Graphs

Edited by Heshan Du
University of Nottingham Ningbo China

Learning Objectives

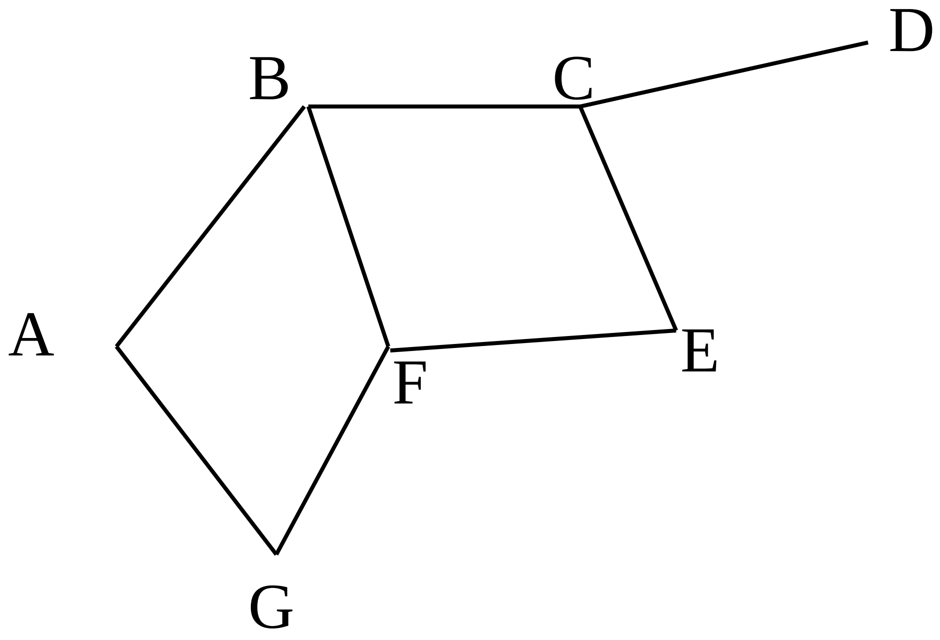
- To be able to *understand* and describe the definition of a graph and its related terminology;
- To be able to *understand* the Graph ADT;
- To be able to *implement* the Graph ADT and analyze the complexity of the methods;
- To be able to *apply* the Graph ADT to solve problems

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

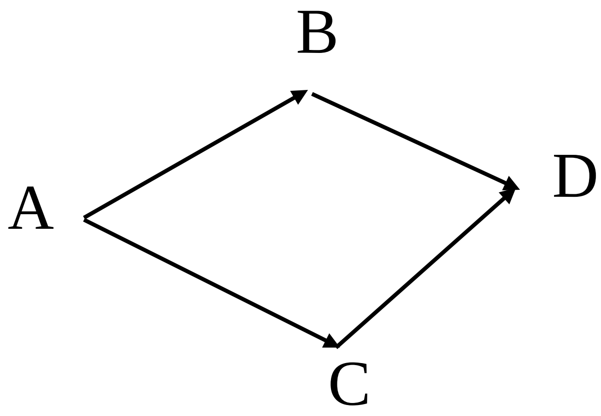
Graphs can be used to represent

- 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)

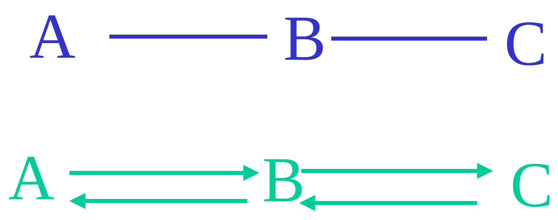


directed graph

Directed and Undirected Graphs

任何无向图都可以被表示为一个等价的有向图。

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



undirected graph

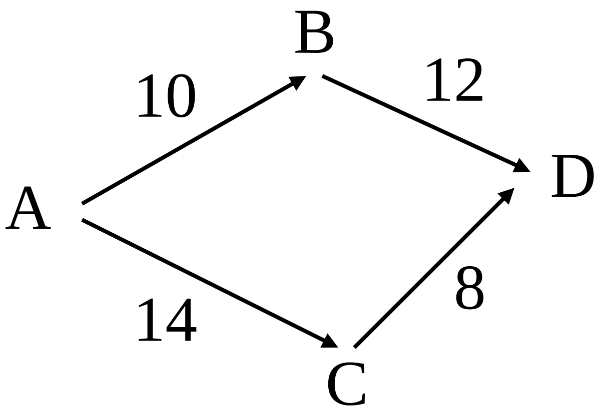
corresponding

directed graph

Weighted and Unweighted Graphs

Graphs can also be

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



weighted graph

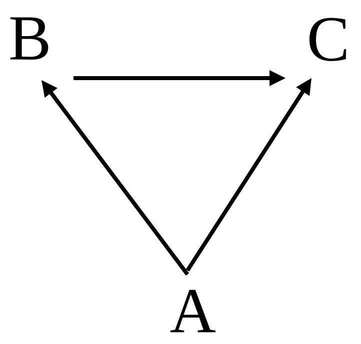
Notation

顶点

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

Example:

- 每条边是两个顶点之间的有序对：
  - 对于**有向图**, $(u, v) \neq (v, u)$
  - 对于**无向图**, $(u, v) \equiv (v, u)$



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.

$$A \longrightarrow B$$

在有向图中：

- **B 是 A 的邻接节点（adjacent to A）**，当且仅当存在一条**从 A 指向 B 的边**：

$(A \to B) \in E$

- 用数学语言描述：

如果 $(A, B) \in E$，则说 **B is adjacent to A**。

Paths and reachability

简单来说：一条路径是图中一系列相连节点的顺序移动，不可跳跃、不逆行。

- 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

如果图中存在从 A 到 B 的路径（无论中间经过多少节点），我们就说：\text{B is reachable from A}

More Terminology

从一个顶点出发，经过一系列边又回到自身的路径。A->B->C->A

- A *cycle* is a path from a vertex to itself

图中不包含任何环的图。A->B->C

- Graph is *acyclic* if it does not have cycles
- Graph is *connected* if there is a path between every pair of vertices 对于任意两个节点 u, v，都存在一条路径连接它们。
- Graph is *strongly connected* if there is a path in both directions between every pair of vertices

- **定义（有向图）**：

对于任意两个顶点 u 和 v，都满足：

  - 有路径 $u \to v$
  - 有路径 $v \to u$

- 所有顶点互相“强可达”。
- 举例：

A ↔ B, B ↔ C, C ↔ A ⇒ Strongly connected

性质	应用于	定义
Cycle	有向或无向图	路径首尾相连
Acyclic	有向/无向	不存在任何环
Connected	**无向图**	任意两点间有路径连接
Strongly Connected	**有向图**	任意两点间**双向**均可达

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 ...

场景	节点代表	边代表	应用算法
棋盘游戏	游戏状态	一步合法移动	状态空间搜索（DFS/BFS, A*）
网络通信	设备	带宽、延迟	最短路径、最大流、最小生成树
城镇交通	城市/车站	距离/时间/价格	Dijkstra, Floyd, A* 路径规划等

Graph ADT

方法名	功能说明
numVertices()	返回图中顶点（节点）的数量。
vertices()	返回图中所有顶点的一个可迭代对象（iterator）。
numEdges()	返回图中边的总数。
edges()	返回图中所有边的可迭代对象。
getEdge(u, v)	如果存在从 u 到 v 的边，返回该边；否则返回 null。对于无向图，getEdge(u, v) 与 getEdge(v, u) 等价。
endVertices(e)	返回一个数组（或元组），包含边 e 的两个端点。如果是有向图，返回顺序为：起点、终点。
opposite(v, e)	返回边 e 上除了 v 以外的另一个顶点。如果 e 不是 v 的邻接边，则抛出错误。

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

方法名	功能说明
outDegree(v)	返回从顶点 v 发出的边的数量（即**出度**）。
inDegree(v)	返回指向顶点 v 的边的数量（即**入度**）。对于无向图，inDegree(v) 与 outDegree(v) 值相同。
outgoingEdges(v)	返回一个可迭代对象，包含所有从顶点 v 发出的边。
incomingEdges(v)	返回一个可迭代对象，包含所有指向顶点 v 的边。对于无向图，incomingEdges(v) 与 outgoingEdges(v) 返回相同内容。

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

方法名	功能说明
insertVertex(x)	创建并返回一个新的顶点，内部存储元素 x。
insertEdge(u, v, x)	创建并返回一个新的边，从顶点 u 到顶点 v，该边存储元素 x。若从 u 到 v 已存在一条边，则抛出错误。
removeVertex(v)	从图中删除顶点 v，并删除所有与该顶点相连的边（即 incident edges）。
removeEdge(e)	从图中删除边 e。

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

问题	简要说明	常用算法
1. 搜索一个顶点（vertex）	给定一个条件（如名称或属性），在图中查找满足条件的顶点。	DFS、BFS、线性扫描
2. 搜索一条边（edge）	给定两个顶点，查找它们之间是否存在边（有向/无向）。	邻接表/矩阵直接查找
3. 查找一条路径（from u to v）	判断是否存在一条从 u 到 v 的路径（可用于连通性检测）。	DFS、BFS
4. 查找最短路径	找出 u 到 v 所有可能路径中**最短的一条**。	Dijkstra（无负权）、Bellman-Ford（负权）、Floyd-Warshall（多源）
5. 环检测（cycle detection）	判断图中是否存在环。	DFS + visited/recursion stack、Union-Find（无向图）

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)

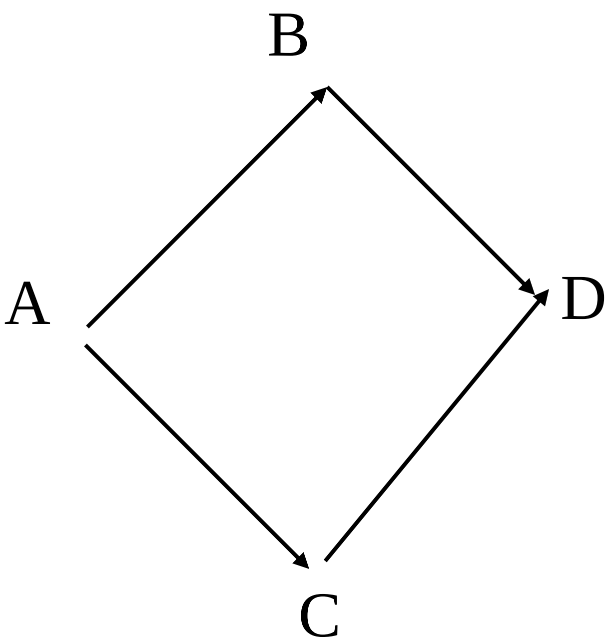
- Represent information about the edges using a two dimensional array, where

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

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

使用二维数组 array[i][j] 表示从节点 i 到 j 的边。
若 array[i][j] == 1，表示存在一条从 i 指向 j 的**有向边**。
对于**无向图**，还需满足 array[i][j] == array[j][i] == 1。

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`**)

对于加权图，邻接矩阵不仅记录是否有边，而是直接记录**边的权重**。
如果从顶点 i 到顶点 j 有一条边，其权重为 w，那么 array[i][j] = w
如果没有边连接 i 和 j，则使用一个特殊的值标记（不能与任何权重混淆），例如：
-1
Integer.MAX_VALUE（在 Java 中常用于表示 “ 无穷大 ” ）

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).

