- B has one neighbor: E



A	0
B	**0**
C	0
D	0
E	**3**
F	0
G	1
H	0
I	0
J	0
K	1
L	2

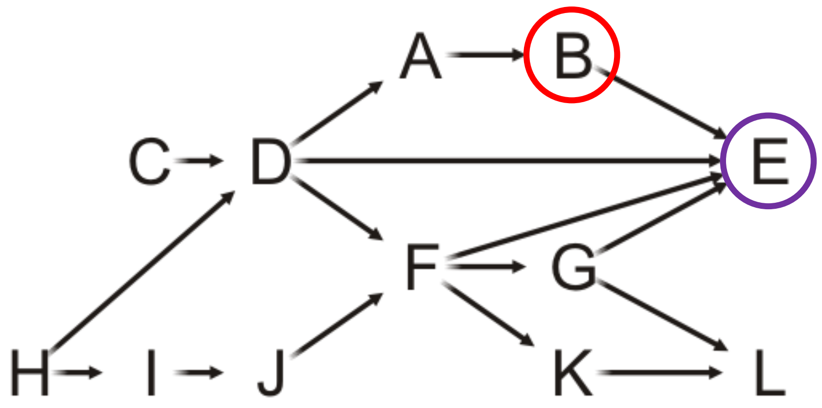
Queue:

C	H	D	I	A	J	B	F				

Example

Pop the front of the queue

- B has one neighbor: E
- Decrement its in-degree



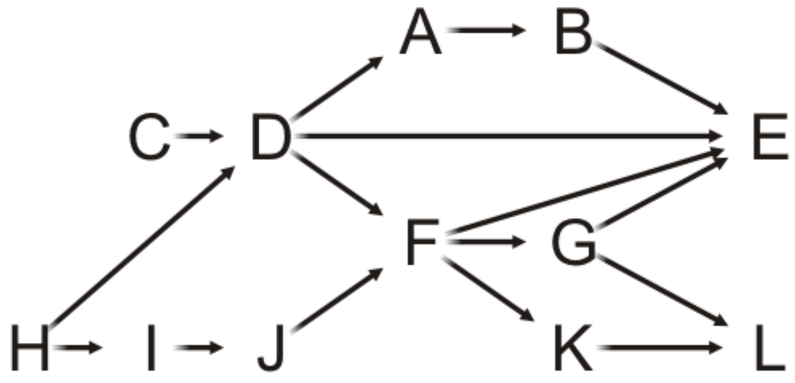
A	0
B	0
C	0
D	0
E	2
F	0
G	1
H	0
I	0
J	0
K	1
L	2

Queue:

C	H	D	I	A	J	B	F				

Example

Pop the front of the queue



A	0	
B	0	
C	0	
D	0	
E	2	
F	0	
G	1	
H	0	
I	0	
J	0	
K	1	
L	2	

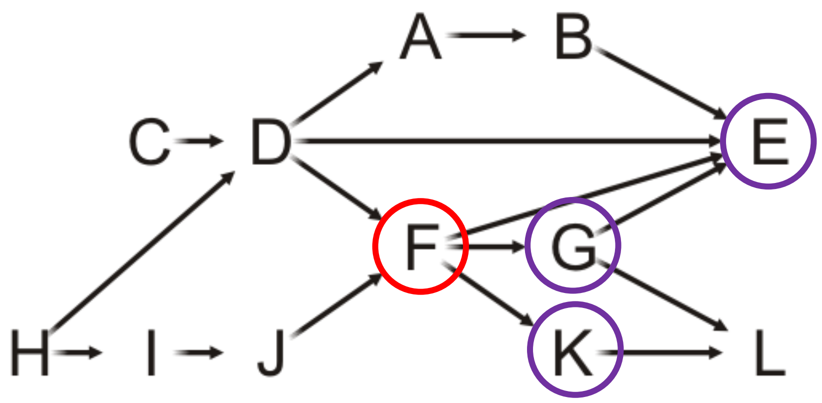
Queue:

C	H	D	I	A	J	B	F				

Example

Pop the front of the queue

- F has three neighbors: E, G and K



A	0
B	0
C	0
D	0
E	**2**
F	**0**
G	**1**
H	0
I	0
J	0
K	**1**
L	2

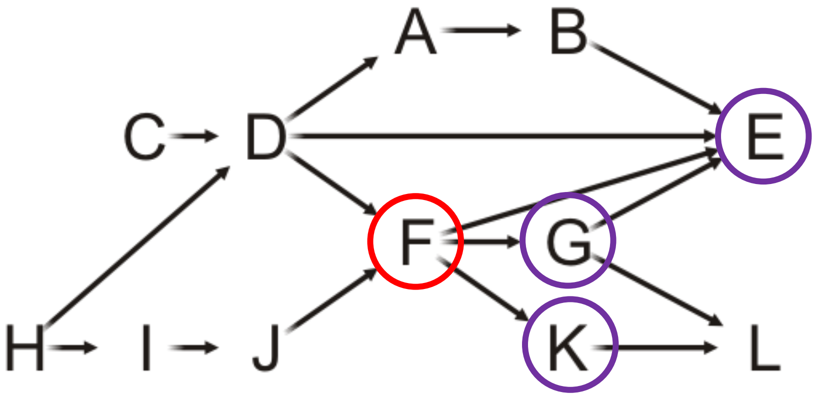
Queue:

C	H	D	I	A	J	B	F				

Example

Pop the front of the queue

- F has three neighbors: E, G and K
- Decrement their in-degrees



Queue:

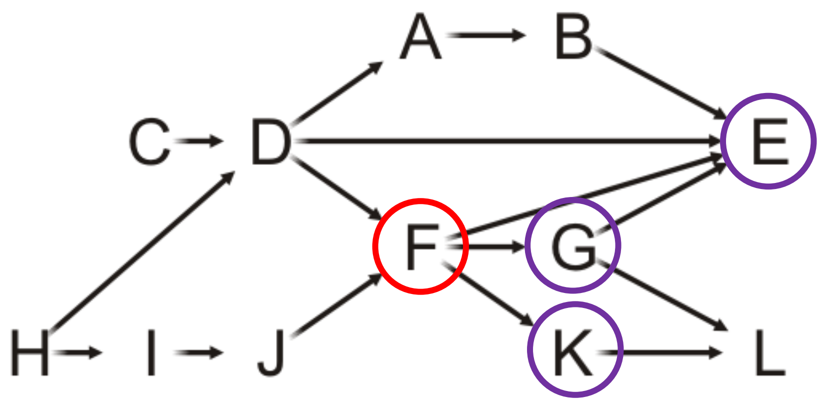
C	H	D	I	A	J	B	F				

A	0
B	0
C	0
D	0
E	**1**
F	**0**
G	**0**
H	0
I	0
J	0
K	**0**
L	2

Example

Pop the front of the queue

- F has three neighbors: E, G and K
- Decrement their in-degrees
    - G and K are decremented to zero,
      so push them onto the queue



