

Graphs

Introduction

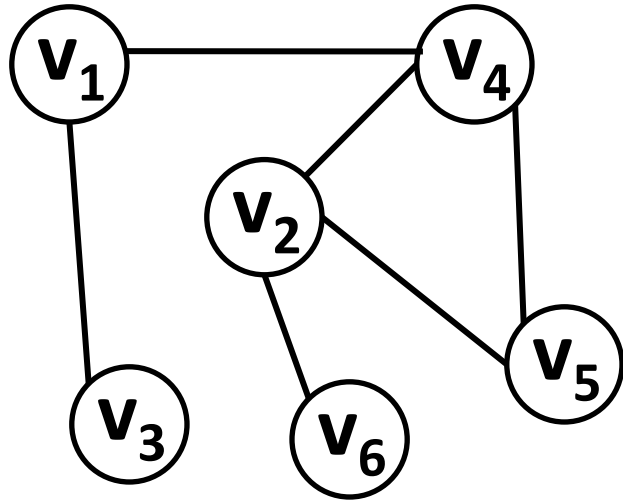
- Generalization of a tree.
- Collection of vertices (or nodes) and connections between them.
- No restriction on
 - The number of vertices.
 - The number of connections between the two vertices.
- Have several real life applications.

Definition

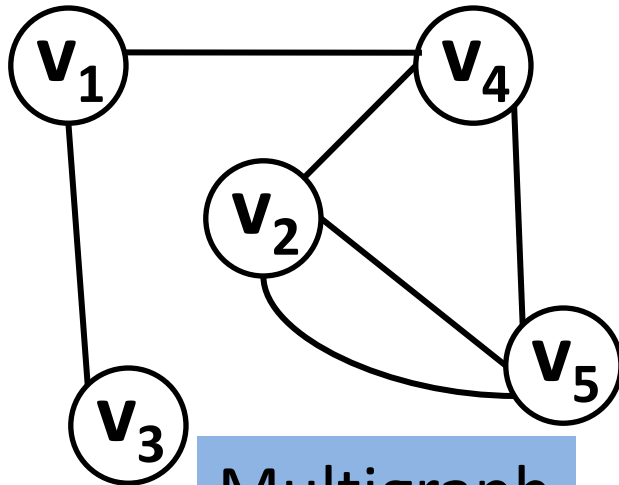
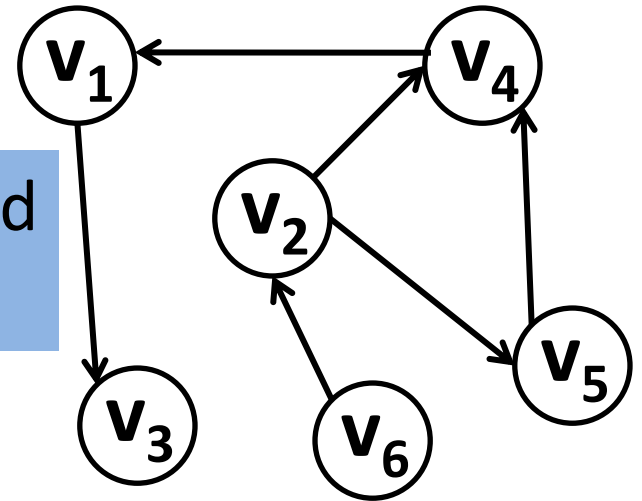
- A graph $G = (V, E)$ consists of a
 - Non-empty set V of *vertices* and
 - Possibly empty set E of *edges*.
- $|V|$ denotes number of vertices.
- $|E|$ denotes number of edges.
- An edge (or arc) is a pair of vertices (v_i, v_j) from V .
 - Simple or undirected graph $(v_i, v_j) = (v_j, v_i)$.
 - Digraph or directed graph $(v_i, v_j) \neq (v_j, v_i)$.
- An edge has an associated **weight** or **cost** as well.

Contd...

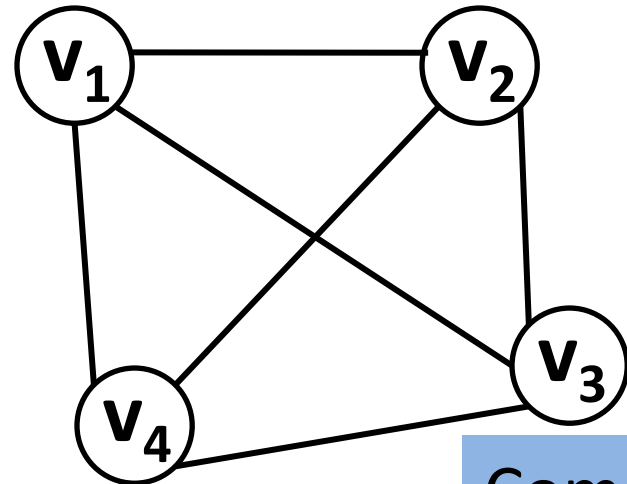
Undirected Graph



Directed Graph



Multigraph

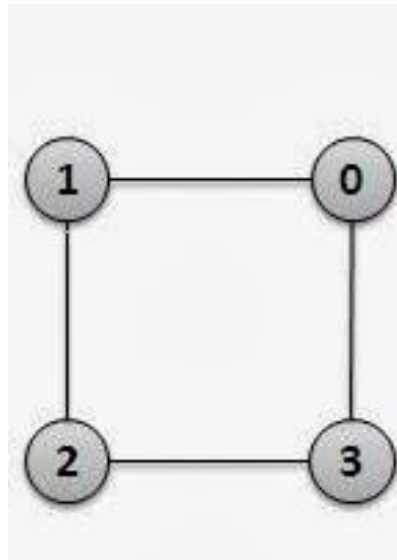


Complete Graph

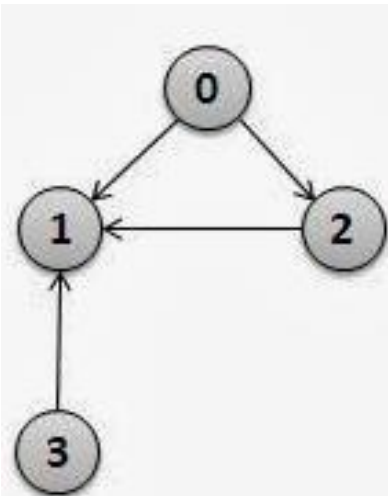
Representation – I

- Adjacency matrix
 - Adjacency matrix for a graph $G = (V, E)$ is a two dimensional matrix of size $|V| \times |V|$ such that each entry of this matrix
$$a[i][j] = \begin{cases} 1 \text{ (or weight), if an edge } (v_i, v_j) \text{ exists.} \\ 0, \text{ otherwise.} \end{cases}$$
 - For an undirected graph, it is always a symmetric matrix, as $(v_i, v_j) = (v_j, v_i)$.

Adjacency matrix

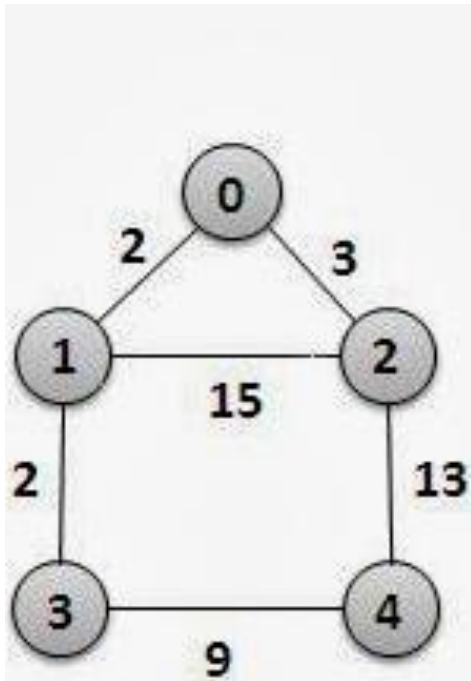


	0	1	2	3
0	0	1	0	1
1	1	0	1	0
2	0	1	0	1
3	1	0	1	0



	0	1	2	3
0	0	1	1	0
1	0	0	0	0
2	0	1	0	0
3	0	1	0	0

Contd... (weighted)



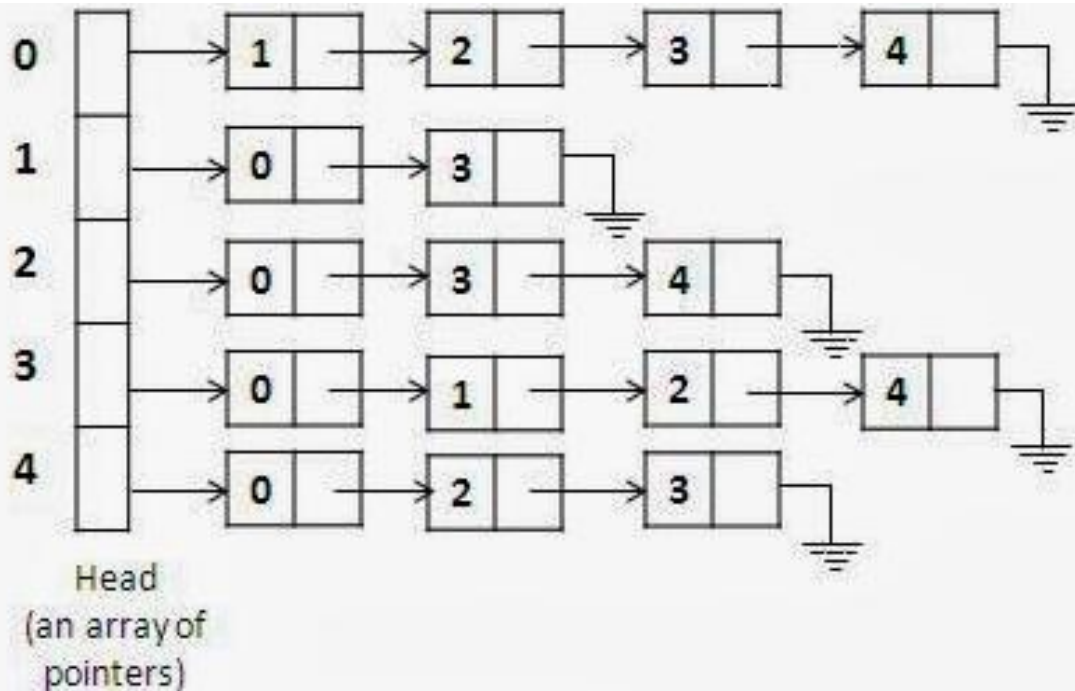
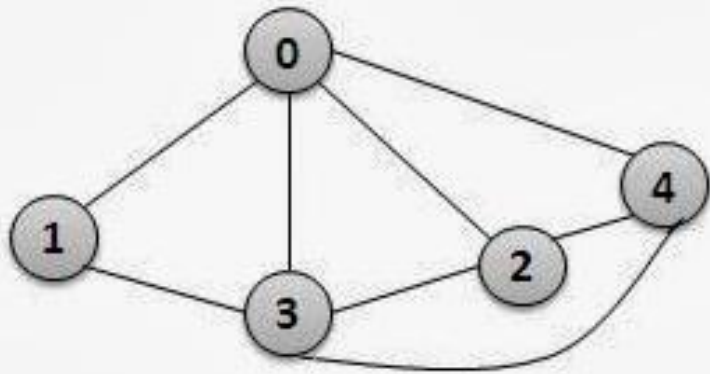
	0	1	2	3	4
0	0	2	3	0	0
1	2	0	15	2	0
2	3	15	0	0	13
3	0	2	0	0	9
4	0	0	13	9	0

Representation – II

- Adjacency list
 - Uses an array of linked lists with size equals to $|V|$.
 - An i^{th} entry of an array points to a linked list of vertices adjacent to v_i .
 - The weights of edges are stored in nodes of linked lists to represent a weighted graph.

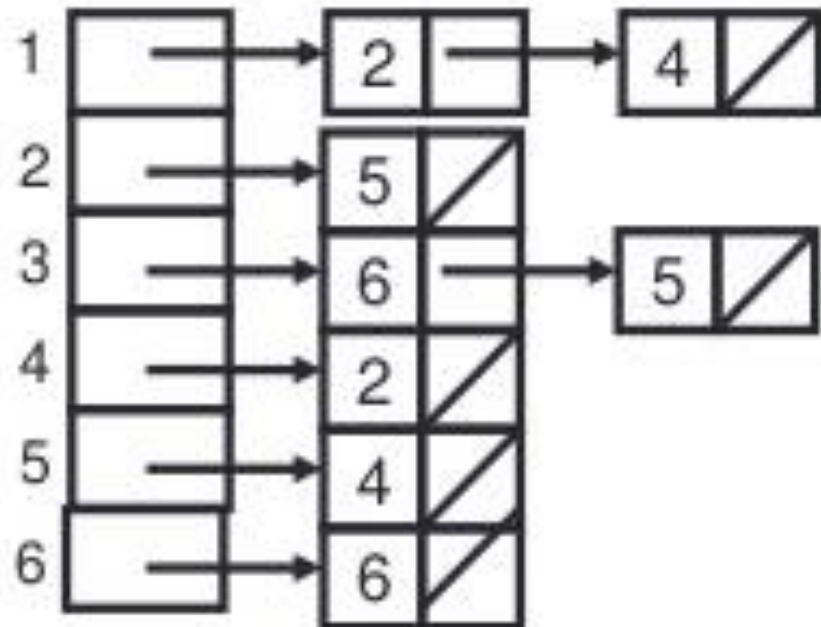
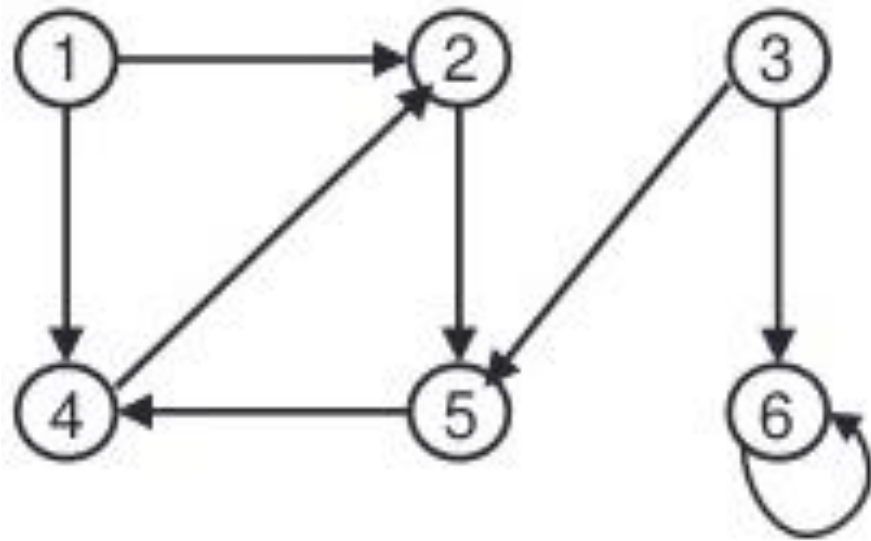
Adjacency List

- Undirected graph



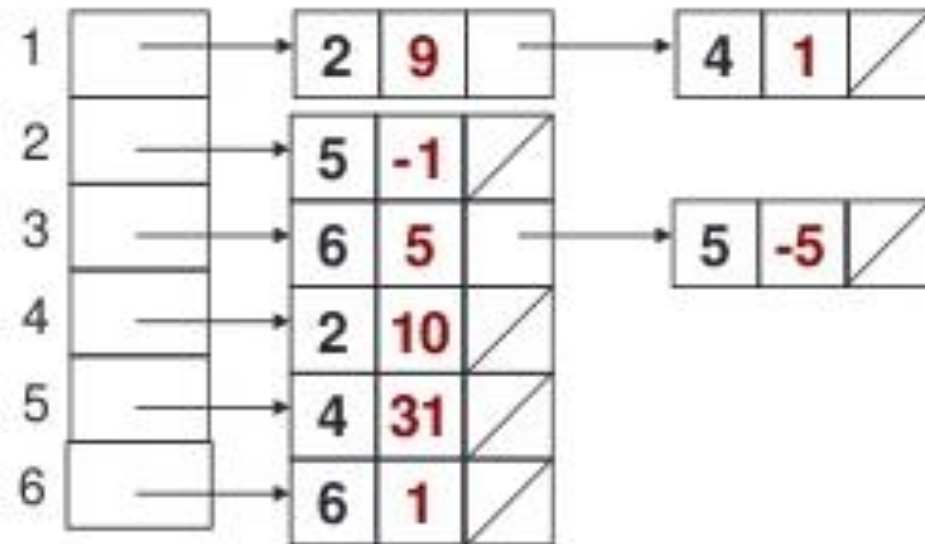
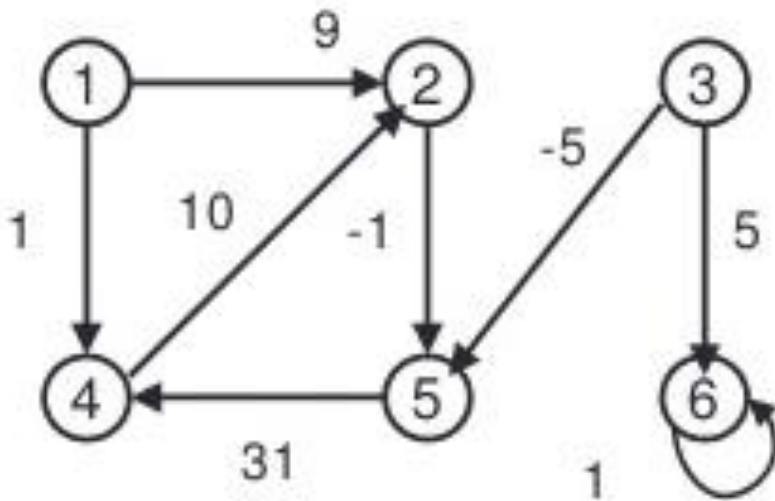
Contd...

- Directed graph



Contd...

- Weighted directed graph



Pros and Cons

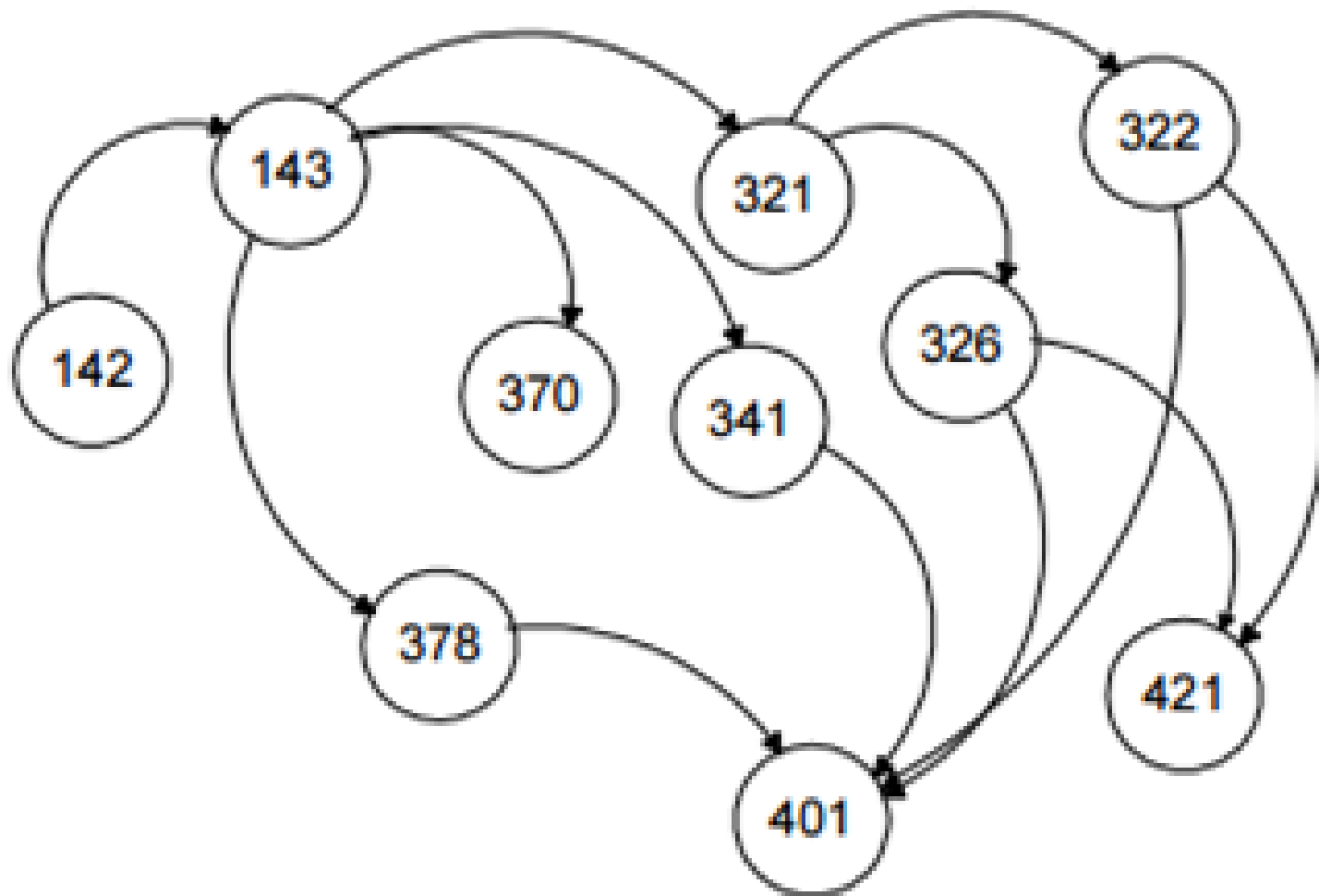
	Adjacency Matrix	Adjacency List
Memory Requirement	$O(V ^2)$	$O(V + E)$
Add Edge	$O(1)$	$O(1)$
Remove Edge	$O(1)$	$O(E)$
Find Edge Existence	$O(1)$	$O(V)$
Find # of Edges	$O(V ^2)$	$O(E)$
Add Vertex	$O(V ^2)$	$O(V)$
Remove Vertex	$O(V ^2)$	$O(E)$
Visit Adjacent Vertices	$O(V)$	$O(\text{Degree of that node})$

Definitions

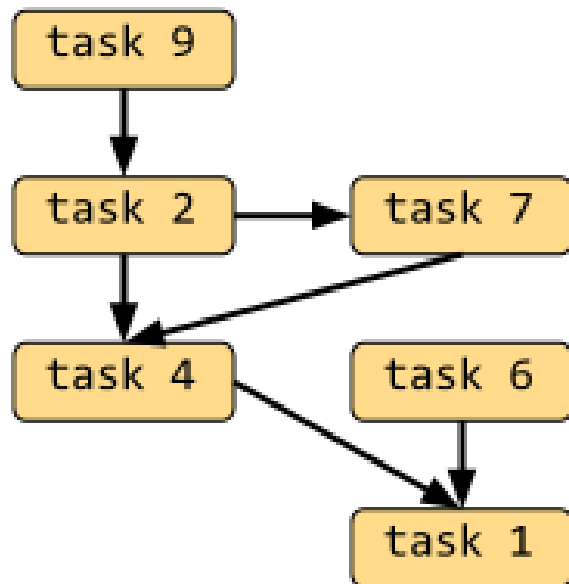
- Diagraph or directed graph.
- A cycle in a diagraph G is a set of edges, $\{(v_1, v_2), (v_2, v_3), \dots, (v_{r-1}, v_r)\}$ where $v_1 = v_r$.
- A diagraph is acyclic if it has no cycles. Also known as directed acyclic graph (**DAG**).
- DAGs are used in many applications to indicate precedence among events. For example,
 - Inheritance between classes.
 - Prerequisites between courses of a degree program.
 - Scheduling constraints between the tasks of a projects.