1. 空间浪费（Space Inefficiency） **稀疏图（Sparse Graph）**中，大部分位置是 0（表示没有边）。但我们仍要为每个节点对分配空间，这在边数远小于节点平方数 O(n^2) 的情况下会**严重浪费内存**。例如，对于 1000 个节点但只有 10 条边，矩阵是 1000 times 1000 的，而非空单元只有 10 个。
2. 遍历效率低若想找出某节点的所有邻接节点，必须遍历整个矩阵的行（或列），即 O(n)，哪怕只有少数几个 1。
3. 缺乏灵活性（Inflexible）节点数固定。如果将来要添加更多节点：必须重建一个更大的二维数组。在图大小未知或动态变化时，**邻接矩阵扩展困难、不灵活**。

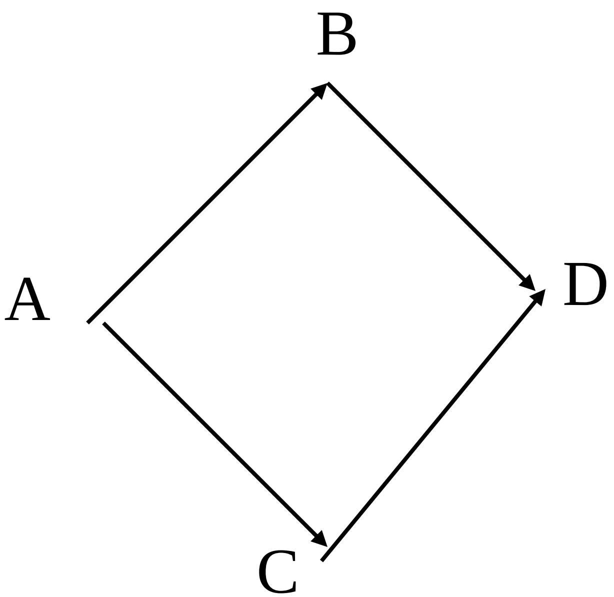
Adjacency List

- 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.

核心思想：
对于每一个顶点 v，**维护一个邻接顶点的列表**（即从 v 出发的所有边的终点）。
具体实现方式：使用一个列表（如 ArrayList 或 LinkedList）保存所有顶点。
或使用一个映射结构（如 HashMap<Vertex, List<Vertex>>）：
键（Key）：图中的一个顶点
值（Value）：从该顶点出发的所有相邻顶点（即邻接顶点）组成的列表。

优点	说明
节省空间	空间复杂度是 $O(V+E)$，适合边远少于节点对的稀疏图
更快地查找邻接点	查找一个顶点的邻接点时间为 $O(\mathrm{degree}(v))$，不会遍历整行
易于动态更新	插入/删除顶点或边时，比邻接矩阵更灵活

Adjacency list



nodes   list of adjacent nodes

A $\longrightarrow$ B, C

B $\longrightarrow$ D

C $\longrightarrow$ D

D $\longrightarrow$

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.

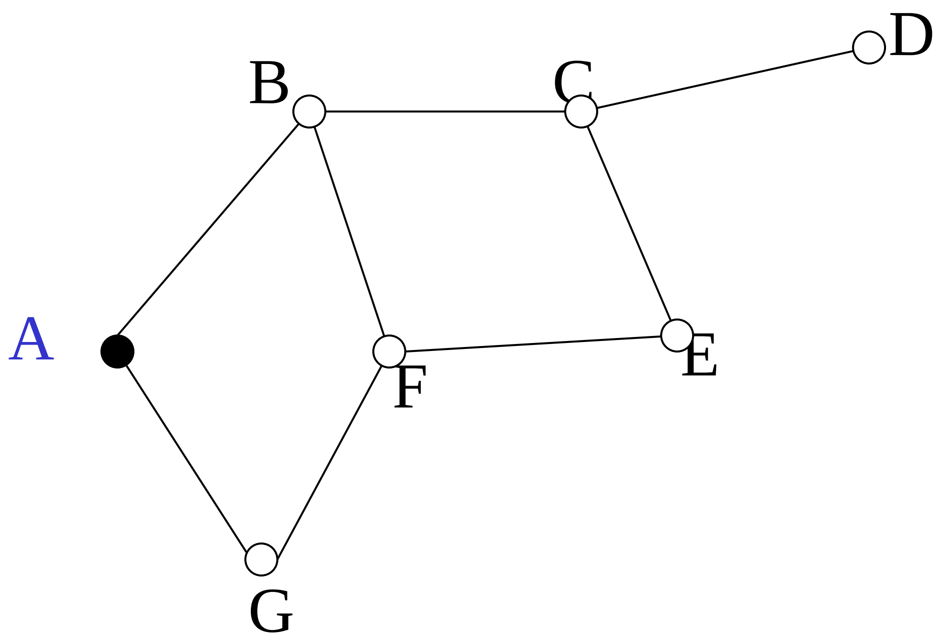
Graph traversals

- 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.

1. Breadth-First Search（BFS）**广度优先遍历**使用**队列**（Queue）从起始点开始，先访问所有邻接点，再逐层展开适合用于找**最短路径（无权图）**
2. Depth-First Search（DFS）**深度优先遍历**使用**栈（Stack）**或递归尽可能向前深入子节点，遇到死路再回溯适合用于找**路径、检测环、连通分量**

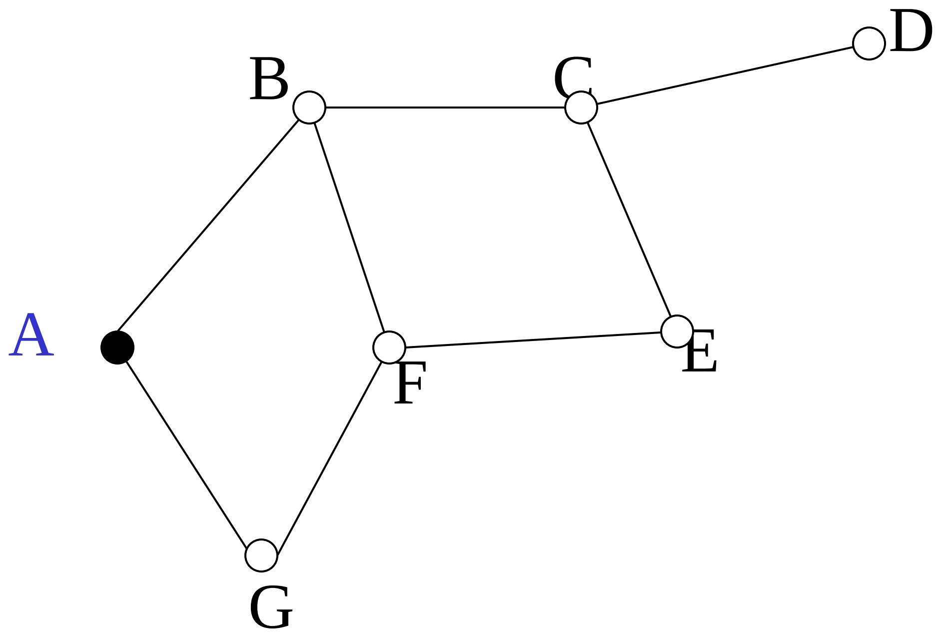
Graph traversal starting from A:

- *Exercise: What might we do?*

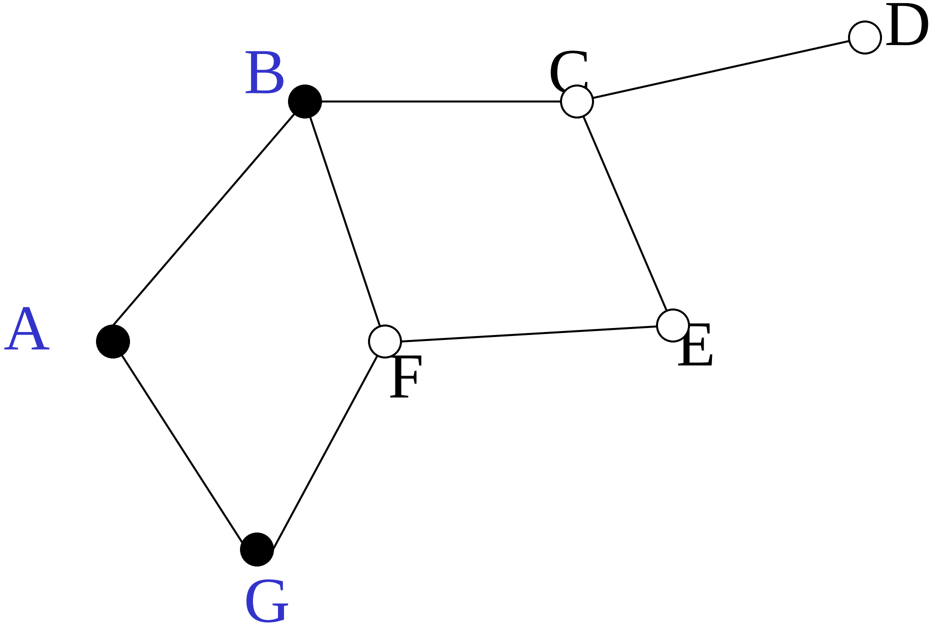


BFS starting from A:

Q={A}



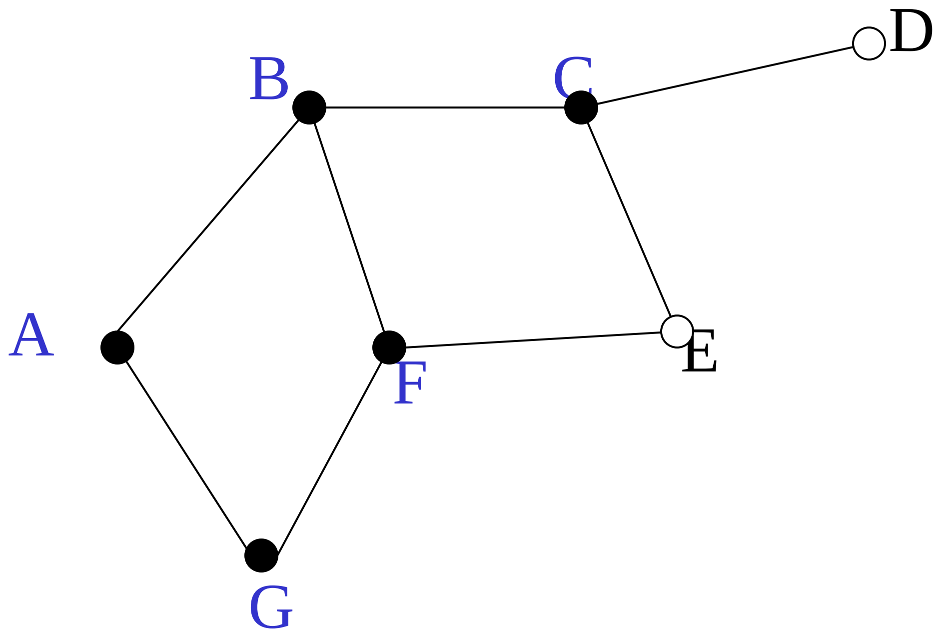
BFS starting from A:



Q={A}

Q={B,G}

BFS starting from A:

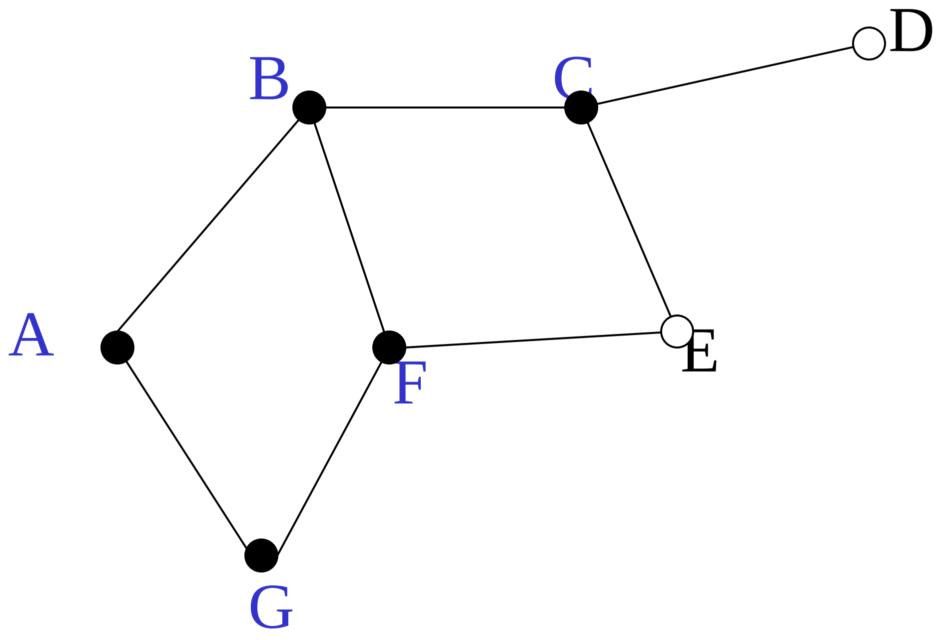


Q={A}

Q={B,G}

Q={G,C,F}

BFS starting from A:



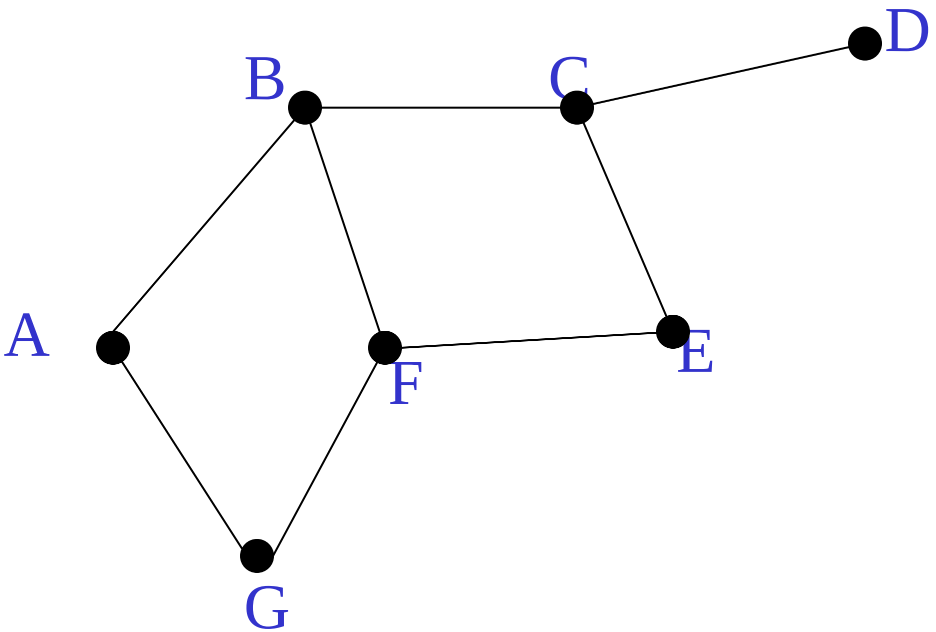
Q={A}

Q={B,G}

Q={G,C,F}

Q={C,F}

BFS starting from A:



Q={A}

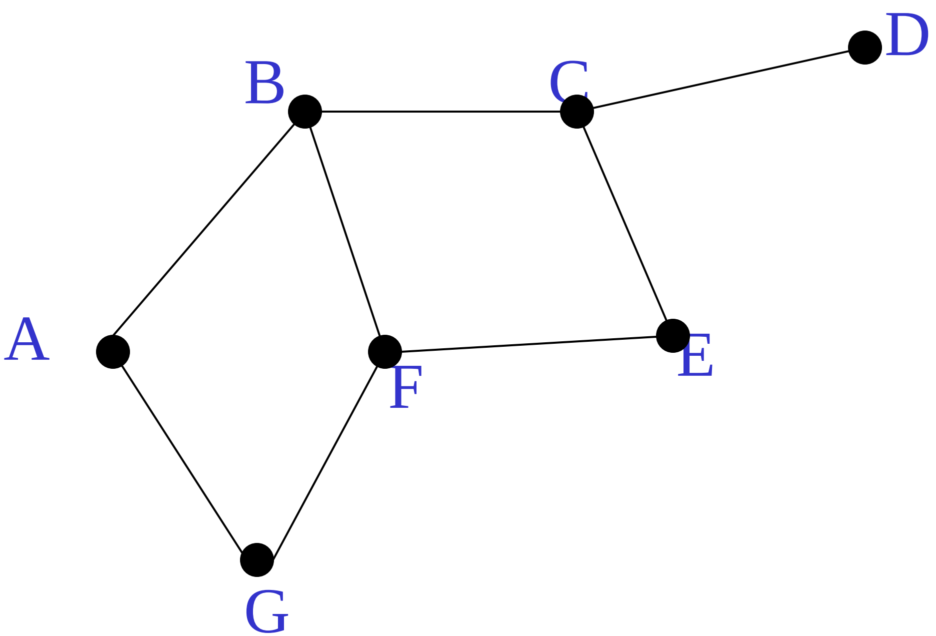
Q={B,G}

Q={G,C,F}

Q={C,F}

Q={F,D,E}

BFS starting from A:



Q={A}

Q={B,G}

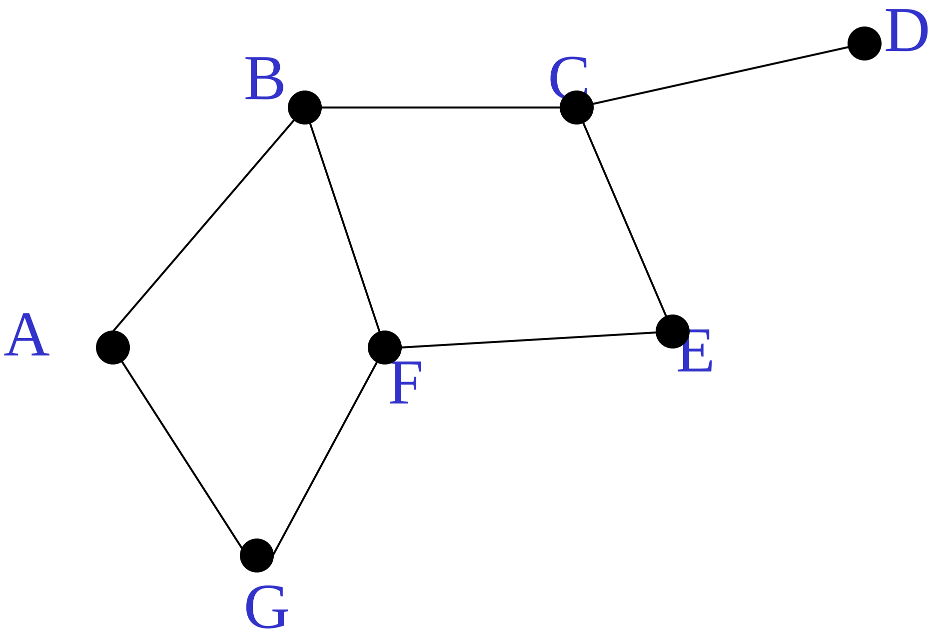
Q={G,C,F}

Q={C,F}

Q={F,D,E}

Q={D,E}

BFS starting from A:



Q={A}

Q={B,G}

Q={G,C,F}

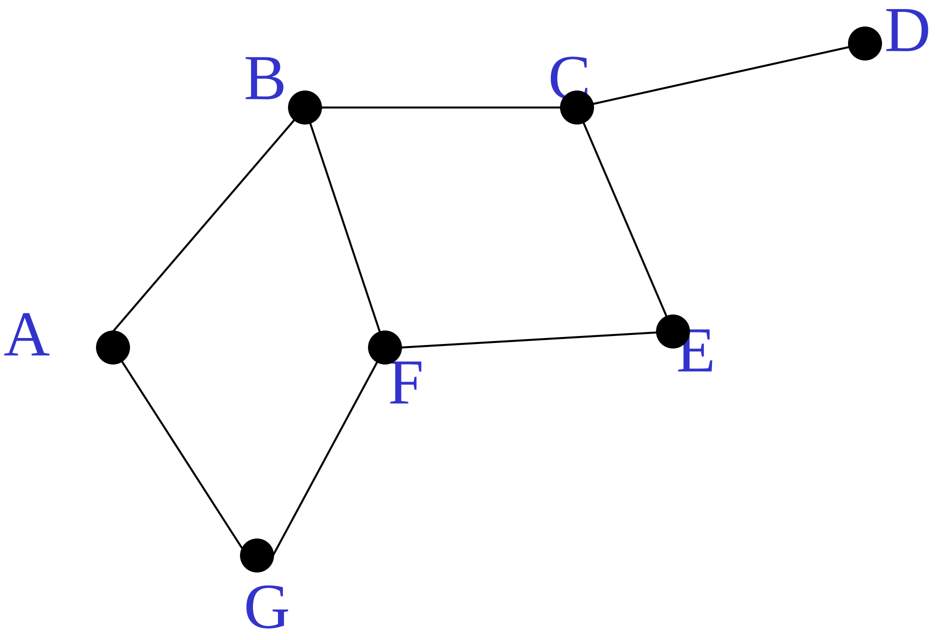
Q={C,F}

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
```

BFS 是按**层级**遍历的：第一层：A第二层：A 的邻居 B, G第三层：B, G 的邻居 C, F第四层：C, F 的邻居 D, E顺序 **不是唯一的**：因为 B 和 G 都是 A 的邻居，它们谁先加入队列会影响后续遍历。所以 A B G C F D E 和 A G B F C D E 都是合法的 BFS 顺序。

Overall Traversal Order: BFS

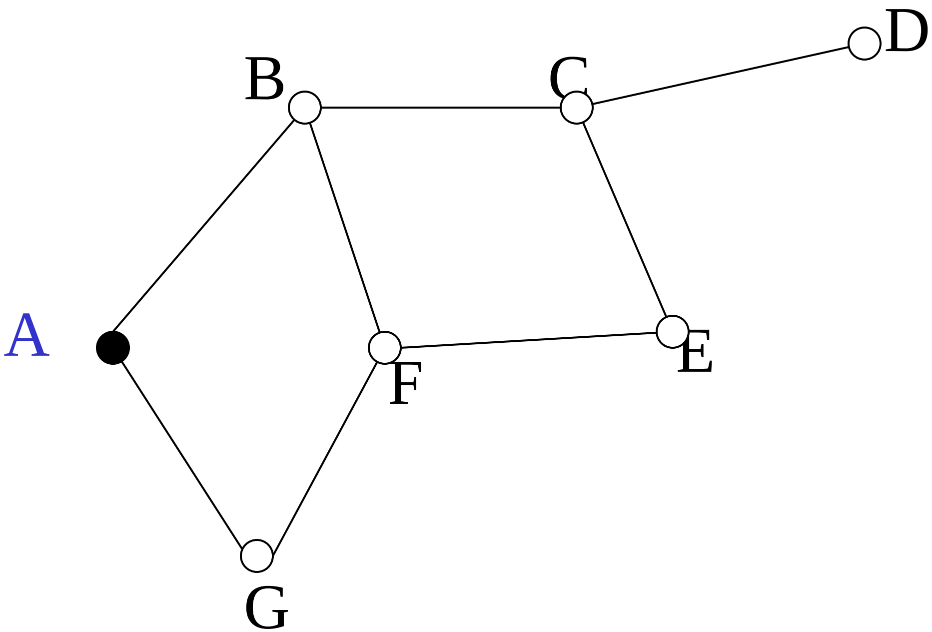
- In this example, the nodes are traversed from the starting point A in this order:

  A B G C F D E

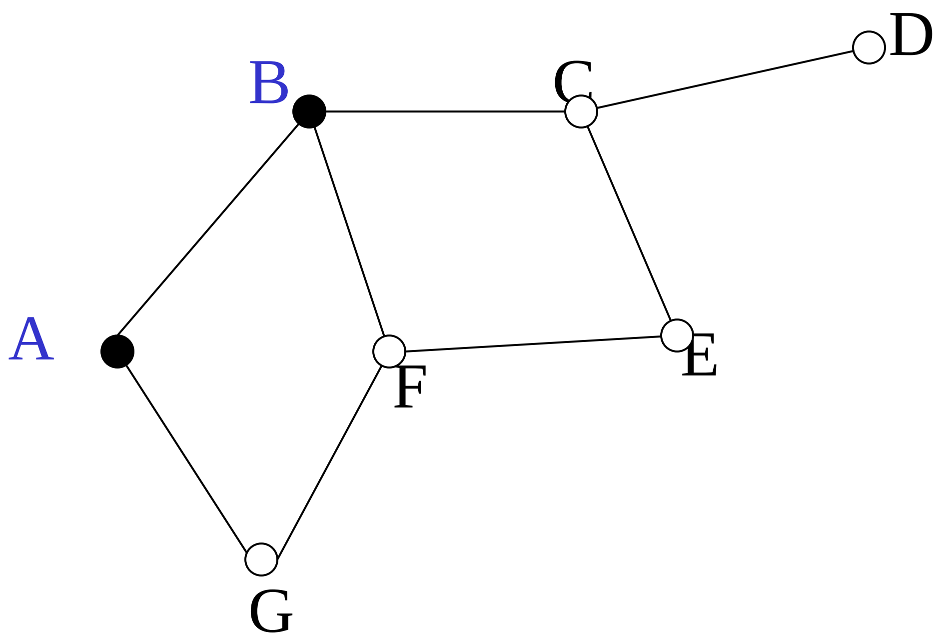
- The BFS order is that those closest to the start point A occur earliest
- The order is not generally unique; e.g. either of B or G could occur first

DFS starting from A:

S={A}



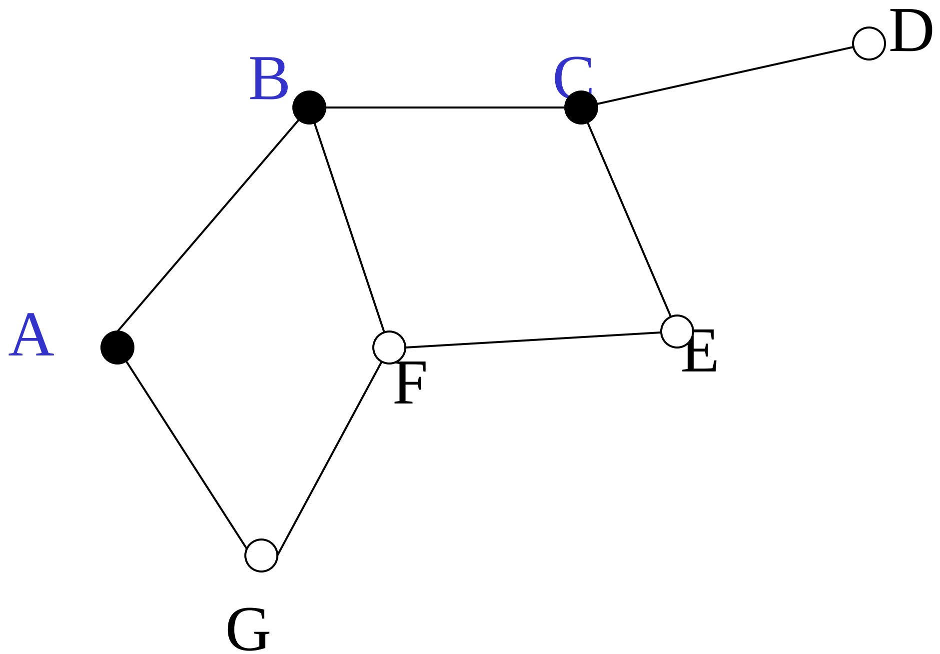
DFS starting from A:



S={A}

S={A,B}

DFS starting from A:

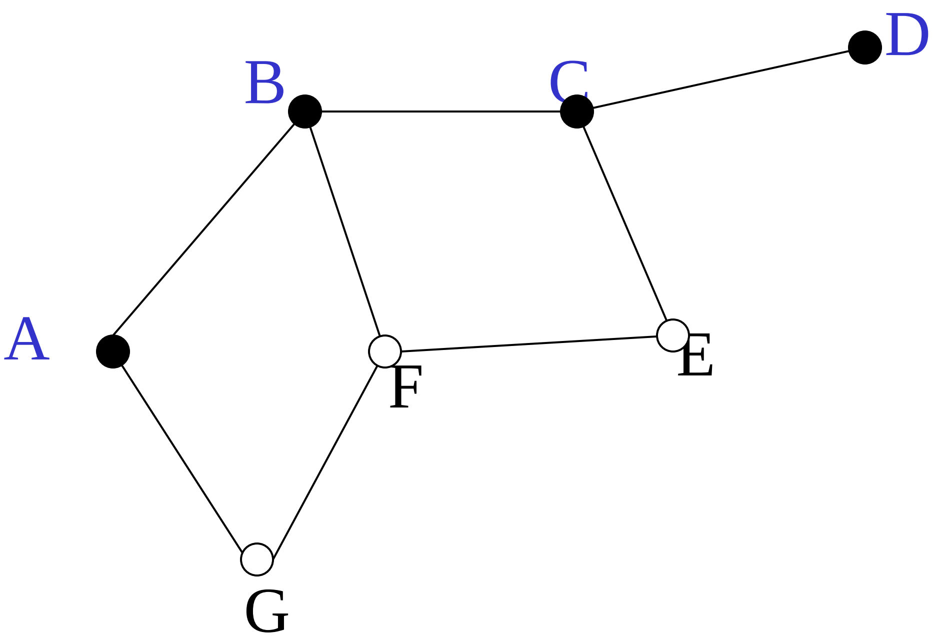


S={A}

S={A,B}

S={A,B,C}

DFS starting from A:



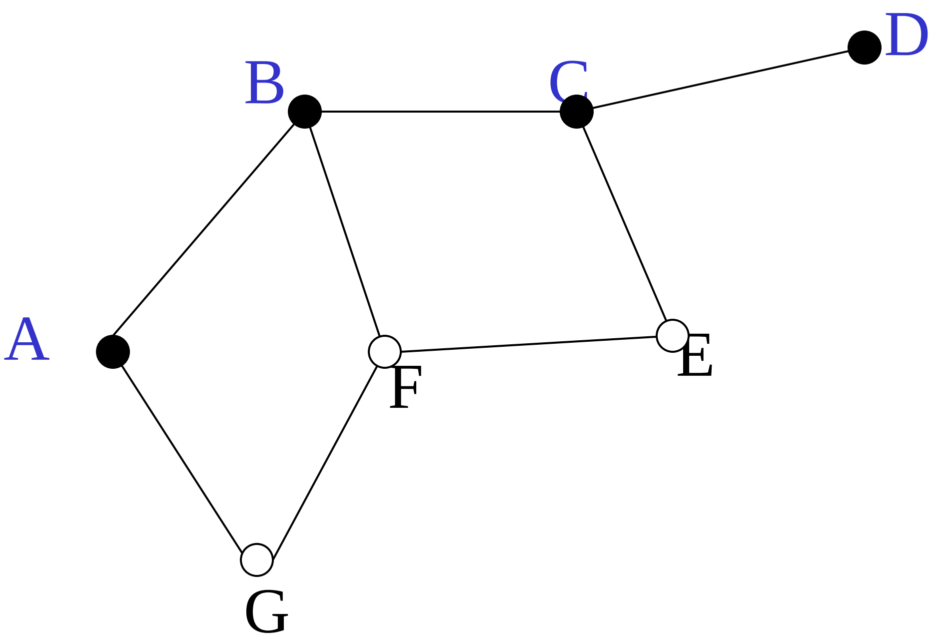
S={A}

S={A,B}

S={A,B,C}

S={A,B,C,D}

DFS starting from A:



S={A}

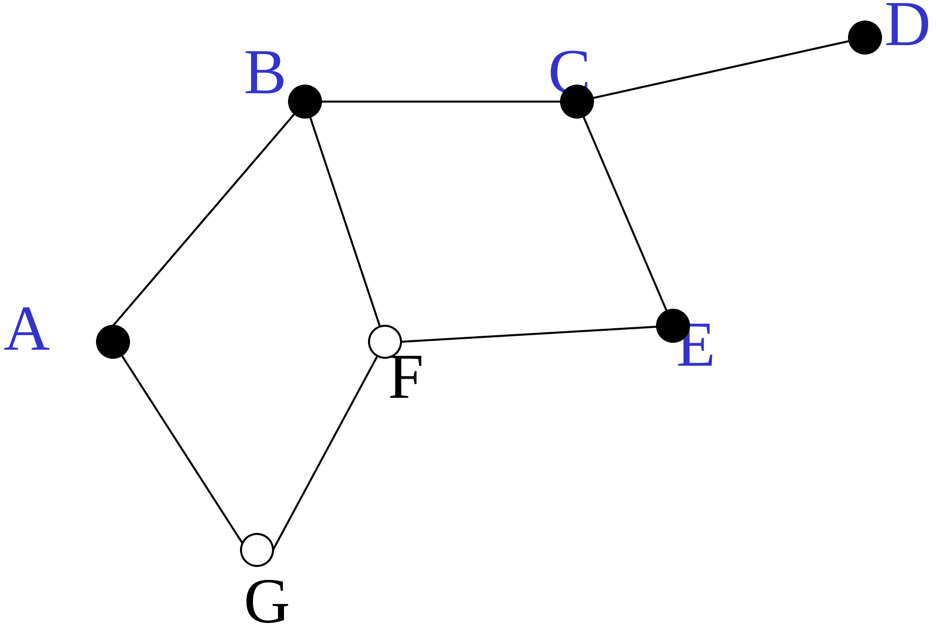
S={A,B}

S={A,B,C}

S={A,B,C,D}

S={A,B,C}

DFS starting from A:



S={A}

S={A,B}

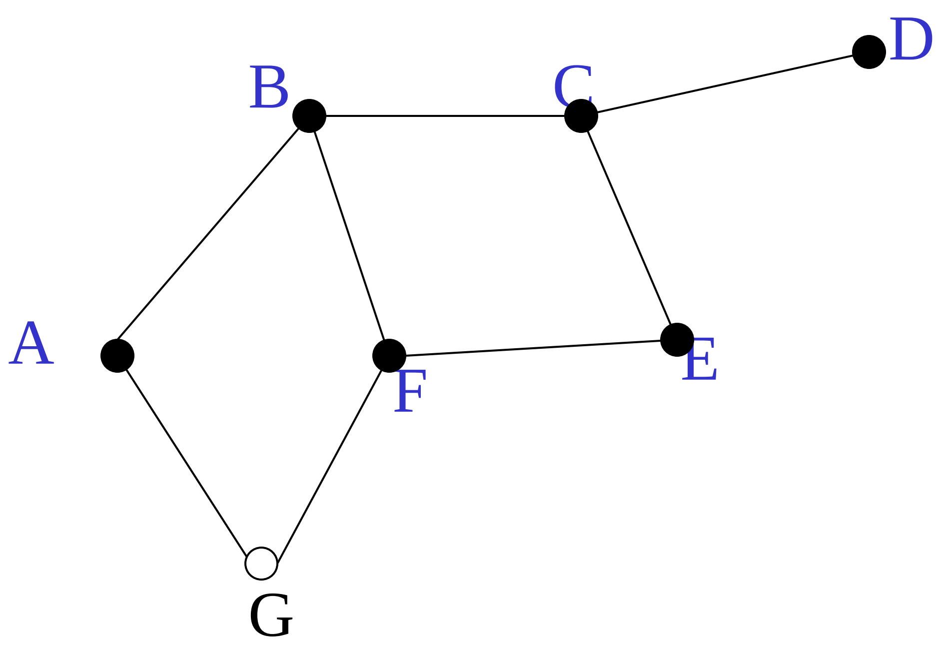
S={A,B,C}

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}

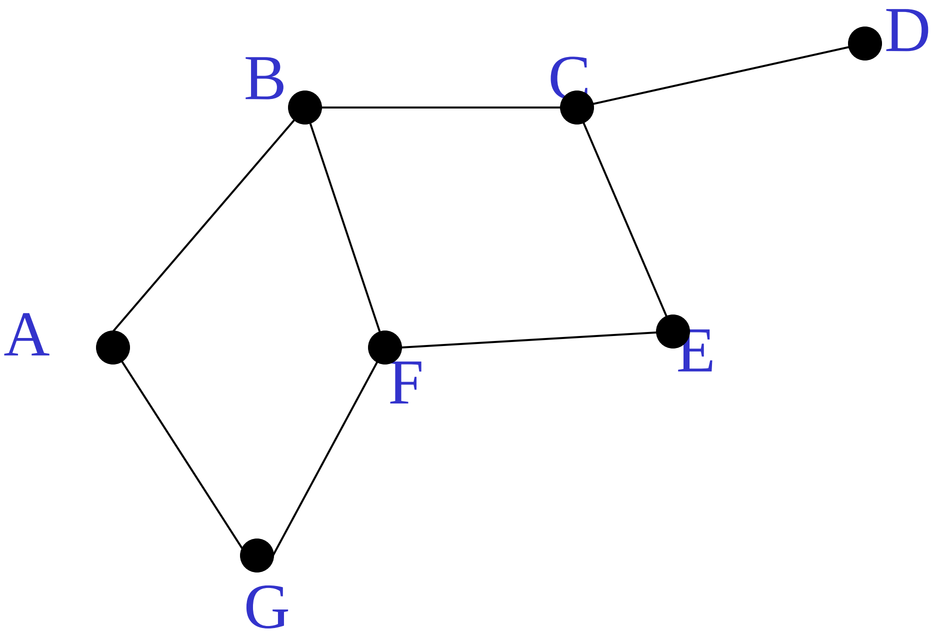
S={A,B,C,D}

