Today’s topics

- Topological sort
- Bipartite graph
- Strongly connected components

References

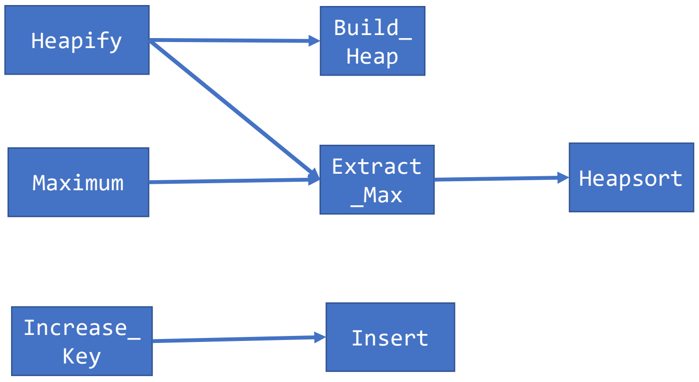
- Read chapter-20 of the CLRS book
- Read chapter-6 from Goodrich and Tamassia book
- https://en.wikipedia.org/wiki/Depth-first_search
- https://en.wikipedia.org/wiki/Bipartite_graph

Topological sort

Example

- Build_Max_Heap
- Max_Heap_Extract_Max
- Max_Heap_Insert
- Max_Heap_Maximum
- Max_Heapify
- Heapsort
- Max_Heap_Increase_Key

Example

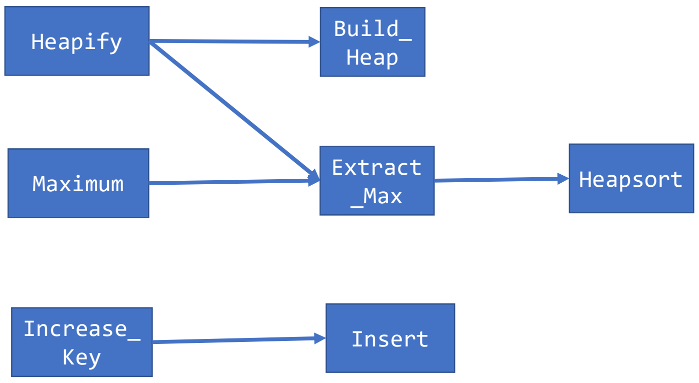


Is this a valid topological order?

Heapify, Build_Heap, Extract_Max, Heapsort, Maximum, Increase_Key, Insert

No

Example

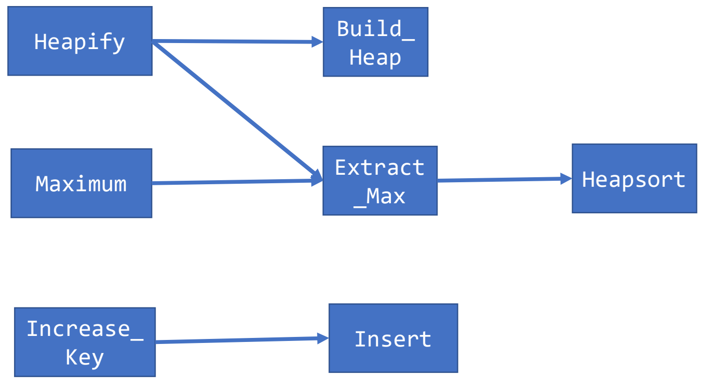


Is this a valid topological order?

Heapify, Maximum, Build_Heap, Extract_Max, Heapsort, Increase_Key, Insert

yes

Example

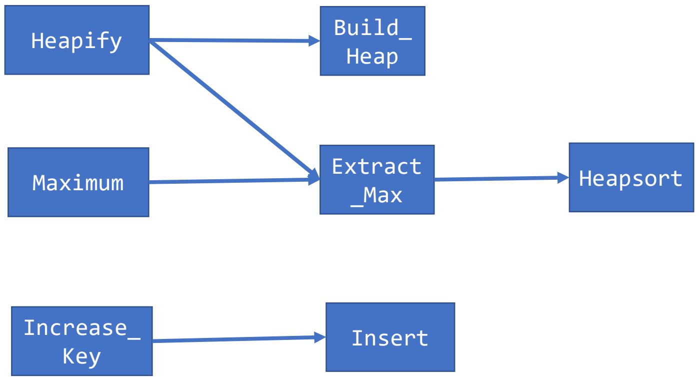


Is this a valid topological order?

Maximum, Heapify, Build_Heap, Extract_Max, Heapsort, Increase_Key, Insert

Yes

Example

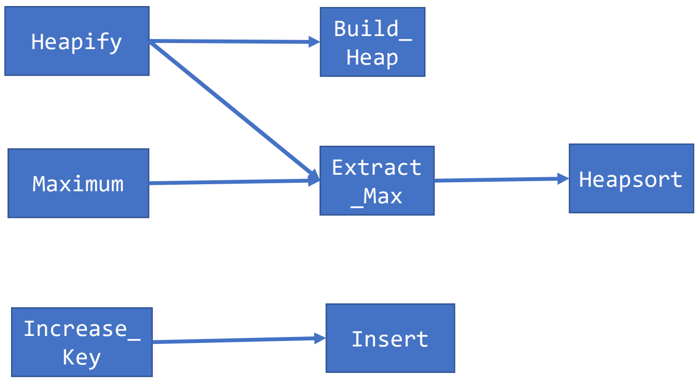


Is this a valid topological order?

Increase_Key, Maximum, Heapify, Build_Heap, Extract_Max, Heapsort, Insert

Yes

Example



Is this a valid topological order?

Increase_Key, Insert, Maximum, Heapify, Build_Heap, Extract_Max, Heapsort

yes

Example

```
int bar(int m, int n) {
  if (m == n)
    return m;
  if (m < n)
     return foo(m, n);
  return bar(m-2, n+1);
}

int foo(int m, int n) {
   if (m == n)
     return m;
   if (m > n)
     return bar(m, n);
   return foo(m+1, n-1);
}
```

What is the dependency graph?

$$\begin{tikzcd} \text{foo} \arrow[r, bend left] & \text{bar} \arrow[l, bend left] \end{tikzcd}$$

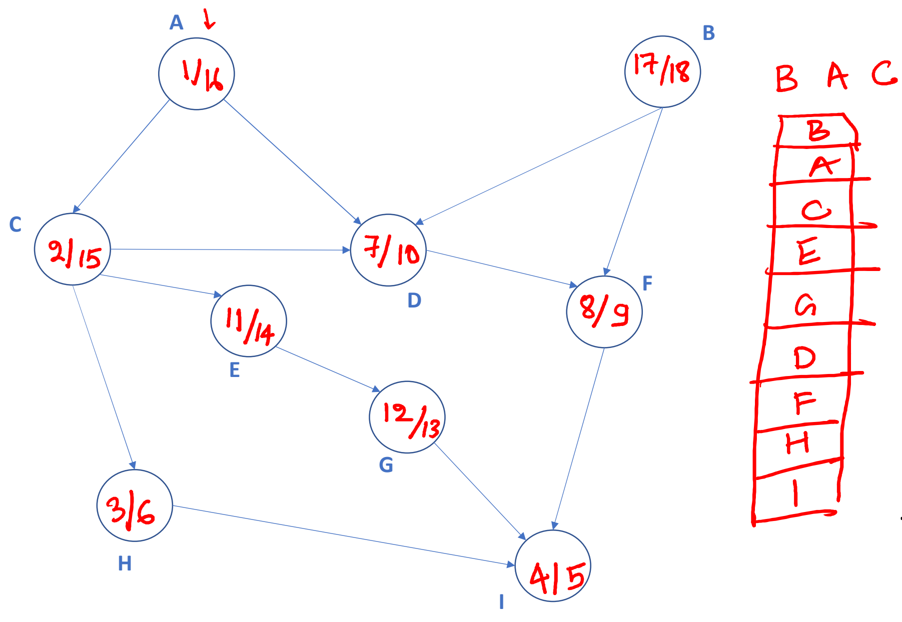
If there is a cycle, a topological sorted order is not possible.

Applications

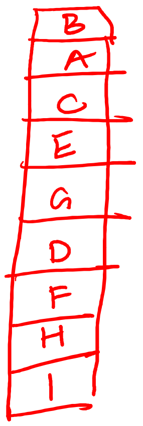
- Job scheduling
- Creating a course plan for a set of courses with prerequisites

Topological sort

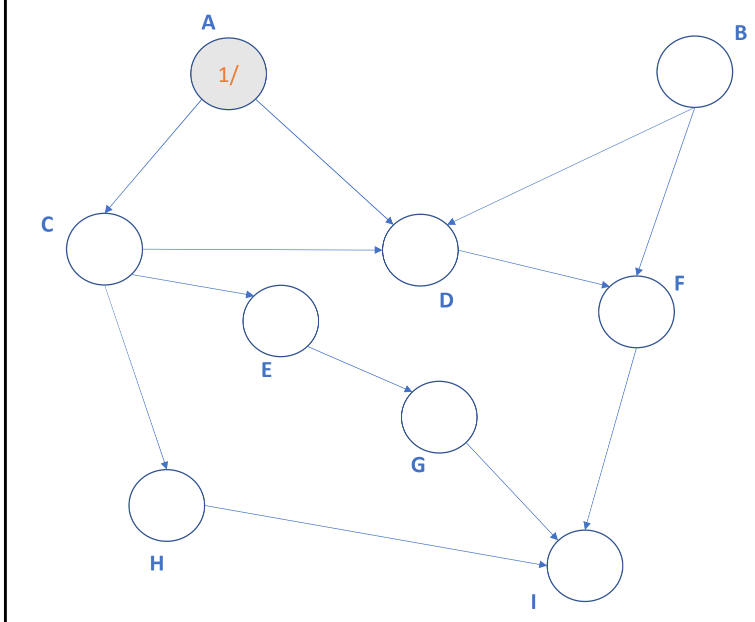
- A directed graph is acyclic if it doesn't have any cycle
- Topological sort is the ordering of vertices in a directed acyclic graph G such that if (u, v) is an edge in G, then u appears before v in the ordering
  - Notice that no such ordering exists for an undirected graph or a directed cyclic graph