Example – Prerequisites between courses

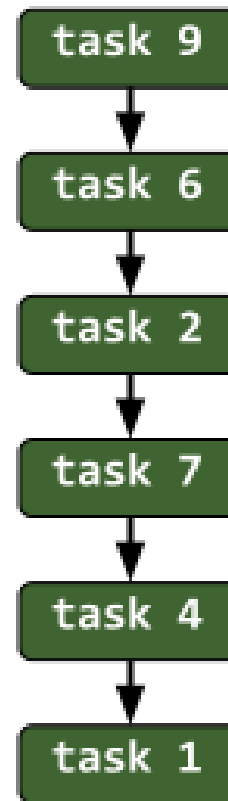
142 → 143 → 378 → 370 → 321 → 341 → 322 →
326 → 421 → 401



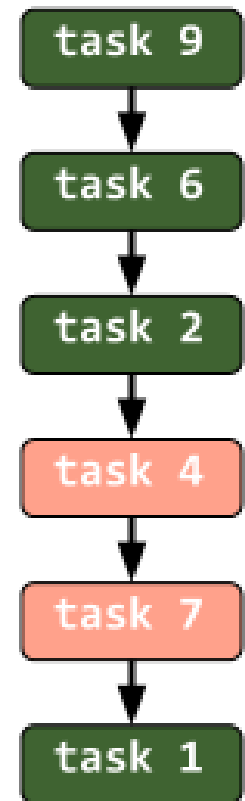
Example – Scheduling



RIGHT

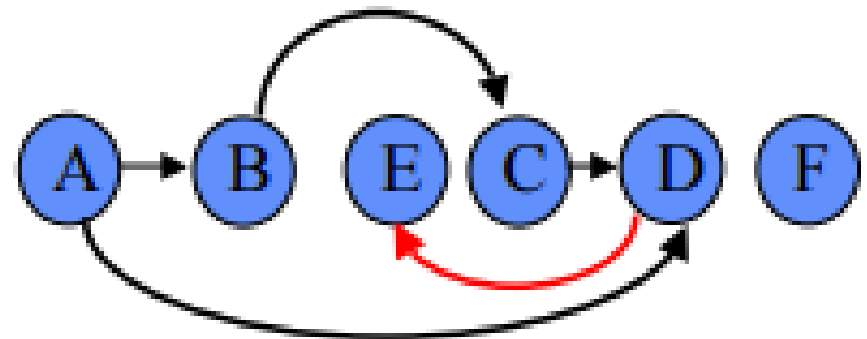
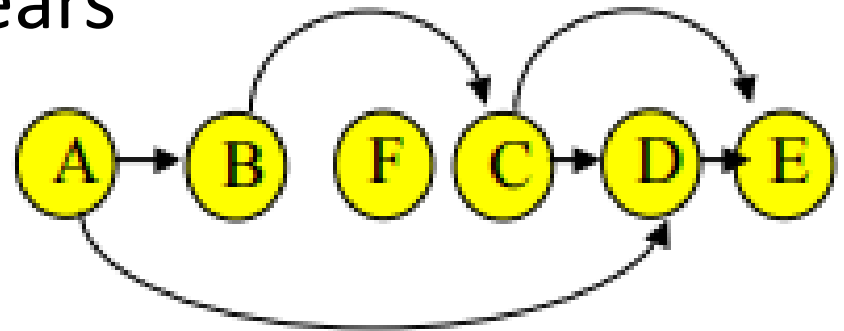
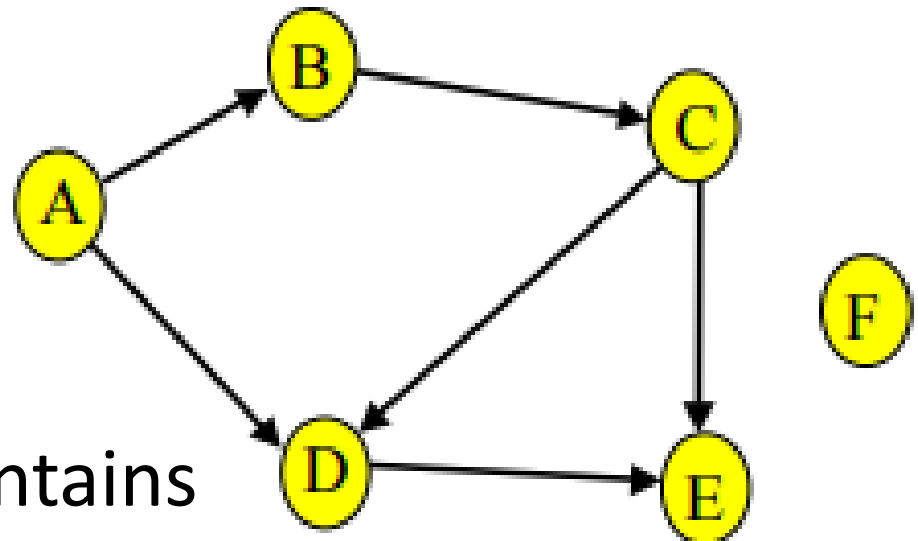


WRONG



Topological Sort

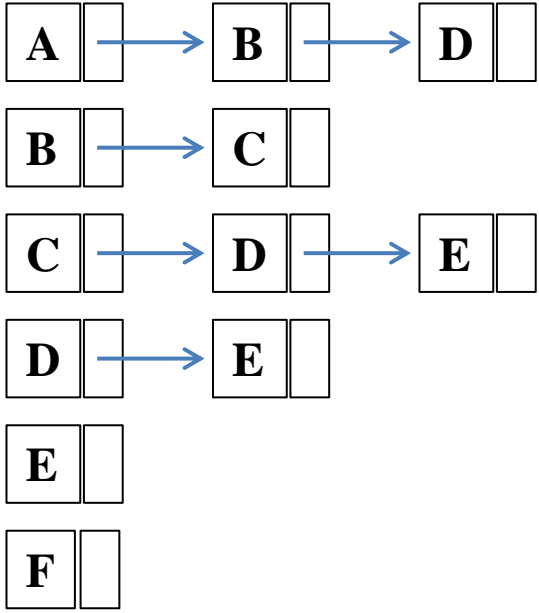
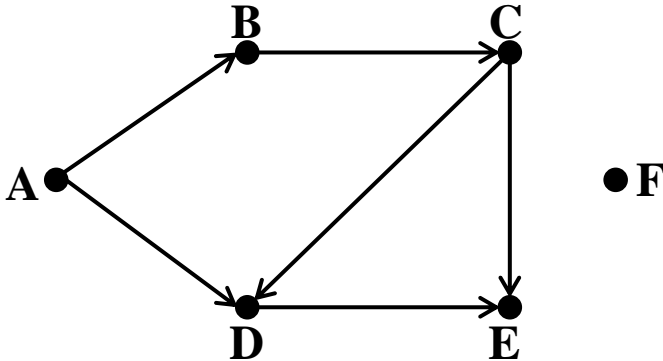
- Let, $G = (V, E)$ be a DAG
- Linear ordering of all its vertices such that if G contains an edge (u, v) , then u appears before v in the ordering.
- Can also be viewed as an ordering of vertices along a horizontal line so that all directed edges go from left to right.



Kahn's algorithm

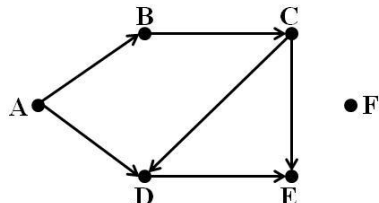
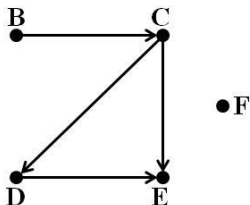
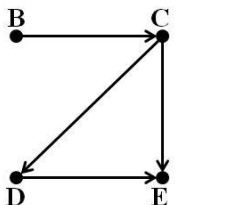
- $L \leftarrow$ Empty list that will contain the sorted elements
 - $S \leftarrow$ Set of all nodes with no incoming edge
1. **while** S is non-empty **do**
 2. remove a node n from S
 3. add n to *tail* of L
 4. **for each** node m with an edge e from n to m **do**
 5. remove edge e from the graph
 6. **if** m has no other incoming edges **then**
 7. insert m into S
 8. **if** graph has edges **then**
 9. return error (graph has at least one cycle)
 10. **else**
 11. return L (a topologically sorted order)

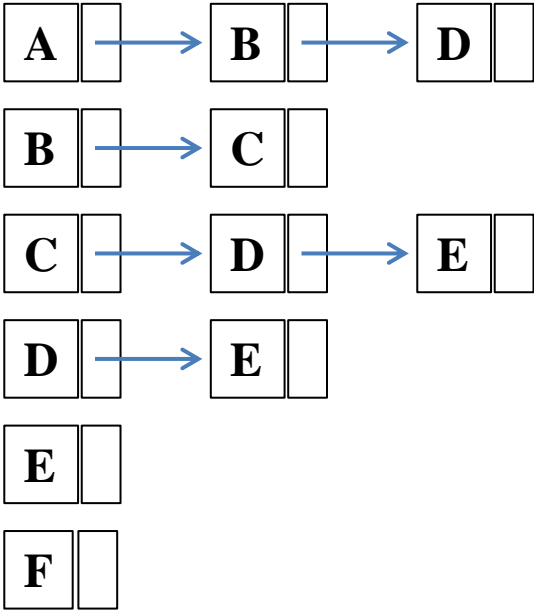
Example – 1



Vertex	In-degree
A	0
B	1
C	1
D	2
E	2
F	0

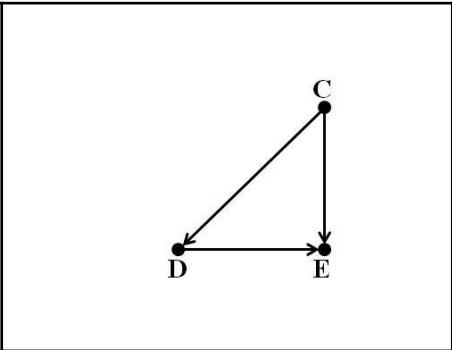
Contd...

L: \emptyset S: $A \rightarrow F$		<table><tr><th>Vertex</th><th>In-degree</th></tr><tr><td>A</td><td>0</td></tr><tr><td>B</td><td>1</td></tr><tr><td>C</td><td>1</td></tr><tr><td>D</td><td>2</td></tr><tr><td>E</td><td>2</td></tr><tr><td>F</td><td>0</td></tr></table>	Vertex	In-degree	A	0	B	1	C	1	D	2	E	2	F	0
Vertex	In-degree															
A	0															
B	1															
C	1															
D	2															
E	2															
F	0															
L: A S: $F \rightarrow B$		<table><tr><th>Vertex</th><th>In-degree</th></tr><tr><td>A</td><td>0</td></tr><tr><td>B</td><td>1 0</td></tr><tr><td>C</td><td>1</td></tr><tr><td>D</td><td>2 1</td></tr><tr><td>E</td><td>2</td></tr><tr><td>F</td><td>0</td></tr></table>	Vertex	In-degree	A	0	B	1 0	C	1	D	2 1	E	2	F	0
Vertex	In-degree															
A	0															
B	1 0															
C	1															
D	2 1															
E	2															
F	0															
L: $A \rightarrow F$ S: B		<table><tr><th>Vertex</th><th>In-degree</th></tr><tr><td>A</td><td>0</td></tr><tr><td>B</td><td>0 1</td></tr><tr><td>C</td><td>1</td></tr><tr><td>D</td><td>1 2</td></tr><tr><td>E</td><td>2</td></tr><tr><td>F</td><td>0</td></tr></table>	Vertex	In-degree	A	0	B	0 1	C	1	D	1 2	E	2	F	0
Vertex	In-degree															
A	0															
B	0 1															
C	1															
D	1 2															
E	2															
F	0															



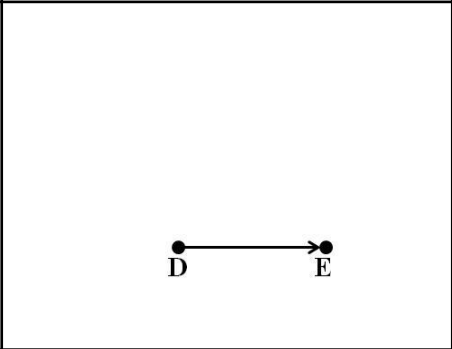
Contd...

L: $A \rightarrow F \rightarrow B$
S: C



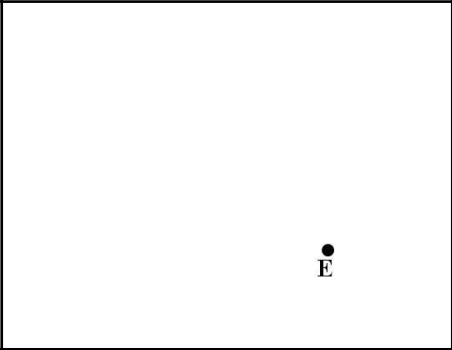
Vertex	In-degree
A	0
B	1 0
C	1 0
D	2 1
E	2
F	0

L: $A \rightarrow F \rightarrow B \rightarrow C$
S: D



Vertex	In-degree
A	0
B	1 0
C	1 0
D	2 1 0
E	2 1
F	0

L: $A \rightarrow F \rightarrow B \rightarrow C \rightarrow D$
S: E

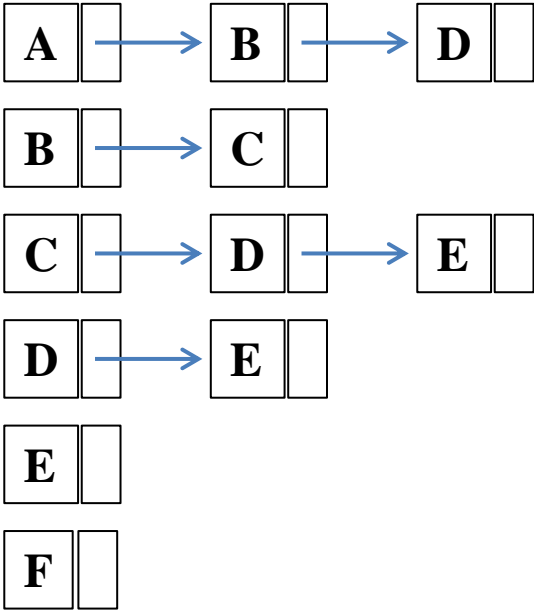


Vertex	In-degree
A	0
B	1 0
C	1 0
D	2 1 0
E	2 1 0
F	0

L: $A \rightarrow F \rightarrow B \rightarrow C \rightarrow D \rightarrow E$
S: \emptyset

NULL

Vertex	In-degree
A	0
B	1 0
C	1 0
D	2 1 0
E	2 1 0
F	0



Required
topological
sort is
A F B C D E

DFS(G)

```
1  for each vertex  $u \in G.V$ 
2       $u.color = \text{WHITE}$ 
3       $u.\pi = \text{NIL}$ 
4   $time = 0$ 
5  for each vertex  $u \in G.V$ 
6      if  $u.color == \text{WHITE}$ 
7          DFS-VISIT( $G, u$ )
```

TOPOLOGICAL-SORT(G)

```
1  call DFS( $G$ ) to compute finishing times  $v.f$  for each vertex  $v$ 
2  as each vertex is finished, insert it onto the front of a linked list
3  return the linked list of vertices
```

DFS-VISIT(G, u)

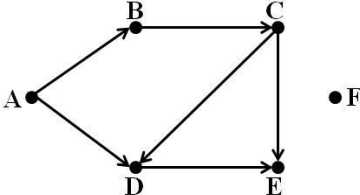
```
1   $time = time + 1$                                 // white vertex  $u$  has just been discovered
2   $u.d = time$ 
3   $u.color = \text{GRAY}$ 
4  for each  $v \in G.Adj[u]$                             // explore edge  $(u, v)$ 
5      if  $v.color == \text{WHITE}$ 
6           $v.\pi = u$ 
7          DFS-VISIT( $G, v$ )
8   $u.color = \text{BLACK}$                                 // blacken  $u$ ; it is finished
9   $time = time + 1$ 
10  $u.f = time$ 
```

Place this check within the for loop (lines 4 – 7).

else if $v.color == \text{GRAY}$
print "Cycle is present"
exit

Example 2

Starting vertex is 'C'



time = 0				
Vertex	color	π	d	f
A	WHITE	NIL		
B	WHITE	NIL		
C	WHITE	NIL		
D	WHITE	NIL		
E	WHITE	NIL		
F	WHITE	NIL		

time = 1				
Vertex	color	π	d	f
A	WHITE	NIL		
B	WHITE	NIL		
C	GRAY	NIL	1	
D	WHITE	NIL		
E	WHITE	NIL		
F	WHITE	NIL		

time = 2				
Vertex	color	π	d	f
A	WHITE	NIL		
B	WHITE	NIL		
C	GRAY	NIL	1	
D	GRAY	C	2	
E	WHITE	NIL		
F	WHITE	NIL		

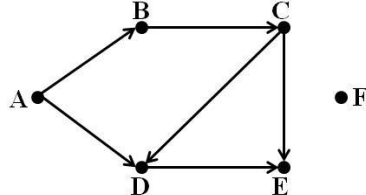
time = 3				
Vertex	color	π	d	f
A	WHITE	NIL		
B	WHITE	NIL		
C	GRAY	NIL	1	
D	GRAY	C	2	
E	GRAY	D	3	
F	WHITE	NIL		

time = 4				
Vertex	color	π	d	f
A	WHITE	NIL		
B	WHITE	NIL		
C	GRAY	NIL	1	
D	GRAY	C	2	
E	BLACK	D	3	4
F	WHITE	NIL		

time = 5				
Vertex	color	π	d	f
A	WHITE	NIL		
B	WHITE	NIL		
C	GRAY	NIL	1	
D	BLACK	C	2	5
E	BLACK	D	3	4
F	WHITE	NIL		

Contd...

