CSCE 4263/5183
Advanced Data Structures

Lecture 17

Graph Traversals

Fall 2025

Prof. Khoa Luu
khoaluu@uark.edu

Notice

- Homework 4 Grading: Available
- Homework 5: released
- Final Project

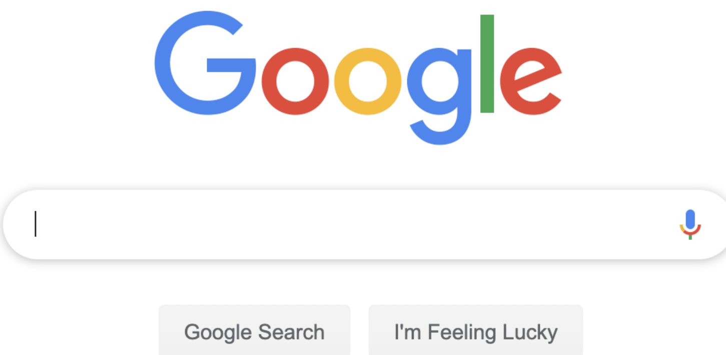
Major Topics In This Course

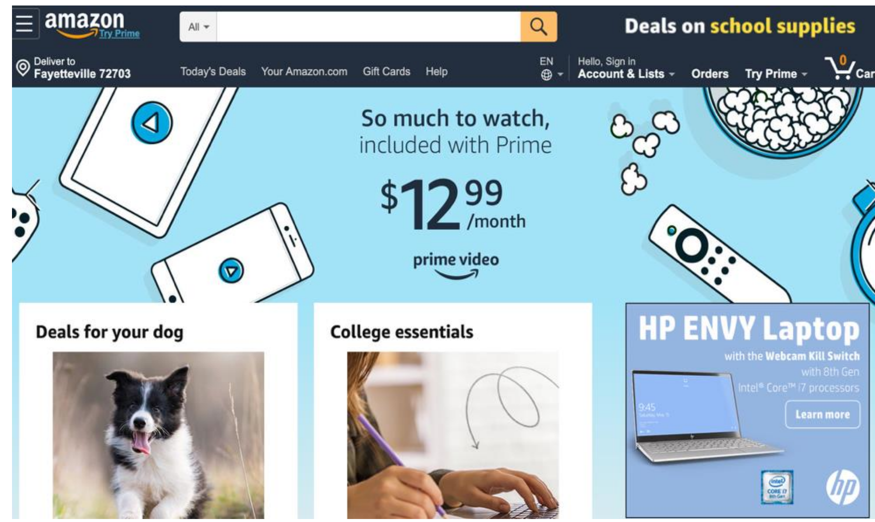
1. Introduction
2. Reviews (Link List, OOP, Binary Tree, BT Search)
3. Self-balancing Binary Search Tree (AVL, Multiway Search, Red-Black)
4. Splay Tree
5. Balanced Search Tree Review
6. Heap Methods
7. Hashing Methods
9. Graph Data Structures
10. Graph CNN
11. Data Structures in Deep Learning
12. Final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

Major Topics In This Course

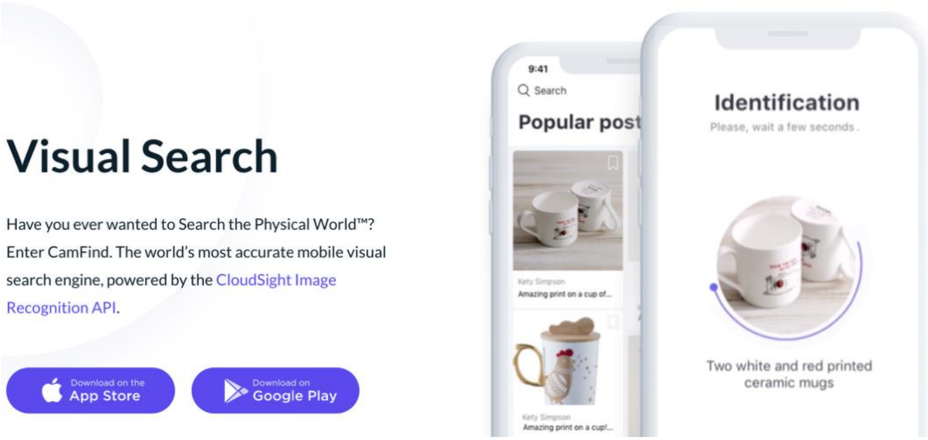
1. Introduction
2. Reviews (Link List, OOP, Binary Tree, BT Search)
3. Self-balancing Binary Search Tree (AVL, Multiway Search, Red-Black)
4. Splay Tree
5. Balanced Search Tree Review
6. Heap Methods
7. Hashing Methods
9. Graph Data Structures
10. Graph CNN
11. Data Structures in Deep Learning
12. Final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

