Graphs

Edited by Heshan Du
University of Nottingham Ningbo China

Learning Objectives

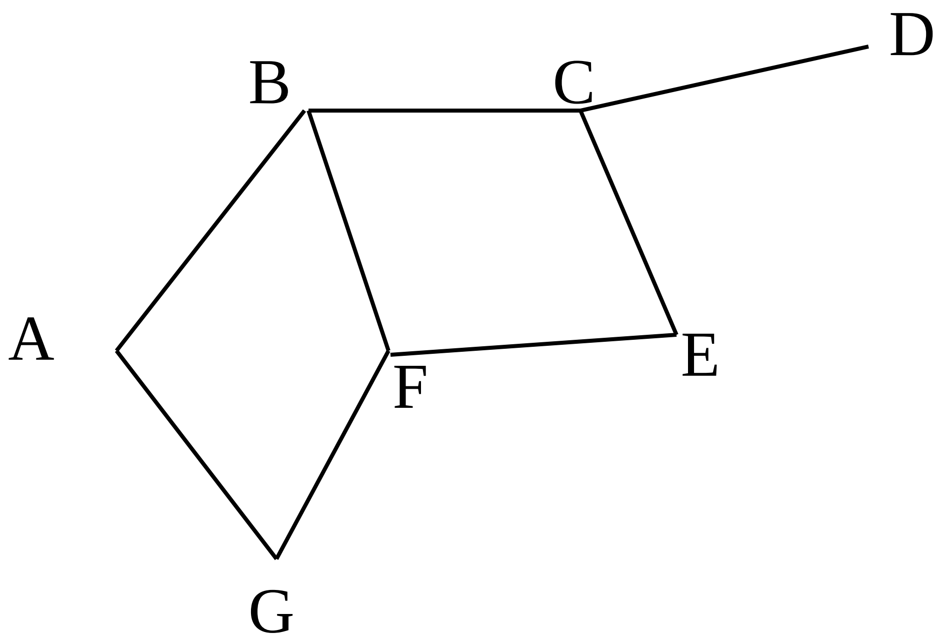
- To be able to *understand* and describe the definition of a graph and its related terminology;
- To be able to *understand* the Graph ADT;
- To be able to *implement* the Graph ADT and analyze the complexity of the methods;
- To be able to *apply* the Graph ADT to solve problems

Learning Objectives

- To be able to *understand* and describe graph traversal algorithms;
- To be able to *implement* graph traversal algorithms and analyze their complexity;
- To be able to *apply* graph traversal algorithms to solve problems

Definition of a graph

A graph is a set of ***nodes***, or ***vertices***, connected by ***edges***.



Applications of Graphs

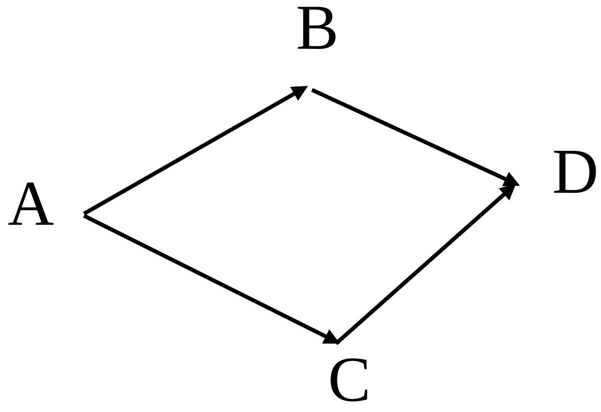
Graphs can be used to represent

- networks (e.g., of computers or roads)
- flow charts
- tasks in some project (some of which should be completed before others), so edges correspond to prerequisites.
- states of an automaton / program

Directed and Undirected Graphs

Graphs can be

- undirected (edges don't have direction)
- directed (edges have direction)



directed graph

Directed and Undirected Graphs

Undirected graphs can be represented as directed graphs where for each edge (X,Y) there is a corresponding edge (Y,X).

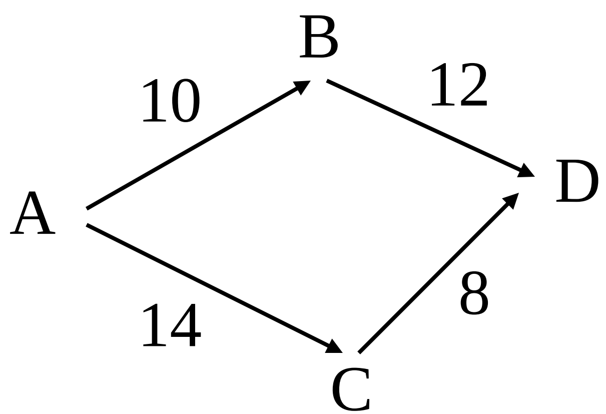
A —— B —— C   undirected graph

A ⇄ B ⇄ C   corresponding directed graph

Weighted and Unweighted Graphs

Graphs can also be

- unweighted (as in the previous examples)
- weighted (edges have weights)

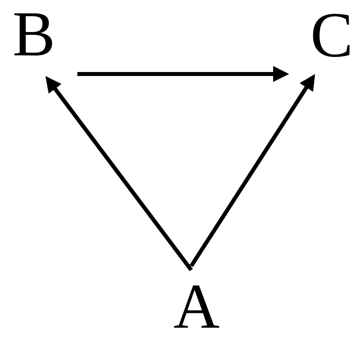


weighted graph

Notation

- Set V of *vertices* (nodes)
- Set E of *edges* ($E \subseteq V \times V$)

Example:



$V = \{A, B, C\}$, $E = \{(A,B), (A,C), (B,C)\}$

Adjacency relation

- Node B is *adjacent* to A if there is an edge from A to B.

$$A \longrightarrow B$$

Paths and reachability

- A *path* from A to B is a sequence of vertices $A_1,\ldots,A_n$ such that there is an edge from A to A_1, from A_1 to A_2, ..., from A_n to B.

$$A \longrightarrow A_1 \longrightarrow A_2 \longrightarrow A_3 \longrightarrow A_4 \longrightarrow A_5 \longrightarrow B$$

- What about the case where there is an edge from A to B?
- A path is a sequence of alternating vertices and edges that starts at a vertex and ends at a vertex such that each edge is incident to its predecessor and successor vertex.
- A vertex B is *reachable* from A if there is a path from A to B

More Terminology

- A *cycle* is a path from a vertex to itself
- Graph is *acyclic* if it does not have cycles
- Graph is *connected* if there is a path between every pair of vertices
- Graph is *strongly connected* if there is a path in both directions between every pair of vertices

Applications of Graphs

For example,

- nodes could represent positions in a board game, and edges the moves that transform one position into another ...
- nodes could represent computers (or routers) in a network and weighted edges the bandwidth between them
- nodes could represent towns and weighted edges road distances between them, or train journey times or ticket prices ...

Graph ADT

numVertices( ): Returns the number of vertices of the graph.

vertices( ): Returns an iteration of all the vertices of the graph.

numEdges( ): Returns the number of edges of the graph.

edges( ): Returns an iteration of all the edges of the graph.

getEdge(u, v): Returns the edge from vertex u to vertex v, if one exists; otherwise return null. For an undirected graph, there is no difference between getEdge(u, v) and getEdge(v, u).

endVertices(e): Returns an array containing the two endpoint vertices of edge e. If the graph is directed, the first vertex is the origin and the second is the destination.

opposite(v, e): For edge e incident to vertex v, returns the other vertex of the edge; an error occurs if e is not incident to v.

Graph ADT

outDegree(v): Returns the number of outgoing edges from vertex v.

inDegree(v): Returns the number of incoming edges to vertex v. For an undirected graph, this returns the same value as does outDegree(v).

outgoingEdges(v): Returns an iteration of all outgoing edges from vertex v.

incomingEdges(v): Returns an iteration of all incoming edges to vertex v. For an undirected graph, this returns the same collection as does outgoingEdges(v).

Graph ADT

insertVertex(x): Creates and returns a new Vertex storing element x.

insertEdge(u, v, x): Creates and returns a new Edge from vertex u to vertex v, storing element x; an error occurs if there already exists an edge from u to v.

removeVertex(v): Removes vertex v and all its incident edges from the graph.

removeEdge(e): Removes edge e from the graph.

Some graph problems

- Searching a graph for a vertex
- Searching a graph for an edge
- Finding a path in the graph (from one vertex to another)
- Finding the shortest path between two vertices
- Cycle detection

More graph problems

- Topological sort (finding a linear sequence of vertices which agrees with the direction of edges in the graph, e.g., for scheduling tasks in a project)

- Minimal spanning tree (deleting as many edges in a graph as possible, so that all vertices are still connected by shortest possible edges, e.g., in network or circuit design.)

How to implement a graph

As with lists, there are several approaches, e.g.,

- using a static indexed data structure
- using a dynamic data structure

Static implementation:Adjacency Matrix

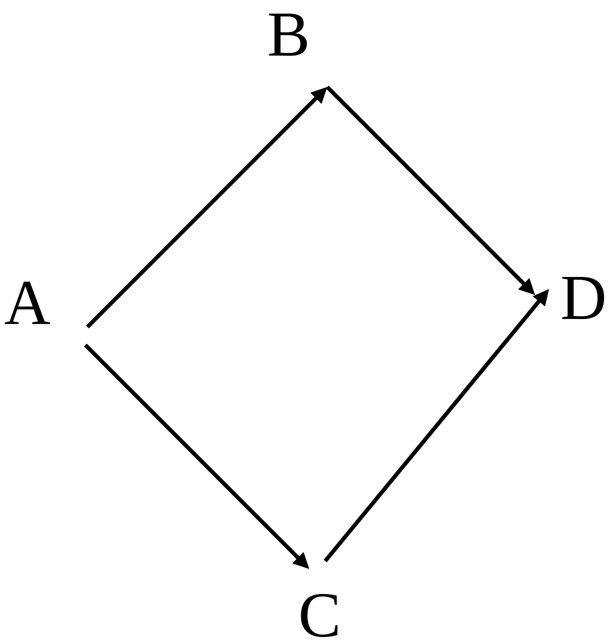
- Store node in the array: each node is associated with an integer (array index)

- Represent information about the edges using a two dimensional array, where

```
array[i][j] == 1
```

iff there is an edge from node with index *i* to the node with index *j*.