B A C E G D F H I

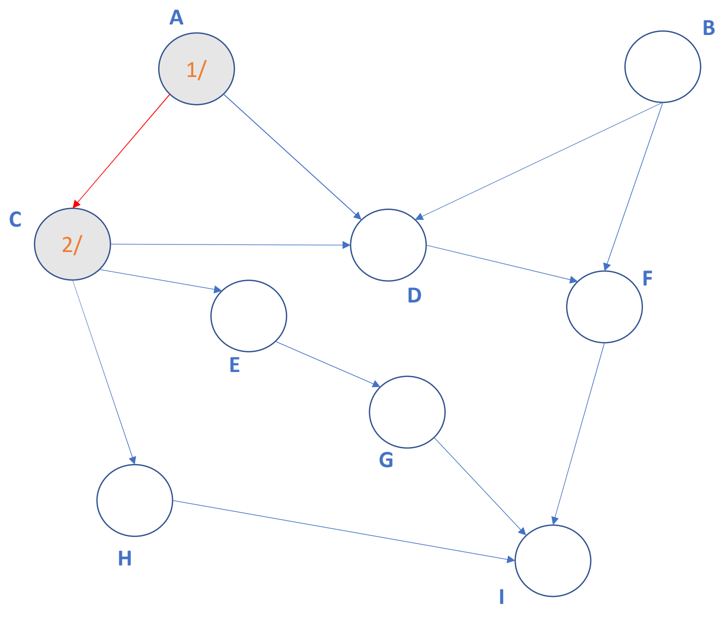


Topological sort

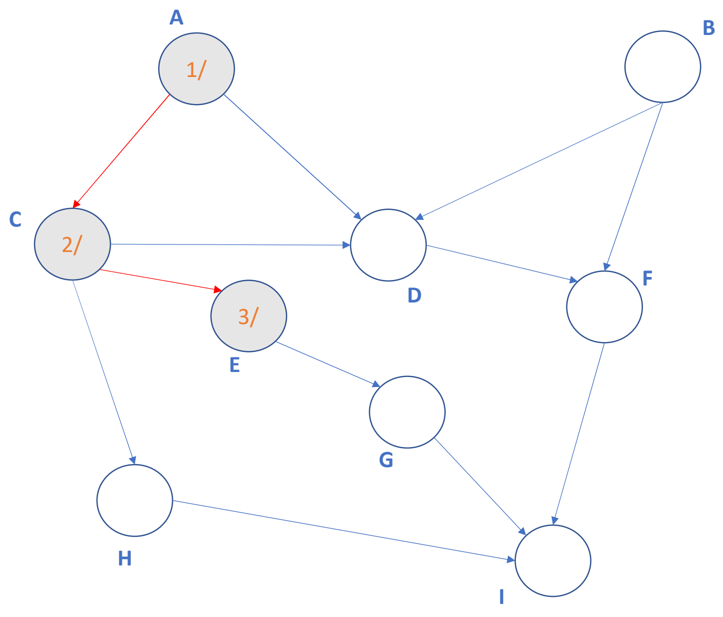


Topological sort

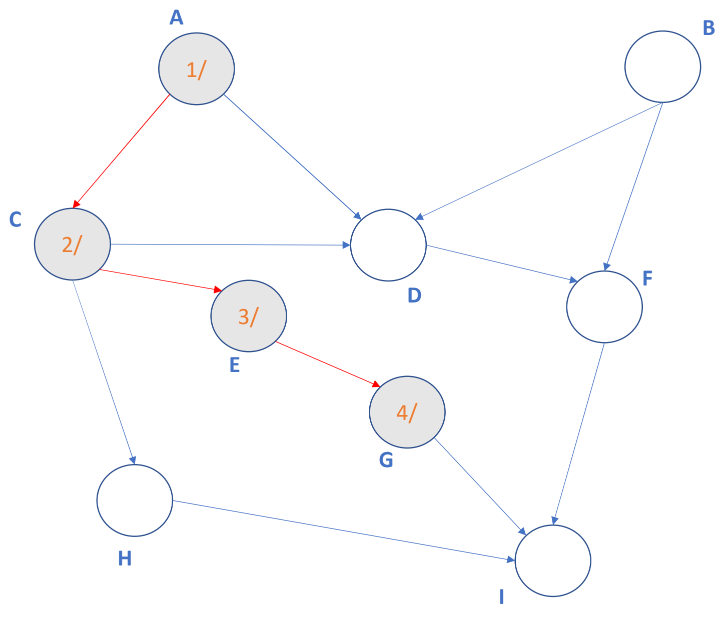
To do the topological sort, we can do DFS and then order the vertices in the reverse order of their finish time. Notice that in the DFS tree, the finish time of an ancestor is always larger than its decedents, and an ancestor will always come before in the topologically sorted order; therefore, printing the vertices in the reverse order of their finish time will always print an ancestor before its descendants. In this example, A is an ancestor of C and D in the DFS tree. But what if we start DFS from D. In that case, how the finish time of A will be larger than D? Notice that if we start from D, A will not appear in the DFS tree corresponding to D because A is not reachable from D. After doing a DFS at D when we iterate through the unvisited vertices, we will encounter A and start DFS at A. At this point also, the finish time of A is guaranteed to be more than D.



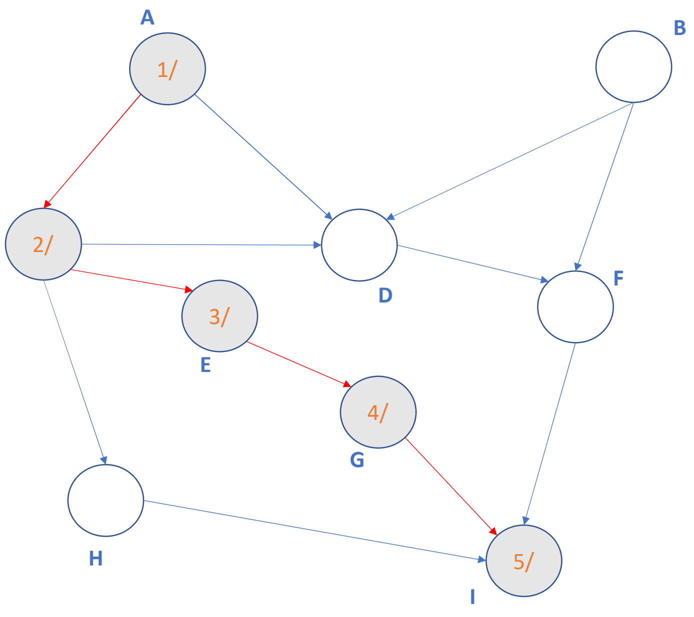
Topological sort



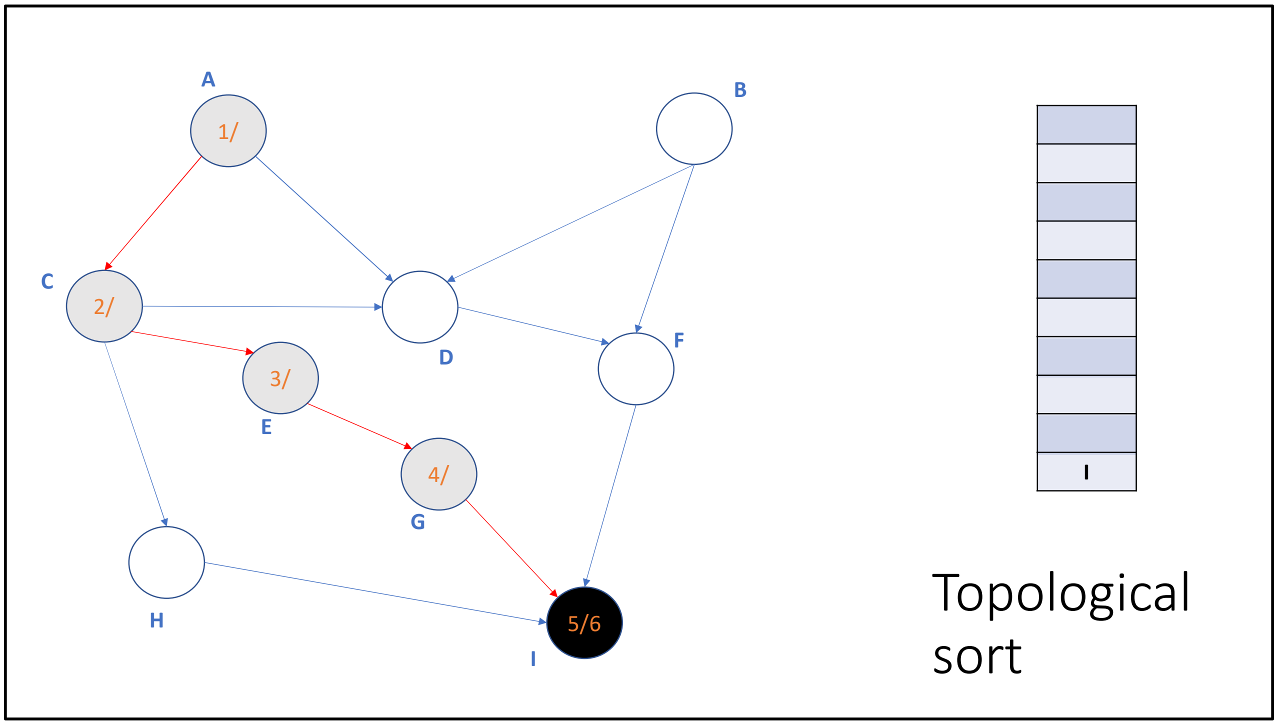
Topological sort



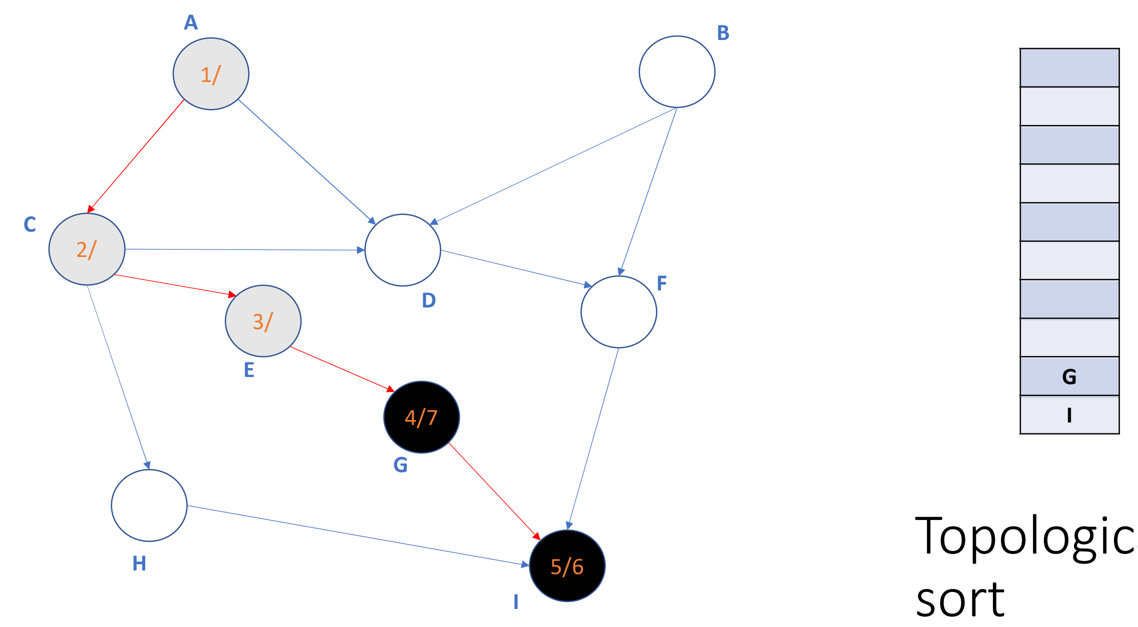
Topological sort



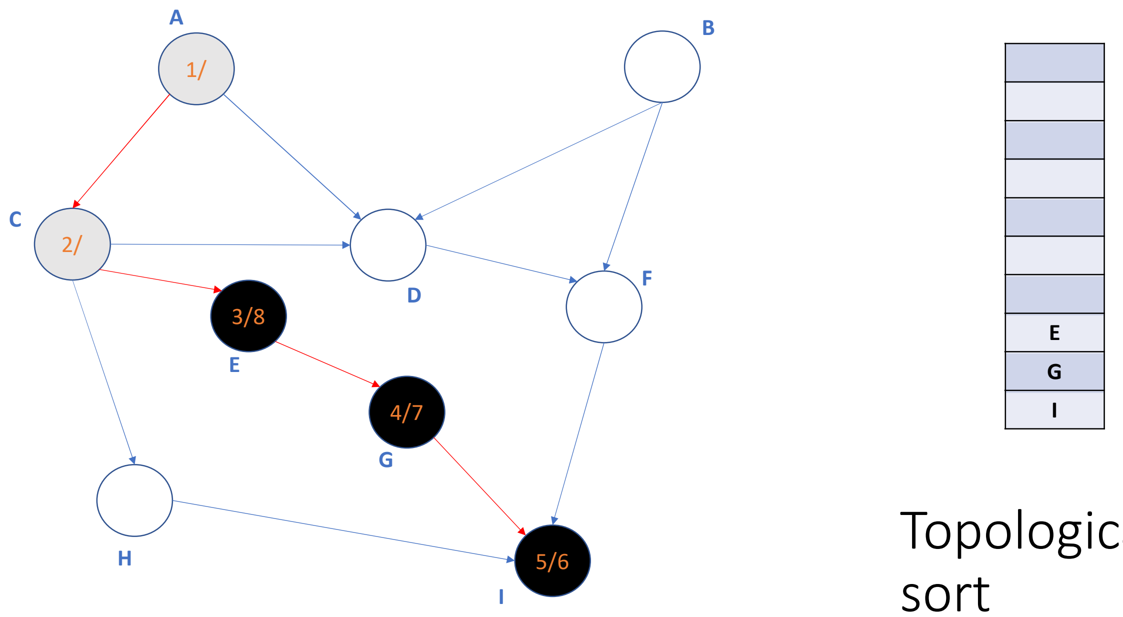
Topological sort



When the finish time is available (i.e., the vertex is marked as black), we add them to a stack. Because the finish time of vertices added later to the stack must have a higher finish time, when this algorithm terminates, the stack already contains all the vertices in the topological order, starting from top to bottom.