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}

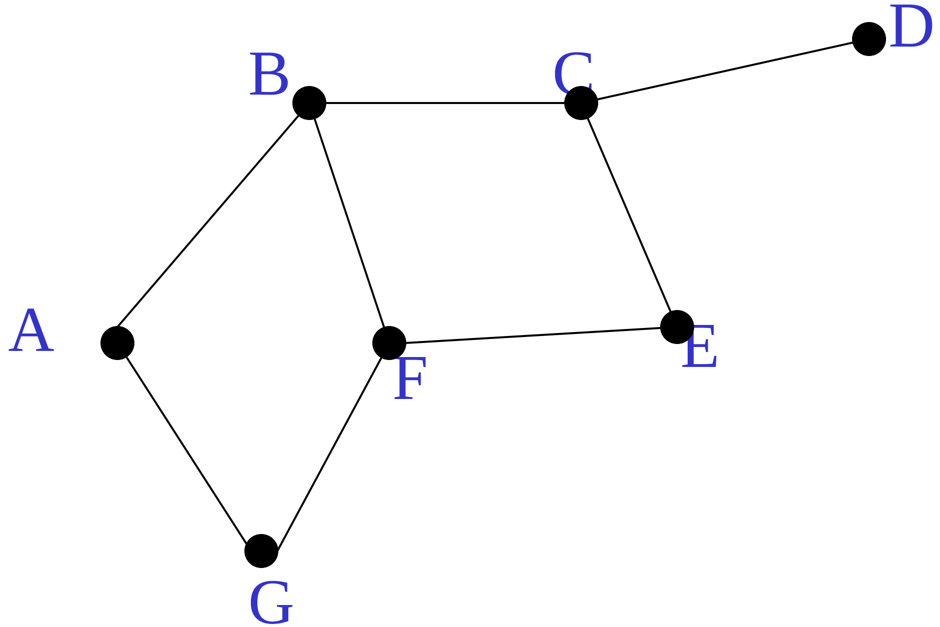
S={A,B,C,E}

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

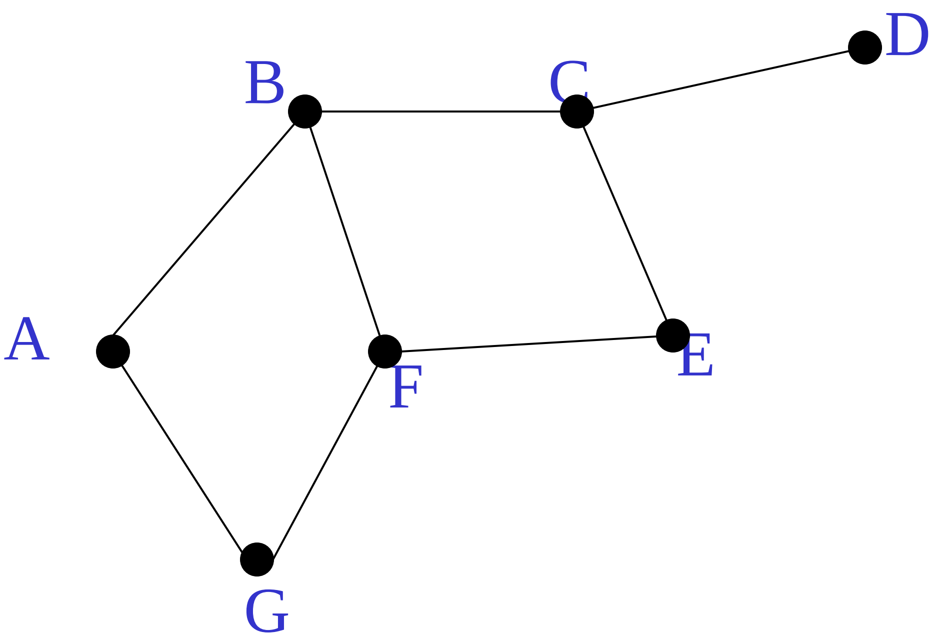
S={A,B,C,E,F,G}

DFS starting from A:

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



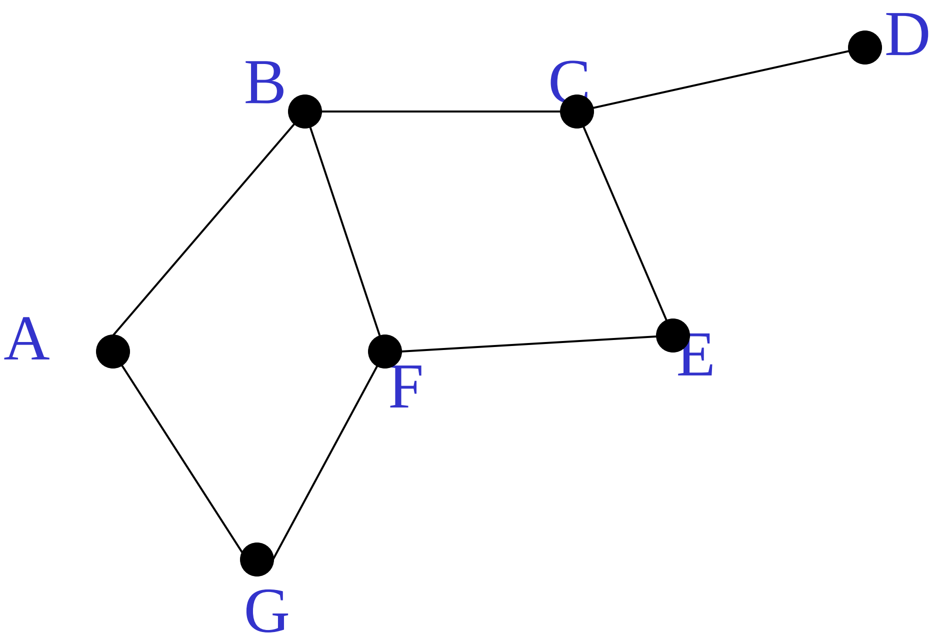
DFS starting from A:



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

S={A,B,C,E}

DFS starting from A:

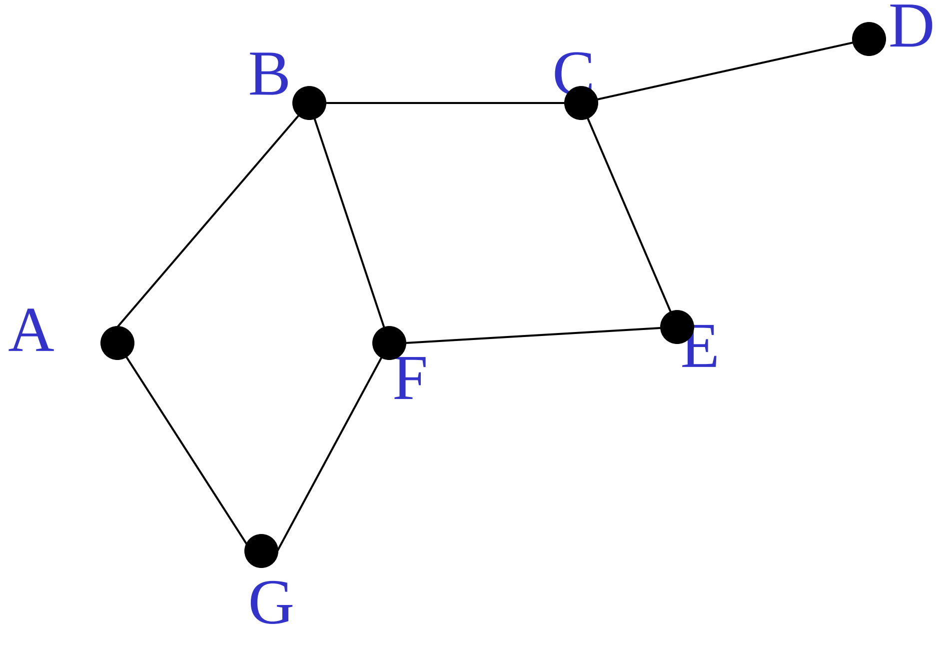


S={A,B,C,E,F}

S={A,B,C,E}

S={A,B,C}

DFS starting from A:



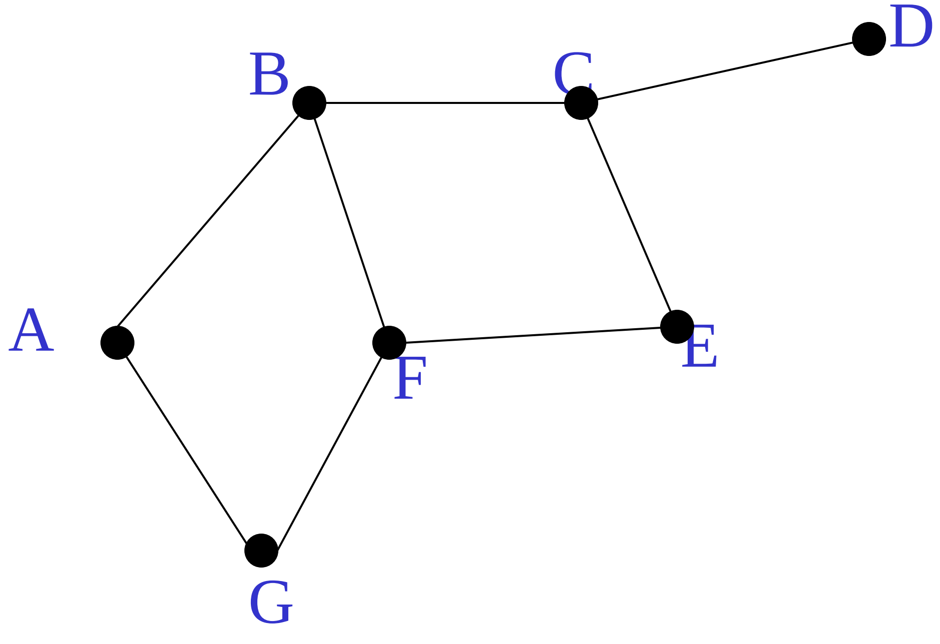
S={A,B,C,E,F}

S={A,B,C,E}

S={A,B,C}

S={A,B}

DFS starting from A:



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

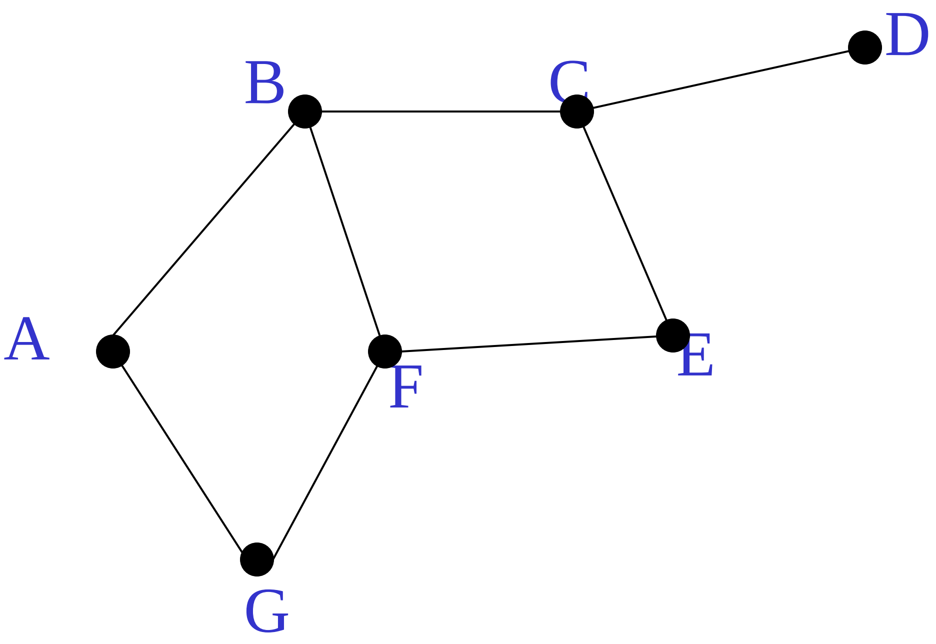
S={A,B,C,E}

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.

DFS 会**尽可能深入图的一条路径**（就像潜水 “ dive ” 一样），直到走不通为止才回退；遍历顺序不是唯一的，和**邻接节点的排列顺序**有关：比如，如果 A 的相邻节点是 {G, B}，而不是 {B, G}，那么遍历顺序可能就变成了 A G F E C D B。

如果在 DFS 过程中遇到一个已经在当前递归栈中的节点（即“灰色”节点），说明存在环。因为递归栈中的节点表示当前路径上的访问状态，如果又遇到了自己，说明路径“回头”，形成了**闭环**。

Modification of depth first search

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

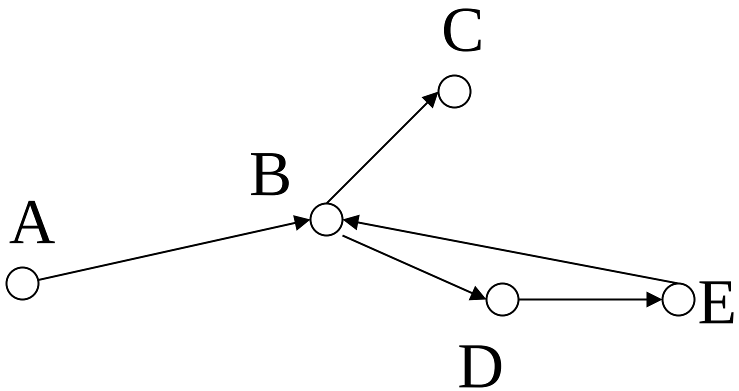
idea: if we encounter a vertex which is already on the stack, we found a loop (stack contains vertices on a path, and if we see the same vertex again, the path must contain a cycle).

