CSE221 Data Structures
Lecture 20: Graph Traversals

Antoine Vigneron
antoine@unist.ac.kr

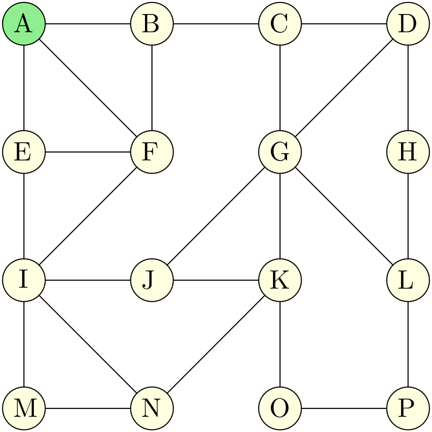
Ulsan National Institute of Science and Technology

November 24, 2021

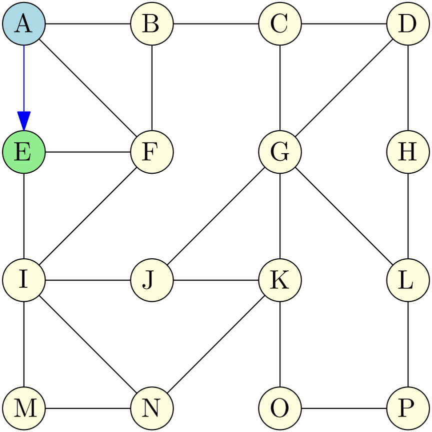
Introduction

- Final exam is on Wednesday 15 December, 20:00–22:00.
- Assignment 3 is due Tomorrow.
- Today's lecture is on algorithms for *undirected* graphs. The two algorithms we present also apply to directed graphs, but some properties will differ. See next lecture.
- Reference for this lecture: Textbook Chapter 13.3.