Starting vertex is 'C'



time = 6				
Vertex	color	π	d	f
A	WHITE	NIL		
B	WHITE	NIL		
C	BLACK	NIL	1	6
D	BLACK	C	2	5
E	BLACK	D	3	4
F	WHITE	NIL		

time = 7				
Vertex	color	π	d	f
A	WHITE	NIL		
B	WHITE	NIL		
C	BLACK	NIL	1	6
D	BLACK	C	2	5
E	BLACK	D	3	4
F	GRAY	NIL	7	

time = 8				
Vertex	color	π	d	f
A	WHITE	NIL		
B	WHITE	NIL		
C	BLACK	NIL	1	6
D	BLACK	C	2	5
E	BLACK	D	3	4
F	BLACK	NIL	7	8

time = 9				
Vertex	color	π	d	f
A	GRAY	NIL	9	
B	WHITE	NIL		
C	BLACK	NIL	1	6
D	BLACK	C	2	5
E	BLACK	D	3	4
F	BLACK	NIL	7	8

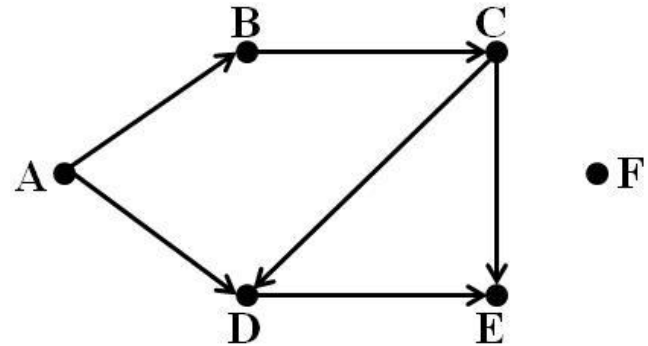
time = 10				
Vertex	color	π	d	f
A	GRAY	NIL	9	
B	GRAY	A	10	
C	BLACK	NIL	1	6
D	BLACK	C	2	5
E	BLACK	D	3	4
F	BLACK	NIL	7	8

time = 11				
Vertex	color	π	d	f
A	GRAY	NIL	9	
B	BLACK	A	10	11
C	BLACK	NIL	1	6
D	BLACK	C	2	5
E	BLACK	D	3	4
F	BLACK	NIL	7	8

Contd...

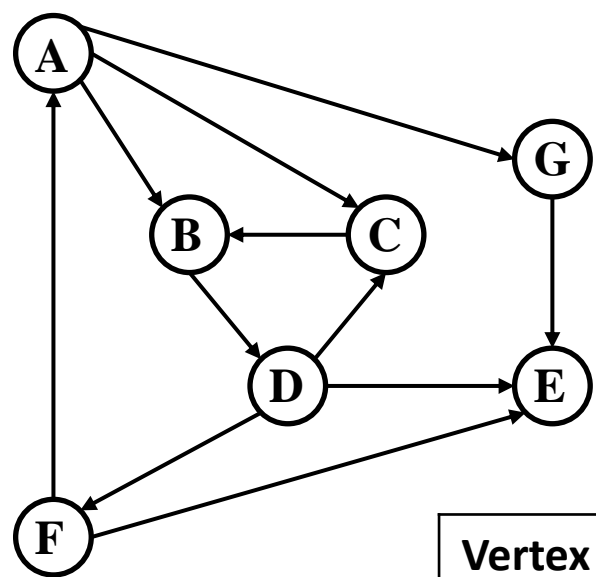
Starting vertex is 'C'

time = 12				
Vertex	color	π	d	f
A	BLACK	NIL	9	12
B	BLACK	A	10	11
C	BLACK	NIL	1	6
D	BLACK	C	2	5
E	BLACK	D	3	4
F	BLACK	NIL	7	8

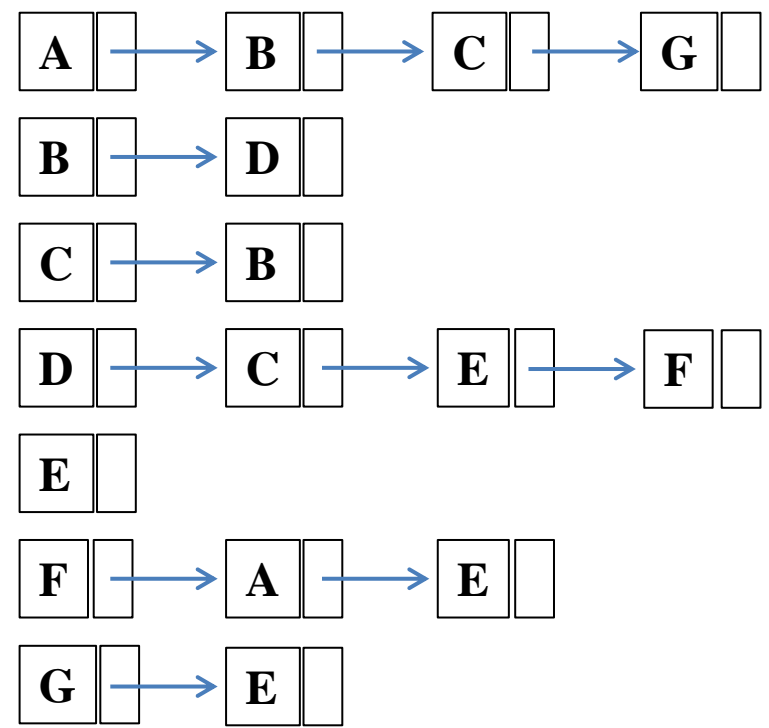


- Arrange vertices in decreasing order of their finishing time to get the topological sort.
- Topological sort is A B F C D E.

Example 3



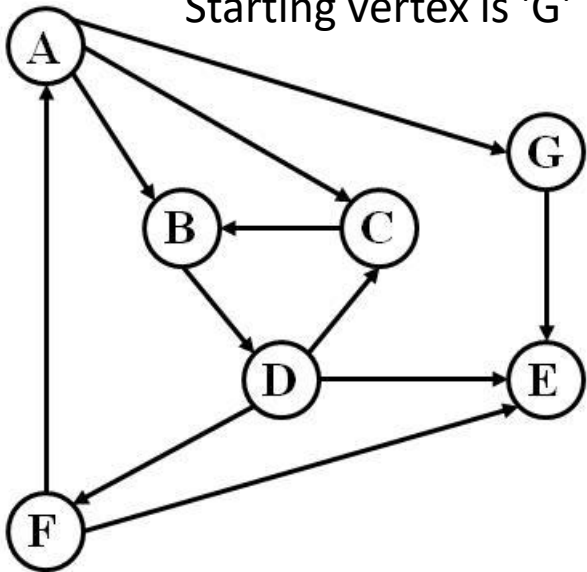
Vertex	In-degree
A	1
B	2
C	2
D	1
E	3
F	1
G	1



As there is no vertex with zero in-degree, kahn's algorithm will report cycle.

Example 4 – Using DFS

Starting vertex is 'G'



time = 0				
Vertex	color	π	d	f
A	WHITE	NIL		
B	WHITE	NIL		
C	WHITE	NIL		
D	WHITE	NIL		
E	WHITE	NIL		
F	WHITE	NIL		
G	GRAY	NIL		

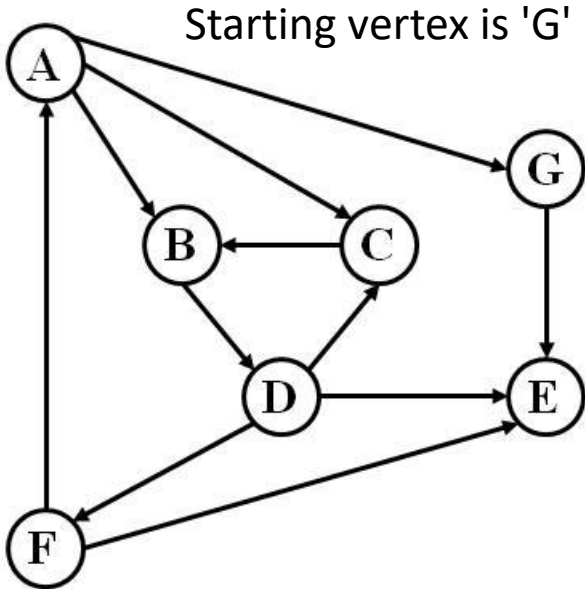
time = 1				
Vertex	color	π	d	f
A	WHITE	NIL		
B	WHITE	NIL		
C	WHITE	NIL		
D	WHITE	NIL		
E	WHITE	NIL		
F	WHITE	NIL		
G	GRAY	NIL	1	

time = 2				
Vertex	color	π	d	f
A	WHITE	NIL		
B	WHITE	NIL		
C	WHITE	NIL		
D	WHITE	NIL		
E	GRAY	G	2	
F	WHITE	NIL		
G	GRAY	NIL	1	

Contd...

time = 3				
Vertex	color	π	d	f
A	WHITE	NIL		
B	WHITE	NIL		
C	WHITE	NIL		
D	WHITE	NIL		
E	BLACK	G	2	3
F	WHITE	NIL		
G	GRAY	NIL	1	

time = 4				
Vertex	color	π	d	f
A	WHITE	NIL		
B	WHITE	NIL		
C	WHITE	NIL		
D	WHITE	NIL		
E	BLACK	G	2	3
F	WHITE	NIL		
G	BLACK	NIL	1	4

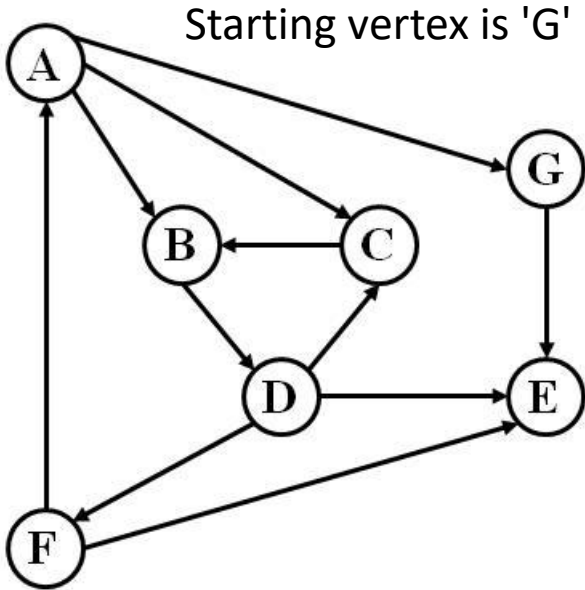


time = 5				
Vertex	color	π	d	f
A	WHITE	NIL		
B	WHITE	NIL		
C	WHITE	NIL		
D	WHITE	NIL		
E	BLACK	G	2	3
F	GRAY	NIL	5	
G	BLACK	NIL	1	4

Contd...

time = 6				
Vertex	color	π	d	f
A	GRAY	F	6	
B	WHITE	NIL		
C	WHITE	NIL		
D	WHITE	NIL		
E	BLACK	G	2	3
F	GRAY	NIL	5	
G	BLACK	NIL	1	4

time = 7				
Vertex	color	π	d	f
A	GRAY	F	6	
B	GRAY	A	7	
C	WHITE	NIL		
D	WHITE	NIL		
E	BLACK	G	2	3
F	GRAY	NIL	5	
G	BLACK	NIL	1	4

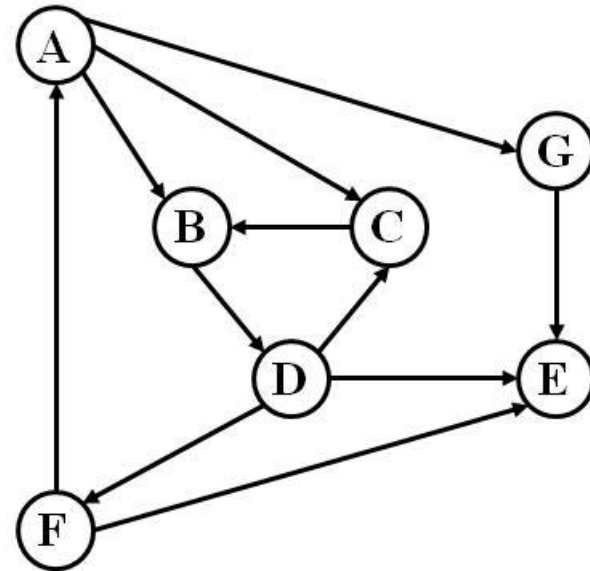


time = 8				
Vertex	color	π	d	f
A	GRAY	F	6	
B	GRAY	A	7	
C	WHITE	NIL		
D	GRAY	B	8	
E	BLACK	G	2	3
F	GRAY	NIL	5	
G	BLACK	NIL	1	4

Contd...

time = 9				
Vertex	color	π	d	f
A	GRAY	F	6	
B	GRAY	A	7	
C	GRAY	D	9	
D	GRAY	B	8	
E	BLACK	G	2	3
F	GRAY	NIL	5	
G	BLACK	NIL	1	4

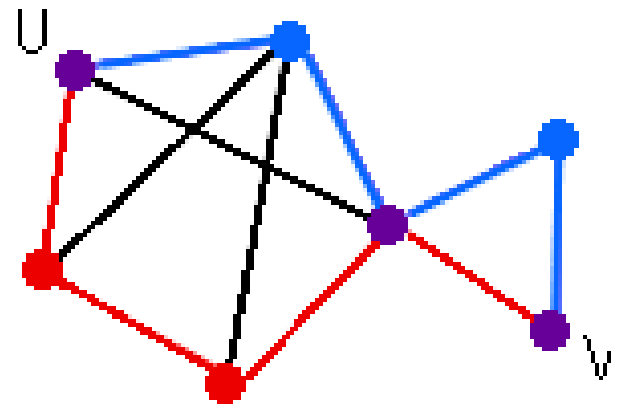
Starting vertex is 'G'



- From vertex 'C' we can only go to vertex 'B' and it's color is GRAY, thus the DFS procedure for topological sort reports cycle in the given graph and exits.

Some Definitions

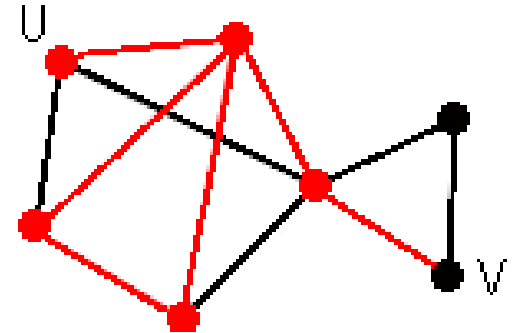
- Walk
 - An alternating sequence of vertices and connecting edges.
 - Can end on the same vertex on which it began or on a different vertex.
 - Can travel over any edge and any vertex any number of times.
- Path
 - A walk that does not include any vertex twice, except that its first and last vertices might be the same.



Contd...

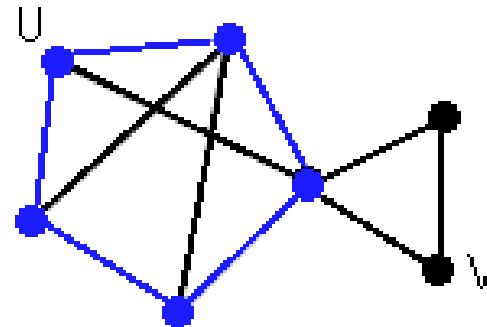
- Trail

- A walk that does not pass over the same edge twice.
- Might visit the same vertex twice, but only if it comes and goes from a different edge each time.



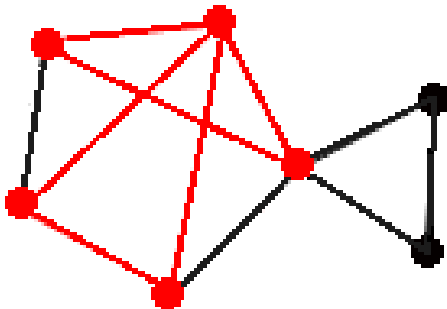
- Cycle

- Path that begins and ends on the same vertex.



- Circuit

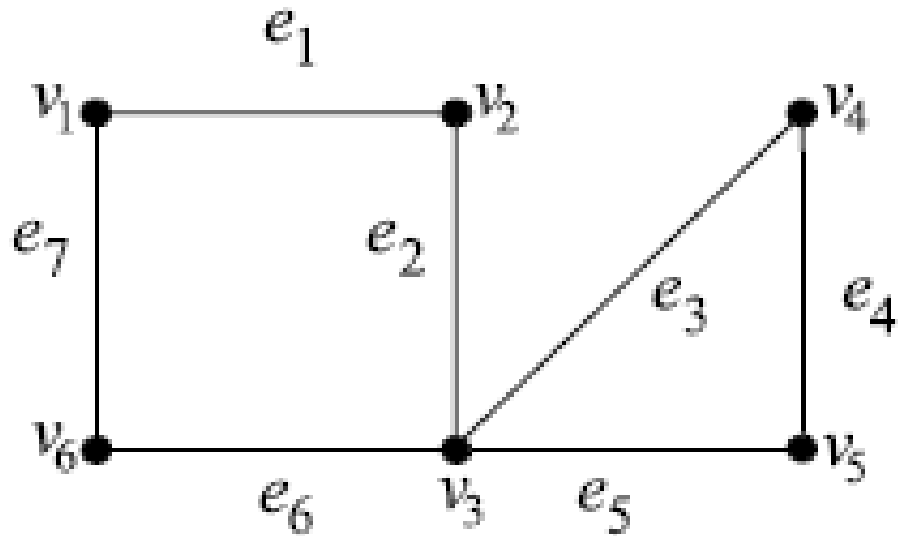
- Trail that begins and ends on the same vertex.



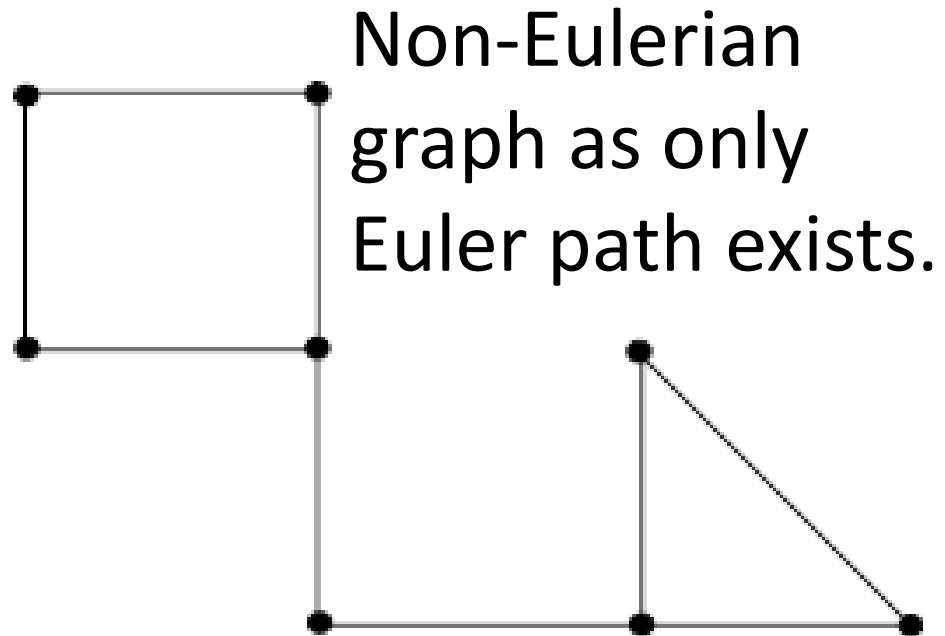
Euler Graphs

- An **Eulerian trail** (or **Eulerian path**) is a trail in a graph which visits every edge exactly once.
- A connected graph has an Eulerian path iff it has **at most two vertices of odd degree**.
- An **Eulerian circuit** (or **Eulerian cycle**) is an Eulerian trail which starts and ends on the same vertex.
- A graph containing an Eulerian circuit is called **Euler graph**.
- A connected graph G is an Euler graph iff **no vertex of G has odd degree**.

Example



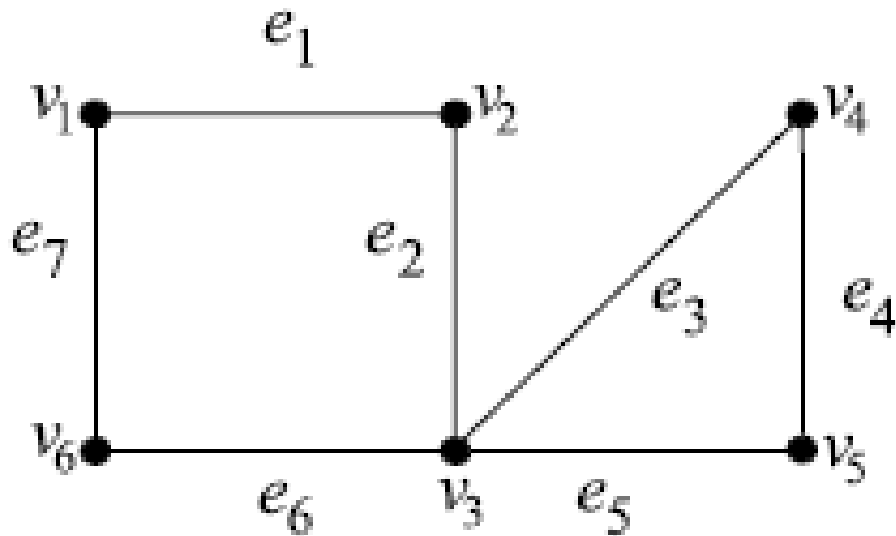
Eulerian graph as
Euler circuit exists.



Fleury's algorithm

1. Start at a vertex of odd degree (Euler path), or, if the graph has none, start with an arbitrarily chosen vertex (Euler circuit).
2. Choose the next edge in the path to be one whose deletion would not disconnect the graph. (Always choose the non-bridge in between a bridge and a non-bridge.)
3. Add that edge to the circuit, and delete it from the graph.
4. Continue until the circuit is complete.

