CSCE 4263/5183
Advanced Data Structures

Lecture 18

Graph Traversals

DFS & BFS

Fall 2025

Prof. Khoa Luu
khoaluu@uark.edu

Major Topics In This Course

1. Introduction
2. Reviews (Link List, OOP, Binary Tree, BT Search)
3. Self-balancing Binary Search Tree (AVL, Multiway Search, Red-Black)
4. Splay Tree
5. Balanced Search Tree Review
6. Heap Methods
7. Hashing Methods
9. Graph Data Structures
10. Graph CNN
11. Data Structures in Deep Learning
12. Final Project Presentations

Major Topics In This Course

1. Introduction
2. Reviews (Link List, OOP, Binary Tree, BT Search)
3. Self-balancing Binary Search Tree (AVL, Multiway Search, Red-Black)
4. Splay Tree
5. Balanced Search Tree Review
6. Heap Methods
7. Hashing Methods
9. Graph Data Structures
10. Graph CNN
11. Data Structures in Deep Learning
12. Final Project Presentations

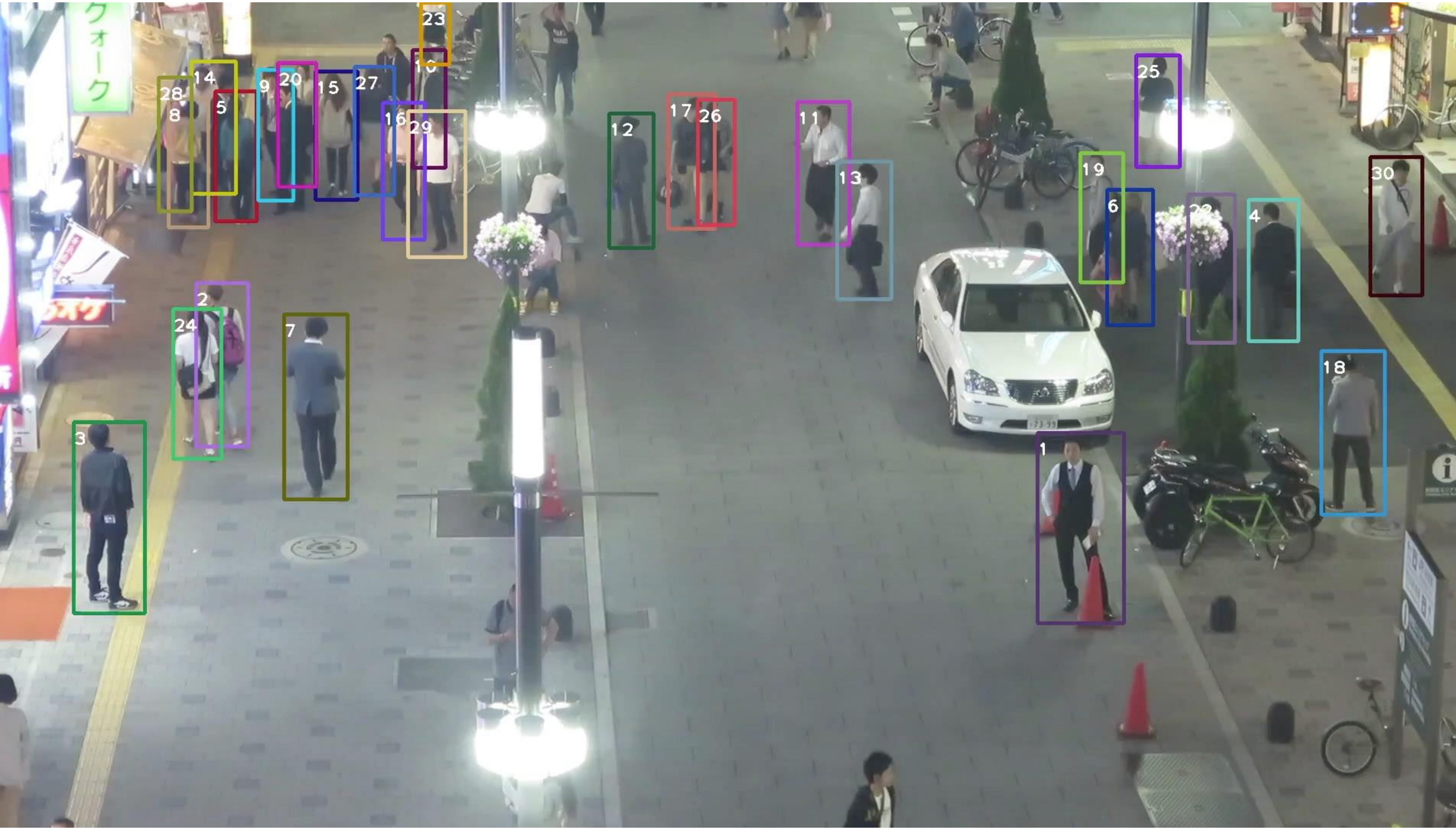
Outline

We will look at DFS & BFS in graphs

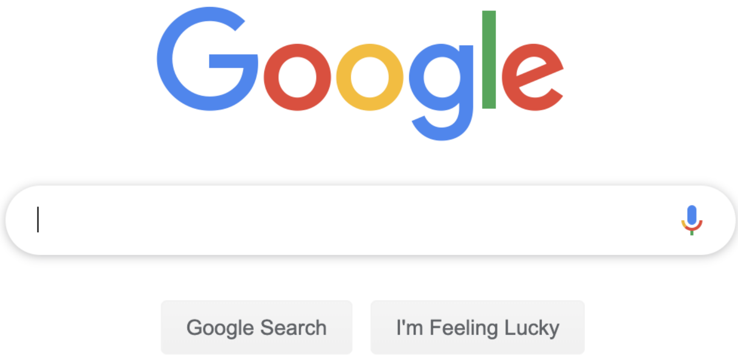
- Graph Data Structures
- Graph Improvements
- Depth-first traversals
- Breath-first traversals

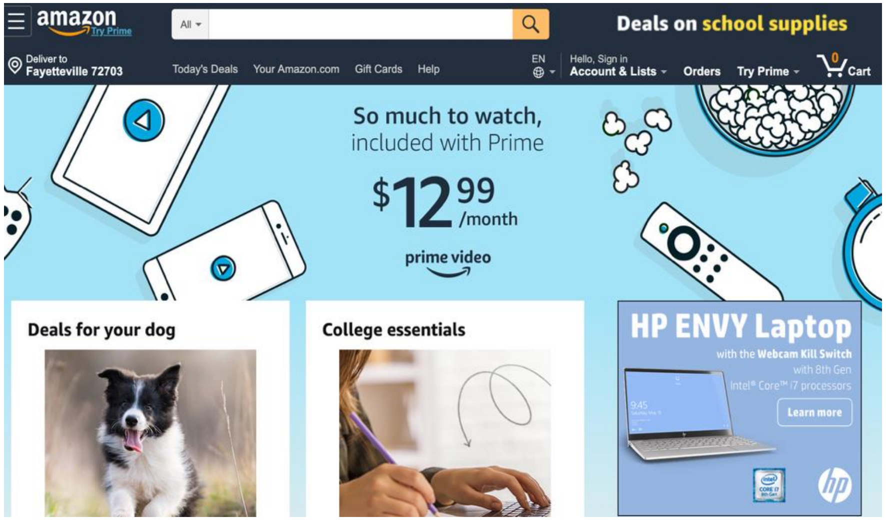
Reading

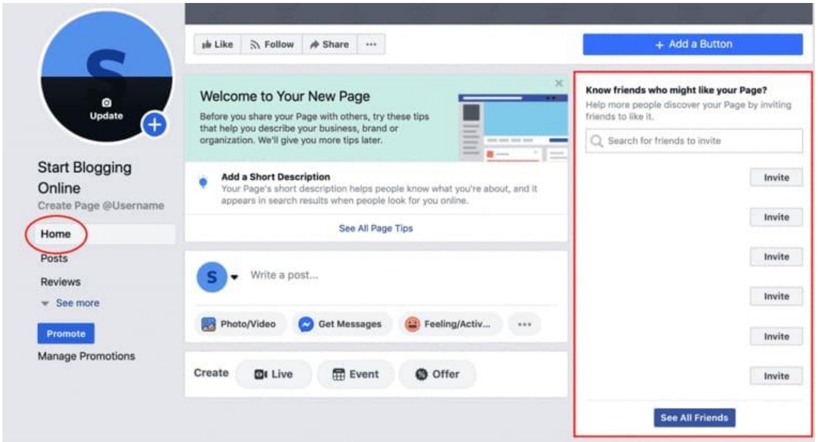
Weiss, Data Structures and Algorithm Analysis in C++, 3rd Ed., Chapters: 9


