Algorithms and Data Structures

Module 1

Lecture 3

Graph traversals: depth-first search, breadth-first search and their applications. Part 2

Adigeev Mikhail Georgievich

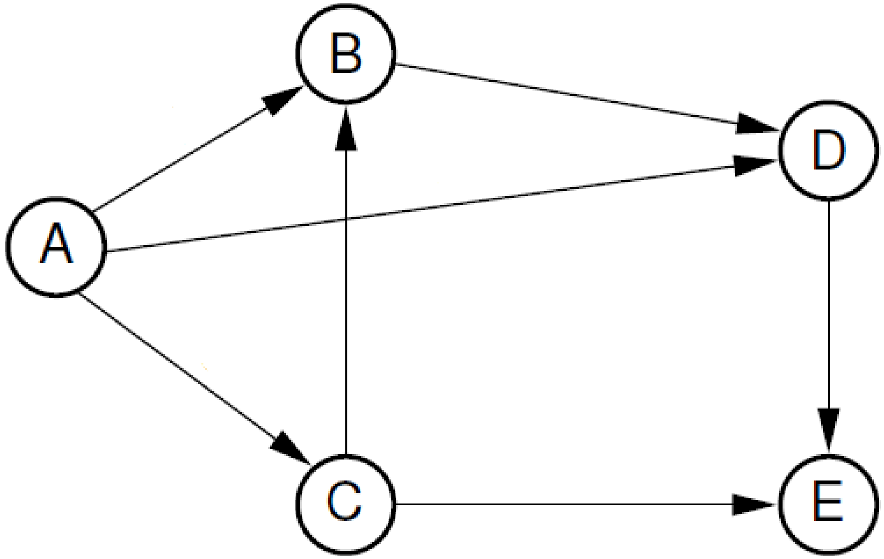
mgadigeev@sfedu.ru

BFS: calculating distances

Graph G=(V,E).

A *distance* between vertices *u* and *v* is the minimum length of the path between *u* and *v*.

dist(A,E) = 2



BFS: calculating distances

For unweighted graphs distances from $s \in V$ to all other vertices can be calculated using BFS.

Idea of the algorithm: `BFS` starts from s and traverses G with 'waves'. Each wave is formed in one iteration of the loop

```
    For each u in Adj(v)
```

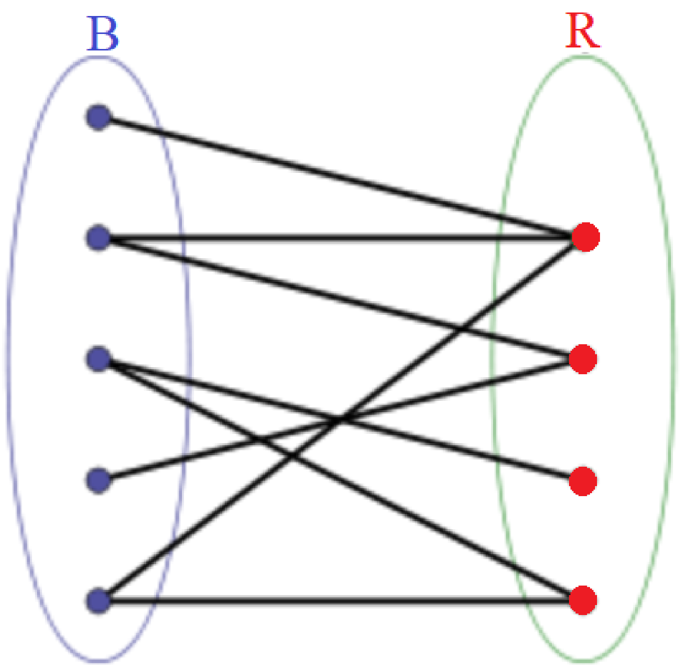
Wave number = distance from s to the vertex which was reached in this wave.

