

# COMP20003 Workshop Week 10

## Graphs

Graphs: concepts, properties, representation  
Graph Search: DFS, BFS  
Graph Traversal  
Implicit Graphs [in preparation for Assignment 3]

LAB:

- do Peer Activities & exercises, including a code filling.

# Graphs: Concepts

Formal definition:

$G = (V, E)$  where

$V = \{v_i\}$  : set of vertices

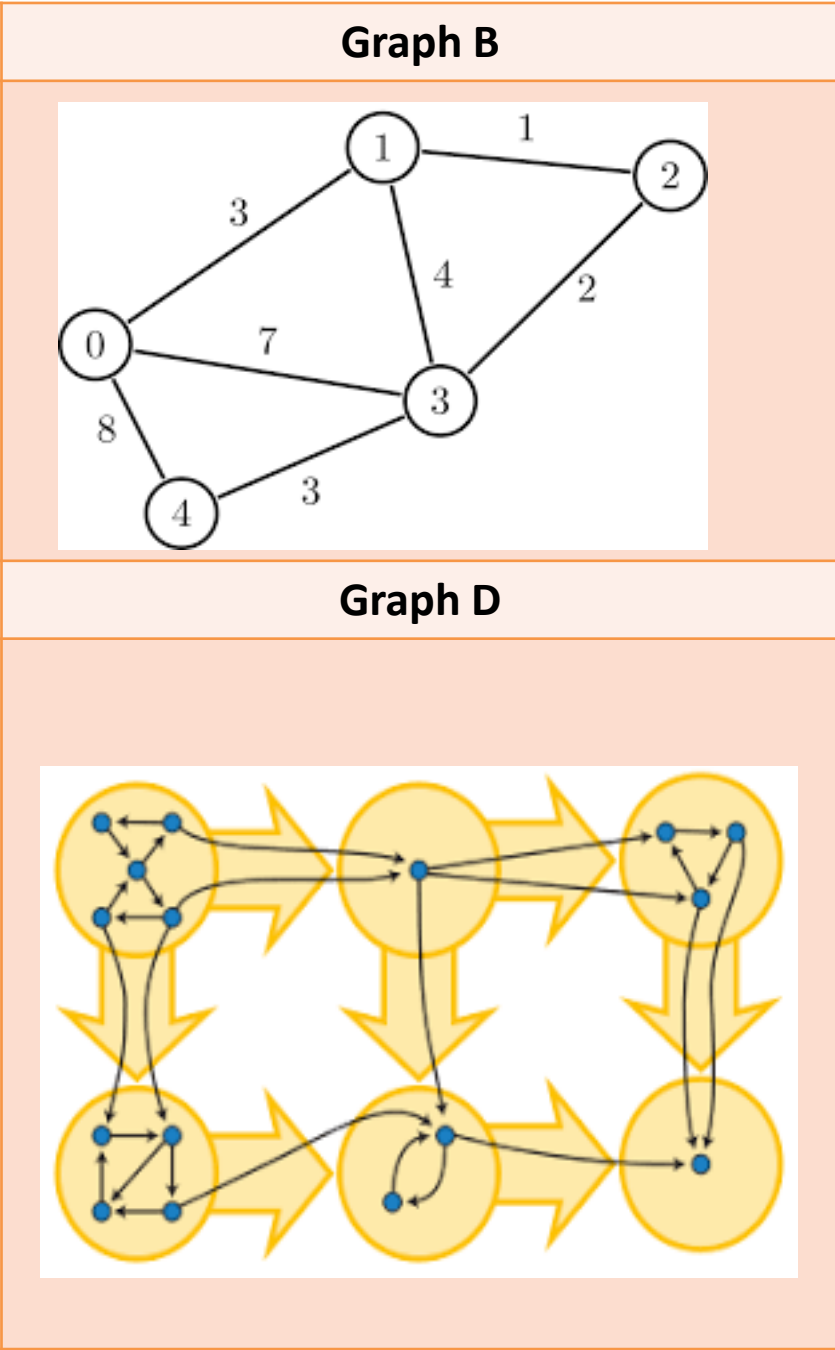
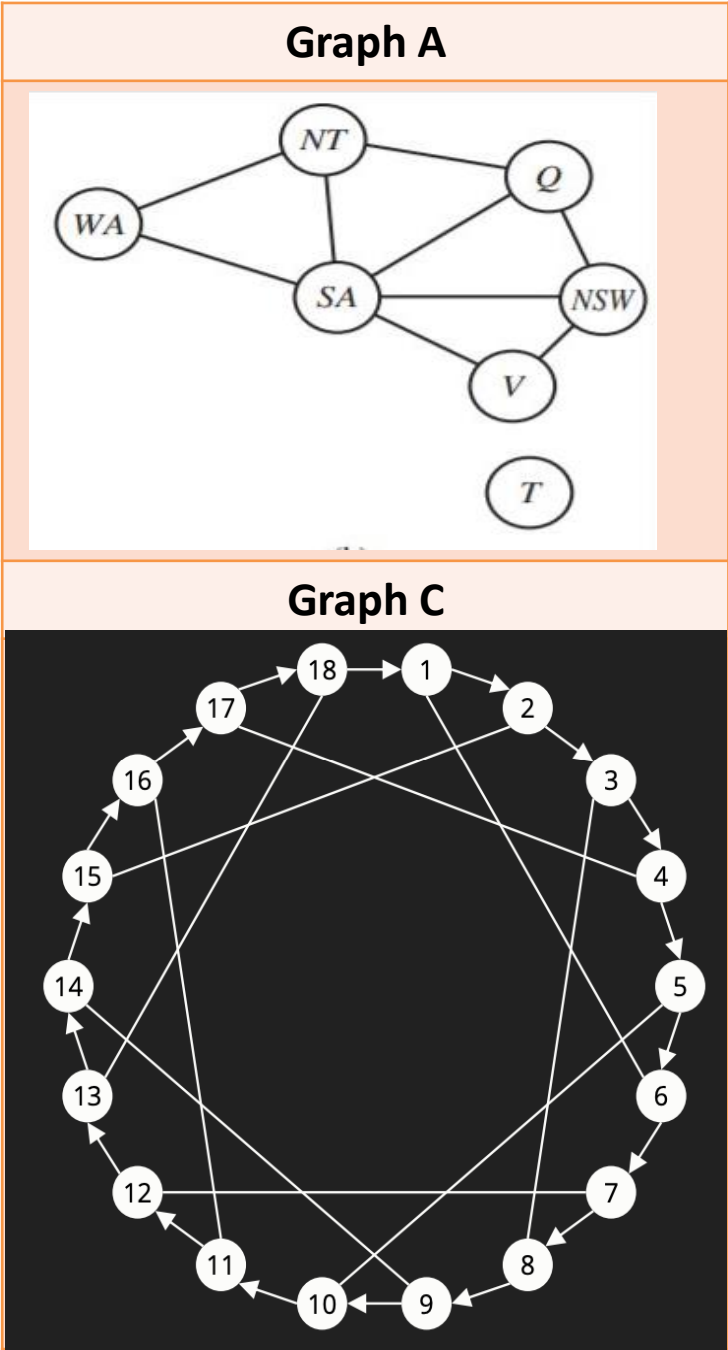
$E = \{(v_i, v_j)\}$  : set of edges;

Graph properties:

- complete OR incomplete
- weighted OR unweighted
- cyclic OR acyclic
- connected OR unconnected
- number of connected components
- dense OR sparse OR neither
- directed (di-graph) OR undirected

For di-graphs only:

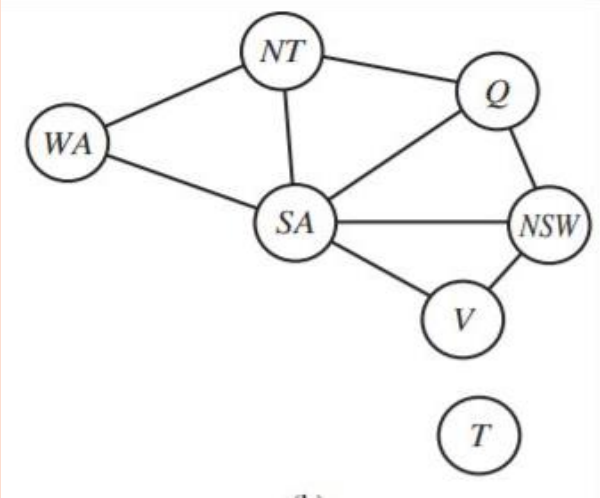
- is a DAG (Directed Acyclic G)?
- strongly OR weakly connected di-graph OR neither
- number of strongly connected components



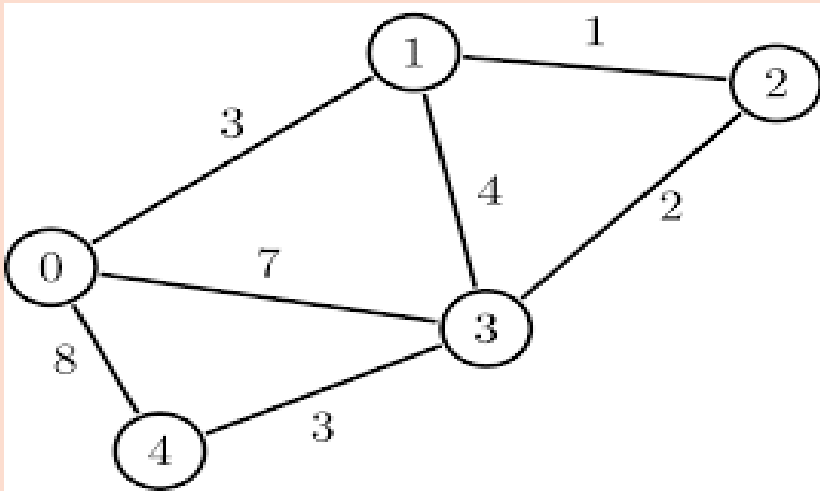
# Graphs: Concepts

	A	B	C	D
complete?	✗	✗	✗	✗
weighted?	✗	✓	✗	✗
cyclic?	✓	✓	✓	✓
connected graph?	✗	✓	✓	✗
number of connected components	2	1	1	6
dense?	✗	✗	✗	?
sparse?	✓	✓	✓	?
directed? (= di-graph?)	✗	✗	✓	✓
for di-graphs only				
weakly connected?	-	-	✗	✓
strongly connected?	-	-	✓	✗
number of strongly connected components	-	-	1	6