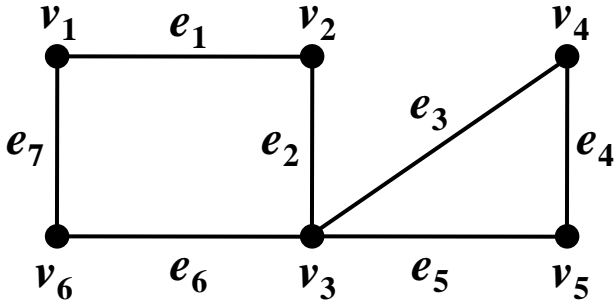
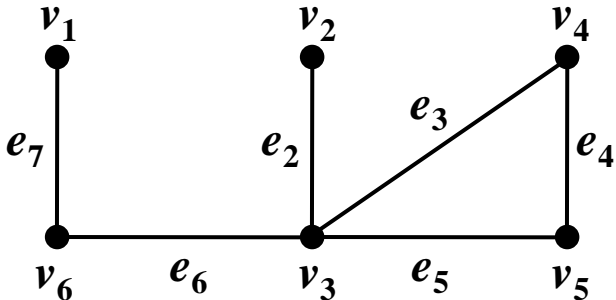
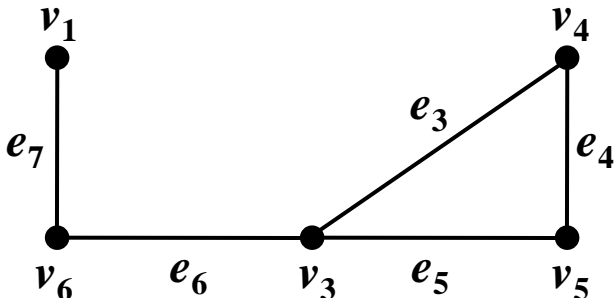
Example – 1



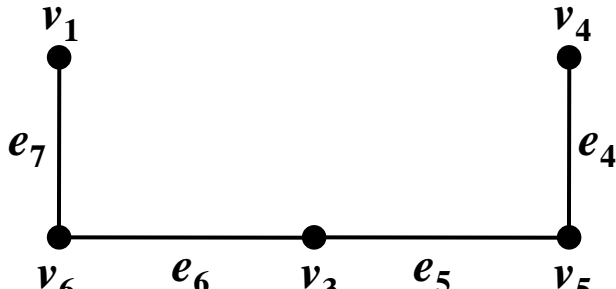
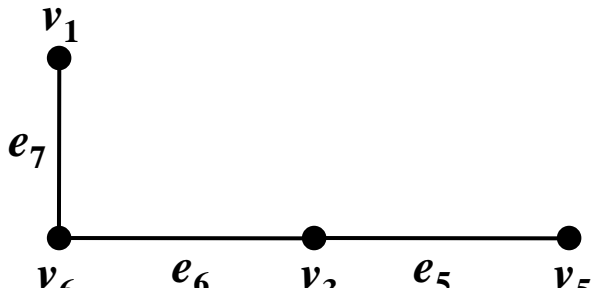
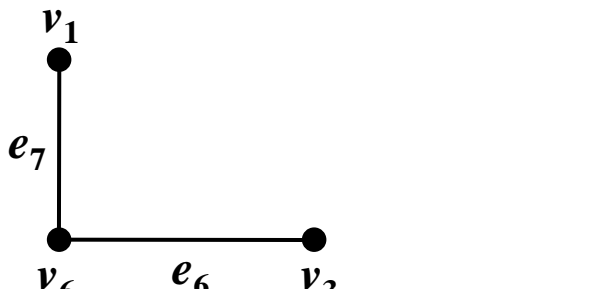
Vertex	Degree
v_1	2
v_2	2
v_3	4
v_4	2
v_5	2
v_6	2

- As degree of each vertex is even, thus Euler circuit exists.

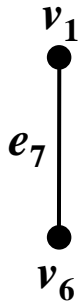
Contd...

Graph	Current Vertex	Trail
	v_1	NULL
	v_2	v_1v_2 or e_1
	v_3	$v_1v_2v_3$ or e_1e_2

Contd...

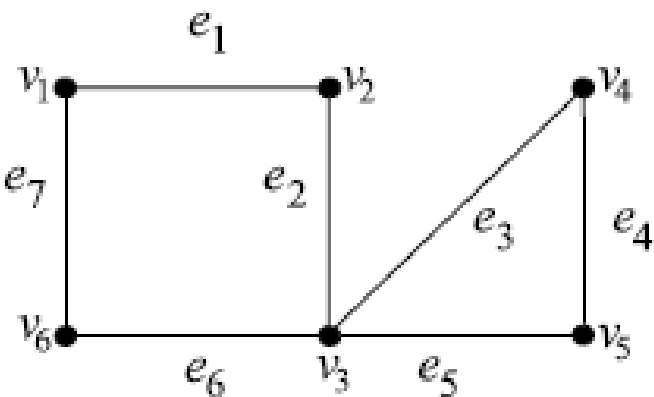
Graph	Current Vertex	Trail
	v_4 v_3v_6 or e_6 is a bridge and can't be selected.	$v_1v_2v_3v_4$ or $e_1e_2e_3$
	v_5	$v_1v_2v_3v_4v_5$ or $e_1e_2e_3e_4$
	v_3	$v_1v_2v_3v_4v_5v_3$ or $e_1e_2e_3e_4e_5$

Contd...

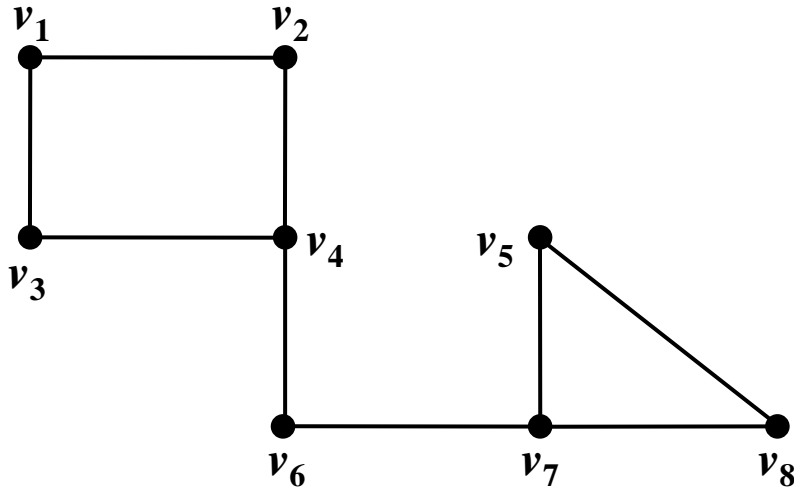
Graph	Current Vertex	Trail
	v_6	$v_1v_2v_3v_4v_5v_3v_6$ or $e_1e_2e_3e_4e_5e_6$
NULL	v_1	$v_1v_2v_3v_4v_5v_3v_6v_1$ or $e_1e_2e_3e_4e_5e_6e_7$

• Required Trail:

$v_1v_2v_3v_4v_5v_3v_6v_1$
or $e_1e_2e_3e_4e_5e_6e_7$



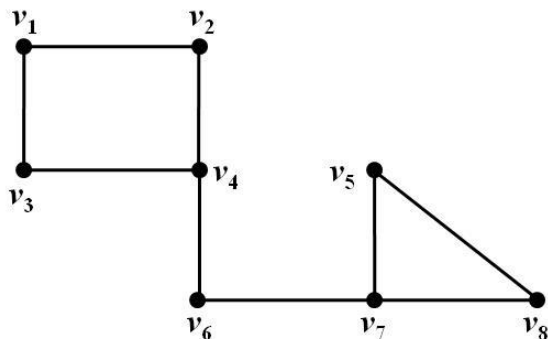
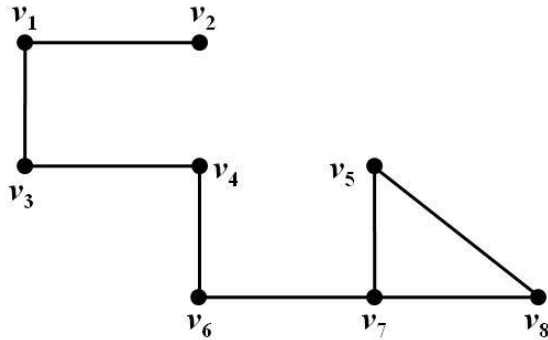
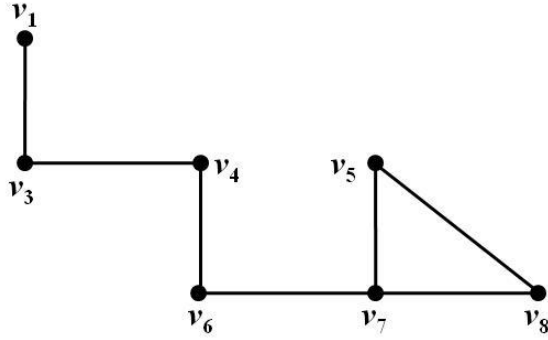
Example – 2



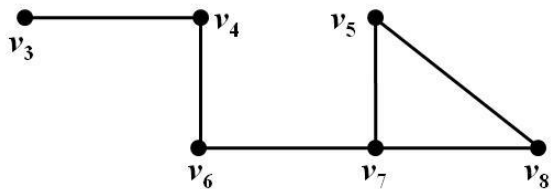
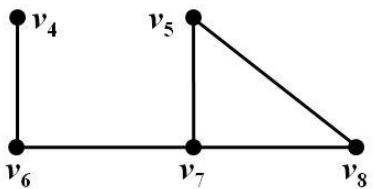
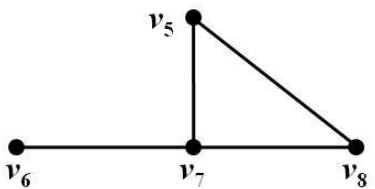
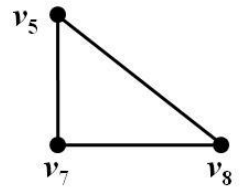
Vertex	Degree
v_1	2
v_2	2
v_3	4
v_4	3
v_5	2
v_6	2
v_7	3
v_8	2

- As degree of all the vertices is even except two (v_4 or v_7), thus Euler path exists.
- Start either from v_4 or v_7 .

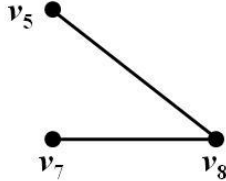
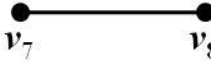
Contd...

Graph	Current Vertex	Trail
	v_4	NULL
	v_2 v_4v_6 is a bridge and can't be selected.	v_4v_2
	v_1	$v_4v_2v_1$

Contd...

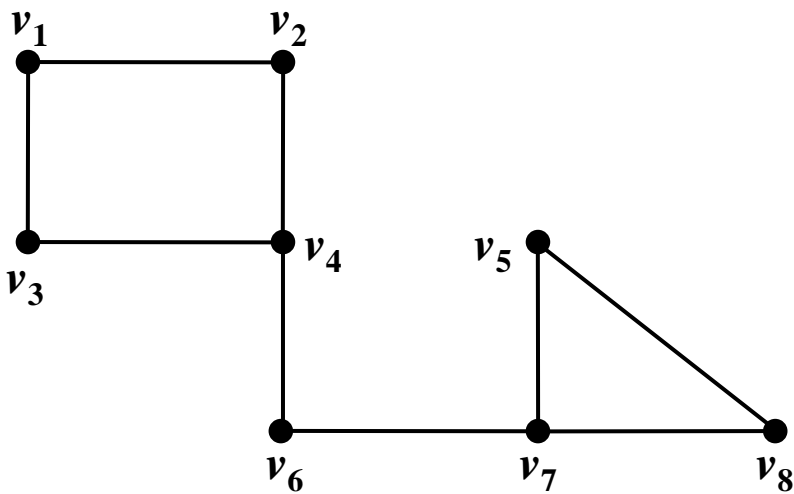
Graph	Current Vertex	Trail
	v_3	$v_4v_2v_1v_3$
	v_4	$v_4v_2v_1v_3v_4$
	v_6	$v_4v_2v_1v_3v_4v_6$
	v_7	$v_4v_2v_1v_3v_4v_6v_7$

Contd...

Graph	Current Vertex	Trail
	v_5	$v_4v_2v_1v_3v_4v_6v_7v_5$
	v_8	$v_4v_2v_1v_3v_4v_6v_7v_5v_8$
NULL	v_7	$v_4v_2v_1v_3v_4v_6v_7v_5v_8v_7$

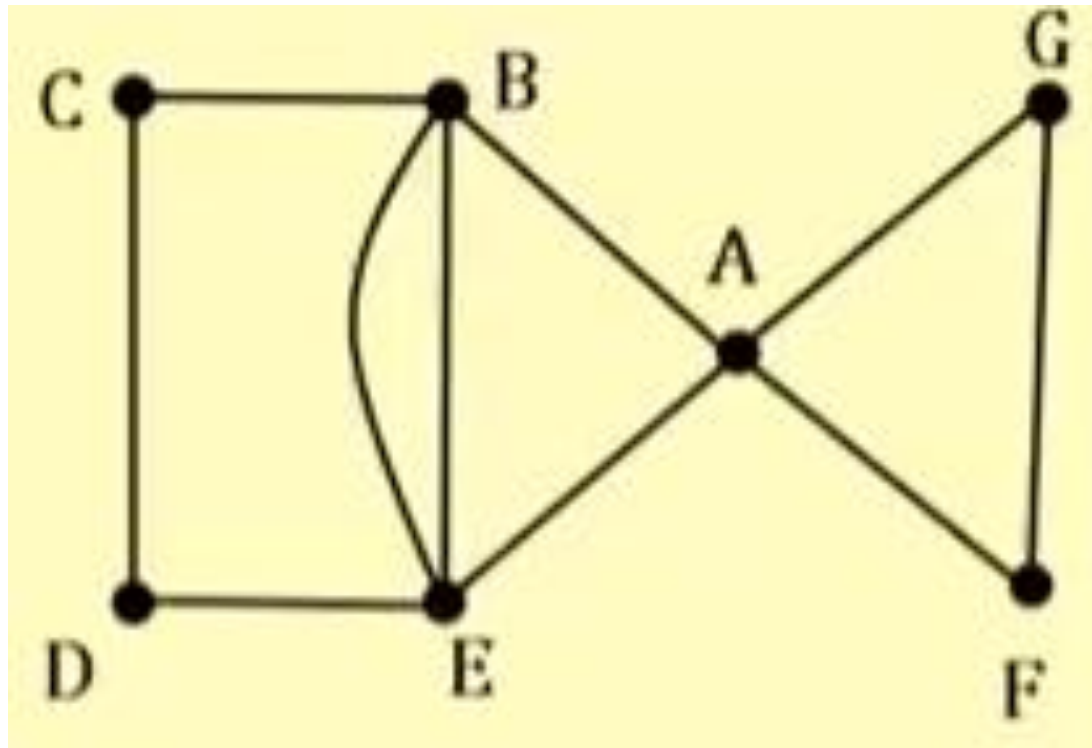
- Required Trail:

$v_4v_2v_1v_3v_4v_6v_7v_5v_8v_7$



Example – 3

- Find Euler circuit using Fleury's algorithm. Start at vertex A.



<https://www.youtube.com/watch?v=vvP4Fg4r-Ns>

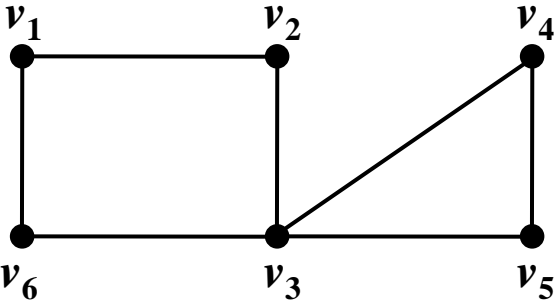
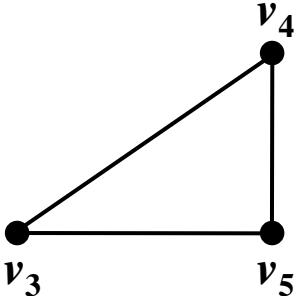
Contd...

- In Fleury's algorithm, the graph traversal is linear in the number of edges, i.e. $O(|E|)$, excluding the complexity of detecting bridges.
- With Tarjan's linear time bridge-finding algorithm, the time complexity is $O(|E|^2)$.
- With Thorup's dynamic bridge-finding algorithm, the time complexity gets improved to $O(|E|(\log |E|)^3 \log \log |E|)$.

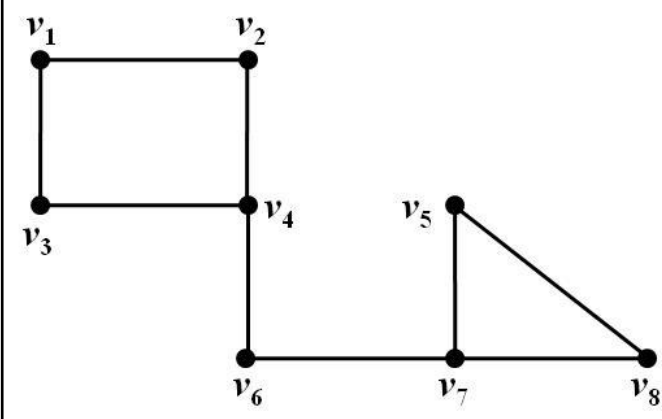
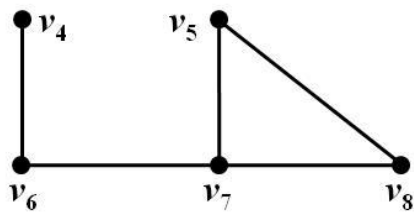
Hierholzer's algorithm

1. Choose any starting vertex v , and follow a trail of edges from that vertex until returning to v . The tour formed in this way is a closed tour, but may not cover all the vertices and edges of the initial graph.
2. As long as there exists a vertex u that belongs to the current tour but that has adjacent edges not part of the tour, start another trail from u , following unused edges until returning to u , and join the tour formed in this way to the previous tour.

Example – 1

Graph	Trail
	$v_1 v_2 \underline{v_3} v_6 v_1$
	$v_1 v_2 v_3 v_4 v_5 v_3 v_6 v_1$
Null	Required Trail: $v_1 v_2 v_3 v_4 v_5 v_3 v_6 v_1$

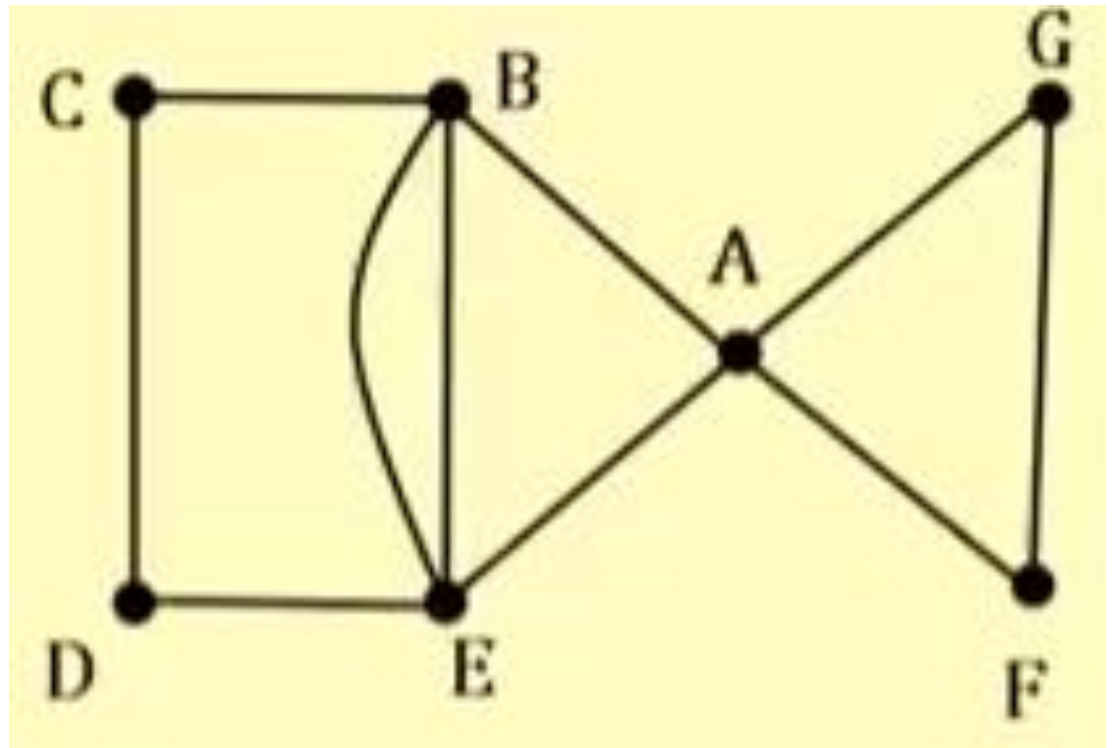
Example – 2

Graph	Trail
	$v_1 v_2 \underline{v_4} v_3$
	<p>Circuit starting and ending at vertex v_4 cannot be formed with the remaining edges.</p> <p>Algorithm terminates reporting graph is not Euler.</p>

Example – 3

- Find Euler circuit using Hierholzer's algorithm. Start at vertex A.