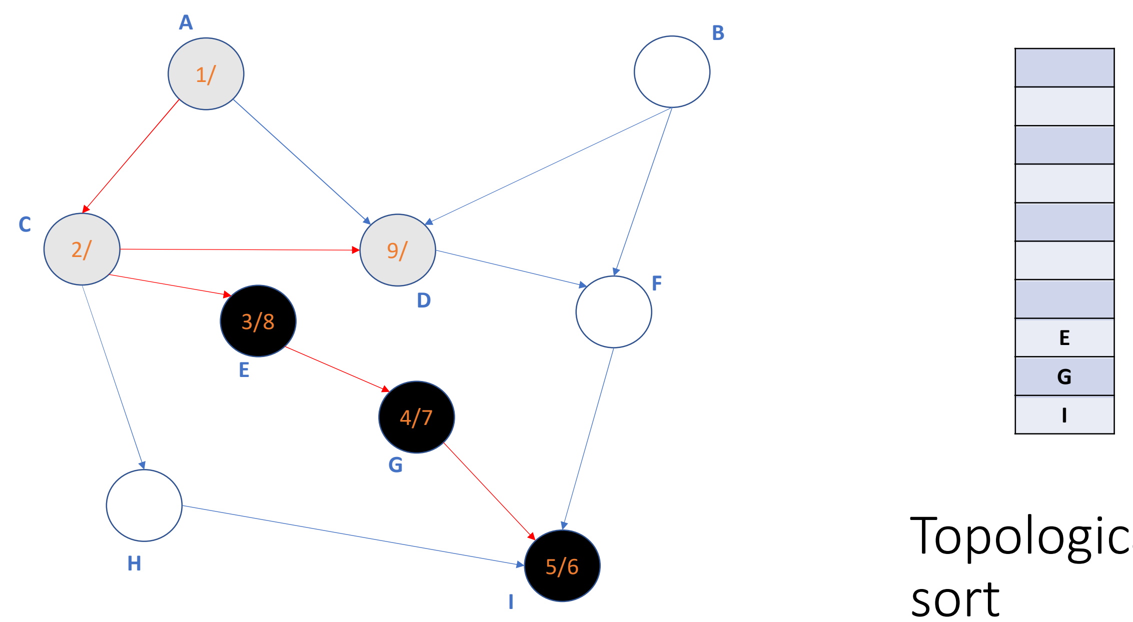


Topological sort

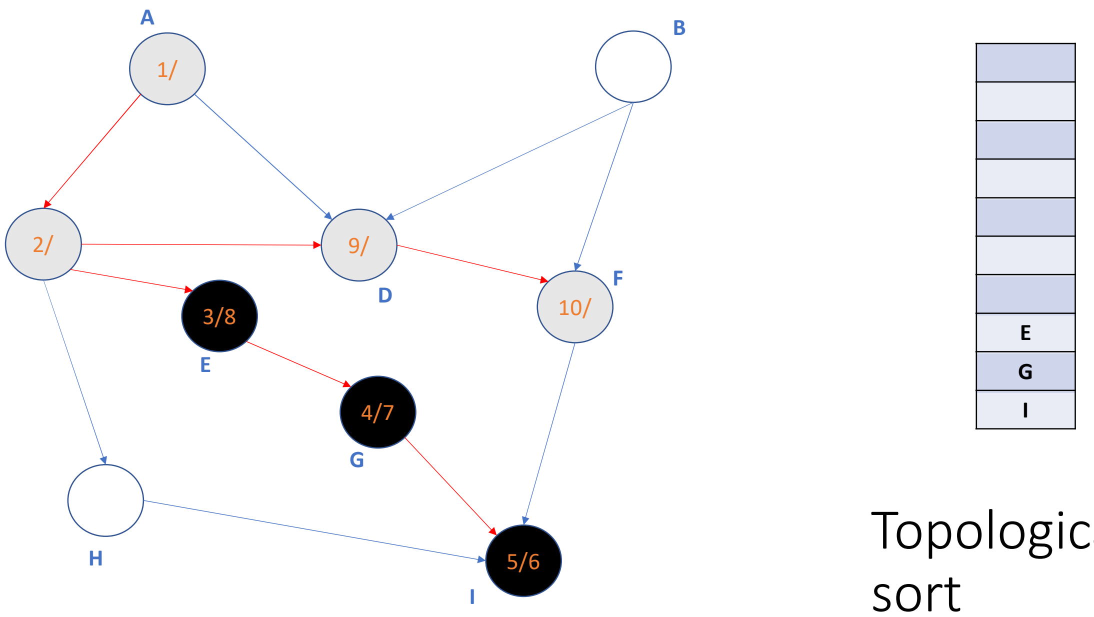


E
G
I

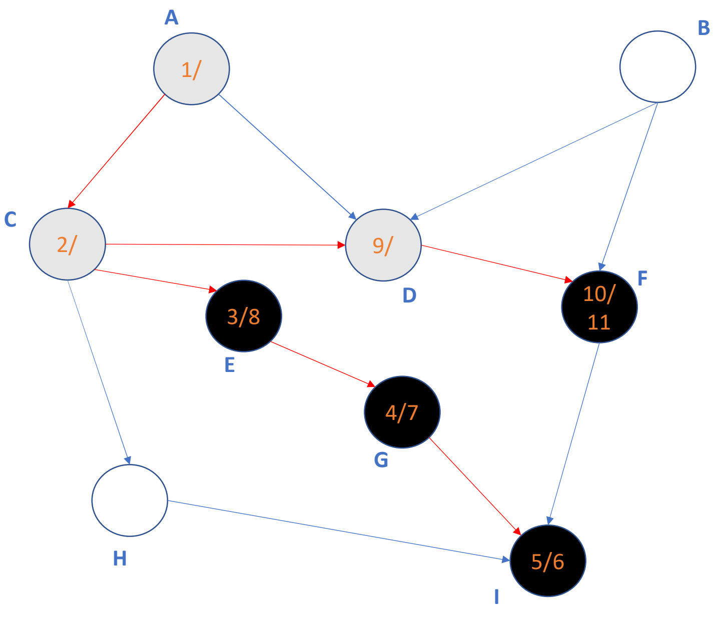
Topological sort

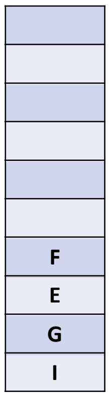


Topological sort

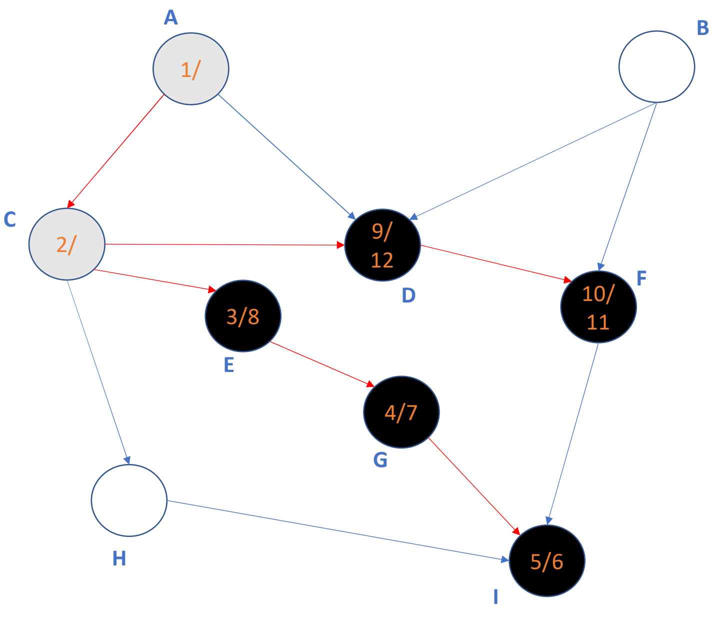


Topological sort



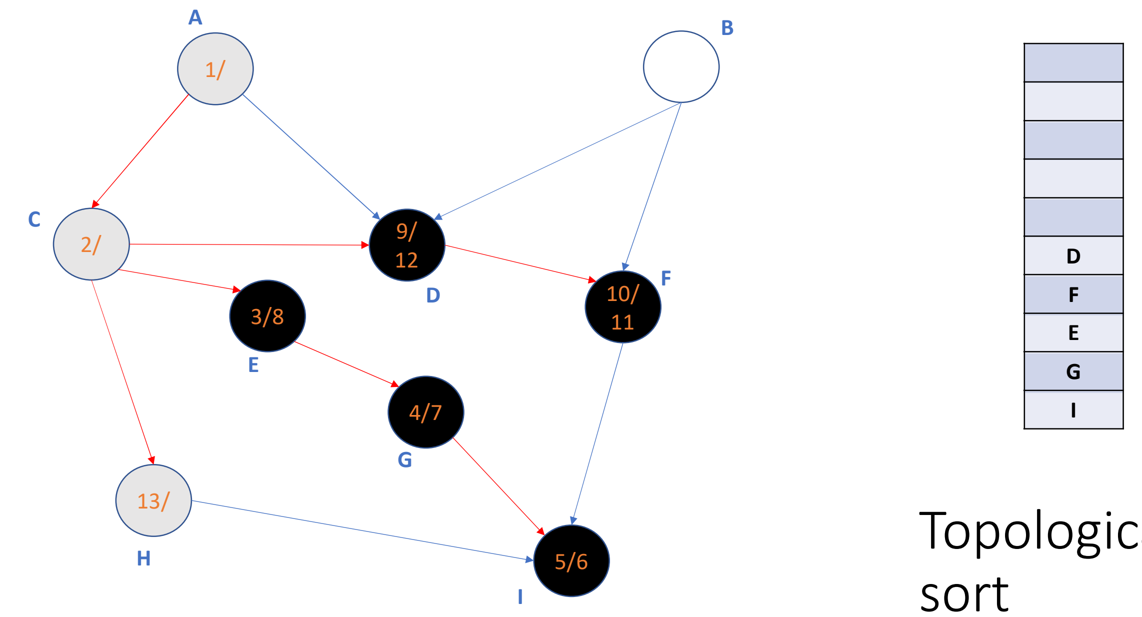


Topological sort

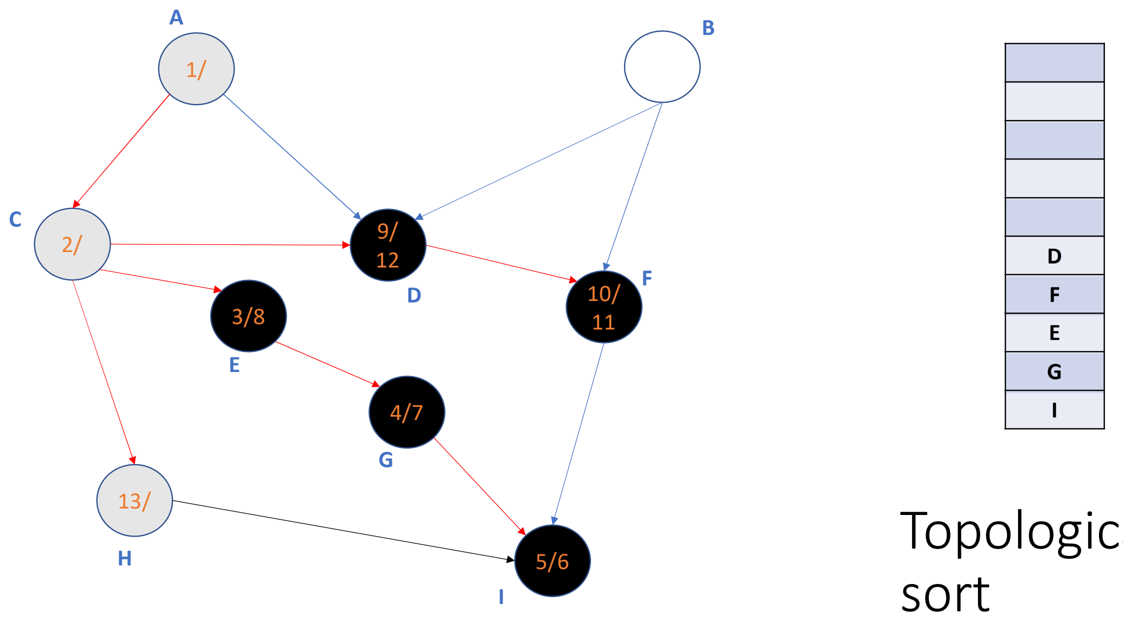


D
F
E
G
I

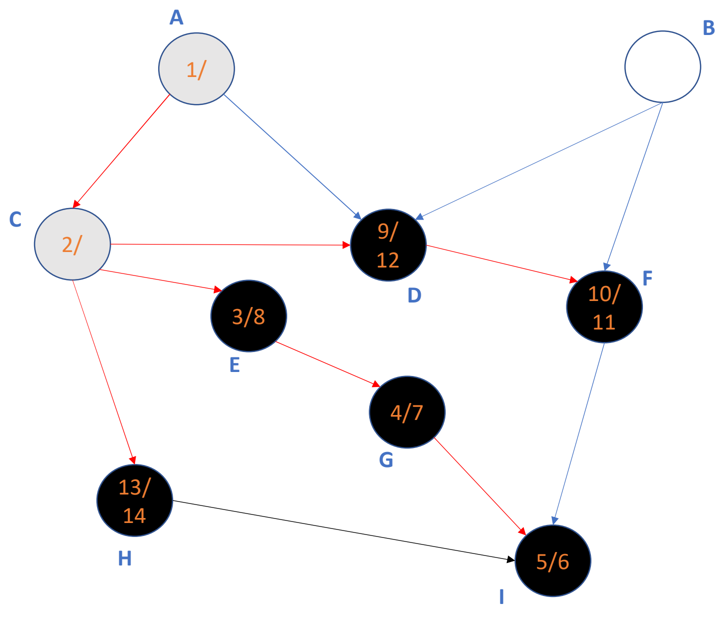
Topological sort



Topological sort

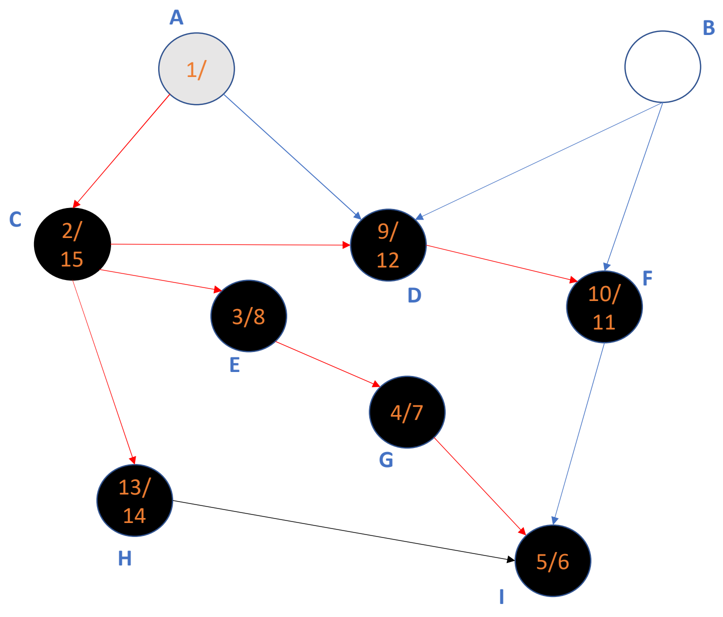


Topological sort



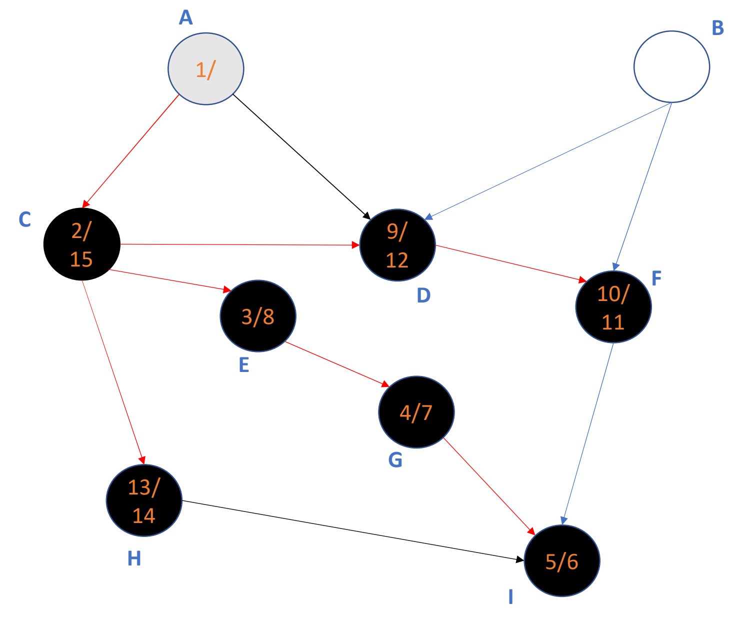
H
D
F
E
G
I

Topological sort



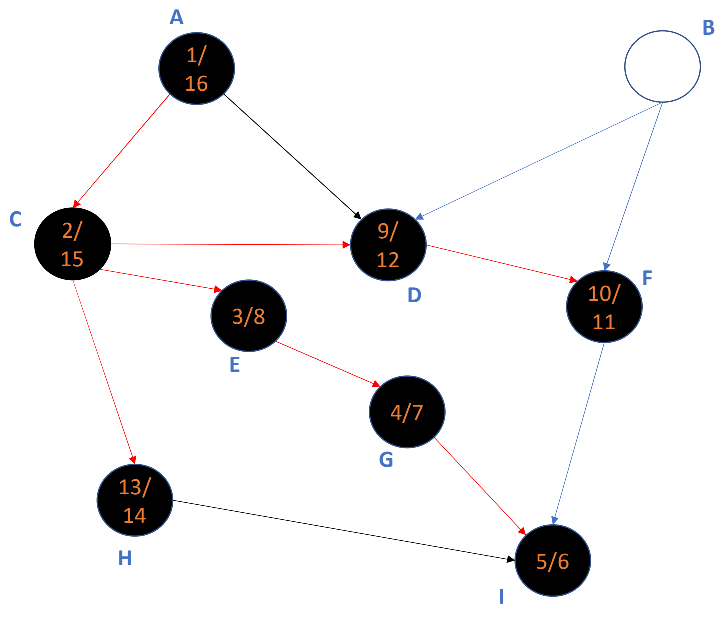
C
H
D
F
E
G
I

Topological sort



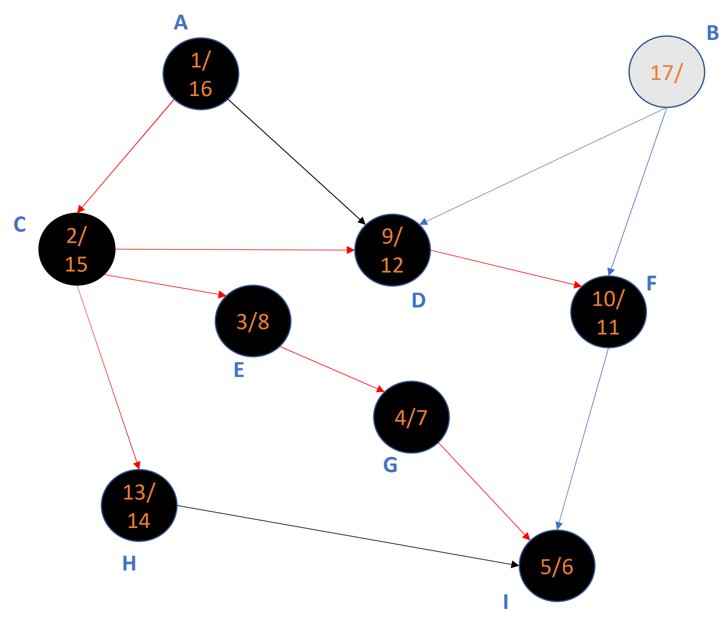
C
H
D
F
E
G
I

Topological sort



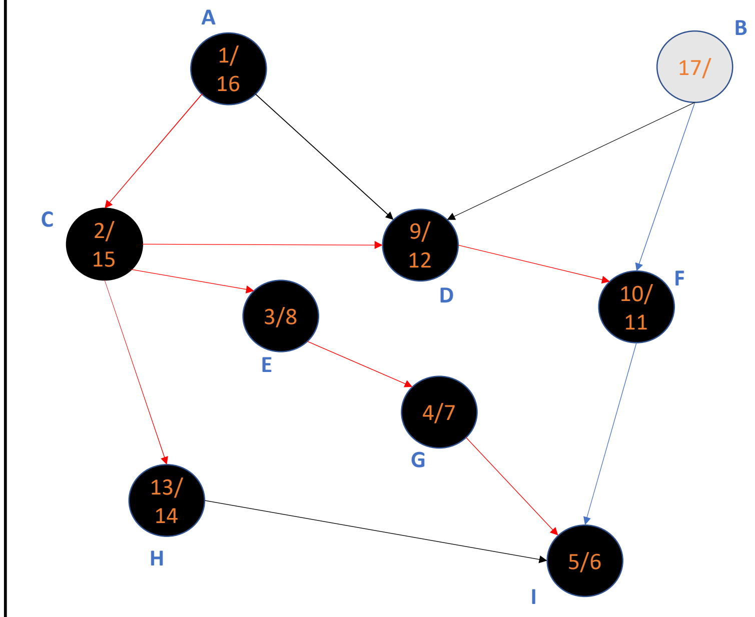
A
C
H
D
F
E
G
I

Topological sort



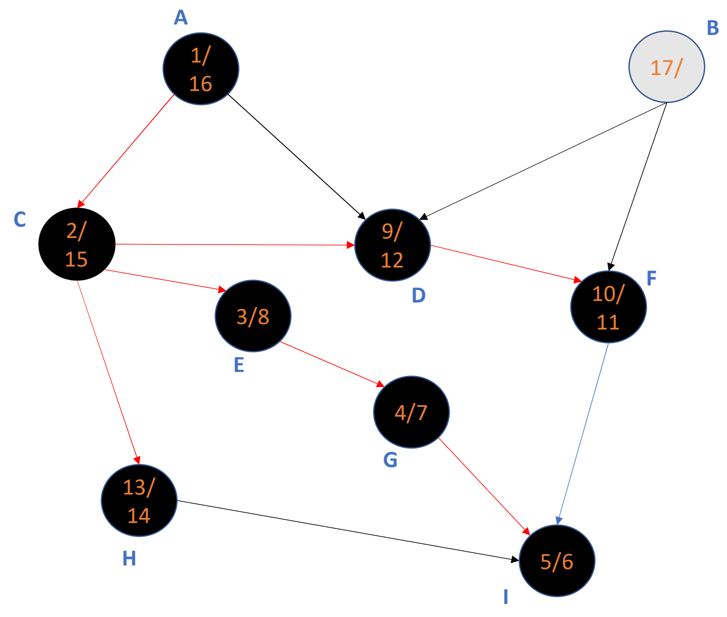
A
C
H
D
F
E
G
I

Topological sort



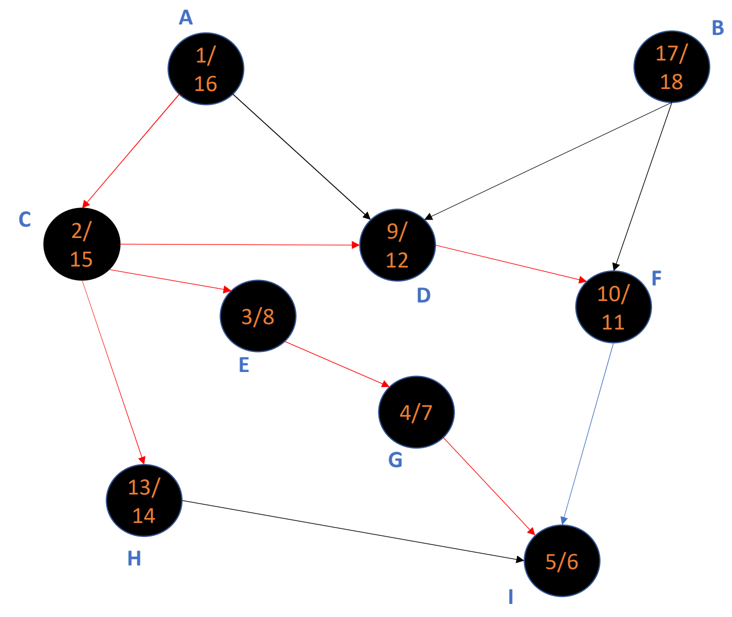
A
C
H
D
F
E
G
I

Topological sort



A
C
H
D
F
E
G
I

Topological sort



B
A
C
H
D
F
E
G
I

Topological sort

Topological sort

```
TOPOLOGICAL_SORT(G)
// Input graph G
// Output: A linked list that contains
// vertices in topological sorted order

