CSE204
Complexity of Algorithms
Graphs

CSSE, Xian Jiaotong-Liverpool University

Year 2019-2020

Connectivity information

Connectivity information can be defined by many kinds of relationships that exist between pairs of objects.

For example, connectivity information is present in city maps, where the objects are roads, and also in the routing tables for the internet, where the objects are computers.

Connectivity information is also present in the parent-child relationships defined by a binary tree, where the objects are tree nodes.

Graphs are one way in which connectivity information can be stored, expressed, and utilized.

Graphs

A *graph* is a set of objects, called *vertices* (or *nodes*), together with a collection of pair-wise connections, called *edges*, between them.

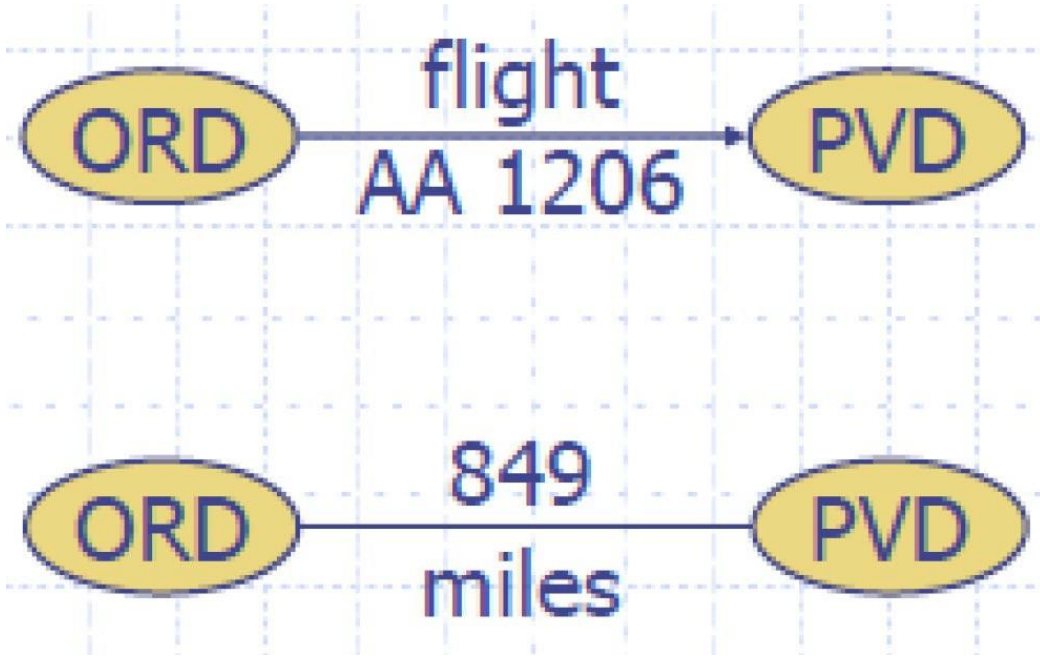
Graphs have applications in a number of different domains, including *mapping, transportation*, *electrical engineering*, and *computer networking*.

More formally, a graph $G = (V, E)$, is a set, V, of *vertices* and a collection, E, of pairs of vertices from V, called *edges*.

Graphs

Edge Types

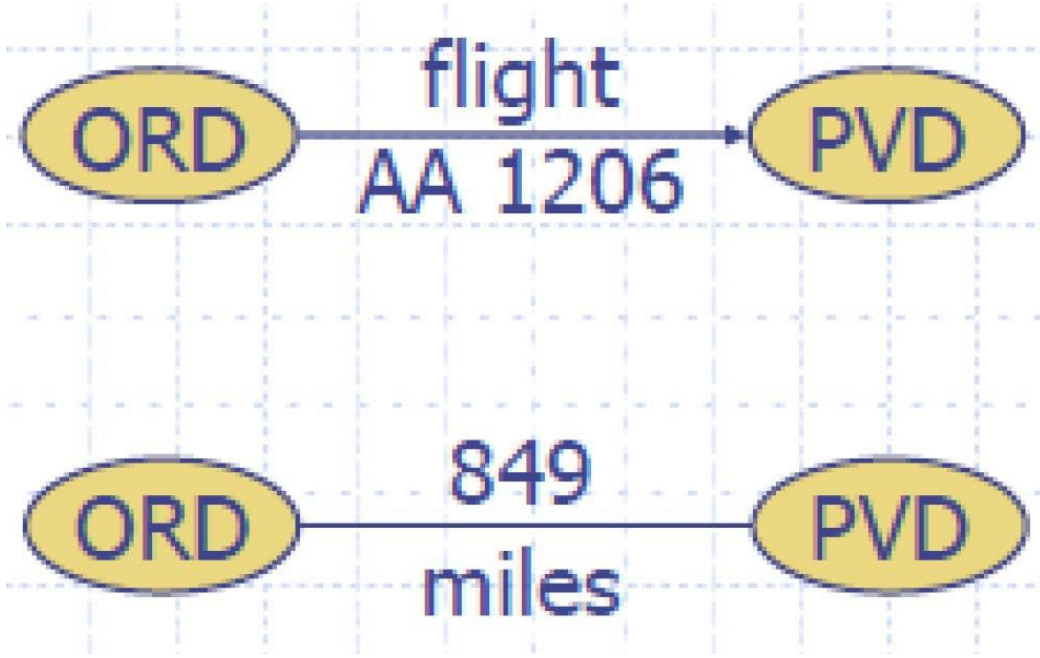
- Directed edge
  - ordered pair of vertices (u,v)
  - first vertex u is the origin
  - second vertex v is the destination
  - e.g., a flight
- Undirected edge
  - unordered pair of vertices v
  - e.g., a flight route



Graphs

Edge Types vs. Graph Types

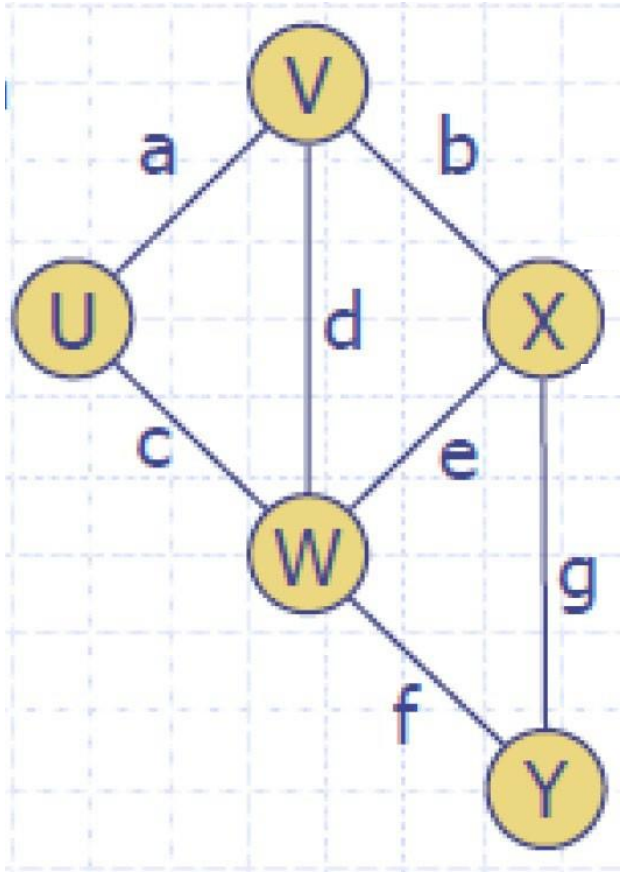
- Directed edge
  - ordered pair of vertices (u,v)
  - first vertex u is the origin
  - second vertex v is the destination
  - e.g., a flight
- Undirected edge
  - unordered pair of vertices v
  - e.g., a flight route
- Directed graph
  - all the edges are directed
  - e.g., flight network
- Undirected graph
  - all the edges are undirected
  - e.g., route network



Graphs

Terminology

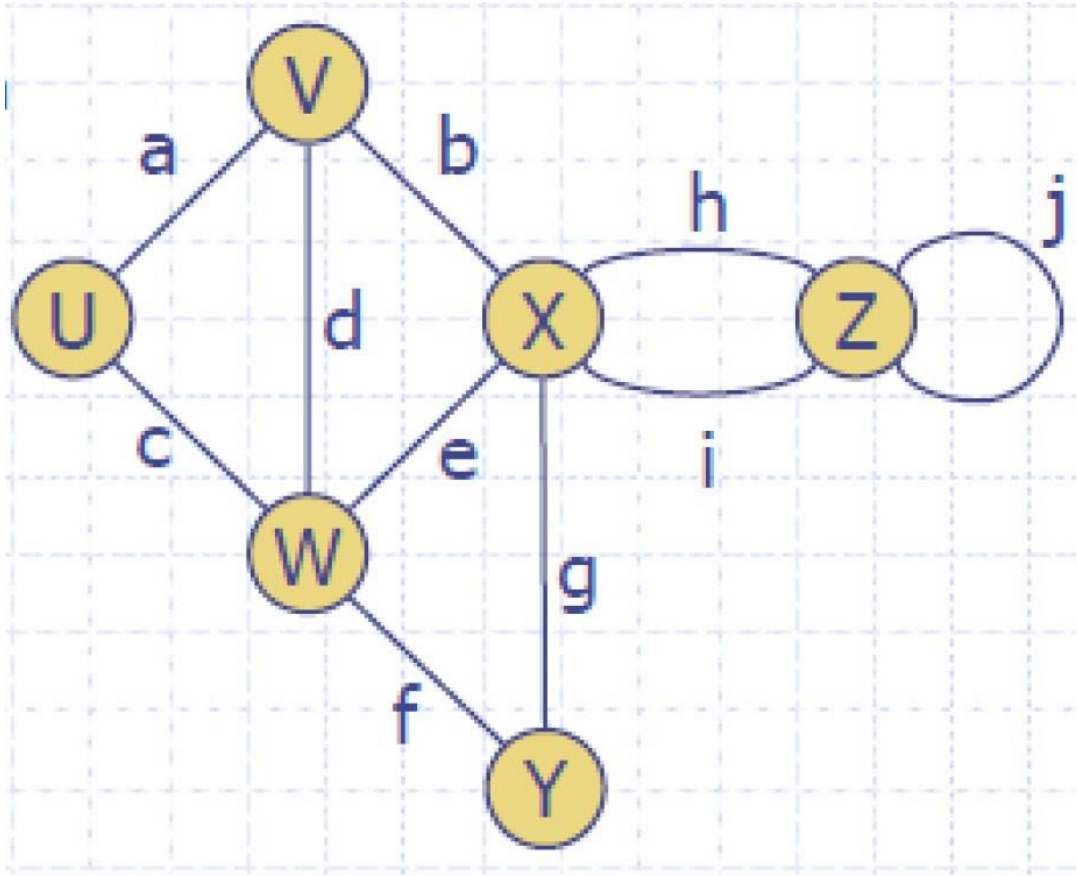
- End vertices (or endpoints) of an edge
  - U and V are the endpoints of a
- Edges incident on a vertex
  - a, d, and b are incident on V
- Adjacent vertices
  - U and V are adjacent
- Degree of a vertex
  - X has degree 3



