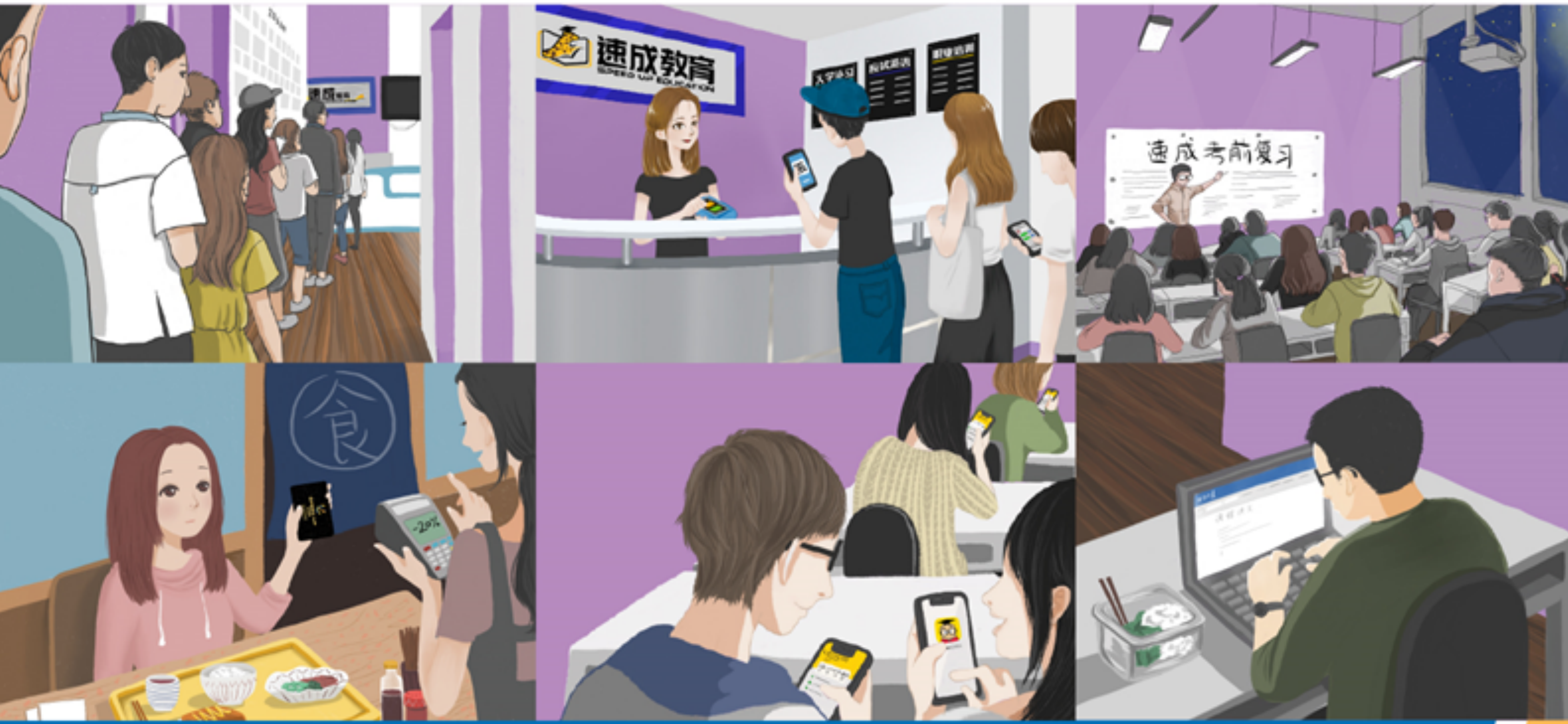
速成教育
SPEED UP EDUCATION



CSCA48

Introduction to Computer Science II

导师： **VC**

UTSC Week 10 Graph | 2025/3/10

Disclaimer

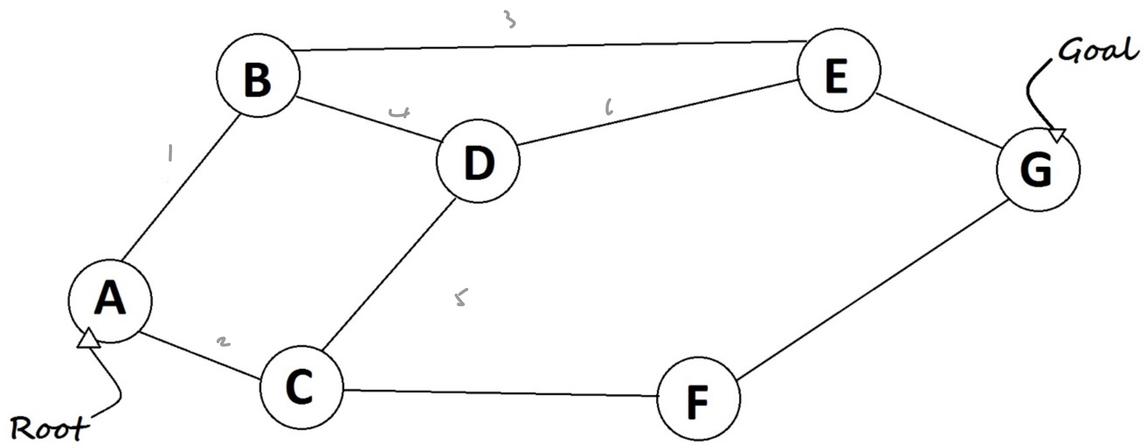
This study handout is provided by Speed Up Education and its affiliated tutors. The study handout seeks to support your study process and should be used as a complement, not substitute to the university course material, lecture notes, problem, sets, past tests and other available resources.

The study handout is distributed for free to the students participating in Speed Up Education classes and is not for sale or other commercial uses whatsoever. We kindly ask you to refrain from copying or selling in part or in whole any information provided in the handout.

Thanks for choosing Speed Up Education. We wish you the best of luck in your exam.

Definition of Graph

- **Definition of a Graph**



- A graph consists of:
  - A set of **nodes** corresponding to **data items we are working with**. The set of nodes is usually called **V**. In books, lecture notes, and future courses, you may find the term **vertex** is used instead of **node**. They are the same thing.
  - A set of **edges** which are the **connections between nodes**, and represent **the relationships** existing between **data items in our collection**. The set of edges is usually called **E**.
  - Together, they define the graph **G = (V, E)**.