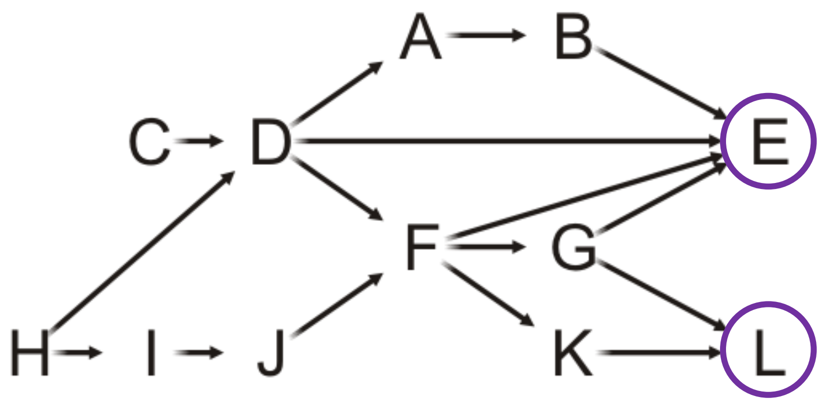
Queue:

C	H	D	I	A	J	B	F	G	K		

A	0
B	0
C	0
D	0
E	**1**
F	**0**
G	**0**
H	0
I	0
J	0
K	**0**
L	2

Example

Pop the front of the queue



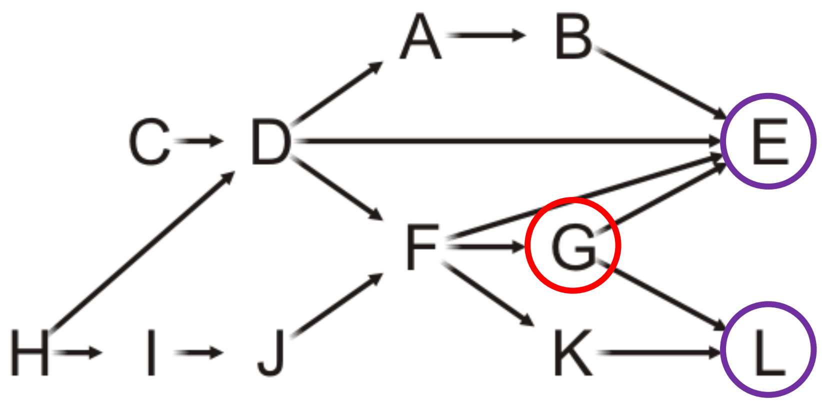
Queue:	C	H	D	I	A	J	B	F	G	K		

A	0
B	0
C	0
D	0
E	1
F	0
G	0
H	0
I	0
J	0
K	0
L	2

Example

Pop the front of the queue

- G has two neighbors: E and L



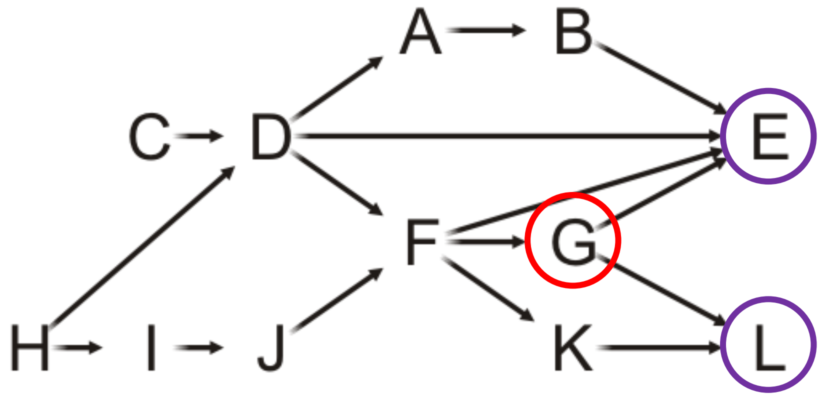
Queue:	C	H	D	I	A	J	B	F	G	K		

A	0
B	0
C	0
D	0
E	**1**
F	0
G	**0**
H	0
I	0
J	0
K	0
L	**2**

Example

Pop the front of the queue

- G has two neighbors: E and L
- Decrement their in-degrees



Queue:

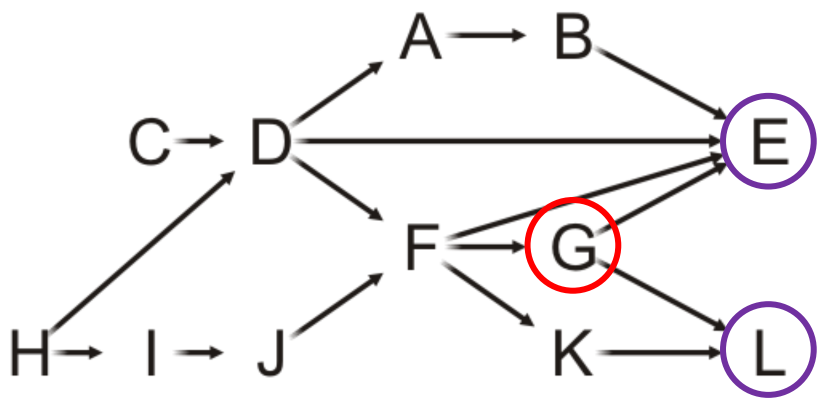
C	H	D	I	A	J	B	F	G	K		

A	0
B	0
C	0
D	0
E	**0**
F	0
G	**0**
H	0
I	0
J	0
K	0
L	**1**

Example

Pop the front of the queue

- G has two neighbors: E and L
- Decrement their in-degrees
  - E is decremented to zero, so push it onto the queue