Graphs

Terminology

- Parallel edges
  - h and i are parallel edges
- Self-loop
  - j is a self-loop

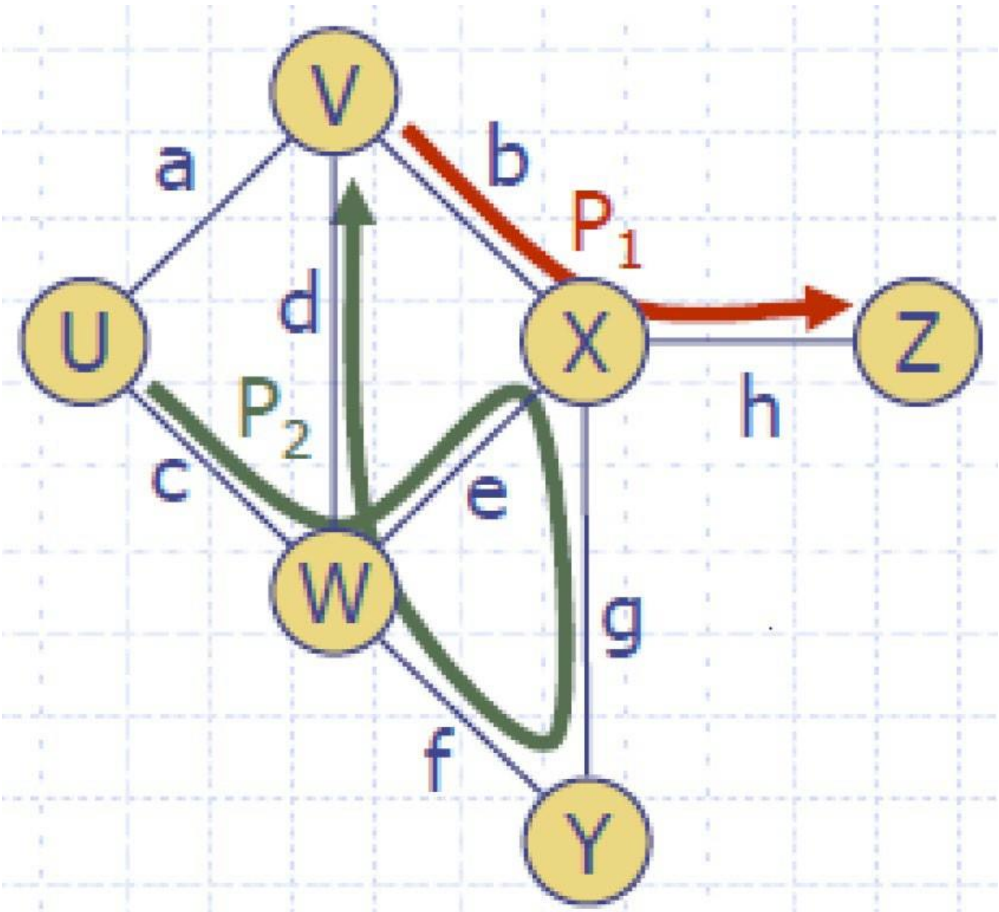


Graphs

- Path
  - sequence of alternating vertices and edges
  - begins with a vertex
  - ends with a vertex
  - each edge is preceded and followed by its endpoints
- Simple path
  - path such that all its vertices and edges are distinct

Examples

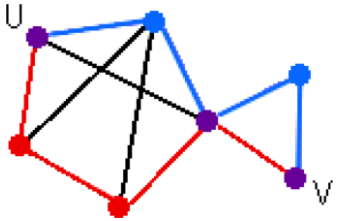
- P_1 is a simple path
- P_2 is a path that is not simple



More graph terminology

A *walk* in a graph is a sequence of alternating vertices and edges, starting at a vertex and ending at a vertex.

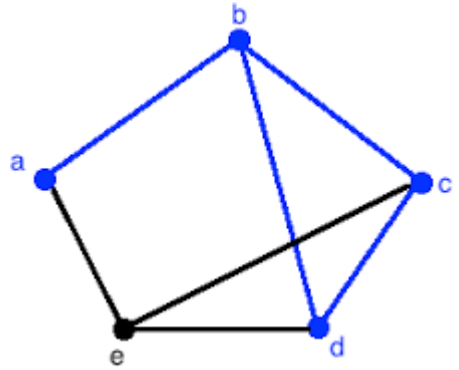
A *trail*: a walk with no repeated edge.



Graph 1:Two paths from U to V

Walk: a-b-c-d-b-c-d

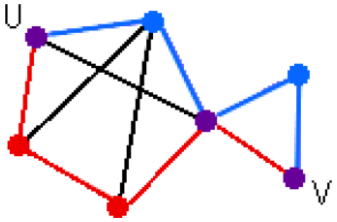
Trail: a-b-c-d-b



Graph 2: Trail

More graph terminology

A *walk* in a graph is a sequence of alternating vertices and edges, starting at a vertex and ending at a vertex.

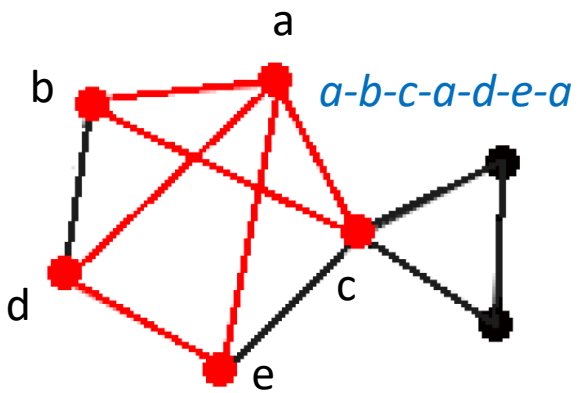


Two paths from U to V

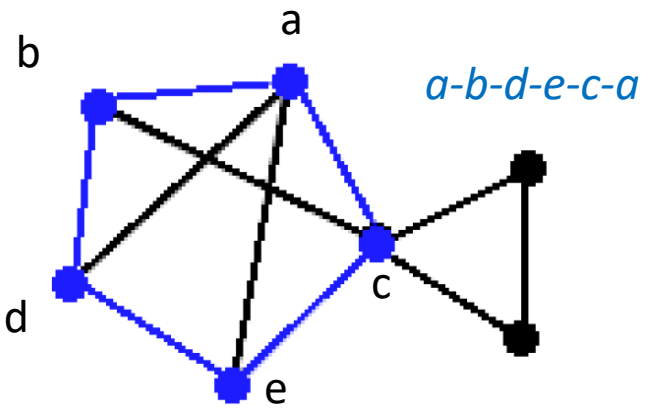
A *trail*: a walk with no repeated edge.

A *circuit* is a walk with the same start and end vertex.

A *cycle* is a circuit where each vertex in the circuit is *distinct* (except for first and last vertex).



Graph 3: circuit



Graph 4:cycle

More graph terminology

A *walk* in a graph is a sequence of alternating vertices and edges, starting at a vertex and ending at a vertex.

A *trail*: a walk with no repeated edge.

A *circuit* is a walk with the same start and end vertex.

A *cycle* is a circuit where each vertex in the circuit is *distinct* (except for first and last vertex).

A *directed walk* is a walk in which all edges are directed and are traversed along their direction. Directed paths, circuits, and cycles are defined similarly.

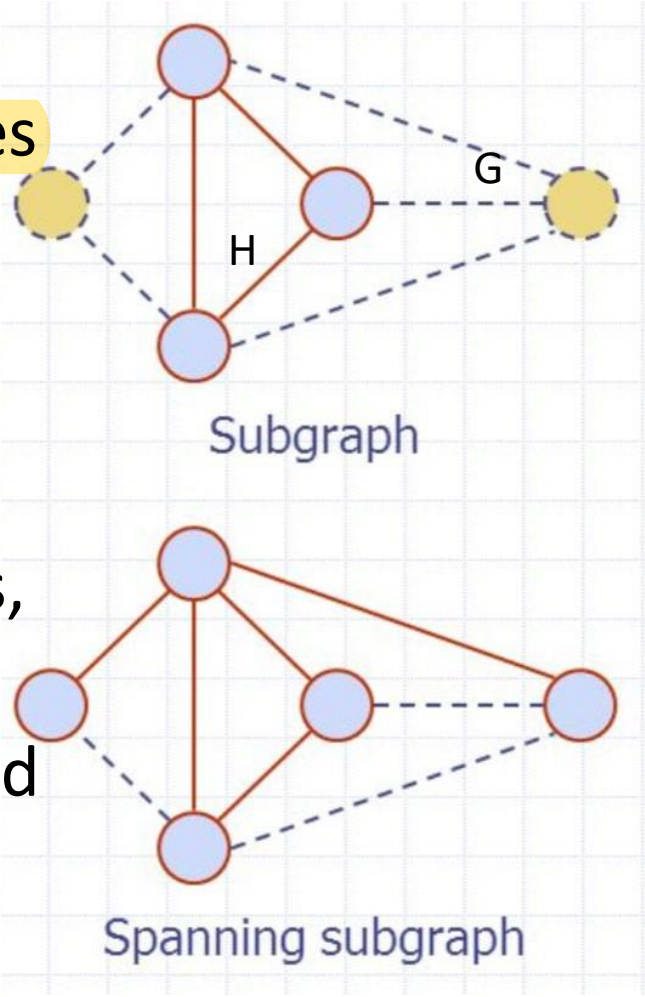
More terminology

A *subgraph* of a graph *G* is a graph *H* whose vertices and edges are *subsets* of the vertices and edges of *G*.

A *spanning subgraph* of *G* is a subgraph of *G* that contains all the vertices of *G*.