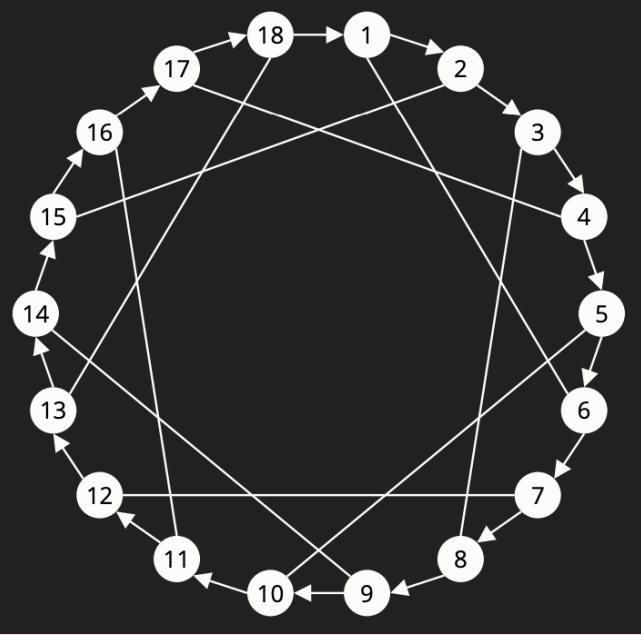
### Graph A



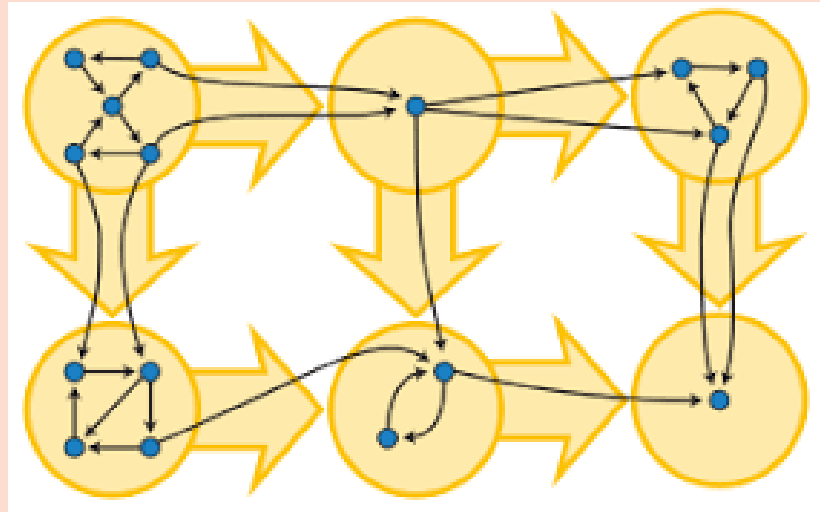
### Graph B



### Graph C



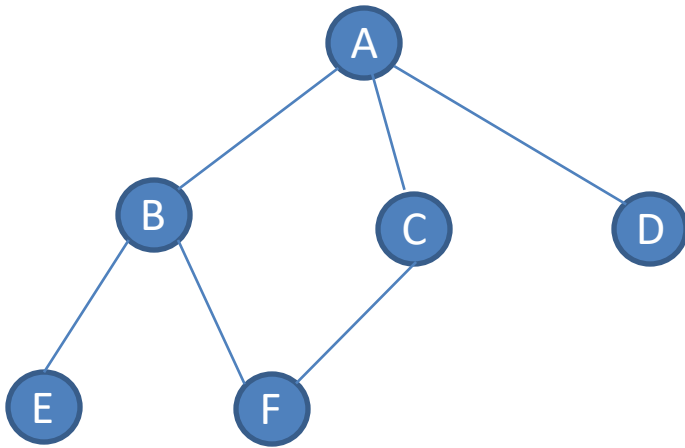
### Graph D



# Graph representation

Vertices: Vertices can be represented as integers  $0..|V|-1$

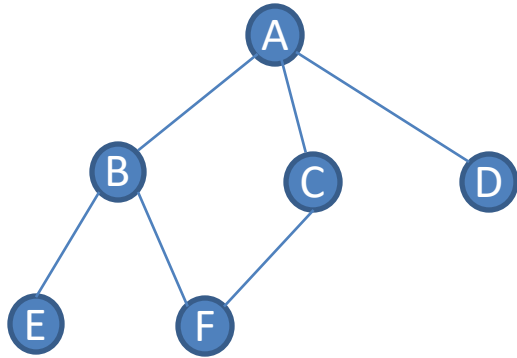
- If need to store some other properties of each vertices (such as vertex label) we can put them in an array:



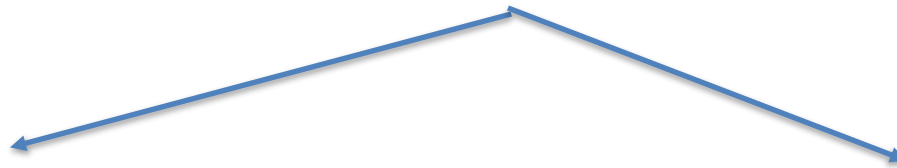
id	label
0	"A"
1	"B"
2	"C"
3	"D"
4	"E"
5	"F"

How to represent Edges: ???

# Graph representation



Edges: depending on how to represent edges, we have different graph representations:



using (an *array of adjacency lists*)

id	label	adj list
0	"A"	B, C, D
1	"B"	A, E, F
2	"C"	A, F
3	"D"	A
4	"E"	B
5	"F"	B, C

using an *adjacency matrix*

A is a matrix of V x V		TO					
		A	B	C	D	E	F
FROM	A	0	1	1	1	0	0
	B	1	0	0	0	1	1
	C	1	0	0	0	0	1
	D	1	0	0	0	0	0
	E	0	1	0	0	0	0
	F	0	1	1	0	0	0

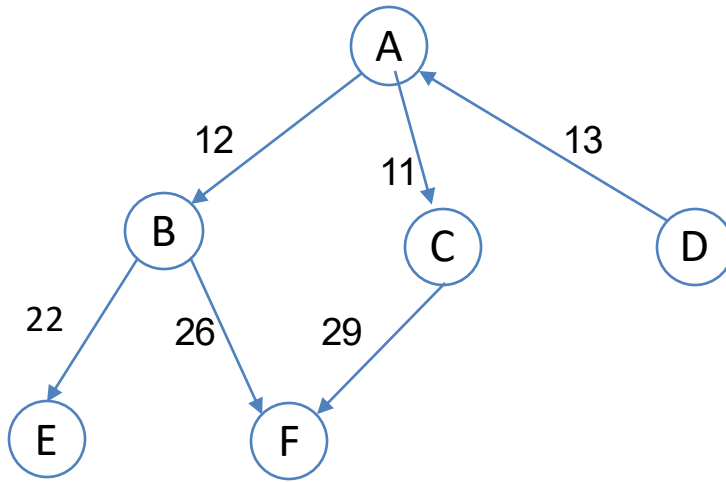
## NOTES

$A[i][j] =$

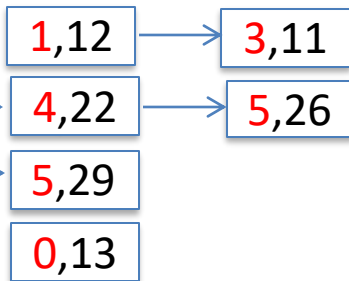
- 1 if there is an edge from  $i \rightarrow j$
- 0 otherwise

For an undirected graph, the matrix is symmetric

# representation: weighted di-graph example



id	name	adj list
0	"A"	
1	"B"	
2	"C"	
3	"D"	
4	"E"	
5	"F"	



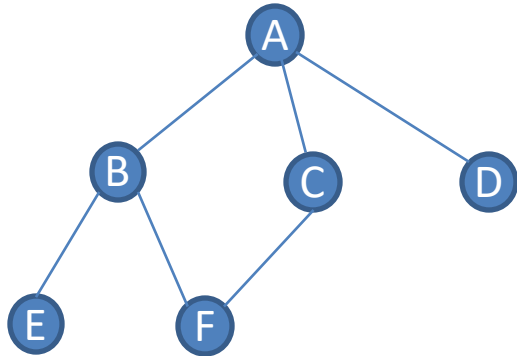
A is a matrix of V x V		TO					
		0	1	2	3	4	5
FROM	0		12	11			
	1					22	26
	2						29
	3	13					
	4						
	5						

## NOTES

$A[i][j] =$

- weight of edge  $i \rightarrow j$
- UNDEFINED otherwise
- depending on situations, UNDEFINED could be set as 0 or  $\infty$