ウォーク

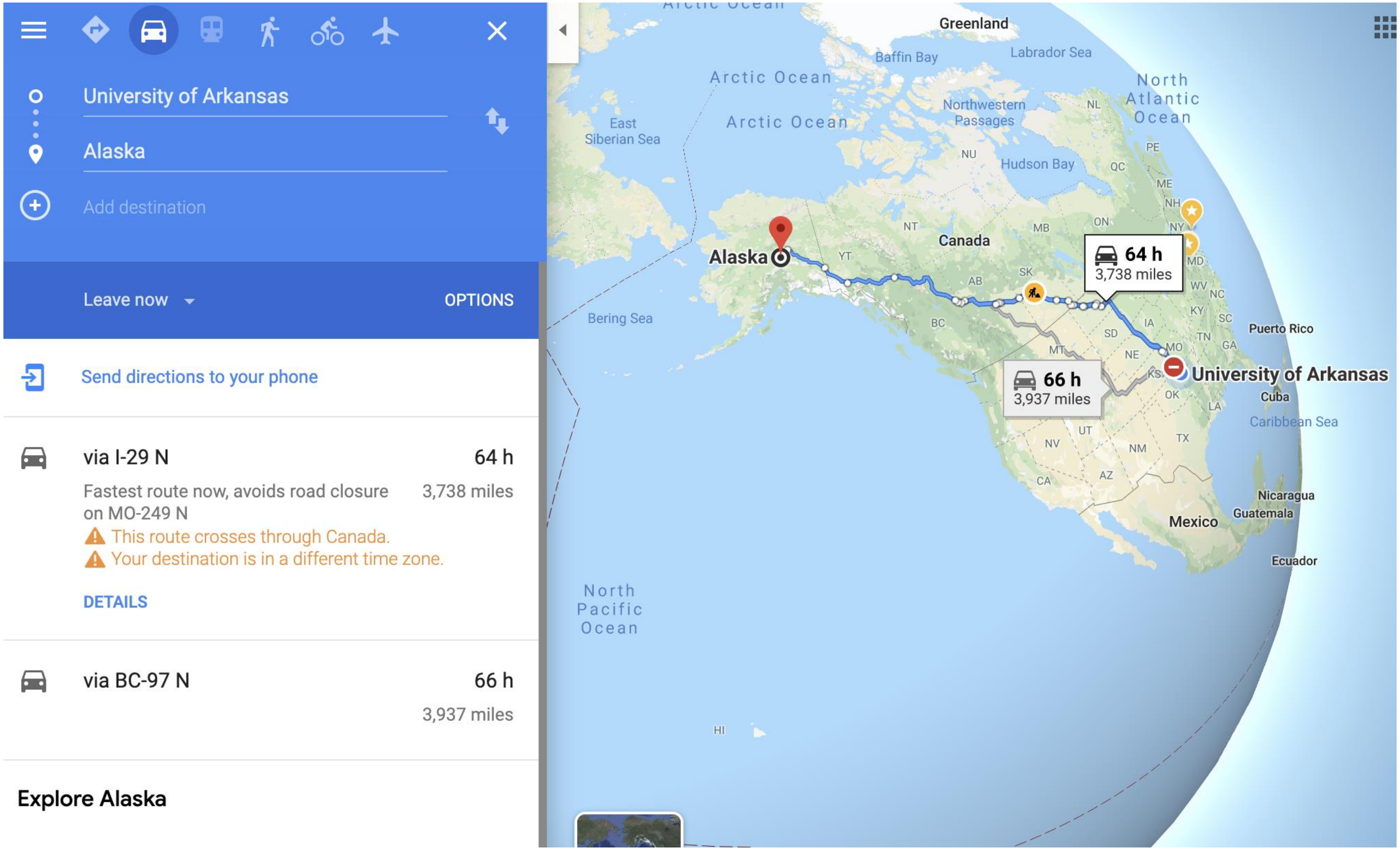
Search - Goal Node







Search – Path to a Goal



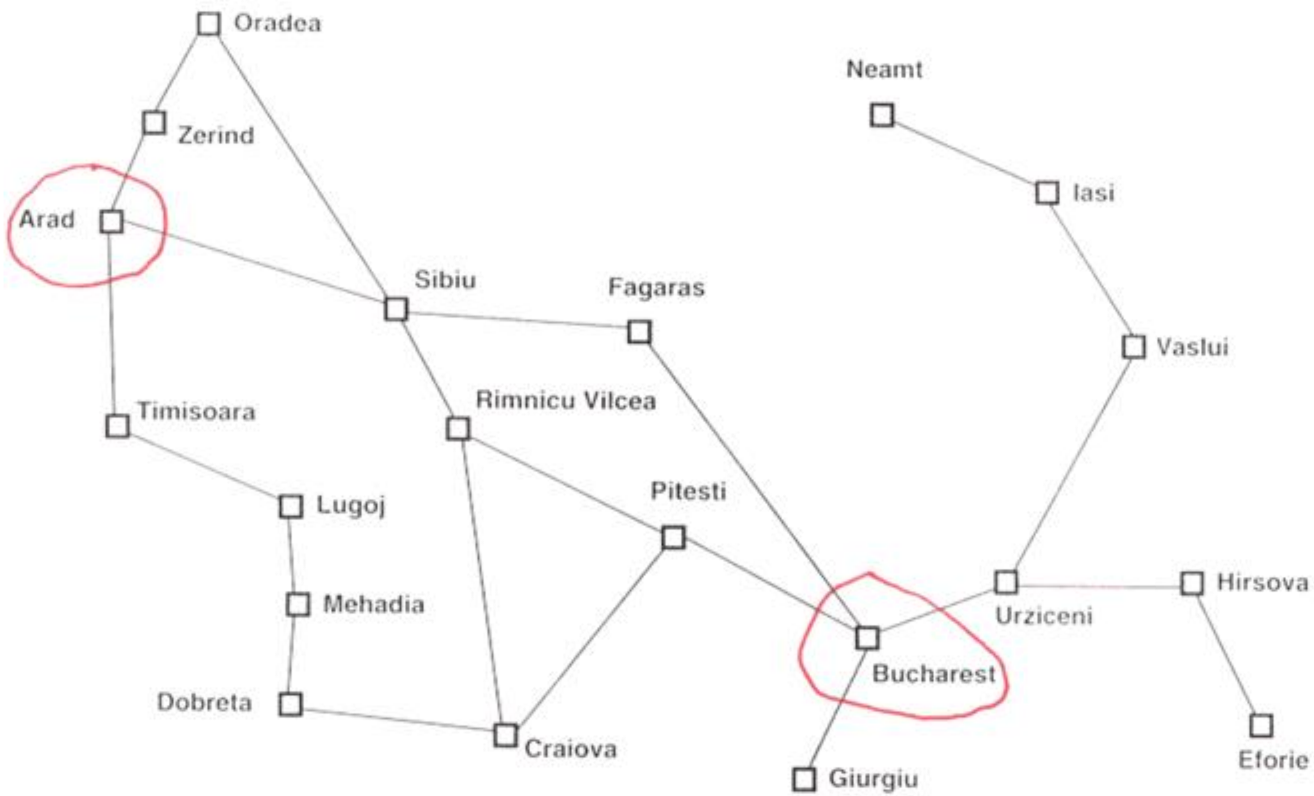
Search

- Goal-based agent
- In some applications we are interested in finding a goal node, in other applications we are interested in finding a path to a goal

No map vs. Map

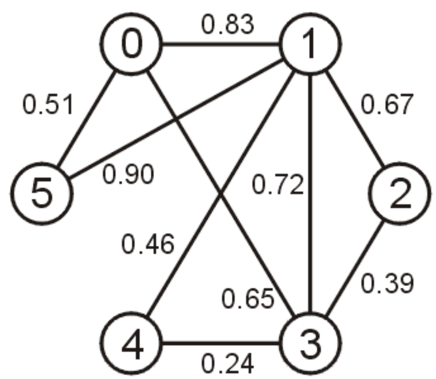
physical search

deliberative search



Background

- To demonstrate these techniques, we will look at storing the edges of the following graph:



Adjacency Matrix

A graph of n vertices may have up to ? edges

Adjacency Matrix

A graph of n vertices may have up to

$$\binom{n}{2} = \frac{n(n-1)}{2} = \mathbf{O}(n^2)$$

edges

The first straight-forward implementation is an adjacency matrix

Adjacency Matrix

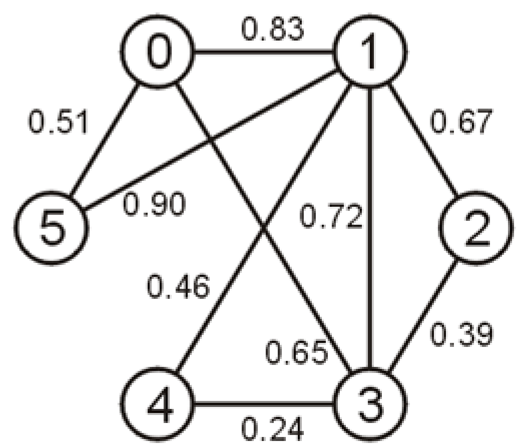
Define an $n \times n$ matrix $\mathbf{A} = (a_{ij})$ and if the vertices v_i and v_j are connected with weight w, then set $a_{ij} = w$ and $a_{ji} = w$

Adjacency Matrix

Define an $n \times n$ matrix $\mathbf{A} = (a_{ij})$ and if the vertices v_i and v_j are connected with weight w, then set $a_{ij} = w$ and $a_{ji} = w$

That is, the matrix is symmetric, *e.g.*,

	0	1	2	3	4	5
0		0.83		0.65		0.51
1	0.83		0.67	0.72	0.46	0.90
2		0.67		0.39		
3	0.65	0.72	0.39		0.24	
4		0.46		0.24		
5	0.51	0.90				