A graph is *connected* if, for any two distinct vertices, there is a path between them.

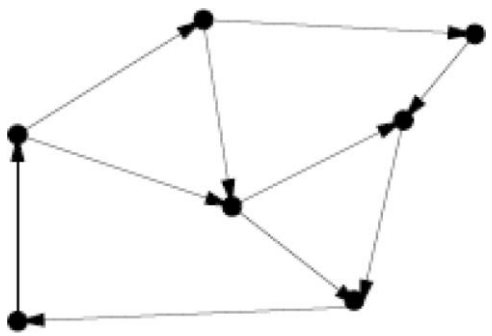
If a graph *G* is *not connected*, its maximal connected subgraphs are called the *connected components* of *G*.



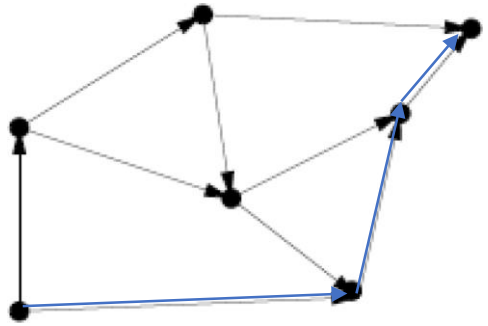
More graph terminology

An acyclic graph does not contain any cycles. Trees are connected acyclic undirected graphs.

Directed acyclic graphs are called DAGs. it's impossible to come back to the same node by traversing the edges. (scheduling problems, (x,y))



Cyclic



Acyclic

Graphs (cont.)

Properties

Property 1

$$\sum_v \deg(v) = 2m$$

Proof: each edge is counted twice

Property 2

In an undirected graph with no self-loops and no multiple edges

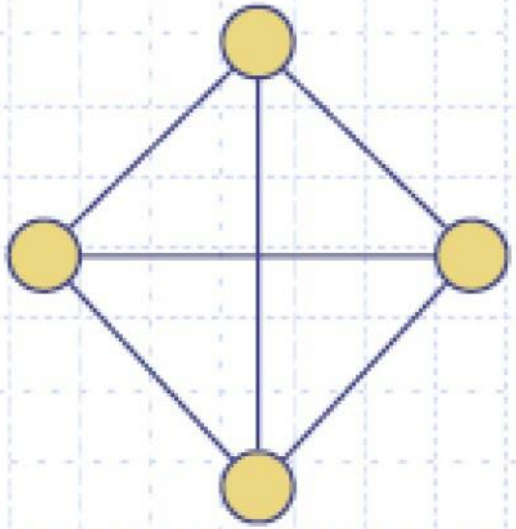
$m \leq n(n-1)/2$

Proof: each vertex has degree at most

$n-1$

Notation

n	number of vertices
m	number of edges
$\deg(v)$	degree of vertex v



Example

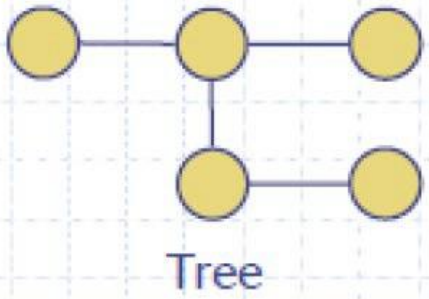
- $n = 4$
- $m = 6$
- $\deg(v) = 3$

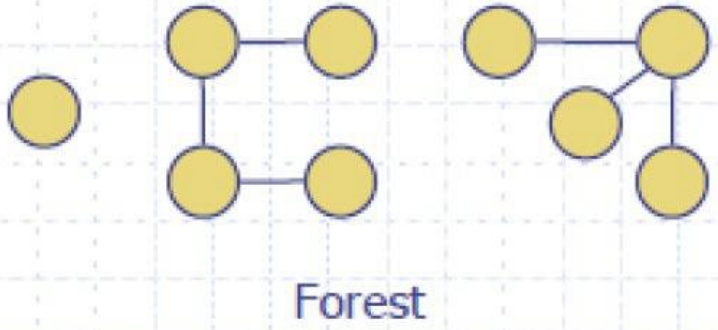
Graphs (cont.)

Trees and Forests

- A (free) tree is an undirected graph T such that
  - T is connected
  - T has no cycles

  This definition of tree is different from the one of a rooted tree
- A forest is an undirected graph without cycles
- The connected components of a forest are trees





Let G be an undirected graph with n vertices and m edges. We have the following observations:

If G is *connected*, then $m \geq n - 1$;

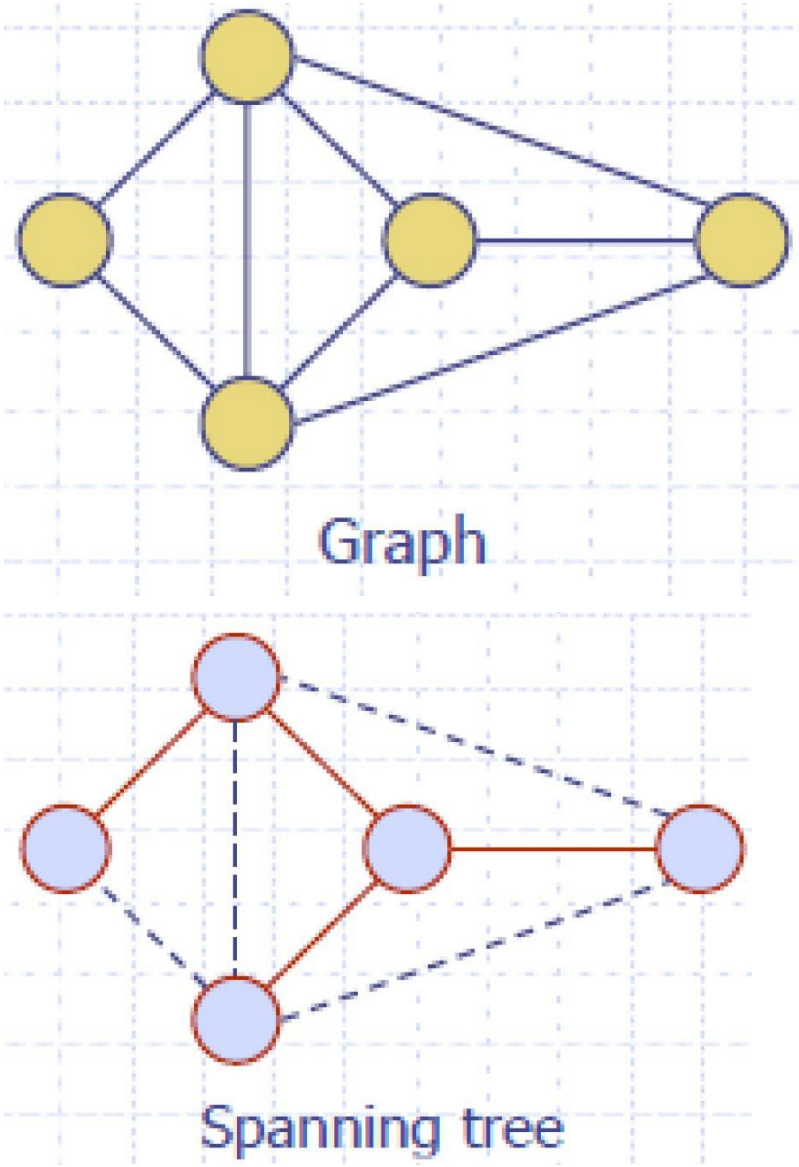
If G is a tree, then $m = n - 1$;

If G is a forest, then $m \leq n - 1$.

Graphs (cont.)

Spanning Trees and Forests

- A spanning tree of a connected, graph is a spanning subgraph that is a tree
- A spanning tree is not unique unless the graph is a tree
- Spanning trees have applications to the design of communication networks
- A spanning forest of a graph is a spanning subgraph that is a forest



Pathfinding algorithms

Depth First Search (DFS)

Breadth First Search (BFS)

Traversal: for exploring a graph

Depth First Search

Depth First Search (DFS) is a means of exploring a graph.

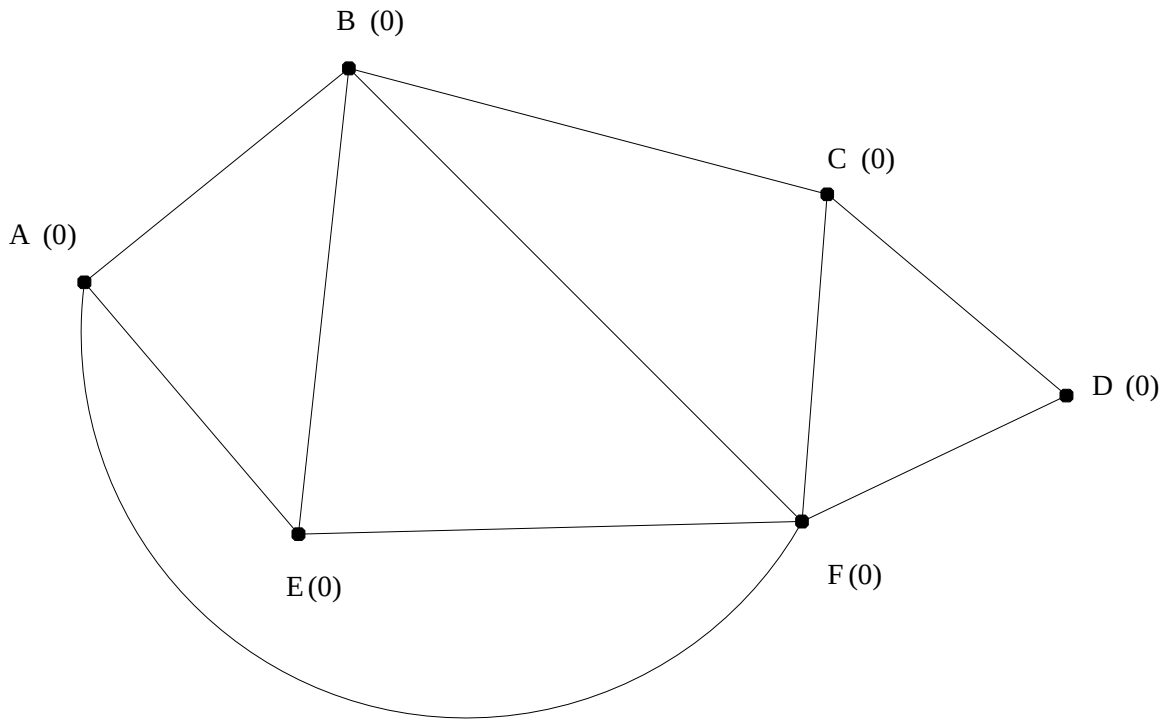
The basic idea is that we start at some vertex, and start moving away from this start vertex.

