Data Structures

Lecture 13.2: Graph Traversals

Nopadon Juneam
Department of Computer Science
Kasetsart university

Outlines

- More terminology about graphs
- Graph traversal
    - Depth-first search and its implementation

More Terminology about Graphs

- ***Path*** := A sequence of alternating vertices and edges that starts at a vertex and ends at a vertex such that each edge is incident to its predecessor and successor vertex

- ***Cycle*** := A path with at least one edge that has the same start and end vertices

- ***Path's length*** := The number of vertices in the path minus one; the number of edges in the path

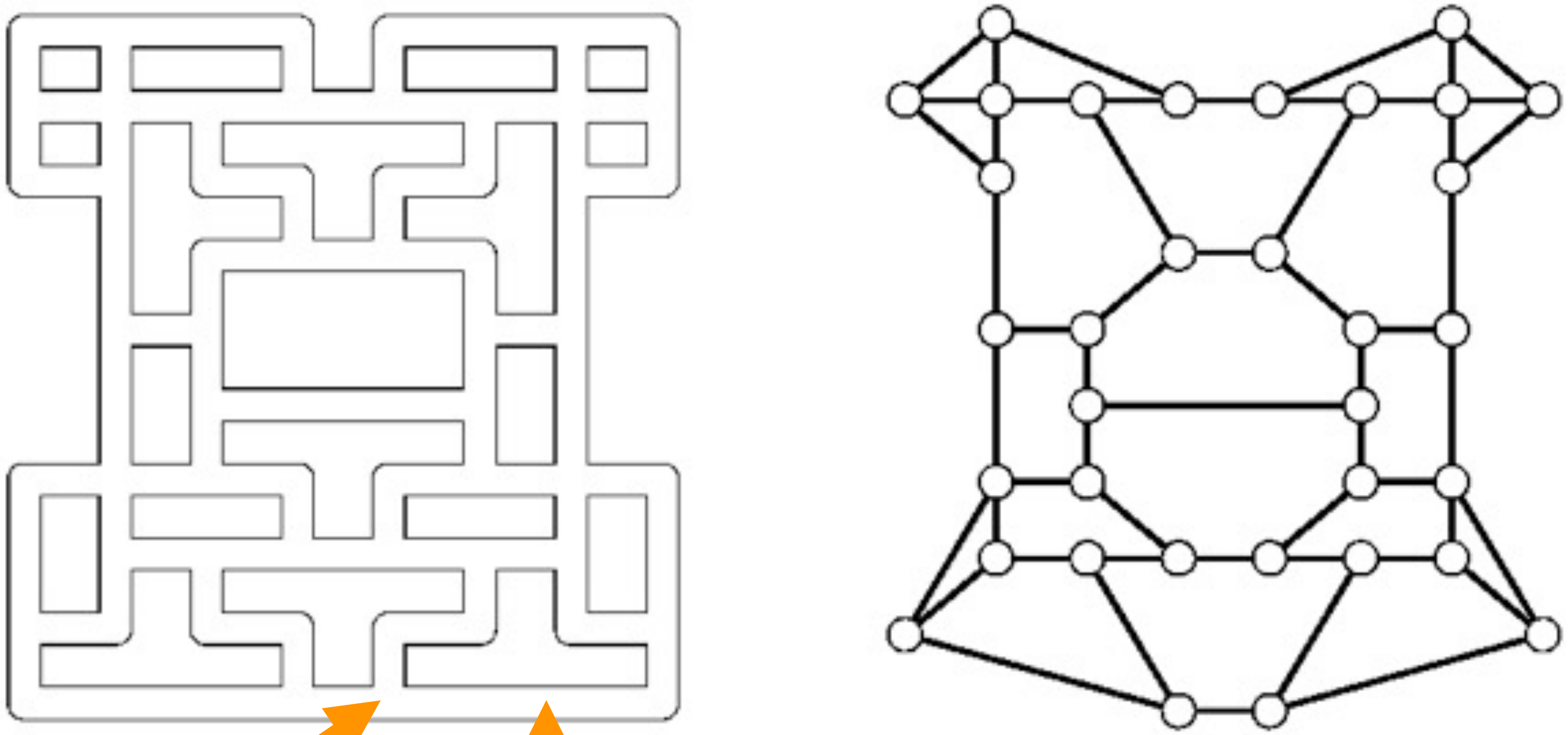
- ***Connected graph*** := A graph is *connected* if, for any two vertices, there is a path between them