# Graph representation : Space Complexity



*Time/Space Complexity of graph algorithms is represented as a function of:*

- $|V|$  (or  $V$ ): number of vertices
- $|E|$  (or  $E$ ): number of edges

*array of adjacency lists*

id	label	adj list
0	"A"	B, C, D
1	"B"	A, E, F
2	"C"	A, F
3	"D"	A
4	"E"	B
5	"F"	B, C

*adjacency matrix*

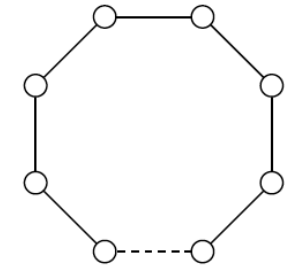
A is a matrix of $V \times V$		TO					
		A	B	C	D	E	F
FROM	A	0	1	1	1	0	0
	B	1	0	0	0	1	1
	C	1	0	0	0	0	1
	D	1	0	0	0	0	0
	E	0	1	0	0	0	0
	F	0	1	1	0	0	0

*Space complexity:*

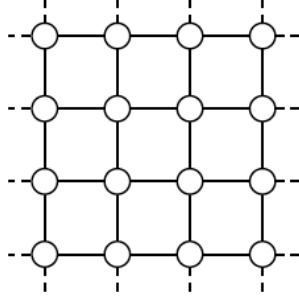
$$\theta(V+E)$$

$$\theta(V^2)$$

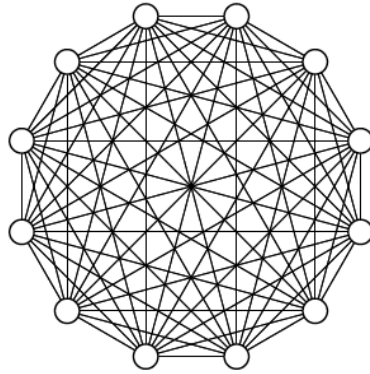
# Graphs Representation: sparse & dense graphs



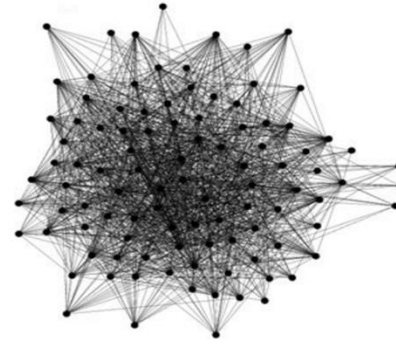
A



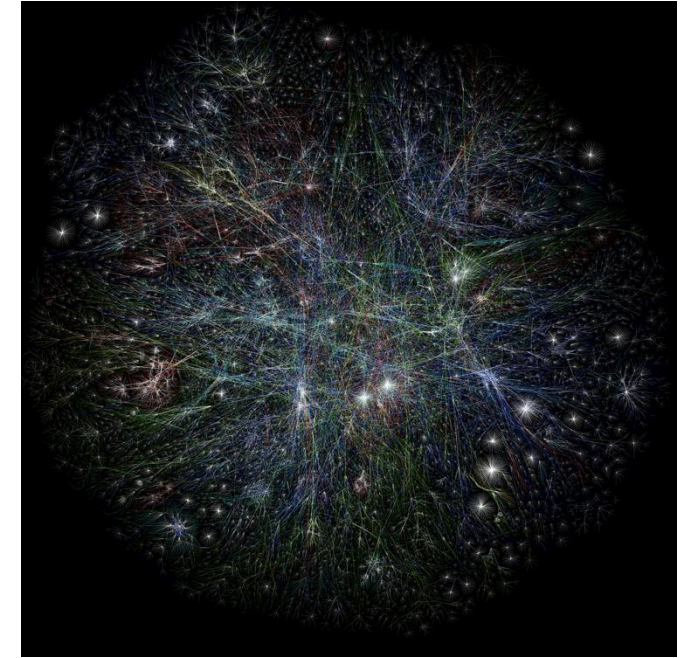
B



C



D



E: The Internet

Is each graph dense/ sparse?

The criteria are:

- Sparse graphs:  $E = O(V)$
  - Dense graphs:  $E = \Theta(V^2)$
- } in those case complexity is a function of  $V$  only

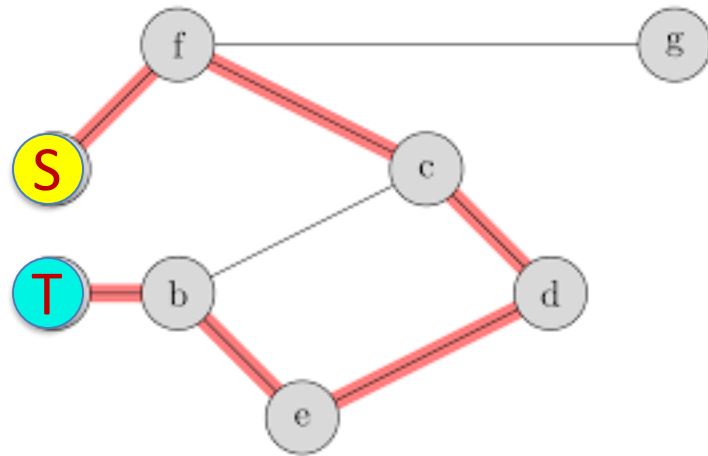
→ some graphs are neither dense nor sparse ☺

**What representation (Adjacency List or Matrix) is better for each case?**

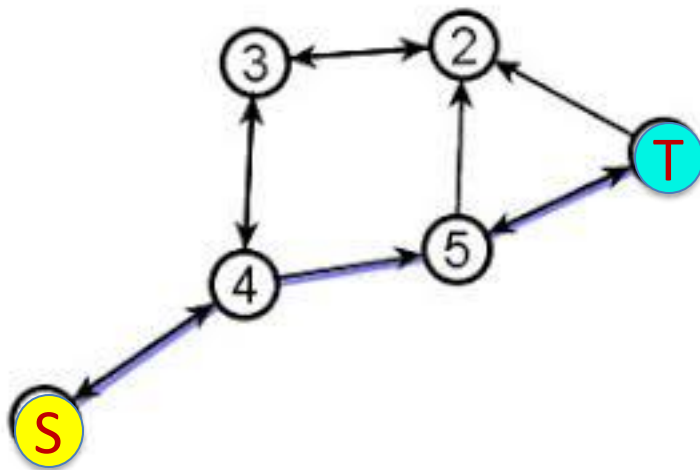
- **Sparse Graphs:**  $O(V+E)$  →
- **Dense Graphs:**  $\Theta(V^2)$  →



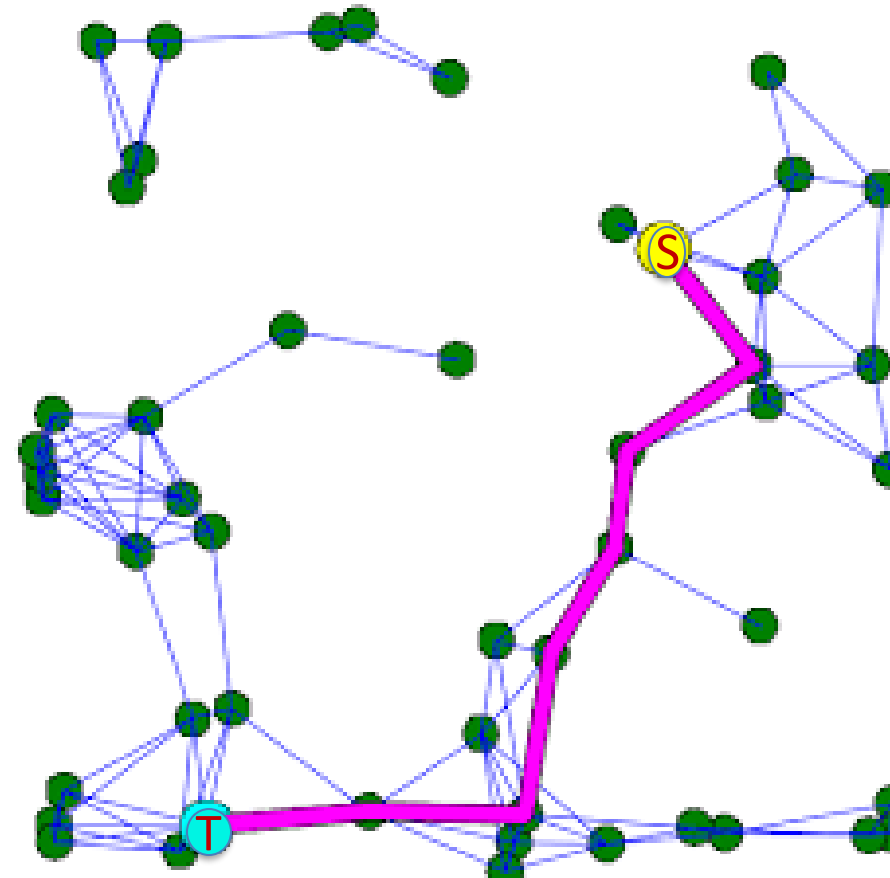
# Paths in unweighted graphs: path length, shortest path



Paths=  $S \rightarrow f \rightarrow c \rightarrow d \rightarrow e \rightarrow b \rightarrow T$ , length=6  
 $S \rightarrow f \rightarrow c \rightarrow b \rightarrow T$ , length=4 [shortest]



Path S=  $S \rightarrow 4 \rightarrow 5 \rightarrow T$ , length= 3 [shortest]