   L = DFS_TSORT(G)
   // DFS_TSORT computes v.f for each vertex
   // and returns a list that contains
   // vertices in the decreasing order of
   // their finish time
```

DFS_TSORT

```
DFS_TSORT(G)

// G is a graph (V, E)
// s is the source vertex
// each vertex contains four
fields: color, d, f, π
// d is discovery time
// f is finish time
// π is predecessor in the DFS
forest

// Output: returns a list that
contains all vertices in the
decreasing order of their finish
time
```

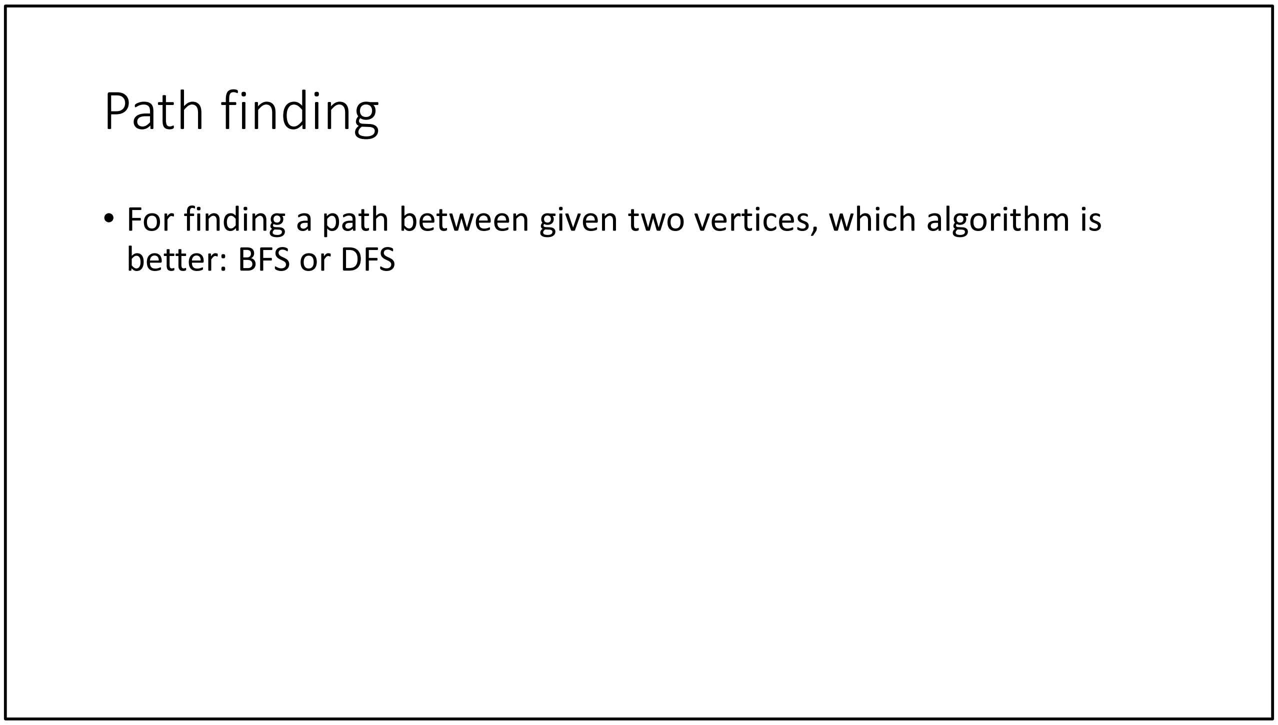
```
DFS_TSORT(G)
for each vertex u ∈ G.V
    u.color = WHITE
    u.π = NIL
time = 0
L = NIL
for each vertex u ∈ G.V
    if u.color == WHITE
       L = DFS-VISIT-TSORT(G, u, L)
return L

DFS-VISIT-TSORT(G, u, L)
time = time + 1
u.d = time
u.color = GRAY
for each vertex v ∈ G.Adj[u]
    if v.color == WHITE
       v.π = u
       L = DFS-VISIT-TSORT(G, v, L)
time = time + 1
u.f = time
u.color = BLACK
L = insert_front(L, u)
return L
```

In this modified algorithm, we add a vertex to the front of a linked list whenever its finish time is updated. Finally, this algorithm returns the linked list that stores the vertices in topologically sorted order.