- ***Connected components*** := If a graph is not connected, its maximal connected subgraphs are called the *connected components* of the graph

Graph Traversals (Graph Search)

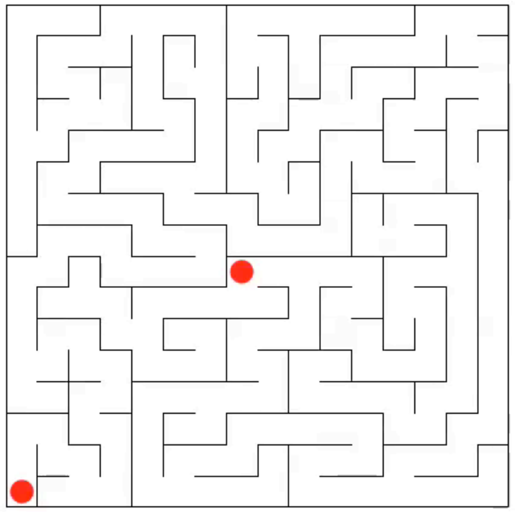
- ***Traverse*** := travel across; visit each element exactly once
    - *Example*: Traverse an array; traverse a linked list
    - Graph’s elements are vertices and edges
- ***Graph traversal*** is a systematic procedure for exploring a graph by examining all of its vertices and edges