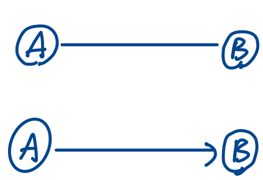
$V = \{A, B, C, D, E, F, G\}$

$|V| = 7$

$E = \{1, 2, 3, 4, \text{-----}\}$

$|E| = 9$

- Two types of graphs:
  - Un-directed graphs
  - Directed graphs



Important terms of Graph

- Neighbours
  - Un-directed graph

v is a neighbour of u

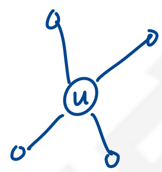
u is a neighbour of v

  - Directed graph

u is a out-neighbour of v

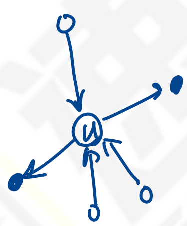
v is a in-neighbour u

- Neighbourhood
  - Un-directed graph



All nodes connected to u

  - Directed graph

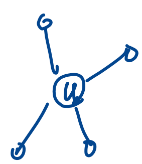


● out-neighbourhood

o in-neighbourhood

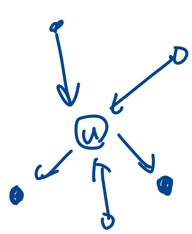
- Degree
  - Un-directed graph

↳ # of neighbours



degree: 4

  - Directed graph