- 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)

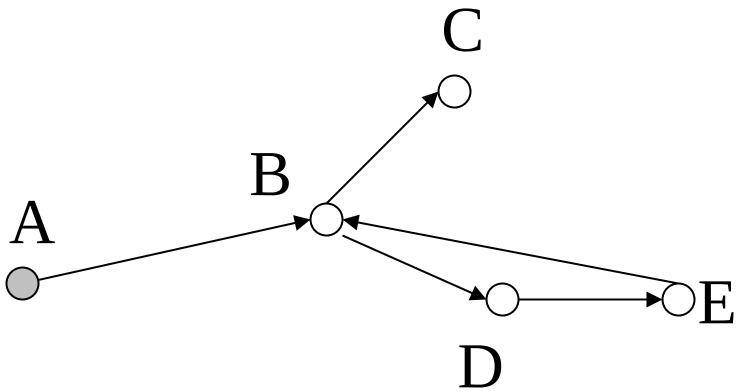
颜色	含义
white	尚未访问（unvisited）
grey	当前正在访问（在 DFS 调用栈上）
black	已完成访问（该节点及其所有可达节点都已访问）

Tracing modified DFS from A

S = {}



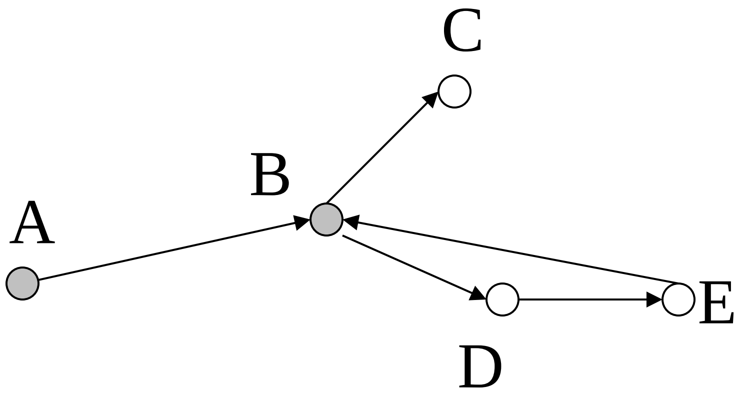
Tracing modified DFS from A



S = {}

S = A

Tracing modified DFS from A



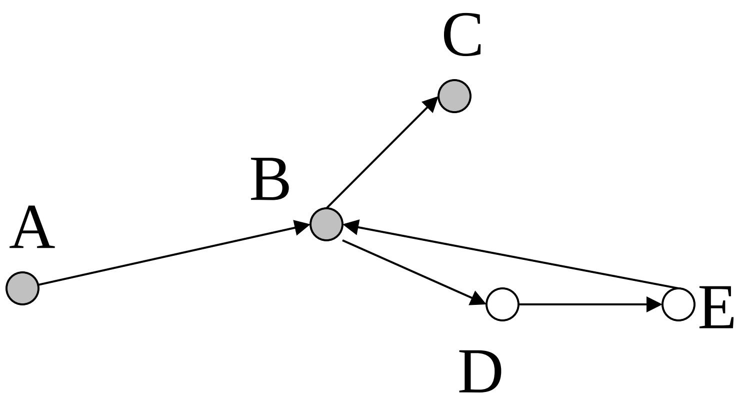
S = {}

S = A

B

S = A

Tracing modified DFS from A



S = {}

S = A

B

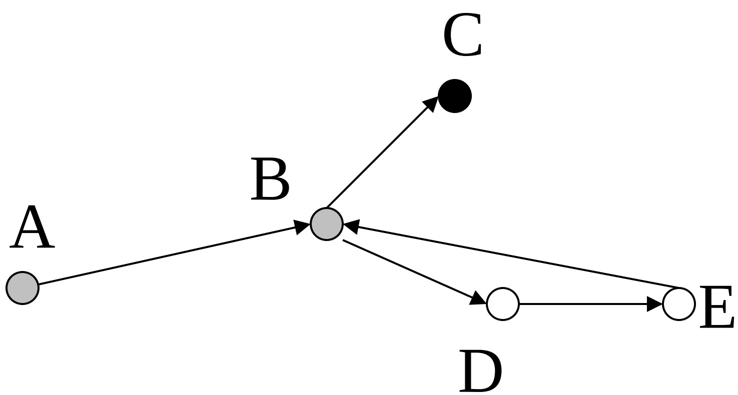
S = A

C

B

S = A

Tracing modified DFS from A



S = {}

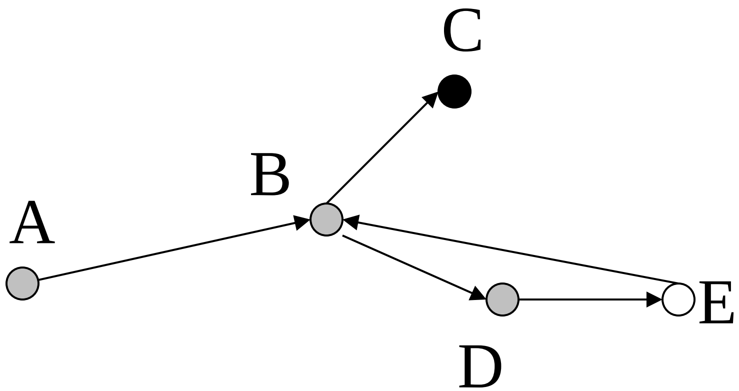
S = A

B
S = A

C
B
S = A

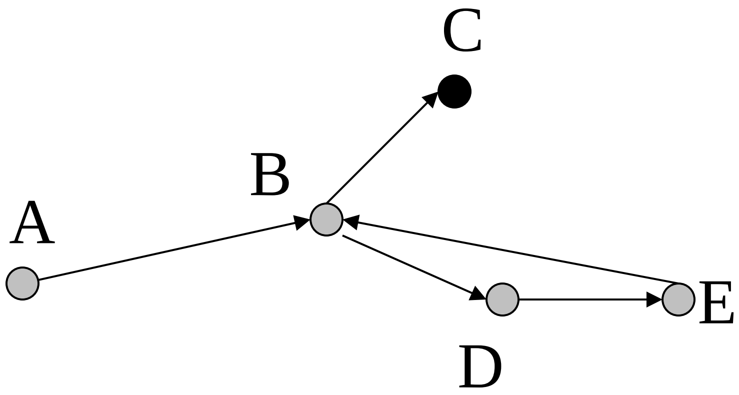
B
pop: S = A

Tracing modified DFS from A



push: D
B
S = A

Tracing modified DFS from A

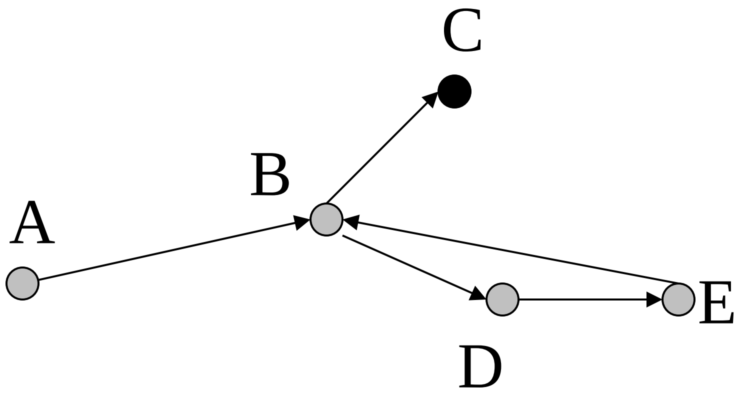


push:

D
B
S = A

E
D
B
S = A

Tracing modified DFS from A



push:

D
B
S = A

E
D
B
S = A

初始化所有节点为 white；
每次访问一个节点，将其设为 grey；
在 DFS 的过程中，如果遇到一个 **grey** 节点（即仍在栈中），
就说明存在 **环**；
完成当前节点的所有 DFS 后，设为 black。

E has a grey neighbour: B!
Found a loop!

颜色状态	意义
white	未访问
grey	正在访问（还未回溯）
black	已访问完（该节点及其可达路径全部处理完）

> ✅ 判断环的依据是：**访问到“灰色”节点。**

Modification of depth first search

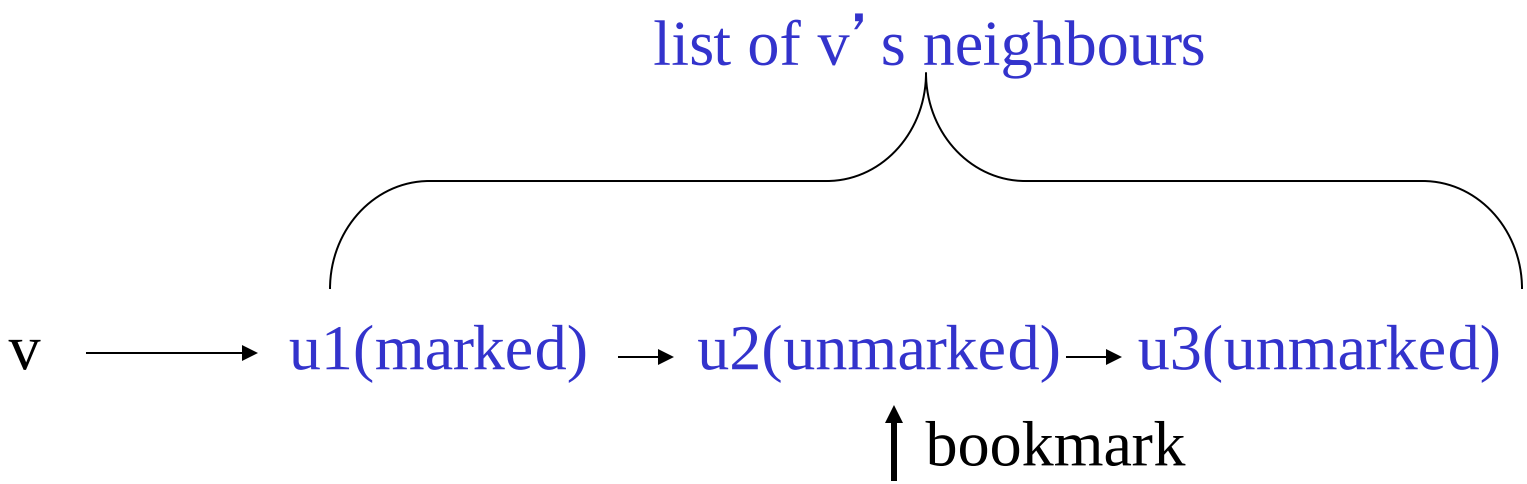
Modified DFS starting from **`v`**:

```
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
```

返回与节点 v**相邻的第一个未标记的顶点**（first unmarked adjacent node）。
实际上是遍历 v 的邻接表 v.neighbors，返回**第一个还没访问的邻居节点**。

Pseudocode for BFS and DFS

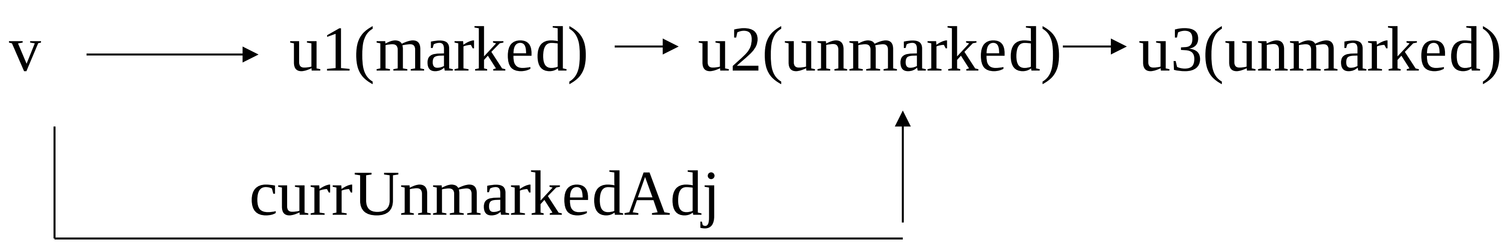
- To compute complexity, we will be referring to an adjacency list implementation
- Assume that we have a method which returns the first unmarked vertex adjacent to a given one:
  `GraphNode firstUnmarkedAdj(GraphNode v)`



在图的遍历中，每次调用 firstUnmarkedAdj(v) 不应该**总是从头开始扫描**v 的邻接节点。图中说明了两个优化方法：

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.



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);}}}
```