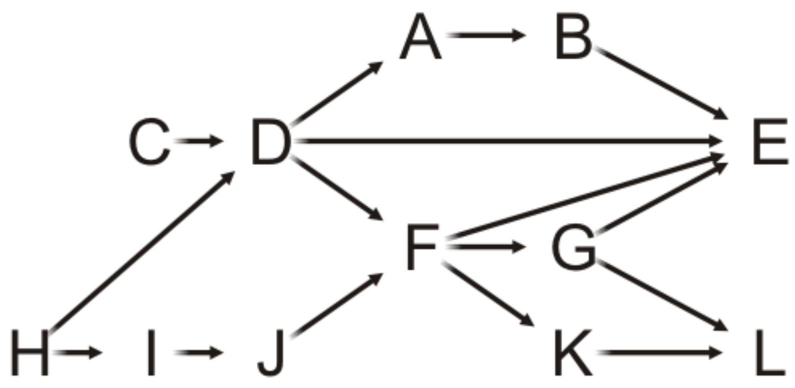
Queue:

C	H	D	I	A	J	B	F	G	K	E	

A	0
B	0
C	0
D	0
E	**0**
F	0
G	**0**
H	0
I	0
J	0
K	0
L	**1**

Example

Pop the front of the queue



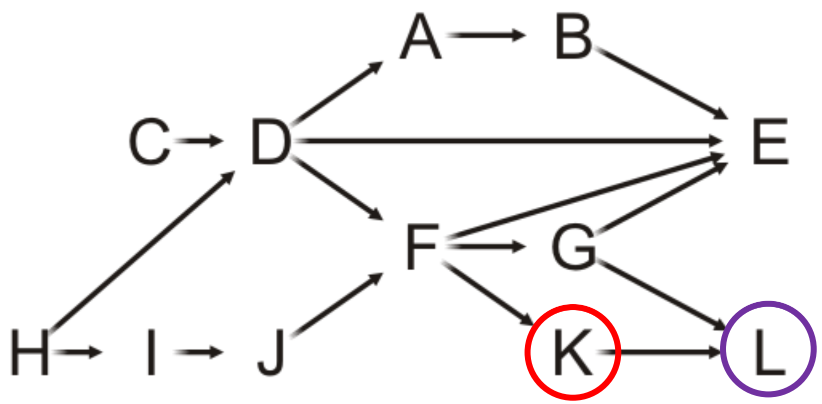
Queue:	C	H	D	I	A	J	B	F	G	K	E	

A	0
B	0
C	0
D	0
E	0
F	0
G	0
H	0
I	0
J	0
K	0
L	1

Example

Pop the front of the queue

- K has one neighbors: L



Queue:

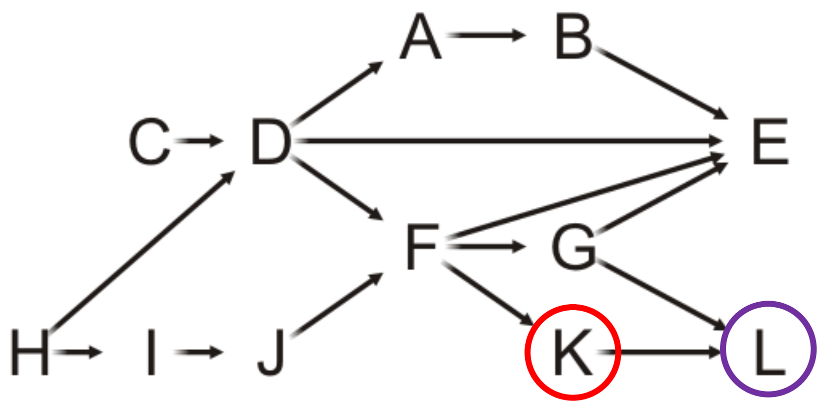
C	H	D	I	A	J	B	F	G	K	E	

A	0
B	0
C	0
D	0
E	0
F	0
G	0
H	0
I	0
J	0
K	**0**
L	**1**

Example

Pop the front of the queue

- K has one neighbors: L
- Decrement its in-degree



Queue:

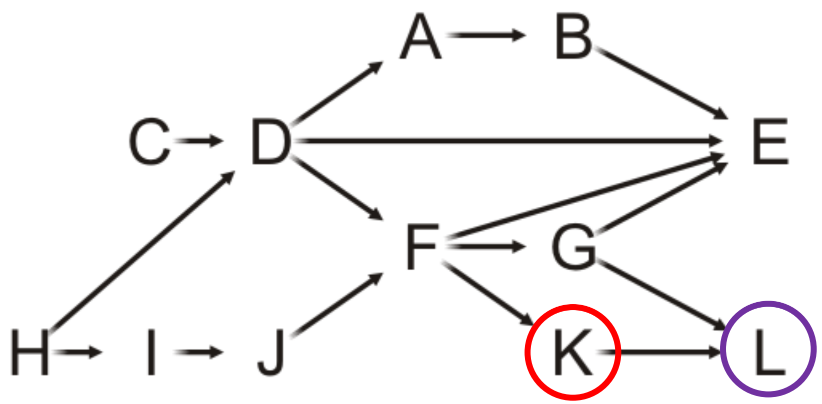
C	H	D	I	A	J	B	F	G	K	E	

A	0
B	0
C	0
D	0
E	0
F	0
G	0
H	0
I	0
J	0
K	**0**
L	**0**

Example

Pop the front of the queue

- K has one neighbors: L
- Decrement its in-degree
  - L is decremented to zero, so push it onto the queue



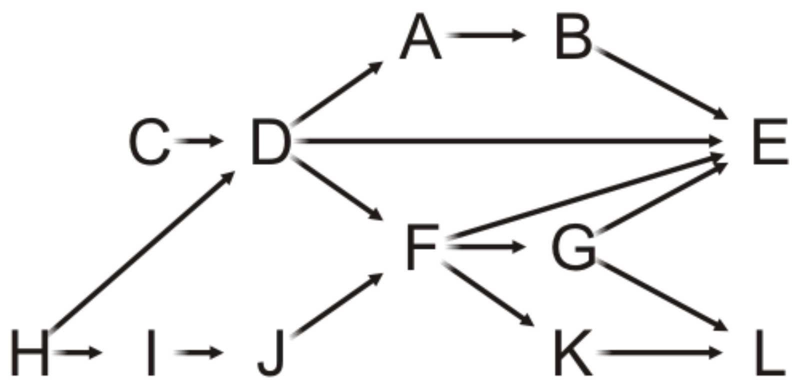
Queue:

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

A	0
B	0
C	0
D	0
E	0
F	0
G	0
H	0
I	0
J	0
K	**0**
L	**0**

Example

Pop the front of the queue



Queue:

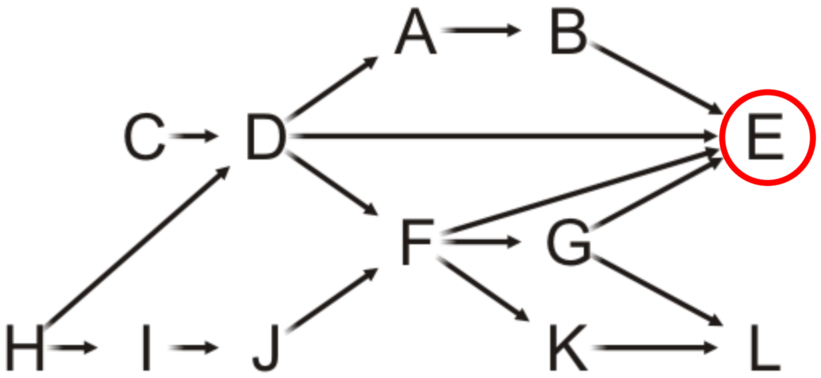
C	H	D	I	A	J	B	F	G	K	E	L

A	0
B	0
C	0
D	0
E	0
F	0
G	0
H	0
I	0
J	0
K	0
L	0

Example

Pop the front of the queue

- E has no neighbors—it is a *sink*



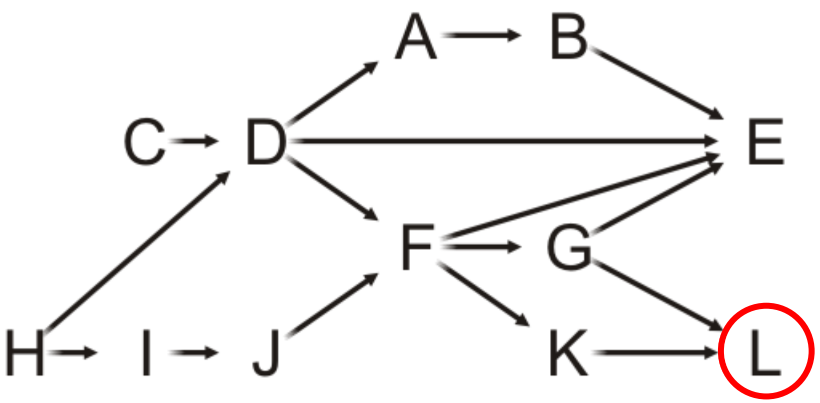
Queue:

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

A	0
B	0
C	0
D	0
E	**0**
F	0
G	0
H	0
I	0
J	0
K	0
L	0

Example

Pop the front of the queue



Queue:

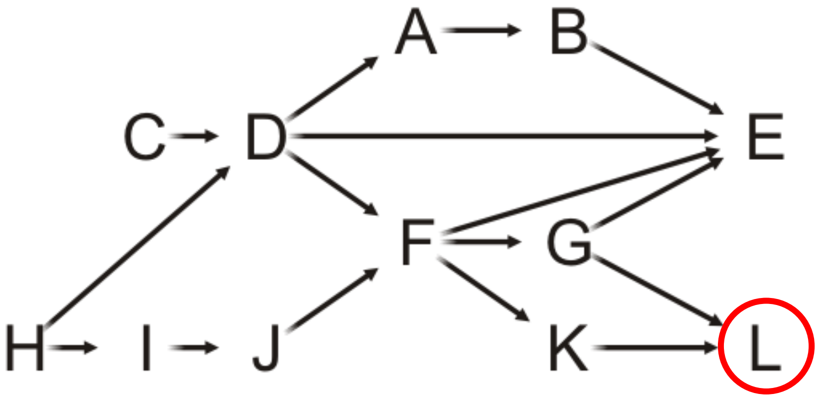
C	H	D	I	A	J	B	F	G	K	E	L

A	0
B	0
C	0
D	0
E	0
F	0
G	0
H	0
I	0
J	0
K	0
L	0

Example

Pop the front of the queue

- L has no neighbors—it is also a *sink*



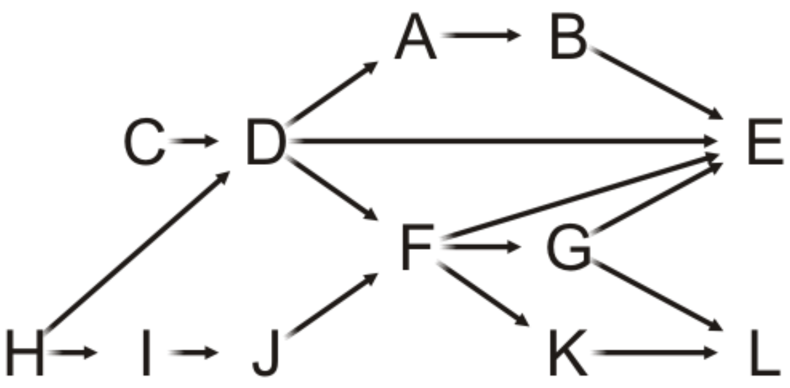
Queue:

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

A	0
B	0
C	0
D	0
E	0
F	0
G	0
H	0
I	0
J	0
K	0
L	**0**

Example

The queue is empty, so we are done



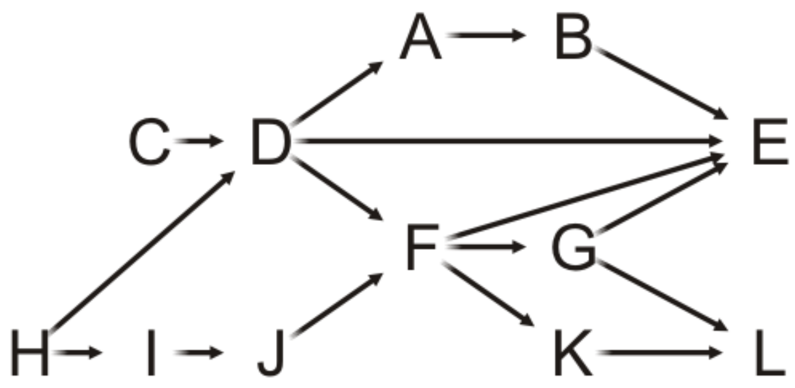
Queue:

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

A	0
B	0
C	0
D	0
E	0
F	0
G	0
H	0
I	0
J	0
K	0
L	0

Example

The enqueue order is the topological sorting

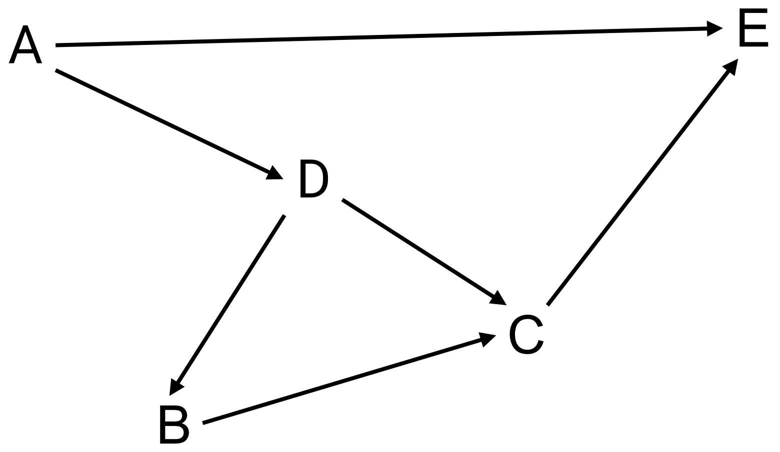


C	H	D	I	A	J	B	F	G	K	E	L

A	0
B	0
C	0
D	0
E	0
F	0
G	0
H	0
I	0
J	0
K	0
L	0

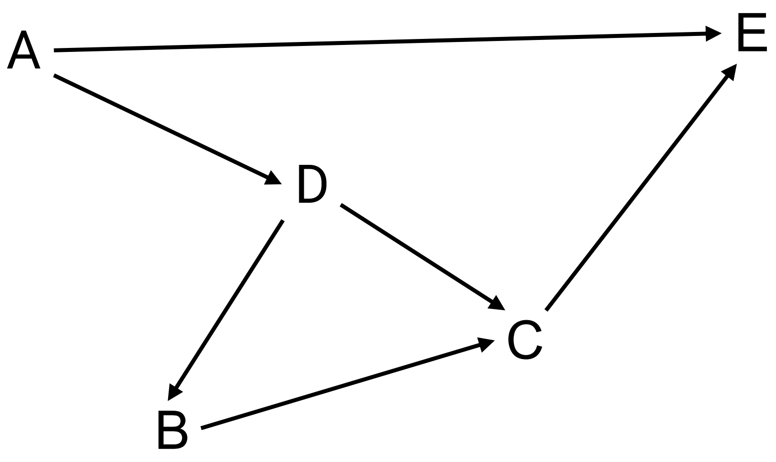
Exercise

Can you compute the topological sort of the following graph?



Exercise

Initialize the array of in-degrees and the queue



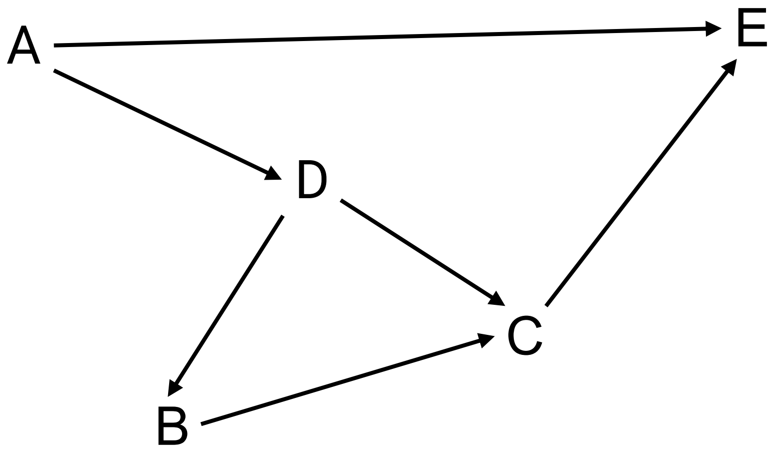
A	0
B	1
C	2
D	1
E	2

Queue:

The queue is empty

Exercise

Push A onto the queue



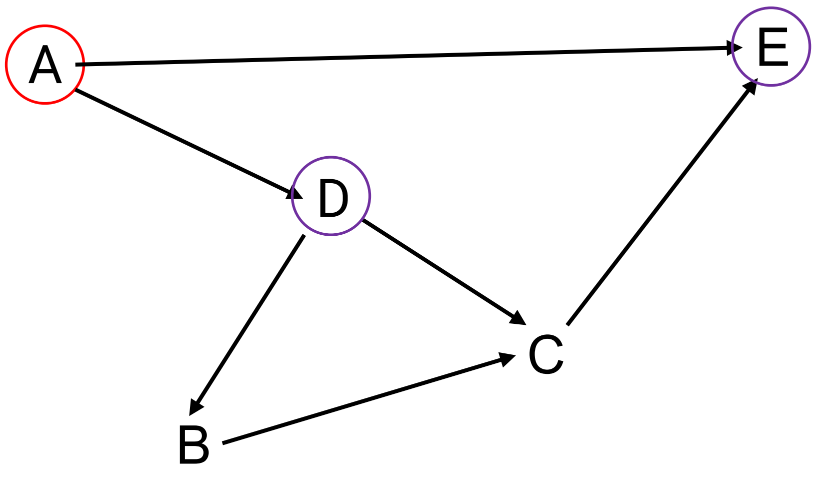
A	**0**
B	1
C	2
D	1
E	2

Queue:	A				

Exercise

Pop the front of the queue

– A has two neighbors: D and E



A	**0**
B	1
C	2
D	**1**
E	**2**

Queue:

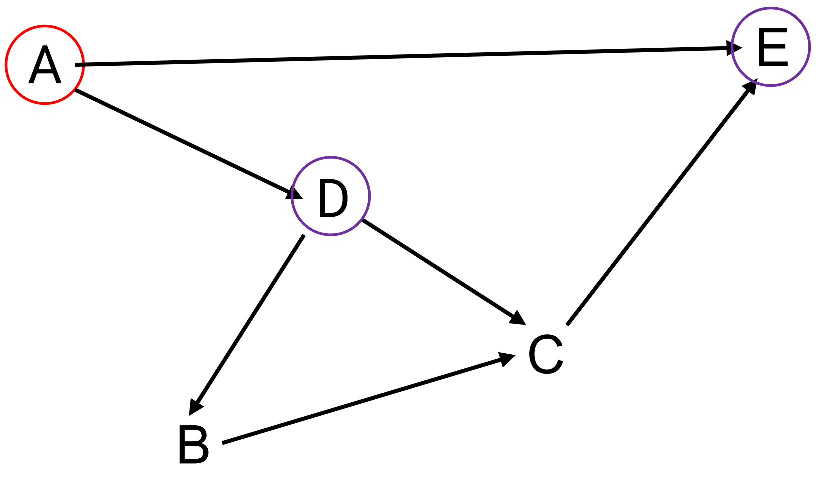
A				

Exercise

Pop the front of the queue

– A has two neighbors: D and E

– Decrement their in-degree



A	0
B	1
C	2
D	0
E	1

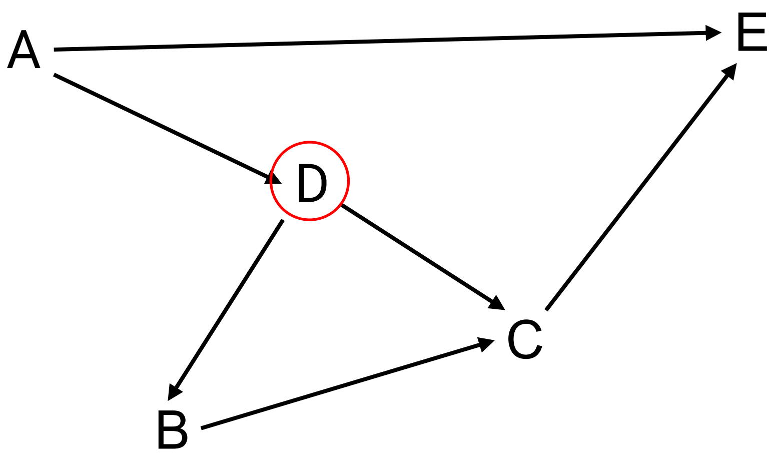
Queue:

A				

Exercise

Pop the front of the queue

- A has two neighbors: D and E
- Decrement their in-degree



A	0
B	1
C	2
D	0
E	1

Queue:

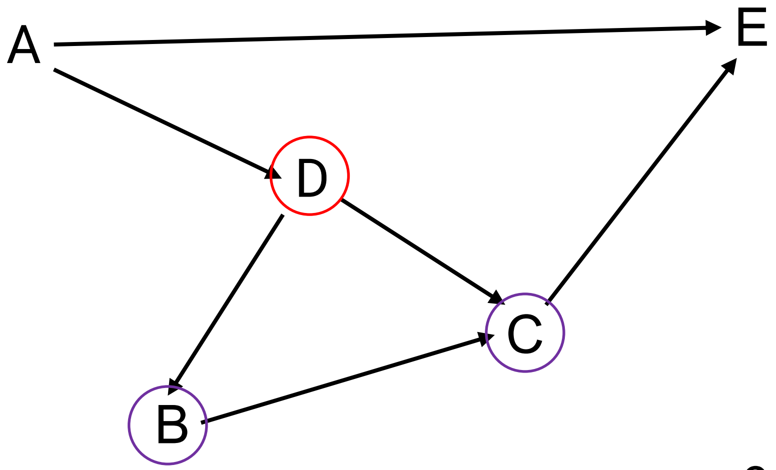
A	D			

D is decremented to zero, so push it onto the queue

Exercise

Pop the front of the queue

– D has two neighbors: B and C



A	0
B	**1**
C	**2**
D	**0**
E	1

Queue:

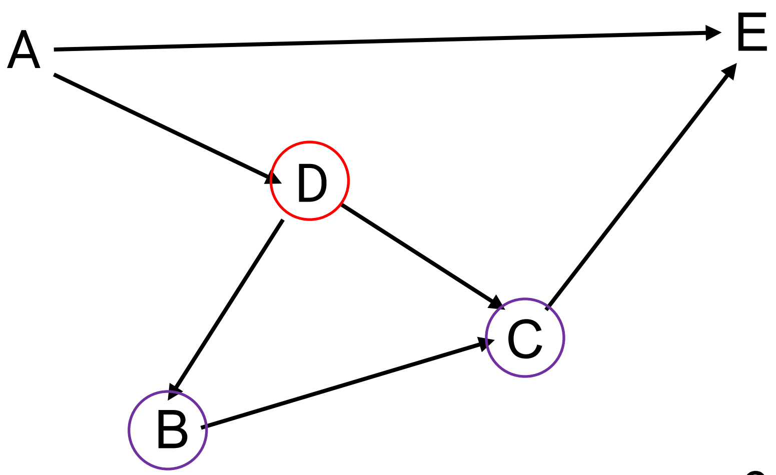
A	D			

Exercise

Pop the front of the queue

– D has two neighbors: B and C

– Decrement their in-degree



A	0
B	**0**
C	**1**
D	**0**
E	1

Queue:

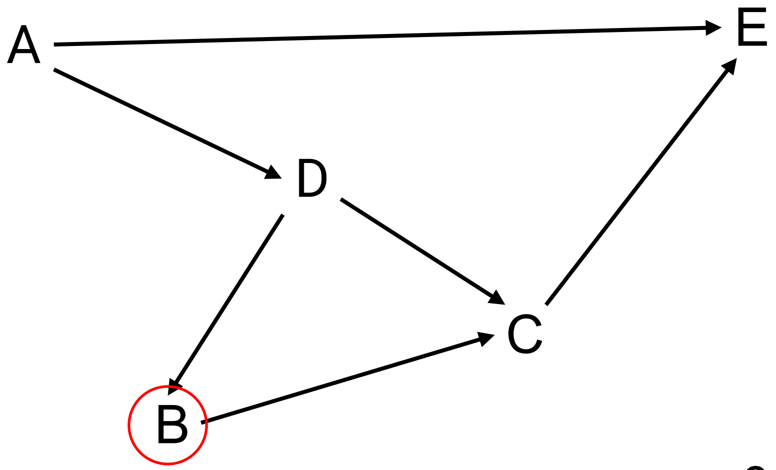
A	D			

Exercise

Pop the front of the queue

– D has two neighbors: B and C

– Decrement their in-degree



A	0
B	**0**
C	**1**
D	**0**
E	1

Queue:

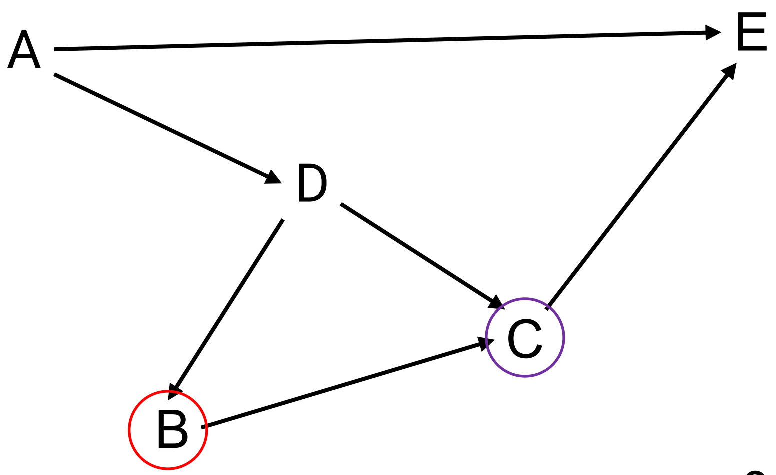
A	D	B		

B is decremented to zero, so push it onto the queue

Exercise

Pop the front of the queue

– B has one neighbor: C



A	0
B	**0**
C	**1**
D	0
E	1

Queue:

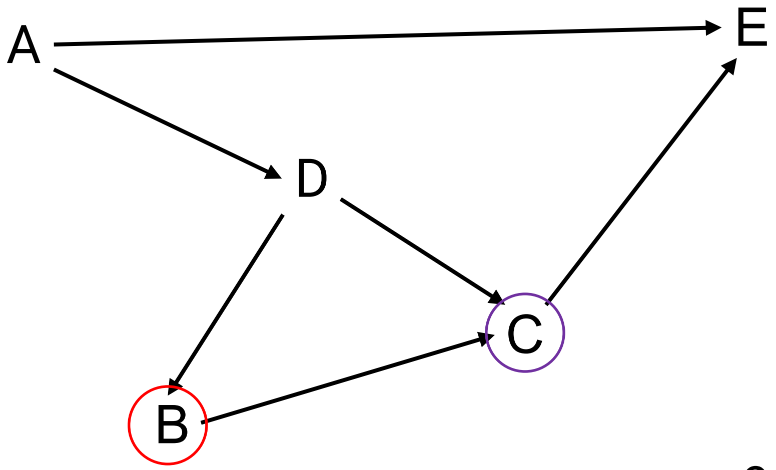
A	D	B		

Exercise

Pop the front of the queue

– B has one neighbor: C

– Decrement its in-degree



A	0
B	**0**
C	**0**
D	0
E	1

Queue:

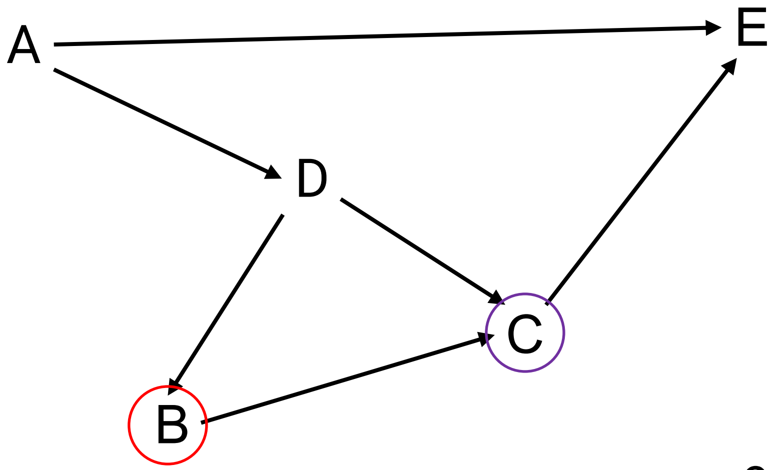
A	D	B		

Exercise

Pop the front of the queue

– B has one neighbor: C

– Decrement its in-degree



A	0
B	**0**
C	**0**
D	0
E	1

Queue:

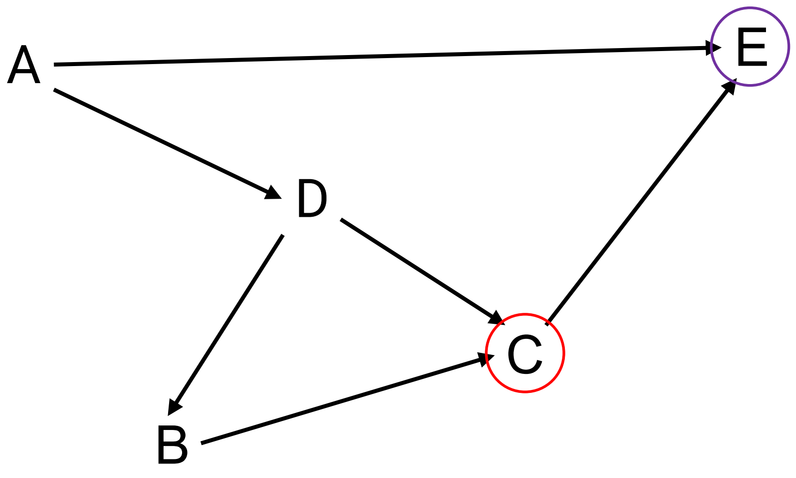
A	D	B	C	

C is decremented to zero, so push it onto the queue

Exercise

Pop the front of the queue

– C has one neighbor: E



A	0
B	0
C	**0**
D	0
E	**1**

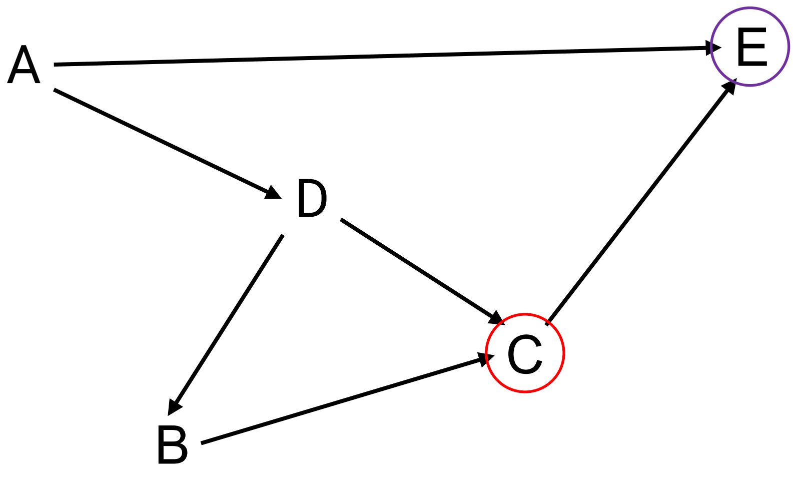
Queue:

A	D	B	C	

Exercise

Pop the front of the queue

- C has one neighbor: E
- Decrement its in-degree



A	0
B	0
C	0
D	0
E	0

Queue:

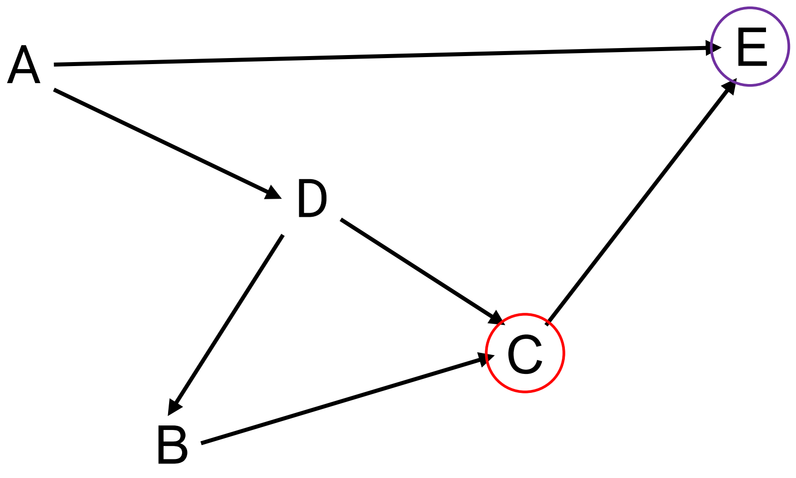
A	D	B	C	

Exercise

Pop the front of the queue

– C has one neighbor: E

– Decrement its in-degree



A	0
B	0
C	0
D	0
E	0

Queue:

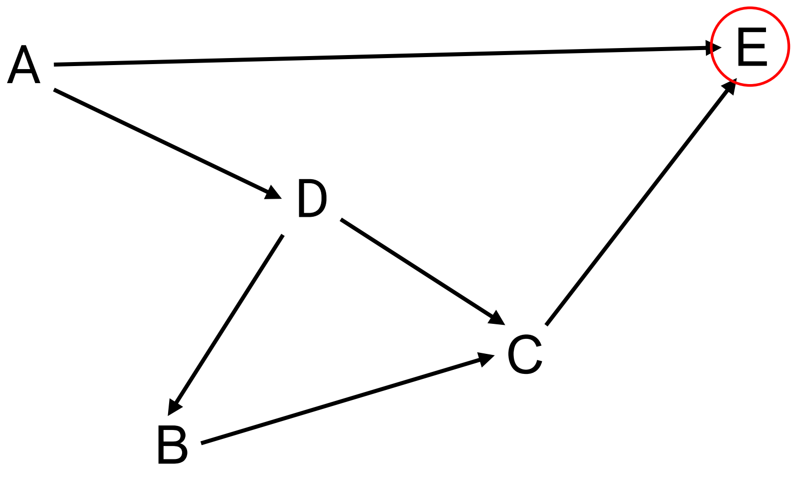
A	D	B	C	E

E is decremented to zero, so push it onto the queue

Exercise

Pop the front of the queue

– E has no neighbors



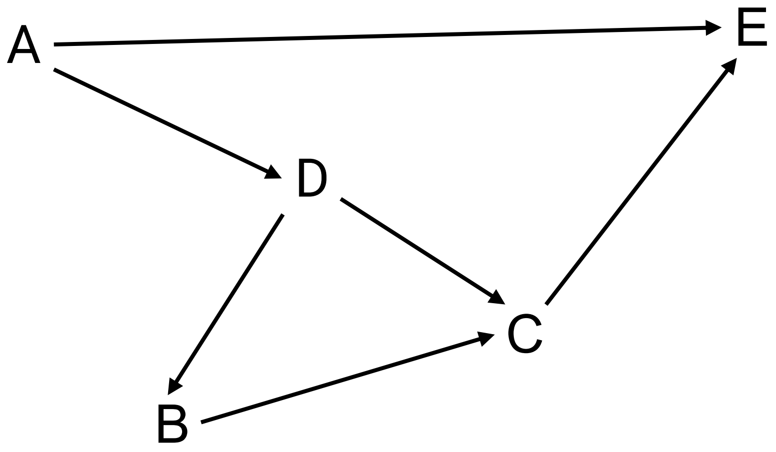
A	0
B	0
C	0
D	0
E	**0**

Queue:

A	D	B	C	E

Exercise

The queue is empty, so we are done



A	0
B	0
C	0
D	0
E	0

Queue:

A	D	B	C	E

Learning outcomes

- Understand the BFS and DFS algorithms
- Understand the algorithms for computing connected components (using BFS and Disjoint-set)
- Know the concept of topological sort and how to compute it