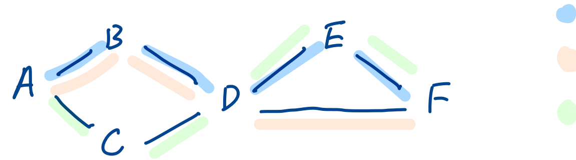


out-degree: 2

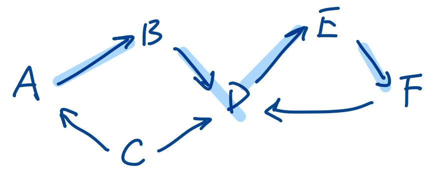
in-degree: 3

Important terms of Graph

- Path
  - Un-directed graph



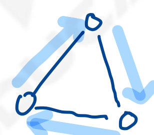
  - Directed graph



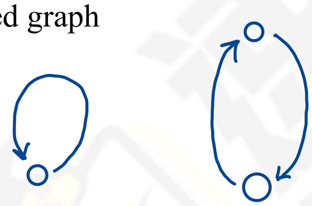
- Cycles
  - Un-directed graph

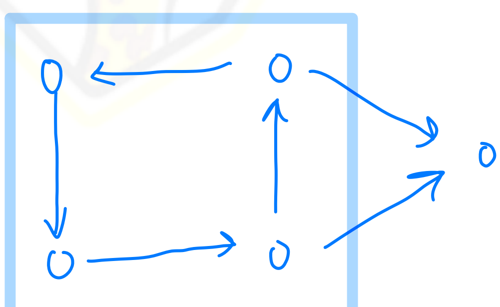
at least 3 nodes

starts and ends at the same nodes



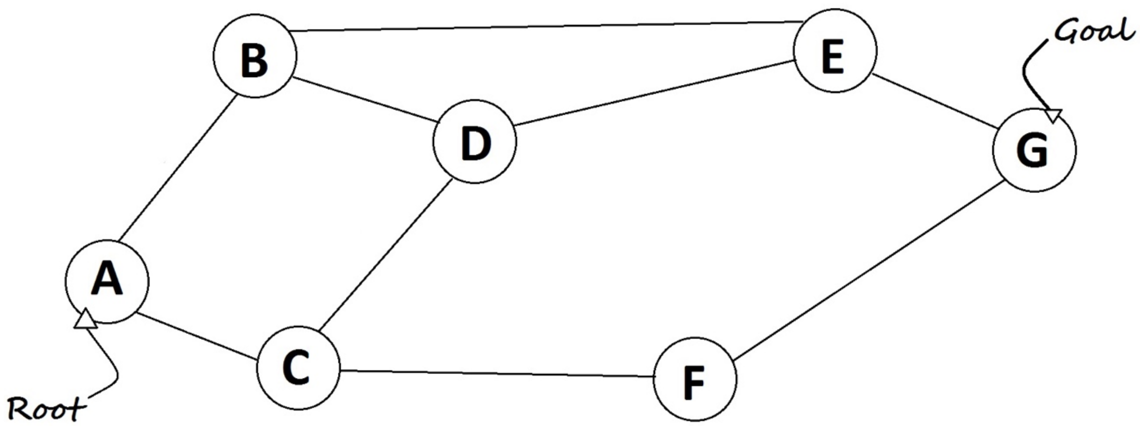
  - Directed graph





Representing Graphs

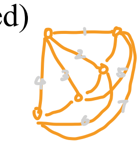
- **Adjacency List**



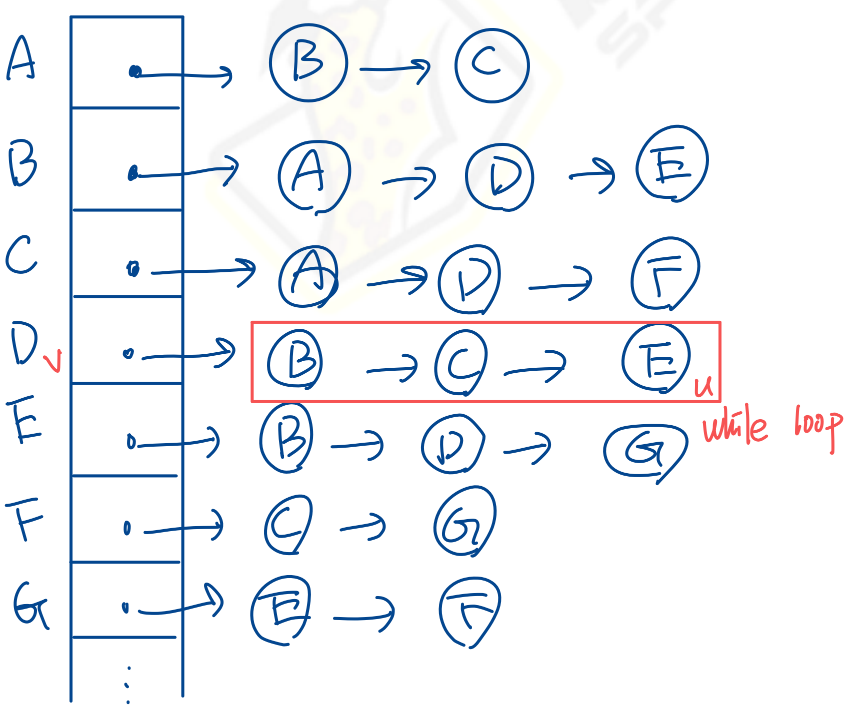
- The adjacency list is an **array** with **one entry per node**. The **i**[th] entry in the array contains a pointer to **a linked list** that stores the indexes of nodes to which node i is connected.