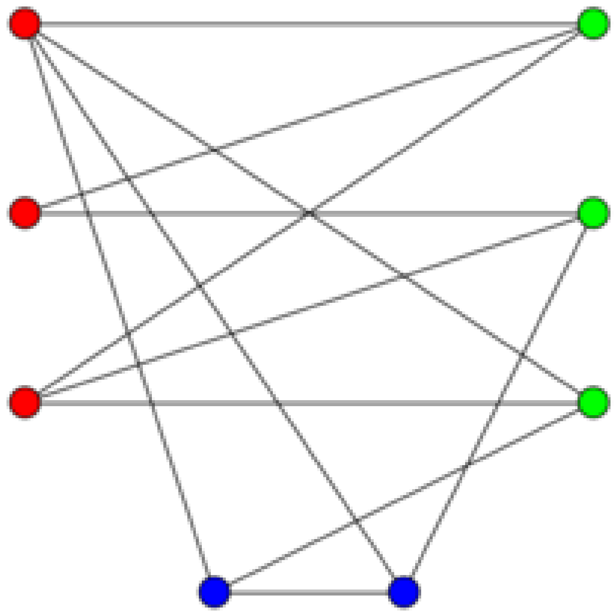
BFS: calculating distances

```
BFS_Visit(s)
For each v ∈ V\{s}: Dist[v] := +∞;
Dist[s] := 0;
Queue.Enqueue(s);
While (!Queue.IsEmpty())
   v = Queue.Dequeue();
   For each u in Adj(v)
        If State[u] = 'unvisited'
             State[u] := 'visited';
             Pred[u] := v;
             Dist[u] := Dist[v]+1;
             Queue.Enqueue(u);
   State[v] := 'processed';
```

BFS: Bipartiteness check

- Graph $G(V, E)$ is called ***bipartite*** iff its vertex set V can be partitioned into two disjoint subsets (***parts***): $V = B \cup R$ such that for each edge $e \in E$ the endpoints of e belong to different subsets.