Maze Exploration: Motivation

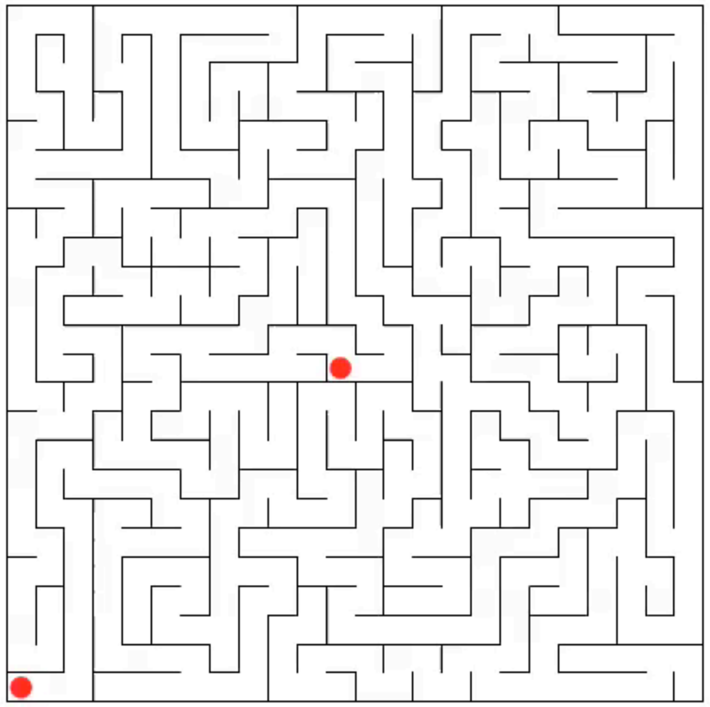


- Maze graph
  - Vertex = intersection
  - Edge = passage between intersections

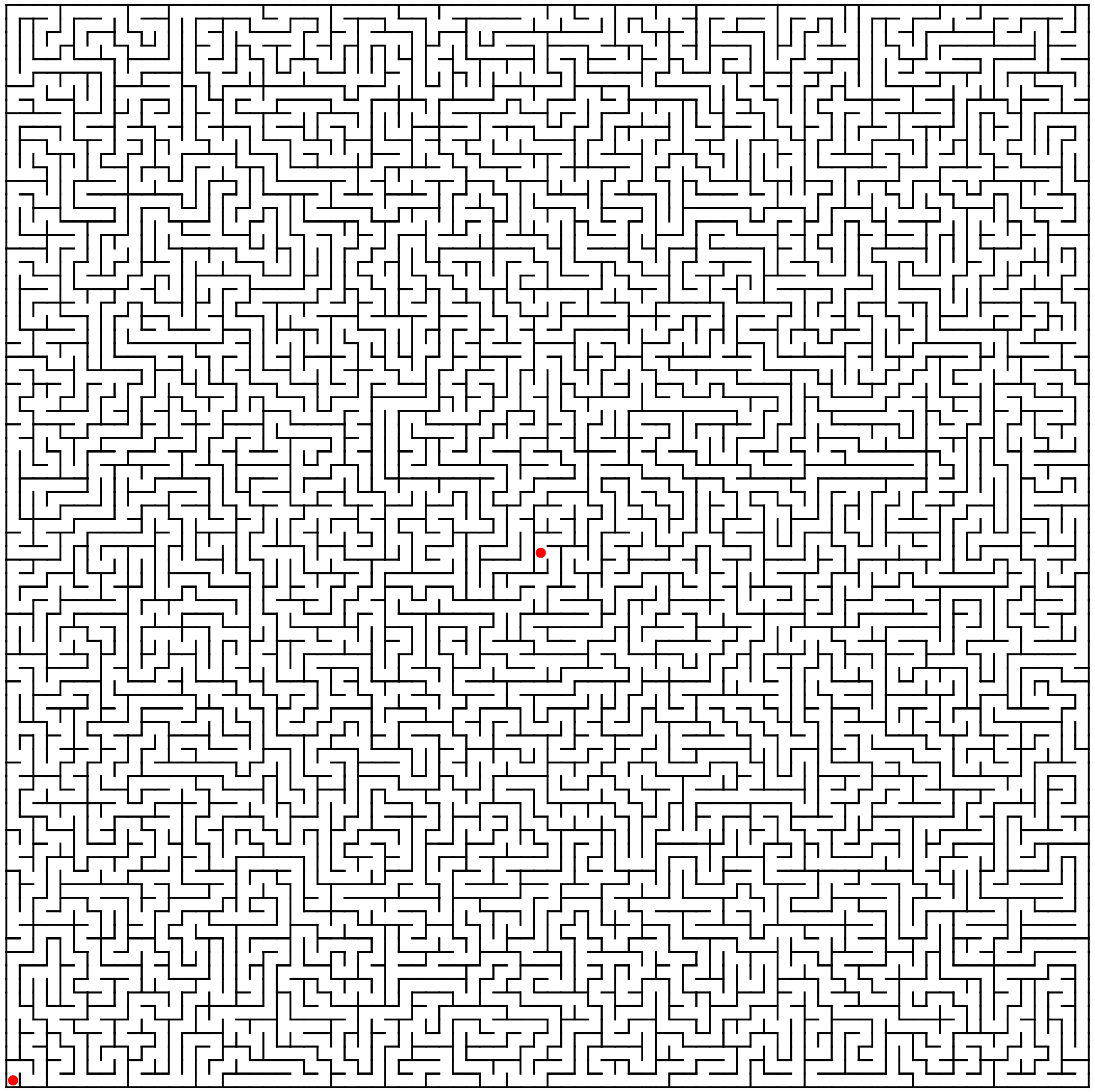
Maze Exploration Problem



Easy



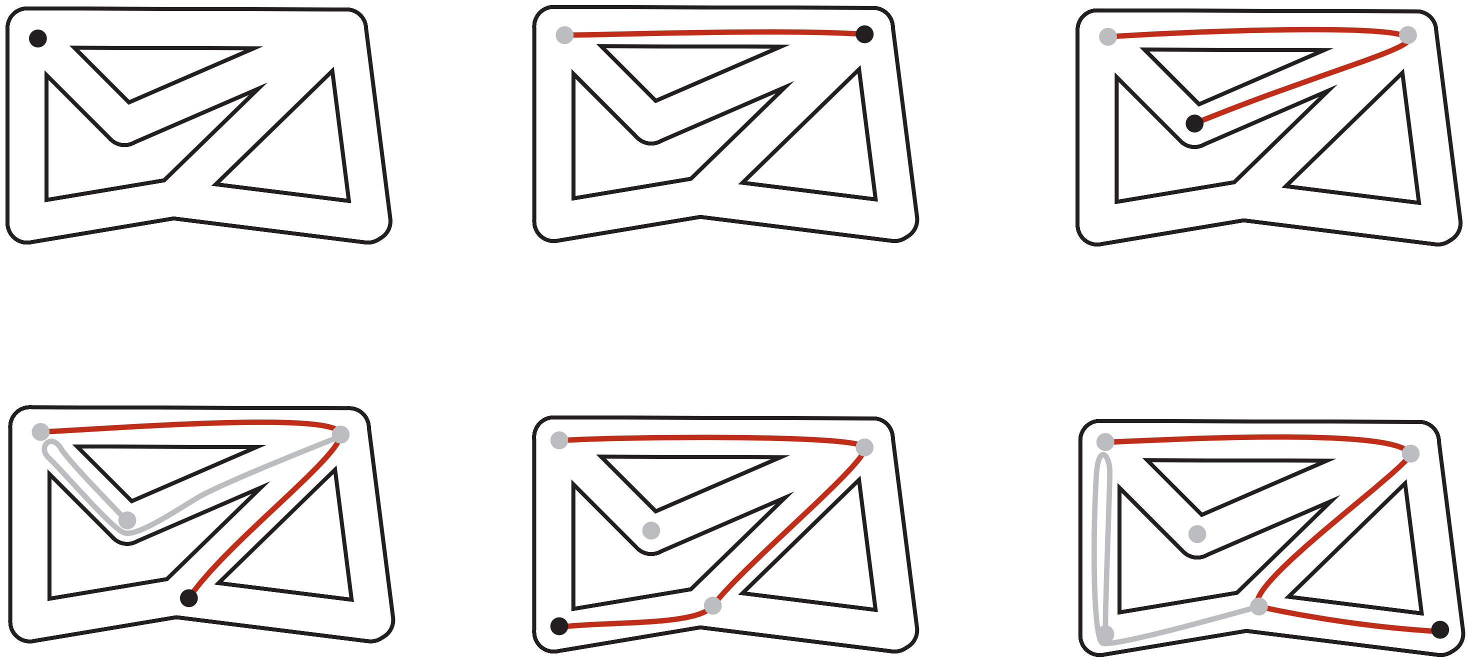
Medium



Challenged

Trémaux’s Algorithm

- Unroll a ball of string behind you
- Mark each visited intersection and each visited passage
- Backtrack when there is no unvisited options (hits a dead end)



Depth-First Search / Depth-First Traversal

- ***Depth-First Search*** (**DFS**): The idea is like exploring a maze
  - Follow the current path until you get stuck (hit a dead end)
  - Backtrack along breadcrumbs (~the string) until reach unvisited neighbor
  - Recursively explore
  - Careful not to repeat a vertex
- *Applications*: DFS can be used for testing a number of properties of graphs
  - Test whether there is a path from one vertex to another
  - Test whether a graph is connected
  - Test wether a graph has a cycle

DFS in Undirected Graph (1)

- Implements breadcrumb by using a map or a list for collecting the visited vertices during the search

```
DFS-init(s, adjList):
   visited = {}
   DFS-visit(s, adjList, visited)
```

- Use recursion when visiting neighbours of the current (starting) vertex `s`

```
DFS-visit(s, adjList, visited):
   visited = visited U {s}
   for v in adjList[s]:
      if v not in visited:
          DFS-visit(v, adjList, visited)
```