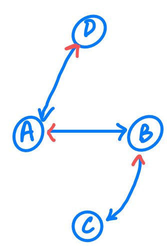
- The **advantage** of being space efficient – if the graph has a large number of nodes, but each node is connected to at most a few neighbours, then the adjacency list stores the required edge information in a very compact format – without wasting memory. **Conversely**, common graph operations such as querying an edge (figuring out whether two nodes are connected) requires list traversal, which as we know can be slow.

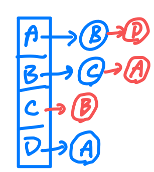
$7 \times 2 \le 5^2$
25



Pros: Space
$O(|E| \times 2) \Rightarrow O(|E|)$



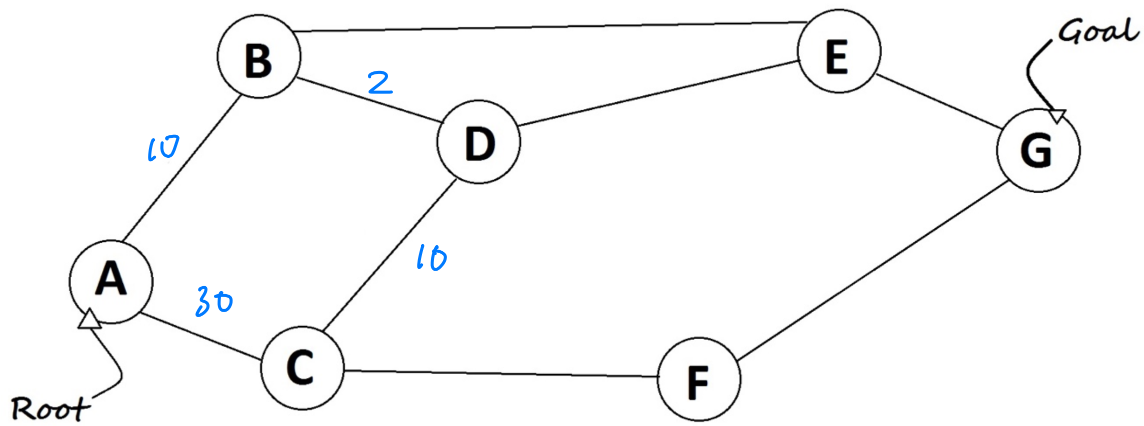




Cons: Search $O(|V|)$

Representing Graphs

- **Adjacency Matrix**



- As the name implies, the **adjacency matrix** is a 2D array of size **N x N**, where **N** is the number of **nodes** in the graph. For **un-directed graphs**, The entry **A[i][j]** is **1** if nodes **i** and **j** are connected, and **zero** otherwise. For **directed** graphs, entry **A[i][j]** is set to **1** if there is an edge **from i to j**, and is **zero otherwise**.