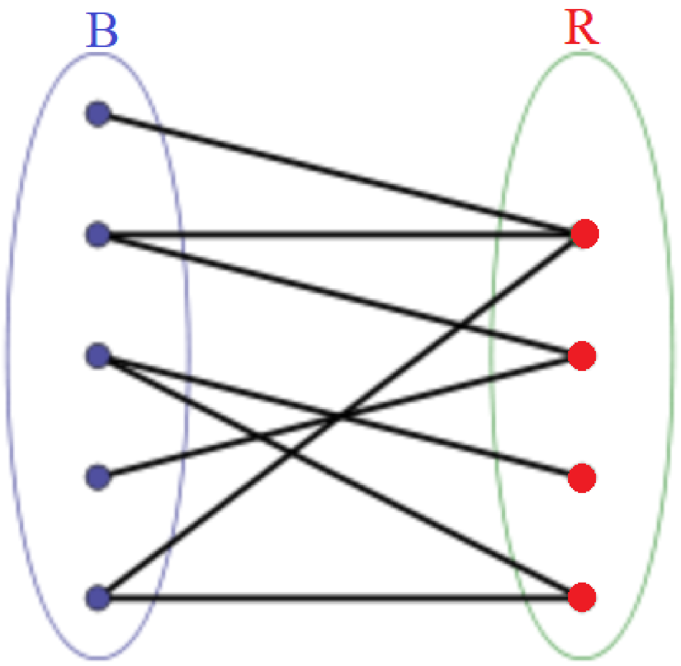


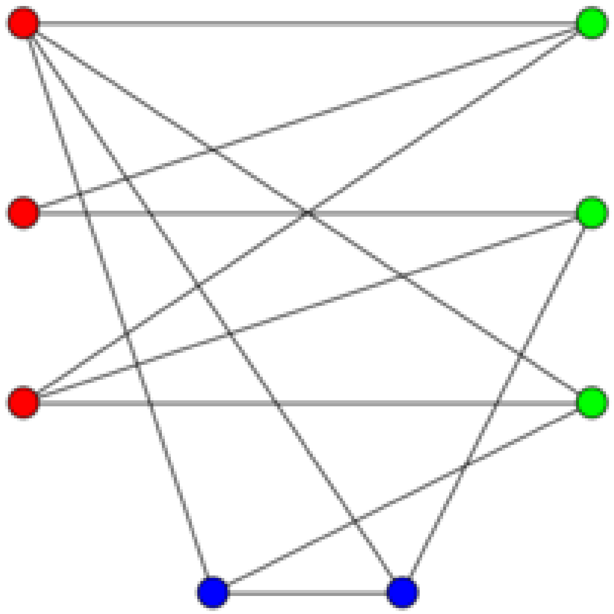


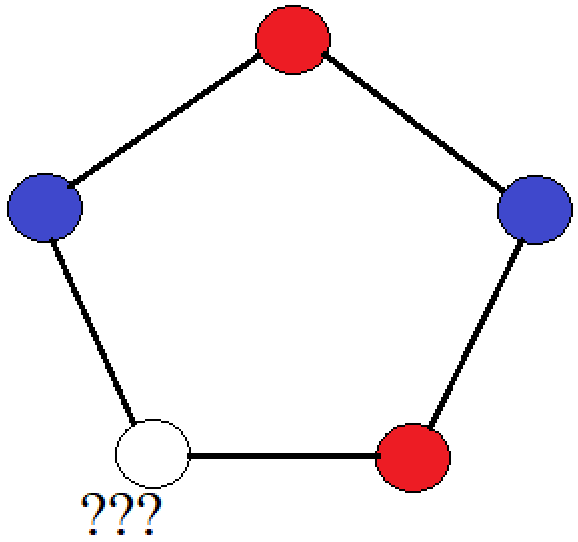
BFS: Bipartiteness check

<u>Theorem</u>. Graph $G(V, E)$ is ***bipartite*** iff it has no cycles of odd length.

<u>Corollary</u>: trees and forests are bipartite graphs.







BFS: Bipartiteness check

Algorithm for bipartiteness check.

Let $G(V, E)$ be a connected graph.

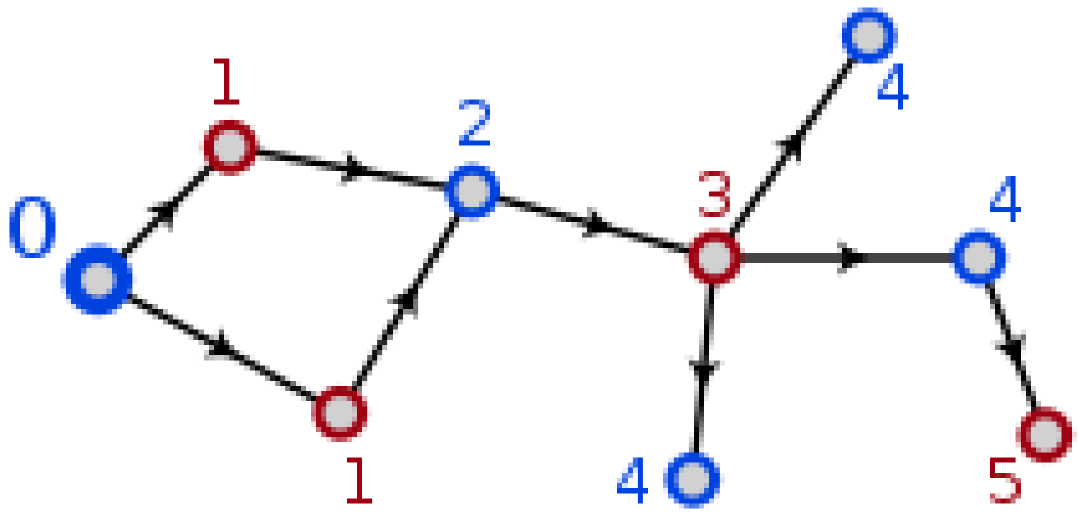
1. $R = B = \emptyset$
2. Select any $s \in V$. d[s]=0.
3. Calculate $d[v]$ - distances from s to all other vertices.
4. For each $v \in V$:

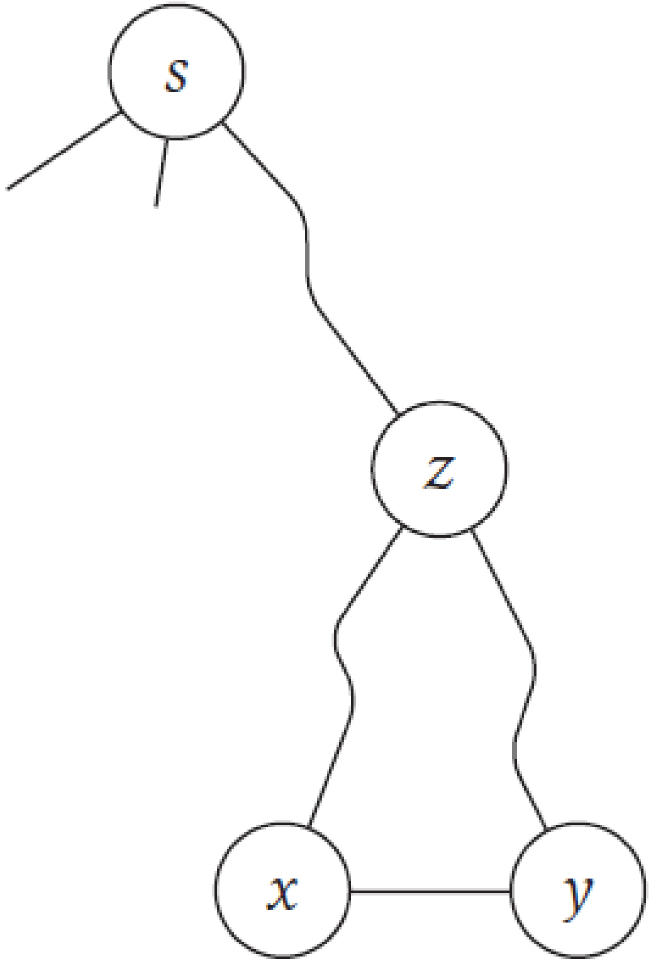
   if d[v] is odd: $R = R \cup \{v\}$

   else: $B = B \cup \{v\}$
5. Scan thru E and check whether the condition holds.