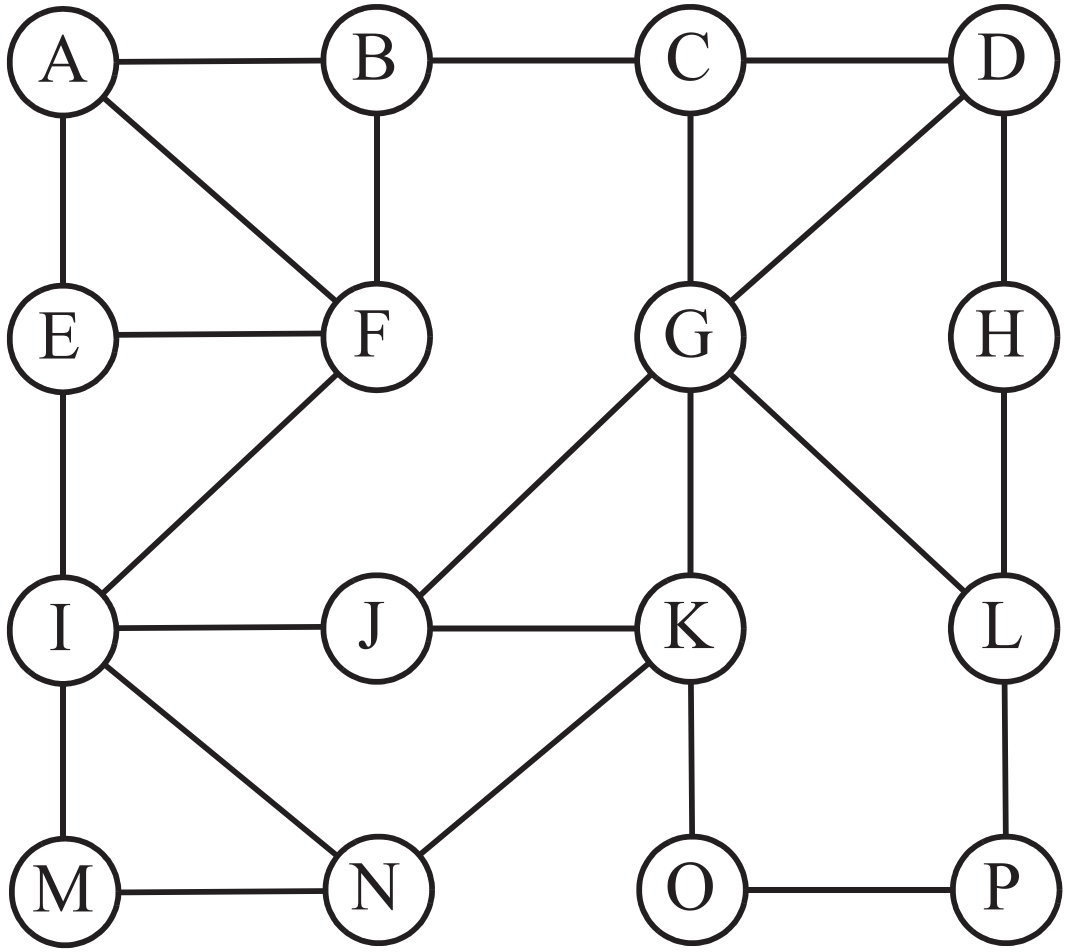
- *Remarks:* `DFS-visit(s, adjList, visited)`only sees stuff reachable from vertex `s`; only the connected component that contains `s`; Therefore, it will explore the entire graph if the graph is connected

DFS in Undirected Graph (2)

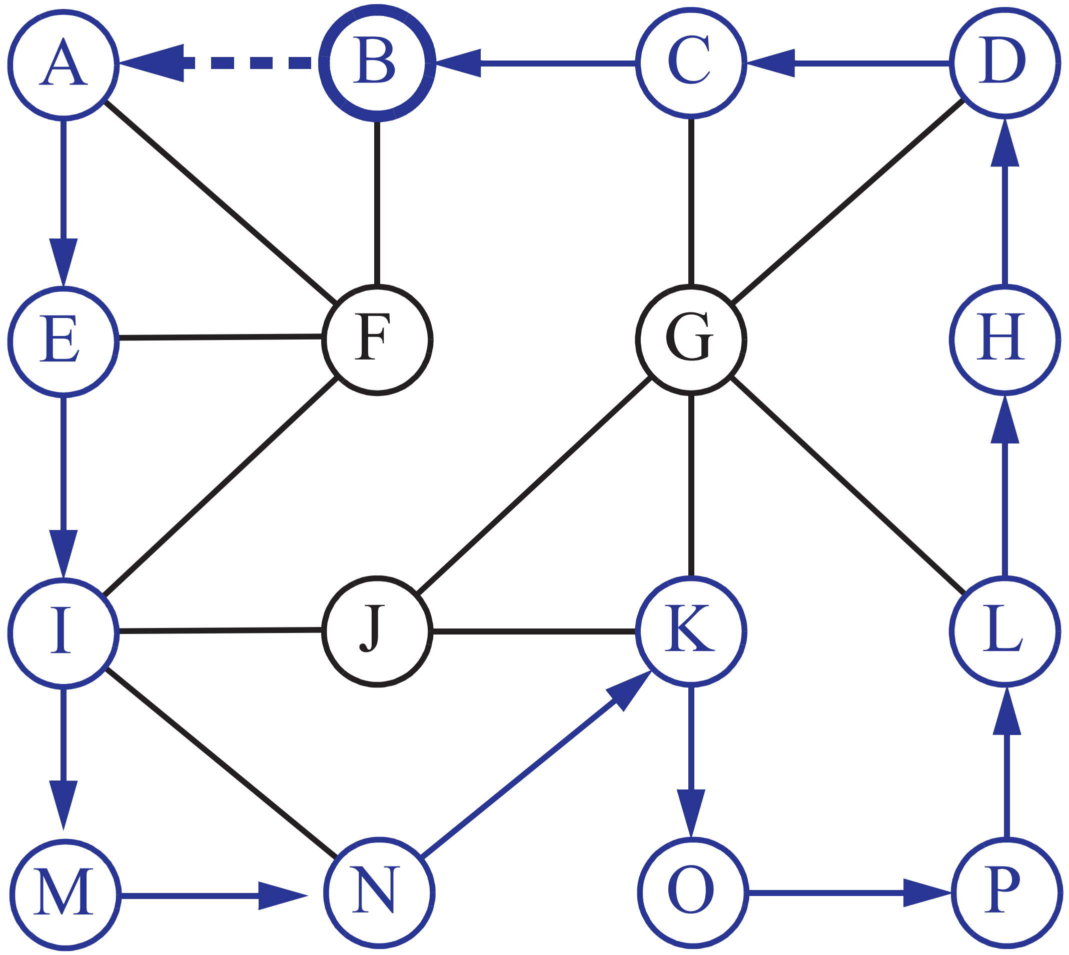
- To really explore the entire graph, we need to apply DFS at each unvisited vertices

```
DFS-explore(V, adjList):
    visited = {}
    for s in V:
        if s not in visited:
            DFS-visit(s, adjList, visited)
```