We travel “as far as possible” before we have to “back up” because we get to a point where we have hit vertices that have all been previously found.

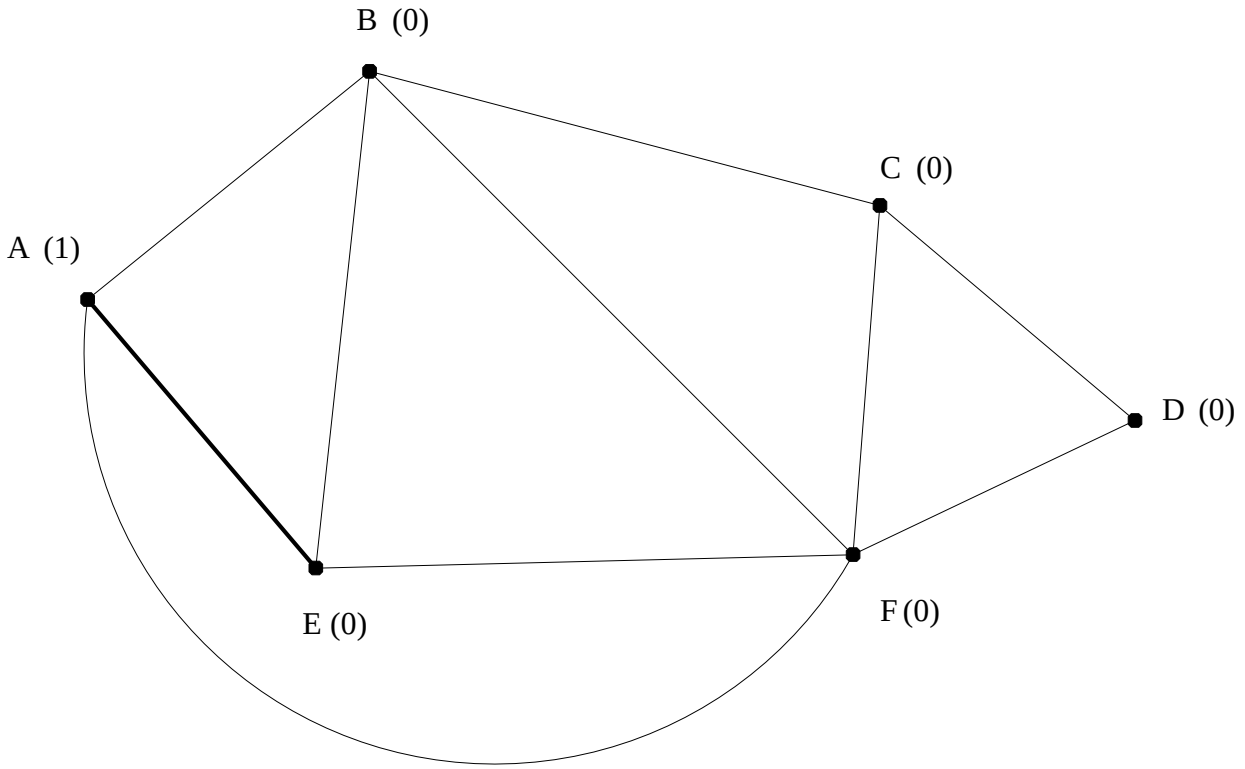
Depth First Search Algorithm (recursive)

```
DFS(G, v)
    1> Input: A graph G and a vertex v of G.
    1> Output: A labeling of the edges of G.
1   for all edges e in G.INCIDENTEDGES(v)
2       do
3           if UNEXPLORED(e)
4               then w ← G.OPPOSITE(v, w)   //Return the endpoint
5                   if UNEXPLORED(w)
6                       then label e as discovery edge
7                           DFS(G, w)
8                   else
9                           Label e as back edge
```

Depth First Search of a Graph

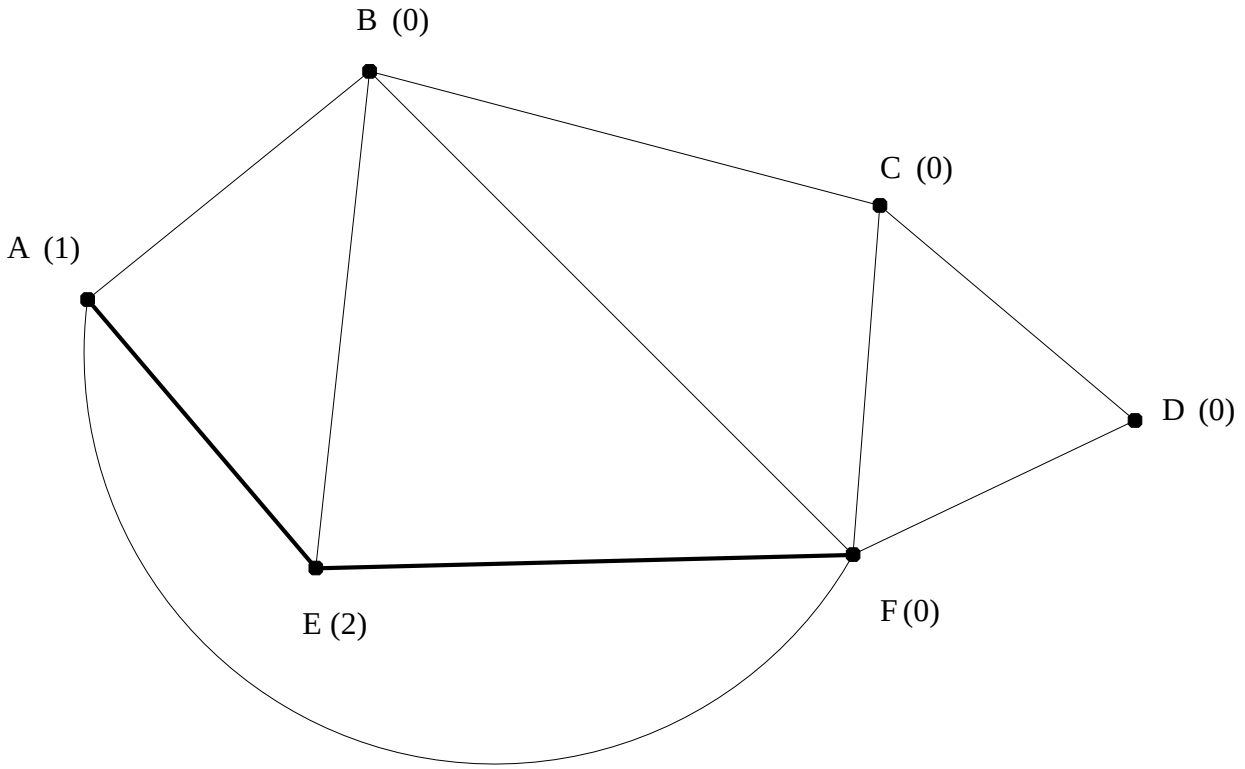


DepthFirstSearch(G,A)



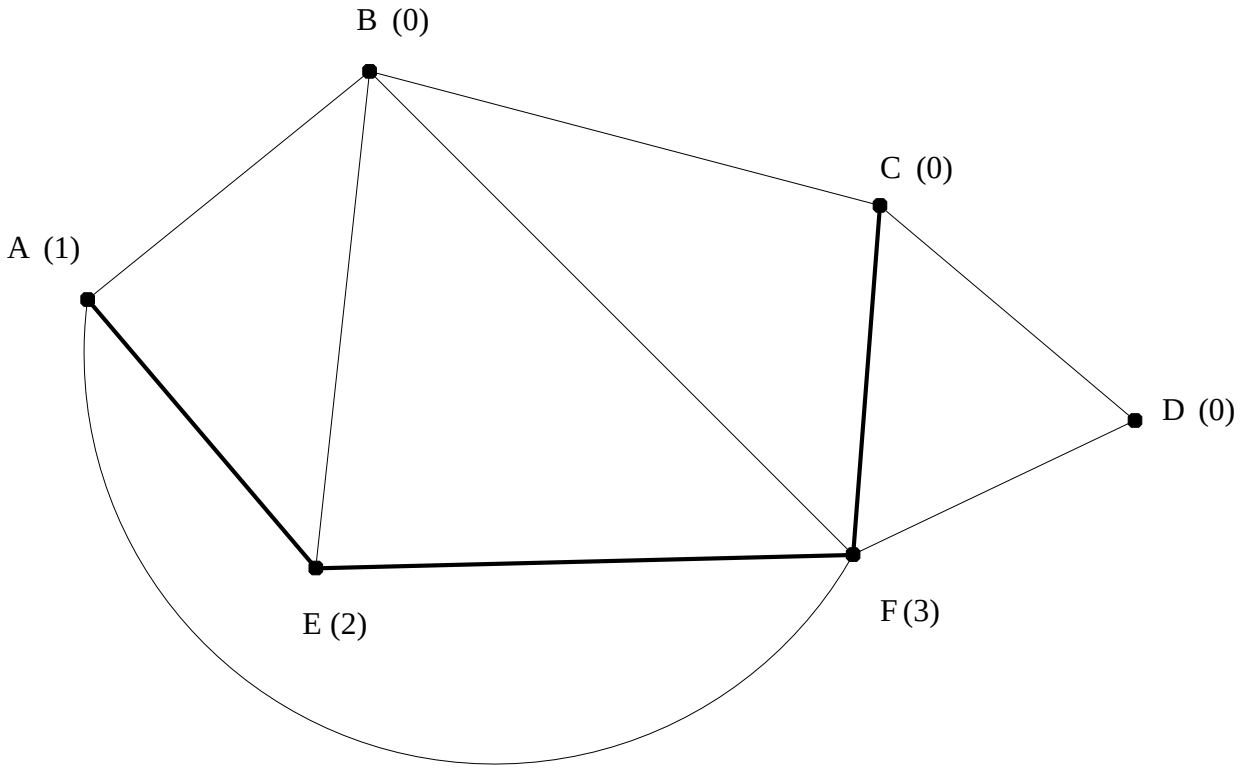
Here we add the edge (*A, E*) to our Depth First Search tree.