Time complexity: $O(|V| + |\mathrm{E}|)$

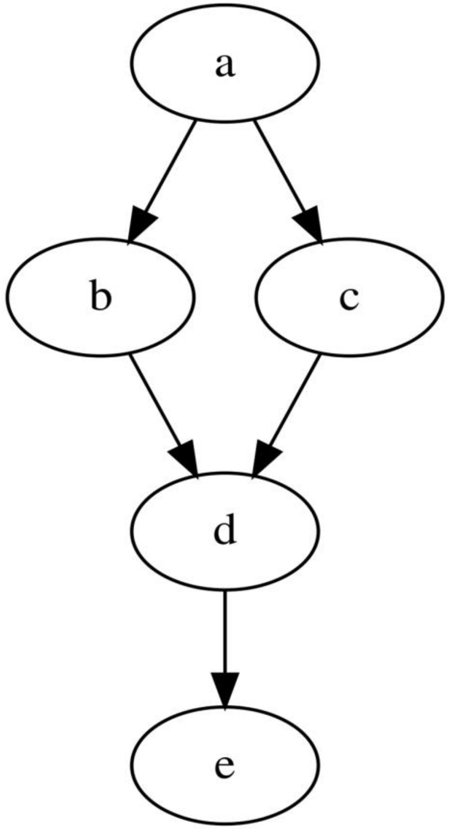
BFS: Bipartiteness check

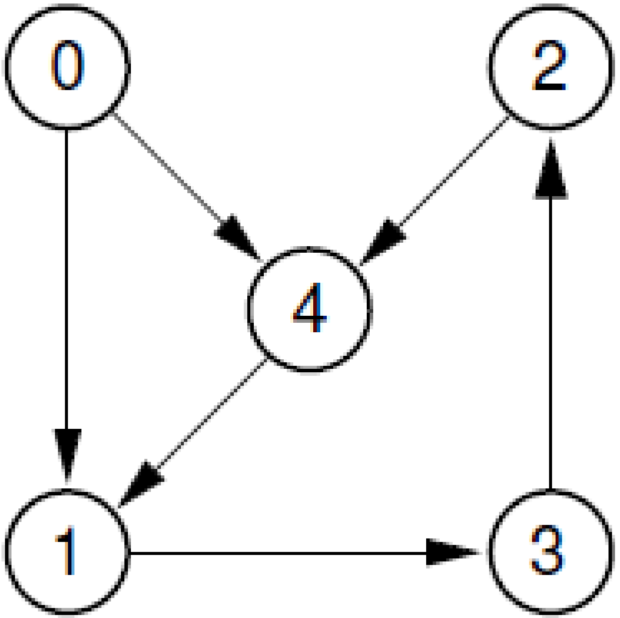




DFS: Detecting cycles

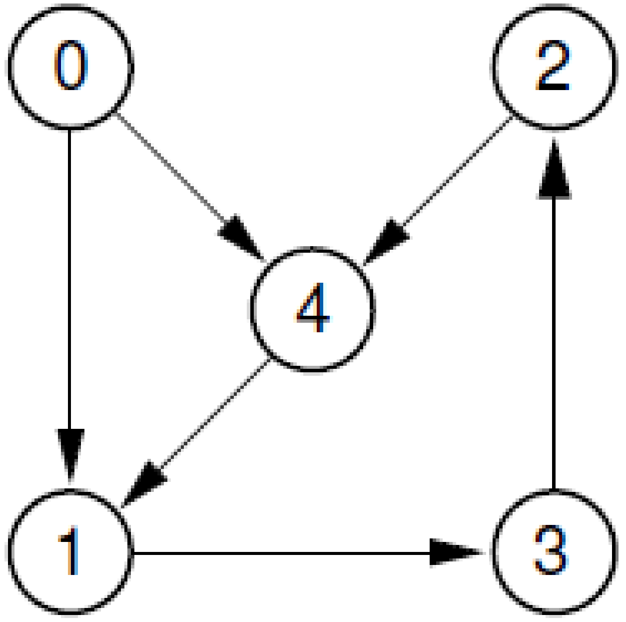
DAG = directed acyclic graph = directed graph with no directed cycl





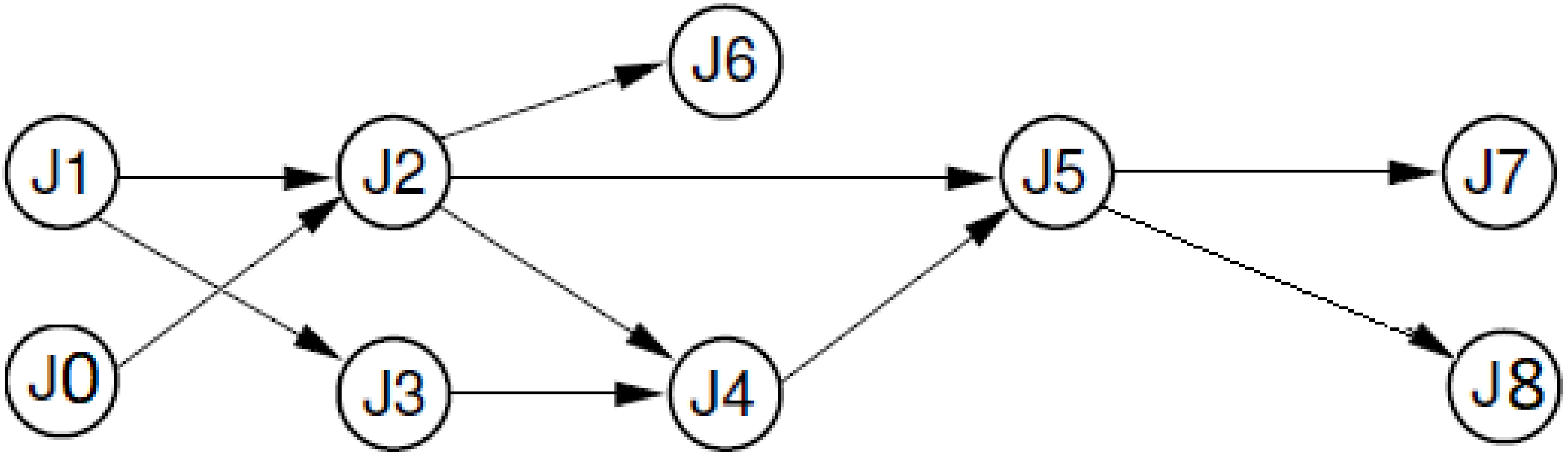
DFS: Detecting cycles

```
DFS(v)
Mark v as 'visited'
Mark v as 'active'
For each u in Adj(v):
      if u is unvisited:
            DFS(u)
      else if u is 'active':
            a cycle found!!!
Mark v as 'inactive'
```