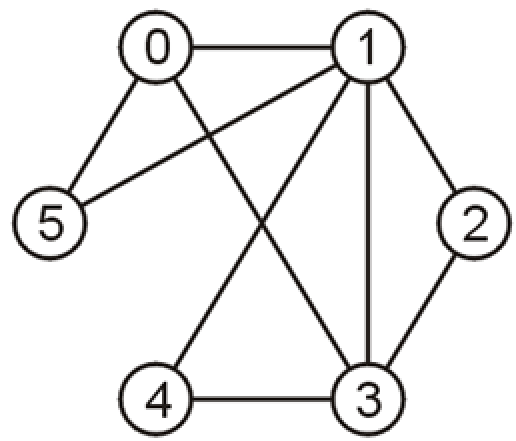
Adjacency Matrix

An unweighted graph may be saved as an array of Boolean values

– vertices v_i and v_j are connected then set

$a_{ij} = a_{ji} = true$

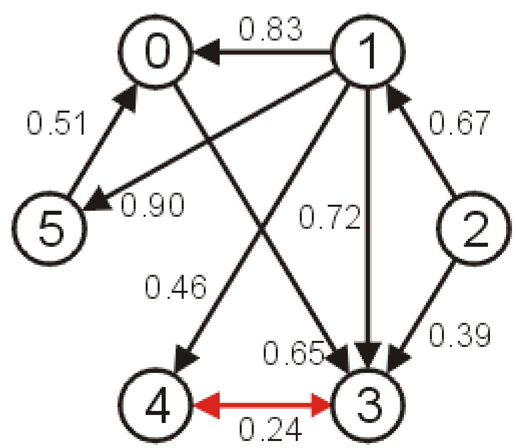
	0	1	2	3	4	5
0		T	F	T	F	T
1	T		T	T	T	T
2	F	T		T	F	F
3	T	T	T		T	F
4	F	T	F	T		F
5	T	T	F	F	F	



Adjacency Matrix

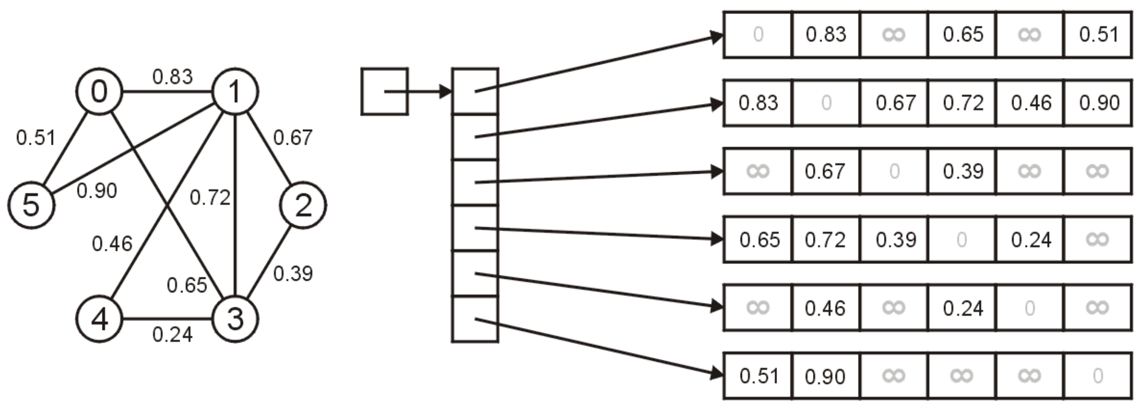
If the graph was directed, then the matrix would not necessarily be symmetric

	0	1	2	3	4	5
0				0.65		
1	0.83			0.72	0.46	0.90
2		0.67		0.39		
3					0.24	
4				0.24		
5	0.51					



Adjacency Matrix

Note, however, that these six arrays could be anywhere in memory...



Adjacency Matrix Improvements

We have now looked at how we can store an adjacency graph in C++

Improvements:

- Two improvements for the array-of-arrays implementations, including:
  - allocating the memory for the matrix in a single contiguous block of code, and
  - a lower-triangular representation; and
- A sparse linked-list implementation

Adjacency Matrix Improvement

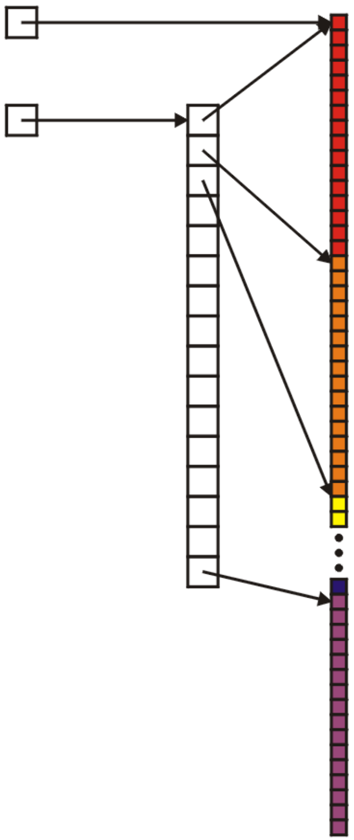
Next, we allocate the addresses:

```
matrix = new double * [16];
double * tmp = new double[256];

for ( int i = 0; i < 16; ++i ) {
   matrix[i] = &( tmp[16*i] );
}
```

This assigns:

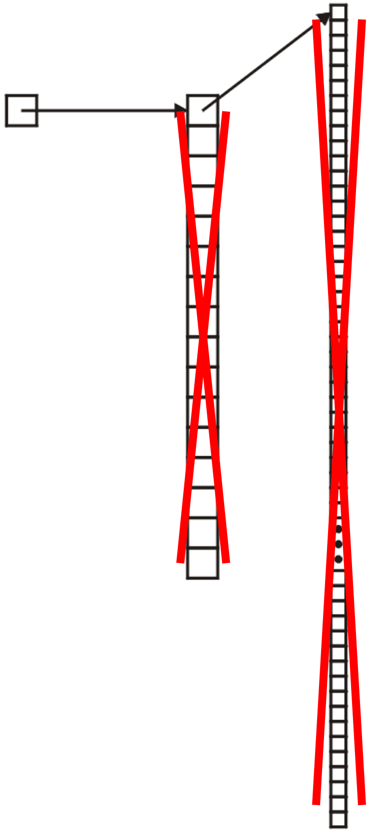
```
matrix[ 0] = &( tmp[  0] );
matrix[ 1] = &( tmp[ 16] );
matrix[ 2] = &( tmp[ 32] );
            .
            .
            .
matrix[15] = &( tmp[240] );
```



Adjacency Matrix Improvement

Deleting this array is easier:

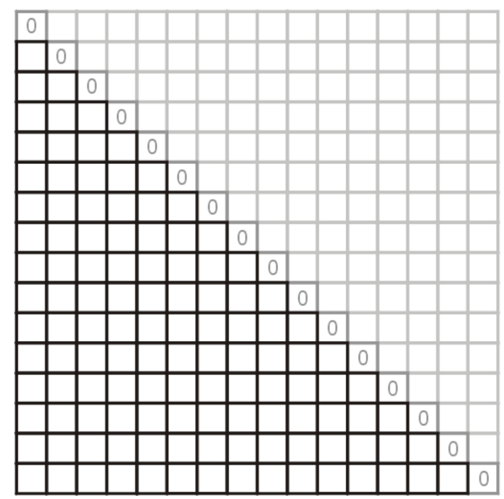
```
delete [] matrix[0];
delete [] matrix;
```



Lower-triangular adjacency matrix

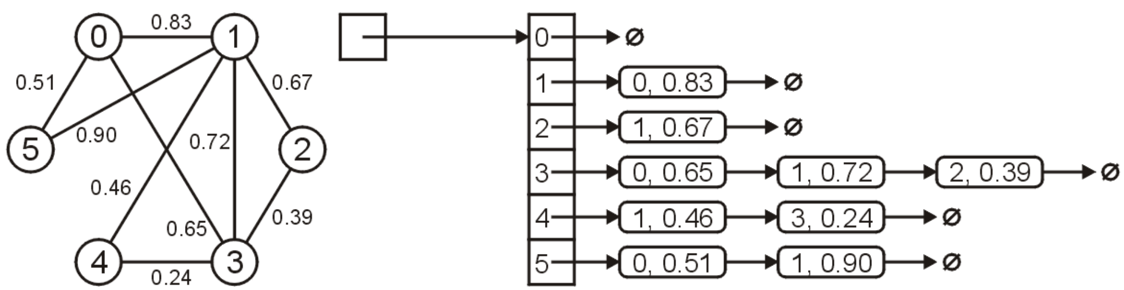
Note also that we are not storing a directed graph: therefore, we really need only store half of the matrix

Thus, instead of 256 entries, we really only require 120 entries



Sparse Matrices

The graph shown below would be stored as



Depth First Search

DFS Algorithm

- Depth-First Search algorithm can be implemented using stack

DFS Algorithm

- Depth-First Search algorithm can be implemented using stack
- A standard DFS implementation puts every vertex of the graph into one in all 2 categories: **Visited** or **Not Visited**.