# Graph Search

## Basic Concept:

- *Graph Search is Path Finding: finding a path from a source node to a destination node.*
- The Task: given a source node **S** and a destination node **D**, find a path from **S** to **D**.
- Output: a path  $S \rightarrow v_1 \rightarrow v_2 \rightarrow \dots \rightarrow v_{n-1} \rightarrow D$  with  $n$  is the path length
- Note: in most of the cases, the objective is to find a *Shortest Path*: a path that has minimal possible length

## Examples:

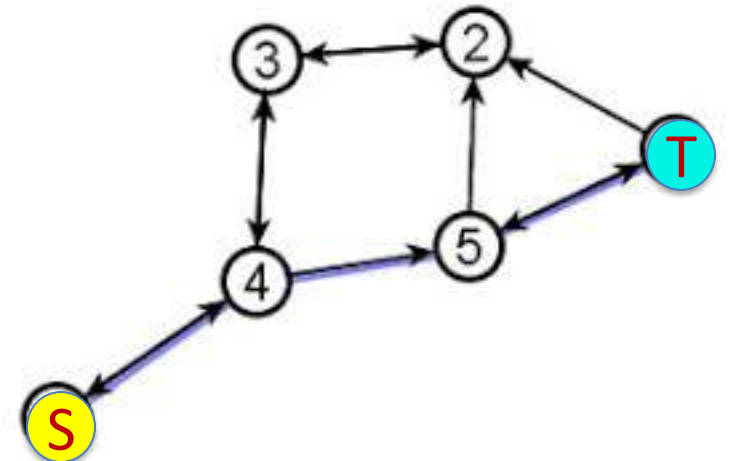
- GPS or Google Map

A **classification** of Graph Search (not quite popular):

- **1S1D**: search for a path from a single Source to a single Destination
- **1S\*D**: single source, all possible destinations
- **\*S\*D**: finding paths between all pairs of nodes

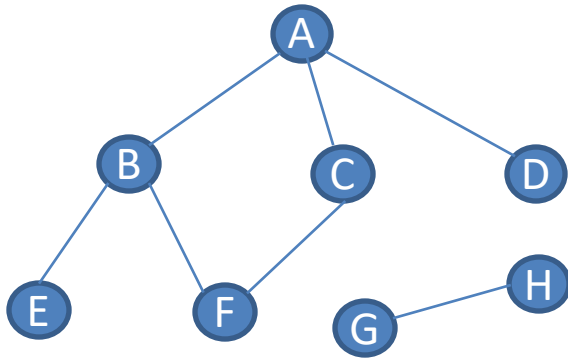
## Two basic strategies:

- Depth-First Search (DFS)
- Breadth-First Search (BFS)

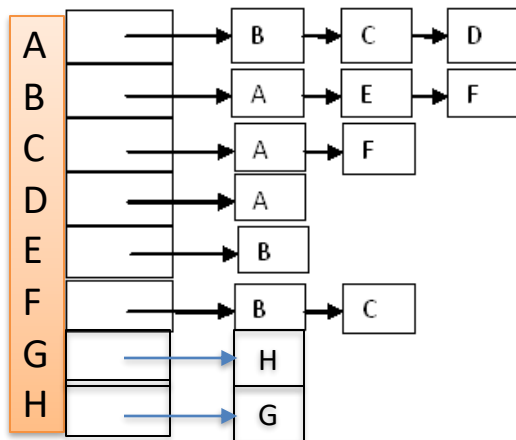


Path S=  $S \rightarrow 4 \rightarrow 5 \rightarrow T$ , length= 3 [shortest]

# Depth-First Search: using recursion



Adjacency-List Array



Q1: List the nodes in order of visited by DFS(A)

? A

Q2: Does DFS(A) help to find the shortest paths from A ?

DFS (source) :

- runs DFS from node source
- will find paths from u to all nodes that are connected to u

the following variables can be set inside or outside the function DFS:

order= 0; // order represents timestamp

for (v=0; v<V; v++)

visited[v] = 0; // mark as "unvisited"

function DFS(int u) {

// arrives u == found a path to node u

visited[u]= ++order;

for each neighbor v of u

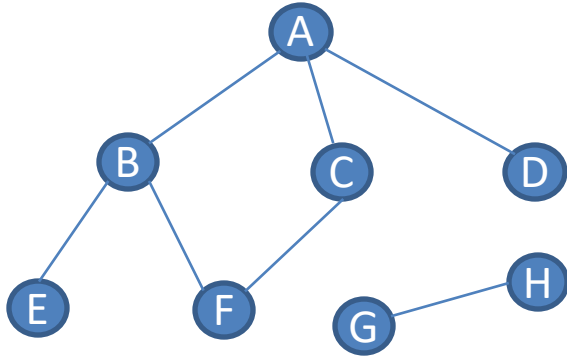
if (!visited[v])

DFS(v);

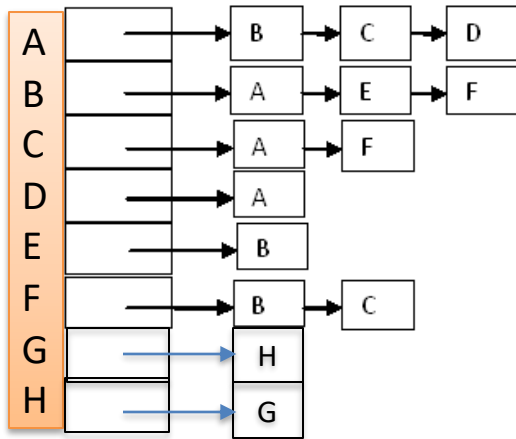
}

# Depth-First Search

## using recursion



Adjacency-List Array

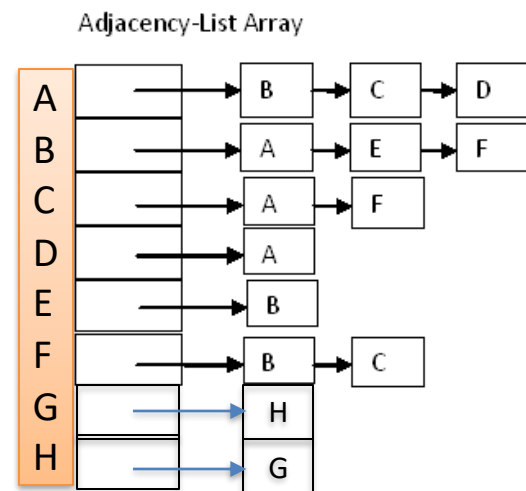
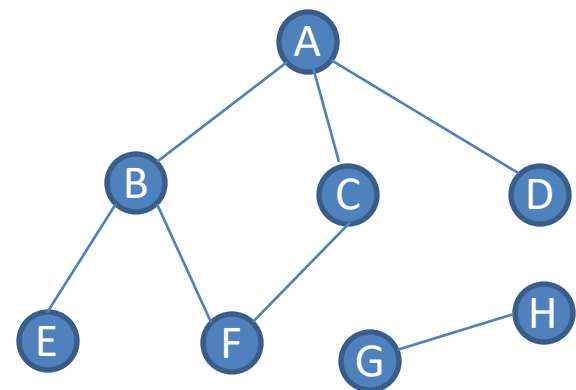


## Notes on stack-based DFS:

- DFS can be done iteratively with an explicit stack instead of recursion (which implicitly employs a stack)
- The order of pushing/popping nodes when using explicit stack can also be obtained from the recursive version as below:

```
function DFS(int u) {  
    // arrives node u  
    // Here u is pushed into stack  
  
    visited[u] = ++order;  
    for each neighbor v of u  
        if (!visited[v])  
            visitDFS(v);  
  
    // departs from node u  
    // Here u is popped out of stack  
}
```

# Breadth-First-Search: using a queue



the following variables can be set inside or outside the function BFS:

`order = 0; // order represents timestamp`

`for (v=0; v<V; v++)`

`visited[v] = 0; // mark as "unvisited"`

```
function BFS(int s) {
```

```
    Q = empty queue
```

```
    enQ(Q, s)
```

```
    while (Q not empty) {
```

```
        u = deQ(Q)
```

```
        // arrives to node u
```

```
        if (!visited[u]) {
```

```
            visited[u] = ++order; // visit u, and mark as visited
```

```
            for each neighbor v of u
```

```
                if (!visited[v]) enQ(Q, v)
```

```
        }
```

```
    }
```

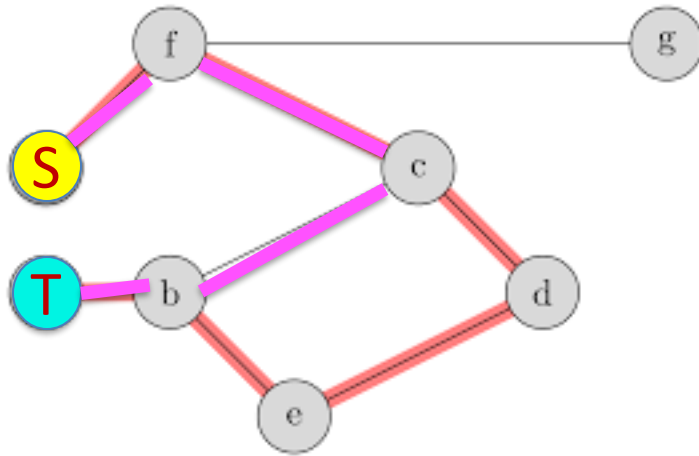
```
}
```

Q1: List the nodes in order of visited by BFS. *On tie choose the smallest element.*