Search - Goal Node

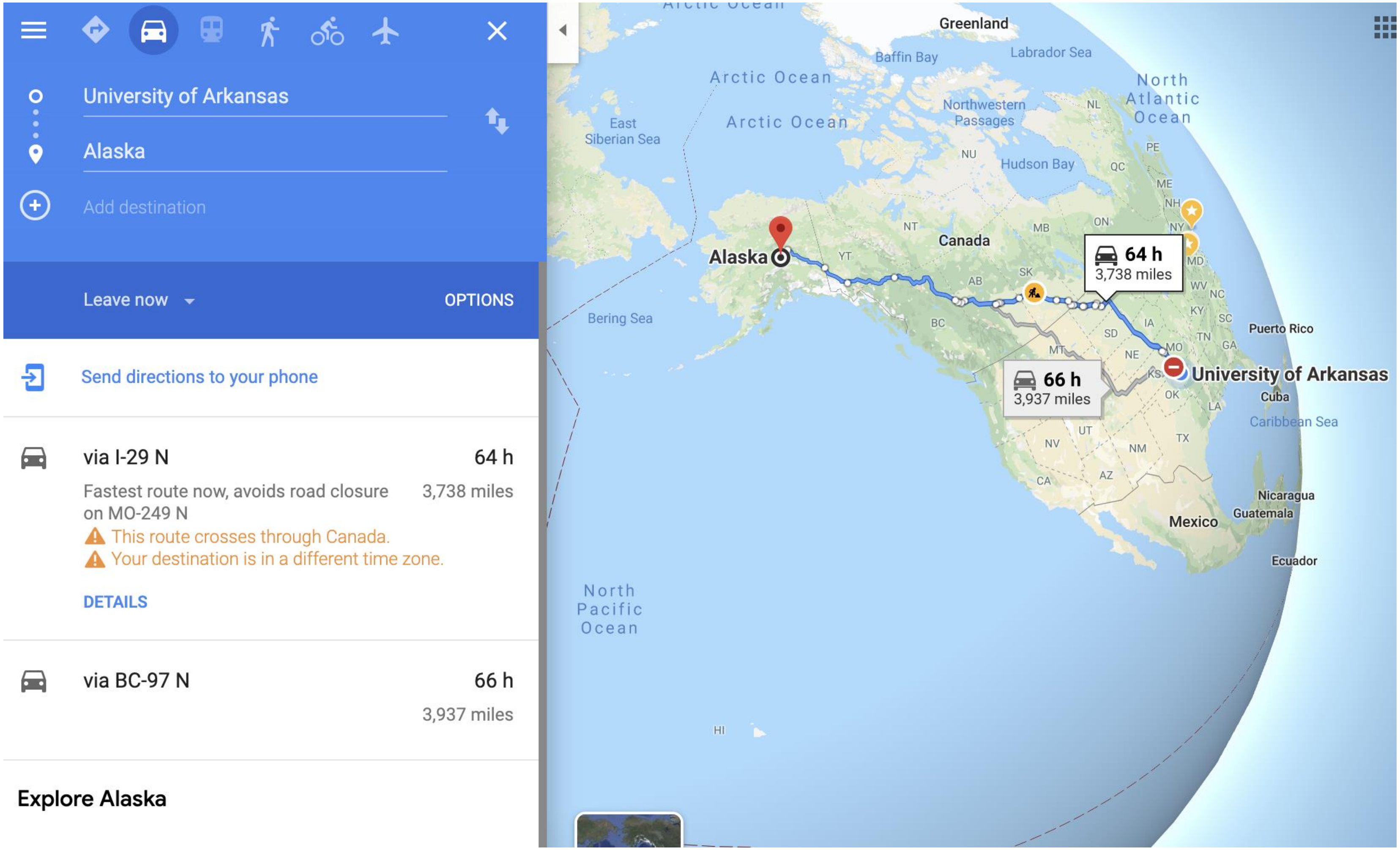




Search - Goal Node



Search – Path to a Goal

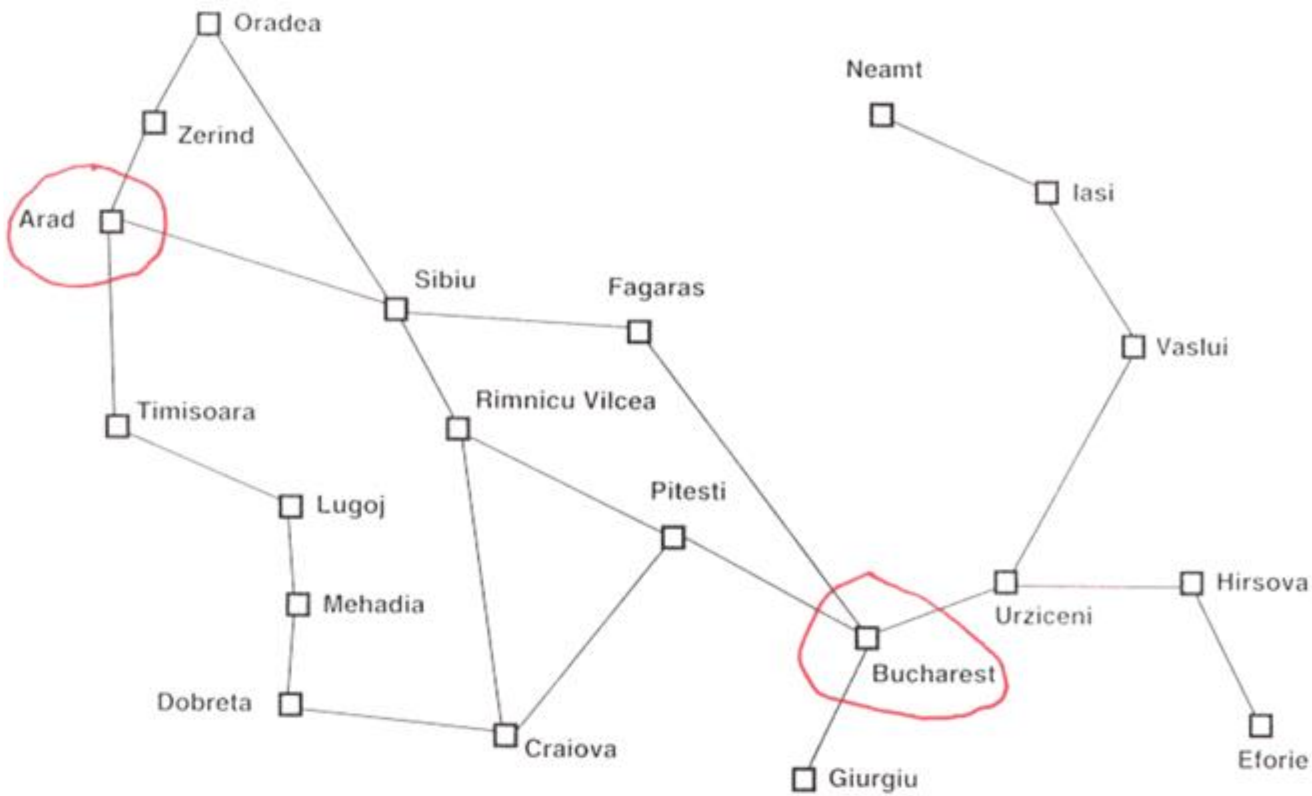


Search

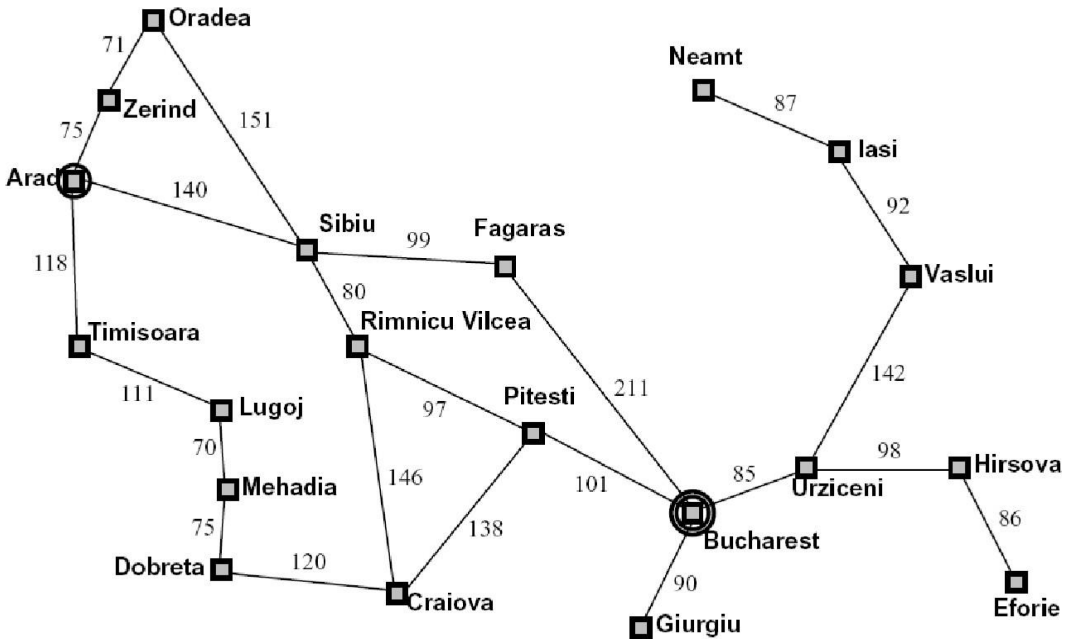
- Goal-based agent
- In some applications we are interested in finding a goal node, in other applications we are interested in finding a path to a goal

No map vs. Map

physical search | deliberative search



Example: Traveling in Romania



- State space:
  - Cities
- Successor function:
  - Roads: Go to adjacent city with cost = distance
- Start state:
  - Arad
- Goal test:
  - Is state == Bucharest?
- Solution?

Strategies

Traversals of graphs are also called *searches*

Strategies

Traversals of graphs are also called *searches*

We can use either breadth-first or depth-first traversals

- Breadth-first requires a queue
- Depth-first requires a stack

Strategies

Traversals of graphs are also called *searches*

We can use either breadth-first or depth-first traversals

- Breadth-first requires a queue
- Depth-first requires a stack

In each case, we will have to track which vertices have been visited requiring $\Theta(|V|)$ memory

- One option is a hash table
- If we can use a bit array, this requires only $|V|/8$ bytes

Strategies

Traversals of graphs are also called *searches*

We can use either breadth-first or depth-first traversals

- Breadth-first requires a queue
- Depth-first requires a stack

In each case, we will have to track which vertices have been visited requiring $\Theta(|V|)$ memory

- One option is a hash table
- If we can use a bit array, this requires only $|V|/8$ bytes

The time complexity cannot be better than and should not be worse than $\Theta(|V| + |E|)$

- Connected graphs simplify this to $\Theta(|E|)$
- Worst case: $\Theta(|V|^2)$

Breadth-first traversal

Consider implementing a breadth-first traversal on a graph:

- Choose any vertex, mark it as visited and push it onto queue
- While the queue is not empty:
  - Pop to top vertex v from the queue
  - For each vertex adjacent to v that has not been visited:
    - Mark it visited, and
    - Push it onto the queue

This continues until the queue is empty

- Note: if there are no unvisited vertices, the graph is connected,

Iterative depth-first traversal

An implementation can use a queue

```
void Graph::depth_first_traversal( Vertex *first ) const {
    unordered_map<Vertex *, int> hash;
    hash.insert( first );
    std::queue<Vertex *> queue;
    queue.push( first );

    while ( !queue.empty() ) {
        Vertex *v = queue.front();
        queue.pop();
        // Perform an operation on v

        for ( Vertex *w : v->adjacent_vertices() ) {
            if ( !hash.member( w ) ) {
                hash.insert( w );
                queue.push( w );
            }
        }
    }
}
```

Breadth-first traversal

The size of the queue is $\mathrm{O}(|V|)$

– The size depends both on:
  - The number of edges, and
  - The out-degree of the vertices

Depth-first traversal

Consider implementing a depth-first traversal on a graph:

- Choose any vertex, mark it as visited
- From that vertex:
  - If there is another adjacent vertex not yet visited, go to it
  - Otherwise, go back to the most previous vertex that has not yet had all of its adjacent vertices visited and continue from there
- Continue until no visited vertices have unvisited adjacent vertices

Depth-first traversal

Consider implementing a depth-first traversal on a graph:

- Choose any vertex, mark it as visited
- From that vertex:
  - If there is another adjacent vertex not yet visited, go to it
  - Otherwise, go back to the most previous vertex that has not yet had all of its adjacent vertices visited and continue from there
- Continue until no visited vertices have unvisited adjacent vertices

Two implementations:

- Recursive
- Iterative

Recursive depth-first traversal

A recursive implementation uses the call stack for memory:

```
void Graph::depth_first_traversal( Vertex *first ) const {
    std::unordered_map<Vertex *, int> hash;
    hash.insert( first );

     first->depth_first_traversal( hash );
}

void Vertex::depth_first_traversal( unordered_map<Vertex *, int> &hash ) const {
    // Perform an operation on this

    for ( Vertex *v : adjacent_vertices() ) {
        if ( !hash.member( v ) ) {
            hash.insert( v );
            v->depth_first_traversal( hash );
        }
    }
}
```

Iterative depth-first traversal

An iterative implementation can use a stack

```
void Graph::depth_first_traversal( Vertex *first ) const {
    unordered_map<Vertex *, int> hash;
    hash.insert( first );
    std::stack<Vertex *> stack;
    stack.push( first );

    while ( !stack.empty() ) {
        Vertex *v = stack.top();
        stack.pop();
        // Perform an operation on v

        for ( Vertex *w : v->adjacent_vertices() ) {
            if ( !hash.member( w ) ) {
                hash.insert( w );
                stack.push( w );
            }
        }
    }
}
```

Iterative depth-first traversal

If memory is an issue, we can reduce the stack size:

- For the vertex:
  - Mark it as visited
  - Perform an operation on that vertex
  - Place it onto an empty stack
- While the stack is not empty:
  - If the vertex on the top of the stack has an unvisited adjacent vertex,
    - Mark it as visited
    - Perform an operation on that vertex
    - Place it onto the top of the stack
  - Otherwise, pop the top of the stack

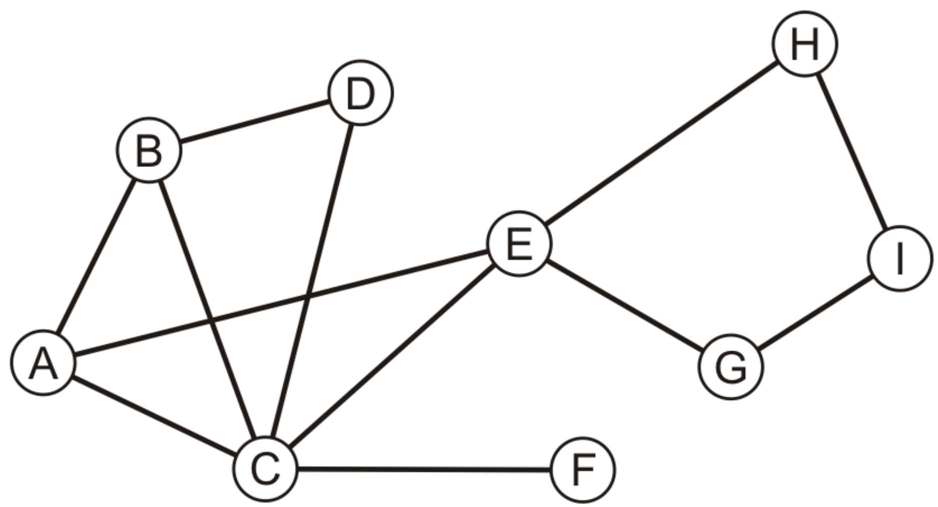
Standard Template Library (STL) approach

An object-oriented STL approach would be create an iterator class:

- The hash table and stack/queue are private member variables created in the constructor
- Internally, it would store the current node
- The auto-increment operator would pop the top of the stack and place any unvisited adjacent vertices onto the stack/queue
- The auto-decrement operator would not be implemented
  - You can’t go back…