Applications of BFS and DFS

Applications

- Bipartite graph
- Strongly connected component



Path finding

- For finding a path between given two vertices, which algorithm is better: BFS or DFS

DFS is better if the target vertex is far from the source vertex. If the target is near the source vertex, BFS might give a better result.

Bipartite graph

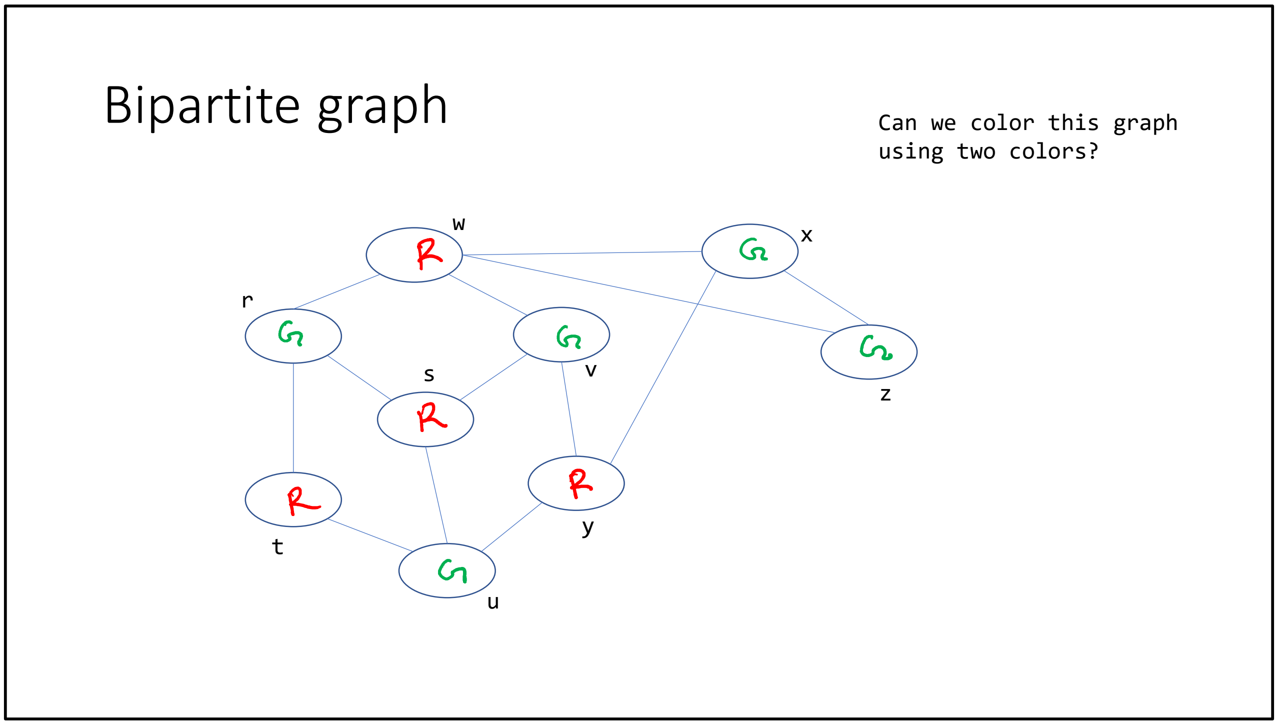
Bipartite graph

- https://en.wikipedia.org/wiki/Bipartite_graph

Bipartite graph

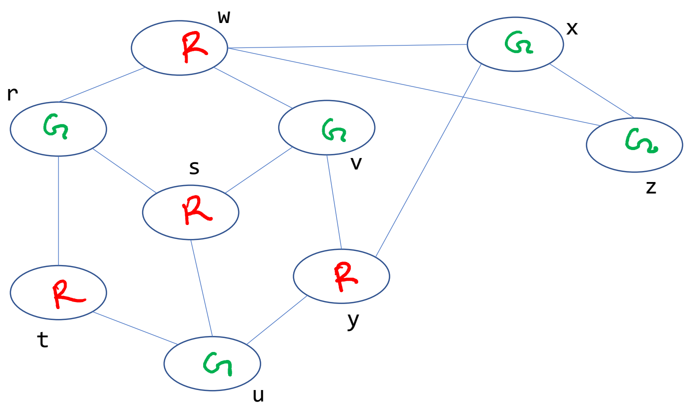
- A undirected connected graph is bipartite if the vertices can be partitioned into two sets, X and Y, such that all the edges have one endpoint in X and the other endpoint in Y

- Another way of looking at this problem is a graph coloring problem in which you need to color the graph using two colors in such a way that two adjacent vertices can’t have the same color



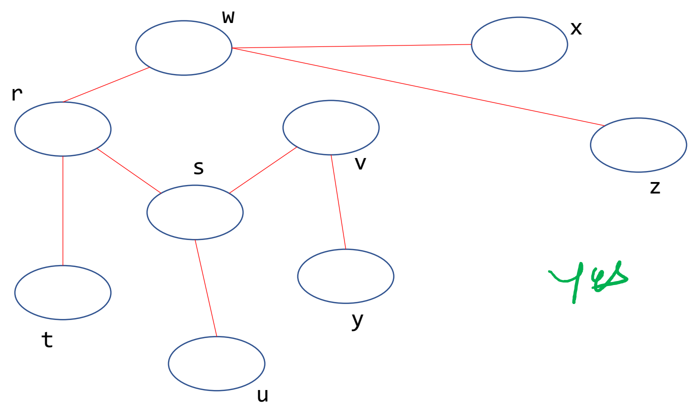
Bipartite graph

Can we color this graph
using two colors?



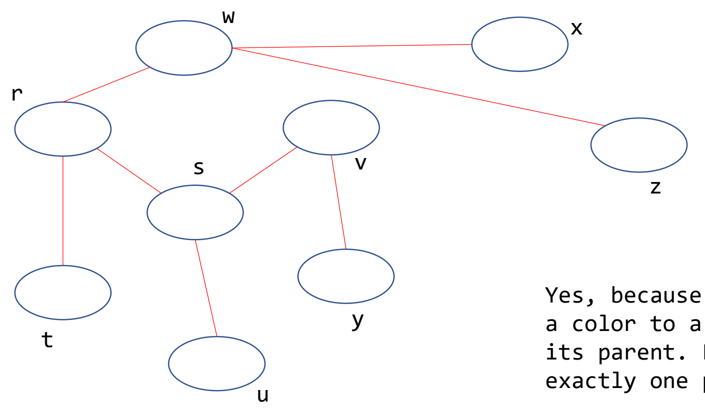
In the graph coloring problem, we want to assign a color to each vertex in such a way that two adjacent vertices do not have the same color. We cannot color this graph using two colors.

Can we color a tree using two colors



Yes

Can we color a tree using two colors

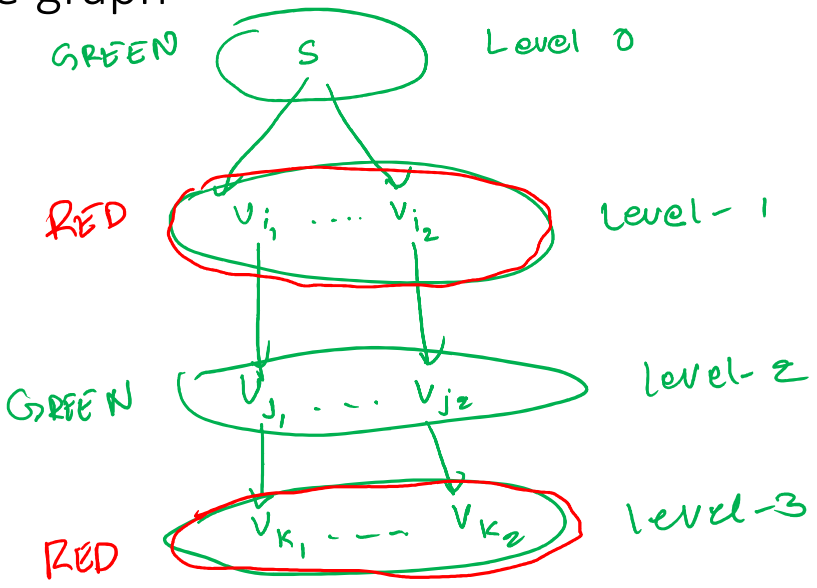


```
Yes, because we can always assign
a color to a child different from
its parent. Every child has
exactly one parent.
```

Bipartite graph

- BFS already partitions the vertices into different levels
- Can we check that a graph is bipartite by assigning colors to different levels?

Bipartite graph



We can assign alternate colors to alternate levels. Let's say we have two colors, green and red; we can assign green to level-0, red to level-1, green to level-2, and so on. If the edges are only between the adjacent levels, then the graph is bipartite. If the edges are between the vertices at the same level, then the graph is not bipartite. Notice that the endpoints of a tree edge are always between adjacent levels. A non-tree edge can be between vertices at adjacent levels or at the same level.

Bipartite graph

- We can assign different colors to adjacent levels, e.g.,
  - Level-0 can be assigned color-1
  - Level-1 can be assigned color-2
  - Level-2 can be assigned color-1
  - Level-3 can be assigned color-2
  - and so on
- The graph is bipartite if the endpoints of all edges are between adjacent levels only
- The graph is not bipartite if both endpoints of an edge belong to the same level

Tree edges

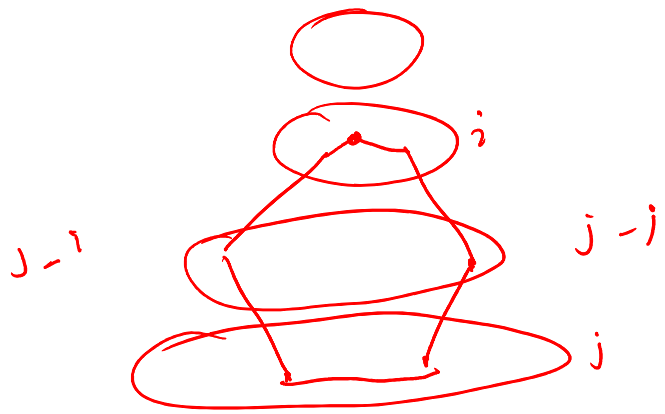
- Tree edges are always between adjacent levels, and therefore, the graph is always bipartite
  - We already discussed that a tree is always bipartite

Non-tree edge

- The endpoints of a non-tree edge can be between vertices at the same level or at an adjacent level
  - In both cases, it will create a cycle

Non-tree edge

- Is the total number of edges in the cycle created by a non-tree edge (u, v) with endpoints at the same level always odd?

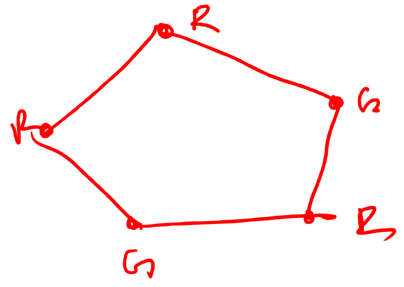


Non-tree edge

- Is the total number of edges in the cycle created by a non-tree edge (u, v) with endpoints at the same level always odd?
  - Let's consider a vertex s is at the lowest level out of all the vertices in the cycle
  - The length of the path from s to u and s to v must be the same, say L
  - Therefore, the number of edges in the cycle is 2 * L + 1, i.e., an odd number

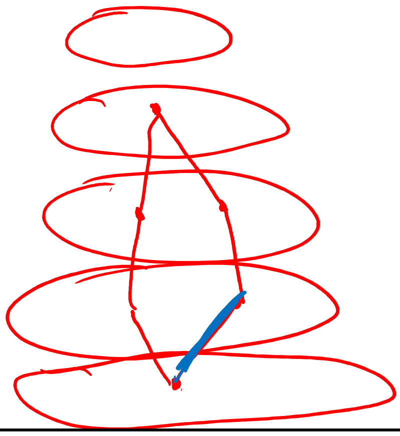
Non-tree edge

- We can’t color a cycle with an odd number of edges with two colors when the number of edges is more than two
  - Therefore, the graph is not bipartite



Non-tree edge

- Is the total number of edges in the cycle created by a non-tree edge (u, v) with endpoints at adjacent levels always even?