? A

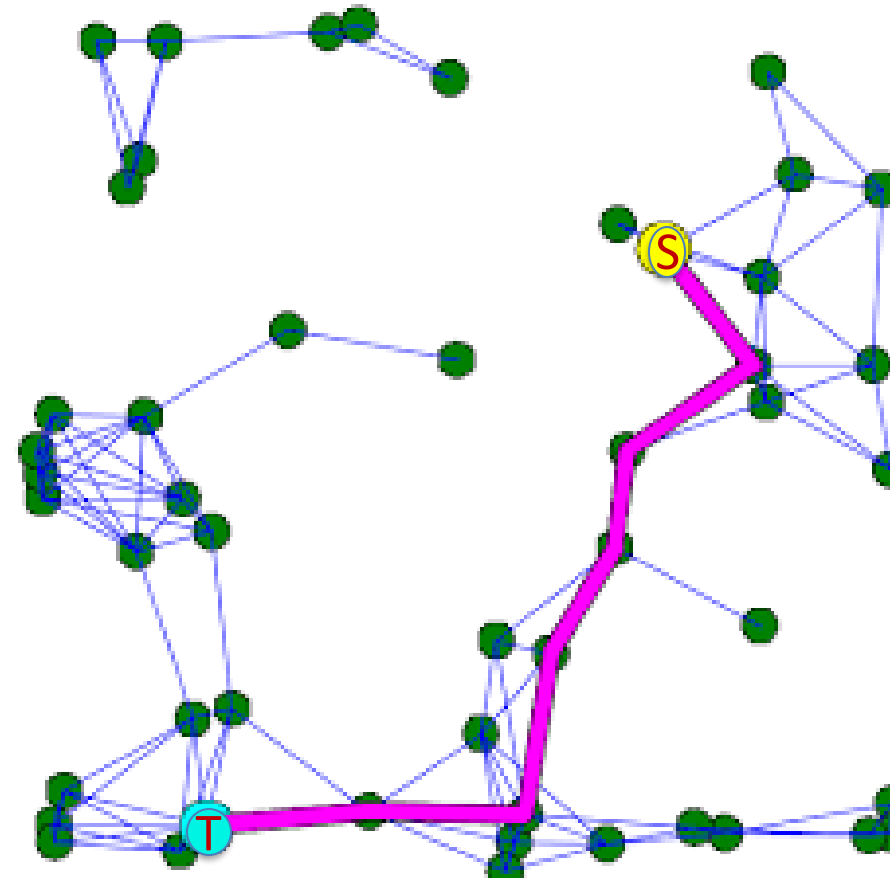
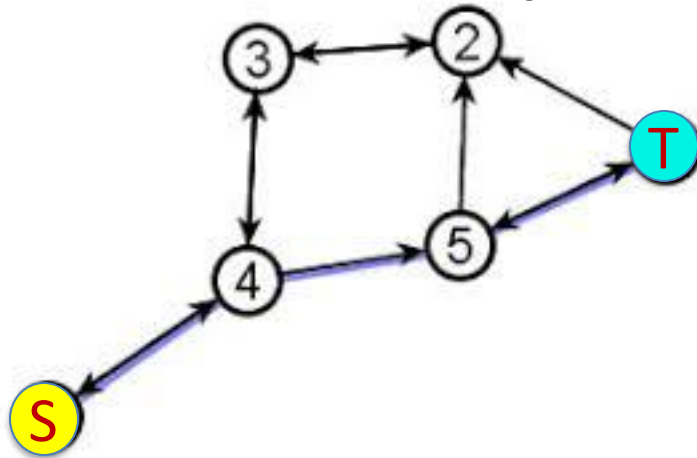
Q2: Does BFS(A) help to find the shortest paths from A?

# BFS (not DFS) helps to find shortest paths in unweighted graphs!

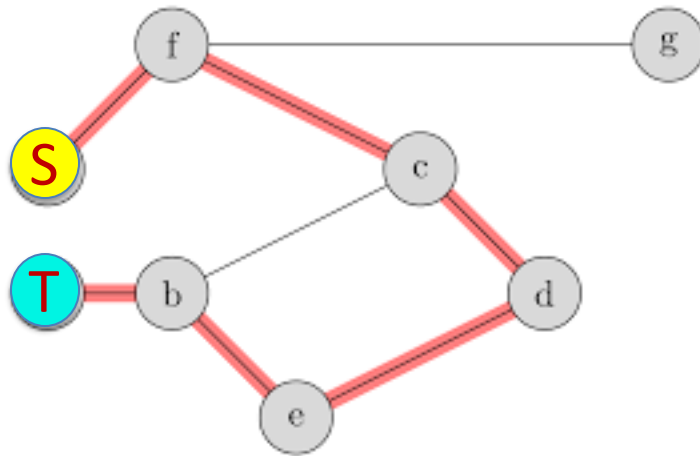


Path=  $S \rightarrow f \rightarrow c \rightarrow b \rightarrow T$ , length=4

Path=  $S \rightarrow 4 \rightarrow 5 \rightarrow T$ , length= 3

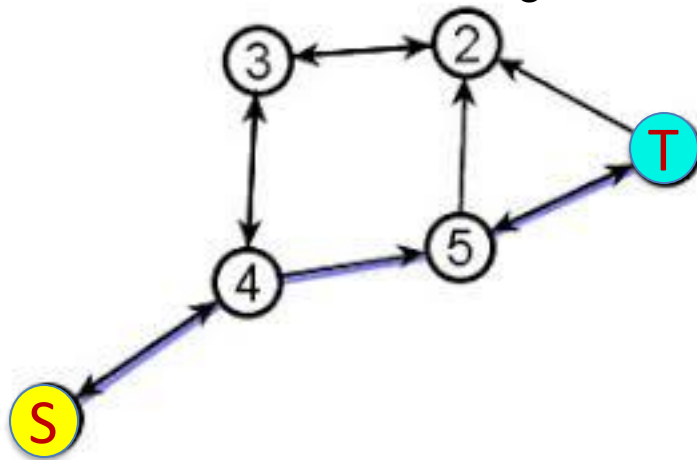


# 1S1D: Adapting BFS for shortest path from a single source to single destination



Path=  $S \rightarrow f \rightarrow c \rightarrow b \rightarrow T$ , length=4

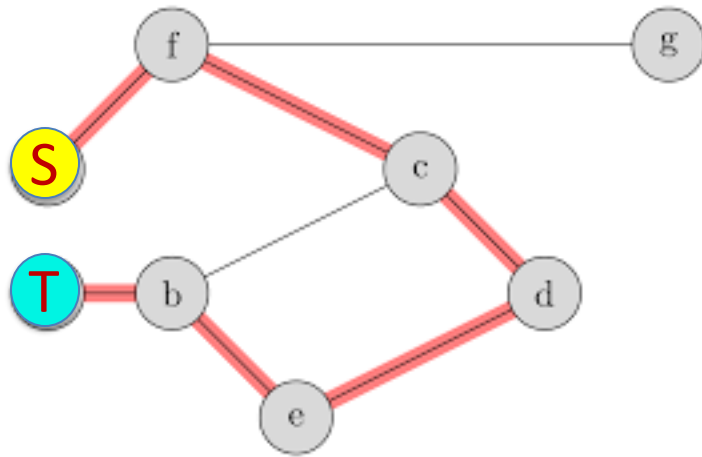
Path=  $S \rightarrow 4 \rightarrow 5 \rightarrow T$ , length= 3



Adapting BFS for 1S1D= find shortest path from a source to a destination in unweighted graphs

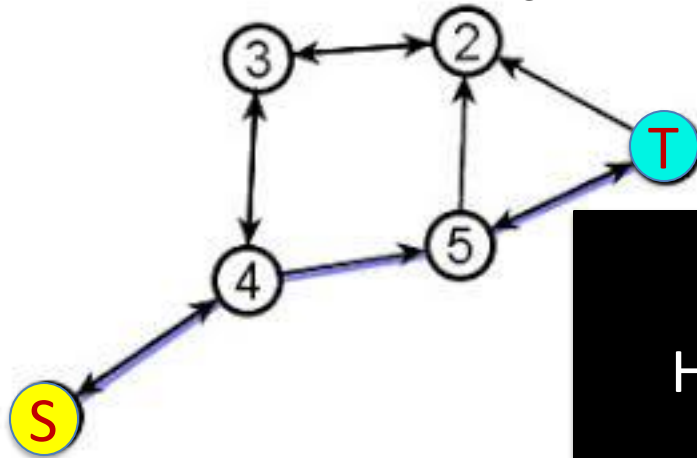
```
function BFS(int source, int dest) {  
    Q= empty queue  
    enQ(Q, source)  
    while (Q not empty) {  
        u = deQ(Q)  
        ??  
        if (!visited[u]) {  
            visited[u]= 1; // visit u, and mark as visited  
            for each (u,v)  
                if (!visited[v]) enQ(Q,v)  
        }  
    }  
}
```

# 1S1D: Adapting BFS for shortest path from a **single source** to **single destination**



Path=  $S \rightarrow f \rightarrow c \rightarrow d \rightarrow e \rightarrow b \rightarrow T$ , length=6

Path=  $S \rightarrow 4 \rightarrow 5 \rightarrow T$ , length= 3



Adapting BF traversal for 1S1D shortest path in unweighted graphs

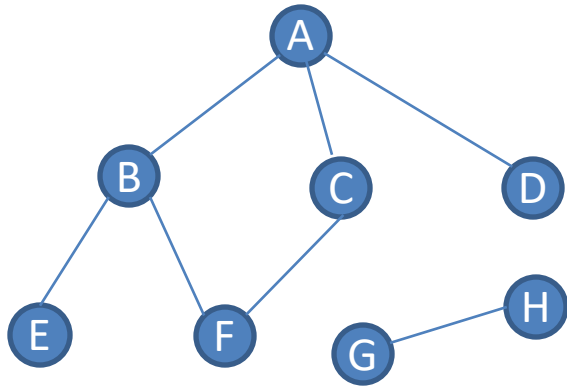
```
function BFS(int source, int dest) {  
    Q= empty queue  
    enQ(Q, source)  
    while (Q not empty) {  
        u = deQ(Q)  
        if (u == dest) return path;  
        if (!visited[u]) {  
            visited[u]= 1; // visit u, and mark as visited  
            for each (u,v)  
                if (!visited[v]) enQ(Q,v)  
        }  
    }  
    return NOTFOUND;  
}
```

Q:

How to retrieve the shortest path from **source** to **dest**?