DFS: Topological sort of a DAG

Topological ordering (sort) is vertex numbering $\tau: V \leftrightarrow \{1, \dots, |V|\}$:
there are no edges (*u*,*v*) in *G*: $\tau(u) > \tau(v)$.



Graphs: definition (lecture 01)

- $v \in V$:

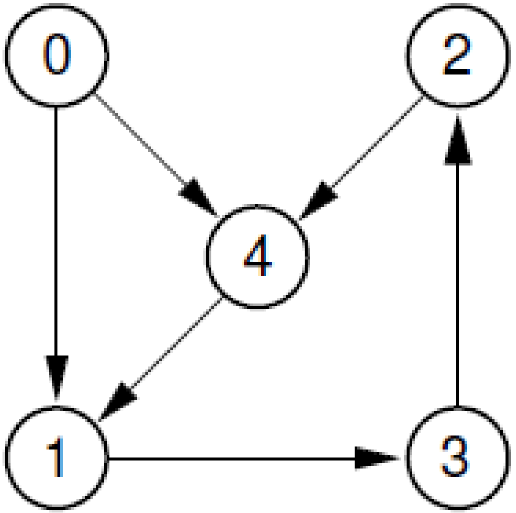
✓ $\deg(v)$ – ***degree*** of vertex v = number of edges incident to v .

✓ $\text{outdeg}(v)$– out-degree of vertex v = number of edges which start from v .

✓ $\text{indeg}(v)$– in-degree of vertex v = number of edges which end at v .