- Solution:
 - ABCDEA
 - AGFABCDEA
 - AGFABEBCDEA



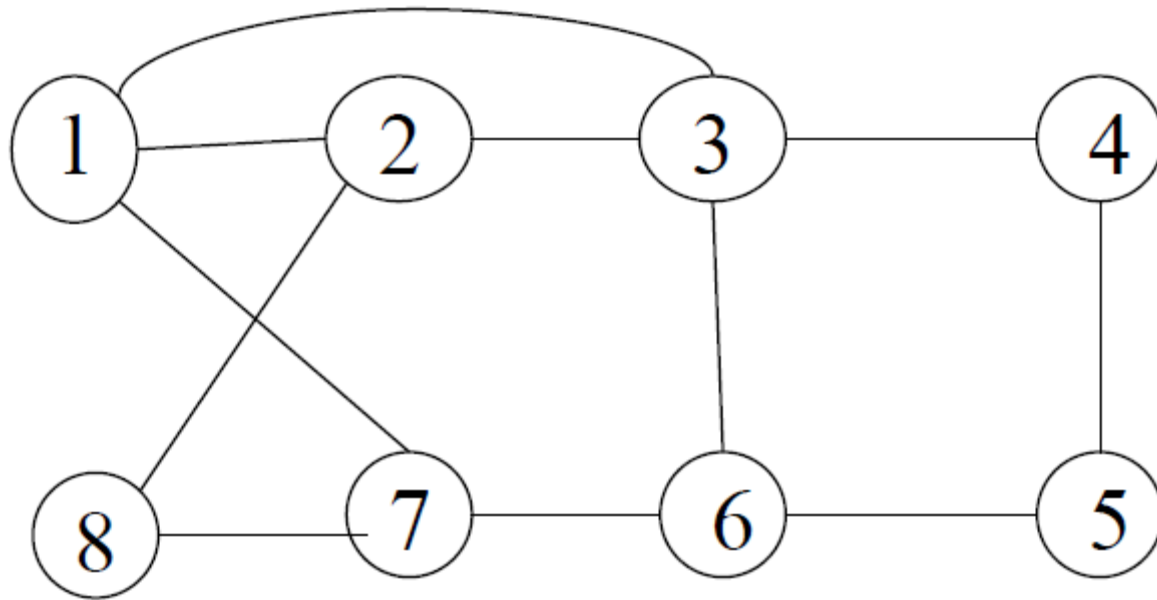
Contd...

- If doubly linked list is used to maintain
 - The set of unused edges incident to each vertex,
 - The list of vertices on the current tour that have unused edges, and
 - The tour itself.
- The individual operations of finding
 - Unused edges exiting each vertex,
 - A new starting vertex for a tour, and
 - Connecting two tours that share a vertex.
- May be performed in constant time, so the algorithm takes linear time, **$O(|E|)$** .

Hamiltonian Graphs

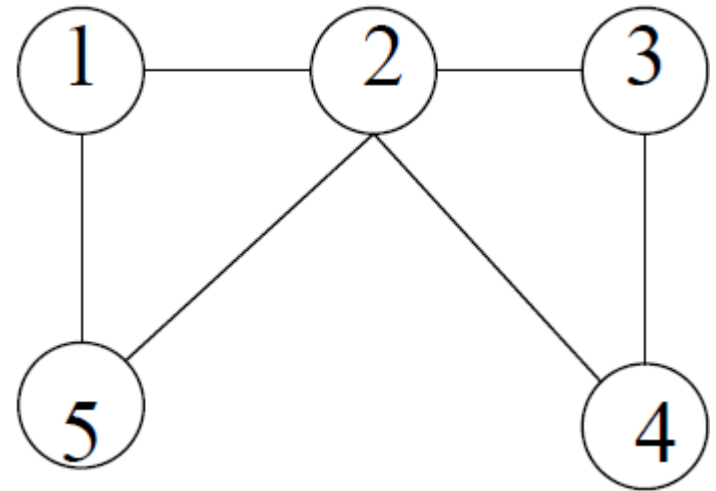
- A path passing through all the vertices of a graph is called a **Hamiltonian path**.
 - A graph containing a Hamiltonian path is said to be traceable.
- A cycle passing through all the vertices of a graph is called a **Hamiltonian cycle** (or **Hamiltonian circuit**).
- A graph containing a Hamiltonian cycle is called a **Hamiltonian graph**.
- **Hamiltonian path problem**: Determine the existence of Hamiltonian paths and cycles in graphs (**NP-complete**).

Example



- Hamiltonian graph
– 128765431

Not a Hamiltonian graph.



Solution 1 – Brute Force Search Algorithm

- A Hamiltonian Path in a graph having N vertices is nothing but a permutation of the vertices of the graph

$$[v_1, v_2, v_3, \dots, v_{N-1}, v_N]$$

such that there is an edge between v_i and v_{i+1} where $1 \leq i \leq N-1$.

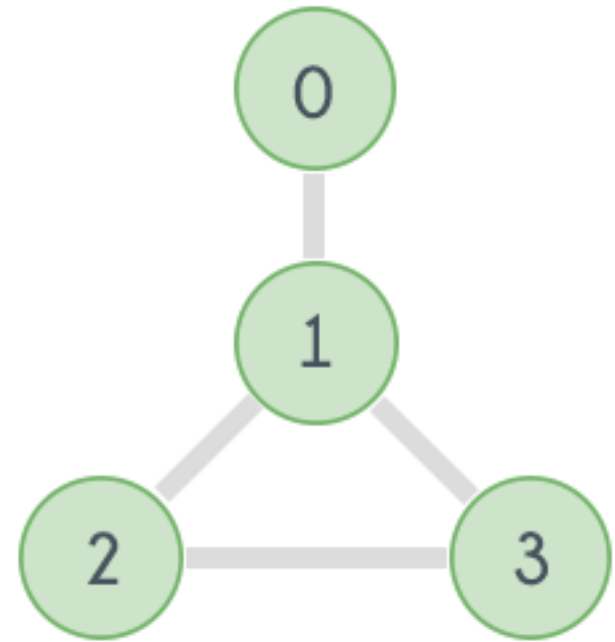
- So it can be checked for all permutations of the vertices whether any of them represents a Hamiltonian Path or not.

Contd...

```
function check_all_permutations(adj[][], n)
  for i = 0 to n
    p[i]=i
  while next permutation is possible
    valid = true
    for i = 0 to n-1
      if adj[p[i]][p[i+1]] == false
        valid = false
        break
    if valid == true
      print_permutation(p)
      return true
    p = get_next_permutation(p)
  return false
```

Example

Not a Hamiltonian graph as path exists and not a circuit.



- | | | | |
|-------------------|-------------|--------------------|--------------------|
| 1. 0-1-2-3 | 7. 1-0-2-3 | 13. 2-0-1-3 | 19. 3-0-1-2 |
| 2. 0-1-3-2 | 8. 1-0-3-2 | 14. 2-0-3-1 | 20. 3-0-2-1 |
| 3. 0-2-1-3 | 9. 1-2-0-3 | 15. 2-1-0-3 | 21. 3-1-0-2 |
| 4. 0-2-3-1 | 10. 1-2-3-0 | 16. 2-1-3-0 | 22. 3-1-2-0 |
| 5. 0-3-1-2 | 11. 1-3-0-2 | 17. 2-3-1-0 | 23. 3-2-0-1 |
| 6. 0-3-2-1 | 12. 1-3-2-0 | 18. 2-3-0-1 | 24. 3-2-1-0 |

Solution 2 – Backtracking

1. Create an empty path array and add vertex 0 to it.
2. Add other vertices, starting from the vertex 1.
3. Before adding a vertex, check for whether it is adjacent to the previously added vertex and not already added.
4. If such a vertex is found, add that vertex as part of the solution.
5. Otherwise, return false.

Contd...

```
1  Algorithm Hamiltonian( $k$ )
2  // This algorithm uses the recursive formulation of
3  // backtracking to find all the Hamiltonian cycles
4  // of a graph. The graph is stored as an adjacency
5  // matrix  $G[1 : n, 1 : n]$ . All cycles begin at node 1.
6  {
7      repeat
8      { // Generate values for  $x[k]$ .
9          NextValue( $k$ ); // Assign a legal next value to  $x[k]$ .
10         if ( $x[k] = 0$ ) then return;
11         if ( $k = n$ ) then write ( $x[1 : n]$ );
12         else Hamiltonian( $k + 1$ );
13     } until (false);
14 }
```

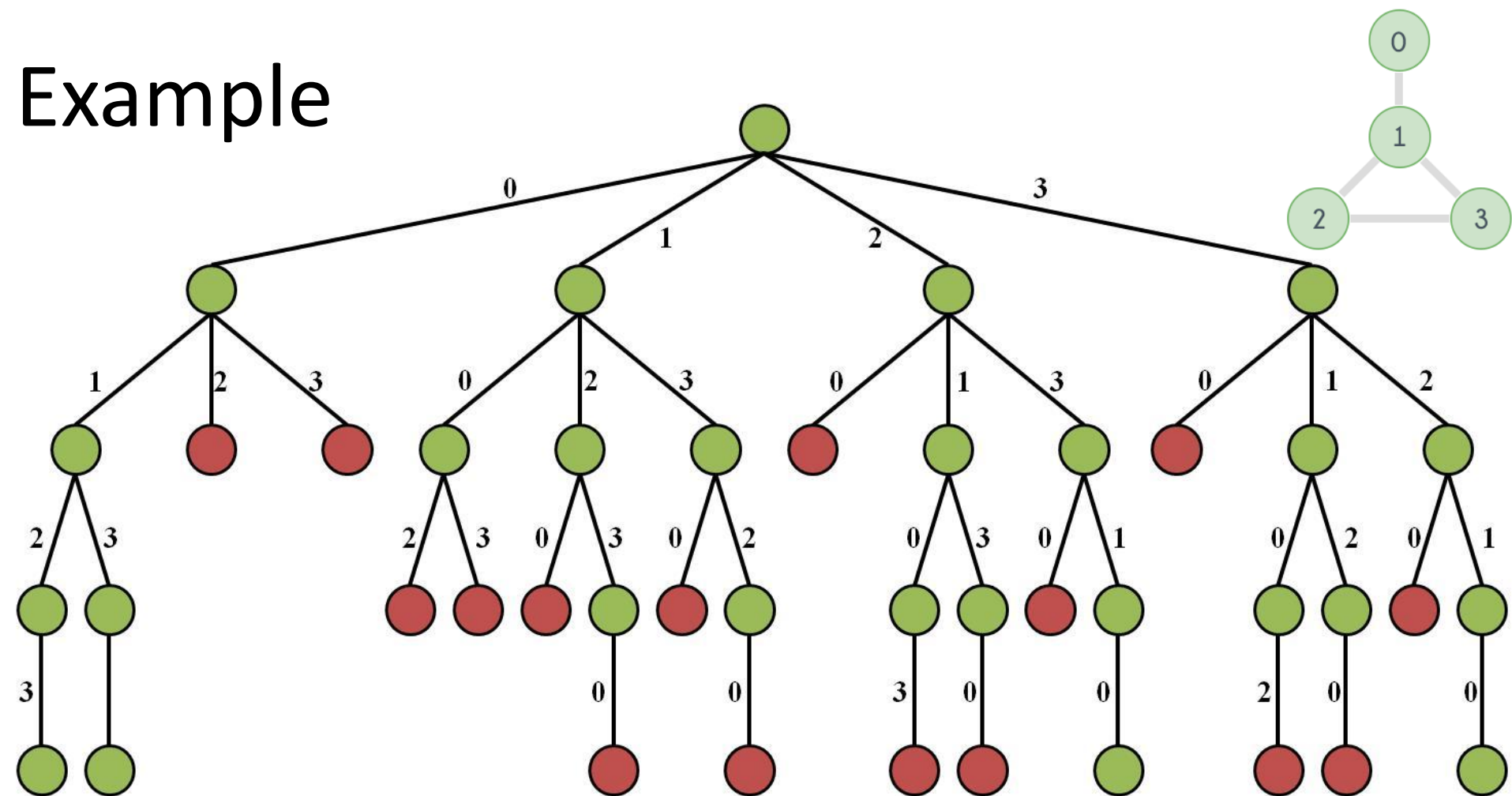
- Let,
 - Array x is a solution vector. $x[i]$ represents the i^{th} visited vertex of the proposed cycle.
 - $G[1:n, 1:n]$ is the adjacency matrix of the given graph.
- Initialize $x[1]$ to 1 (as vertex 1 is the starting vertex) and $x[2:n]$ to zero, then call Hamiltonian(2).


```

1  Algorithm NextValue( $k$ )
2  //  $x[1 : k - 1]$  is a path of  $k - 1$  distinct vertices. If  $x[k] = 0$ , then
3  // no vertex has as yet been assigned to  $x[k]$ . After execution,
4  //  $x[k]$  is assigned to the next highest numbered vertex which
5  // does not already appear in  $x[1 : k - 1]$  and is connected by
6  // an edge to  $x[k - 1]$ . Otherwise  $x[k] = 0$ . If  $k = n$ , then
7  // in addition  $x[k]$  is connected to  $x[1]$ .
8  {
9      repeat
10     {
11          $x[k] := (x[k] + 1) \bmod (n + 1)$ ; // Next vertex.
12         if ( $x[k] = 0$ ) then return;
13         if ( $G[x[k - 1], x[k]] \neq 0$ ) then
14             { // Is there an edge?
15                 for  $j := 1$  to  $k - 1$  do if ( $x[j] = x[k]$ ) then break;
16                 // Check for distinctness.
17                 if ( $j = k$ ) then // If true, then the vertex is distinct
18                     if ( $((k < n) \text{ or } ((k = n) \text{ and } G[x[n], x[1]] \neq 0))$ 
19                         then return;
20             }
21     } until (false);
22 }

```

Example



Only four paths are there which are ending with green leaf nodes. None of them is a circuit as starting and ending vertices are not adjacent in any of the obtained path, thus this graph is non-hamiltonian.

Solution 3 – Dynamic Programming

- Uses the concept of bit-masking to represent subset of vertices.
 - One bit per vertex.
 - Presence of 1 means a vertex is included in the subset.
- Compute matrix $dp[\text{\#ofVertices}][1..2^{\text{\#ofVertices}} - 1]$ and a *path* matrix to store the last visited vertex.
- Column number in matrix dp represents the corresponding bit-mask.
 - Example: If there are 4 vertices then column number 10 = $(1010)_2$, which represents vertices 2 and 4 are included in the subset. LSB is used for vertex 1 and MSB is used for vertex 4.

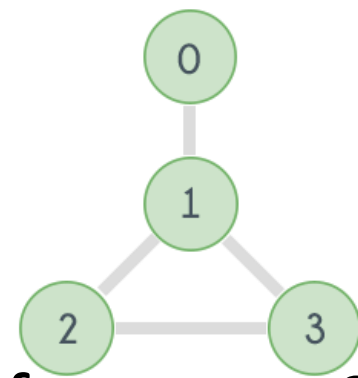
Contd...

- Use following equations to fill matrix dp .

$dp[i][j] = 1,$	<ul style="list-style-type: none">• if count of 1's in bit-mask of $j = 1$• i^{th} bit in the bit-mask of $j = 1$
$dp[i][j] = dp[k][j \text{ XOR } 2^i],$	<ul style="list-style-type: none">• if count of 1's in bit-mask of $j > 1$• i^{th} and k^{th} bit in the bit-mask of $j = 1$• $(i, j) \in E$
$dp[i][j] = 0,$	<ul style="list-style-type: none">• Otherwise.

- Iff last column (i.e. $2^{\text{\#ofVertices}} - 1$) of matrix dp has all 1's then graph is hamiltonian.

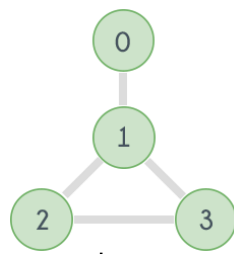
Example – 1



- Number of vertices = 4.
- Thus bit-mask has four bits. LSB is used for vertex 0 and MSB for vertex 3.
- $Dp[0..3][1..15]$
- Let starting vertex be zero. Last visited vertex is written within simple brackets in this example.
- Bit-mask with count of 1's = 1 are
 - $2^0 = 1 = (0001)_2 \Rightarrow dp[0][1] = 1$. Last visited vertex = \emptyset
 - $2^1 = 2 = (0010)_2 \Rightarrow dp[1][2] = 1$. Last visited vertex = 0
 - $2^2 = 4 = (0100)_2 \Rightarrow dp[2][4] = 1$. Last visited vertex = \emptyset
 - $2^3 = 8 = (1000)_2 \Rightarrow dp[3][8] = 1$. Last visited vertex = \emptyset

As only 0 and 1 are connected.

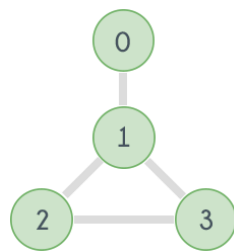
Contd...



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	1 (\emptyset)	0		0				0							
1	0	1 (v_0)		0				0							
2	0	0		1 (\emptyset)				0							
3	0	0		0				1 (\emptyset)							

- If a subset has single vertex, a path is possible.

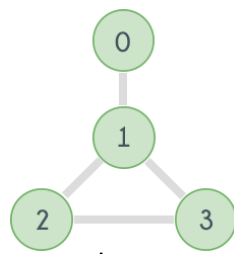
Contd...



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	1 (\emptyset)	0	1 (v_1)	0				0							
1	0	1 (v_0)	1 (v_0)	0				0							
2	0	0	0	1 (\emptyset)				0							
3	0	0	0	0				1 (\emptyset)							

- Now $3 = (0011)_2$, subset is $\{v_0, v_1\}$.
 - v_2 and v_3 are not present, so $dp[2][3] = dp[3][3] = 0$.
 - $dp[0][3]$ means vertex v_1 is already present and we are trying to add v_0 . Since path exists after adding v_0 , so put 1 and last visited vertex will be v_1 .
 - Similarly, $dp[1][3]$ means vertex v_0 is already present and we are trying to add v_1 . Since path exists after adding v_1 , so put 1 and last visited vertex will be v_0 .

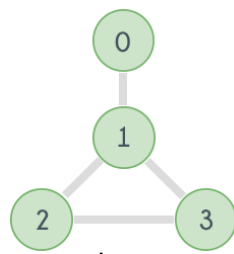
Contd...



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	1 (\emptyset)	0	1 (v_1)	0	0			0							
1	0	1 (v_0)	1 (v_0)	0	0			0							
2	0	0	0	1 (\emptyset)	0			0							
3	0	0	0	0	0			1 (\emptyset)							

- Now $5 = (0101)_2$, subset is $\{v_0, v_2\}$.
 - v_1 and v_3 are not present, so $dp[1][5] = dp[3][5] = 0$.
 - $dp[0][5]$ means vertex v_2 is already present and we are trying to add v_0 . No path exists after adding v_0 , so put 0.
 - Similarly, $dp[2][5]$ means vertex v_0 is already present and we are trying to add v_2 . No path exists after adding v_2 , so put 0.

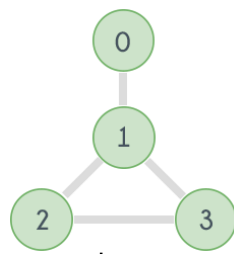
Contd...



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	1 (\emptyset)	0	1 (v_1)	0	0	0		0							
1	0	1 (v_0)	1 (v_0)	0	0	1 (v_2)		0							
2	0	0	0	1 (\emptyset)	0	1 (v_1)		0							
3	0	0	0	0	0	0		1 (\emptyset)							

- Now $6 = (0110)_2$, subset is $\{v_1, v_2\}$.
 - v_0 and v_3 are not present, so $dp[0][6] = dp[3][6] = 0$.
 - $dp[1][6]$ means vertex v_2 is already present and we are trying to add v_1 . Since path exists after adding v_1 , so put 1 and last visited vertex will be v_2 .
 - Similarly, $dp[2][6]$ means vertex v_1 is already present and we are trying to add v_2 . Since path exists after adding v_2 , so put 1 and last visited vertex will be v_1 .

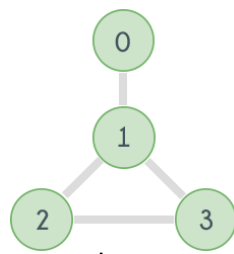
Contd...



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	1 (\emptyset)	0	1 (v_1)	0	0	0	1 (v_1)	0							
1	0	1 (v_0)	1 (v_0)	0	0	1 (v_2)	0	0							
2	0	0	0	1 (\emptyset)	0	1 (v_1)	1 (v_1)	0							
3	0	0	0	0	0	0	0	1 (\emptyset)							

- Now $7 = (0111)_2$, subset is $\{v_0, v_1, v_2\}$.
 - v_3 is not present, so $dp[3][7] = 0$.
 - $dp[0][7]$ means vertex v_1 and v_2 are already present and we are trying to add v_0 . Since path exists after adding v_0 via v_1 so put 1 and last visited vertex will be v_1 .
 - $dp[1][7]$ means vertex v_0 and v_2 are already present and we are trying to add v_1 . Since no path exists between v_0 and v_2 initially, so put 0.
 - $dp[2][7]$ means vertex v_0 and v_1 are already present and we are trying to add v_2 . Since path exists after adding v_2 via v_1 so put 1 and last visited vertex will be v_1 .

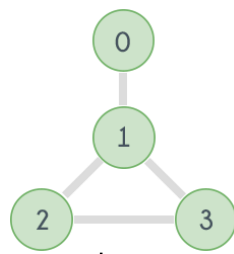
Contd...