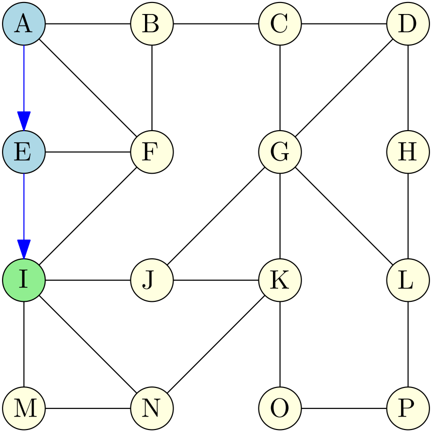
Depth-First Search



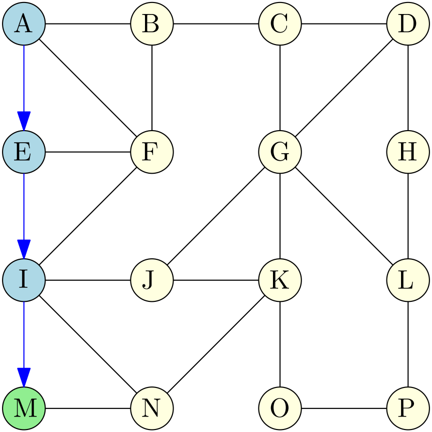
Depth-First Search



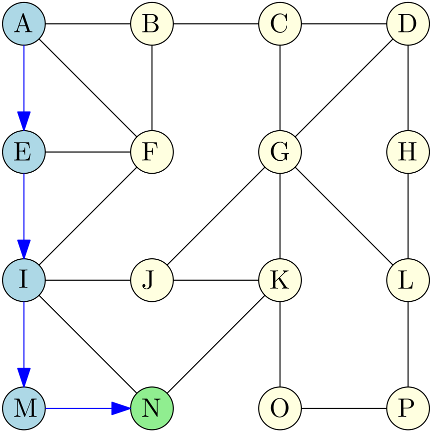
Depth-First Search



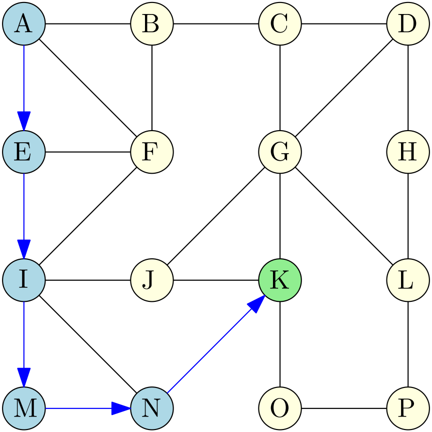
Depth-First Search



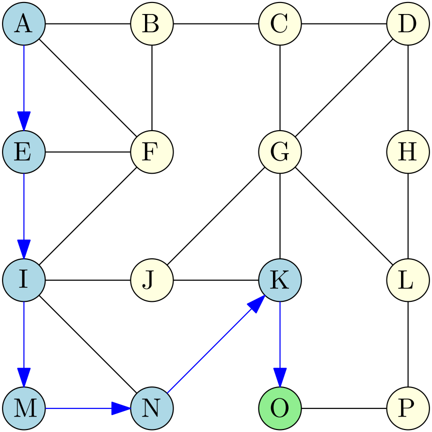
Depth-First Search



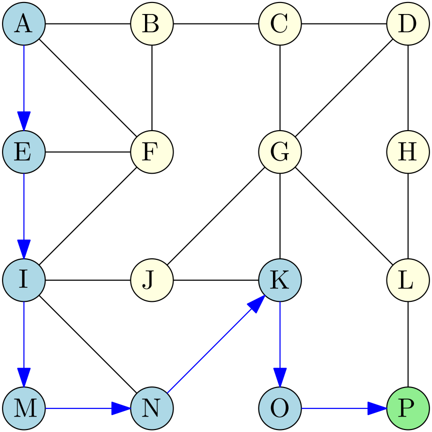
Depth-First Search



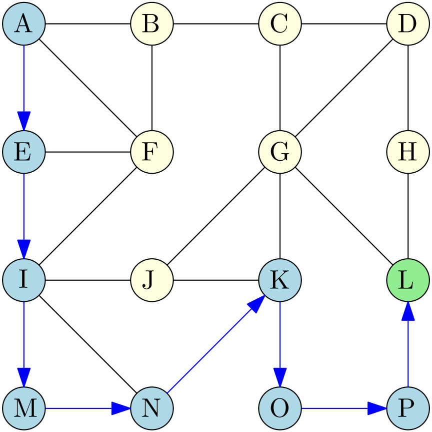
Depth-First Search



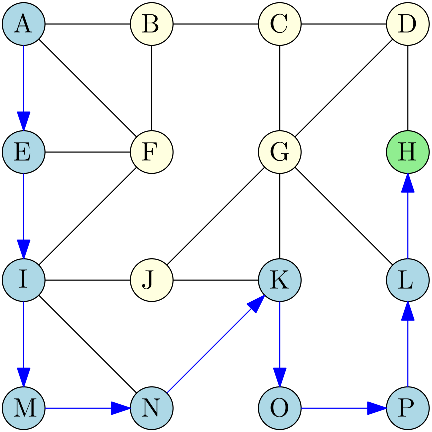
Depth-First Search



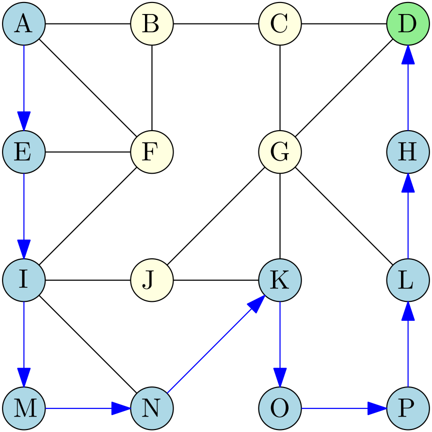
Depth-First Search