✓ v is a ***source*** iff $\text{indeg}(v) = 0$

✓ v is a ***sink*** iff $\text{outdeg}(v) = 0$



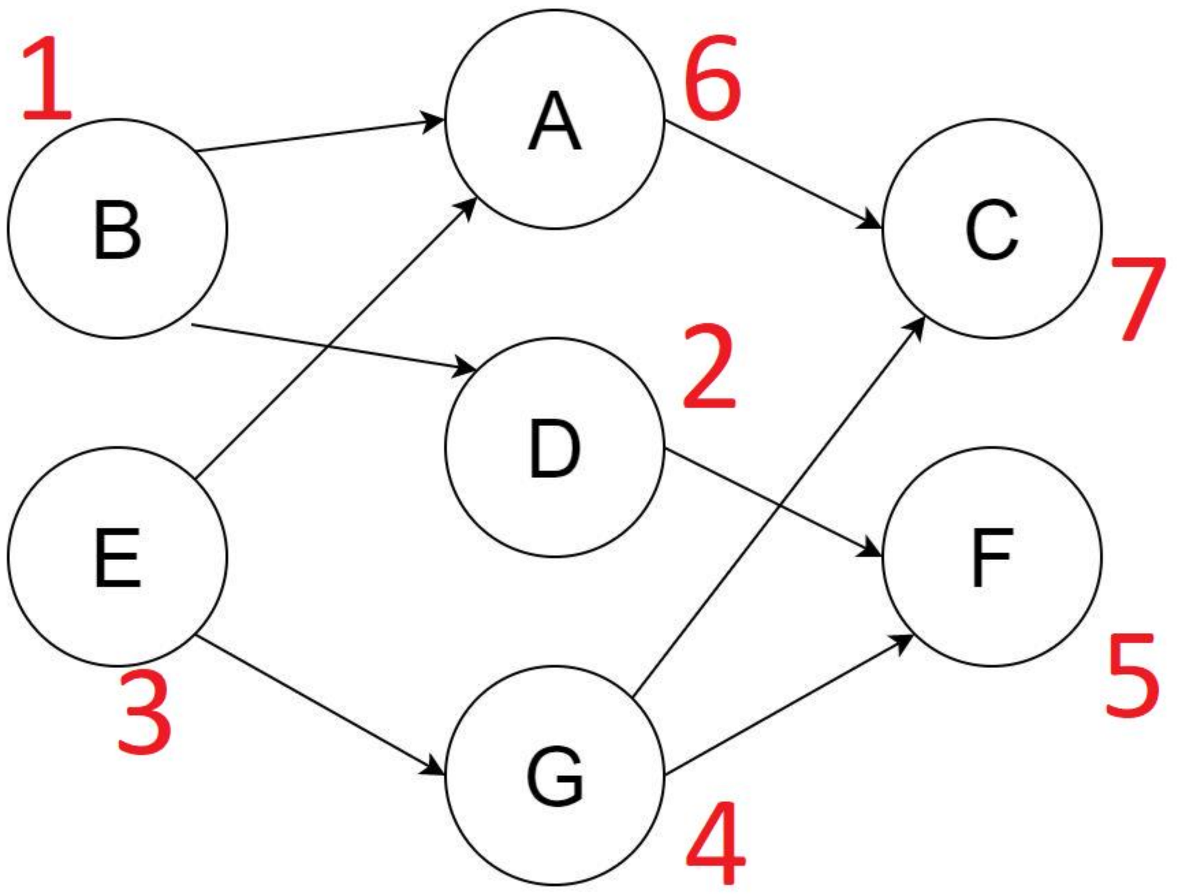
DFS: Topological sort of a DAG

- Assign a vertex 'topological number' just before leaving this vertex: initialize `CurTopNum` with $n = |V|$, then run DFS:

```
DFS(v)
~~PreVisit(v)~~
Mark v as 'visited'
For each u in Adj(v):
    if u is unvisited: DFS(u)
PostVisit(v)
```

```
PostVisit(v)
TopNum[v] = CurTopNum
CurTopNum--
```

DFS: Topological sort of a DAG



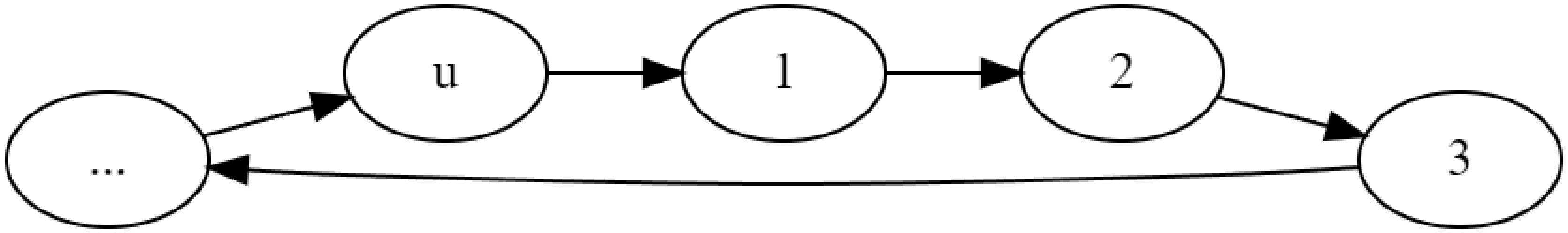
Topological sort of a DAG

Theorem. A directed graph G has a topological sort iff G is a DAG.

Proof

⇒ Suppose that G is not acyclic, i.e. it contains a directed cycle.

In this case, the vertices of the cycle cannot be numerated according the topological sort requirement.



Topological sort of a DAG

$\Leftarrow$ Let G(V,E) be a DAG. Let us see, how topological sort for G can be built.