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	1 (\emptyset)	0	1 (v_1)	0	0	0	1 (v_1)	0	0						
1	0	1 (v_0)	1 (v_0)	0	0	1 (v_2)	0	0	0						
2	0	0	0	1 (\emptyset)	0	1 (v_1)	1 (v_1)	0	0						
3	0	0	0	0	0	0	0	1 (\emptyset)	0						

- Now $9 = (1001)_2$, subset is $\{v_0, v_3\}$.
 - v_1 and v_2 are not present, so $dp[1][9] = dp[2][9] = 0$.
 - $dp[0][9]$ means vertex v_3 is already present and we are trying to add v_0 . No path exists after adding v_0 , so put 0.
 - Similarly, $dp[3][9]$ means vertex v_0 is already present and we are trying to add v_3 . No path exists after adding v_3 , so put 0.

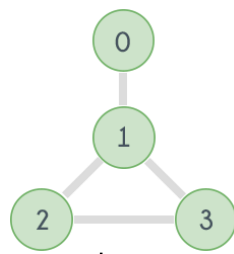
Contd...



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	1 (\emptyset)	0	1 (v_1)	0	0	0	1 (v_1)	0	0	0					
1	0	1 (v_0)	1 (v_0)	0	0	1 (v_2)	0	0	0	1 (v_3)					
2	0	0	0	1 (\emptyset)	0	1 (v_1)	1 (v_1)	0	0	0					
3	0	0	0	0	0	0	0	1 (\emptyset)	0	1 (v_1)					

- Now $10 = (1010)_2$, subset is $\{v_1, v_3\}$.
 - v_0 and v_2 are not present, so $dp[0][10] = dp[2][10] = 0$.
 - $dp[1][10]$ means vertex v_3 is already present and we are trying to add v_1 . Since path exists after adding v_1 , so put 1 and last visited vertex will be v_3 .
 - Similarly, $dp[3][10]$ means vertex v_1 is already present and we are trying to add v_3 . Since path exists after adding v_3 , so put 1 and last visited vertex will be v_1 .

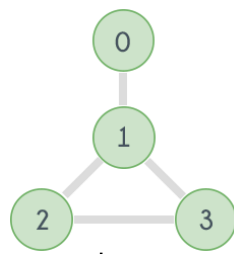
Contd...



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	1 (\emptyset)	0	1 (v_1)	0	0	0	1 (v_1)	0	0	0	1 (v_1)				
1	0	1 (v_0)	1 (v_0)	0	0	1 (v_2)	0	0	0	1 (v_3)	0				
2	0	0	0	1 (\emptyset)	0	1 (v_1)	1 (v_1)	0	0	0	0				
3	0	0	0	0	0	0	0	1 (\emptyset)	0	1 (v_1)	1 (v_1)				

- Now $11 = (1011)_2$, subset is $\{v_0, v_1, v_3\}$.
 - v_2 is not present, so $dp[2][11] = 0$.
 - $dp[0][11]$ means vertex v_1 and v_3 are already present and we are trying to add v_0 . Since path exists after adding v_0 via v_1 so put 1 and last visited vertex will be v_1 .
 - $dp[1][11]$ means vertex v_0 and v_3 are already present and we are trying to add v_1 . Since no path exists between v_0 and v_3 initially, so put 0.
 - $dp[3][11]$ means vertex v_0 and v_1 are already present and we are trying to add v_3 . Since path exists after adding v_3 via v_1 so put 1 and last visited vertex will be v_1 .

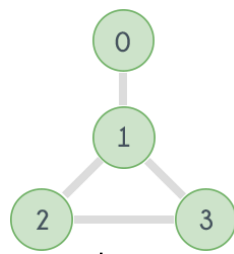
Contd...



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	1 (\emptyset)	0	1 (v_1)	0	0	0	1 (v_1)	0	0	0	1 (v_1)	0			
1	0	1 (v_0)	1 (v_0)	0	0	1 (v_2)	0	0	0	1 (v_3)	0	0			
2	0	0	0	1 (\emptyset)	0	1 (v_1)	1 (v_1)	0	0	0	0	1 (v_3)			
3	0	0	0	0	0	0	0	1 (\emptyset)	0	1 (v_1)	1 (v_1)	1 (v_2)			

- Now $12 = (1100)_2$, subset is $\{v_2, v_3\}$.
 - v_0 and v_1 are not present, so $dp[0][12] = dp[1][12] = 0$.
 - $dp[2][12]$ means vertex v_3 is already present and we are trying to add v_2 . Since path exists after adding v_2 , so put 1 and last visited vertex will be v_3 .
 - Similarly, $dp[3][12]$ means vertex v_2 is already present and we are trying to add v_3 . Since path exists after adding v_3 , so put 1 and last visited vertex will be v_2 .

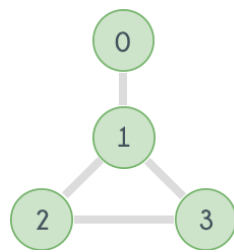
Contd...



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	1 (\emptyset)	0	1 (v_1)	0	0	0	1 (v_1)	0	0	0	1 (v_1)	0	0		
1	0	1 (v_0)	1 (v_0)	0	0	1 (v_2)	0	0	0	1 (v_3)	0	0	0		
2	0	0	0	1 (\emptyset)	0	1 (v_1)	1 (v_1)	0	0	0	0	1 (v_3)	0		
3	0	0	0	0	0	0	0	1 (\emptyset)	0	1 (v_1)	1 (v_1)	1 (v_2)	0		

- Now $13 = (1101)_2$, subset is $\{v_0, v_2, v_3\}$.
 - v_1 is not present, so $dp[1][13] = 0$.
 - $dp[0][13]$ means vertex v_2 and v_3 are already present and we are trying to add v_0 . Since v_0 is neither connected to v_2 nor v_3 , so no path exists after adding v_0 thus put 0.
 - $dp[2][13]$ means vertex v_0 and v_3 are already present and we are trying to add v_2 . Since no path exists between v_0 and v_3 initially, so put 0.
 - $dp[3][13]$ means vertex v_0 and v_2 are already present and we are trying to add v_3 . Since no path exists between v_0 and v_2 initially, so put 0.

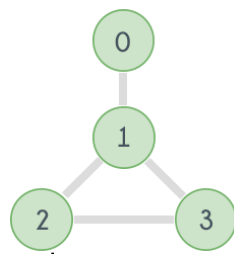
Contd...



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	1 (\emptyset)	0	1 (v_1)	0	0	0	1 (v_1)	0	0	0	1 (v_1)	0	0	0	
1	0	1 (v_0)	1 (v_0)	0	0	1 (v_2)	0	0	0	1 (v_3)	0	0	0	1 (v_2/v_3)	
2	0	0	0	1 (\emptyset)	0	1 (v_1)	1 (v_1)	0	0	0	0	1 (v_3)	0	1 (v_1/v_3)	
3	0	0	0	0	0	0	0	1 (\emptyset)	0	1 (v_1)	1 (v_1)	1 (v_2)	0	1 (v_1/v_2)	

- Now $14 = (1110)_2$, subset is $\{v_1, v_2, v_3\}$.
 - v_0 is not present, so $dp[0][14] = 0$.
 - $dp[1][14]$ means vertex v_2 and v_3 are already present and we are trying to add v_1 . Since path exists after adding v_1 via v_2 as well as via v_3 , so put 1 and last visited vertex will be v_2/v_3 .
 - $dp[2][14]$ means vertex v_1 and v_3 are already present and we are trying to add v_2 . Since path exists after adding v_2 via v_1 as well as via v_3 , so put 1 and last visited vertex will be v_1/v_3 .
 - $dp[3][14]$ means vertex v_1 and v_2 are already present and we are trying to add v_3 . Since path exists after adding v_3 via v_1 as well as via v_2 , so put 1 and last visited vertex will be v_1/v_2 .

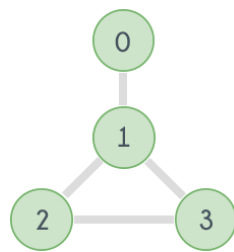
Contd...



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	1 (\emptyset)	0	1 (v_1)	0	0	0	1 (v_1)	0	0	0	1 (v_1)	0	0	0	1 (v_1)
1	0	1 (v_0)	1 (v_0)	0	0	1 (v_2)	0	0	0	1 (v_3)	0	0	0	1 (v_2/v_3)	0
2	0	0	0	1 (\emptyset)	0	1 (v_1)	1 (v_1)	0	0	0	0	1 (v_3)	0	1 (v_1/v_3)	1 (v_1/v_3)
3	0	0	0	0	0	0	0	1 (\emptyset)	0	1 (v_1)	1 (v_1)	1 (v_2)	0	1 (v_1/v_2)	1 (v_1/v_2)

- Now $15 = (1111)_2$, subset is $\{v_0, v_1, v_2, v_3\}$.
 - $dp[0][15]$ means vertex v_1, v_2 , and v_3 are already present and we are trying to add v_0 . Since path exists after adding v_0 via v_1 , so put 1 and last visited vertex will be v_1 .
 - $dp[1][15]$ means vertex v_0, v_2 , and v_3 are already present and we are trying to add v_1 . Since no path exists between v_0, v_2 , and v_3 initially, so put 0.
 - $dp[2][15]$ means vertex v_0, v_1 , and v_3 are already present and we are trying to add v_2 . Since path exists after adding v_2 via v_1 as well as via v_3 , so put 1 and last visited vertex will be v_1/v_3 .
 - $dp[3][15]$ means vertex v_0, v_1 , and v_2 are already present and we are trying to add v_3 . Since path exists after adding v_3 via v_1 as well as via v_2 , so put 1 and last visited vertex will be v_1/v_2 .

Contd...



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	(\emptyset)	0	(v_1)	0	0	0	(v_1)	0	0	0	(v_1)	0	0	0	(v_1)
1	0	(v_0)	(v_0)	0	0	(v_2)	0	0	0	(v_3)	0	0	0	(v_2/v_3)	0
2	0	0	0	(\emptyset)	0	(v_1)	(v_1)	0	0	0	0	(v_3)	0	(v_1/v_3)	(v_1/v_3)
3	0	0	0	0	0	0	0	(\emptyset)	0	(v_1)	(v_1)	(v_2)	0	(v_1/v_2)	(v_1/v_2)

- All entries in the last column are not '1', so the graph is not hamiltonian.
 - To check if hamiltonian path exists, one has to trace the path.
 - Note:
 - $p(S,x) = y$, means reaching vertex x after visiting all the vertices in S via vertex y .
 - Use the information from the column where bits corresponding to vertices $S \cup x$ are set.
 - $p(\emptyset,x)$ means reaching vertex x from the starting vertex.
 - Assuming that the cycle exists, thus the end vertex should be same as the starting vertex, i.e. 0 in the considered example.
 - $p(\{v_1, v_2, v_3\}, v_0) = v_1$
 - $p(\{v_2, v_3\}, v_1) = v_2$
 - $p(\{v_3\}, v_2) = v_3$
 - $p(\emptyset, v_3) = \text{Not possible.}$
 - Path is $v_3 \rightarrow v_2 \rightarrow v_1 \rightarrow v_0$
- OR

OR

OR

OR

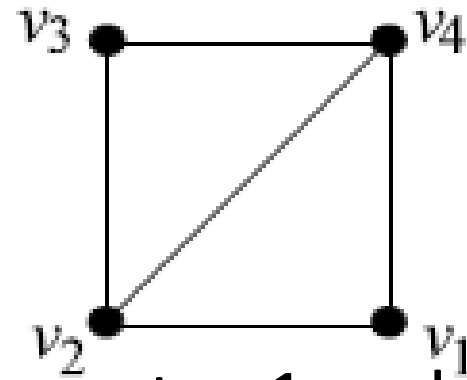
$p(\{v_2, v_3\}, v_1) = v_3$

$p(\{v_2\}, v_3) = v_2$

$p(\emptyset, v_2) = \text{Not possible.}$

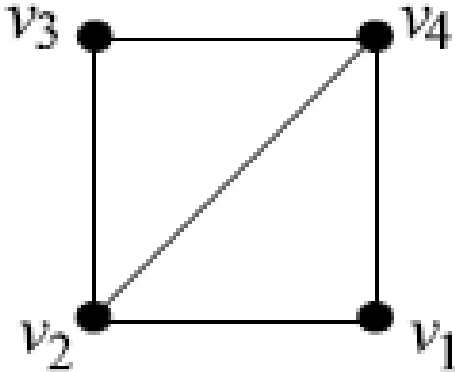
Path is $v_2 \rightarrow v_3 \rightarrow v_1 \rightarrow v_0$

Example – 2



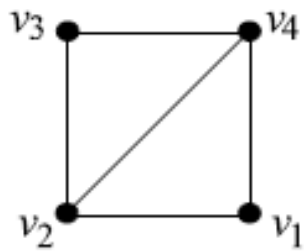
- Number of vertices = 4.
- Thus bit-mask has four bits. LSB is used for vertex 1 and MSB for vertex 4.
- $dp[1..4][1..15]$
- Let starting vertex be v_1 . Last visited vertex is written within simple brackets in this example.
- Bit-mask with count of 1's = 1 are
 - $2^0 = 1 = (0001)_2 \Rightarrow dp[1][1] = 1$. Last visited vertex = \emptyset
 - $2^1 = 2 = (0010)_2 \Rightarrow dp[2][2] = 1$. Last visited vertex = v_1
 - $2^2 = 4 = (0100)_2 \Rightarrow dp[3][4] = 1$. Last visited vertex = \emptyset
 - $2^3 = 8 = (1000)_2 \Rightarrow dp[4][8] = 1$. Last visited vertex = v_1

Contd...

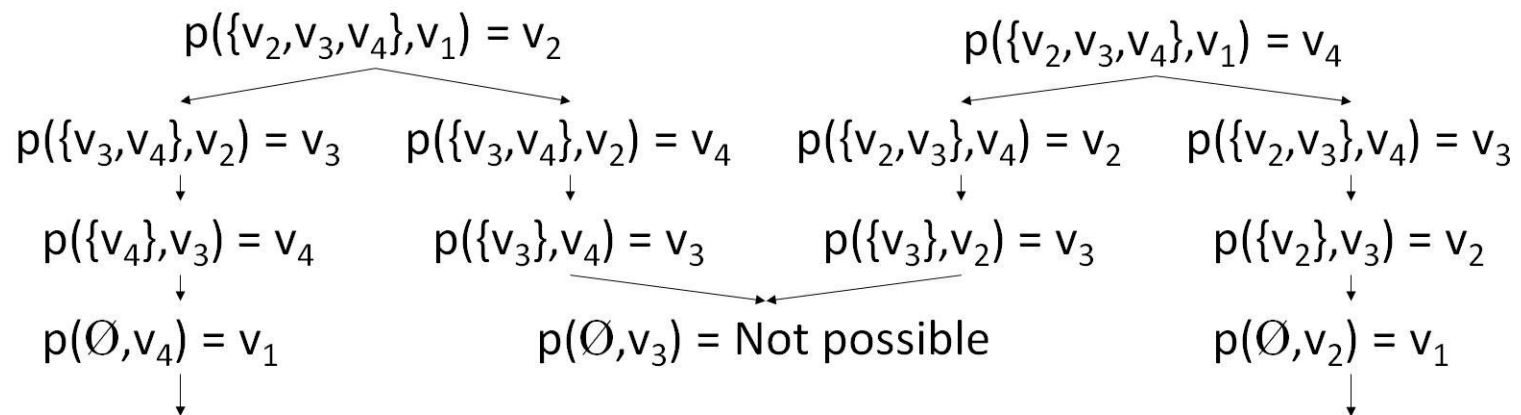


	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	1 (\emptyset)	0	1 (v_2)	0	0	0	1 (v_2)	0	1 (v_4)	0	1 (v_2/v_4)	0	1 (v_4)	0	1 (v_2/v_4)
2	0	1 (v_1)	1 (v_1)	0	0	1 (v_3)	0	0	0	1 (v_4)	1 (v_1/v_4)	0	0	1 (v_3/v_4)	1 ($v_1/v_3/v_4$)
3	0	0	0	1 (\emptyset)	0	1 (v_2)	1 (v_2)	0	0	0	0	1 (v_4)	1 (v_4)	1 (v_2/v_4)	1 (v_2/v_4)
4	0	0	0	0	0	0	0	1 (v_1)	1 (v_1)	1 (v_2)	1 (v_1/v_2)	1 (v_3)	0	1 (v_2/v_3)	1 ($v_1/v_2/v_3$)

Contd...



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	(\emptyset)	0	(v_2)	0	0	0	(v_2)	0	(v_4)	0	(v_2/v_4)	0	(v_4)	0	(v_2/v_4)
2	0	(v_1)	(v_1)	0	0	(v_3)	0	0	0	(v_4)	(v_1/v_4)	0	0	(v_3/v_4)	($v_1/v_3/v_4$)
3	0	0	0	(\emptyset)	0	(v_2)	(v_2)	0	0	0	0	(v_4)	(v_4)	(v_2/v_4)	(v_2/v_4)
4	0	0	0	0	0	0	0	(v_1)	(v_1)	(v_2)	(v_1/v_2)	(v_3)	0	(v_2/v_3)	($v_1/v_2/v_3$)



Cycle is $v_1 \rightarrow v_4 \rightarrow v_3 \rightarrow v_2 \rightarrow v_1$

Cycle is $v_1 \rightarrow v_2 \rightarrow v_3 \rightarrow v_4 \rightarrow v_1$

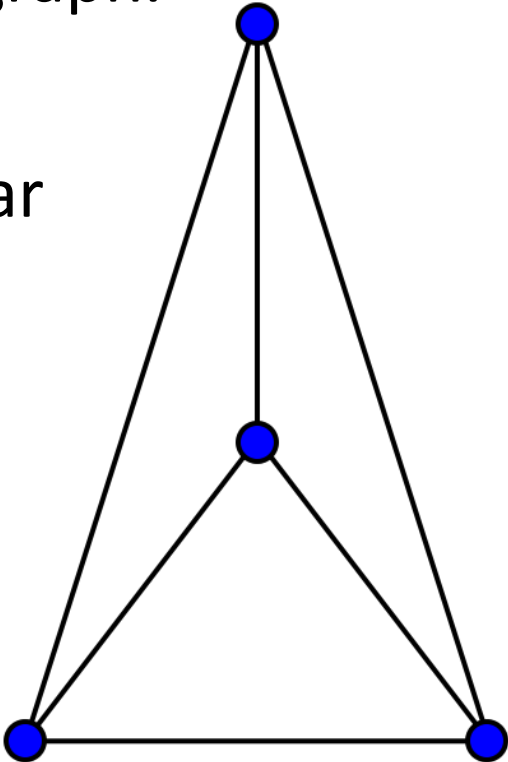
Applications

- Painting road lines,
- Plowing roads after a snowstorm,
- Checking meters along roads,
- Garbage pickup routes, etc.

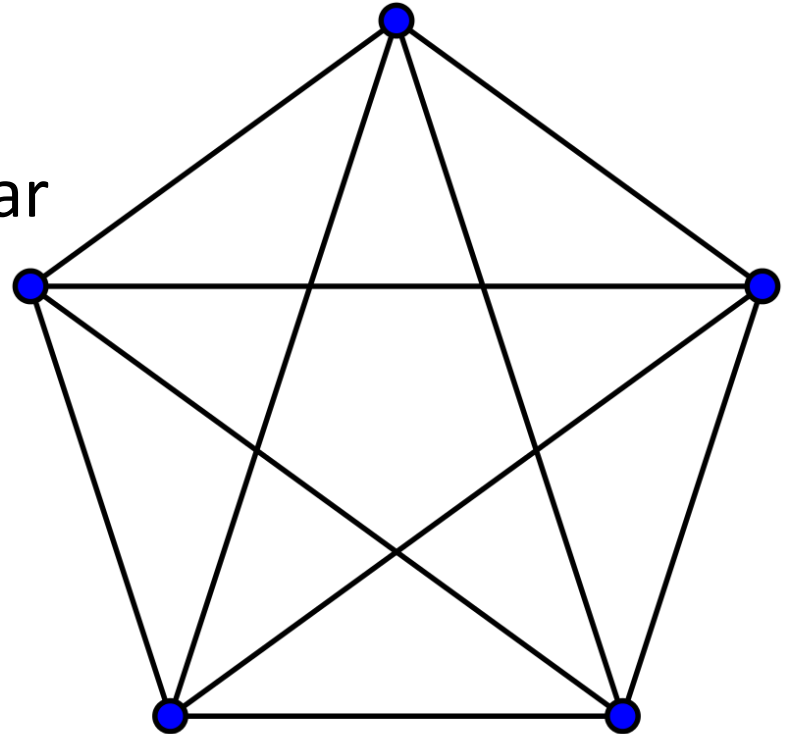
Planar Graphs

- If a graph G can be drawn on a plane (or a sphere) so that the edges only intersect at vertices, then it is planar.
- Such a drawing of a planar graph is a planar embedding of the graph.