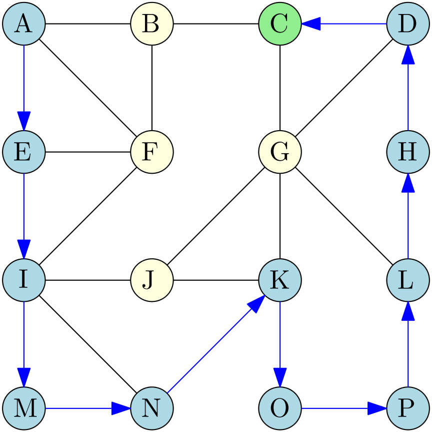
Depth-First Search



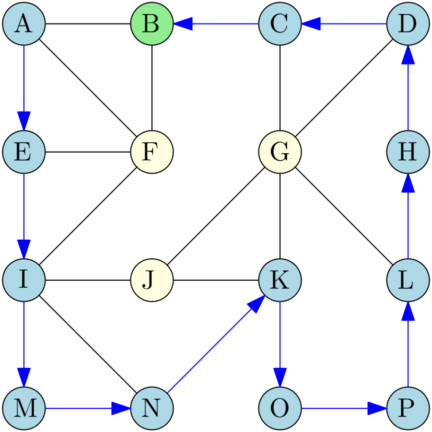
Depth-First Search



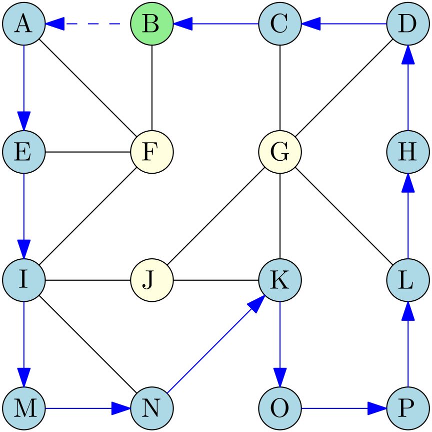
Depth-First Search



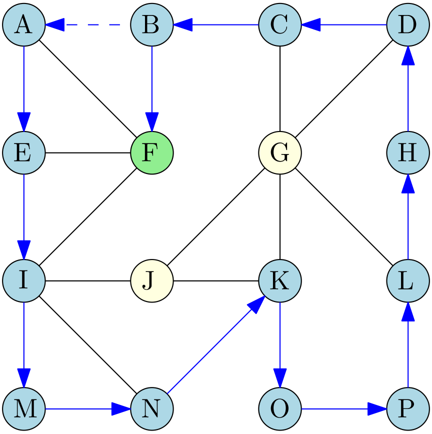
Depth-First Search



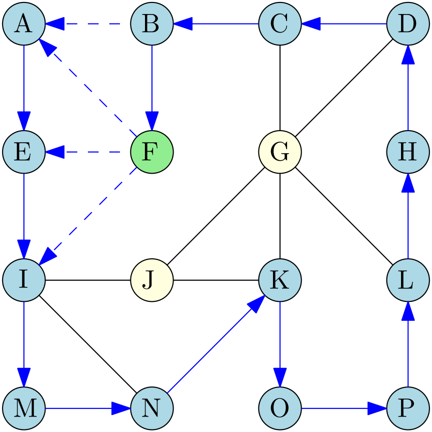
Depth-First Search



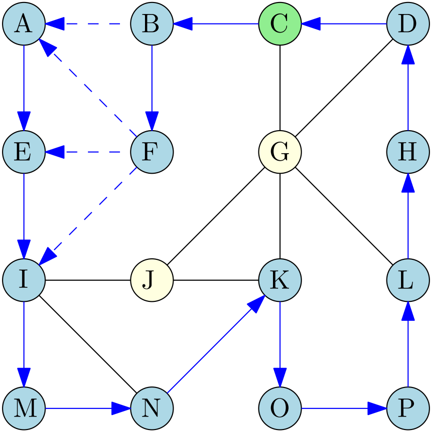
Depth-First Search



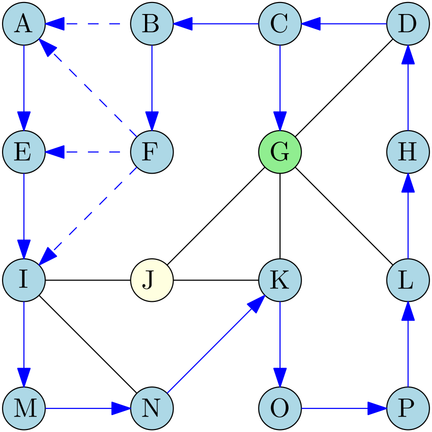
Depth-First Search



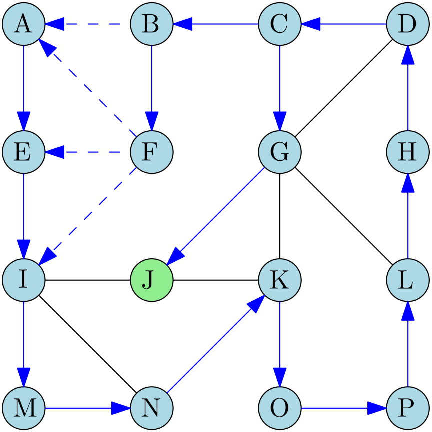
Depth-First Search



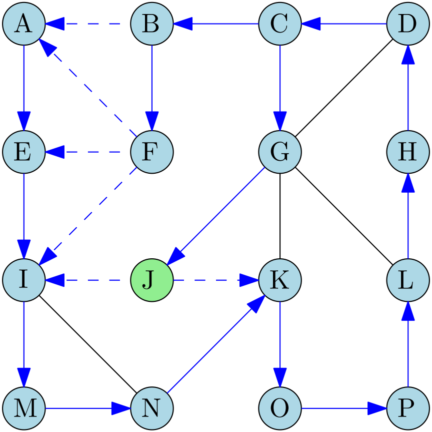