DepthFirstSearch(G,E)



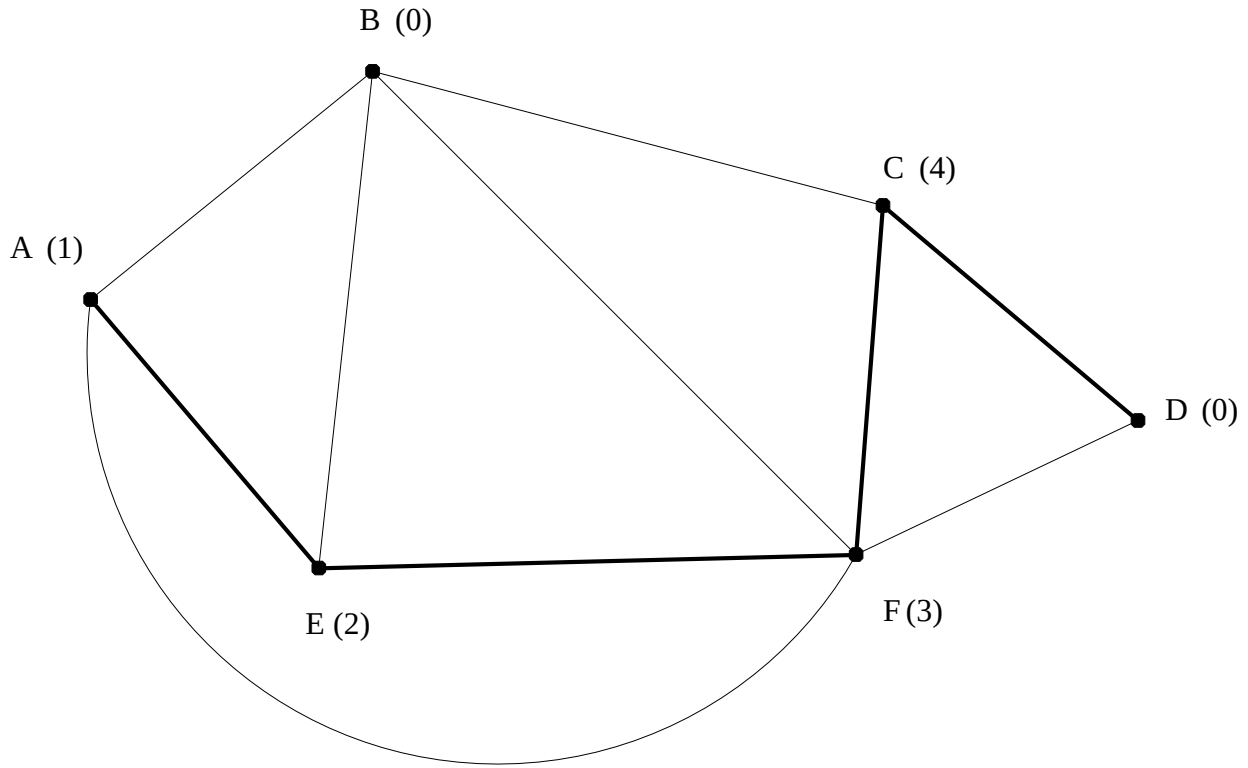
Here we add the edge (*E*,*F*) to our Depth First Search tree.

DepthFirstSearch(G,F)



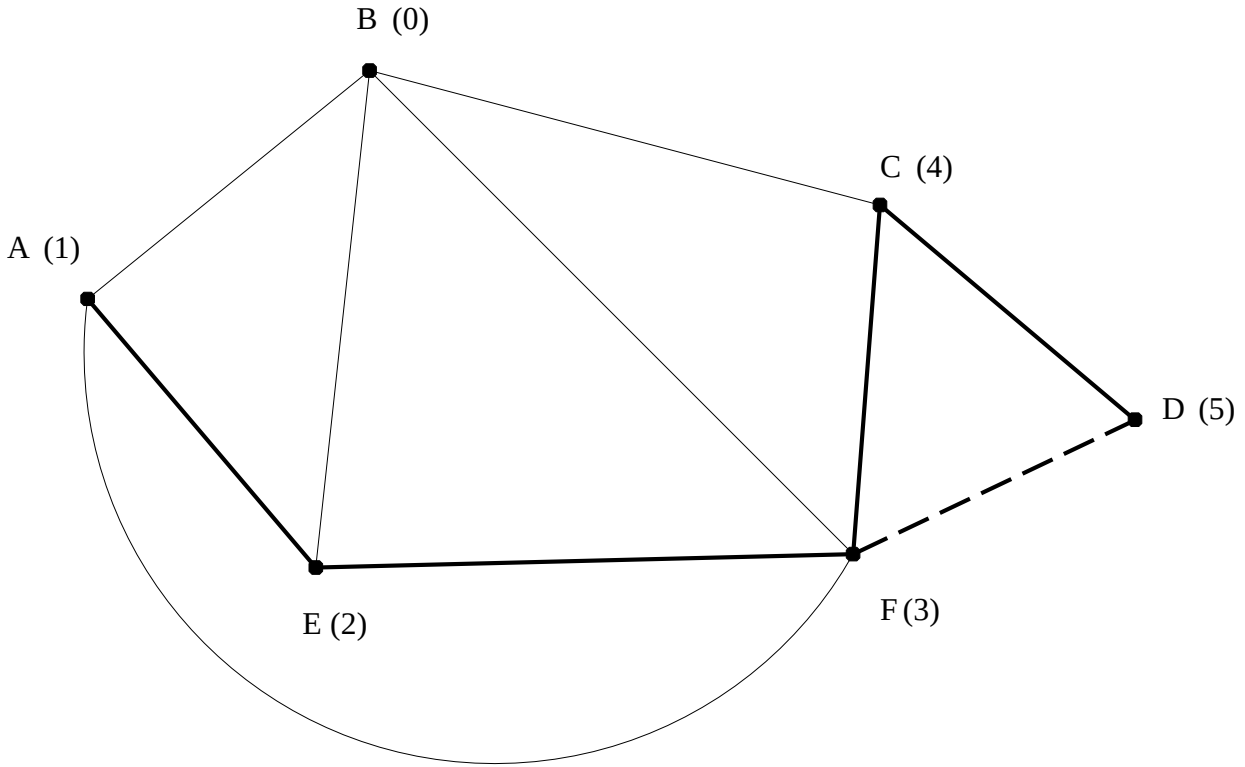
Here we add the edge (F, C) to our Depth First Search tree.

DepthFirstSearch(G,C)



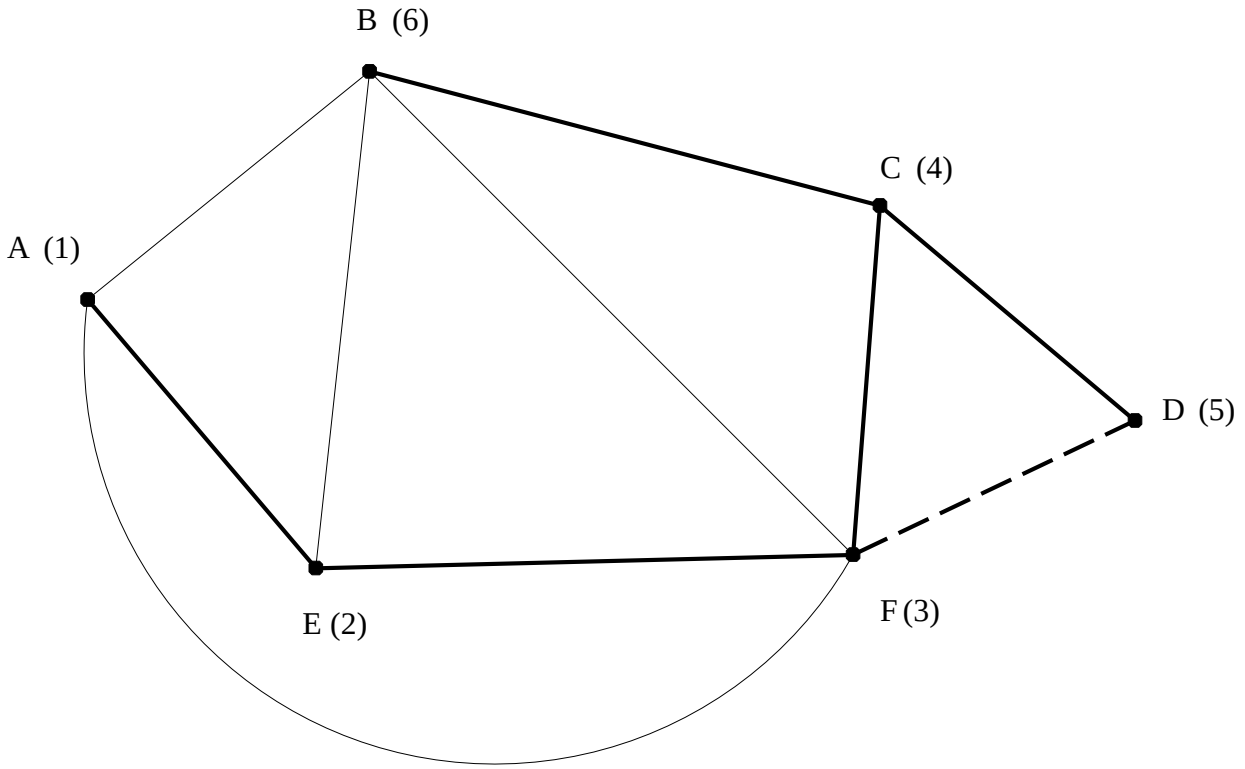
Here we add the edge (*C, D*) to our Depth First Search tree.