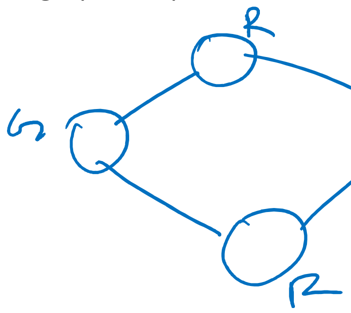


Non-tree edge

- Is the total number of edges in the cycle created by a non-tree edge (u, v) with endpoints at adjacent levels always even?
  - Let's consider s is at the lowest level out of all the vertices in the cycle
  - Therefore, the number of edges in the cycle is 2 * L, where L is the length of the path from s to v (assuming v is at the higher level)

Non-tree edge

- We can always color a cycle with an even number of edges with two colors
  - Therefore, the graph is bipartite



Bipartite graph

- How can we check if the endpoints of a non-tree edge are at the same level?

Bipartite graph

- How can we check if the endpoints of a non-tree edge are at the same level?
  - We can use the distance field in the vertex. Distance and level are the same.
  - If during BFS, we encounter a vertex v that has been visited and its distance from the source is the same as the current vertex u, this means that (u, v) is a non-tree edge whose endpoints are at the same level

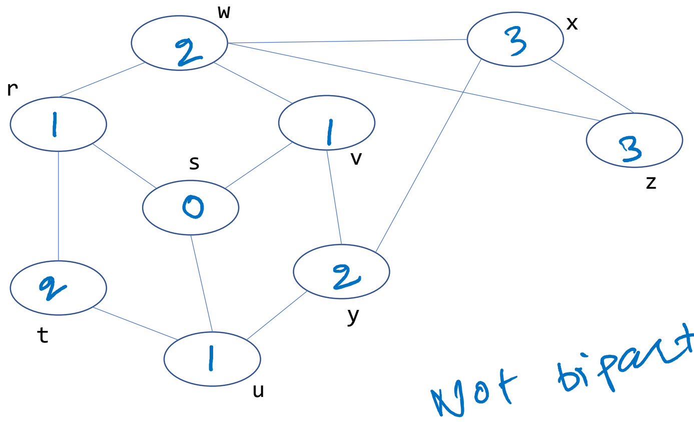
Bipartite graph

```
BFS_For_Bipartite(G, s)

// G is a graph (V, E)
// s is the source vertex
// each vertex contains three
fields, color, d, π

// Output: return 1 if the graph
is bipartite; otherwise, return 0
```

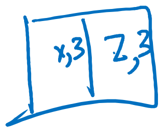
```
1. BFS_For_Bipartite(G, s)

2. for each vertex u ∈ G.V – {s}
3.     u.color = WHITE
4.     u.d = ∞
5.     u.π = NIL

6. s.color = GRAY
7. s.d = 0
8. s.π = NIL
9. Q = ϕ
10.ENQUEUE(Q, s)
11.while Q ≠ ϕ
12.    u = DEQUEUE(Q)
13.    for each vertex v in G.Adj[u]
14.        if v.color == WHITE
15.            v.color = GRAY
16.            v.d = u.d + 1
17.            v.π = u
18.            ENQUEUE(Q, v)
19.        else if u.d == v.d
20.            return 0
21.    u.color = BLACK
22.return 1
```

We are checking the distances at line-19.

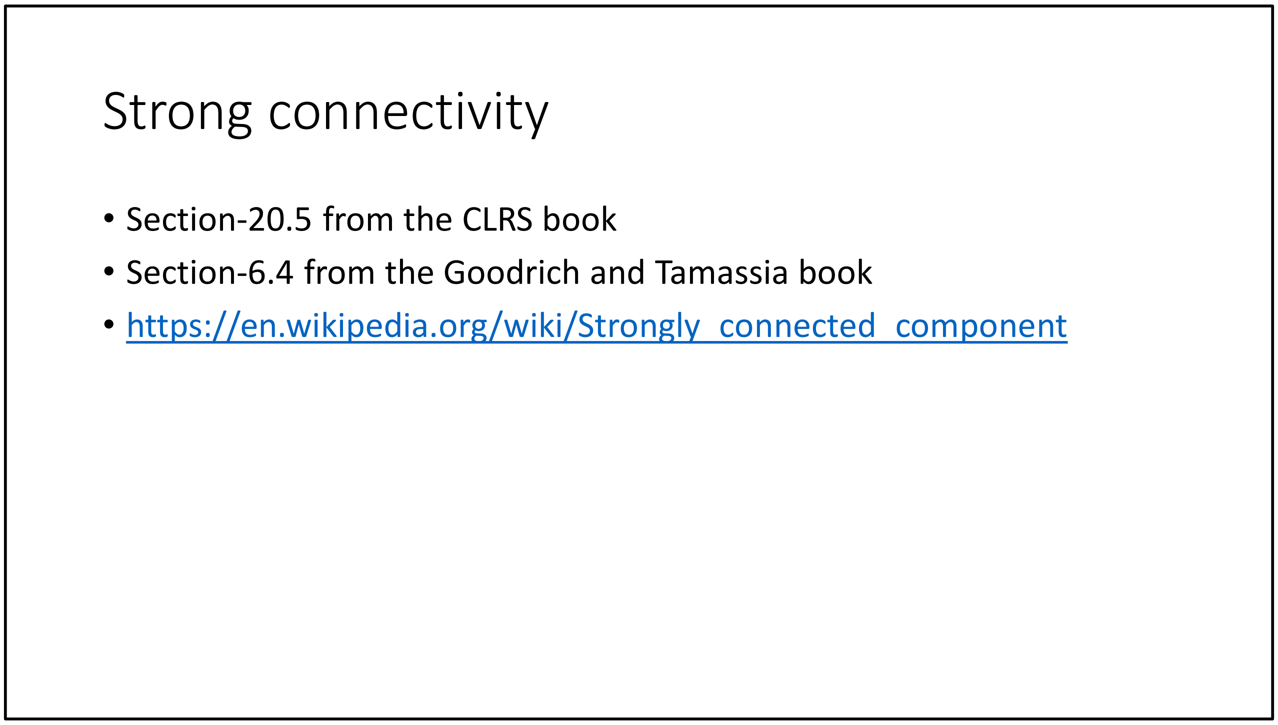
Bipartite graph



Not bipartite.



Strong connectivity



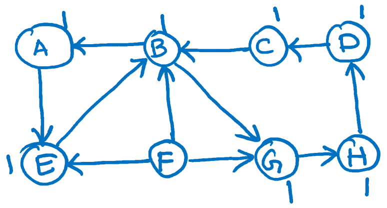
Strong connectivity

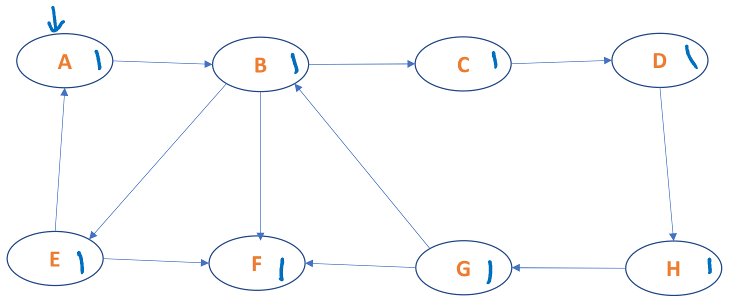
- Section-20.5 from the CLRS book
- Section-6.4 from the Goodrich and Tamassia book
- https://en.wikipedia.org/wiki/Strongly_connected_component

Strong connectivity

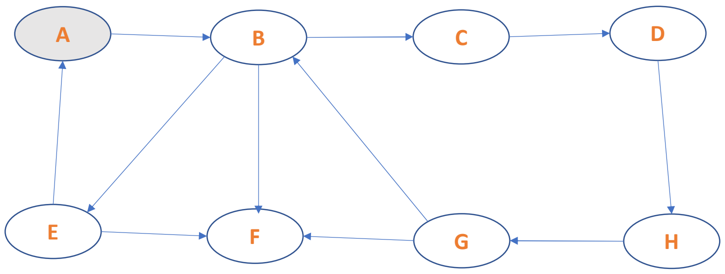
- A directed graph is strongly connected if every vertex is reachable from all other vertices

Strong connectivity



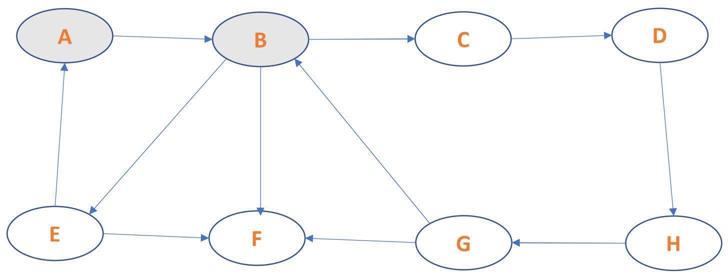


Strong connectivity

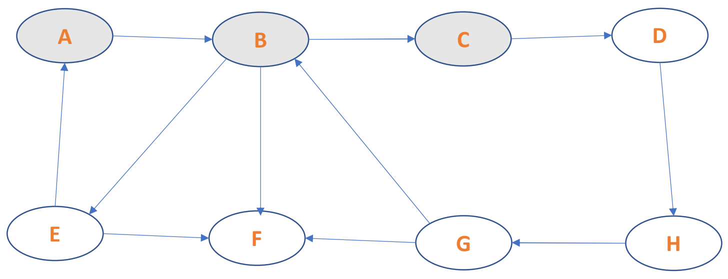


First, we will do a DFS to find all vertices reachable from A.

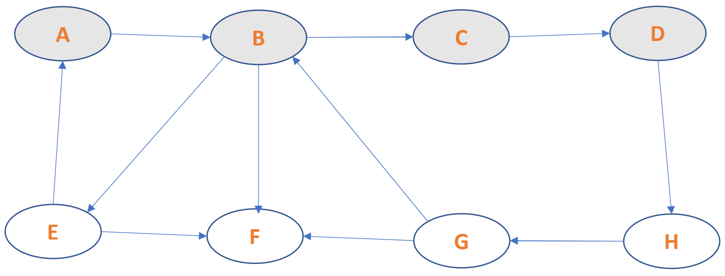
Strong connectivity



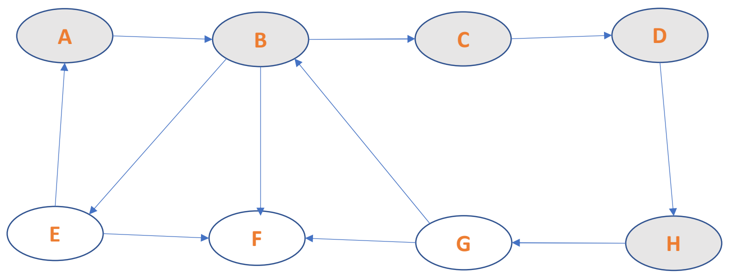
Strong connectivity



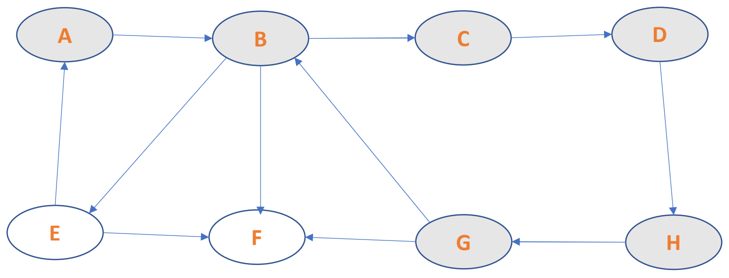
Strong connectivity



Strong connectivity



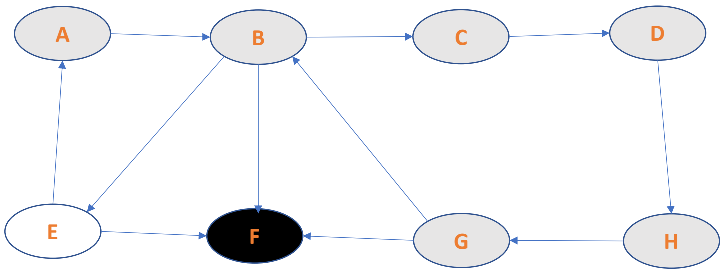
Strong connectivity



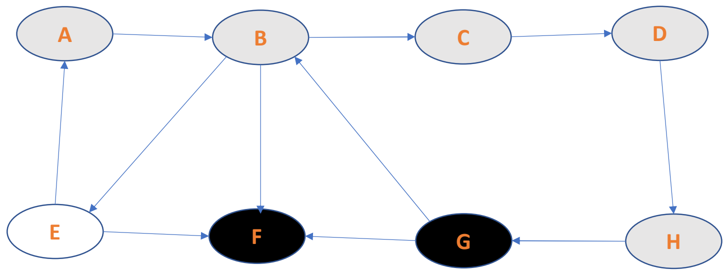
Strong connectivity

$$\begin{tikzcd}
A \arrow[r] & B \arrow[r] \arrow[d] \arrow[dl] & C \arrow[r] & D \arrow[d] \\
E \arrow[u] \arrow[r] & F & G \arrow[l] \arrow[ul] & H \arrow[l]
\end{tikzcd}$$