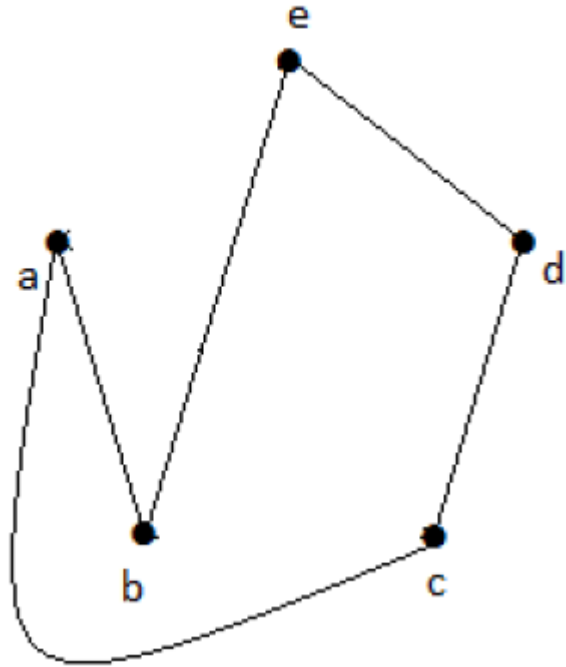
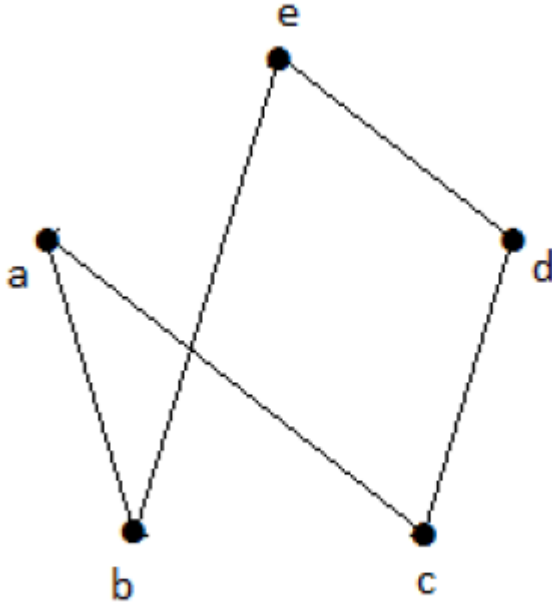
- Planar



- Nonplanar



Contd...



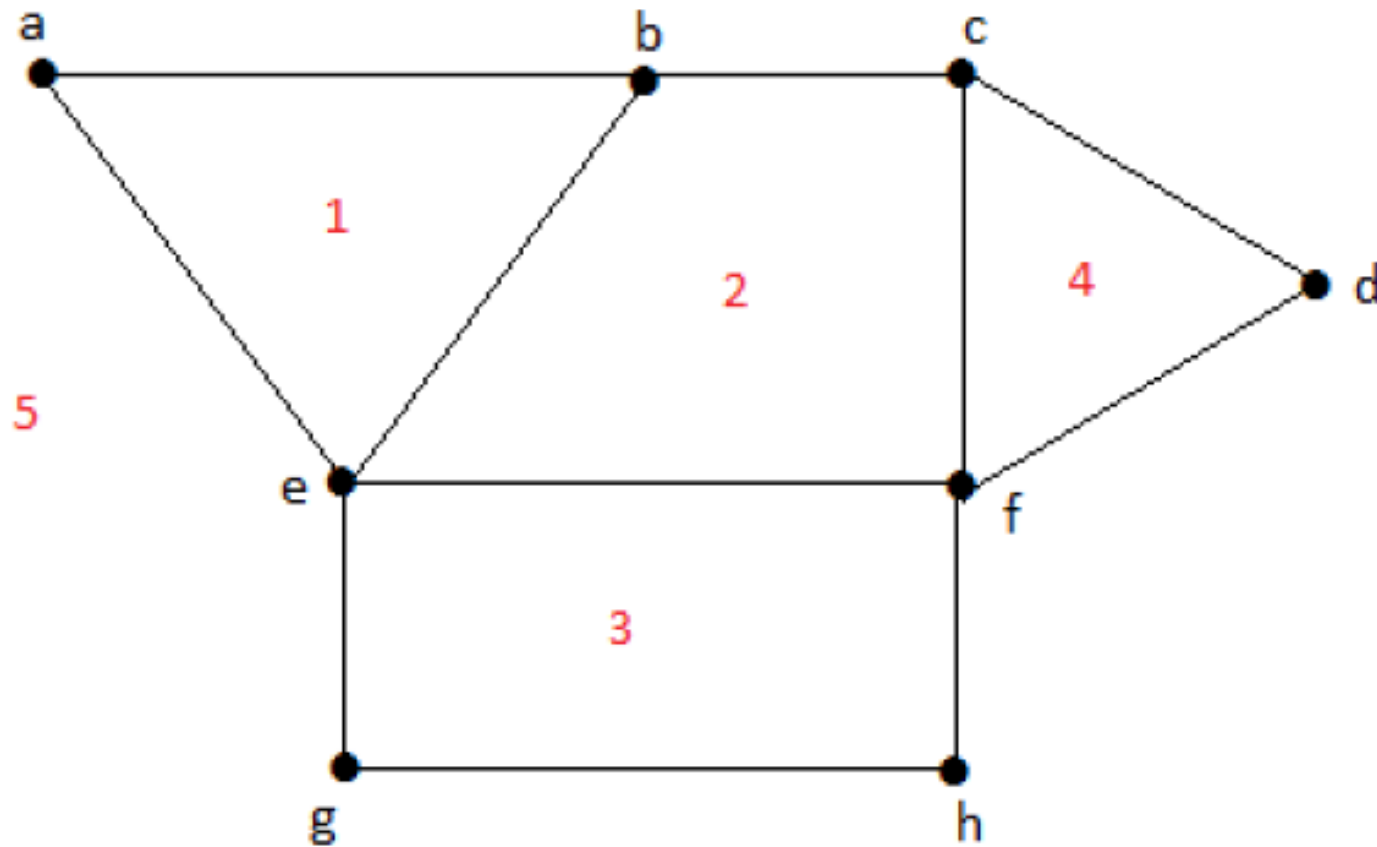
- A non-planar graph converted in to a planar graph.

Euler's formula

- For a finite, connected, planar graph, if
 - v is the number of vertices,
 - e is the number of edges, and
 - f is the number of faces (regions bounded by edges, including the outer, infinitely large region).
 - Then,

$$v - e + f = 2$$

Example

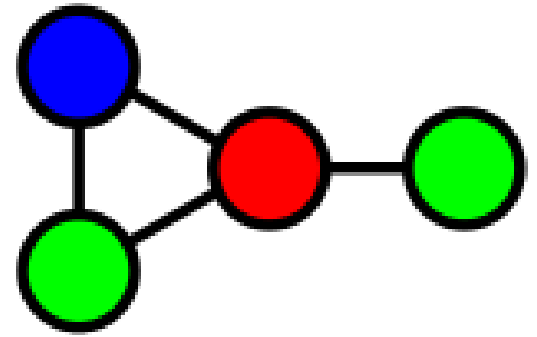


$$8 - 11 + 5 = 2$$

Graph Coloring

- Special case of labeling graph elements subject to certain constraints.
 - Traditionally, "colors" are used as labels.
- Several forms
 - **Vertex coloring.** Color vertices of a graph such that no two adjacent vertices share the same color.
 - **Edge coloring.** Color edges such that no two adjacent edges share the same color.
 - **Face coloring of a planar graph.** Assign color to each face or region so that no two faces that share a boundary have the same color.

Contd...



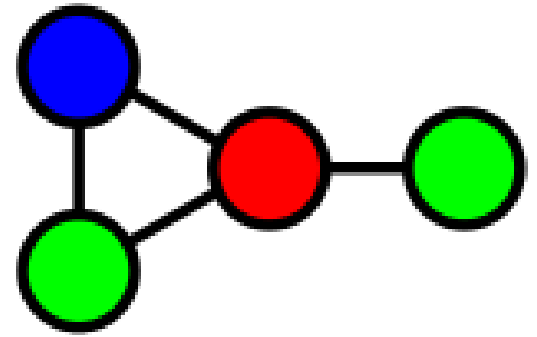
- The simplest form is vertex coloring.
- Formally,

If C be the set of colors, then find a function

$$f : V \rightarrow C$$

- such that if there is an edge (vw) , then $f(v) \neq f(w)$,
and
- C is of minimum cardinality.

Definitions



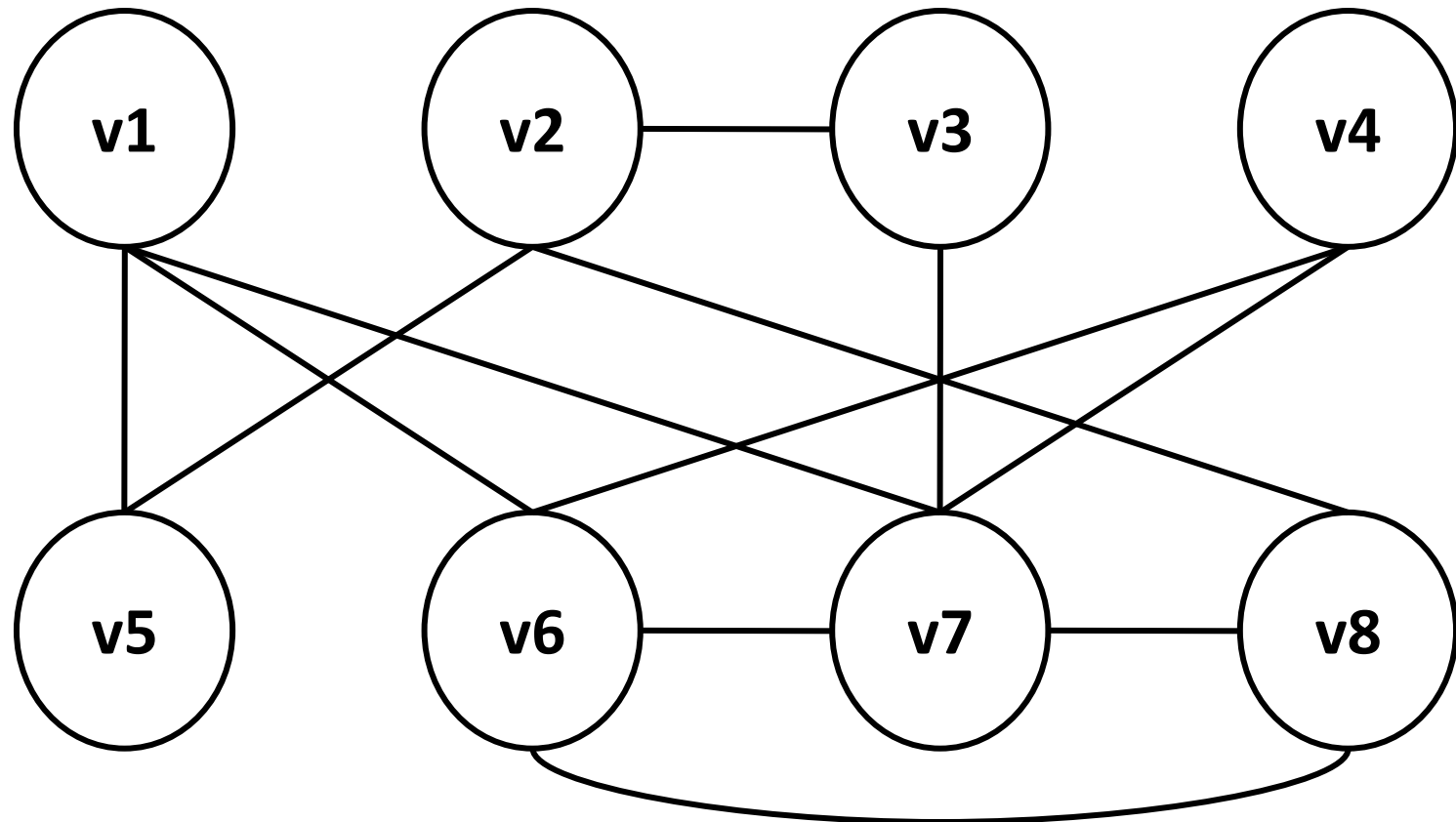
- **k -coloring.** Coloring using at most k colors.
- **Chromatic number of a graph ($\chi(G)$).** The smallest number of colors needed to color a graph G .
- **k -colorable.** A graph that can be assigned a k -coloring is k -colorable.
- Note: k -coloring partitions the vertex set into k independent sets.

Sequential Coloring – Greedy Approach

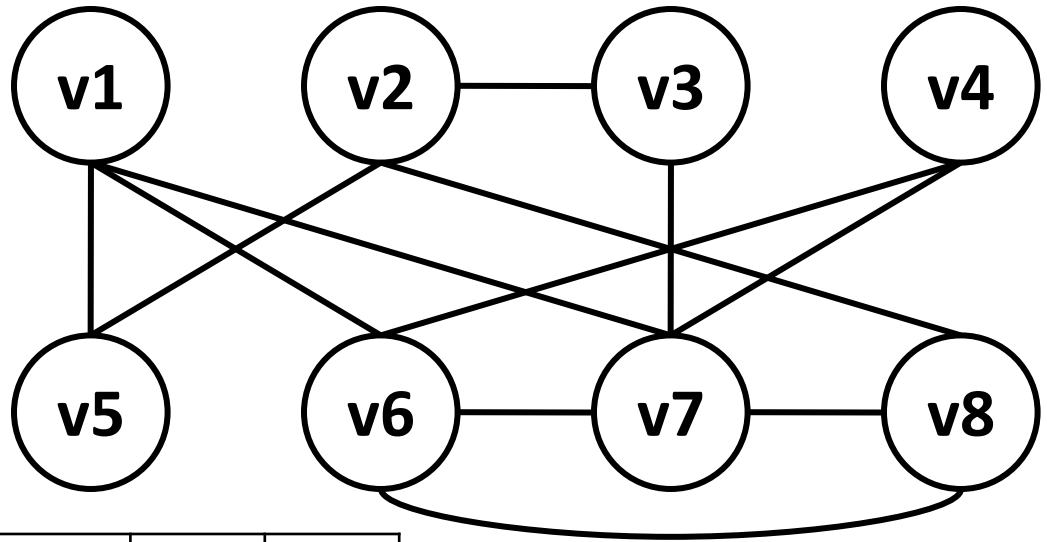
- sequentialColoringAlgorithm(graph = (V,E))
 1. Put vertices in a certain order $\{v_1, v_2, \dots, v_n\}$.
 2. Put colors in a certain order $\{c_1, c_2, \dots, c_k\}$.
 3. for $i = 1$ to n
 4. j = the smallest index of color that does not appear in any neighbor of v_i .
 5. color(v_i) = j .

If every node in G has degree $\leq d$, then the algorithm uses at most $d + 1$ colors for G .

Example



Example



Order of vertices

v1	v2	v3	v4	v5	v6	v7	v8
c1	c1	c2	c1	c2	c2	c3	c4

$$\begin{aligned} \text{Number of colors required} &\leq \text{Maximum degree} + 1 \\ &\leq \max\{\deg(v1), \deg(v2), \deg(v3), \deg(v4), \deg(v5), \\ &\quad \deg(v6), \deg(v7), \deg(v8)\} + 1 \\ &\leq \max\{3, 3, 2, 2, 2, 4, 5, 3\} + 1 \\ &\leq 5 + 1 \\ &\leq 6. \end{aligned}$$

Welsh–Powell algorithm

- If vertices are ordered according to their degrees (decreasing), then

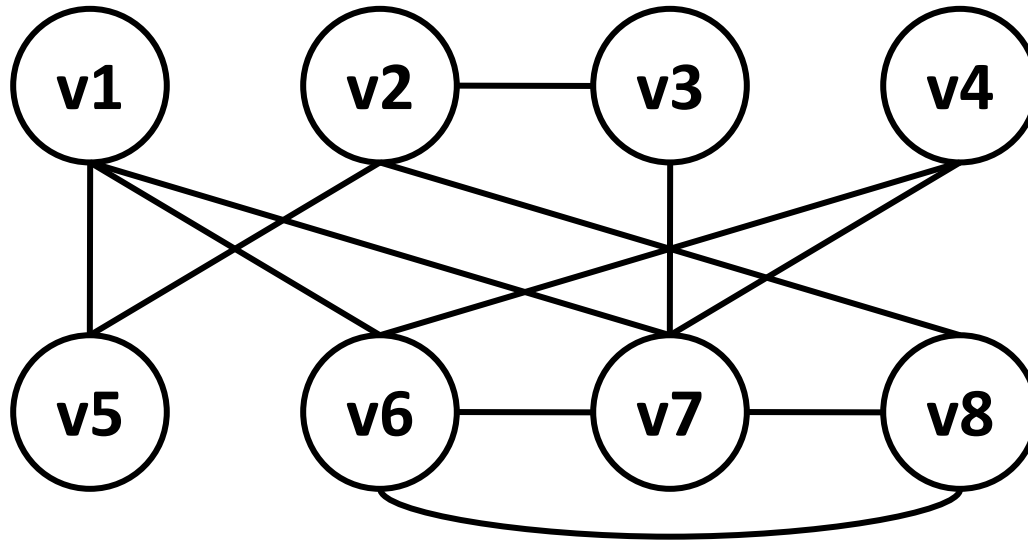
The number of colors needed to color the graph is at most

$$\max_i \min(i, \deg(v_i) + 1)$$

Number of colors for graph at slide 42.

- Number of colors required $\leq \max_i \min(i, \deg(v_i) + 1)$
- If vertices are arranged as $\{v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8\}$
 $\leq \max\{\min(1, 4), \min(2, 4), \min(3, 3), \min(4, 3),$
 $\min(5, 3), \min(6, 5), \min(7, 6), \min(8, 4)\}$
 $\leq \max\{1, 2, 3, 3, 3, 5, 6, 4\}$
 $\leq 6.$
- Arrangement as per degrees $\{v_7, v_6, v_1, v_2, v_8, v_3, v_4, v_5\}$
 $\leq \max\{\min(1, 6), \min(2, 5), \min(3, 4), \min(4, 4),$
 $\min(5, 4), \min(6, 3), \min(7, 3), \min(8, 3)\}$
 $\leq \max\{1, 2, 3, 4, 4, 3, 3, 3\}$
 $\leq 4.$

Example



$v7$	$v6$	$v1$	$v2$	$v8$	$v3$	$v4$	$v5$
$c1$	$c2$	$c3$	$c1$	$c3$	$c2$	$c3$	$c2$

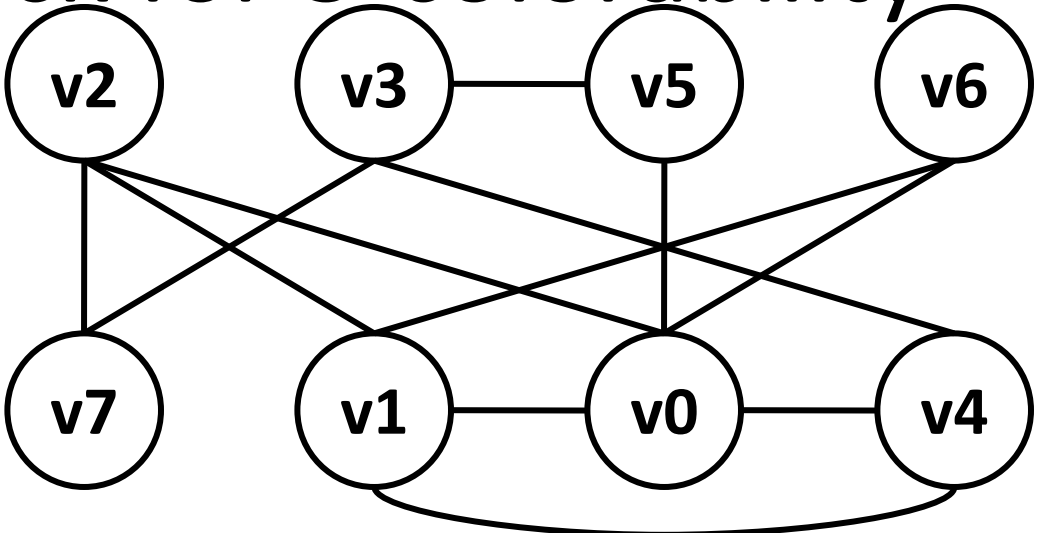
m-Coloring Problem

- Given an undirected graph and a number m , determine if the graph can be colored with at most m colors such that no two adjacent vertices have the same color.
- Input:
 - An adjacency matrix, $graph[V][V]$ where V is the number of vertices in the graph.
 - An integer m which is maximum number of colors that can be used.
- Output:
 - An array $color[V]$ containing colors assigned to all the vertices in the range 1 to m . False will be returned, if the graph cannot be colored with m colors.

Backtracking

1. If all colors are assigned,
2. Print vertex assigned colors.
3. Else
4. Trying all possible colors, assign a color to the vertex.
5. If color assignment is possible, recursively assign colors to next vertices.
6. If color assignment is not possible, de-assign color, return False.