DFS Algorithm

- Depth-First Search algorithm can be implemented using stack
- A standard DFS implementation puts every vertex of the graph into one in all 2 categories: **Visited** or **Not Visited**.
- This algorithm is to visit all the vertex of the graph.

DFS Algorithm

- Depth-First Search algorithm can be implemented using stack
- A standard DFS implementation puts every vertex of the graph into one in all 2 categories: **Visited** or **Not Visited**.
- This algorithm is to visit all the vertex of the graph.
- It avoids cycles.

DFS Algorithm

The DFS algorithm follows as (5 steps):

DFS Algorithm

The DFS algorithm follows as (5 steps):

1. Start by putting any one of the graph's vertex on top of the stack

DFS Algorithm

The DFS algorithm follows as (5 steps):

1. Start by putting any one of the graph's vertex on top of the stack
2. Take the top item of the stack and add it to the visited list of the vertex

DFS Algorithm

The DFS algorithm follows as (5 steps):

1. Start by putting any one of the graph's vertex on top of the stack
2. Take the top item of the stack and add it to the visited list of the vertex
3. Create a list of that adjacent node of the vertex

DFS Algorithm

The DFS algorithm follows as (5 steps):

1. Start by putting any one of the graph's vertex on top of the stack
2. Take the top item of the stack and add it to the visited list of the vertex
3. Create a list of that adjacent node of the vertex
4. The ones which aren't in the visited list of vertexes to the top of the stack.

DFS Algorithm

The DFS algorithm follows as (5 steps):

1. Start by putting any one of the graph's vertex on top of the stack
2. Take the top item of the stack and add it to the visited list of the vertex
3. Create a list of that adjacent node of the vertex
4. The ones which aren't in the visited list of vertexes to the top of the stack.
5. Keep repeating steps 2 and 3 until the stack is empty

DFS Pseudocode (stack implementation)

```
DFS(G,v) ( v is the vertex where the search starts )
    Stack S := {}; ( start with an empty stack )
    for each vertex u, set visited[u] := false;
    push S, v;
    while (S is not empty) do
            u := pop S;
            if (not visited[u]) then
                visited[u] := true;
                for each unvisited neighbor w of u
                        push S, w;
            end if
    end while
END DFS()
```

DFS Pseudocode (recursive implementation)

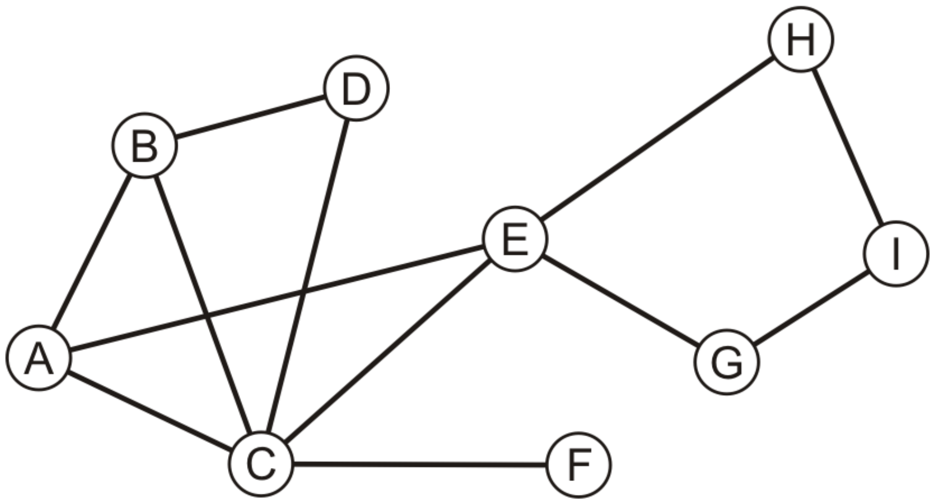
```
DFS(G, u)
{
    u.visited = true
    for each v ∈ G.Adj[u]
        if v.visited == false
            DFS(G,v)
}

init()
{
    for each u ∈ G
        u.visited = false

    for each u ∈ G
       DFS(G, u)
}
```

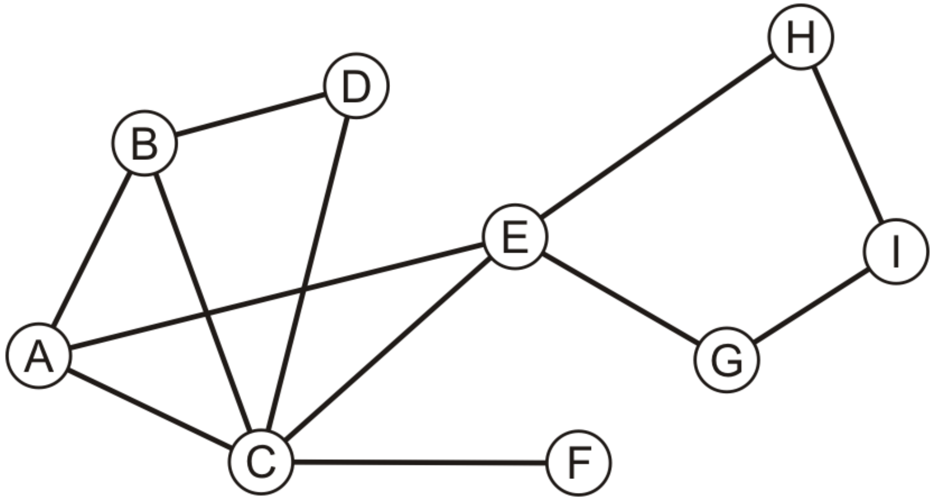
Example - DFS

Consider this graph



Example - DFS

Perform a recursive depth-first traversal on this graph

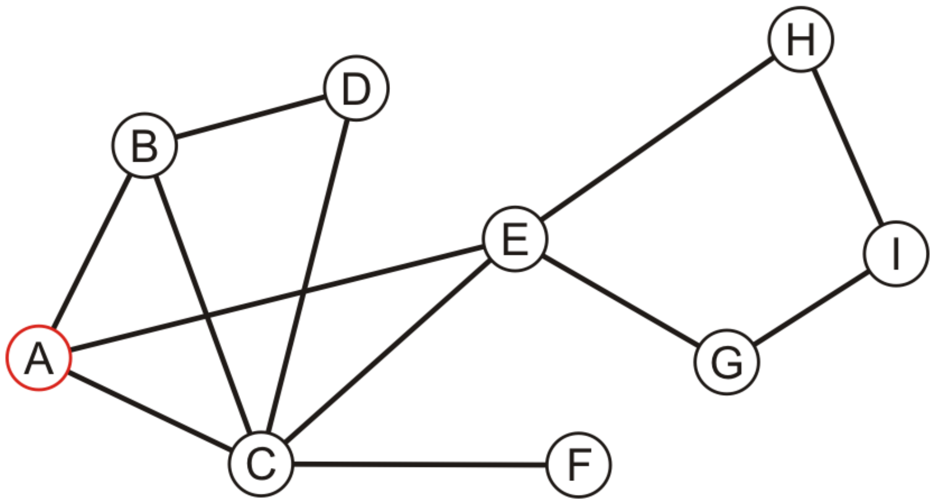


Example - DFS

Performing a recursive depth-first traversal:

– Visit the first node

A

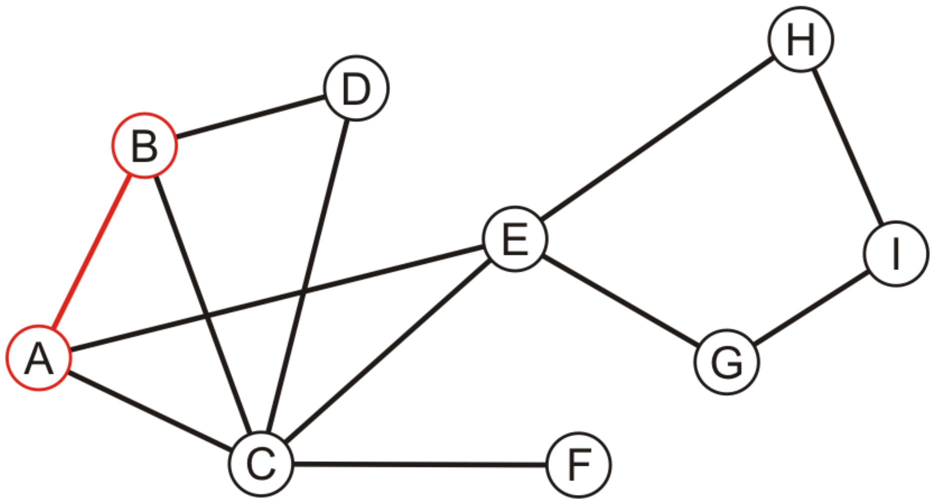


Example - DFS

Performing a recursive depth-first traversal:

– A has an unvisited neighbor

A, B

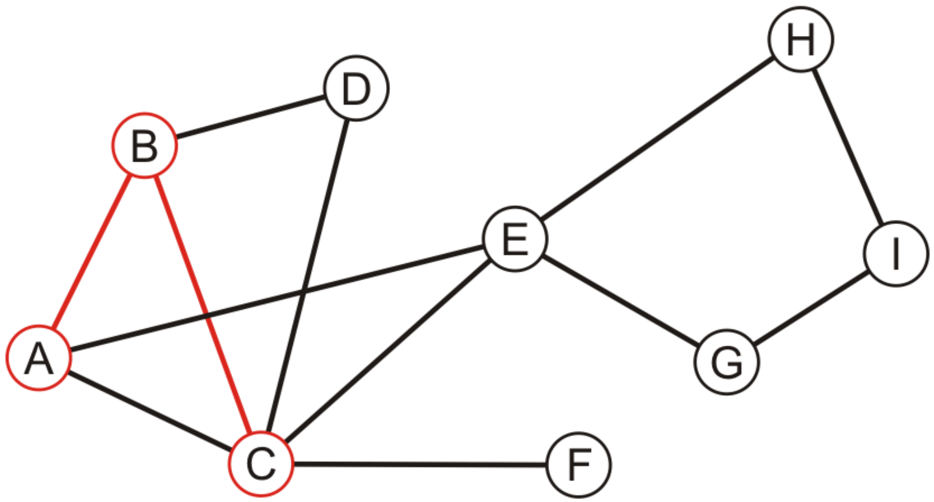


Example - DFS

Performing a recursive depth-first traversal:

– B has an unvisited neighbor

A, B, C

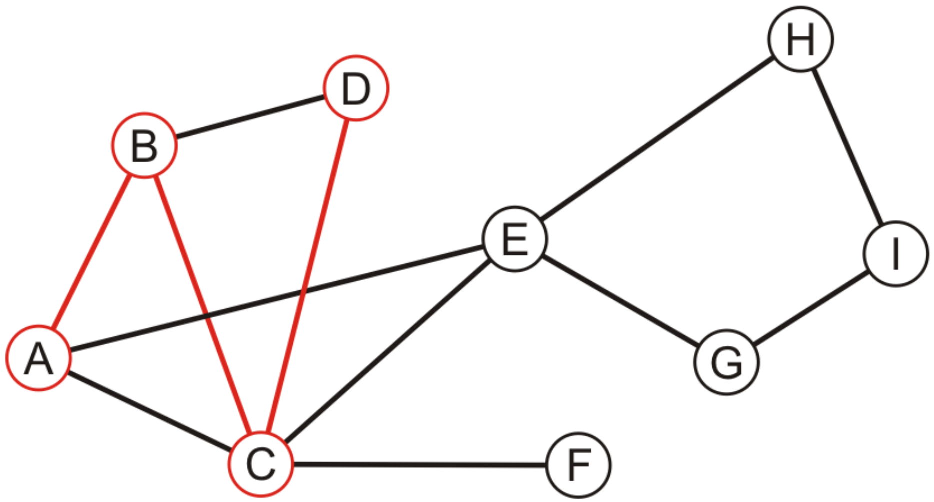


Example - DFS

Performing a recursive depth-first traversal:

– C has an unvisited neighbor

A, B, C, D

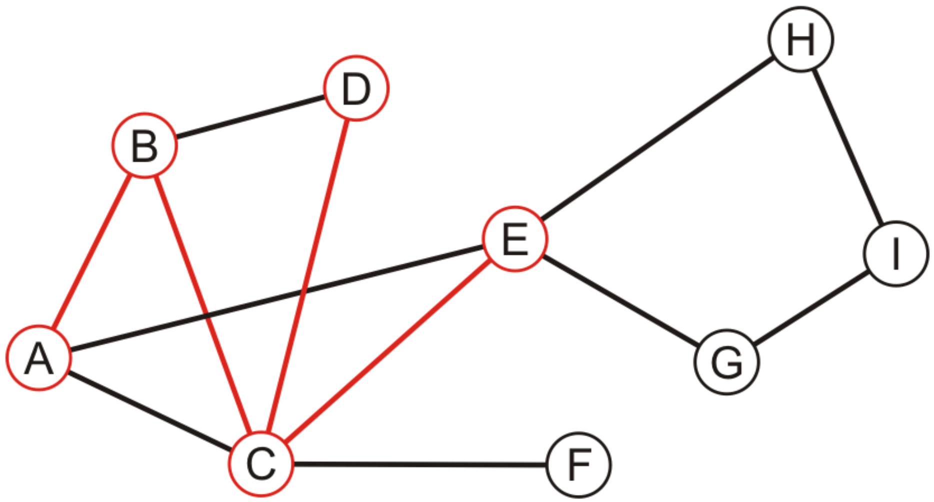


Example - DFS

Performing a recursive depth-first traversal:

– D has no unvisited neighbors, so we return to C

A, B, C, D, E

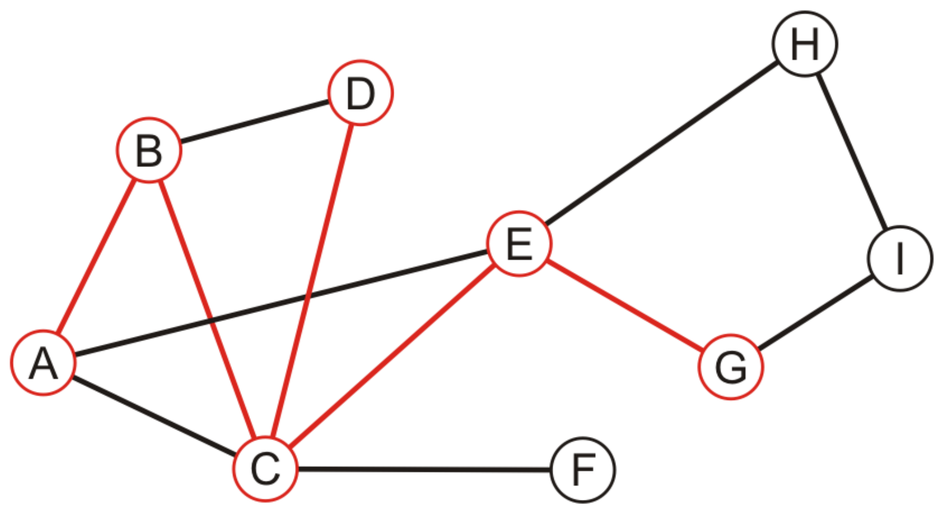


Example - DFS

Performing a recursive depth-first traversal:

– E has an unvisited neighbor

A, B, C, D, E, G

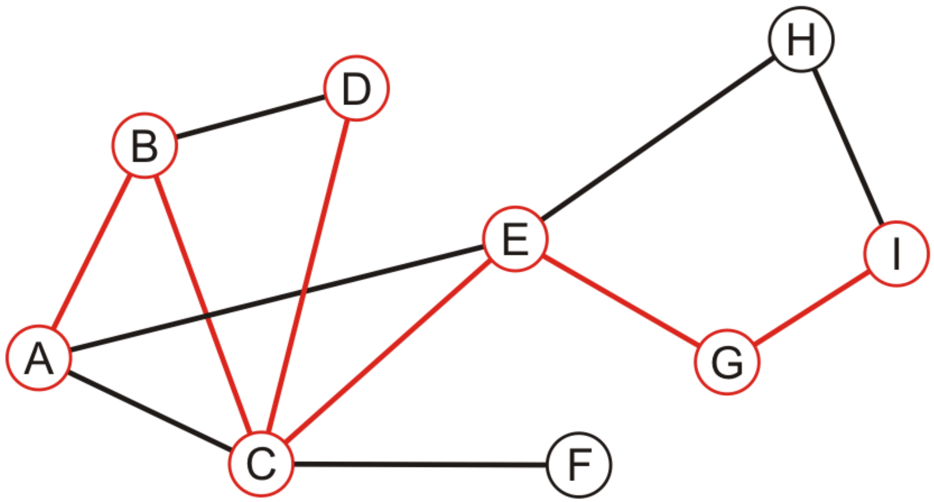


Example - DFS

Performing a recursive depth-first traversal:

– G has an unvisited neighbor

A, B, C, D, E, G, I

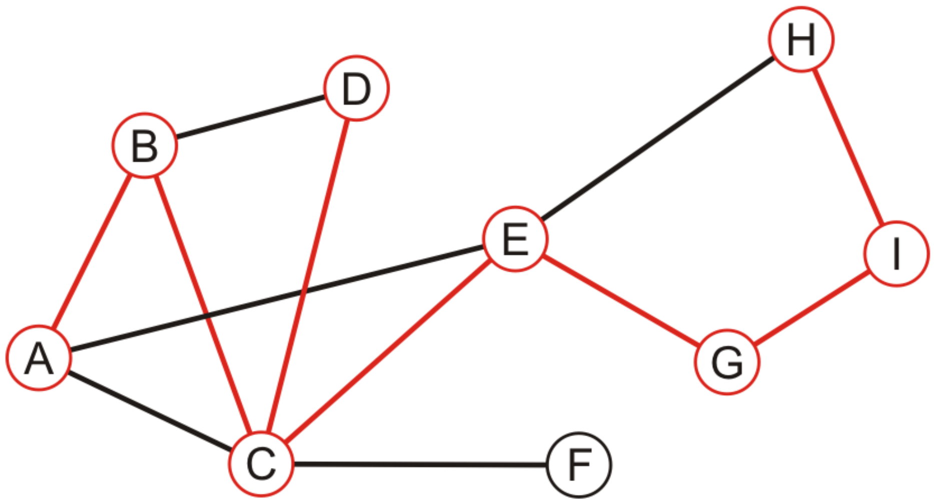


Example - DFS

Performing a recursive depth-first traversal:

- H has no unvisited neighbor

A, B, C, D, E, G, I, H

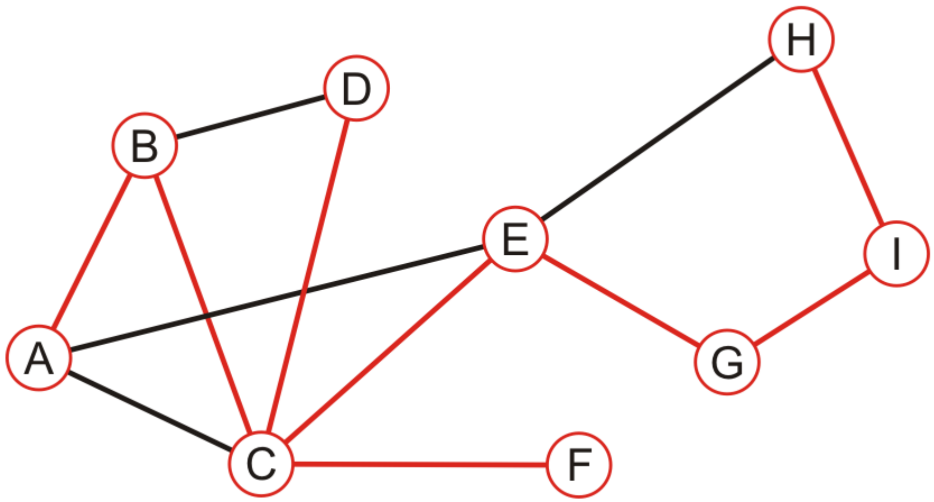


Example - DFS

Performing a recursive depth-first traversal:

– We recurse back to C which has an unvisited neighbor

A, B, C, D, E, G, I, H, F

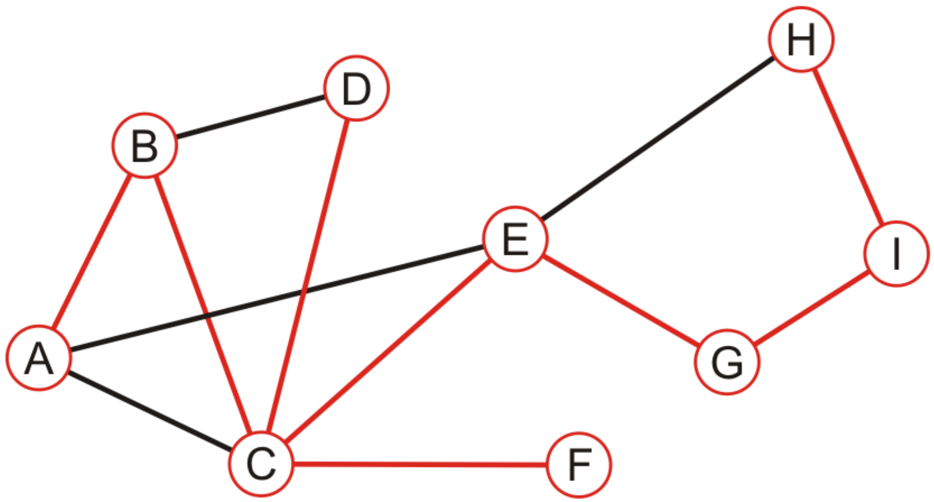


Example - DFS

Performing a recursive depth-first traversal:

- We recurse finding that no other nodes have unvisited neighbors

A, B, C, D, E, G, I, H, F



Breadth First Search

What is Breadth First Search?

- Can be implemented using data structures like a dictionary and lists

What is Breadth First Search?

- Can be implemented using data structures like a dictionary and lists
- Almost the same in tree and graph

What is Breadth First Search?

- Can be implemented using data structures like a dictionary and lists
- Almost the same in tree and graph
- However, the graph may contain cycles, so it may traverse to the same node again

BFS Algorithm

- A standard BFS implementation puts every vertex of the graph into one in all 2 categories: **Visited** or **Not Visited**.
- This algorithm is to visit all the vertex of the graph.
- It avoids cycles.

BFS Algorithm

The BFS algorithm follows as (5 steps):

BFS Algorithm

The BFS algorithm follows as (5 steps):

1. Put any one of the graph's vertices at the back of the queue

BFS Algorithm

The BFS algorithm follows as (5 steps):

1. Put any one of the graph's vertices at the back of the queue
2. Take the front item of the queue and add it to the visited list

BFS Algorithm

The BFS algorithm follows as (5 steps):

1. Put any one of the graph’s vertices at the back of the queue
2. Take the front item of the queue and add it to the visited list
3. Create a list of that vertex's adjacent nodes

BFS Algorithm

The BFS algorithm follows as (5 steps):

1. Put any one of the graph's vertices at the back of the queue
2. Take the front item of the queue and add it to the visited list
3. Create a list of that vertex's adjacent nodes
4. Add those which are not within the visited list to the rear of the queue

BFS Algorithm

The BFS algorithm follows as (5 steps):

1. Put any one of the graph’s vertices at the back of the queue
2. Take the front item of the queue and add it to the visited list
3. Create a list of that vertex's adjacent nodes
4. Add those which are not within the visited list to the rear of the queue
5. Keep continuing steps two and three till the queue is empty.

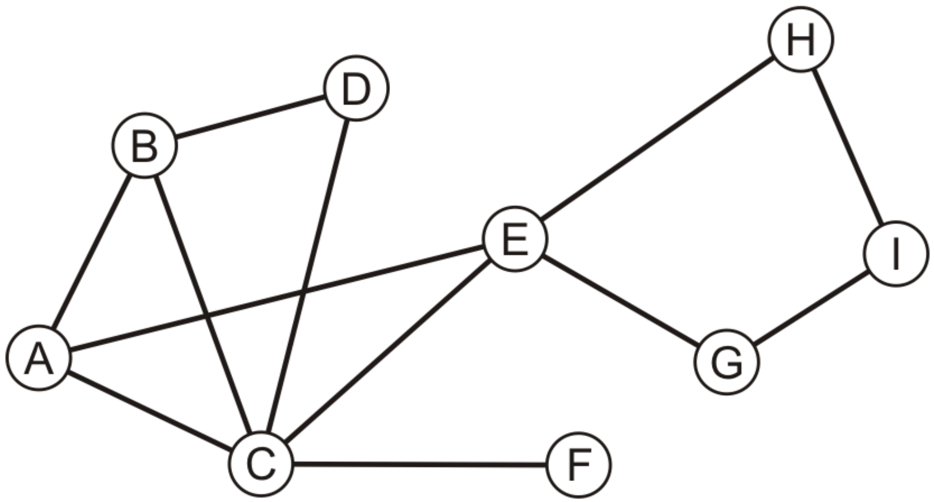
BFS Algorithm

The BFS algorithm follows as:

```
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) neighbors of u
```