# Graph Traversal = What? How?



**Traversal:** visit nodes of a graph in a systematic way.

Understanding the 2 main strategies for graph traversal:

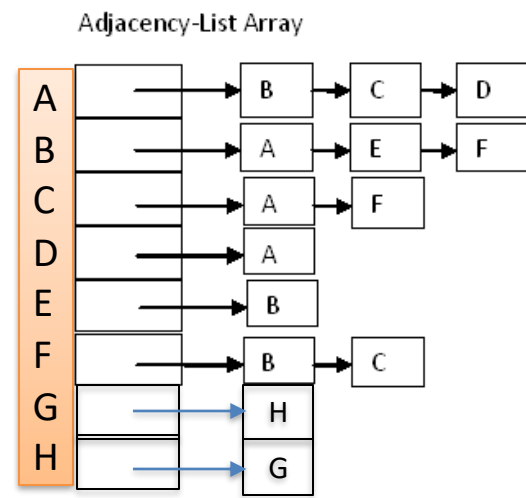
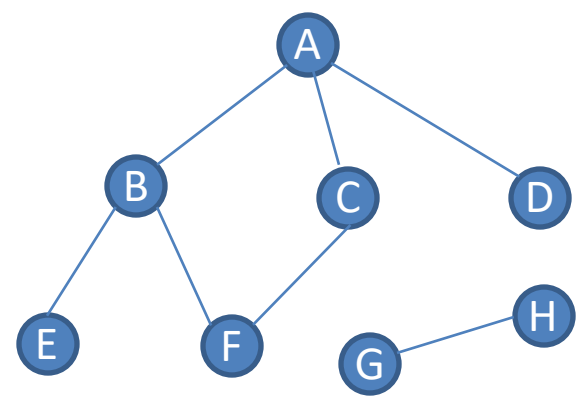
- DFS (Depth-First Search ) Traversal
- BFS (Breath-First Search) Traversal

Comparing with binary tree travel

- pre-order: DFS
- level-order: BFS

*How to apply DFS/BFS for graph traversal?*

# Breath-First-Search Traversal



- need to keep track of whether a node has been visited to avoid duplicate visits.
- need to call BFS repeatedly on all nodes to cover all connected components.

Q: List the nodes in order of visited by BFS. *On tie choose the smallest element.*

? A

Complexity=

- using Adj Lists:  $O(?) \theta(?)$
- using Adj matrix:  $O(?) \theta(?)$
- what if graph is sparse/dense?

	adj list	adj matrix
sparse	$\theta ( \ )$	$\theta ( \ )$
dense	$\theta ( \ )$	$\theta ( \ )$

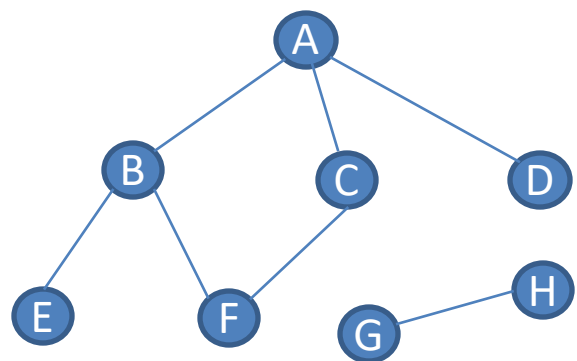
BFS Traversal:

```
order= 0; // order represents timestamp
for (v=0; v<V; v++)
    visited[v] = 0; //mark as "unvisited"
for (v=0; v<V; v++)
    if (!visited(v)) BFS(v);
```

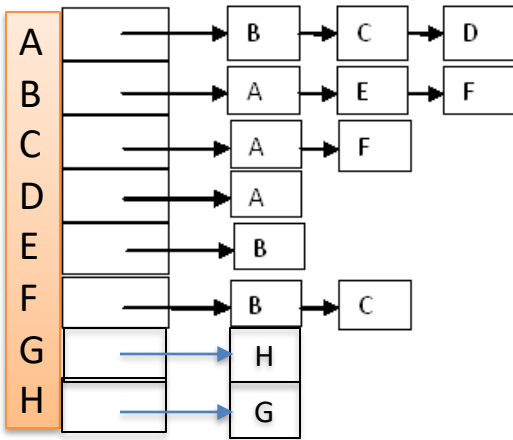
function BFS(int u) {  
  Q= empty queue  
  enQ(Q, u)  
  while (Q not empty) {  
    u = deQ(Q)  
    if (!visited[u]) {  
      visited[u]= ++order; // visit u, and mark as visited  
      for each neighbor v of u  
        if (!visited[v]) enQ(Q,v)  
    }  
  }  
}

COMP20003.W10.avo 18

# Depth-First Traversal: using recursion



Adjacency-List Array



	adj list	adj matrix
sparse	$\theta ( \quad )$	$\theta ( \quad )$
dense	$\theta ( \quad )$	$\theta ( \quad )$

```
DFS Traversal:
order= 0;
for (v=0; v<V; v++)
    visited[v] = 0; //mark as "unvisited"
for (v=0; v<V; v++)
    if (!visited(v)) DFS(v);
```

function DFS(int u) {

```
    visited[u]= ++order;
    for each neighbor v of u
        if (!visited[v])
            DFS(v);
```

The Visit

}

Q: List the nodes in order of visited by DFS.  
? A

Other questions:

- Complexity in comparison with BFS traversal?

# FEIT Small Class Surveys

The surveys were sent to you from "[no-reply@unimelb.edu.au](mailto:no-reply@unimelb.edu.au)" with the subject "**FEIT Subject Surveys**".

Please complete them if you haven't done so.

# The Milk Pouring Problem & Implicit Graphs

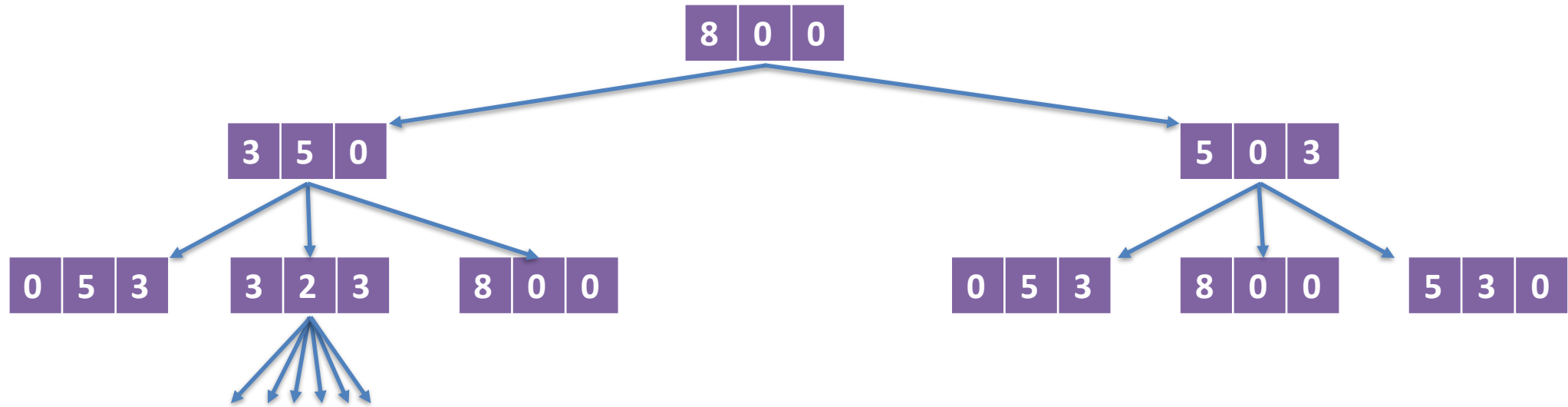
You are provided with three cans of varying capacities, denoted as A, B, and C. The objective is to measure precisely one litre of milk using these three cans, given that:

- Each has an integer capacity greater than 1.
- Initially, the first can is filled with milk while the others remain empty.
- The only permissible action is to pour milk from one can to another until the source can is empty or the destination can is full.