DepthFirstSearch(G,D)



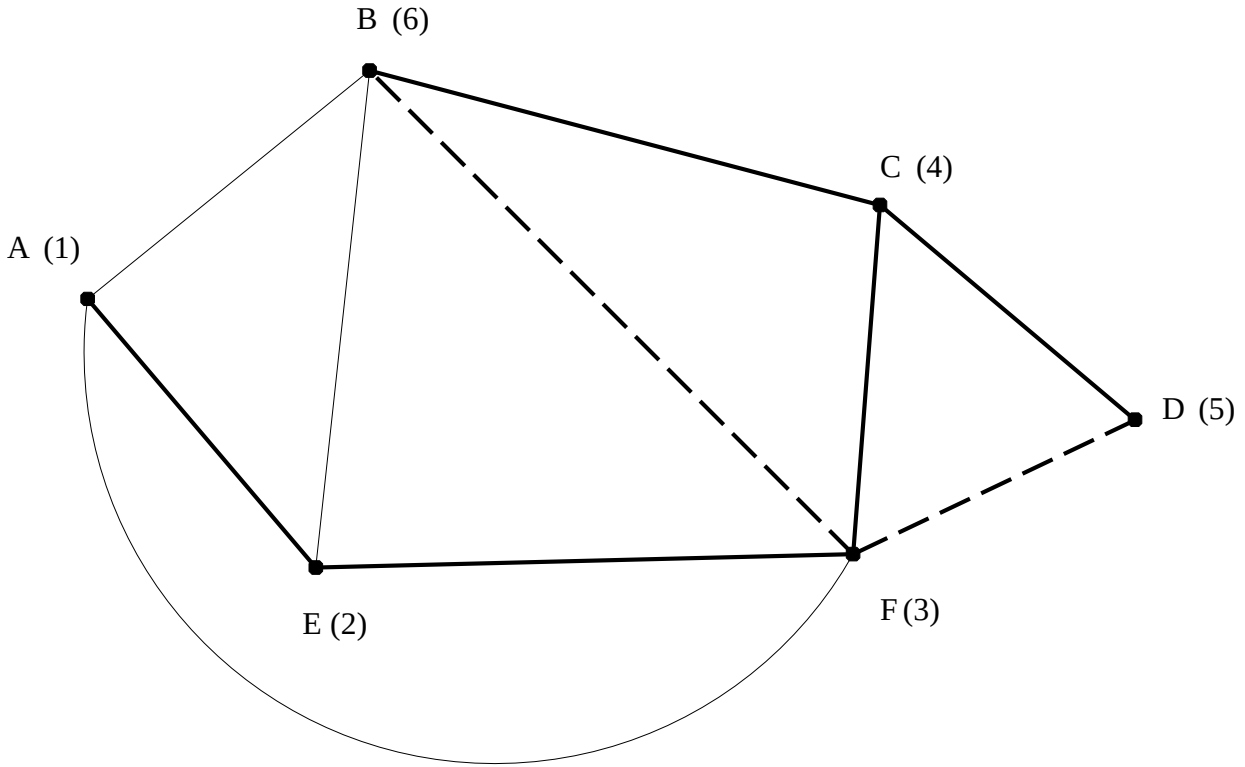
Here don't add any edge as the neighbor of *D* (*F*) has already been visited. The edge (*D, F*) is a *back edge*.

DepthFirstSearch(G,C)



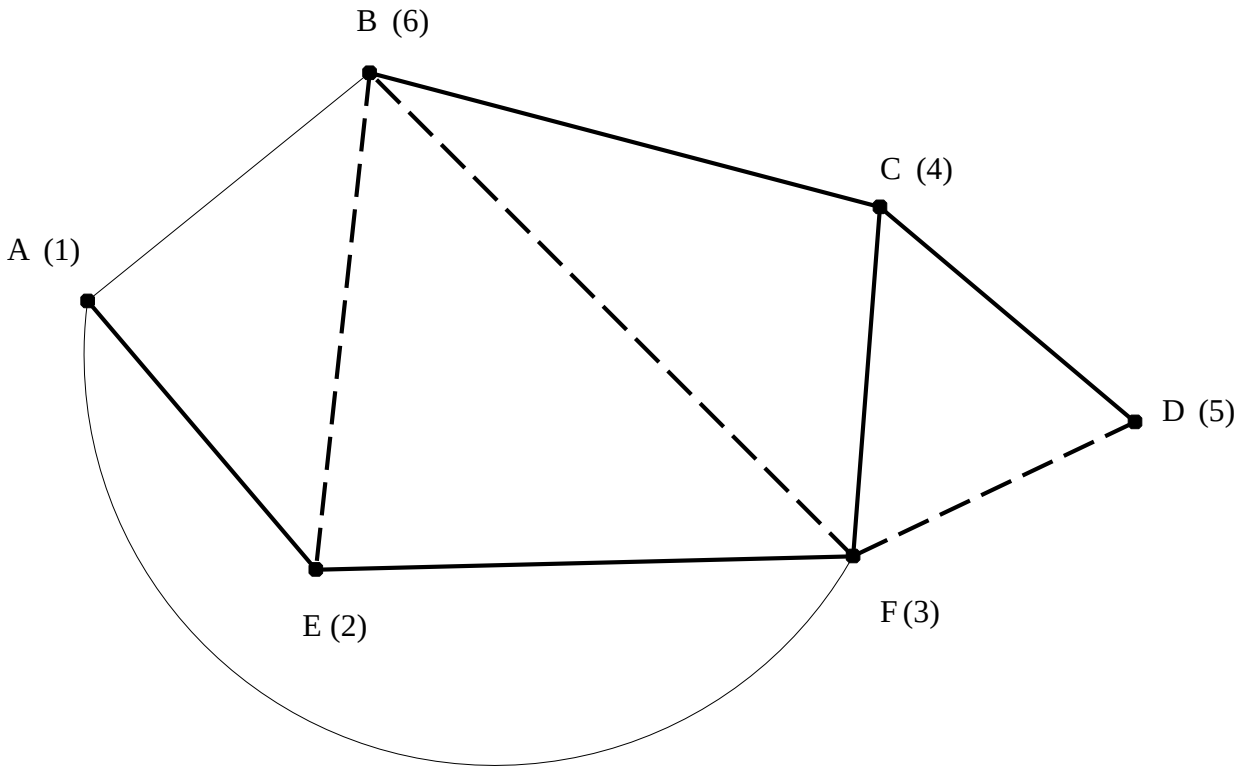
Here we add the edge (*C, B*) to our Depth First Search tree.

DepthFirstSearch(G,B)



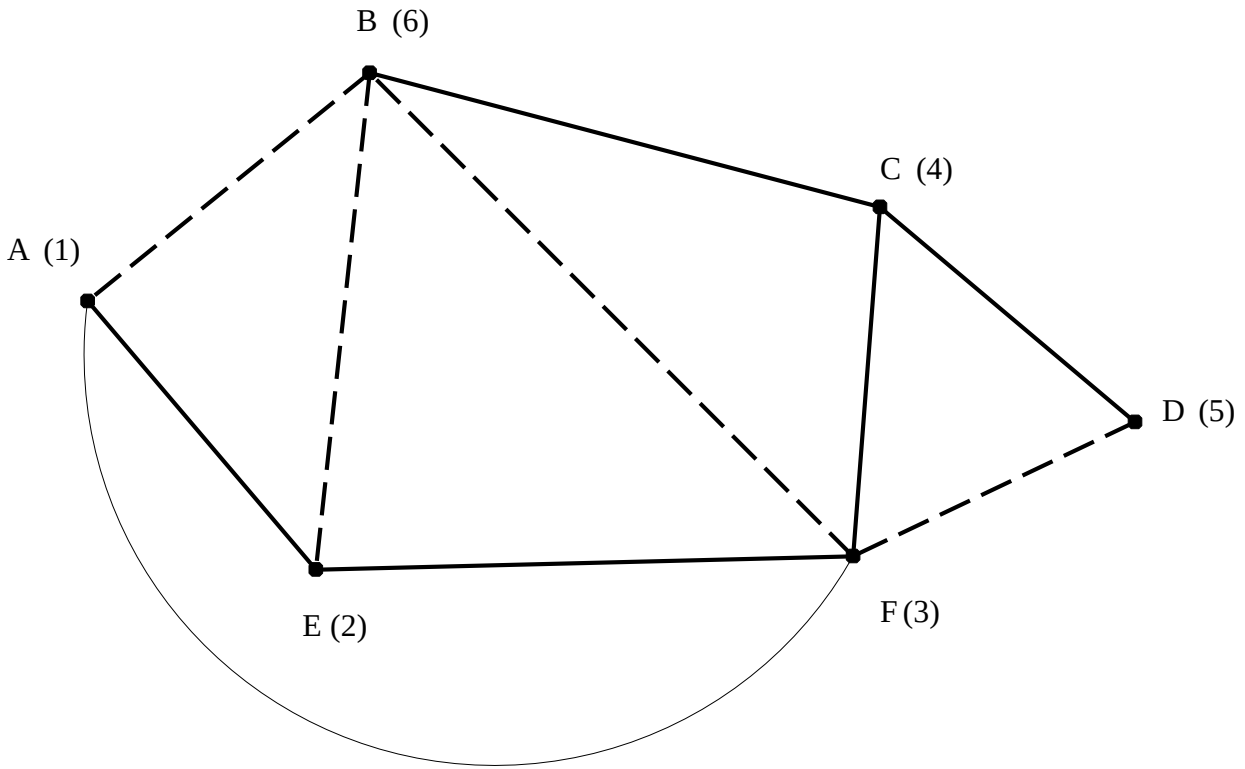
Here we don't add edge (*B, F*) as *F* has already been visited, so (*B, F*) is a *back edge*.