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

Algorithm for Topological sort based on sources:

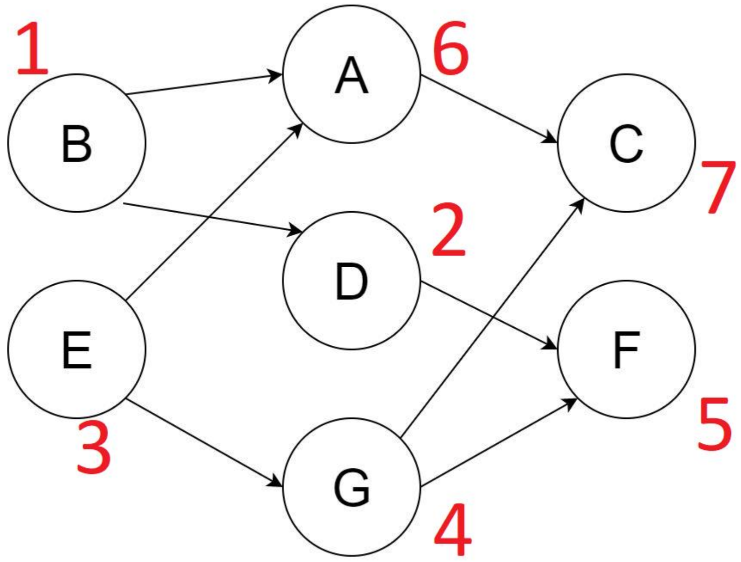
1. Create counter and initialize it with 1.
2. While $|V| > 0$
   - Find a source and assign it the current counter value.
   - Remove this source from the graph.
   - Increase the counter by 1.

Topological sort of a DAG

The resulting numeration is a topological sort.

1) All vertices have numbers. This is due to the fact that after removing a source the graph is still a DAG, so the algorithm is running until all vertices are numbered.

2) For each arc, the number of the starting vertex is less than the number of the finishing vertex.

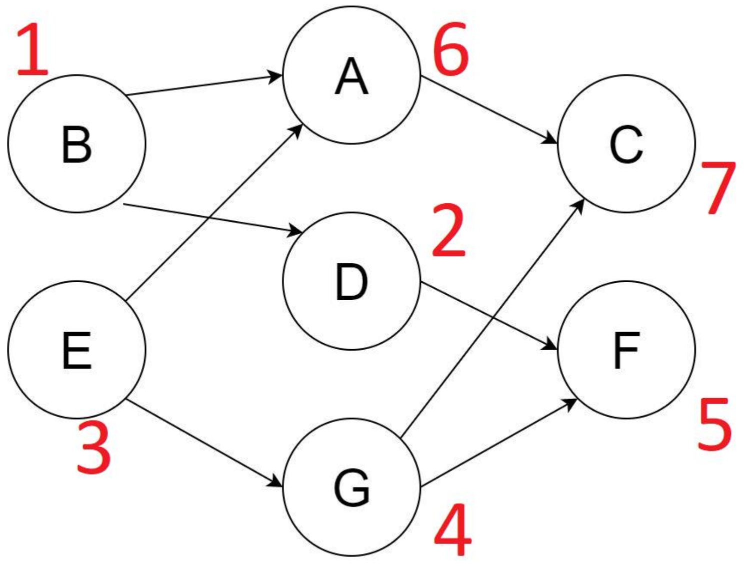


Topological sort of a DAG

DFS can also be used for building topological sort.

1. Create counter and initialize it with the number of vertices ($n = |V|$).
2. Run depth-first-search. Before leaving a vertex, assign it the current counter value as the topological number; the counter is decreased by 1.

Complexity of the topological sort: $O(n + m)$.



Topological sort of a DAG

```
DFS_TopSort(G)
For each v ∈ V:
    State[v] := 'unvisited';
    Pred[v] := NULL;
    Time_In[v] := NULL;
    Time_Out[v] := NULL;
    TopNum[v] := NULL;
CurTime : = 0;
CurTopNum : = n;
For each v ∈ V:
    If State[v] = 'unvisited'
        DFS_TopSort_Visit(v);
```

Topological sort of a DAG

```
DFS_TopSort_Visit(v)
State[v] := 'visited';
CurTime := CurTime + 1;
Time_In[v] := CurTime;
For each u in Adj(v)
    If State[u] = 'unvisited'
        Pred[u] := v;
        DFS_Visit(u);
State[v] := 'processed';
CurTime := CurTime + 1;
Time_Out[v] := CurTime;
TopNum[v] := CurTopNum;
CurTopNum := CurTopNum - 1;
```