Example - BFS

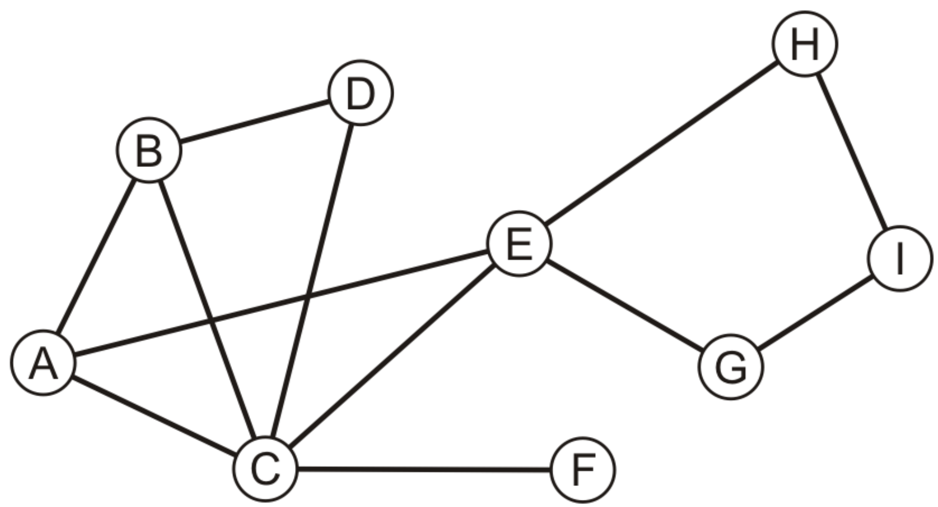
Consider this graph



Example - BFS

Performing a breadth-first traversal

– Push the first vertex onto the queue



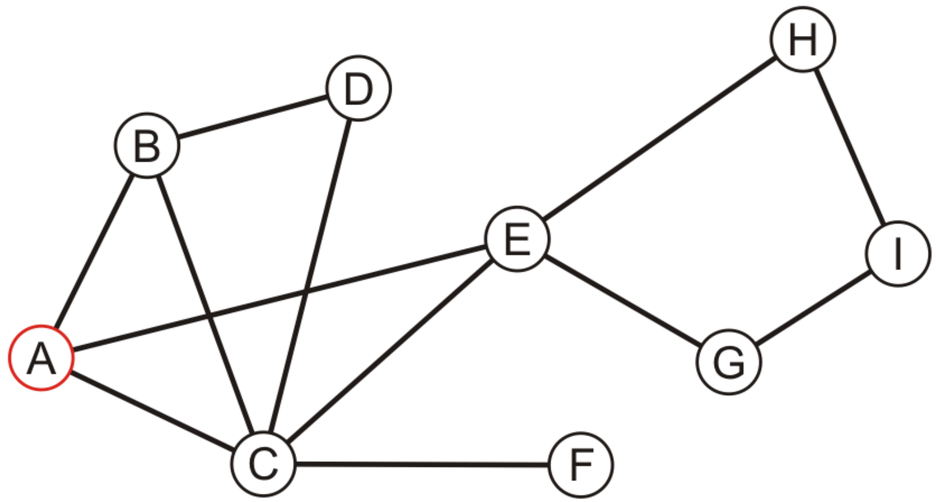
A					

Example - BFS

Performing a breadth-first traversal

– Pop A and push B, C and E

A



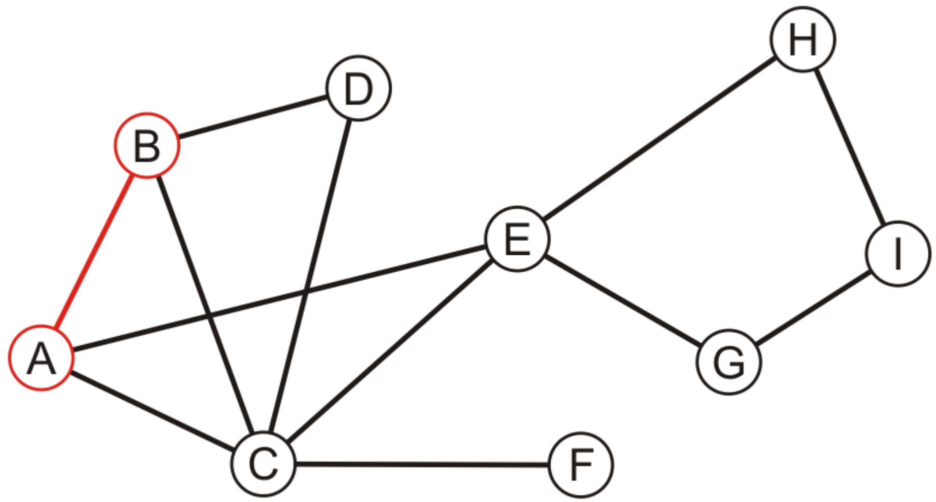
B	C	E			

Example - BFS

Performing a breadth-first traversal:

– Pop B and push D

A, B



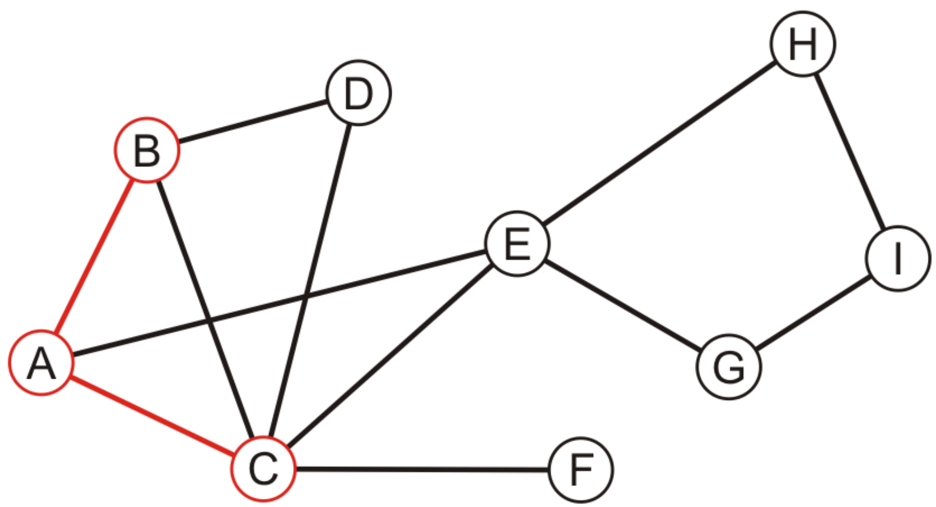
C	E	D			

Example - BFS

Performing a breadth-first traversal:

– Pop C and push F

A, B, C



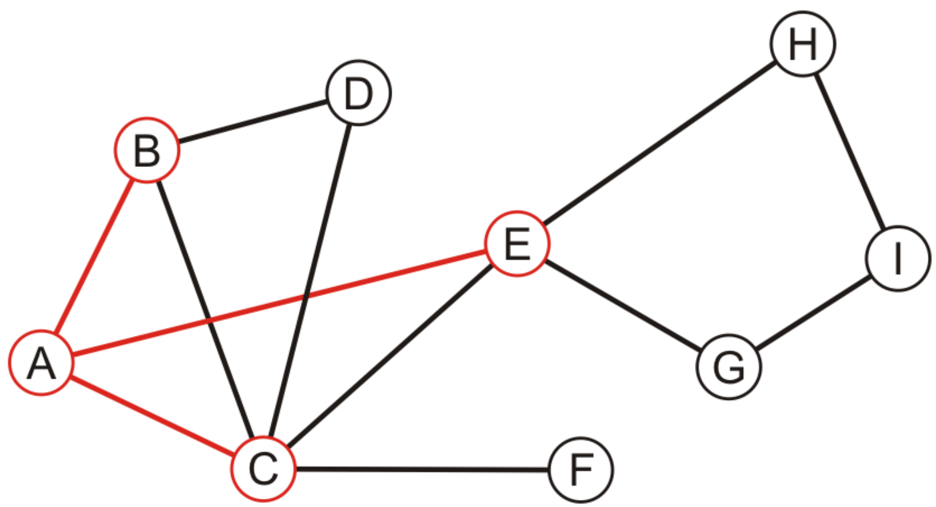
E	D	F			

Example - BFS

Performing a breadth-first traversal:

- Pop E and push G and H

A, B, C, E



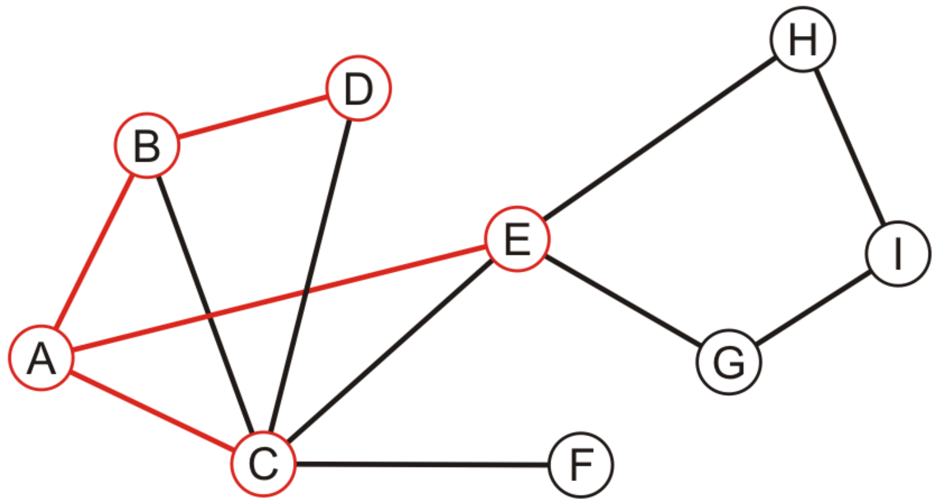
D	F	G	H		

Example - BFS

Performing a breadth-first traversal:

– Pop D

A, B, C, E, D



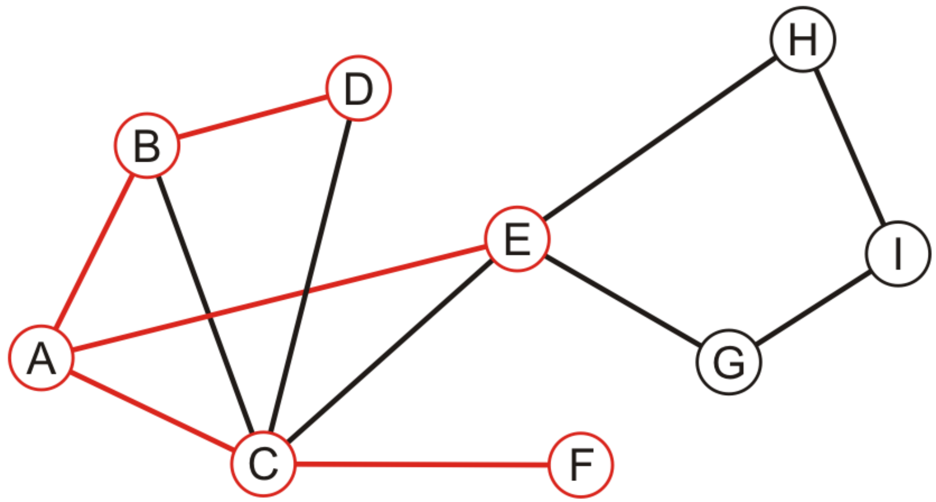
F	G	H			

Example - BFS

Performing a breadth-first traversal:

– Pop F

A, B, C, E, D, F



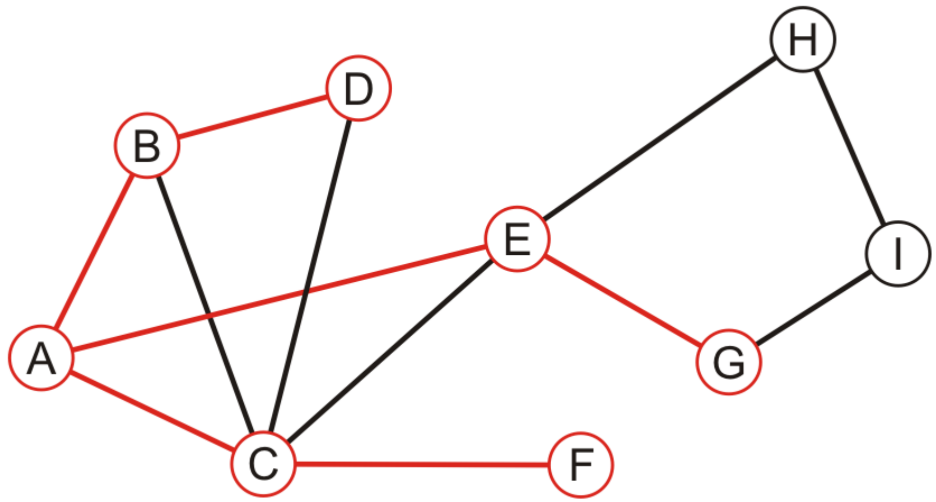
G	H				

Example - BFS

Performing a breadth-first traversal:

– Pop G and push I

A, B, C, E, D, F, G



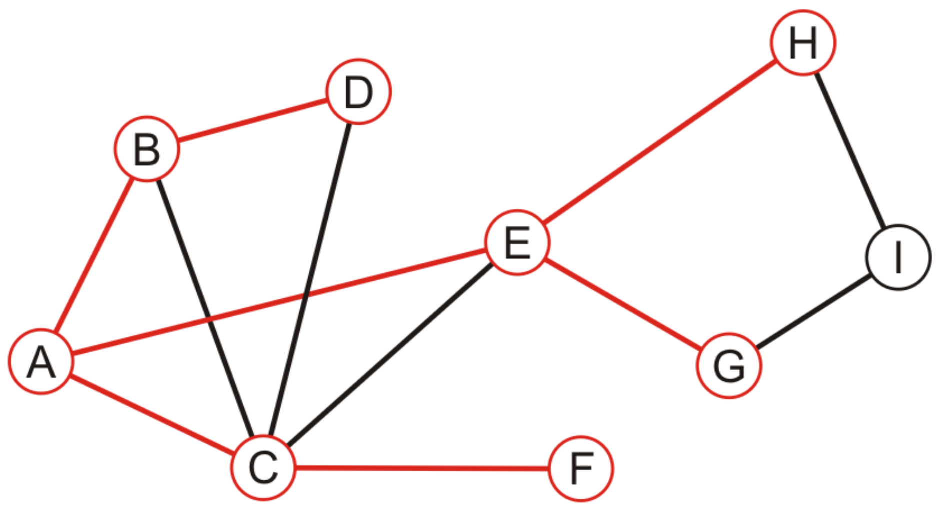
H	I				

Example - BFS

Performing a breadth-first traversal:

– Pop H

A, B, C, E, D, F, G, H



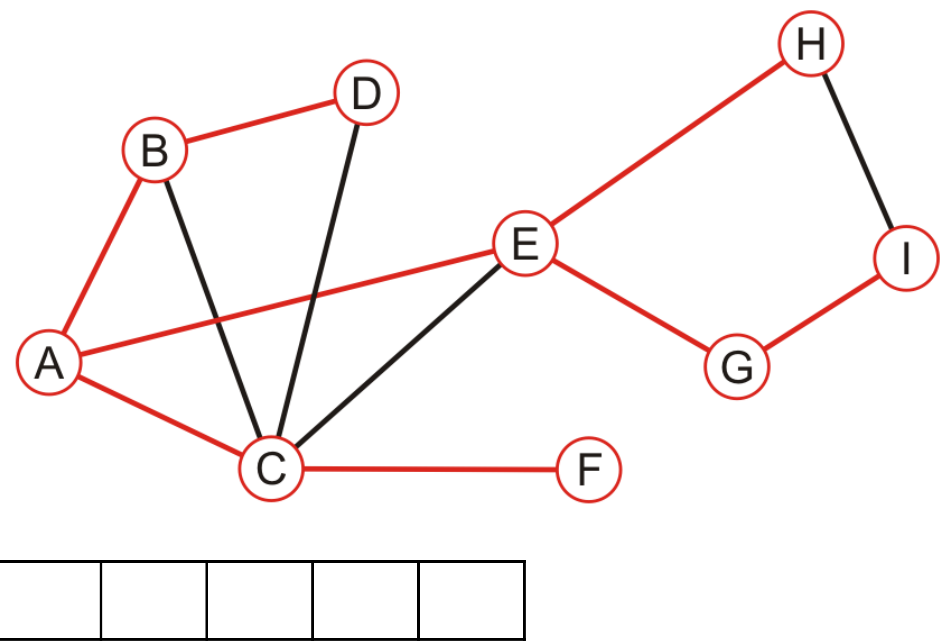
I					

Example - BFS

Performing a breadth-first traversal:

- Pop I

A, B, C, E, D, F, G, H, I

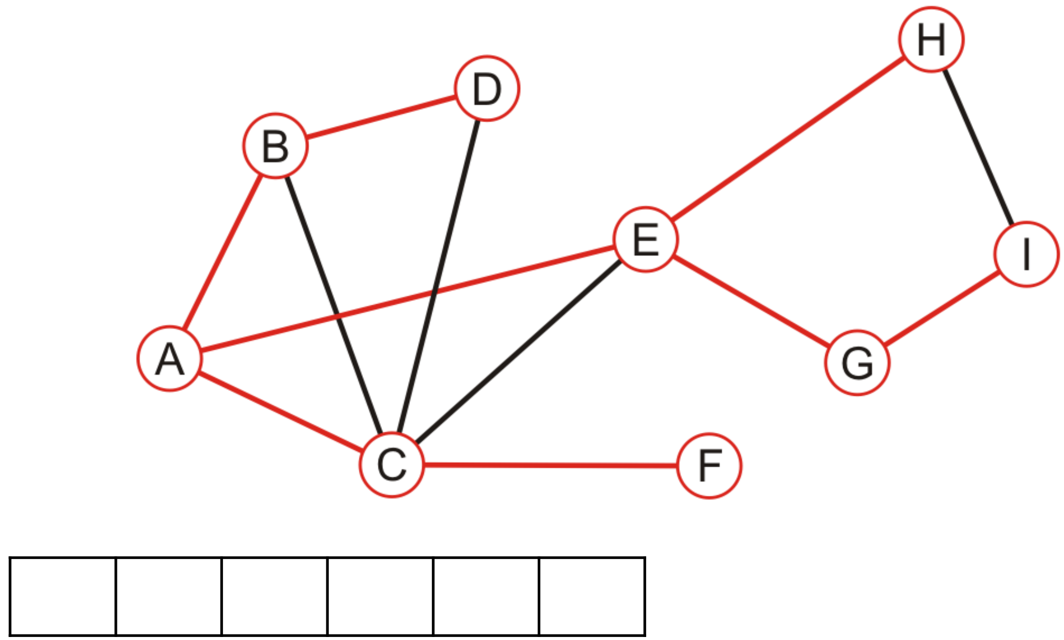


Example - BFS

Performing a breadth-first traversal:

– The queue is empty: we are finished

A, B, C, E, D, F, G, H, I

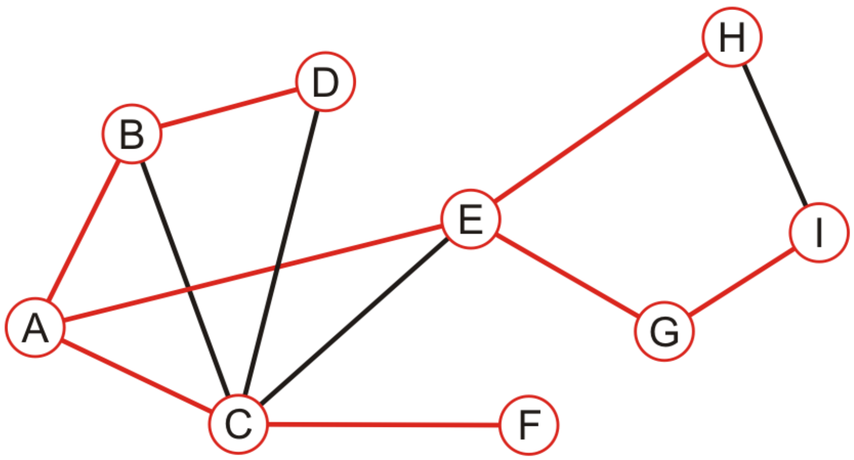


Comparison – DFS v.s BFS

The order in which vertices can differ greatly

- An iterative depth-first traversal may also be different again

A, B, C, E, D, F, G, H, I



A, B, C, D, E, G, I, H, F