Example

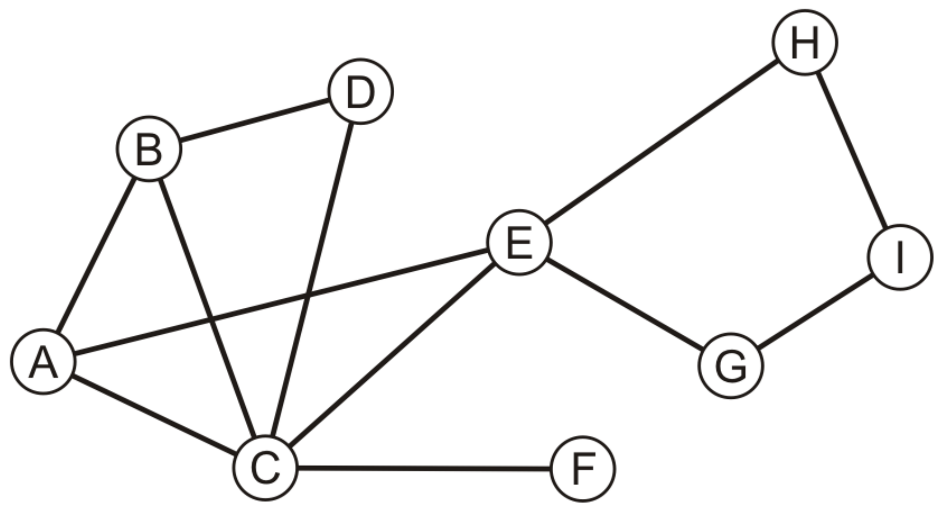
Consider this graph



Example

Performing a breadth-first traversal

– Push the first vertex onto the queue



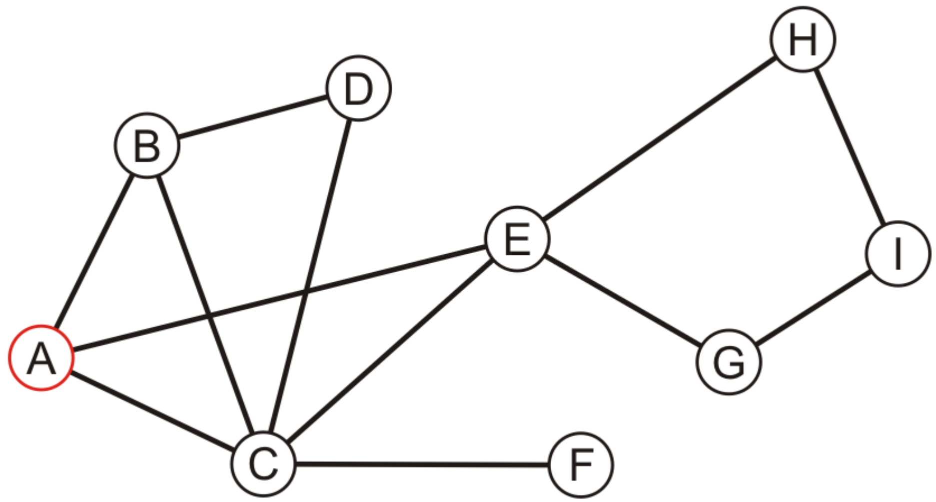
A					

Example

Performing a breadth-first traversal

– Pop A and push B, C and E

A



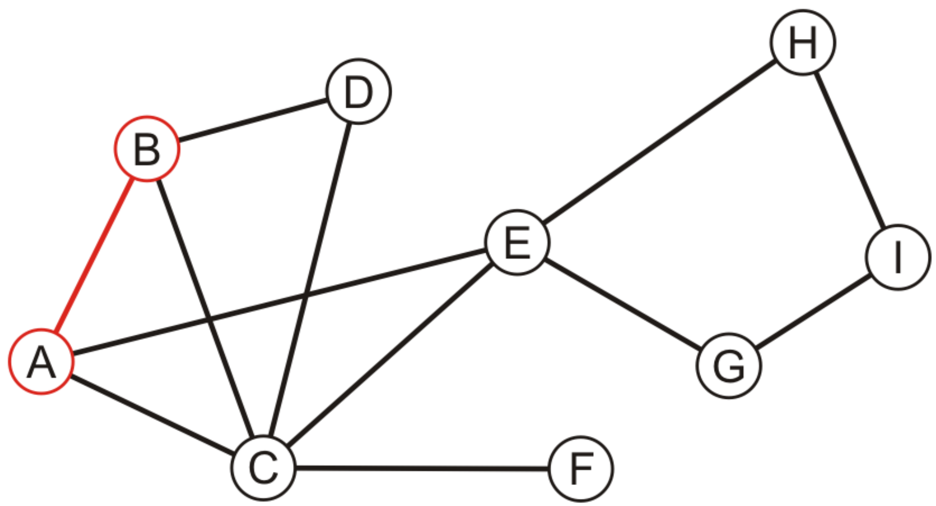
B	C	E			

Example

Performing a breadth-first traversal:

– Pop B and push D

A, B



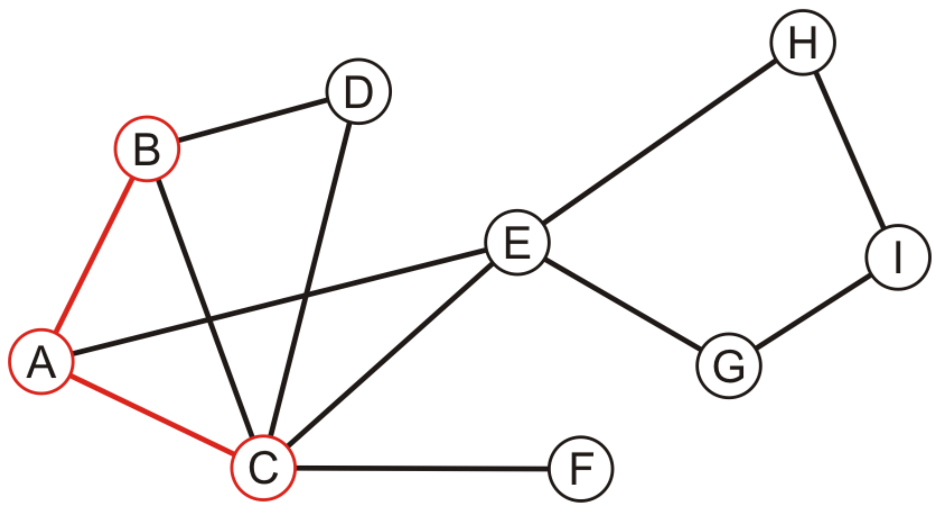
C	E	D			

Example

Performing a breadth-first traversal:

– Pop C and push F

A, B, C



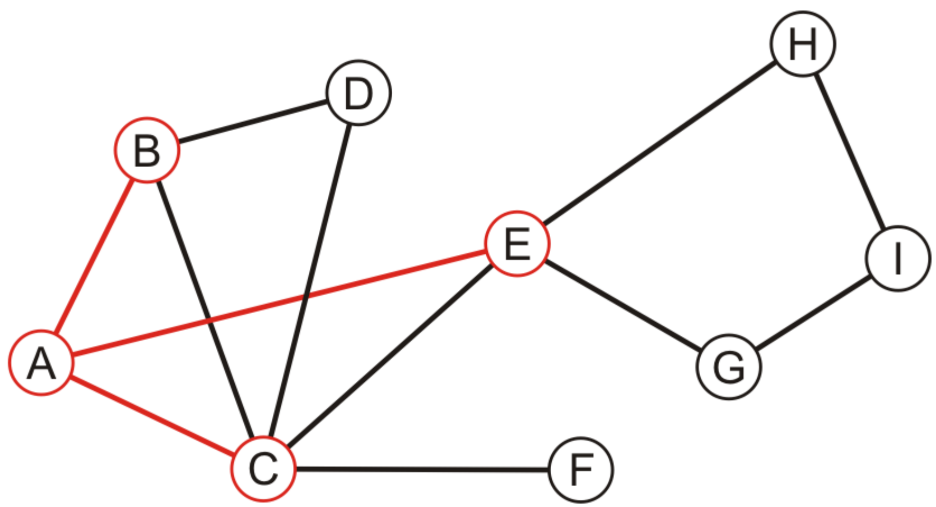
E	D	F			

Example

Performing a breadth-first traversal:

– Pop E and push G and H

A, B, C, E



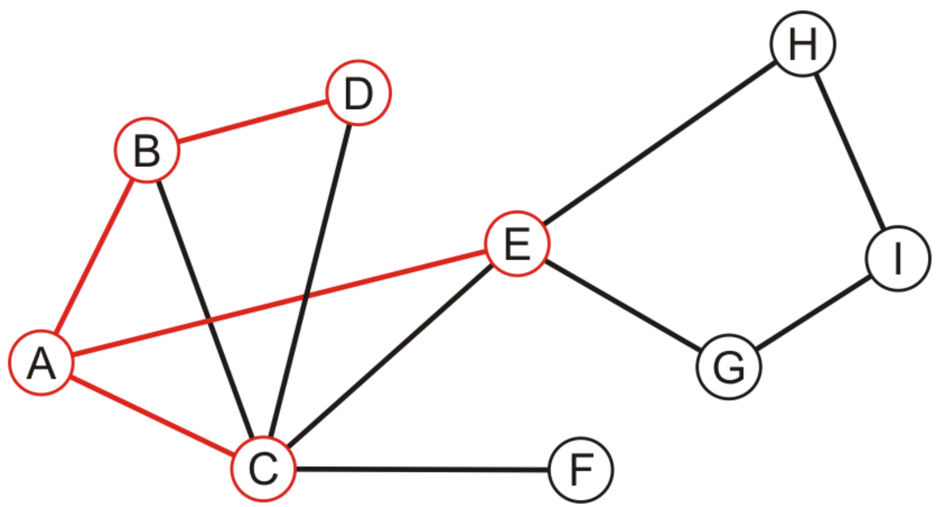
D	F	G	H		

Example

Performing a breadth-first traversal:

- Pop D

A, B, C, E, D



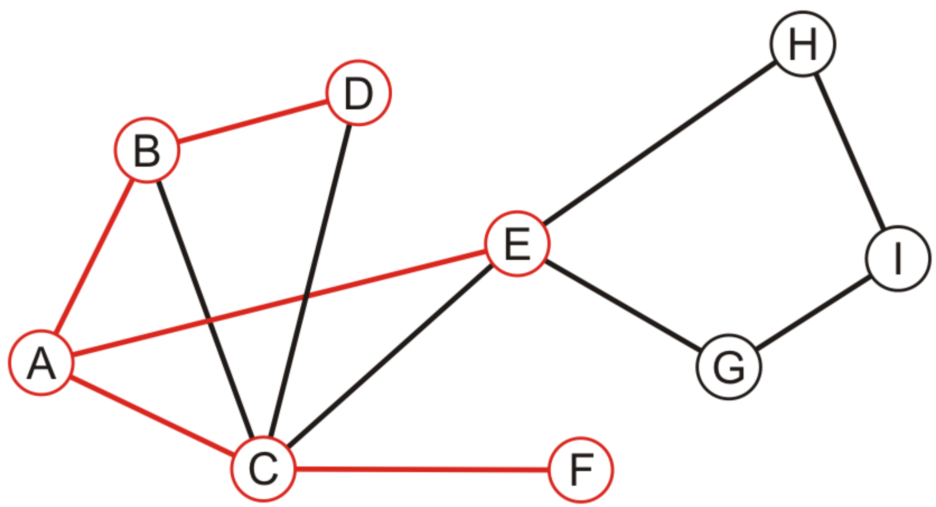
F	G	H			

Example

Performing a breadth-first traversal:

- Pop F

A, B, C, E, D, F



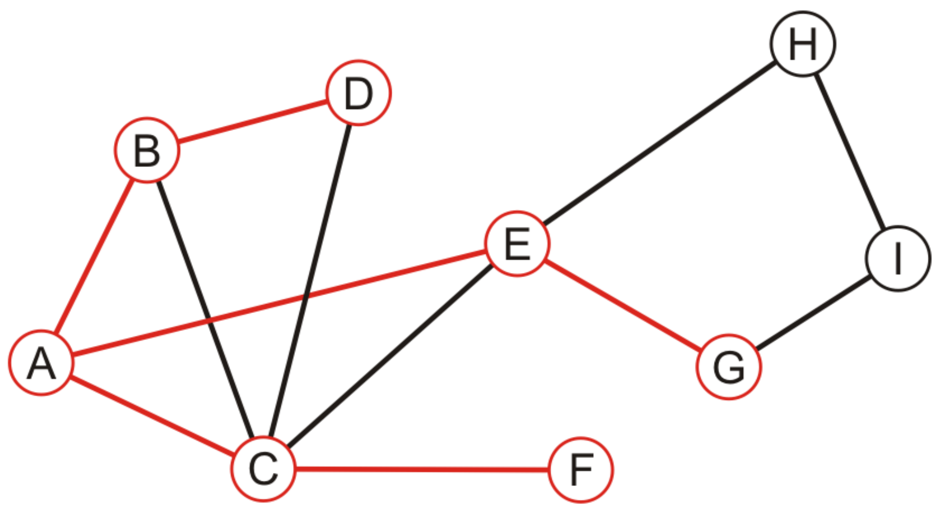
G	H				

Example

Performing a breadth-first traversal:

– Pop G and push I

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



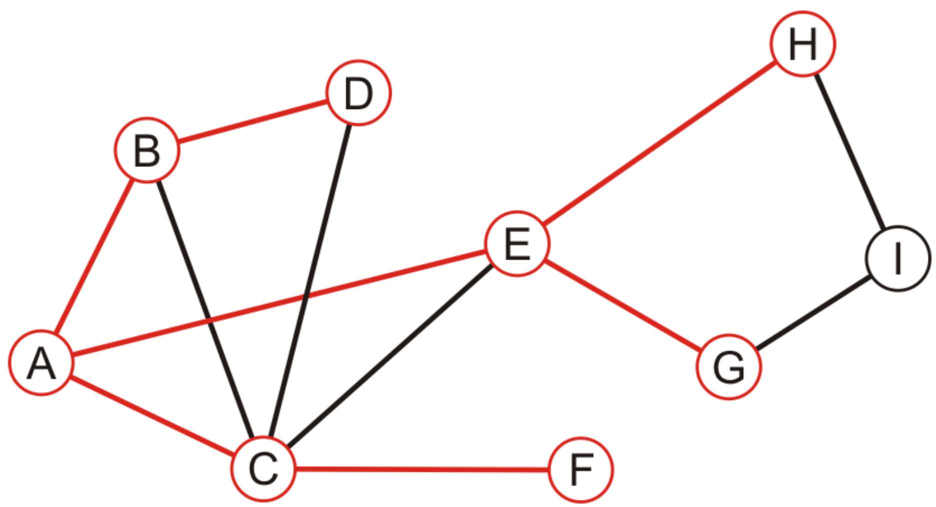
H	I				

Example

Performing a breadth-first traversal:

– Pop H

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



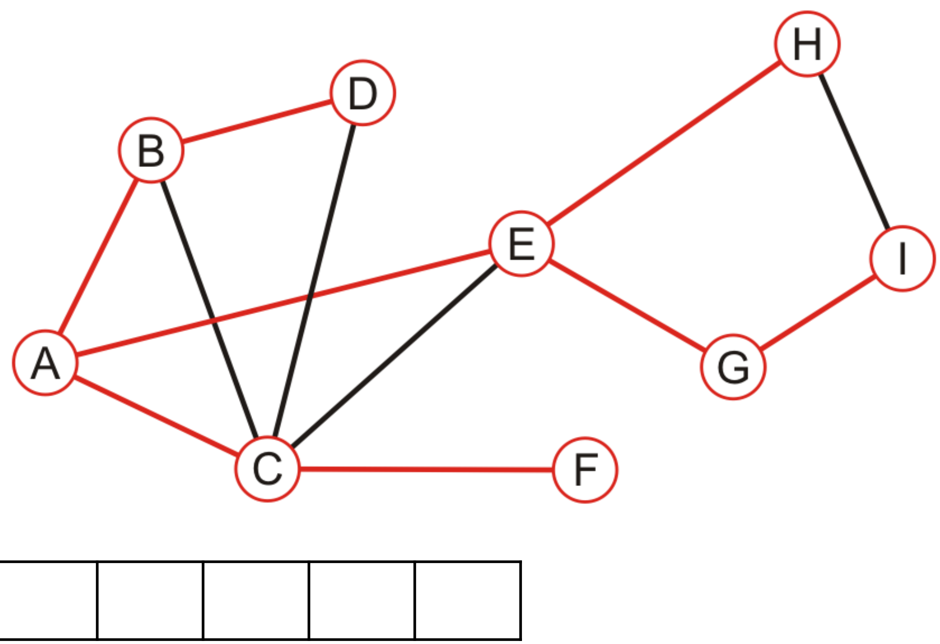
I					

Example

Performing a breadth-first traversal:

– Pop I

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

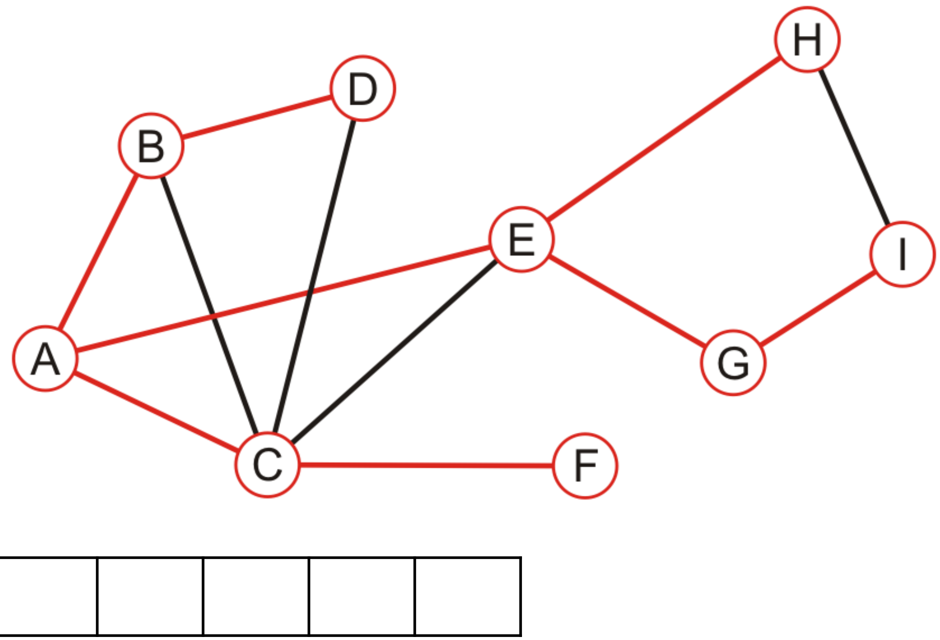


Example

Performing a breadth-first traversal:

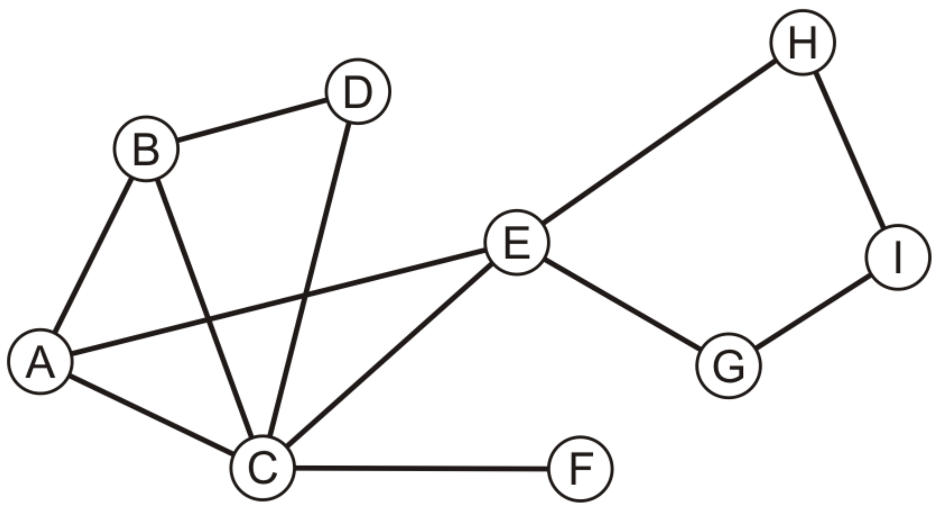
– The queue is empty: we are finished

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



Example

Perform a recursive depth-first traversal on this same graph

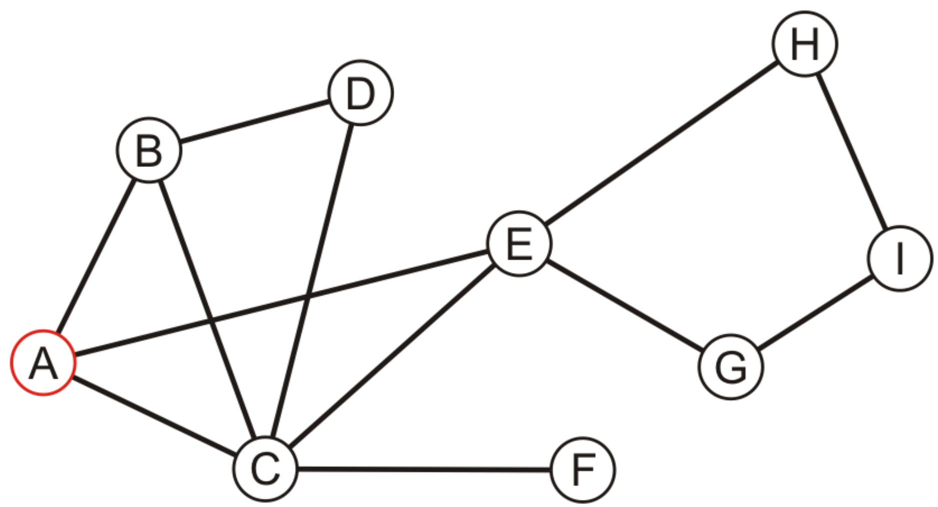


Example

Performing a recursive depth-first traversal:

– Visit the first node

A

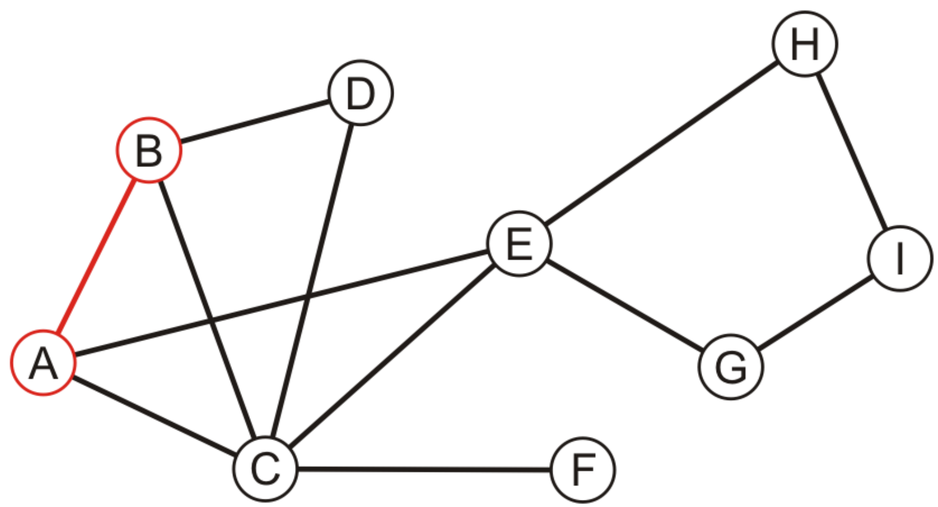


Example

Performing a recursive depth-first traversal:

– A has an unvisited neighbor

A, B

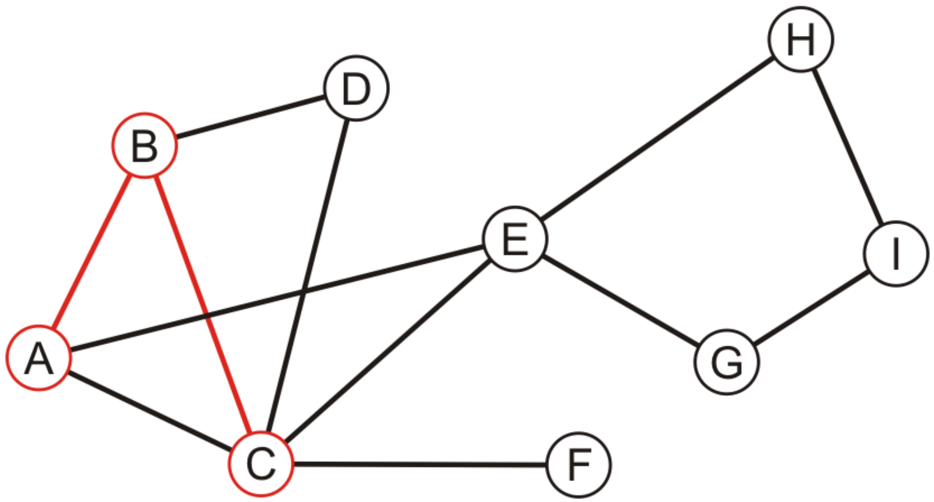


Example

Performing a recursive depth-first traversal:

– B has an unvisited neighbor

A, B, C

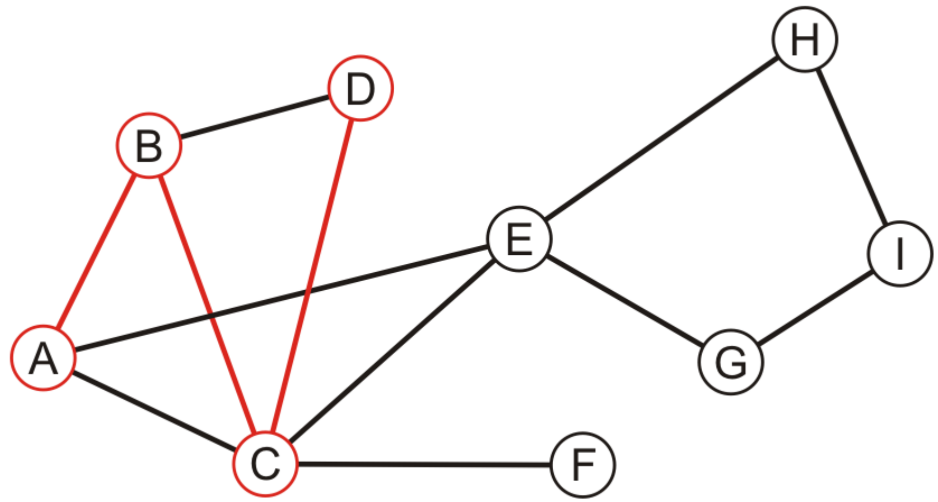


Example

Performing a recursive depth-first traversal:

– C has an unvisited neighbor

A, B, C, D

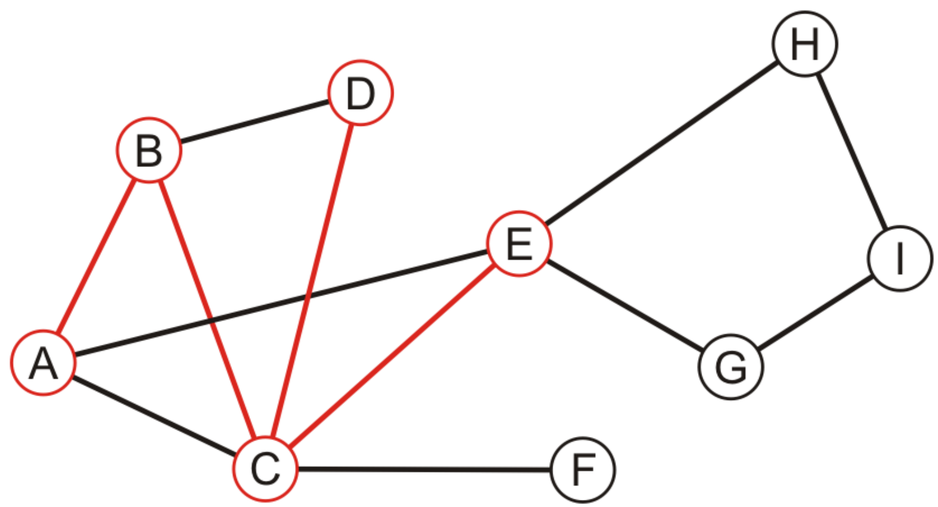


Example

Performing a recursive depth-first traversal:

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

A, B, C, D, E

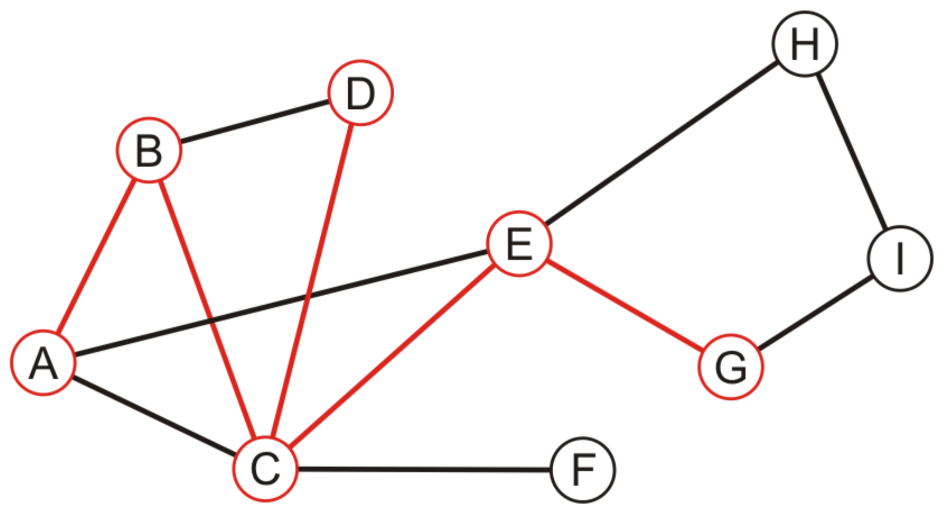


Example

Performing a recursive depth-first traversal:

- E has an unvisited neighbor

A, B, C, D, E, G

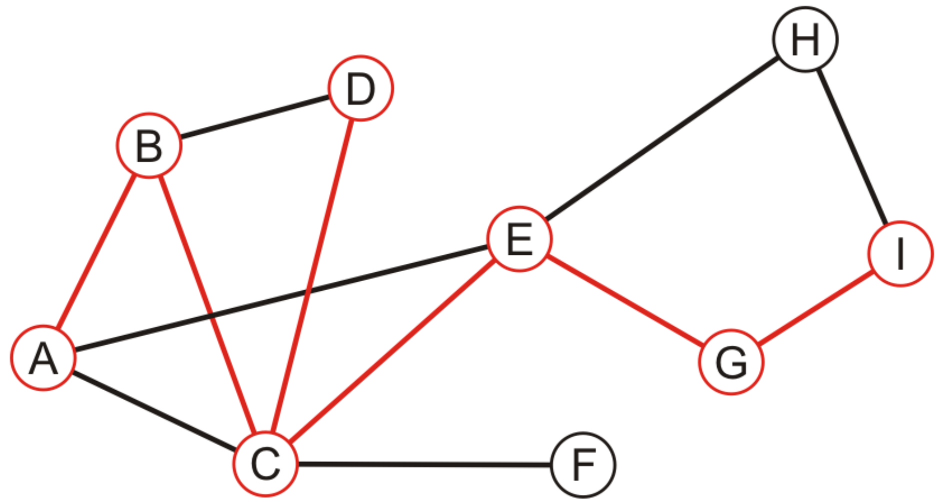


Example

Performing a recursive depth-first traversal:

– F has an unvisited neighbor

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

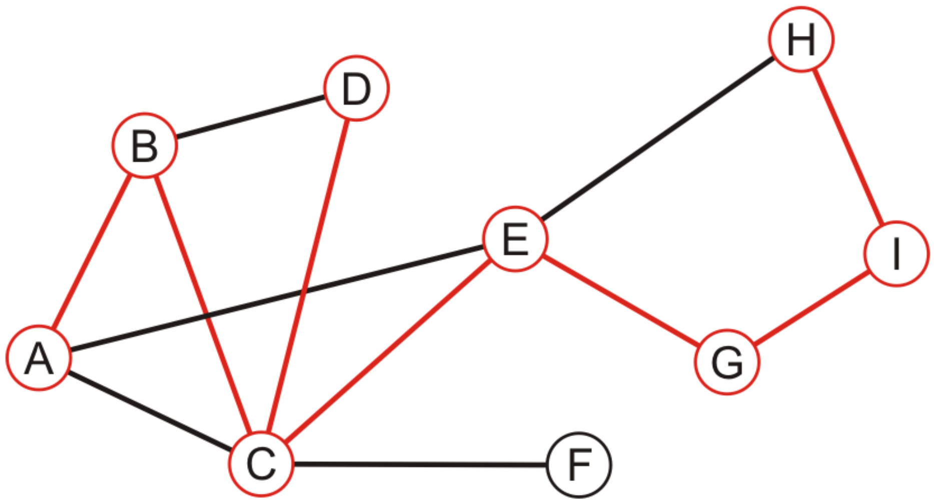


Example

Performing a recursive depth-first traversal:

– H has an unvisited neighbor

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

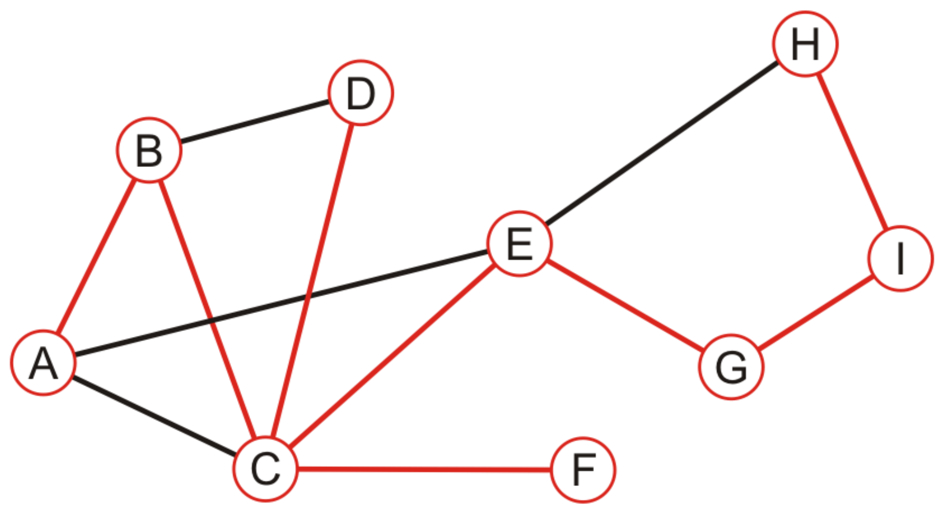


Example

Performing a recursive depth-first traversal:

– We recurse back to C which has an unvisited neighbour

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

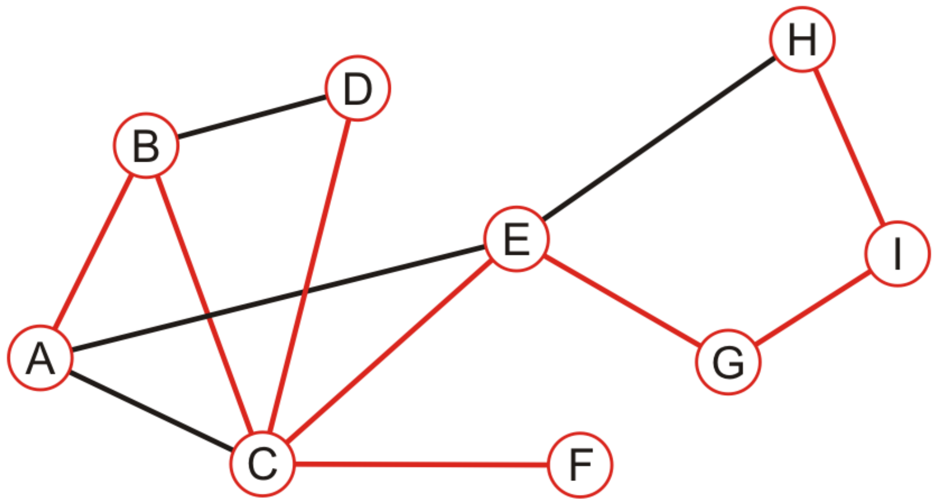


Example

Performing a recursive depth-first traversal:

– We recurse finding that no other nodes have unvisited neighbours

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

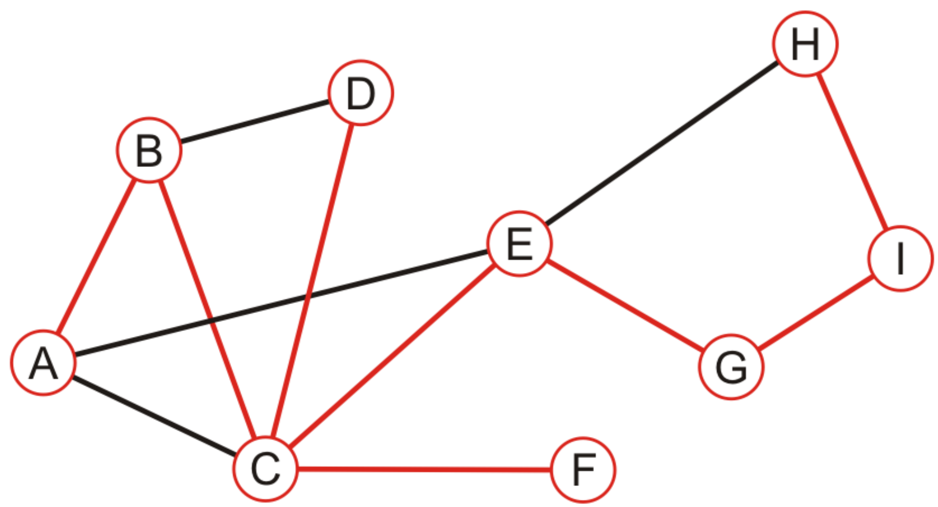


Comparison

Performing a recursive depth-first traversal:

– We recurse finding that no other nodes have unvisited neighbours

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

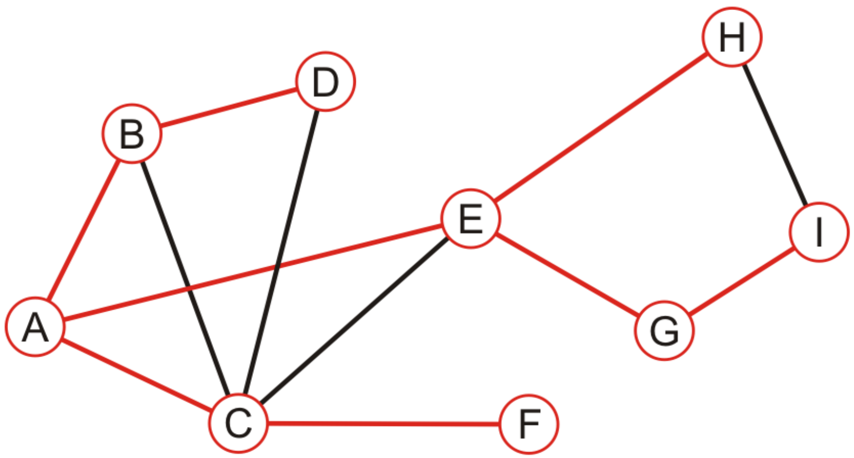


Comparison

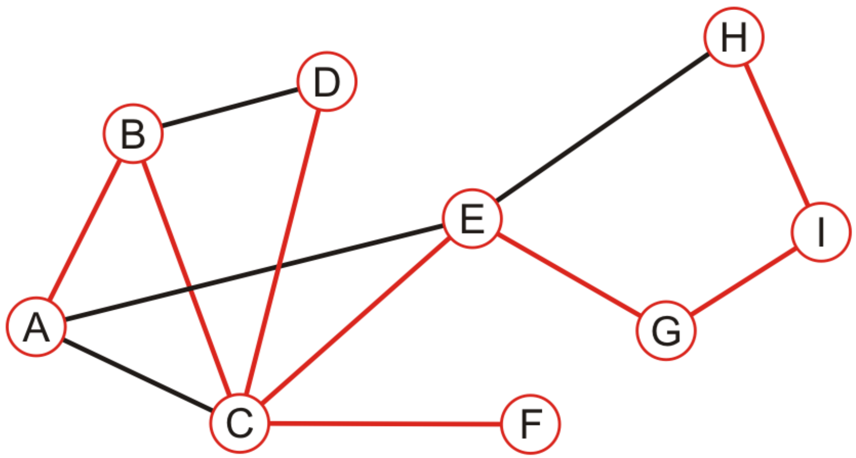
The order in which vertices can differ greatly

- 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



Applications

Applications of tree traversals include:

- Determining connectiveness and finding connected sub-graphs
- Determining the path length from one vertex to all others
- Testing if a graph is bipartite
- Determining maximum flow
- Cheney's algorithm for garbage collection

Summary

This topic covered graph traversals

- Considered breadth-first and depth-first traversals
- Depth-first traversals can recursive or iterative
- More overhead than traversals of rooted trees
- Considered a STL approach to the design
- Considered an example with both implementations
- They are also called *searches*