Depth-First Search



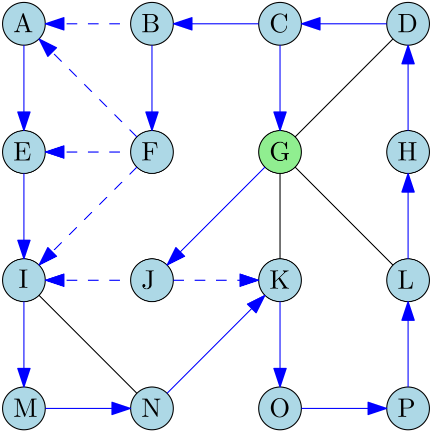
Depth-First Search



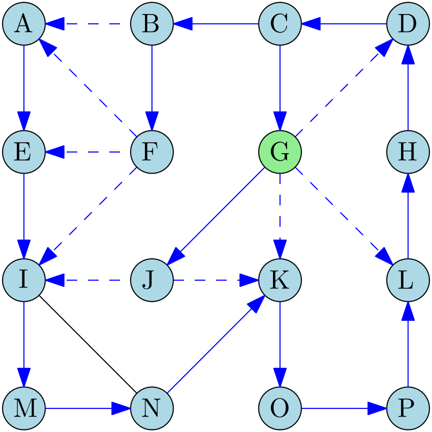
Depth-First Search



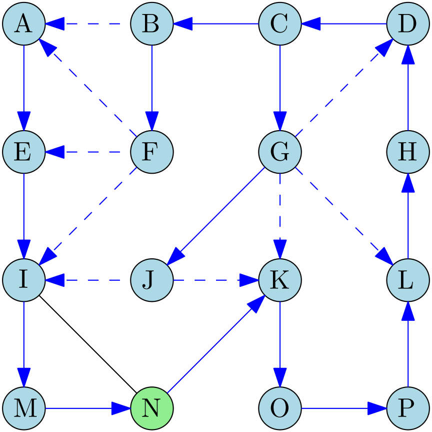
Depth-First Search



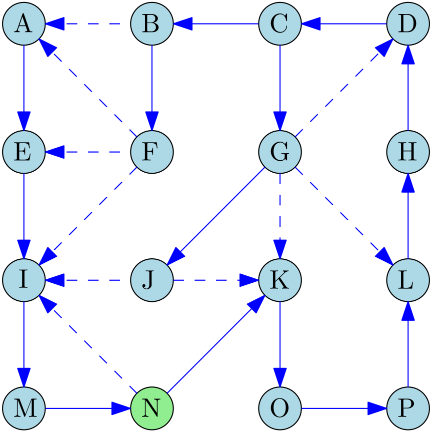
Depth-First Search



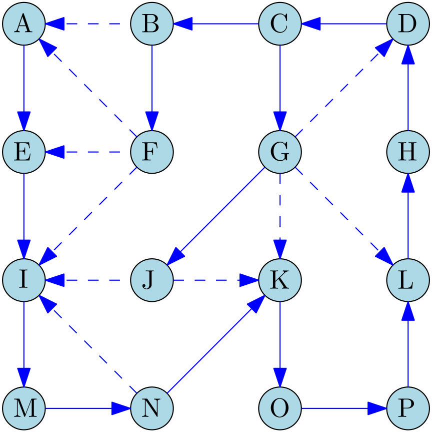
Depth-First Search



Depth-First Search



Depth-First Search



Depth-First Search

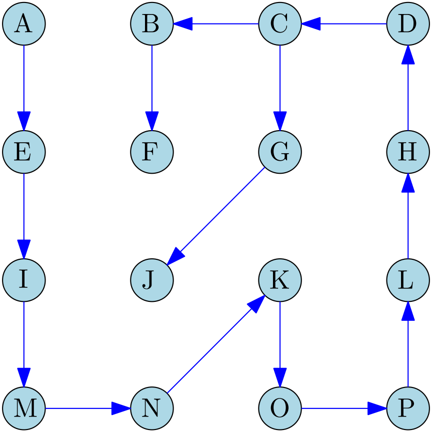
- *Depth-first search* (DFS) is an algorithm that visits all the nodes and edges of a connected graph. It proceeds as follows:

DFS

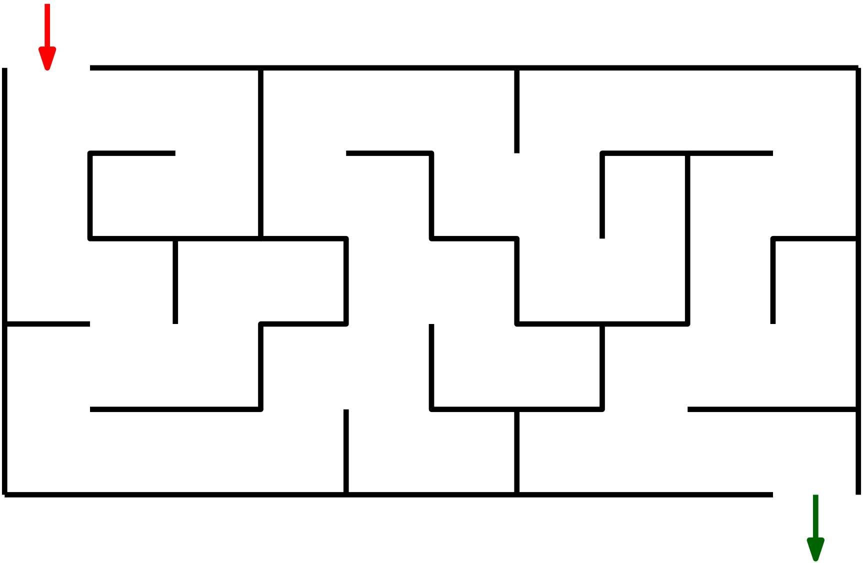
- If an adjacent node has not been visited yet, move to that node.
- Otherwise, backtrack.

- In addition, it label the edges as follows:
  - *tree edges*, (also called *discovery edges*) which are used to discover new vertices,
  - *back edges*, which led to already discovered vertices.
- The *tree edges* form a spanning tree.
- Each back edge connects a vertex to one of its ancestor in this tree.