Example



A	B	C	D
0	1	2	3

node indices

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

adjacency matrix

Weighted graphs

- For weighted graphs, place weights in matrix (if there is no edge we use a value which can not be confused with a weight, e.g., -1 or **`Integer.MAX_VALUE`**)

Disadvantages of adjacency matrices

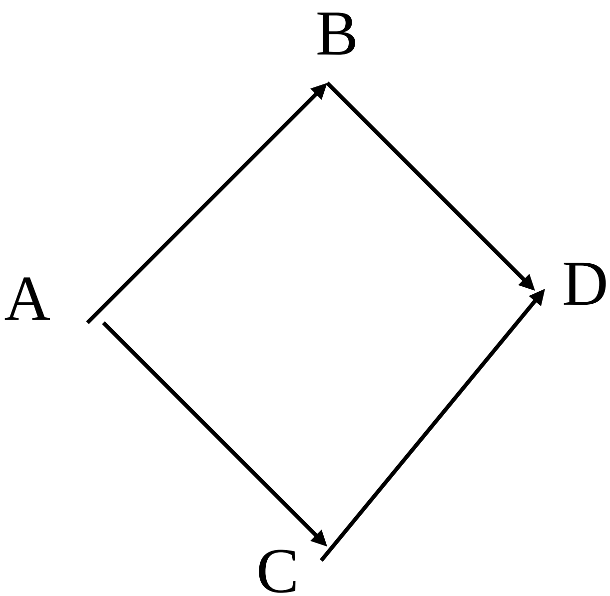
- Sparse graphs with few edges for number of vertices result in many zero entries in adjacency matrix—this *wastes space* and makes many algorithms *less efficient* (e.g., to find nodes adjacent to a given node, we have to iterate through the whole row even if there are few 1s there).

- Also, if the number of nodes in the graph may change, matrix representation is too *inflexible* (especially if we don't know the maximal size of the graph).

Adjacency List

- For every vertex, keep a list of adjacent vertices.
- Keep a list of vertices, or keep vertices in a Map (e.g. HashMap) as keys and lists of adjacent vertices as values.

Adjacency list



nodes — list of adjacent nodes

A → B, C

B → D

C → D

D →

Reading

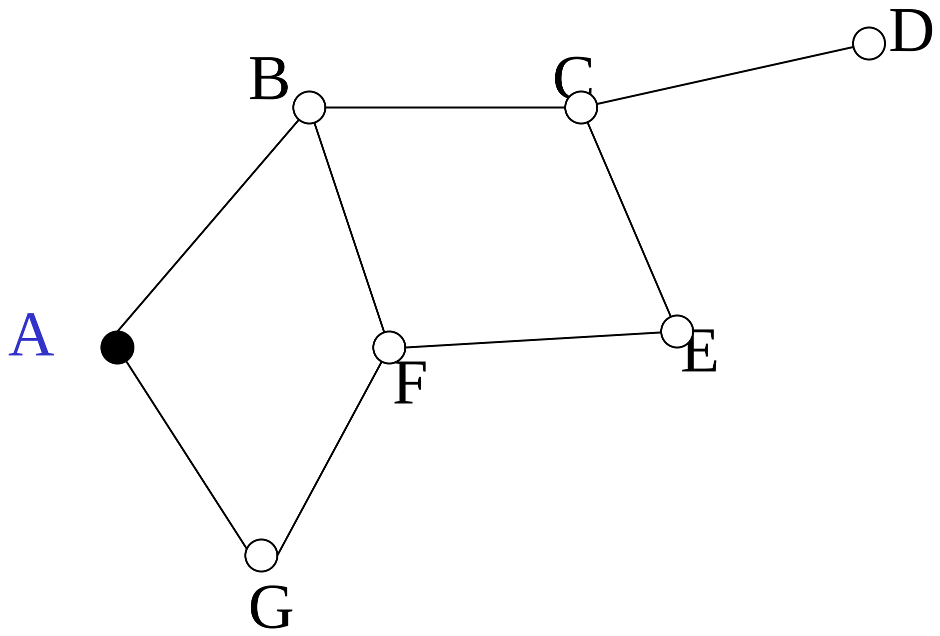
- Goodrich and Tamassia (Ch. 14) have a somewhat different Graph implementation, where edges are first-class objects.
- In general, choice of implementation depends on what we want to do with a graph.

Graph traversals

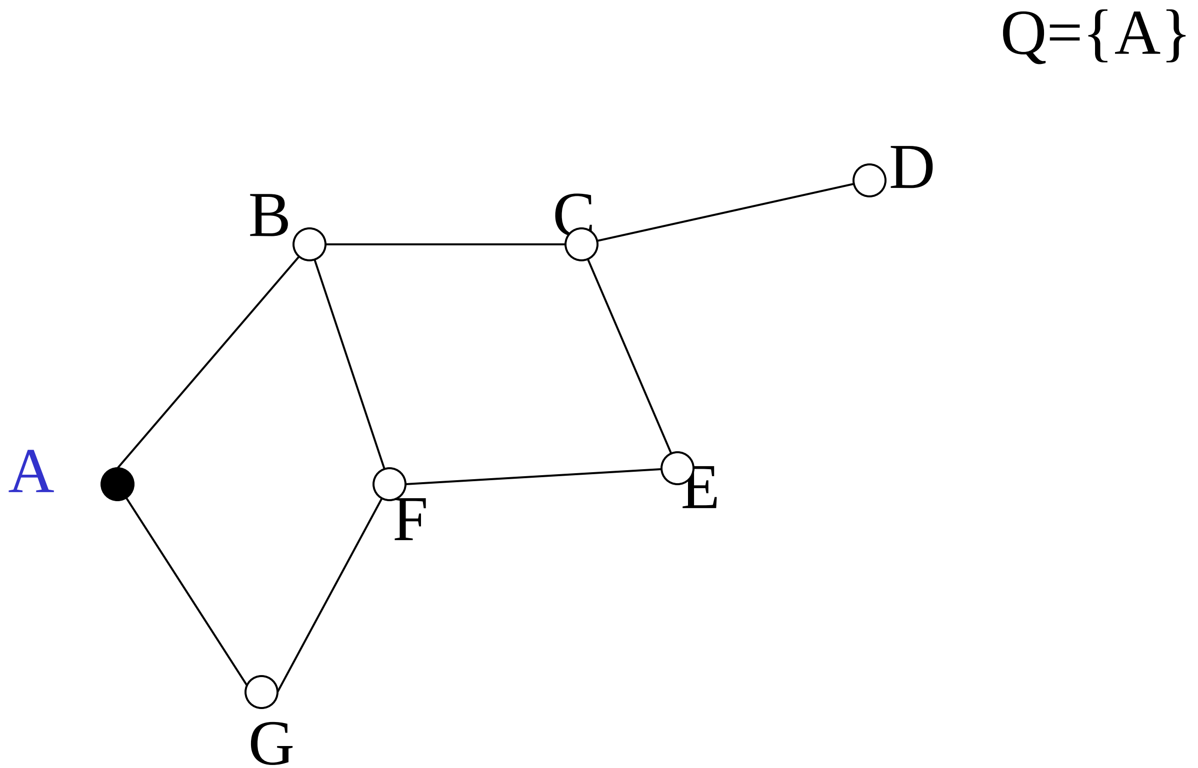
- In this lecture, we look at two ways of visiting all vertices in a graph: *breadth-first search* and *depth-first search*.
- Traversal of the graph is used to perform tasks such as searching for a certain node
- It can also be slightly modified to search for a path between two nodes, check if the graph is connected, check if it contains loops, and so on.

Graph traversal starting from A:

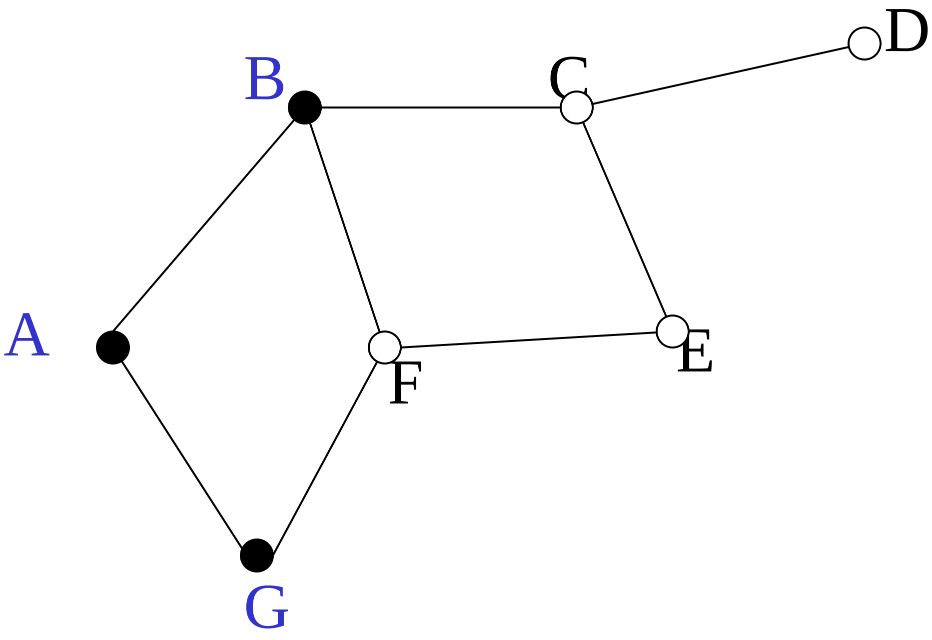
- *Exercise: What might we do?*



BFS starting from A:



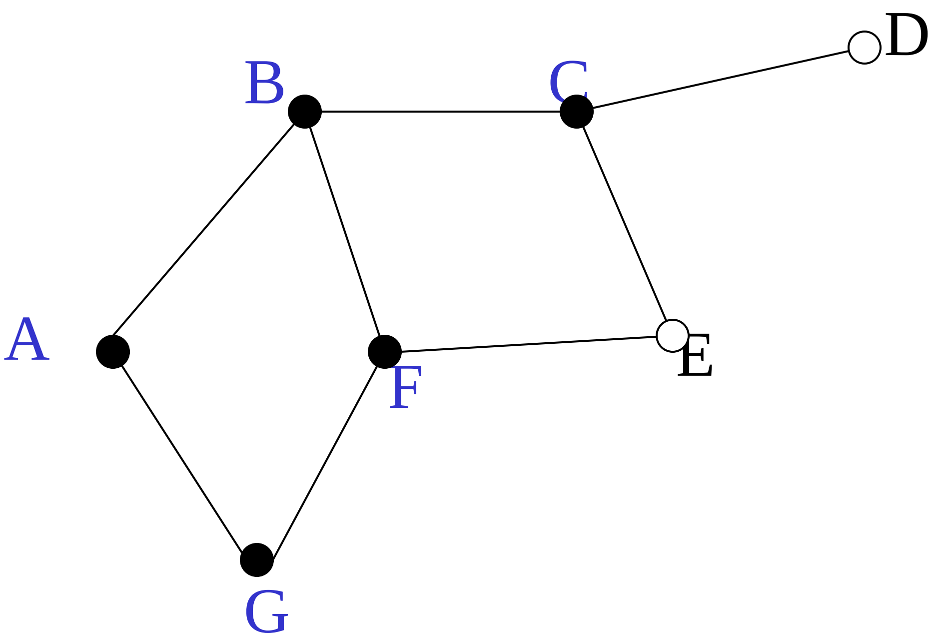
BFS starting from A:



Q={A}

Q={B,G}

BFS starting from A:

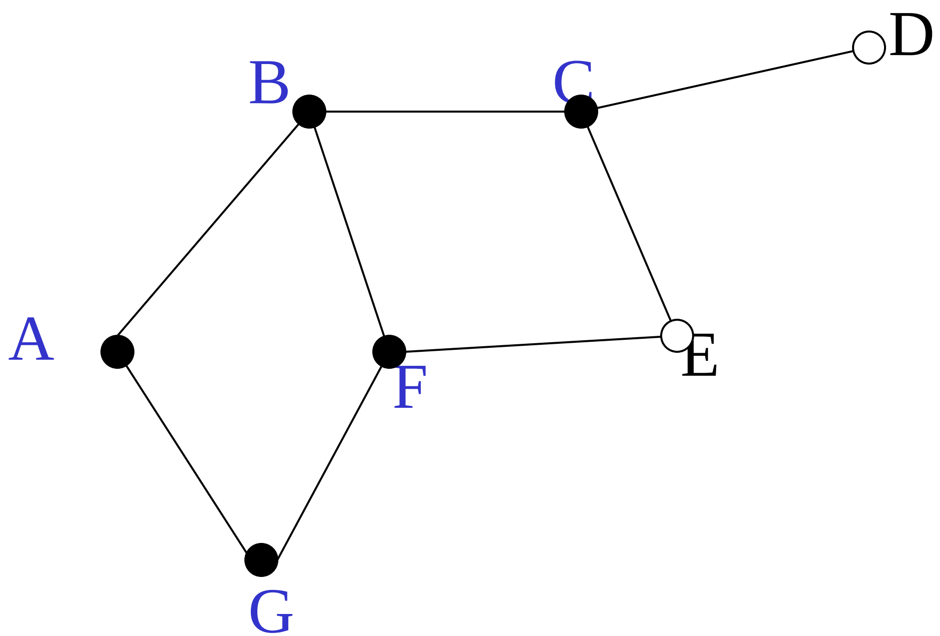


Q={A}

Q={B,G}

Q={G,C,F}

BFS starting from A:



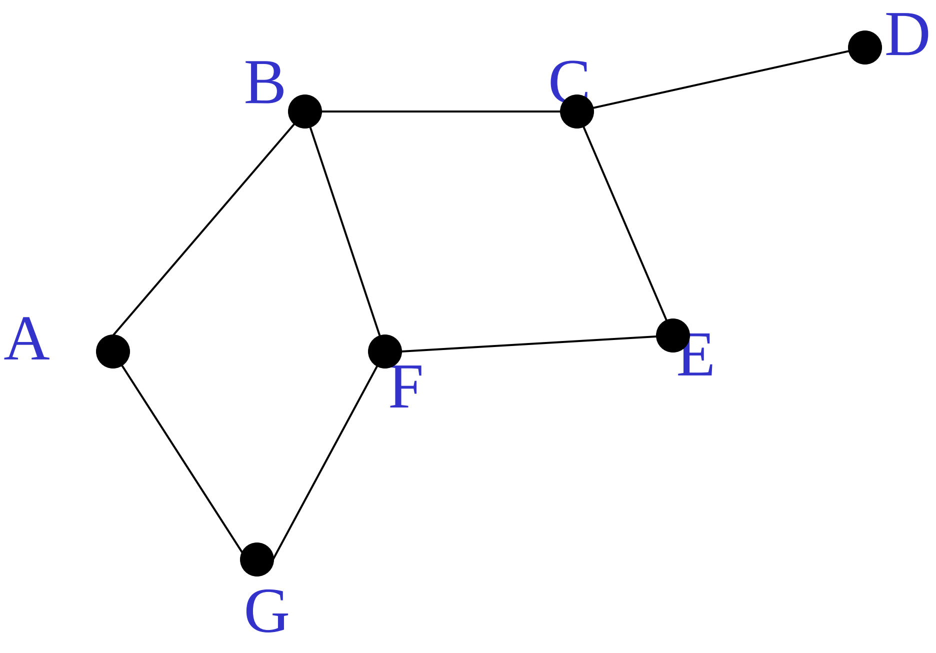
Q={A}

Q={B,G}

Q={G,C,F}

Q={C,F}

BFS starting from A:



Q={A}

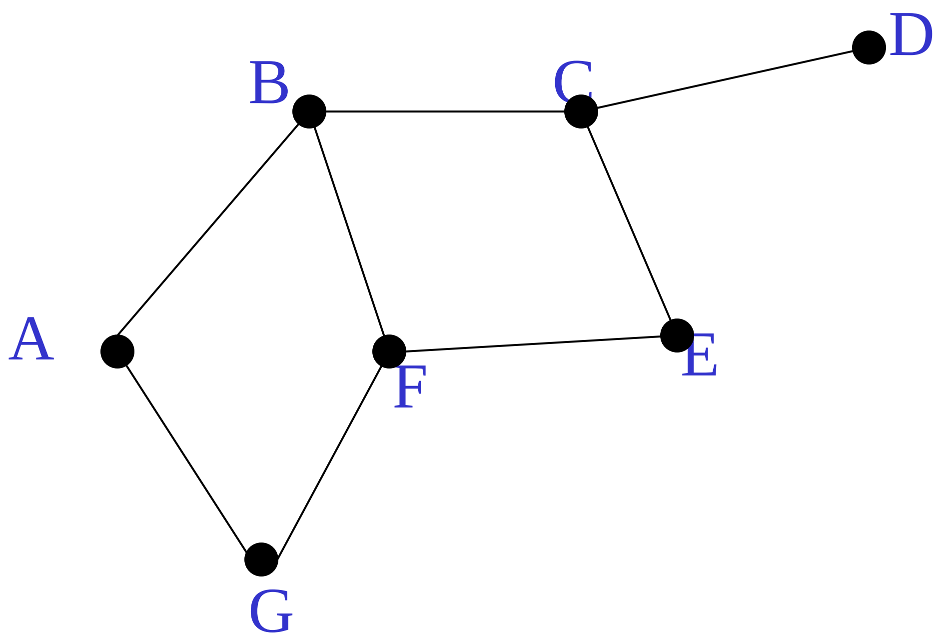
Q={B,G}

Q={G,C,F}

Q={C,F}

Q={F,D,E}

BFS starting from A:



Q={A}

Q={B,G}

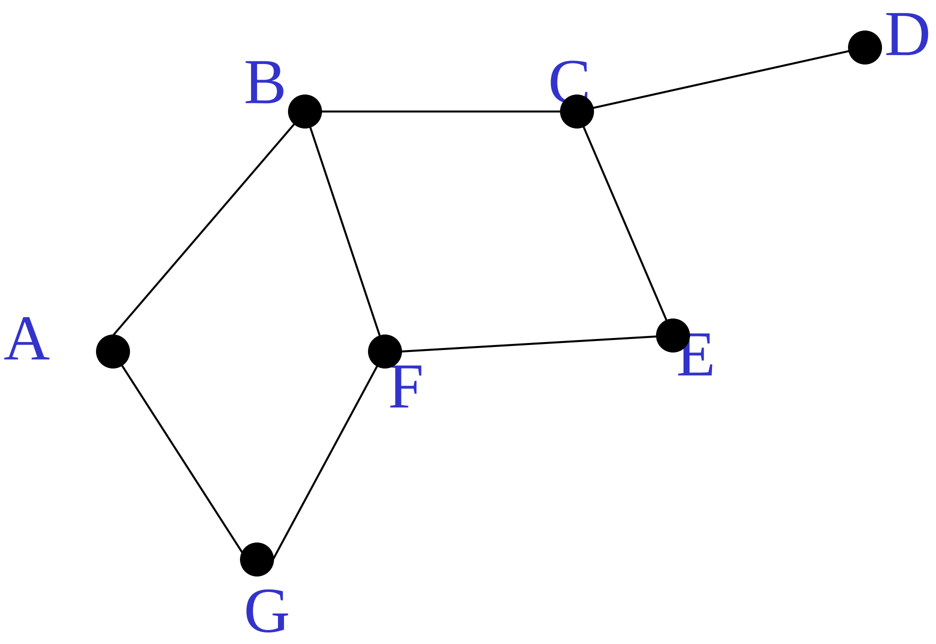
Q={G,C,F}

Q={C,F}

Q={F,D,E}

Q={D,E}

BFS starting from A:



Q={A}

Q={B,G}

Q={G,C,F}

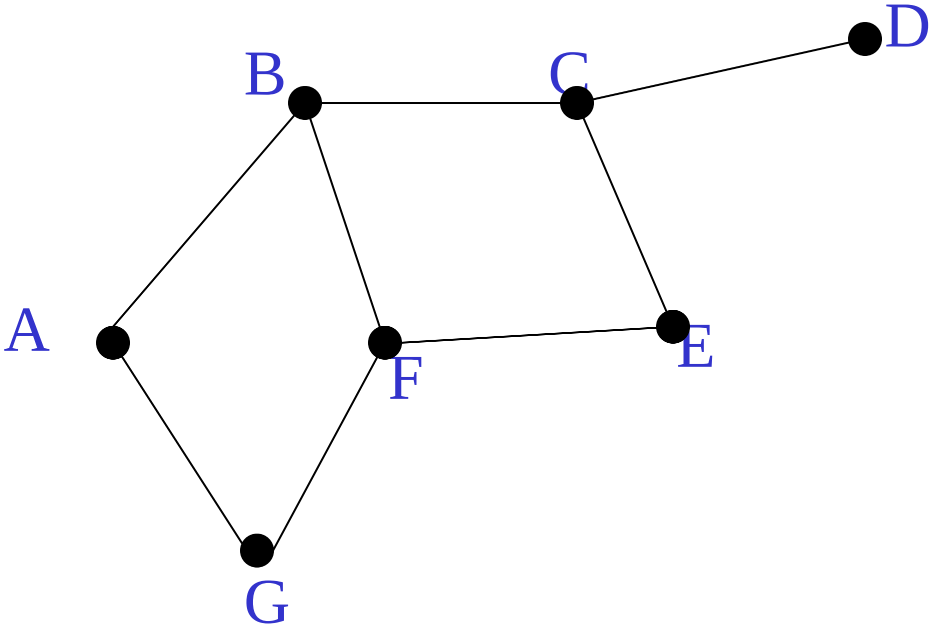
Q={C,F}

Q={F,D,E}

Q={D,E}

Q={E}

BFS starting from A:



Q={A}

Q={B,G}

Q={G,C,F}

Q={C,F}

Q={F,D,E}

Q={D,E}

Q={E}

Q={}

Breadth first search

BFS starting from vertex v:

```
create a queue Q
mark v as visited and put v into Q
while Q is non-empty
   remove the head u of Q
   mark and enqueue all (unvisited)
   neighbours of u
```

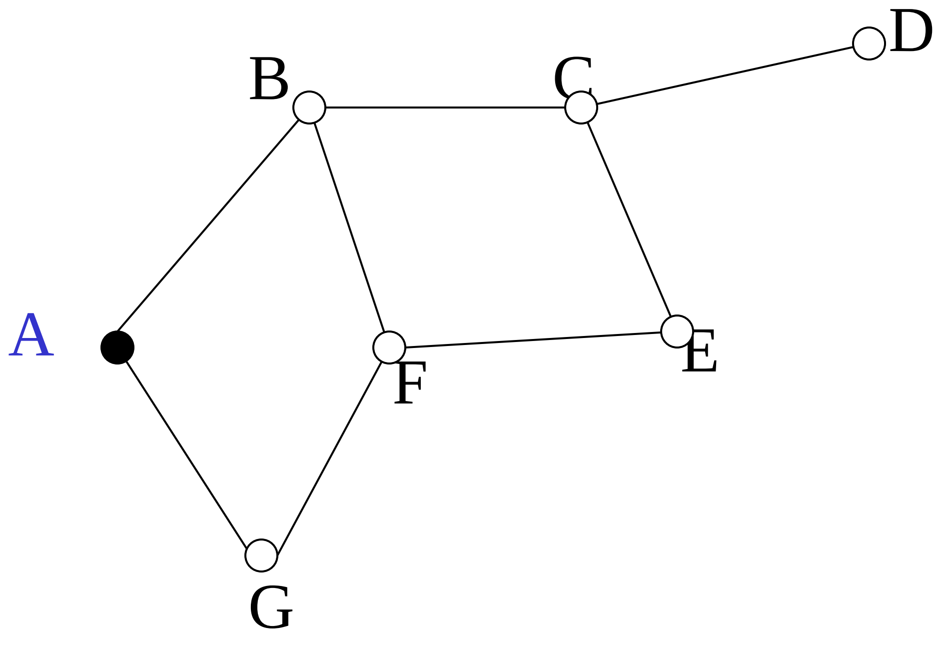
Overall Traversal Order: BFS

- In this example, the nodes are traversed from the starting point A in this order:

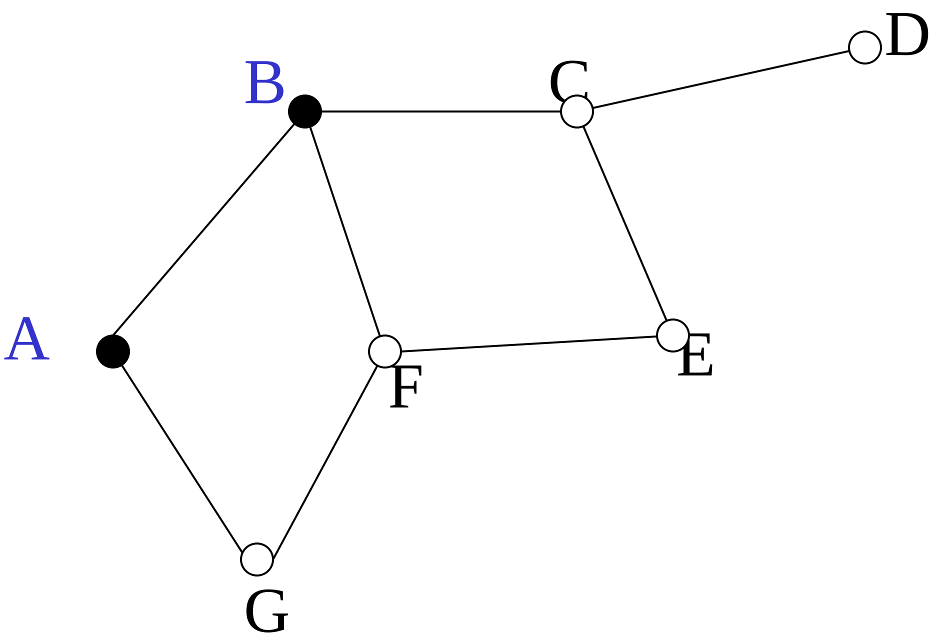
  A B G C F D E
- The BFS order is that those closest to the start point A occur earliest
- The order is not generally unique; e.g. either of B or G could occur first

DFS starting from A:

S={A}



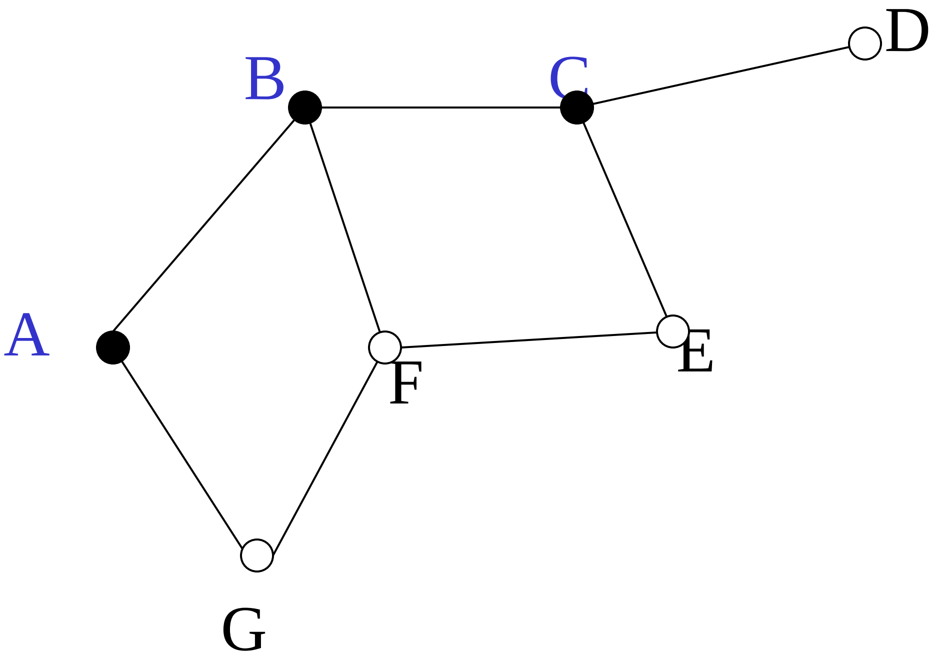
DFS starting from A:



S={A}

S={A,B}

DFS starting from A:

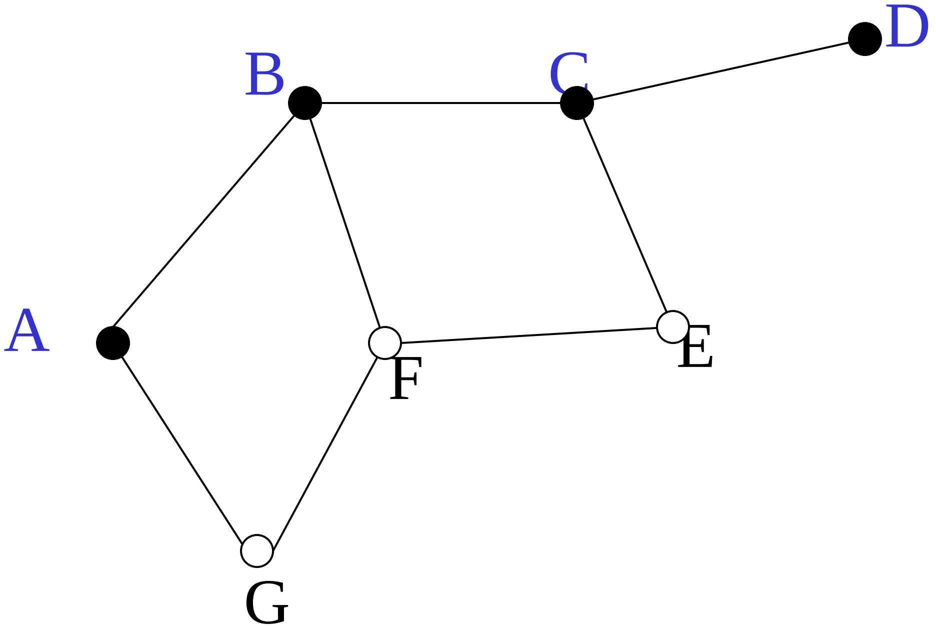


S={A}

S={A,B}

S={A,B,C}

DFS starting from A:



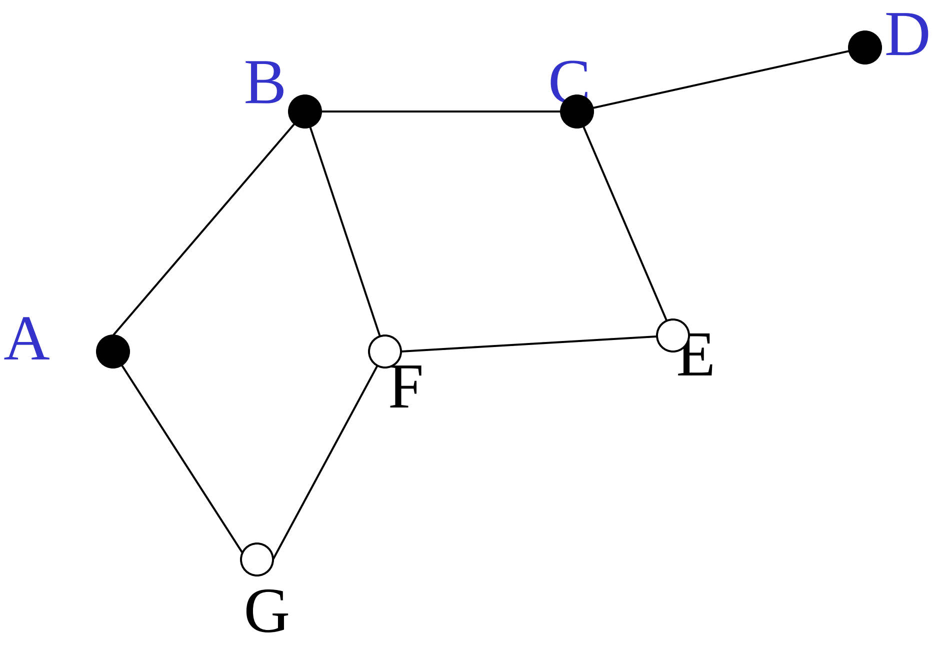
S={A}

S={A,B}

S={A,B,C}

S={A,B,C,D}

DFS starting from A:



S={A}

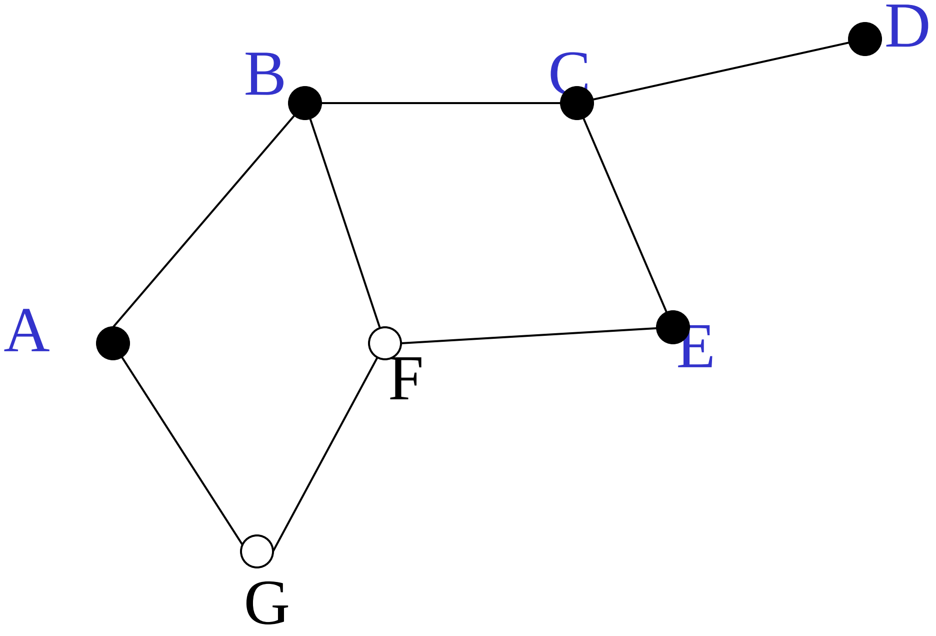
S={A,B}

S={A,B,C}

S={A,B,C,D}

S={A,B,C}

DFS starting from A:



S={A}

S={A,B}

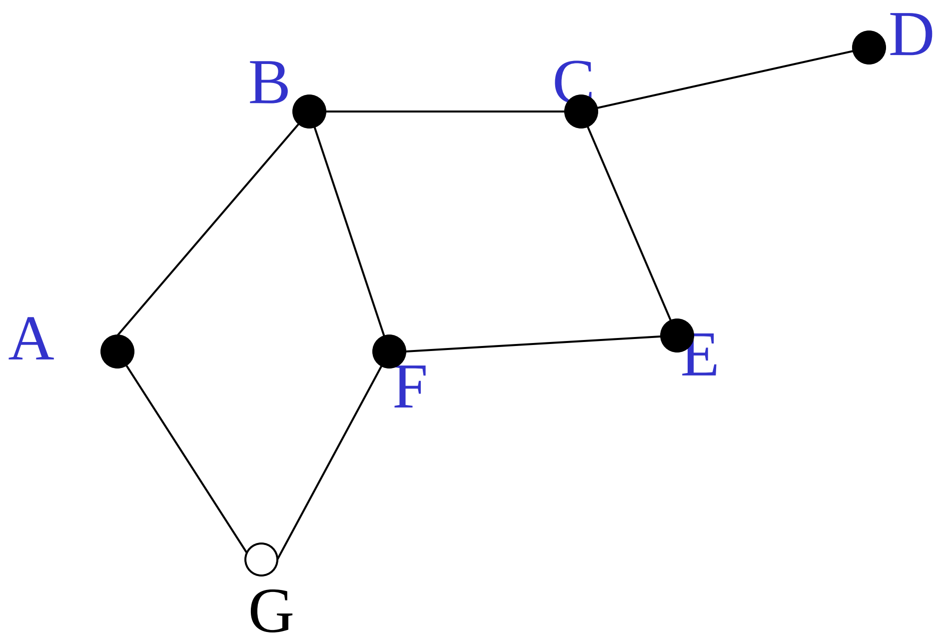
S={A,B,C}

S={A,B,C,D}

S={A,B,C}

S={A,B,C,E}

DFS starting from A:



S={A}

S={A,B}

S={A,B,C}

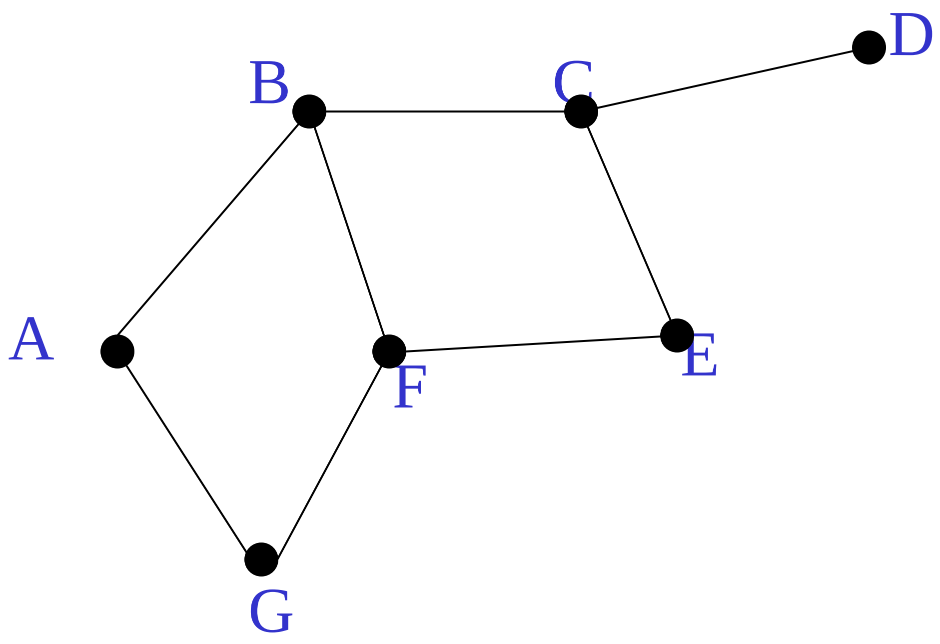
S={A,B,C,D}

S={A,B,C}

S={A,B,C,E}

S={A,B,C, E, F}

DFS starting from A:



S={A}

S={A,B}

S={A,B,C}

S={A,B,C,D}

S={A,B,C}

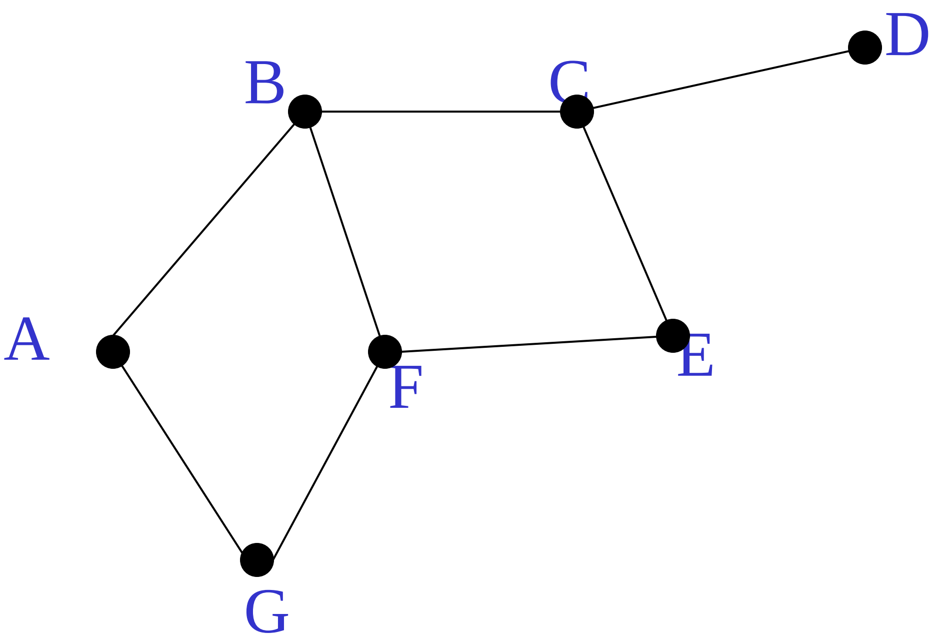
S={A,B,C,E}

S={A,B,C,E,F}

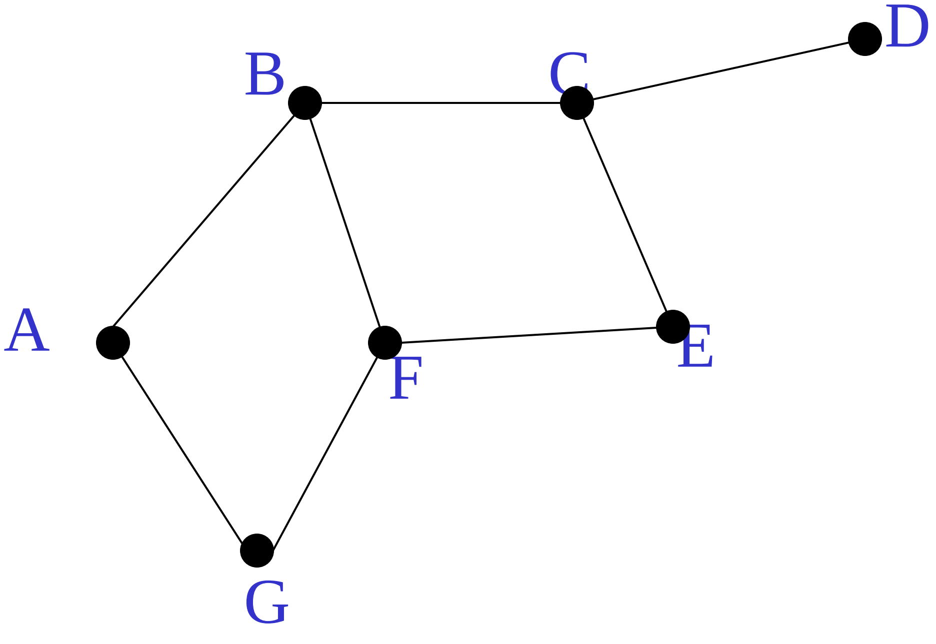
S={A,B,C,E,F,G}

DFS starting from A:

S={A,B,C,E,F}



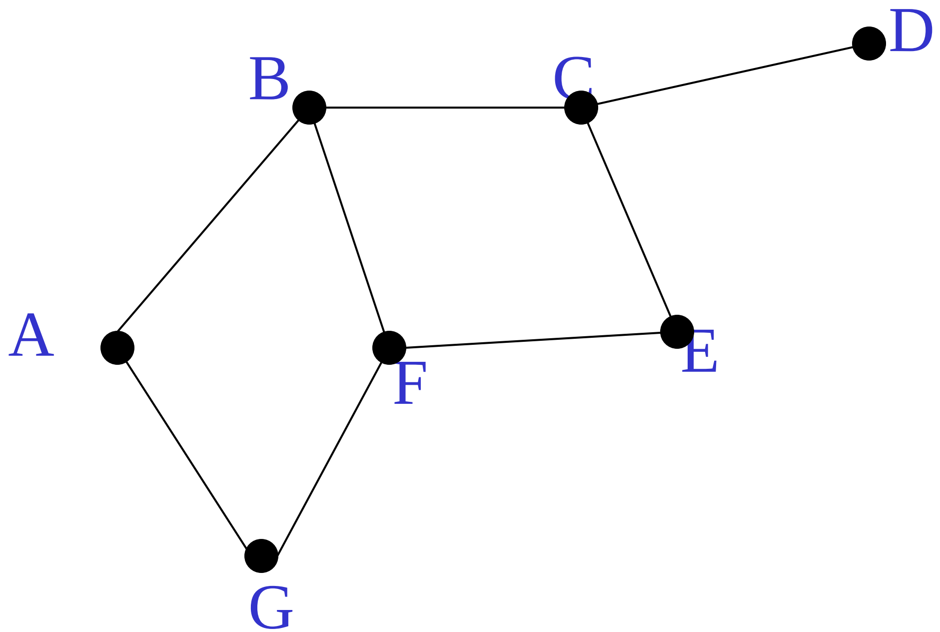
DFS starting from A:



S={A,B,C,E,F}

S={A,B,C,E}

DFS starting from A:

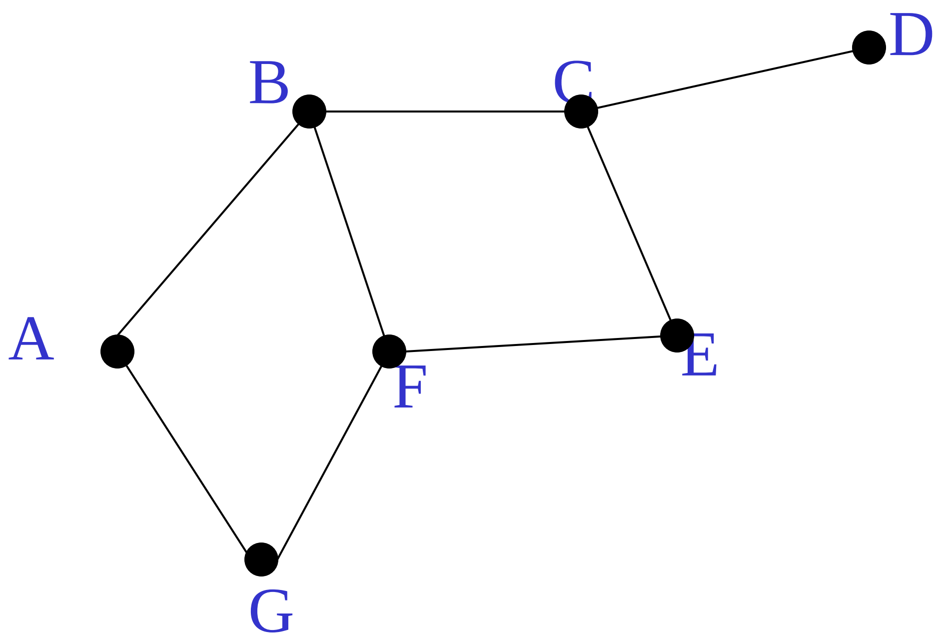


S={A,B,C,E,F}

S={A,B,C,E}

S={A,B,C}

DFS starting from A:



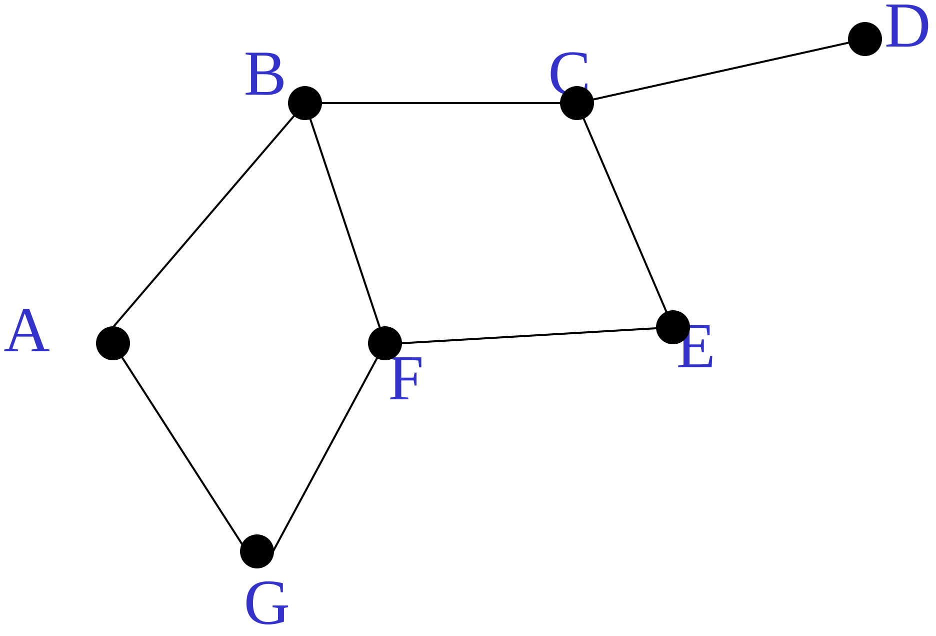
S={A,B,C,E,F}

S={A,B,C,E}

S={A,B,C}

S={A,B}

DFS starting from A:



S={A,B,C,E,F}

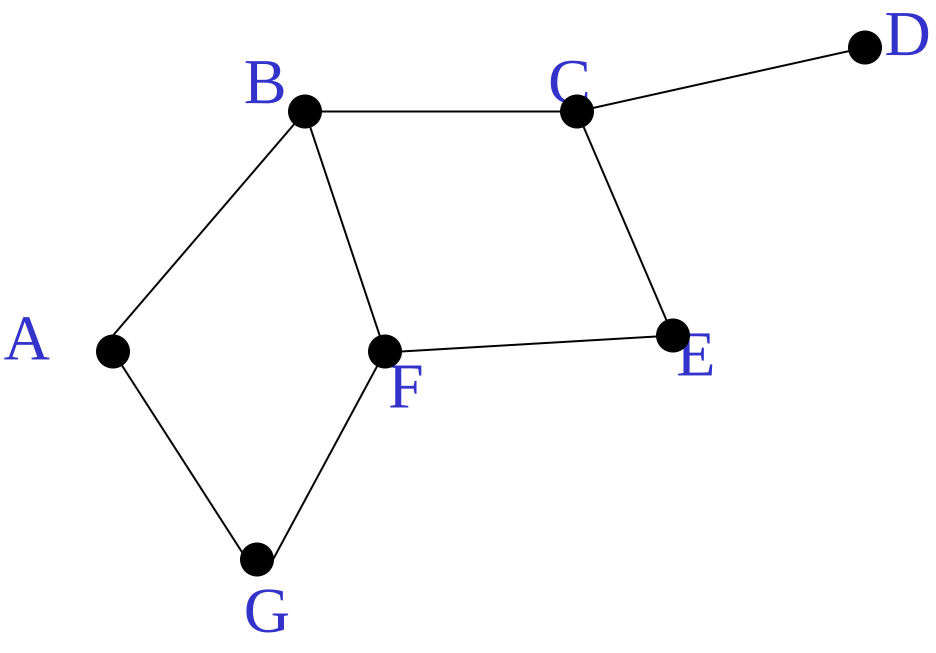
S={A,B,C,E}

S={A,B,C}

S={A,B}

S={A}

DFS starting from A:



S={A,B,C,E,F}

S={A,B,C,E}

S={A,B,C}

S={A,B}

S={A}

S={}

Simple DFS

DFS starting from vertex v:

```
create a stack S
mark v as visited and push v onto S
while S is non-empty
  peek at the top u of S
  if u has an (unvisited)neighbour w,
  mark w and push it onto S
  else pop S
```

Overall Traversal Order: DFS

- In this example, the nodes are traversed from the starting point A in this order:

  A B C D E F G

- The DFS search tends to “dive”.
- The order is not generally unique; e.g. either of B or G could occur first.

Modification of depth first search

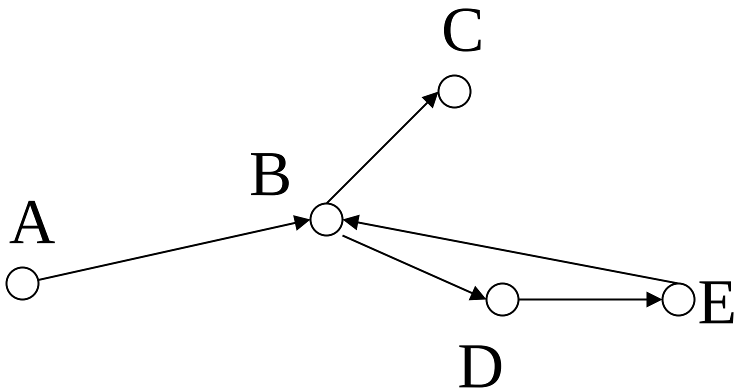
- How to get DFS to detect cycles in a directed graph:

idea: if we encounter a vertex which is already on the stack, we found a loop (stack contains vertices on a path, and if we see the same vertex again, the path must contain a cycle).