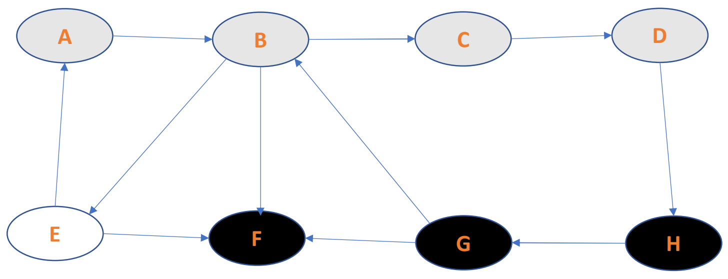
Strong connectivity



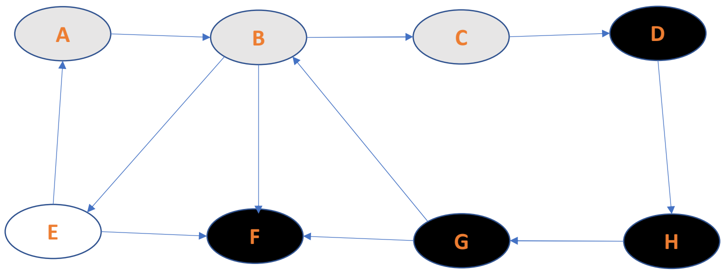
Strong connectivity



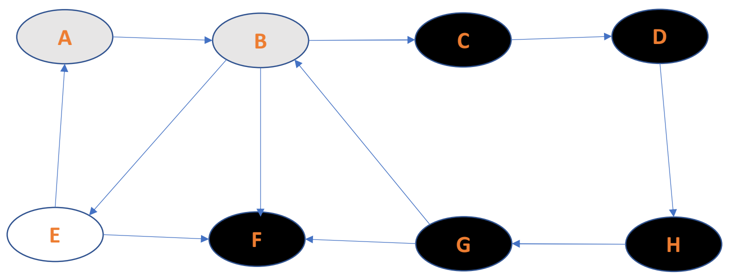
Strong connectivity



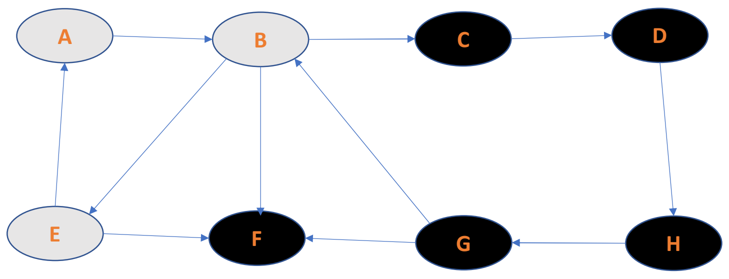
Strong connectivity

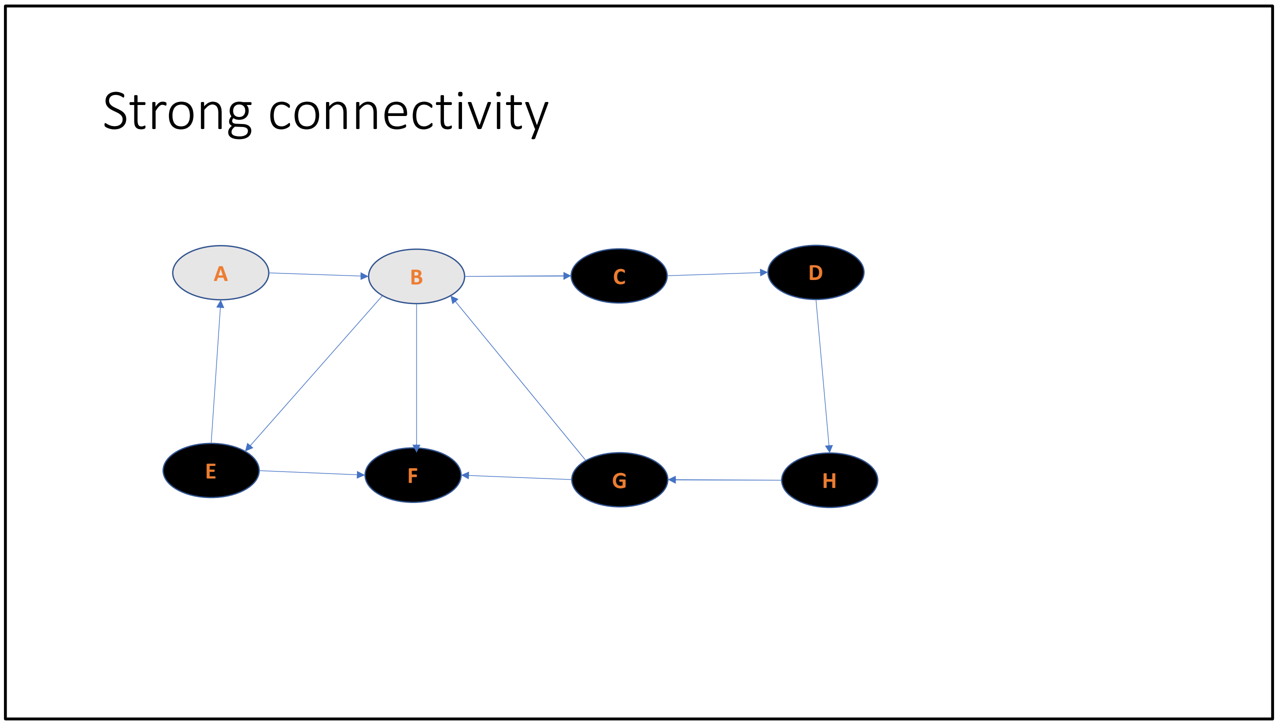


Strong connectivity

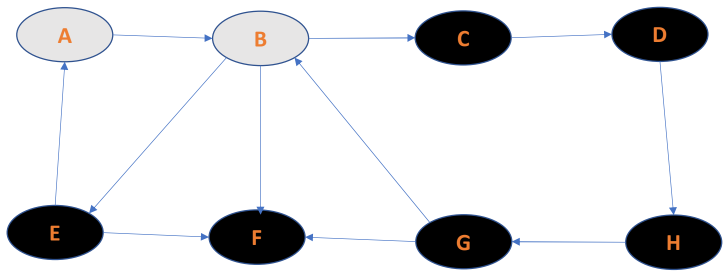


Strong connectivity

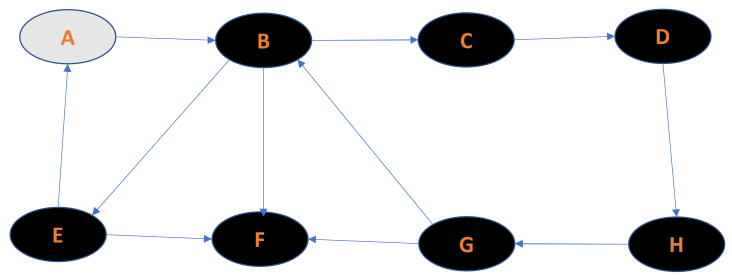




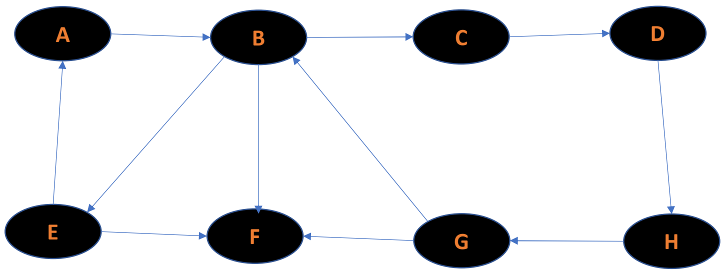
Strong connectivity



Strong connectivity



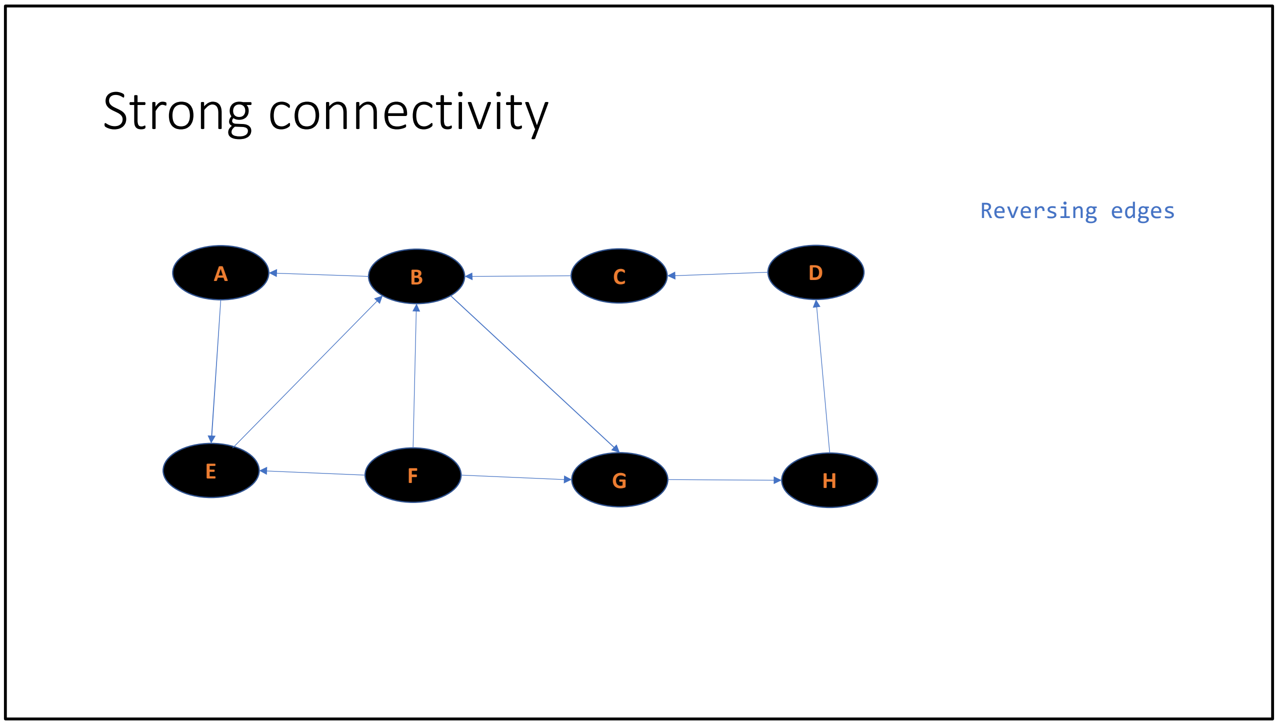
Strong connectivity



The graph can be strongly connected because all vertices are reachable from A. Next, we need to check whether we can reach A from all other vertices to conclude that this graph is indeed strongly connected.

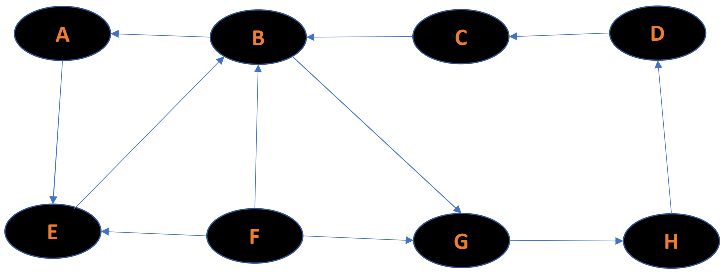
Strong connectivity

- Reverse all edges if all vertices are reachable via some starting vertex
- Run DFS again on the starting vertex



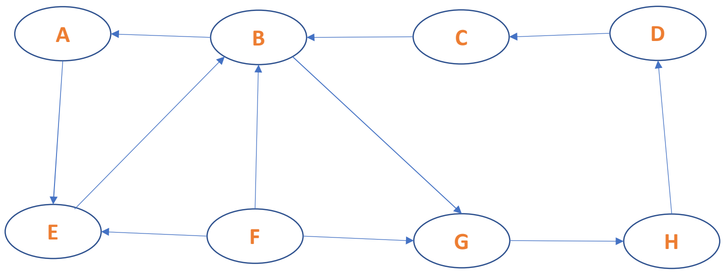
Strong connectivity

Reversing edges



To find out that A is reachable from all other vertices, we can reverse all edges and run BFS at A again. If all vertices are reachable via A, it means that we can reach A via all vertices because the path from A to another vertex computed during DFS is a reversed path from A to that vertex.