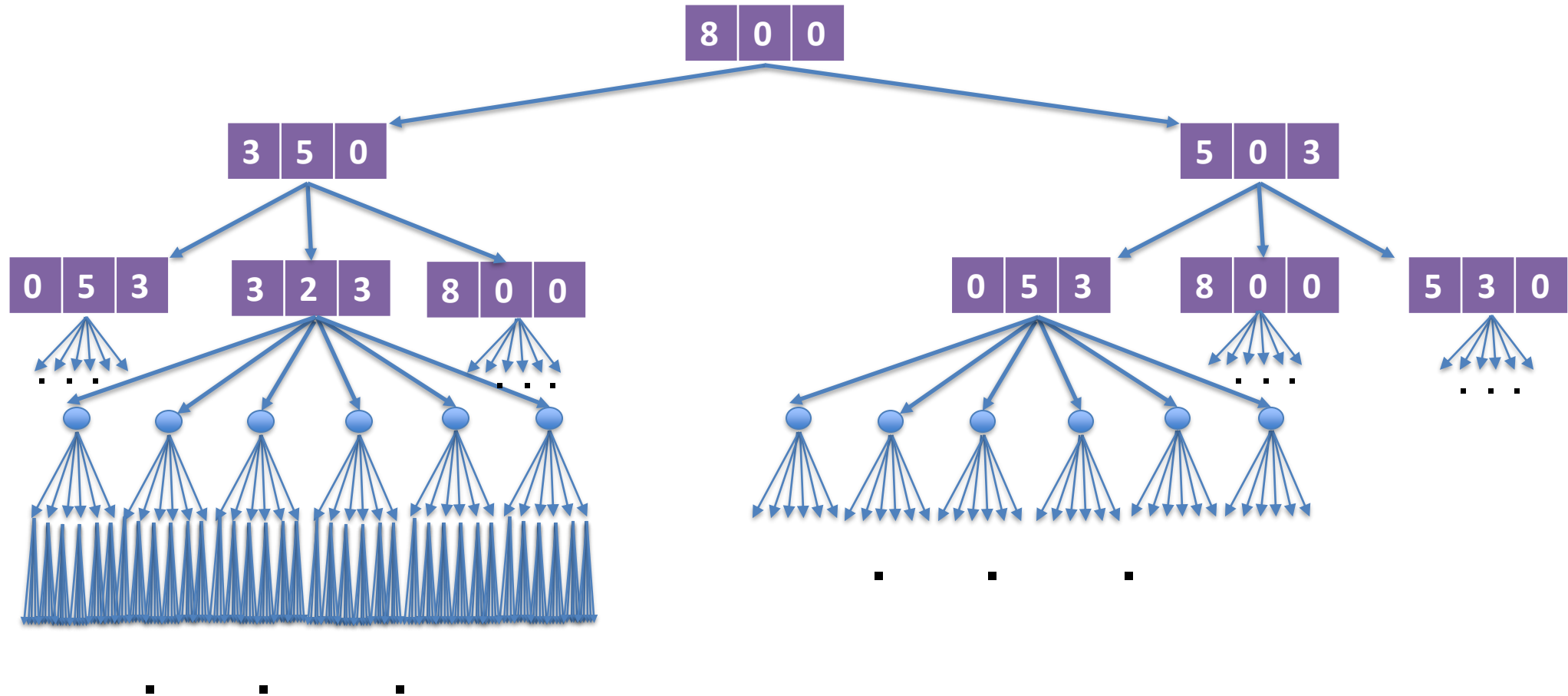
One classical example of this task involves given capacities for cans A, B, and C, with values of 8, 5, and 3, respectively.

*How to solve the problem algorithmically?*

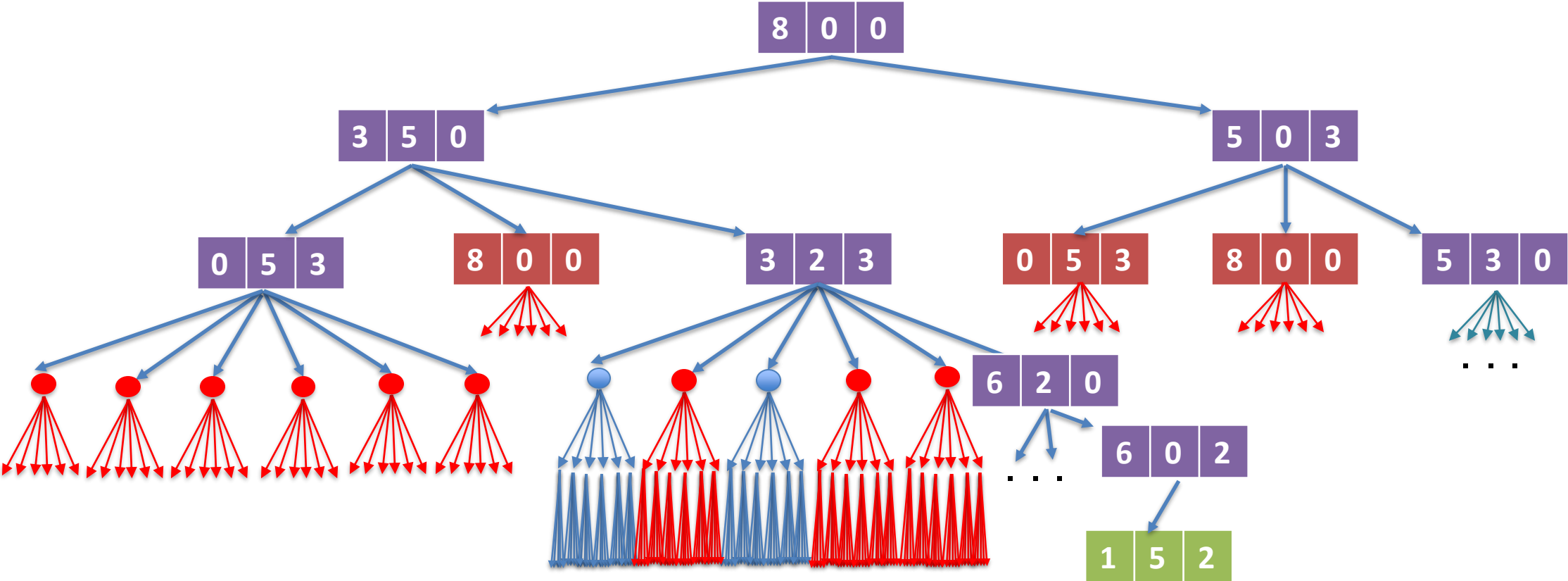
# milk pouring A=8, B=5, C=3



# milk pouring A=8, B=5, C=3



milk pouring  $A=8$ ,  $B=5$ ,  $C=3$ : optimization is important!





# The Milk Pouring Problem & Implicit Graphs

Now, let's consider a scenario where the capacities are parameters  $A$ ,  $B$ ,  $C$ . Your task is to:

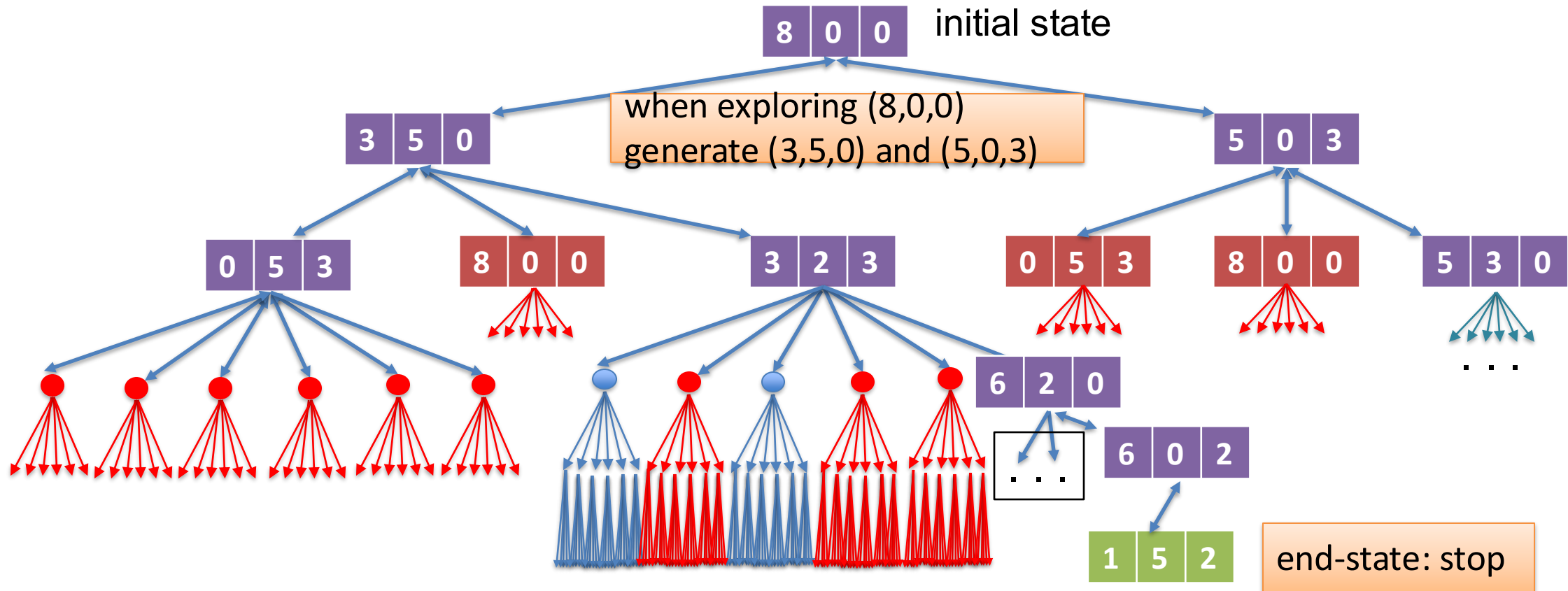
- a) Formulate the task using graphs.
- b) Provide a brief description of an algorithm for solving the reformulated task. For simplicity, assume that a solution does exist.
- c) Suppose that  $d$  is the number of pourings in the solution, express the algorithm's complexity as a function of  $d$ .

in this task, a tuple 

a	b	c
---	---	---

 fully describes the 1 configuration. It's called a state of the task. A state can be seen as a node in a graph.

the edge from node A to node B represent a possible transformation from A to B using a valid pouring action.