DepthFirstSearch(G,B)



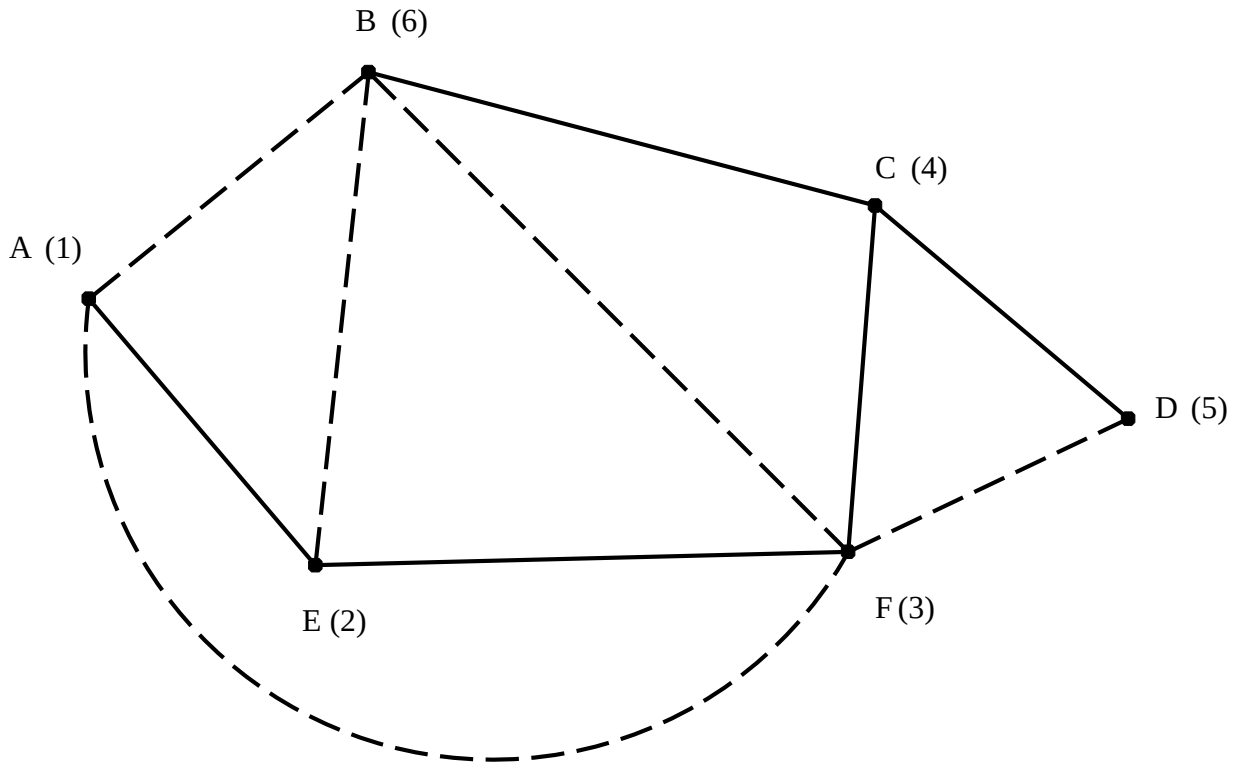
Here we don't add the edge (*B, E*) to our Depth First Search tree as *E* has already been visited,so the edge (*B, E*) is a *back edge*.

DepthFirstSearch(G,B)



Here we don't add the edge (*B, A*) to our Depth First Search tree as *A* has already been visited. The edge (*B, A*) is a *back edge*.

DepthFirstSearch(G,F)



Here we don't add the edge (*F, A*) to our Depth First Search tree as *A* has already been visited. The edge (*F, A*) is a *back edge*.

Depth First Search (DFS) Algorithm (non-recursive)

We can write a non-recursive version of the DFS Algorithm.

To control the search we use a *stack* data structure. This means that the most recently visited vertex is the one from which we continue our search.

Non-recursive DFS algorithm

```
DFS(G, v)
    ▷ Input: A graph G and a vertex v of G.
    ▷ Output: A labeling of the discovery and back edges of G.
 1  S.push(v)
 2  VISITED(v) ← true
 3  while not S.isEmpty() do
 4          u ← S.top()
 5          if there is an edge (u, w) and not VISITED(w) then
 6                  S.push(w)
 7                  VISITED(w) ← true
 8                  parent(w) ← u
 9                  Label the edge (u, w) as a discovery edge
10          elseif there is an edge (u, w) and VISITED(w) then
11                  Label the edge (u, w) as a back edge
12          else
13                  S.pop()
```

Breadth First Search

Breadth First Search (BFS) is the other common way of exploring a graph.

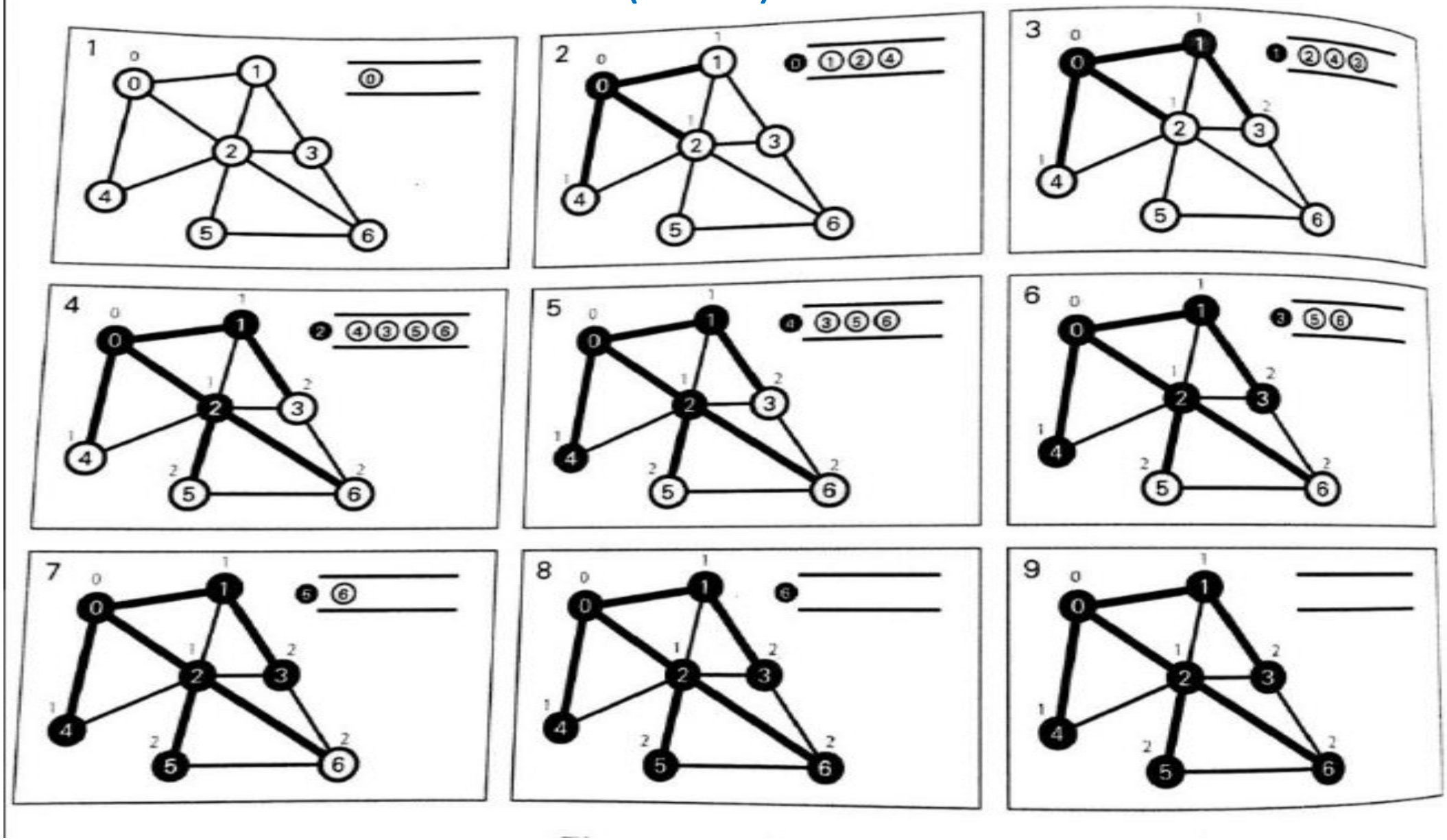
In contrast to the DFS method, the Breadth First Search algorithm starts at a vertex and first explores the entire neighborhood of that vertex.

So the DFS method generates “long, skinny” search trees, while the BFS method generates “short, bushy” ones.

Breadth First Search Algorithm

```
BFS(G, v)
    ▷ Input: A graph G and a vertex v of G.
    ▷ Output: A labeling of the edges of G.
 1  Q.enqueue(v)
 2  VISITED(v) ← true
 3  while not Q.isEmpty() do
 4          u ← Q.dequeue()
 5          for each edge (u, w) incident to u do
 6                  if not VISITED(w) then
 7                          Label (u, w) as a discovery edge
 8                          VISITED(w) ← true
 9                          Q.enqueue(w)
10                  else
11                          Label (u, w) as a cross edge
```

Breadth First Search (BFS)



DFS vs. BFS

DFS traversal:

- Is better for answering *complex connectivity questions*.
- Produces a spanning tree, such that, all non-tree edges are *back edges*.
  - Back edge: It is an edge (u, v) such that v is ancestor of node u but not part of DFS tree.

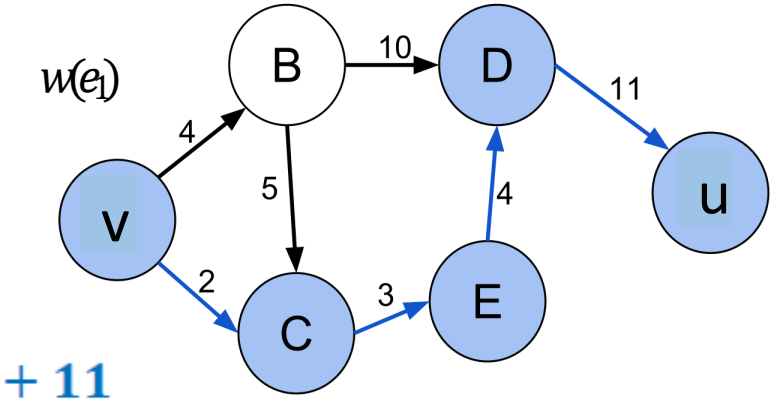
BFS traversal:

- Finds *shortest paths* in a graph.
- Produces a spanning tree, such that, all non-tree edges are *cross edges*.
  - It is a edge which connects two node such that they do not have any ancestor and a descendant relationship between them.

Weighted Graphs

A *weighted graph* is a graph that has a numerical label $w(e)$ associated with each edge e, called the *weight* of e.

The length (or weigh) of a path P is the sum of the weights of the edges $e_1, \ldots, e_k$ of P, i.e., $w(P) = \sum_{i=1}^{k} w(e_i)$

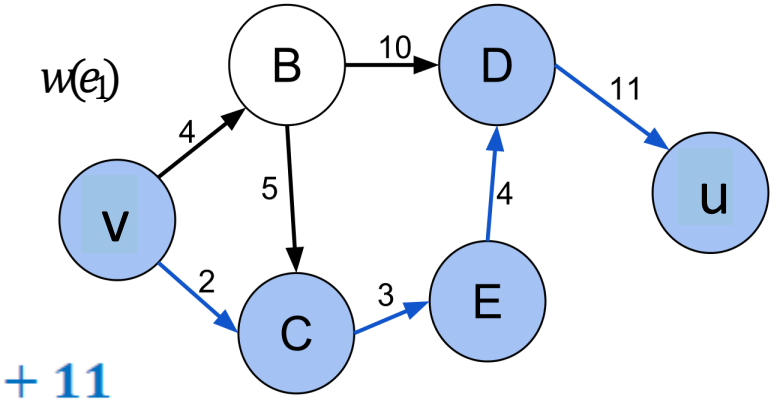


$$w(P) = 2 + 3 + 4 + 11$$

Weighted Graphs

A *weighted graph* is a graph that has a numerical label $w(e)$ associated with each edge e, called the *weight* of e.