- Adjacency matrices have the same advantages and disadvantages of arrays: Edge queries now have no overhead, unlike linked lists which require list traversal. Adding or deleting edges, and finding out whether two nodes are connected requires a single access to the matrix. Conversely, they are not space efficient. Even in the small example above, you can see that the majority of the entries in the matrix are zero. For a very large graph, the adjacency matrix will waste a significant amount of space and may in fact not fit in memory!

from → to

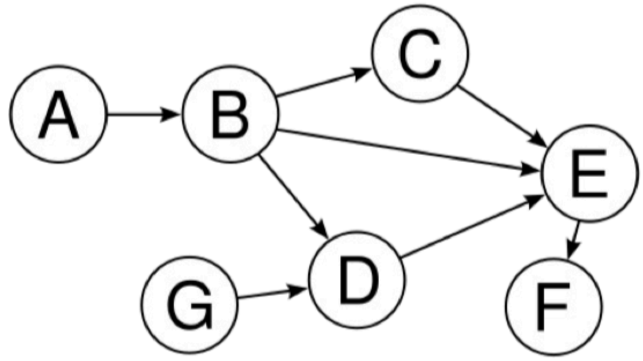
$$
\begin{array}{c|ccccccc}
 & A & B & C & D & E & F & G \\
\hline
A & 0 & \cancel{1}_{10} & \cancel{1}_{30} & 0 & 0 & 0 & 0 \\
B & \cancel{1}_{10} & 0 & 0 & \cancel{1}_{2} & \cancel{1}_{10} & 0 & 0 \\
C & \cancel{1}_{30} & 0 & & & & & \\
D_v & 0 & \cancel{1}_{2} & & & u & & \\
E & 0 & \cancel{1}_{10} & & & & & \\
F & 0 & 0 & & & & & \\
G & 0 & 0 & & & & &
\end{array}
$$

Pros: Search $O(1)$

$AdjM[v][u] \; != 0$
$\quad\quad\quad D \quad E$

Cons: Space $O(|V|^2)$

Representing Graphs – Practice



- Show the Adjacency List, use indexes 0 though 6, starting with A at index 0.

A → B
B → C → D → E
C
⋮

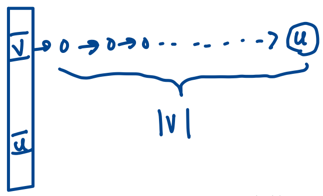
- Show the Adjacency Matrix for the graph.

	A	B	C	D	E	...
A	.	1	.	-	.	-
B	.	.	1	1	1	. - - -
C						
D						
E						
F						
⋮						

Complexity of fundamental operations on Graphs

- **Edge query**: Finding out whether two nodes (u,v) are connected

Adj. List: $O(|V|)$



Adj. Graph: $O(1)$

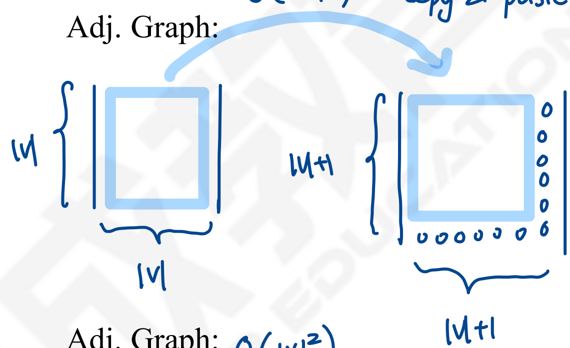
$M[v][u] \ != 0$?

- **Inserting a node**: Adding a new node to the graph

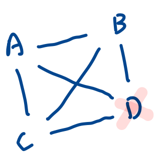
Adj. List: $O(1)$

Assume array has enough memory

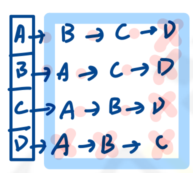
Adj. Graph: $O(|V|^2)$ copy & paste



- **Removing a node**:



Adj. List: $O(|E|)$

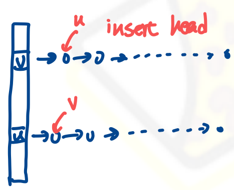


Adj. Graph: $O(|V|^2)$

copy & paste

- **Inserting an edge**:

Adj. List: $O(1)$



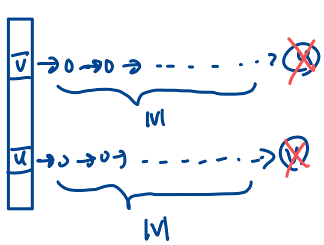
Adj. Graph: $O(1)$

$M[v][u] = 1$ (cost)

$M[u][v] = 1$

- **Removing an edge**:

Adj. List: $O(|V|)$



Adj. Graph: $O(1)$

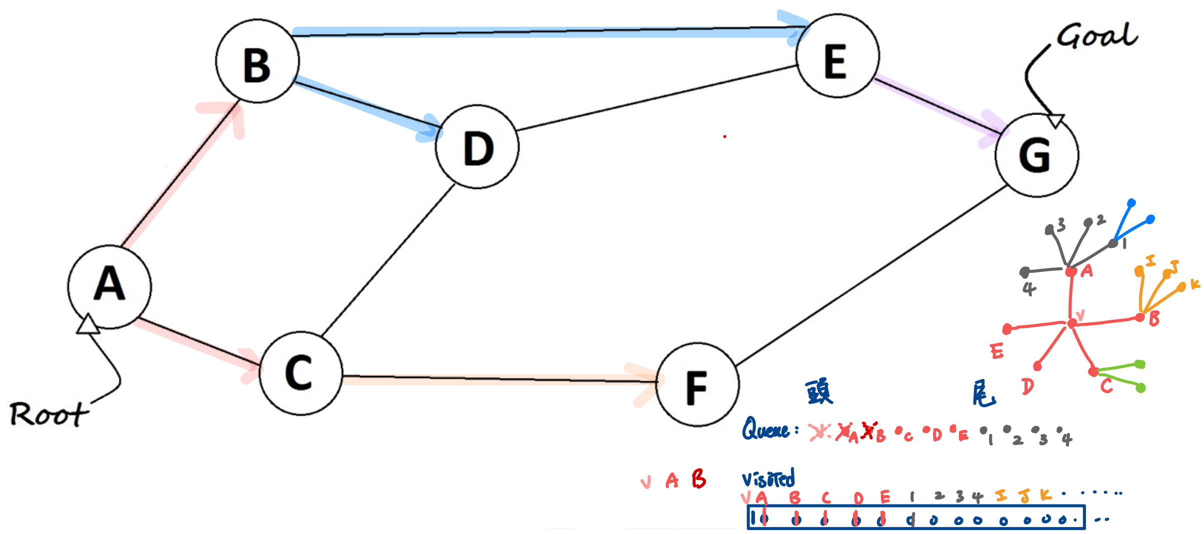
$M[v][u] = 0$

$M[u][v] = 0$

Graph Search

B C D E F G

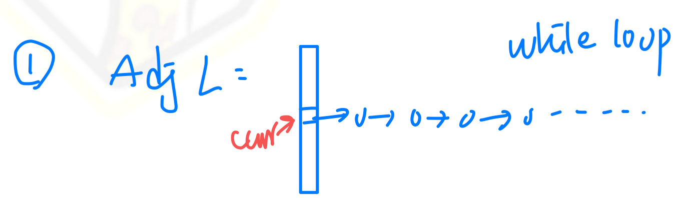
- **Breadth-First Search (BFS)**

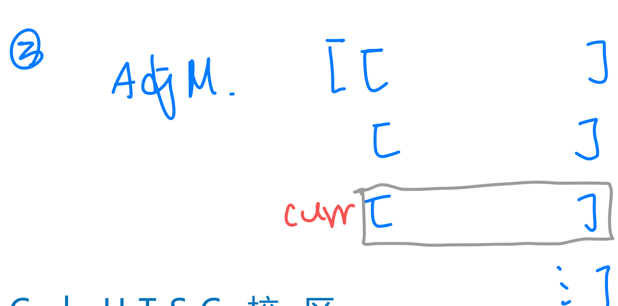


- It starts at the root and explores all of its children in the next level (neighbors) before moving to each of the root children, and then, it explores the children of the root children, and so on. The algorithm uses a queue to perform the BFS.

- Pseudocode: (Iterative)

```
1. Add root node to the queue, and mark it as visited (already explored).
2. Loop on the queue as long as it's not empty.
   1. Get and remove the node at the top of the queue(current).
   2. For every non-visited child of the current node, do the following:
       1. Mark it as visited.
       2. Check if it's the goal node, if so, then return it.
       3. Otherwise, push it to the queue.
3. If queue is empty, then goal node was not found!
```