The graph is implicit because we do not store all nodes and edges at the start. We instead do BFS by:

- generating the initial node and enqueue it, then do a loop while the queue is not empty:
  - dequeue a state this state to generate all possible next states
  - if a next state is an end state (like the **green**): end the algorithm
  - otherwise if **it haven't been seen before (blue)**: enqueue it

- Start with the peer activity W10.10
- Do W10.1- W10.6
- Programming: W10.7

## **Assignment 3:**

- Read/Try before next Workshop
- Aim to Finish during next week

# Games & Implicit Graphs: The Sokoban

A process of playing a game can be viewed as a graph:

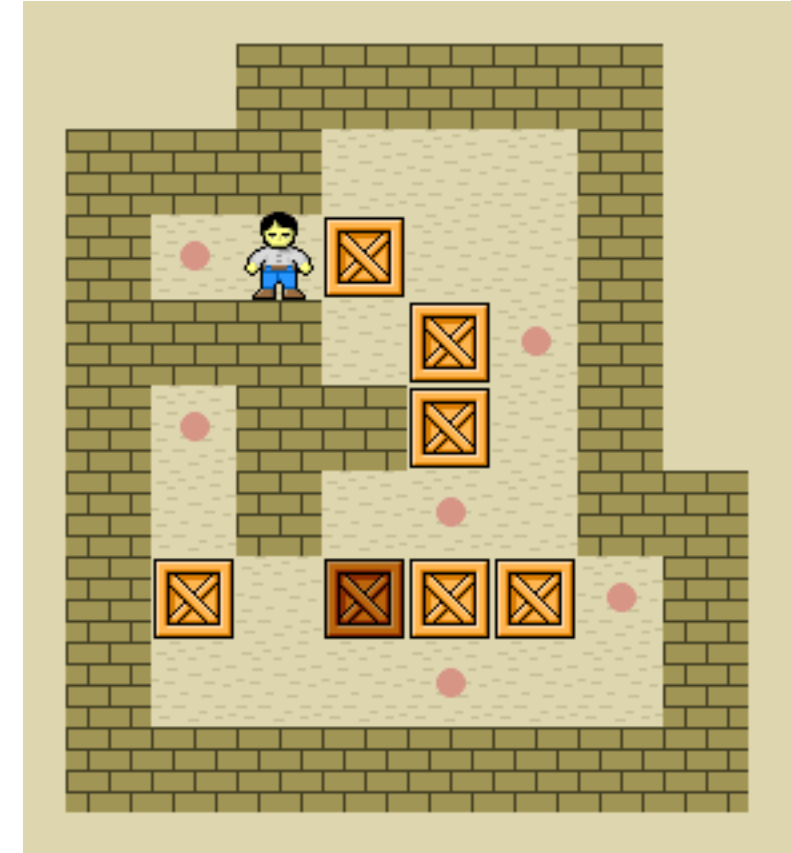
- a particular state of the game is a node
- a move or possible move is an edge, which transfer a node to a new node

Such graph potentially has a huge and perhaps unknown size, but we can just generate the nodes on the spot. It is an *implicit graph*.

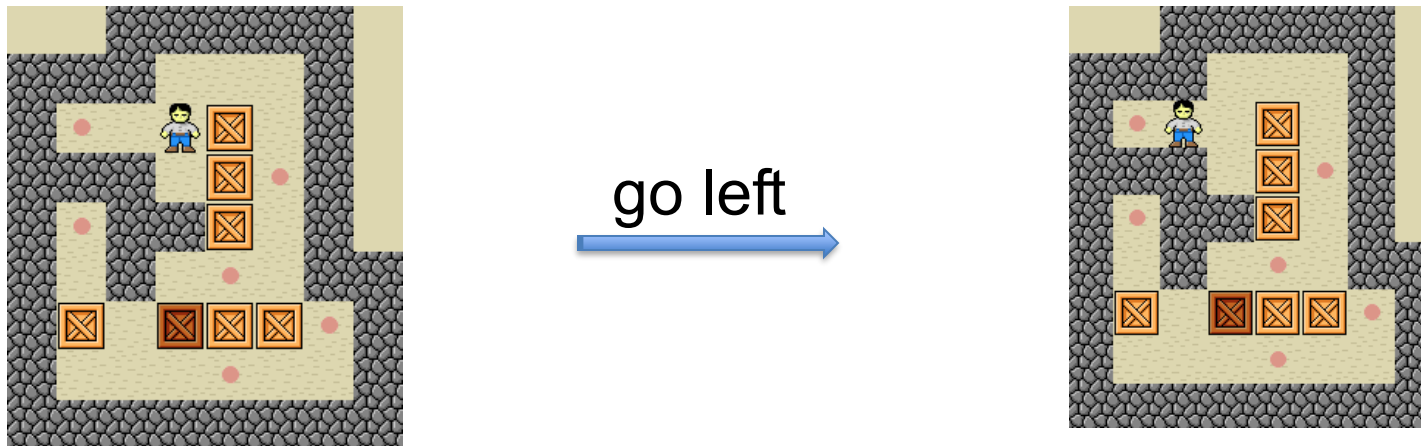
AI program builds the implicit graph move-by-move, and finds a shortest path from the initial state (node) to a winning state (node).

Examples:

- searching for a solution to a puzzle such as Rubik's Cube
- searching for a solution to the Sokoban game

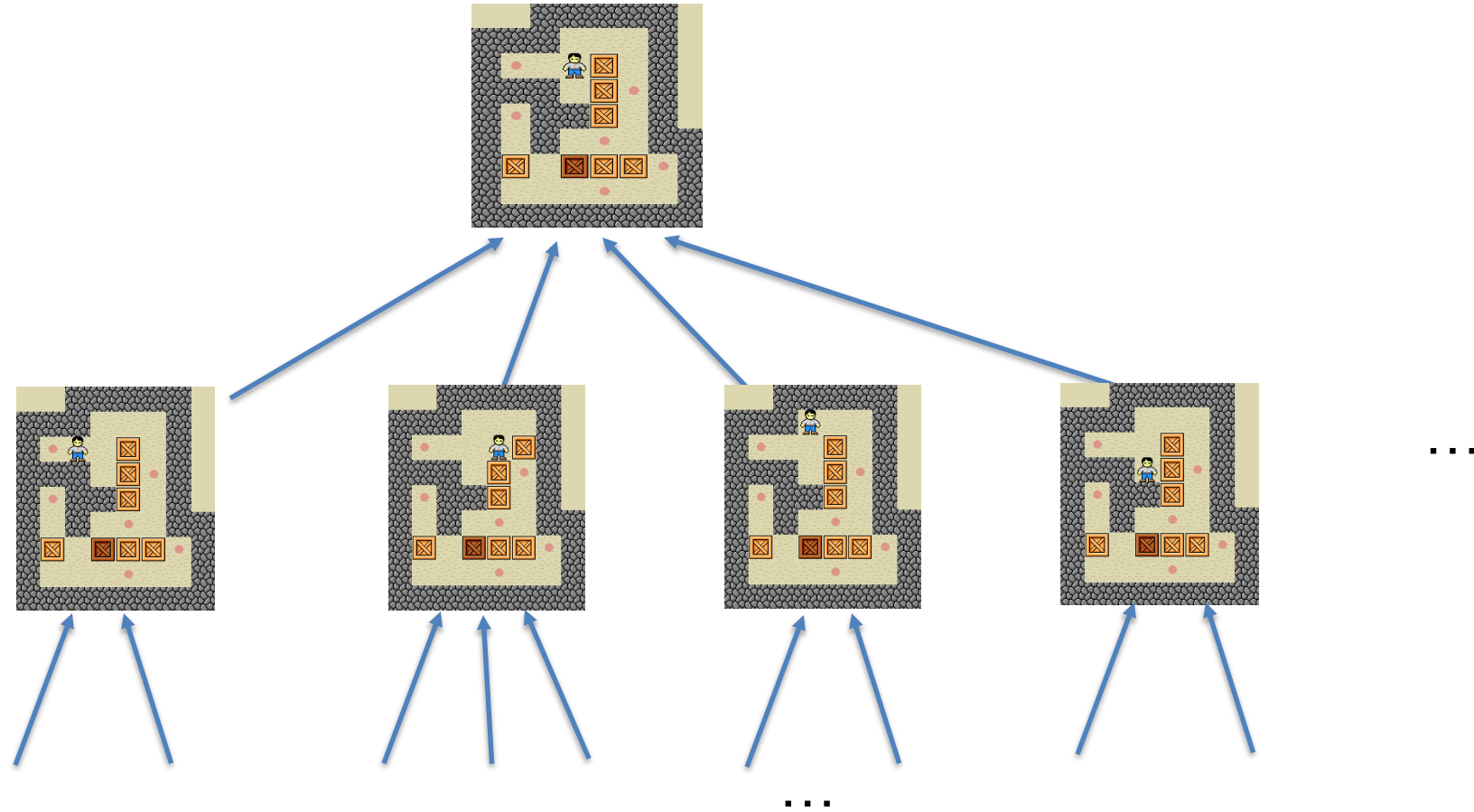


# Games & Implicit Graphs



We represent the game implicitly as a graph:

- A particular configuration of the game board is called a *state*
- When a move is performed, the board goes from one state to another state
- So: **A state is a vertex, and a move is an edge of the graph**



We look at the state and generate all possible actions from that state, then turn into edges.  
With implicit graphs we usually **don't** generate the whole graph.  
We just need to find a shortest path from the root to a the winning node.  
Use BFS !



# FEIT Small Class Surveys

The surveys were sent to you from "[no-reply@unimelb.edu.au](mailto:no-reply@unimelb.edu.au)" with the subject "**FEIT Subject Surveys**".

Please complete them if you haven't done so.