- Instead of visited and unvisited, use three colours:
  - **white =** unvisited
  - **grey =** on the stack
  - **black =** finished (we backtracked from it, seen everywhere we can reach from it)

Tracing modified DFS from A

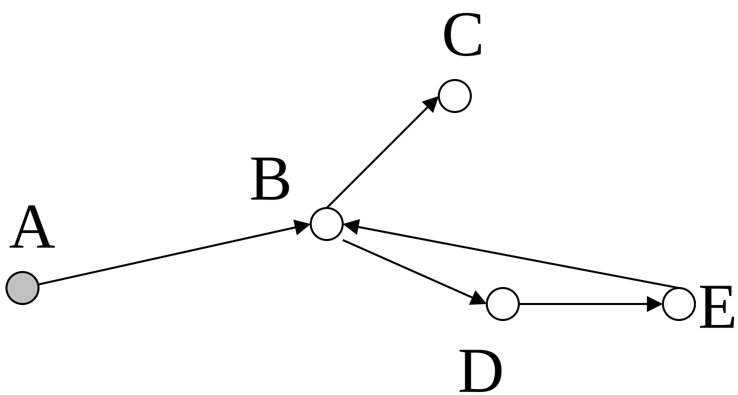
$S = \{\}$



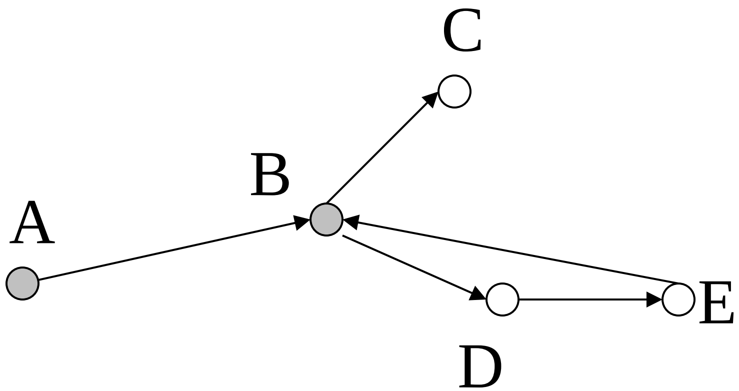
Tracing modified DFS from A

S = {}

S = A



Tracing modified DFS from A

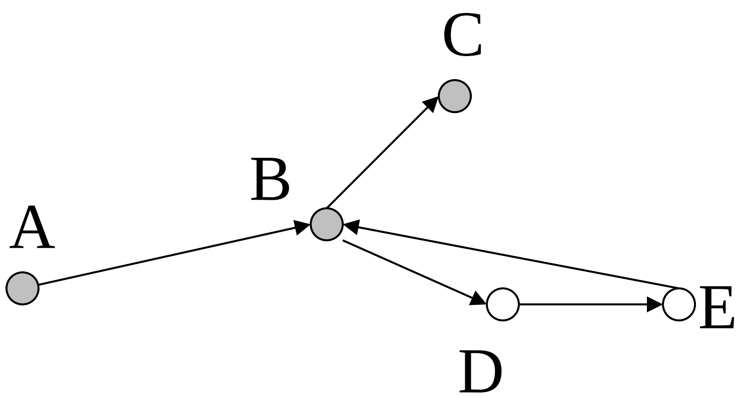


S = {}

S = A

B
S = A

Tracing modified DFS from A



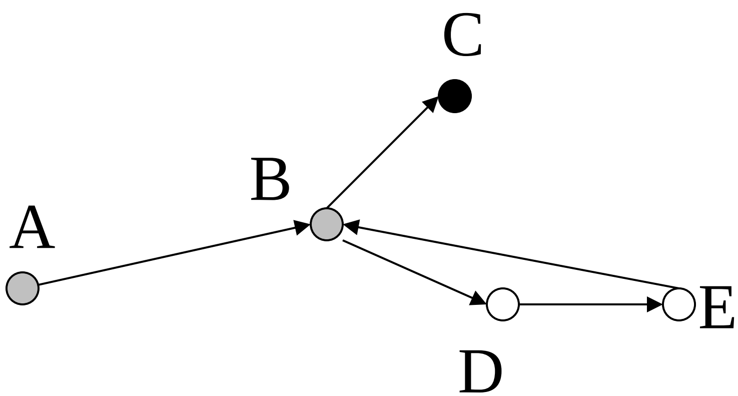
S = {}

S = A

B
S = A

C
B
S = A

Tracing modified DFS from A



S = {}

S = A

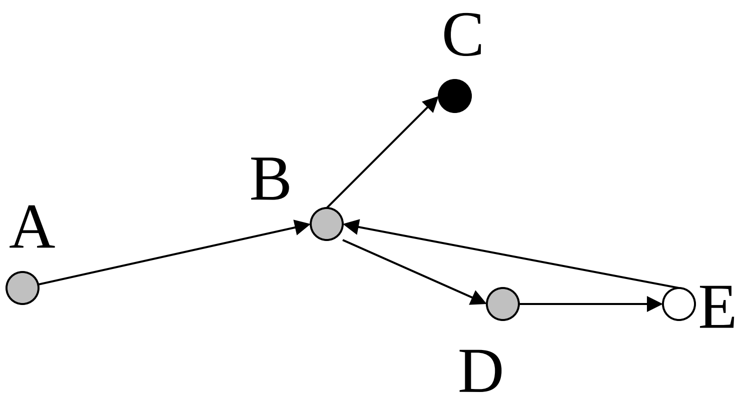
B
S = A

C
B
S = A

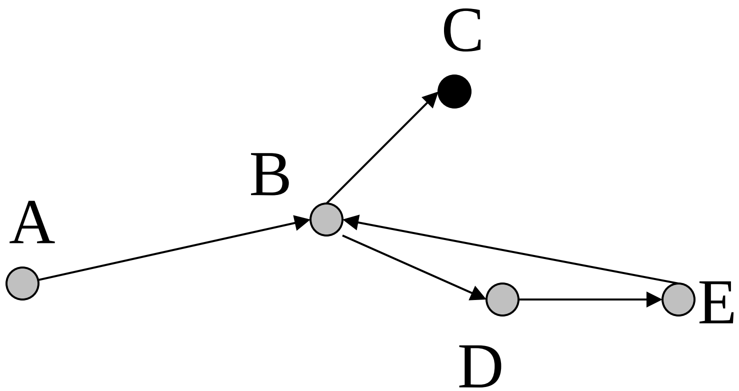
B
pop: S = A

Tracing modified DFS from A

push: D
B
S = A



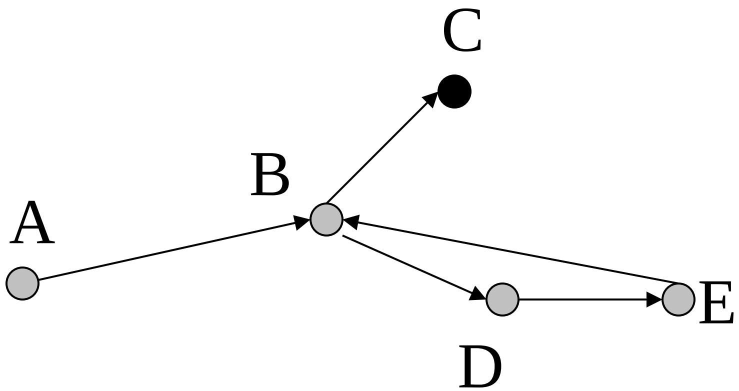
Tracing modified DFS from A



push: D
B
S = A

E
D
B
S = A

Tracing modified DFS from A



push: D
B
S = A

E
D
B
S = A

E has a grey neighbour: B!

Found a loop!

Modification of depth first search

Modified DFS starting from **`v`**:

```
all vertices coloured white
create a stack S
colour v grey and push v onto S
while S is non-empty
  peek at the top u of S
  if u has a grey neighbour, there is a
  cycle
  else if u has a white neighbour w,
  colour w grey and push it onto S
  else colour u black and pop S
```

Pseudocode for BFS and DFS

- To compute complexity, we will be referring to an adjacency list implementation
- Assume that we have a method which returns the first unmarked vertex adjacent to a given one:
  `GraphNode firstUnmarkedAdj(GraphNode v)`

$$v \longrightarrow \overbrace{\text{u1(marked)} \to \text{u2(unmarked)} \to \text{u3(unmarked)}}^{\text{list of v's neighbours}}$$

↑ bookmark

Implementation of firstUnmarkedAdj()

- We keep a pointer into the adjacency list of each vertex so that we do not start to traverse the list of adjacent vertices from the beginning each time.
- Or we use the same iterator for this list, so when we call next() it returns the next element in the list – again does not start from the beginning.

v → u1(marked) → u2(unmarked) → u3(unmarked)

currUnmarkedAdj

Pseudocode for breadth-first search starting from vertex s

```
s.marked = true; // marked is a field in
                 // GraphNode
Queue Q = new Queue();
Q.enqueue(s);
while(! Q.isempty()) {
   v = Q.dequeue();
   u = firstUnmarkedAdj(v);
   while (u != null){
      u.marked = true;
      Q.enqueue(u);
      u = firstUnmarkedAdj(v);}}}
```

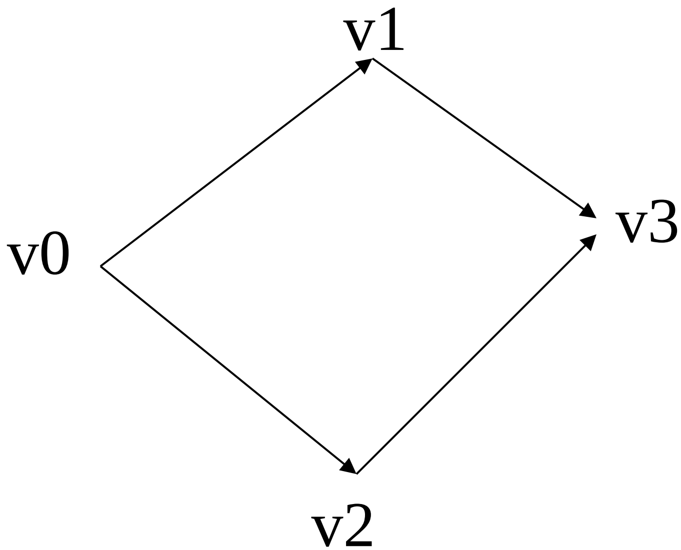
Pseudocode for DFS

```
s.marked = true;
Stack S = new Stack();
S.push(s);
while(! S.isempty()){
    v = S.peek();
    u = firstUnmarkedAdj(v);
    if (u == null) S.pop();
    else {
        u.marked = true;
        S.push(u);
    }
}
```

Time Complexity of BFS and DFS

- In terms of the number of vertices |V|: two nested loops over |V|, hence $O(|V|^2)$.
- More useful complexity estimate is in terms of the number of edges. Usually, the number of edges is less than $|V|^2$.