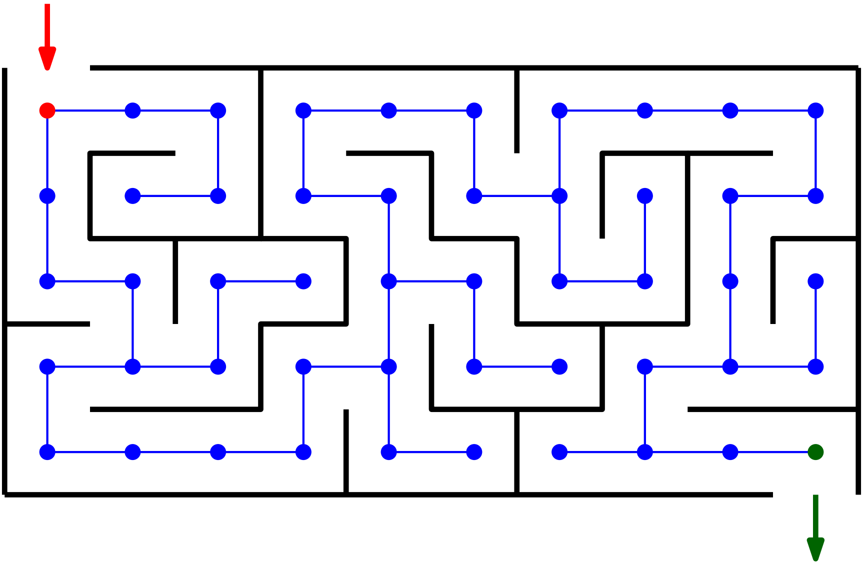
Tree Edges



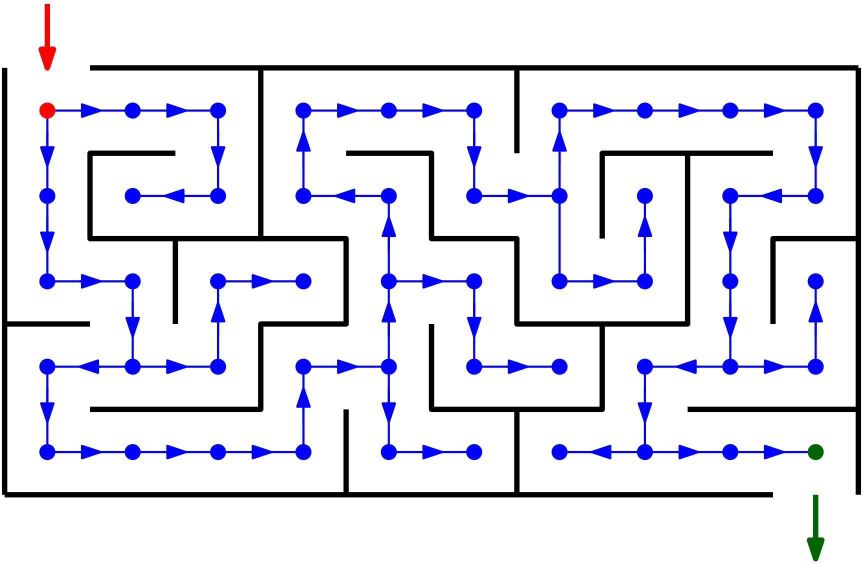
Application: Getting out of a Maze



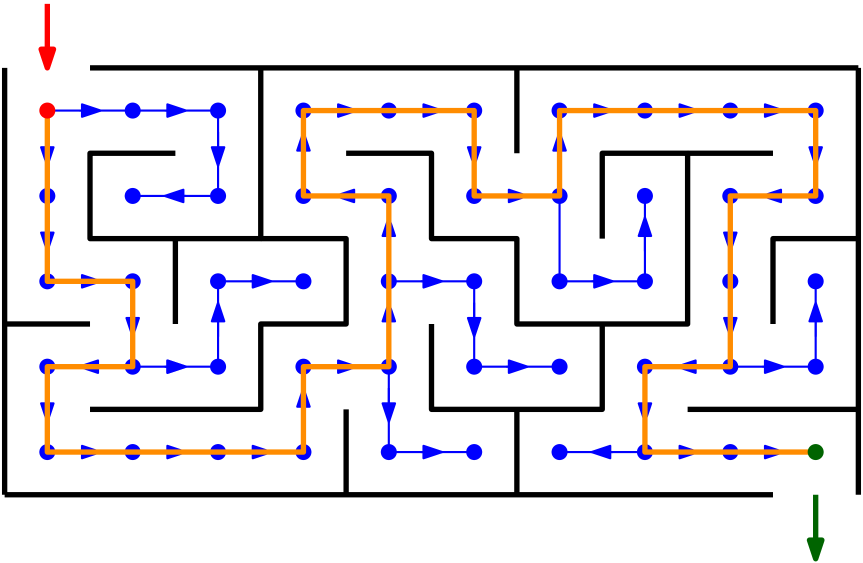
Application: Getting out of a Maze



Application: Getting out of a Maze



Application: Getting out of a Maze



Pseudocode

Depth-First Search

```
procedure DFS(G, v)
    label v as visited
    for all edges e in v.incidentEdges() do
        if edge e is unexplored then
            w ← e. opposite(v)
            if vertex w is unexplored then
                label e as a tree edge
                recursively call DFS(G, w)
            else
                label e as a back edge
```

- Remark: Before running DFS, we need to label all vertices and edges as unexplored, which takes $O(n + m)$ time.

Properties

Proposition

Let G be an undirected graph on which a DFS traversal starting at a vertex s has been performed. Then the traversal visits all vertices in the connected component of s , and the tree edges form a spanning tree of the connected component of s.

Proof.

All the nodes are visited. Otherwise, take an unexplored node v, and the first node w on a path from s to v that is not visited. Take the node u that comes before w on this path. Then u was visited. But then w must have been visited too, a contradiction.
As we never construct a tree edge leading to an explored vertex, we do not form cycles, hence the tree edges form a tree. □

Properties

Proposition

Let G be a graph with n vertices and m edges represented with an adjacency list. A DFS traversal of G can be performed in $O(n+m)$ time, and can be used to solve the following problems in $O(n+m)$ time:

- *Testing whether G is connected.*
- *Computing a spanning tree of G, if G is connected.*
- *Computing the connected components of G.*
- *Computing a path between two given vertices of G, if it exists.*
- *Computing a cycle in G, or reporting that G has no cycles.*

The Decorator Pattern

- In order to run DFS, we need to be able to mark vertices as visited.
- So each node in our data structure should have a field especially designed for DFS.
- An alternative is to use the *decorator pattern*:

Definition

We say that an object is *decorable* if it supports the following functions:

- **set**(a, x): Set the value of attribute a to x.
- **get**(a): Return the value of attribute a

- We add *decorations* (also called *attributes*) to existing objects.
- Each decoration is identified by a key identifying this decoration and by a value associated with the key.
- Our keys will be strings.

```cpp
Object* yes = new Object;                    // decorator values
Object* no = new Object;
Decorator v;                                 // a decorable object
// ...
v.set("visited", yes);             // set "visited" attribute
// ...
if (v.get("visited") == yes) cout << "v was visited";
else cout << "v was not visited";
```