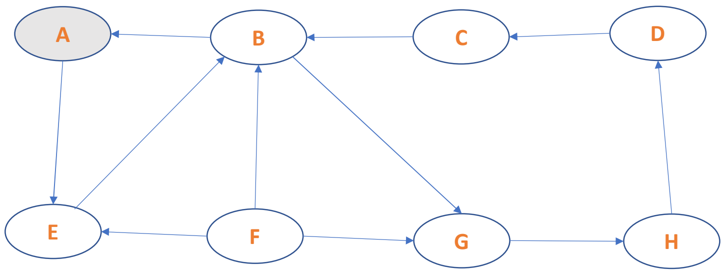
Strong connectivity



Reversing edges

Reset all colors

Strong connectivity

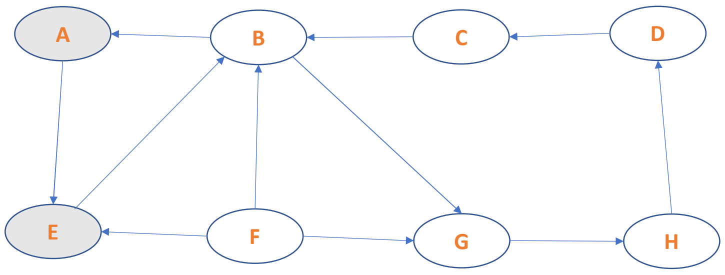


Reversing edges

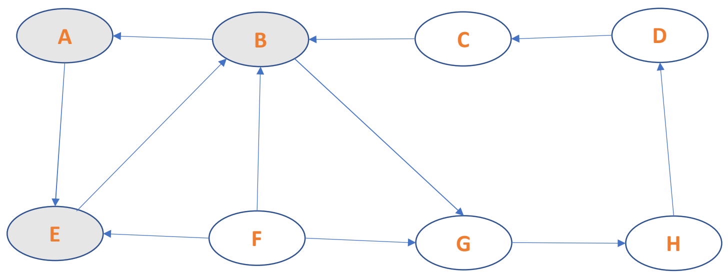
Reset all colors

Running DFS on A

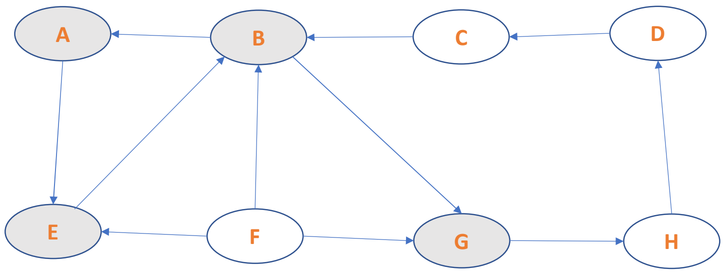
Strong connectivity



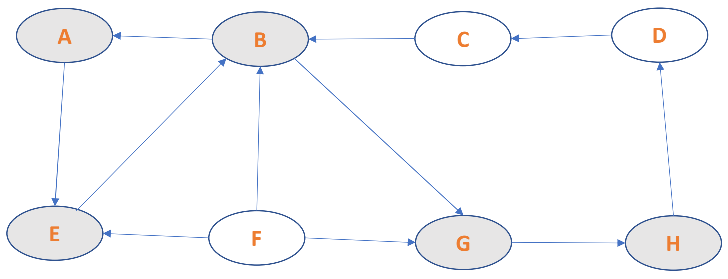
Strong connectivity



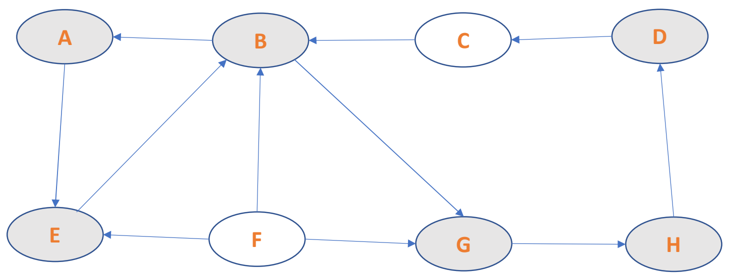
Strong connectivity



Strong connectivity



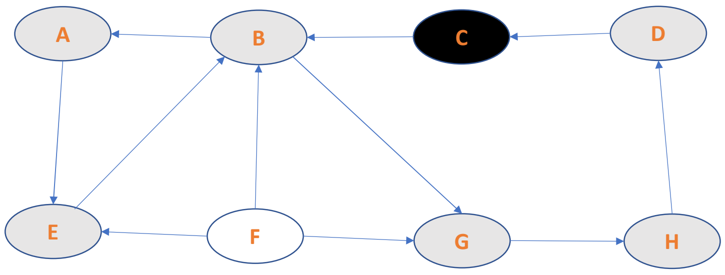
Strong connectivity



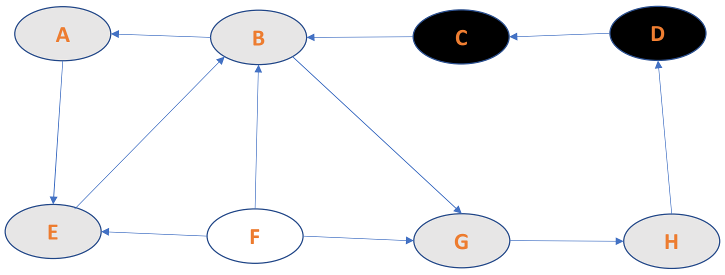
Strong connectivity

$$\begin{tikzcd}
A \arrow[d] & B \arrow[l] \arrow[dr] & C \arrow[l] & D \arrow[l] \\
E \arrow[ur] & F \arrow[u] \arrow[l] \arrow[r] & G \arrow[r] & H \arrow[u]
\end{tikzcd}$$

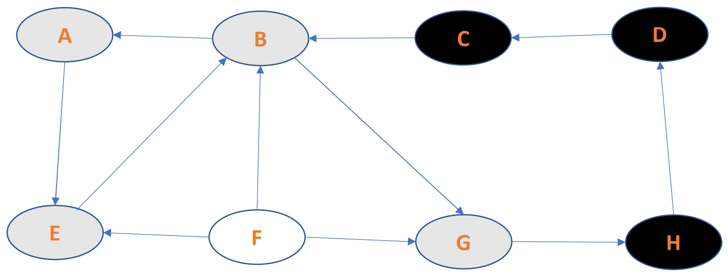
Strong connectivity



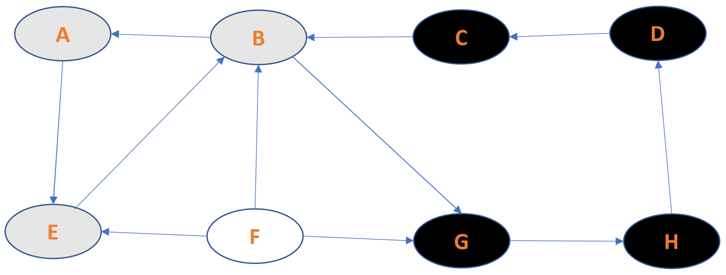
Strong connectivity



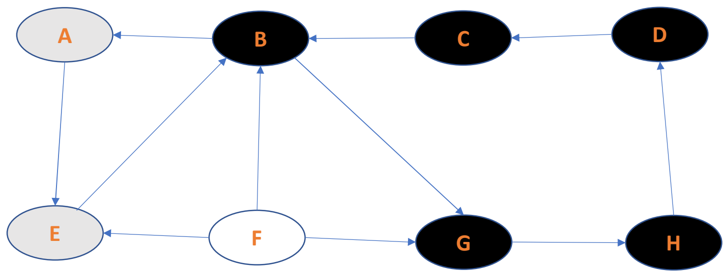
Strong connectivity



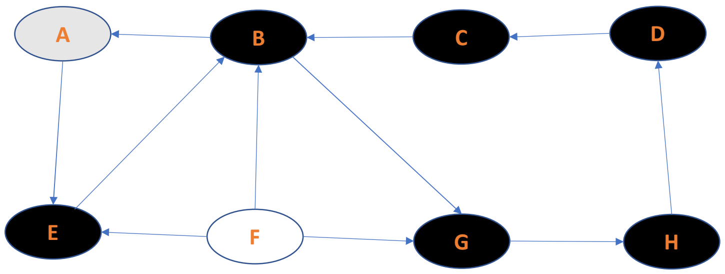
Strong connectivity



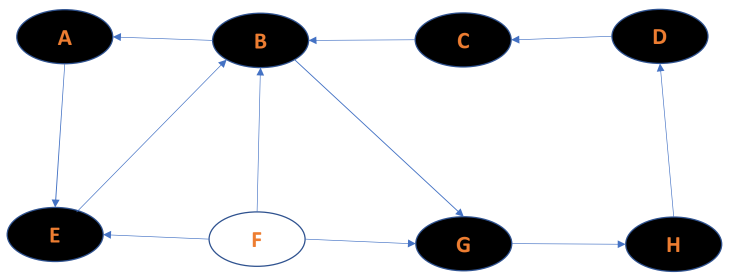
Strong connectivity



Strong connectivity

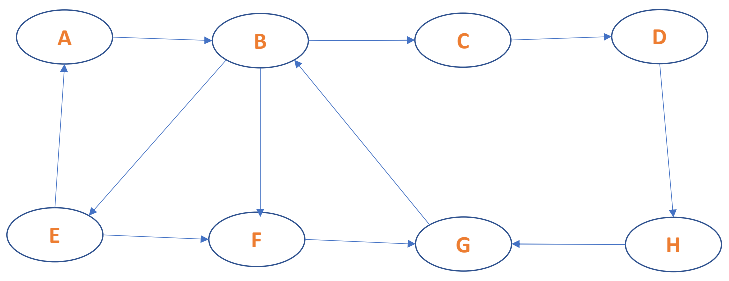


Strong connectivity



The graph is not strongly connected because not all vertices are reachable from A in the reversed graph.

Strong connectivity



```
Exercise:
Try finding
strongly connected
components for
this graph.
```

Strongly connected components (SCC)

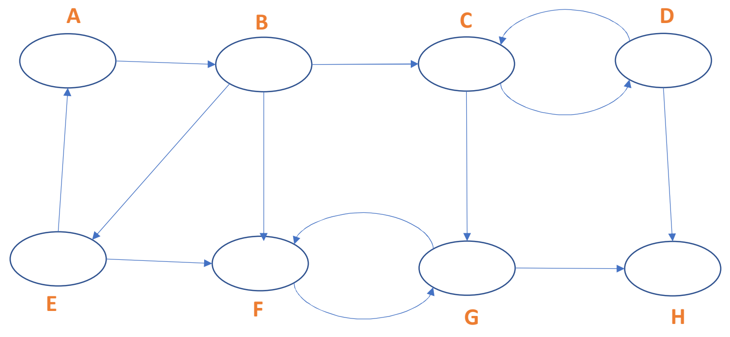
SCC

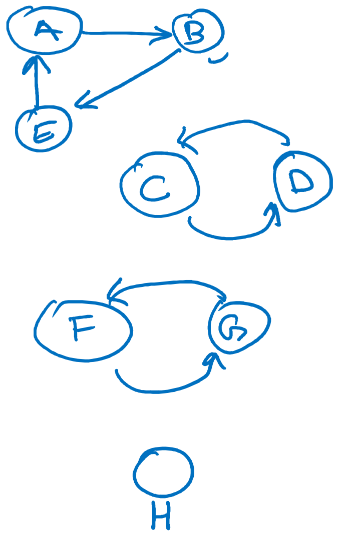
- A strongly connected component of a directed graph G is a strongly connected maximal subgraph

SCC

- Find all strongly connected components of a graph

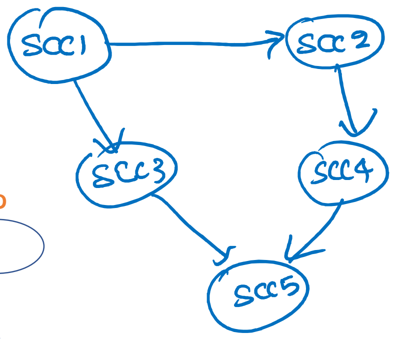
SCC

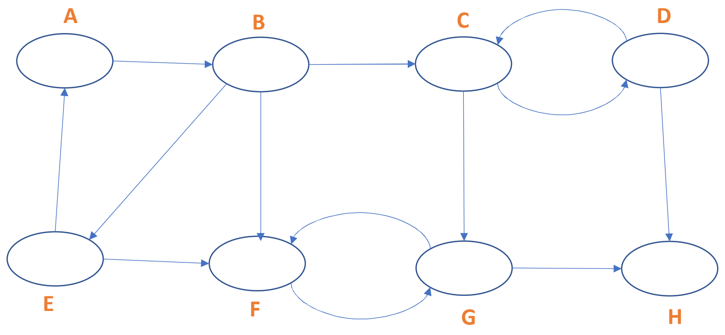




A strongly connected component is a maximal subgraph that is also strongly connected. We have listed four subgraphs on this slide that are strongly connected and maximal. Notice that we can't add other vertices or edges to these subgraphs without violating the strongly connected property.

SCC





To find the strongly connected components (SCCs), we can run DFS at a given vertex (say A) to find all vertices reachable from A and then again run DFS of the reversed graph to find the vertices from which A is reachable. The intersection of vertices computed in these two passes will give us the SCC that contains A. We can then remove the vertices present in the SCC for A and rerun this algorithm. Notice that this algorithm works correctly; however, it is very inefficient. If we create a graph G whose vertices are SCCs and edges are edges between SCCs, the resulting graph must be acyclic. The reason behind this is that if G contains a cycle, and S1 and S2 are two SCCs on a cyclic path, in that case, we can reach from S1 to S2 and vice versa. It means that S1 and S2 combined can also be considered as an SCC. As in the presence of a cycle, S1 and S2 will not be SCCs; G must be acyclic. We will use this observation to compute SCCs efficiently.