Time complexity of BFS



Adjacency lists:

V E

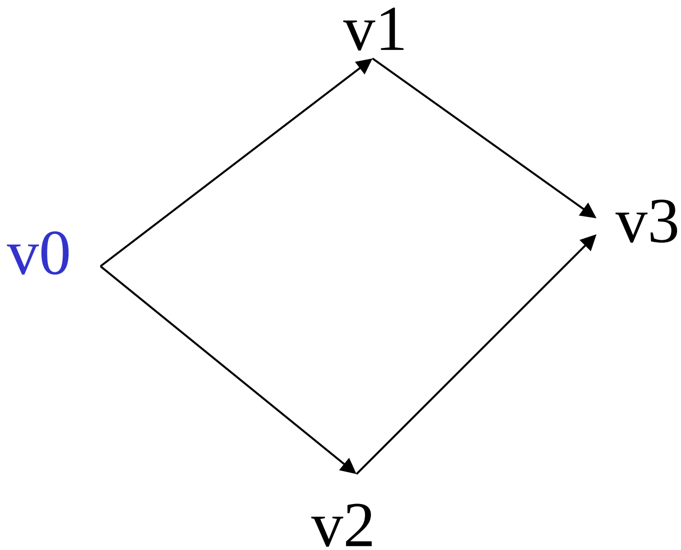
v0: {v1,v2}

v1: {v3}

v2: {v3}

v3: {}

Time complexity of BFS



Adjacency lists:

V E

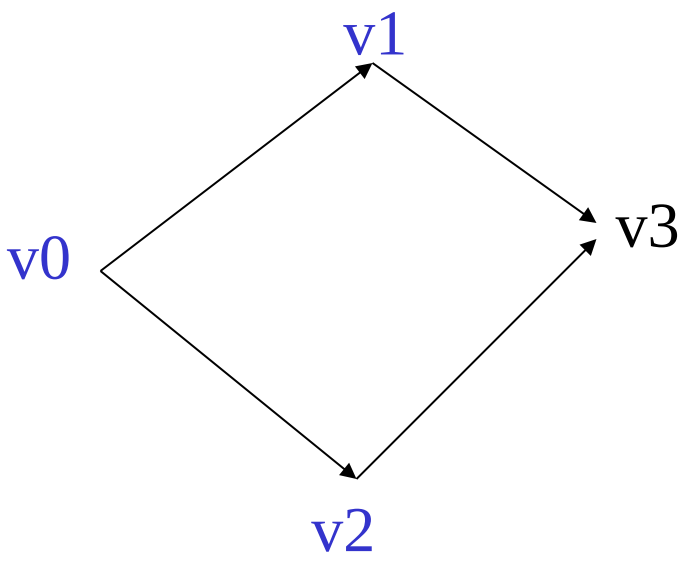
v0: {v1,v2} mark, enqueue v0

v1: {v3}

v2: {v3}

v3: {}

Time complexity of BFS



Adjacency lists:

V E

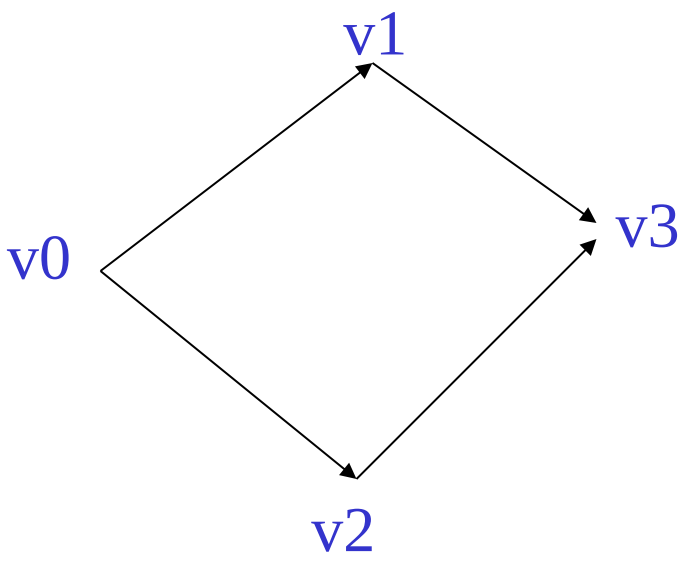
v0: {v1,v2} dequeue v0;
mark, enqueue v1,v2

v1: {v3}

v2: {v3}

v3: {}

Time complexity of BFS



Adjacency lists:

V E

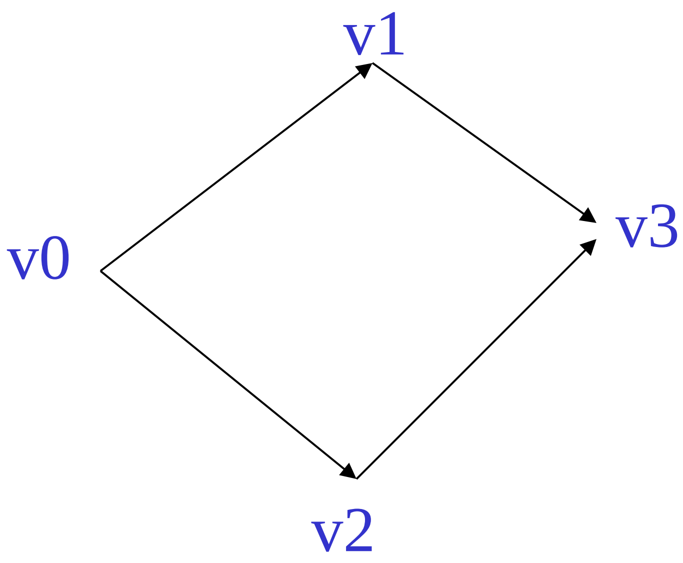
v0: {v1,v2}

v1: {v3} dequeue v1; mark, enqueue v3

v2: {v3}

v3: {}

Time complexity of BFS



Adjacency lists:

V E

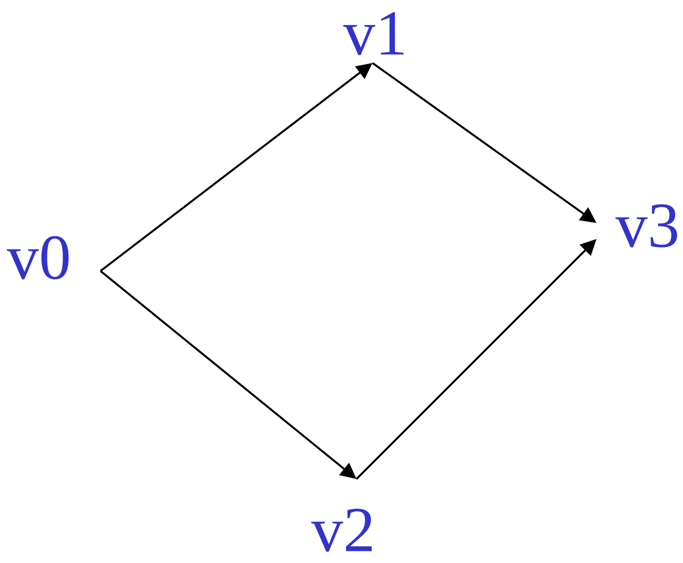
v0: {v1,v2}

v1: {v3}

v2: {v3} dequeue v2, check its adjacency list (v3 already marked)

v3: {}

Time complexity of BFS



Adjacency lists:

V E

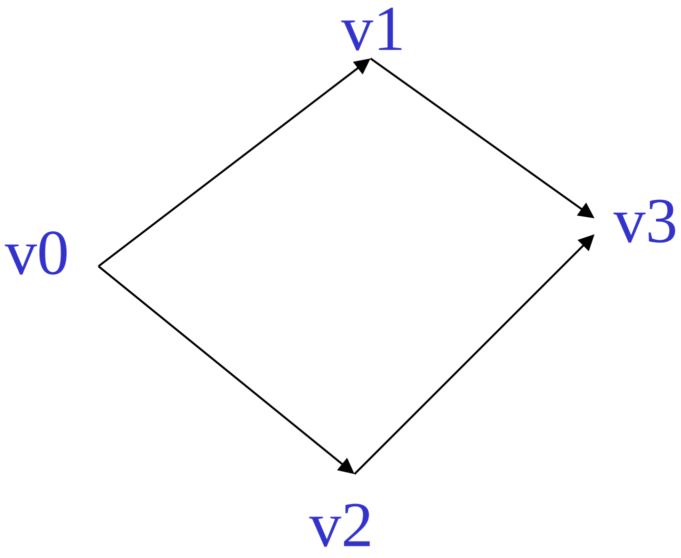
v0: {v1,v2}

v1: {v3}

v2: {v3}

v3: {} dequeue v3; check its adjacency list

Time complexity of BFS



Adjacency lists:

V E

v0: {v1,v2} |E0| = 2

v1: {v3} |E1| = 1

v2: {v3} |E2| = 1

v3: {} |E3| = 0

Total number of steps:
|V| + |E0| + |E1| + |E2| +|E3|
=

= |V|+|E|.

Complexity of breadth-first search

- Assume an adjacency list representation, $|V|$ is the number of vertices, $|E|$ is the number of edges.
- Each vertex is enqueued and dequeued at most once.
- Scanning for all adjacent vertices takes $O(|E|)$ time, since sum of lengths of adjacency lists is $|E|$.
- Gives a $O(|V|+|E|)$ time complexity.

Complexity of depth-first search

- Each vertex is pushed on the stack and popped at most once.
- For every vertex we check what the next unvisited neighbour is.
- In our implementation, we traverse the adjacency list only once. This gives O(|V|+|E|) again.