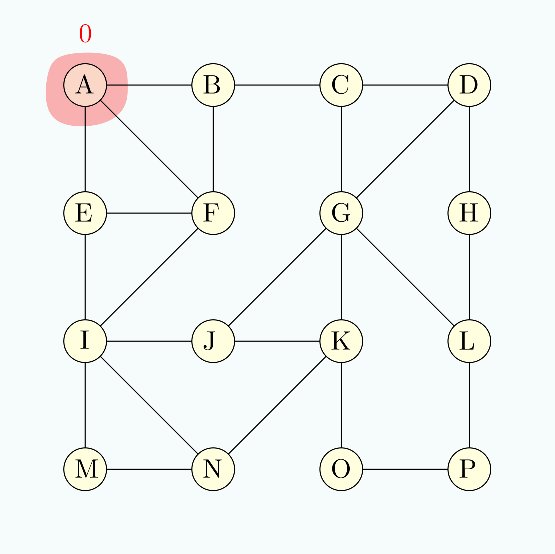
```cpp
class Decorator {
private:                                              // member data
  std::map<string,Object*> map;                       // the map
public:
  Object* get(const string& a)
    { return map[a]; }                // get value of attribute
  void set(const string& a, Object* d)
    { map[a] = d; }                                   // set value
};
```

DFS Traversal using Decorable Positions

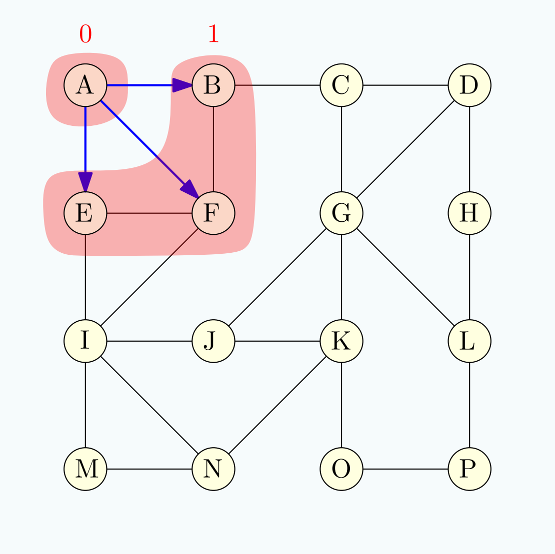
Depth-First Search

procedure DFS(G, v)
    v.set("status", *visited*)
    for all edges e in v.incidentEdges() **do**
        if e.get("status")=*unexplored* **then**
            $w \leftarrow e$. opposite(v)
            if w.get("status")=*unexplored* **then**
                e.set("status",*tree_edge*)
                DFS(G, w)
            else
                e.set("status",*back*)