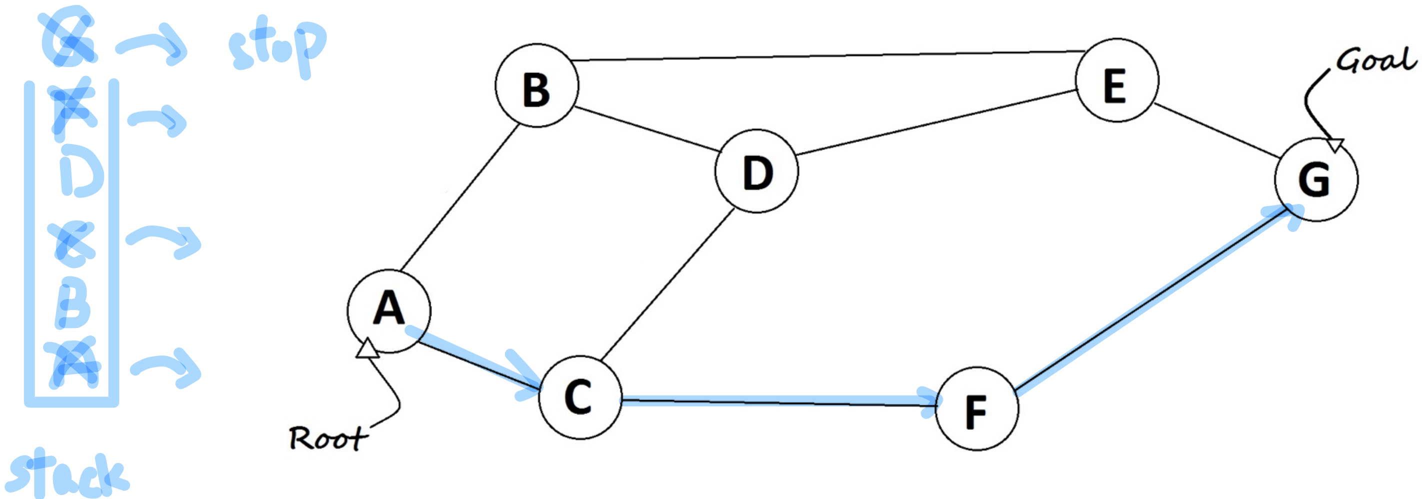




for (int i = 0; i < |V|; i++)
if (adjM [curr] [i] != 0)
if (visited [i] == 0)
⋮

Graph Search

- **Depth-First Search (DFS)**

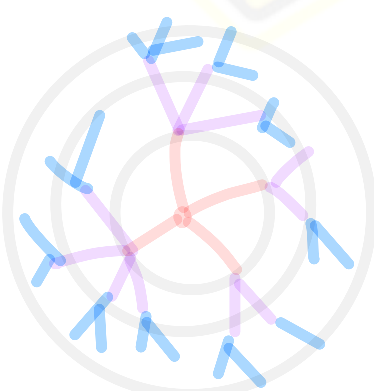


- It starts at the root and explores one of its children's sub tree, and then move to the next child's sub tree, and so on. It uses stack, or recursion to perform the DFS.

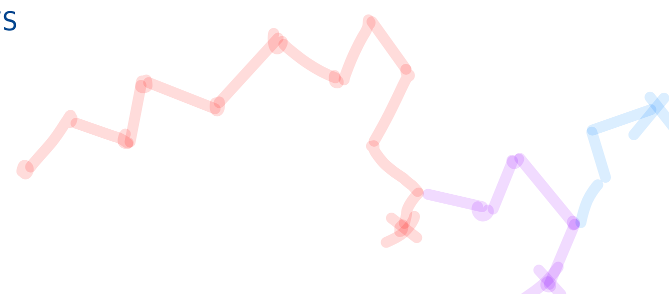
- Pseudocode: (Iterative)

```
1. Add root node to the stack.
2. Loop on the stack as long as it's not empty.
    1. Get the node at the top of the stack(current),
       mark it as visited,  remove it.
    2. For every non-visited child of the current node, do the following:
         1. Check if it's the goal node, if so, then return this child node.
         2. Otherwise, push it to the stack.
    3. If stack is empty, then goal node was not found!
```