Example – Check for 3-colorability

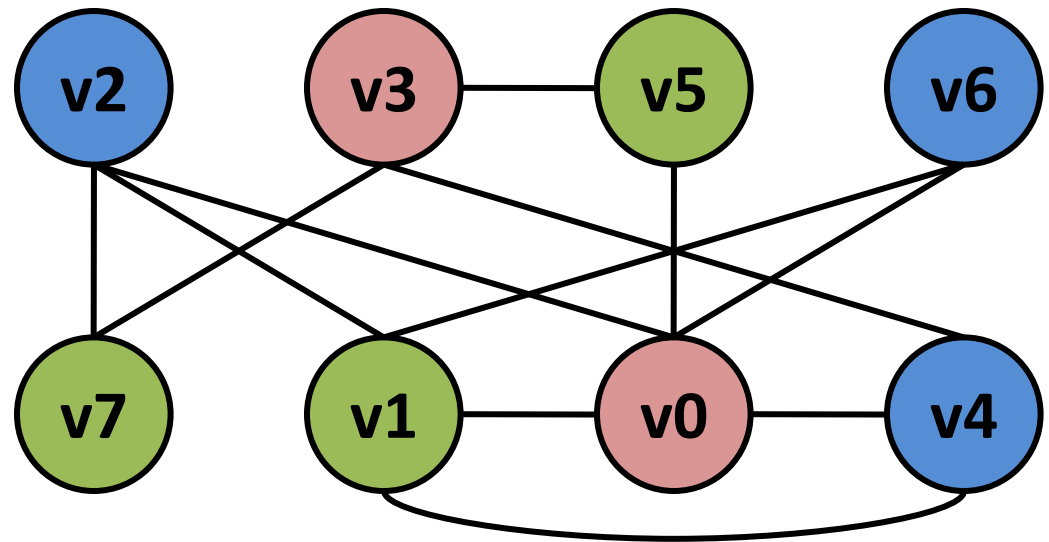


7	c2	€1	c2	
6	c3	€1	€2	c3
5	c2	€1	c2	
4	c3	€1	€2	c3
3	c1			
2	c3	€1	€2	c3
1	c2	€1	c2	
0	c1			

	0	1	2	3	4	5	6	7
0	0	1	1	0	1	1	1	0
1	1	0	1	0	1	0	1	0
2	1	1	0	0	0	0	0	1
3	0	0	0	0	1	1	0	1
4	1	1	0	1	0	0	0	0
5	1	0	0	1	0	0	0	0
6	1	1	0	0	0	0	0	0
7	0	0	1	1	0	0	0	0

Applications

- Scheduling.
- Frequency Assignment.
- Register Allocation.
- Bipartite Graphs.
- Map Coloring, etc.



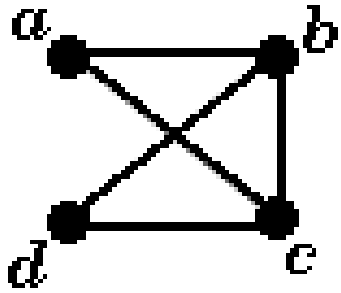
Isomorphism

- Two graphs are *isomorphic* when the vertices of one can be re-labeled to match the vertices of the other in a way that preserves adjacency.
- Formally,
Graphs G_1 and G_2 are *isomorphic* if there exists a one-to-one function, called an *isomorphism*,

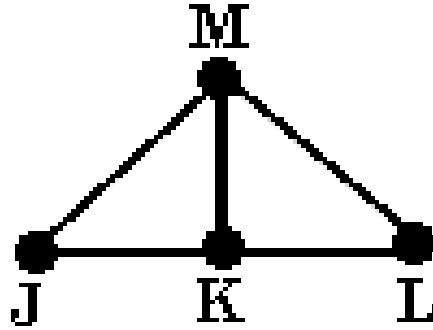
$$f: V(G_1) \rightarrow V(G_2)$$

such that uv is an element of $E(G_1)$ if and only if $f(u)f(v)$ is an element of $E(G_2)$.

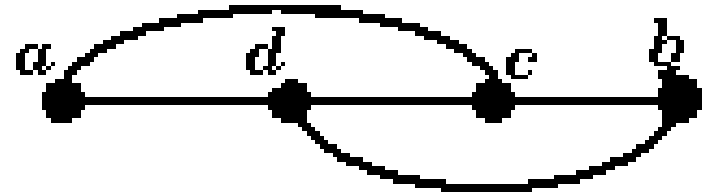
Example



G_1



G_2



G_3

- G_1 and G_2

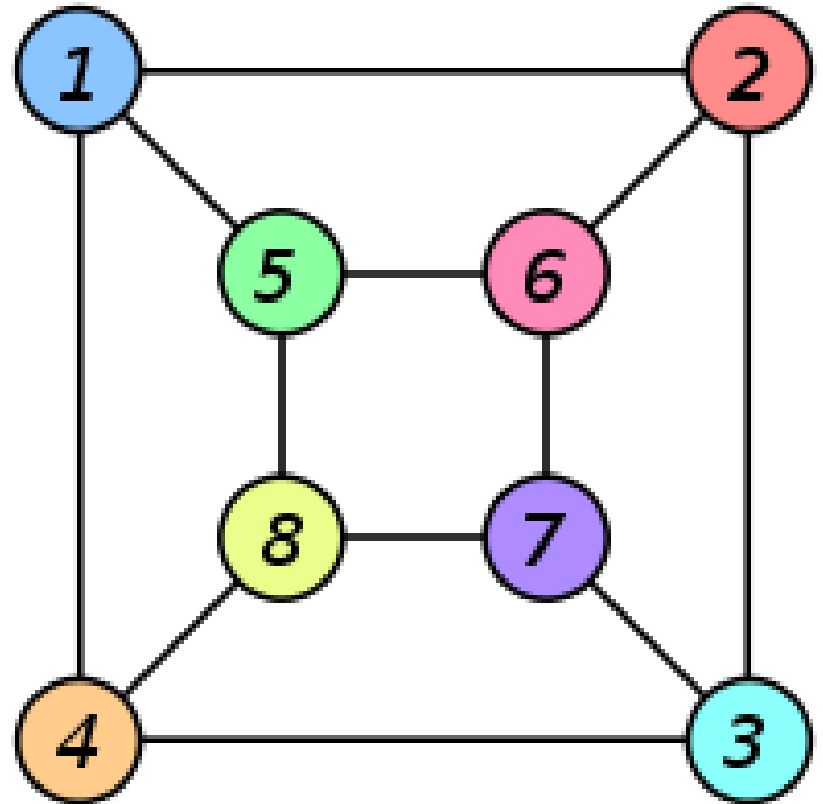
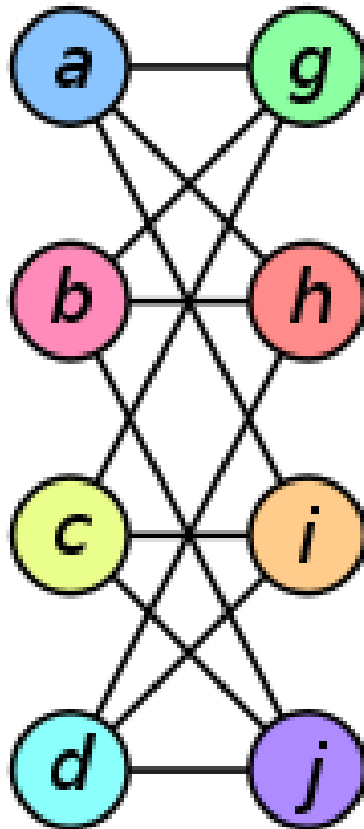
$$f(a) = J, f(b) = K, f(c) = M, \text{ and } f(d) = L.$$

- G_1 and G_3

$$f(a) = a, f(b) = d, f(c) = c, \text{ and } f(d) = b.$$

Example

- $f(a) = 1$
- $f(b) = 6$
- $f(c) = 8$
- $f(d) = 3$
- $f(g) = 5$
- $f(h) = 2$
- $f(i) = 4$
- $f(j) = 7$



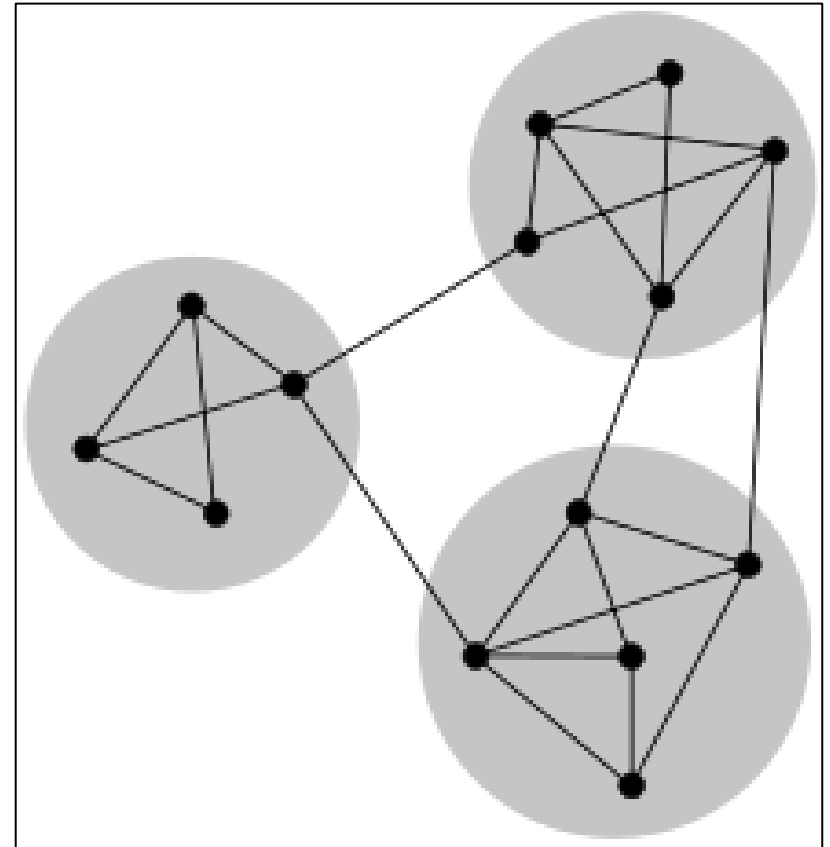
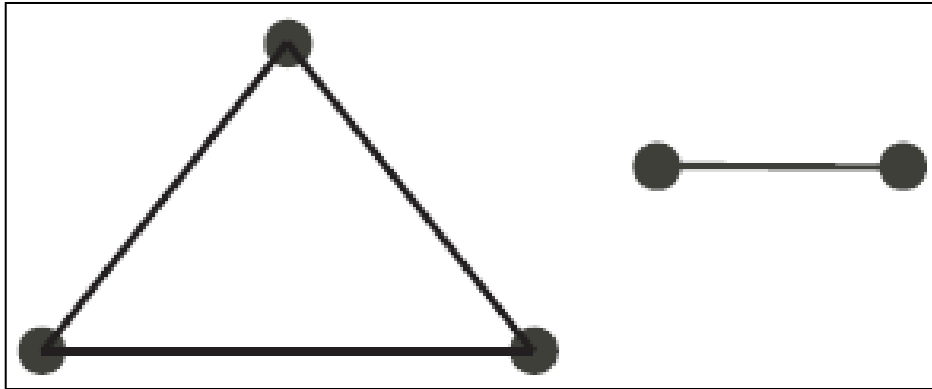
Graph Isomorphism Problem

- Determining whether two finite graphs are isomorphic or not?
- It is neither an NP-complete problem nor a P-problem, although this has not been proved (Skiena 1990).
- There is a famous complexity class called **graph isomorphism complete** which is thought to be entirely disjoint from both NP-complete and from P.
- Applications:
 - Cheminformatics,
 - Mathematical chemistry (identification of chemical compounds), and
 - Electronic design automation (verification of equivalence of various representations of the design of an electronic circuit).

Connected and Disconnected Graph

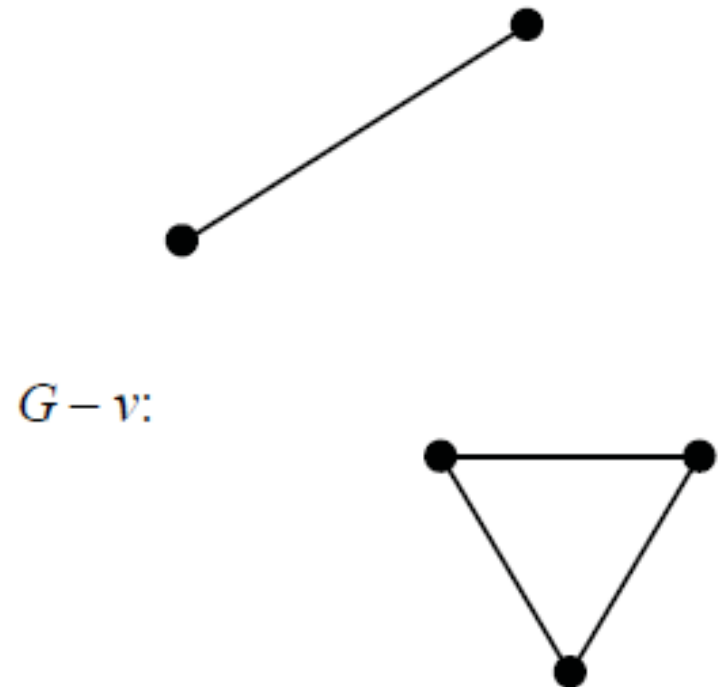
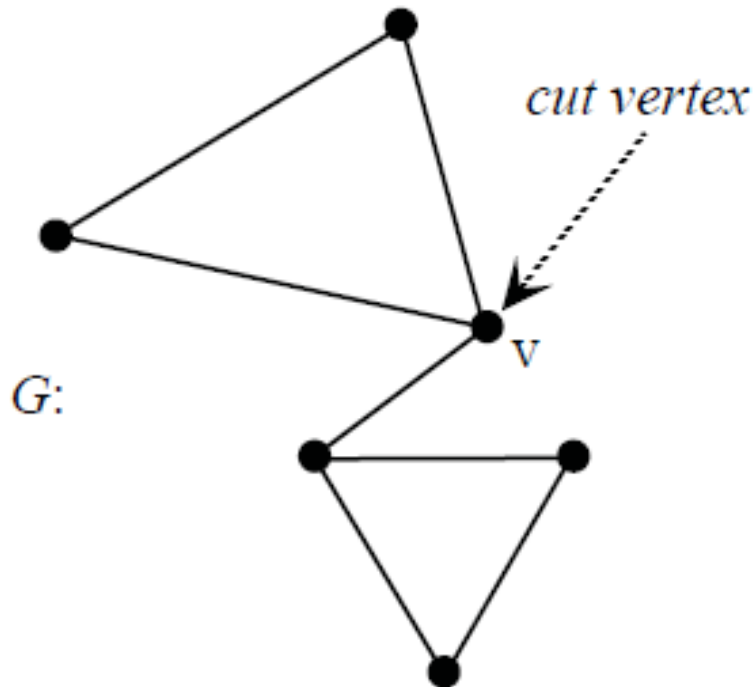
- A graph is connected if all the vertices are connected to each other, i.e.
 - A path exists between every pair of vertices.
 - No unreachable vertices.
- A graph is disconnected if it is not connected.
- A graph with just one vertex is connected.
- An edgeless graph with two or more vertices is disconnected.

Example



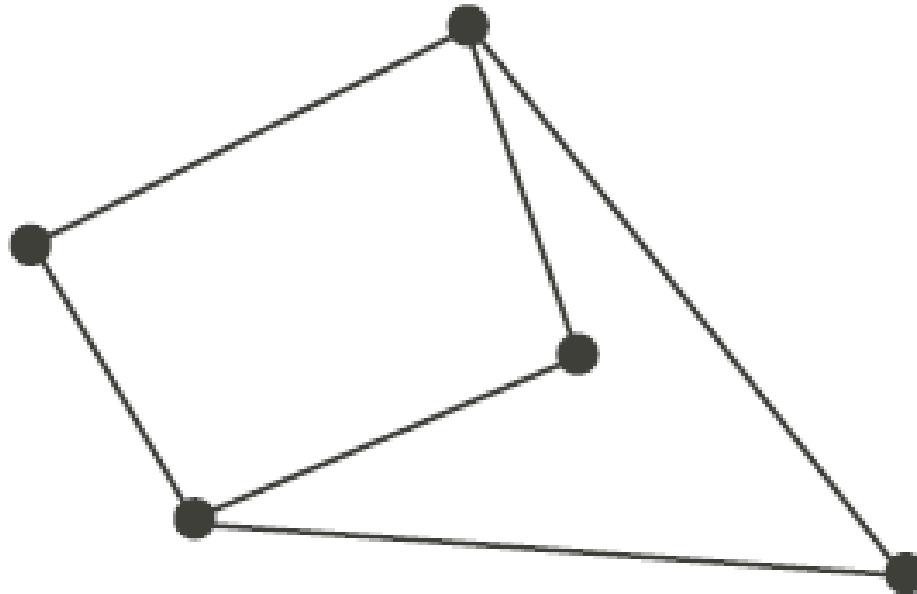
Cut Vertex

- A vertex v of a graph G is a cut vertex or an articulation vertex of G if the graph $G - v$ consists of a greater number of components than G .



Contd...

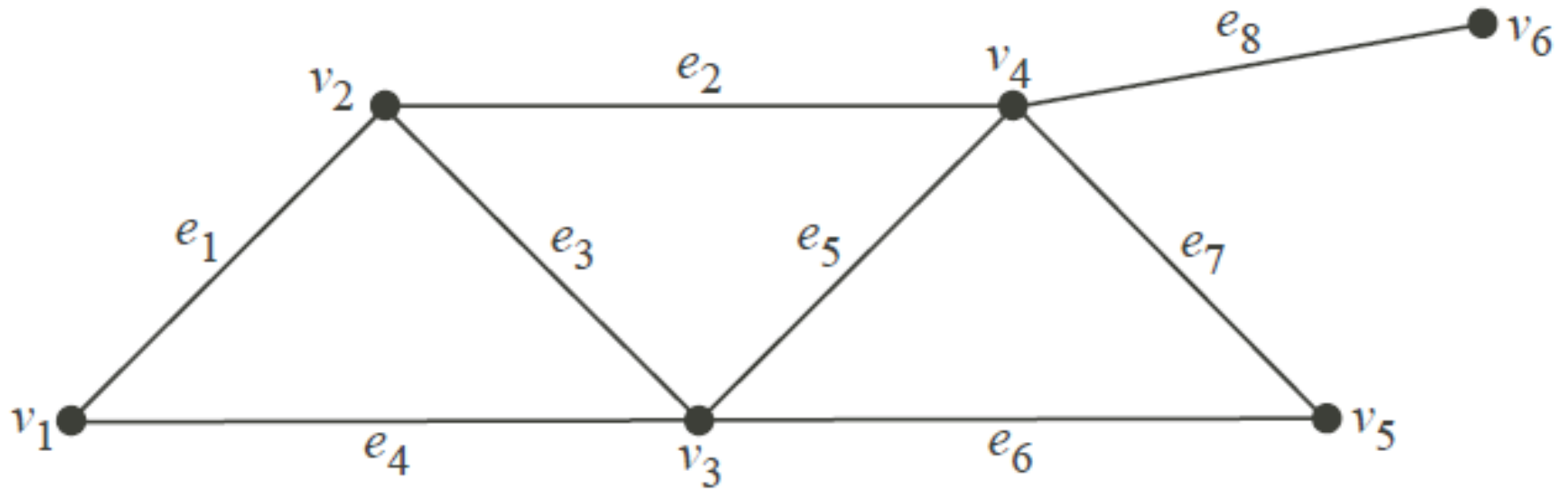
- A graph is separable if it is not connected or if there exists at least one cut vertex in the graph.



Cut-set

- A cut set of the connected graph $G = (V, E)$ is an edge set $F \subseteq E$ such that
 1. $G - F$ (remove the edges of F one by one) is not connected, and
 2. $G - H$ is connected whenever $H \subset F$.

Example



- $\{e_1, e_4\}$, $\{e_6, e_7\}$, $\{e_1, e_2, e_3\}$, $\{e_8\}$, $\{e_3, e_4, e_5, e_6\}$, $\{e_2, e_5, e_7\}$, $\{e_2, e_5, e_6\}$ and $\{e_2, e_3, e_4\}$.

Cut

- A cut is a partition of the vertices of a graph into two disjoint subsets.

- Formally,

In a graph $G = (V, E)$, a pair of subsets V_1 and V_2 of V satisfying

$$V = V_1 \cup V_2, V_1 \cap V_2 = \emptyset, V_1 \neq \emptyset, V_2 \neq \emptyset$$

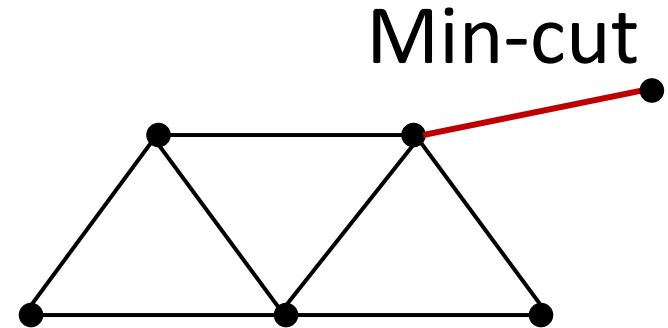
is called a cut (or a partition) of G . Denoted as (V_1, V_2) .

- It is also defined as an edge set.

$\text{cut}(V_1, V_2) = \{\text{those edges with one end vertex in } V_1 \text{ and the other end vertex in } V_2\}.$

Contd...

- In an unweighted undirected graph, the size or weight of a cut is the number of edges crossing the cut.



- In a weighted graph, the value or weight is defined by the sum of the weights of the edges crossing the cut.