The length (or weigh) of a path P is the sum of the weights of the edges $e_1, \ldots, e_k$ of P, i.e., $w(P) = \sum_{i=1}^{k} w(e_i)$



V-B-D-U: 25
V-B-C-E-D-U: 27
V-C-E-D-U: 20

$$w(P) = 2 + 3 + 4 + 11$$

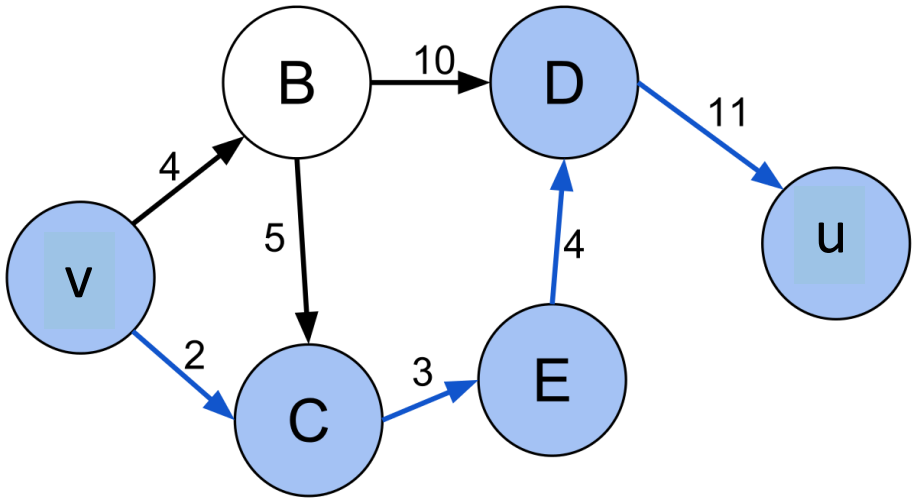
The distance from a vertex u to a vertex v in G, denoted $d(u,v)$ is the value of a minimum length path (also called a shortest path) from u to v.

Single-Source Shortest Paths

Often in the case of weighted graphs, we want to consider the following problem:

For some fixed vertex v,find the shortest path from v to all other vertices $u \neq v$ in G (viewing weights on edges as distances).

This problem is known as the *Single-Source Shortest Path* problem (SSSP for short).



Greedy Approach to SSSP

There is an interesting approach for solving the SSSP based on the *greedy method*.

The main idea in applying the greedy method to SSSP is to perform a "weighted" Breadth First Search.

One algorithm using this design pattern in known as *Dijkstra's algorithm*.

Assume that all edges in the graph have *non-negative weights*.

Greedy Approach to SSSP

There is an interesting approach for solving the SSSP based on the *greedy method*.

The main idea in applying the greedy method to SSSP is to perform a "weighted" Breadth First Search.

One algorithm using this design pattern in known as *Dijkstra's algorithm*.

Assume that all edges in the graph have *non-negative weights*.

Let *v* be a *source vertex* and let $D[u]$ represent the *temporary distance* in *G* from v to u. Initially we take $D[v]=0$ and $D[u]=+\infty$ for all $u \neq v$

At the start of the algorithm, all entries in the array D are temporary, but after each round of the algorithm one entry in D becomes *fixed*.

Edge Relaxation

Assume that C is a set of vertices for which entries in D are fixed (i.e. shortest distances between v and all $w \in C$ have been found), and suppose that entry $D[u]$ was the one fixed in the *most recent round*.

For any vertex z for which $D[z]$ is temporary, perform the *edge relaxation*:

- If $D[u] + w(\{u, z\}) < D[z]$ then $D[z] \leftarrow D[u] + w(\{u, z\})$.

In other words, if we find a better path from v to z (through the vertex u), we update $D[z]$ to this smaller value.

Fixing the next entry of D

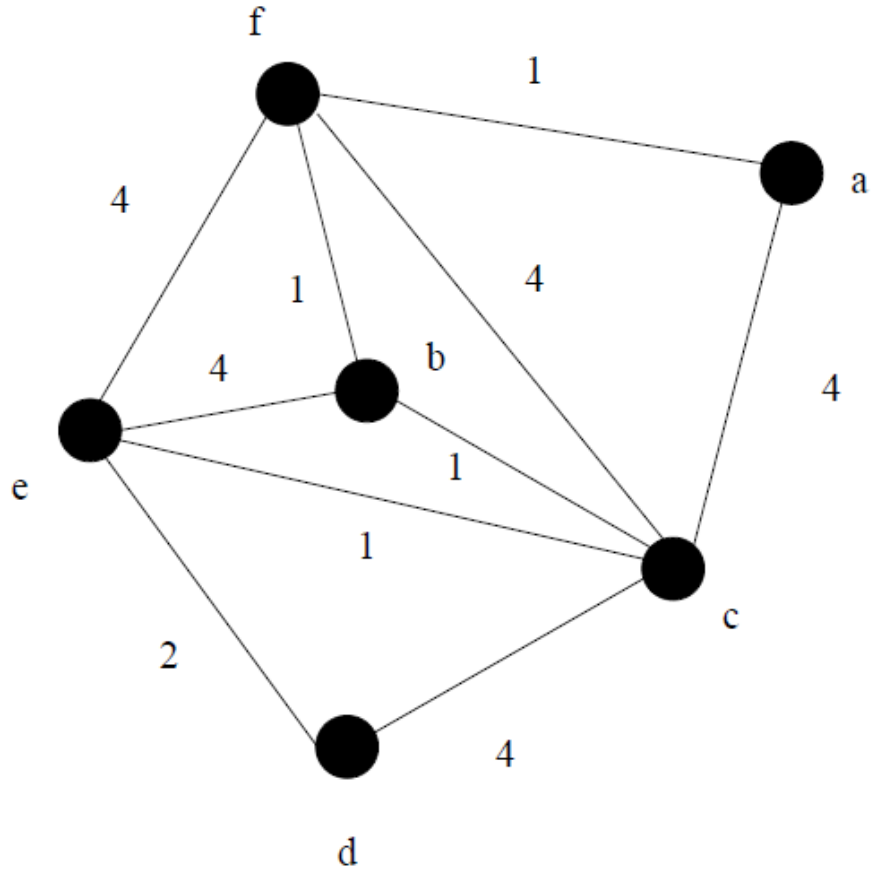
When the edge relaxation step is completed for all temporarily labeled vertices we then

- Fix one entry in *G* (among vertices still outside *C*) with the smallest weight currently available, and add that new vertex to *C*.
- Proceed to the next stage of edge relaxation based on this extended set of vertices *C*.

Dijkstra's algorithm

```
DIJKSTRA(G, v)
 1  D[v] ← 0
 2  for each u ≠ v do
 3          D[u] ← +∞
 4  Let Q be a priority queue (heap) having all vertices of G
    using the D labels as keys.
 5  while NOTEMPTY(Q) do
 6          u ← REMOVEMIN(Q)
 7          for each z s.t. (u, z) ∈ E do
 8                  if D[u] + w({u, z}) < D[z]
 9                    then D[z] ← D[u] + w({u, z})
10                         key(z) ← D[z]
11  return D
```

Dijkstra's algorithm

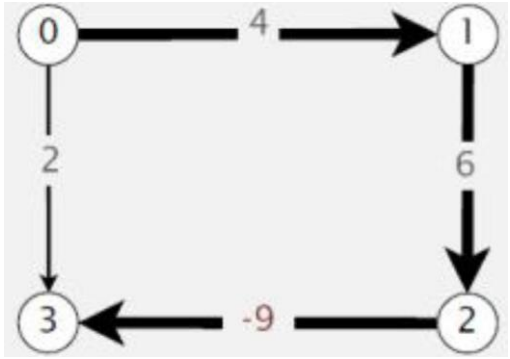


D	[a	b	c	d	e	f]	C
	∞	∞	∞	0	∞	∞	ϕ
	∞	∞	4	**0**	2	∞	$\{d\}$
	∞	6	3	**0**	**2**	6	$\{d, e\}$
	7	4	**3**	**0**	**2**	6	$\{c, d, e\}$
	7	**4**	**3**	**0**	**2**	5	$\{b, c, d, e\}$
	6	**4**	**3**	**0**	**2**	**5**	$\{b, c, d, e, f\}$
	6	**4**	**3**	**0**	**2**	**5**	$\{a, b, c, d, e, f\}$

Shortest paths with negative weights: failed attempts

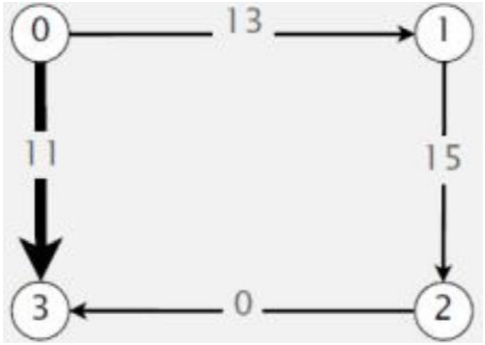
Shortest paths with negative weights: failed attempts

Dijkstra. Doesn't work with negative edge weights.



Dijkstra selects vertex 3 immediately after 0
But shortest path from 0 to 3 is $0 \to 1 \to 2 \to 3$.

Re-weighting. Add a constant to every edge weight doesn't work.



Adding 9 to each edge weight changes the shortest path from $0 \to 1 \to 2 \to 3$ to $0 \to 3$

Conclusion. Need a different algorithm.

Bellman-Ford Algorithm

The *Bellman-Ford* algorithm can find shortest paths in graphs that have *negative weight edges*.

However, the assumption in this case is that the graph *G* is *directed*.

The Bellman-Ford algorithm also uses the notion of edge relaxation, but it does not use it with the greedy method.

Instead it performs a relaxation of every edge *exactly* ($V - 1$) times.

This relies on the fact that, if there are no negative cycles, then there is a shortest path from *v* to any other vertex in *G* with at most $V - 1$ edges.