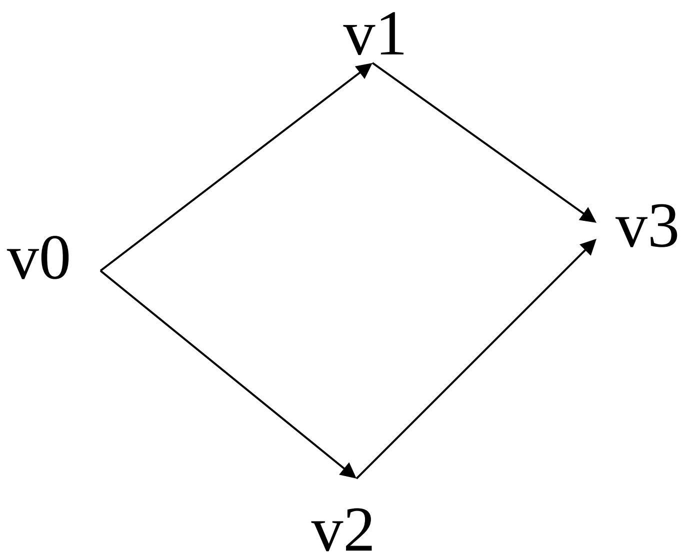
Pseudocode for DFS

```
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

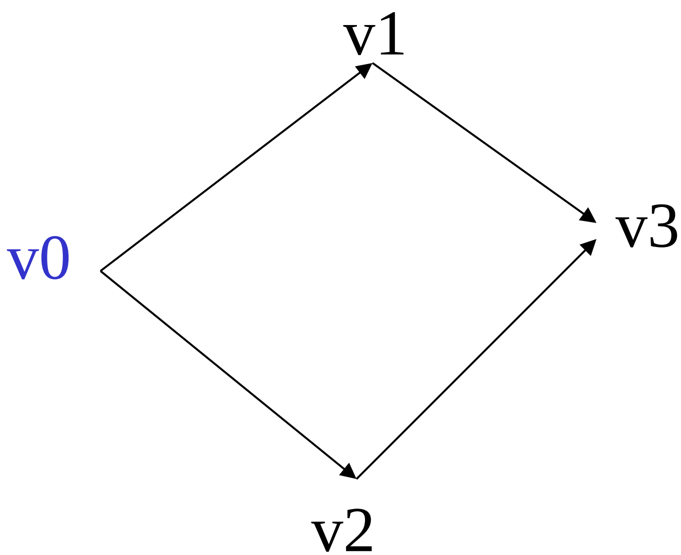
v0: {v1,v2}

v1: {v3}

v2: {v3}

v3: {}

Time complexity of BFS



Adjacency lists:

V E

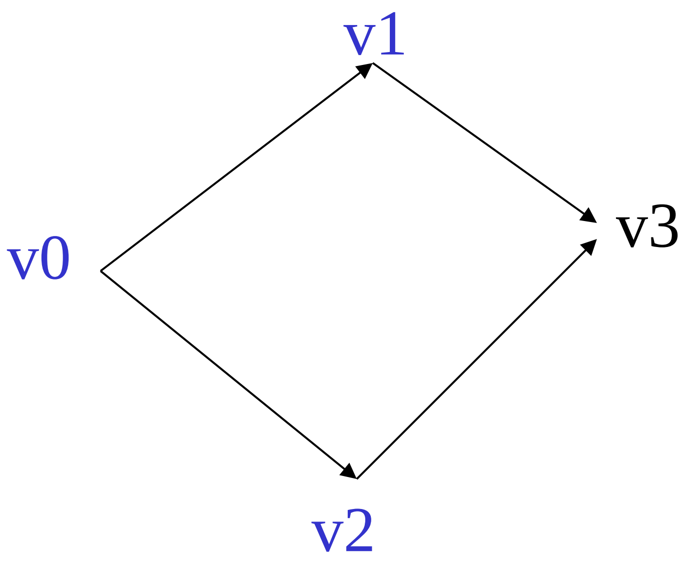
v0: {v1,v2} mark, enqueue v0

v1: {v3}

v2: {v3}

v3: {}

Time complexity of BFS



Adjacency lists:

V E

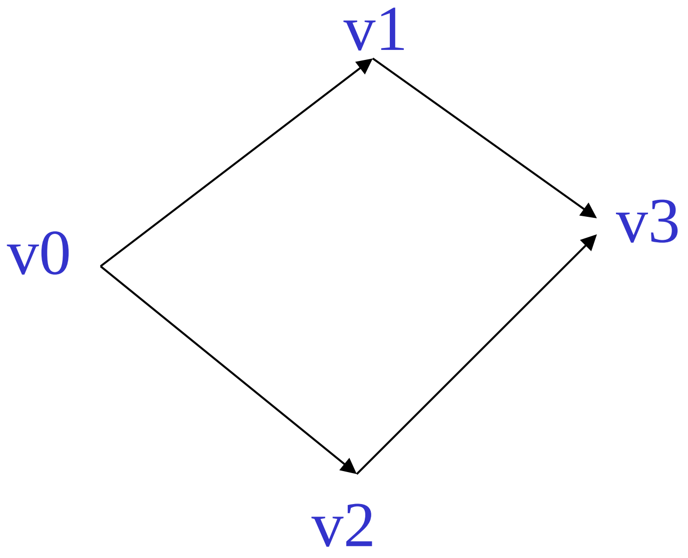
v0: {v1,v2} dequeue v0;
mark, enqueue v1,v2

v1: {v3}

v2: {v3}

v3: {}

Time complexity of BFS



Adjacency lists:

V E

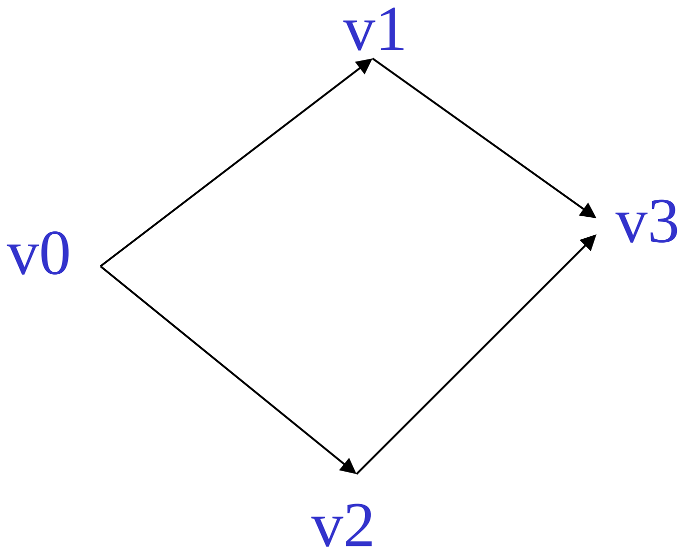
v0: {v1,v2}

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

v2: {v3}

v3: {}

Time complexity of BFS



Adjacency lists:

V E

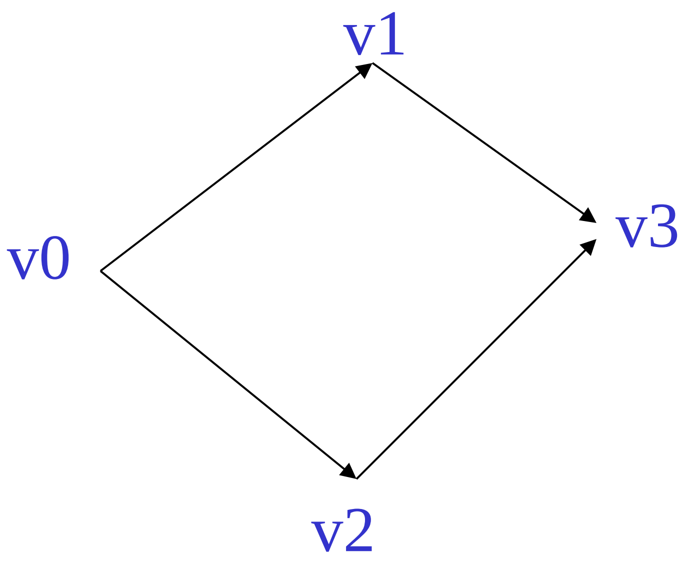
v0: {v1,v2}

v1: {v3}

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

v3: {}

Time complexity of BFS



Adjacency lists:

V E

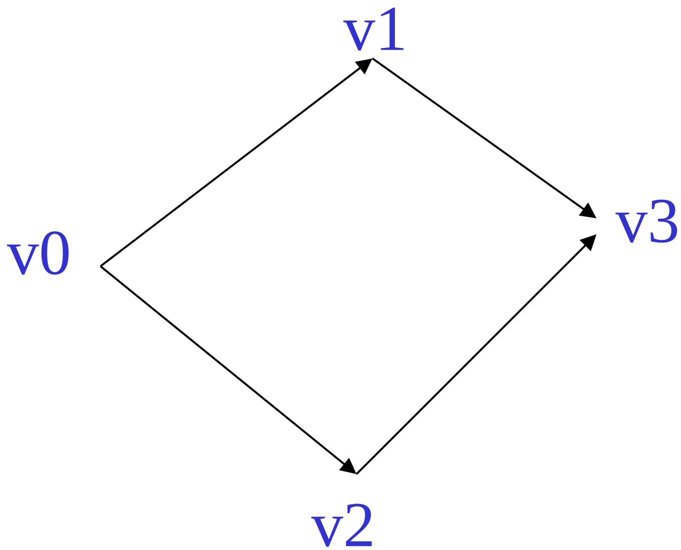
v0: {v1,v2}

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.