BFS

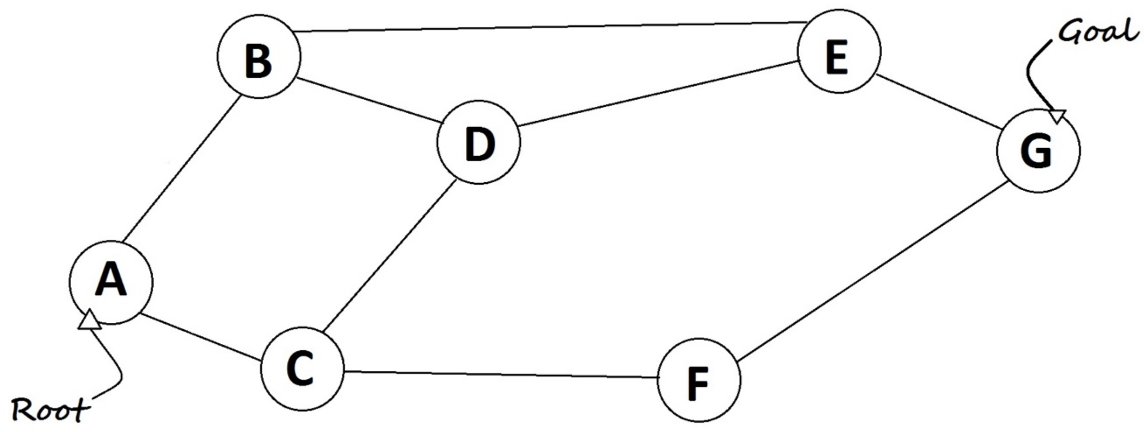


DFS



Graph Search

- **Depth-First Search (DFS)**



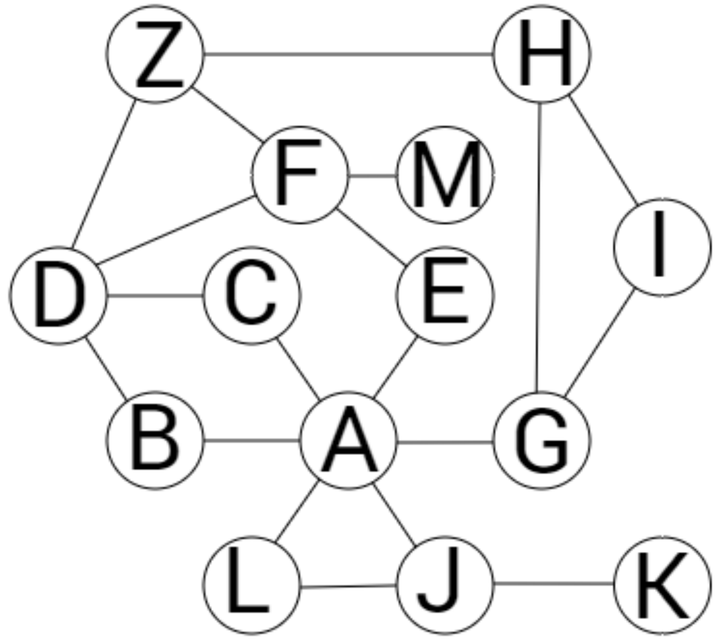
- It starts at the root and explores one of its children's sub tree, and then move to the next child's sub tree, and so on. It uses stack, or recursion to perform the DFS.

- Pseudocode: (Recursive)

```
1. Mark the current node as visited(initially current node is the root node)
2. Check if current node is the goal, If so, then return it.
3. Iterate over children nodes of current node, and do the following:
    1. Check if a child node is not visited.
    2. If so, then, mark it as visited.
    3. Go to it's sub tree recursively until you find the goal node
       (In other words, do the same steps here passing the child node as the
        current node in the next recursive call).
    4. If the child node has the goal node in this sub tree, then, return it.
4. If goal node is not found, then goal node is not in the tree!
```

Graph Search – Practice

- Perform BFS on the given graph.



- Perform DFS on the given graph.

```
                              int goal
void DFS(int adjM[N][N], int v, int visited[N]) {
   printf("…")      if (v == goal) return True;

   // mark it as visited
   visited[v] = 1;

   // check all the neighbours of v
   for (int i = 0; i < N; i++) {
      // if there is an edge between v and i
      // and i is not visited
      if (adjM[v][i] != 0 && visited[i] == 0)
         // do the same thing on i (recurse on i)
         DFS(adjM, i, visited)
}                 goal
```