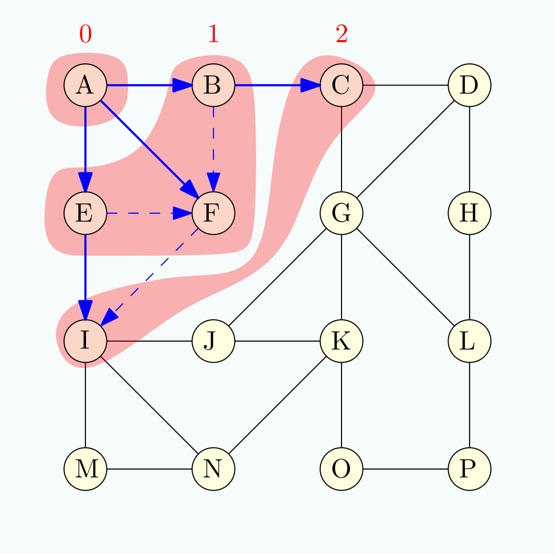
Breadth-First Search



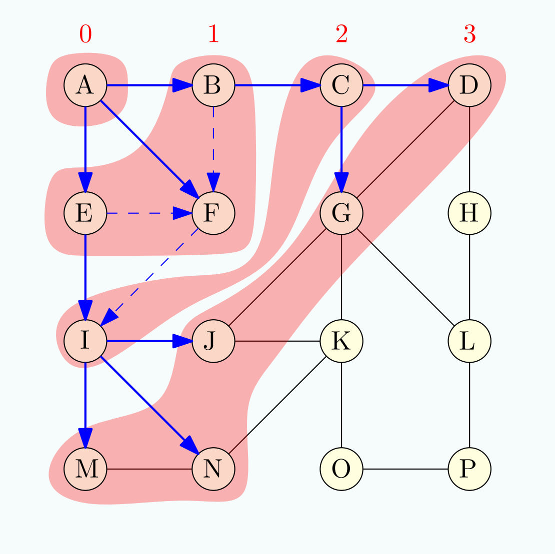
Breadth-First Search



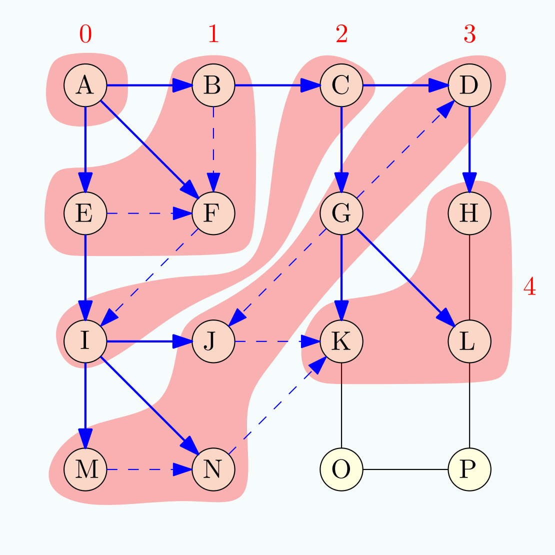
Breadth-First Search



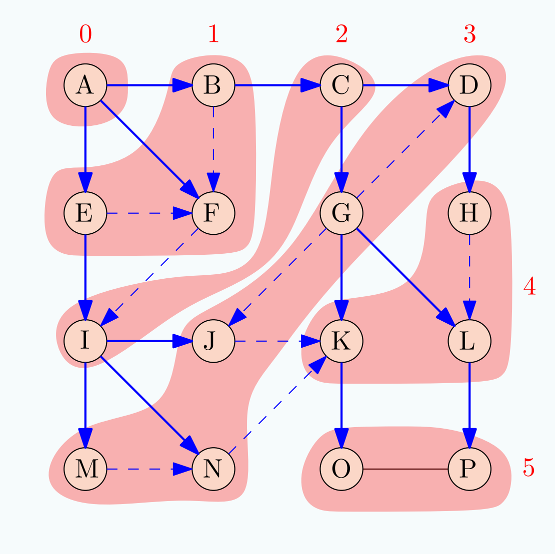
Breadth-First Search



Breadth-First Search



Breadth-First Search



Breadth-First Search

- *Breadth-first search* traverses the graph level by level: we first visit L_0, then L_1, L_2 ... Vertices in L_i are adjacent to vertices in L_{i-1}.
- Similarly as DFS, some edges allow us to discover new vertices. They are also called *tree edges*, and form a spanning tree.
- The edges that lead to already discovered vertices are called *cross edges*. As opposed to the *back edges* from DFS, cross edges never connect a vertex to one of its ancestors.

Pseudocode

Breadth-First Search

```
procedure BFS(G, s)
    initialize collection L_0 to contain vertex s
    i ← 0
    while L_i ≠ ∅ do
        create an empty collection L_{i+1}
        for all vertices v ∈ L_i do
            for all edges e ∈ v.incidentEdges() do
                if edge e is unexplored then
                    w ← e.opposite(v)
                    if vertex w is unexplored then
                        label e as a tree edge
                        insert w into L_{i+1}
                    else
                        label e as a cross edge
        i ← i + 1
```

Properties

- Before running BFS, we need to label all the edges as unexplored.

Proposition

Let G be an undirected graph on which a BFS traversal starting at vertex s has been performed. Then

- *The traversal visits all vertices in the connected component of s.*
- *The discovery-edges form a spanning tree T, which we call the BFS tree, of the connected component of s.*
- *For each vertex v at level i, the path of the BFS tree T between s and v has i edges, and any other path of G between s and v has at least i edges.*
- *If (u, v) is an edge that is not in the BFS tree, then the level numbers of u and v differ by at most 1.*

Properties

Proposition

Let G be a graph with n vertices and m edges represented with the adjacency list structure. A BFS traversal of G takes $O(n+m)$ time. Also, there exist $O(n+m)$-time algorithms based on BFS for the following problems:

- *Testing whether G is connected.*
- *Computing a spanning tree of G , if G is connected.*
- *Computing the connected components of G.*
- *Given a start vertex s of G , computing, for every vertex v of G, a path with the minimum number of edges between s and v , or reporting that no such path exists.*
- *Computing a cycle in G , or reporting that G has no cycles.*