DFS Example (1)

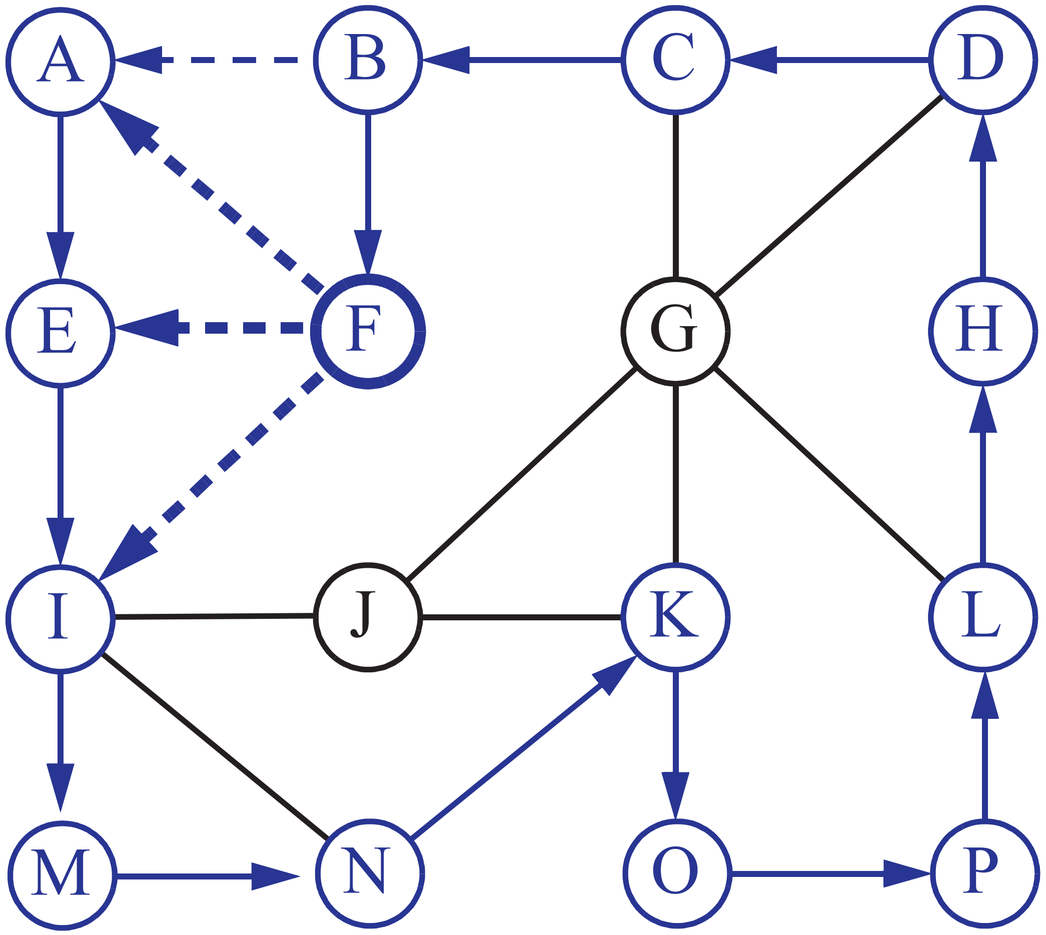


(a)

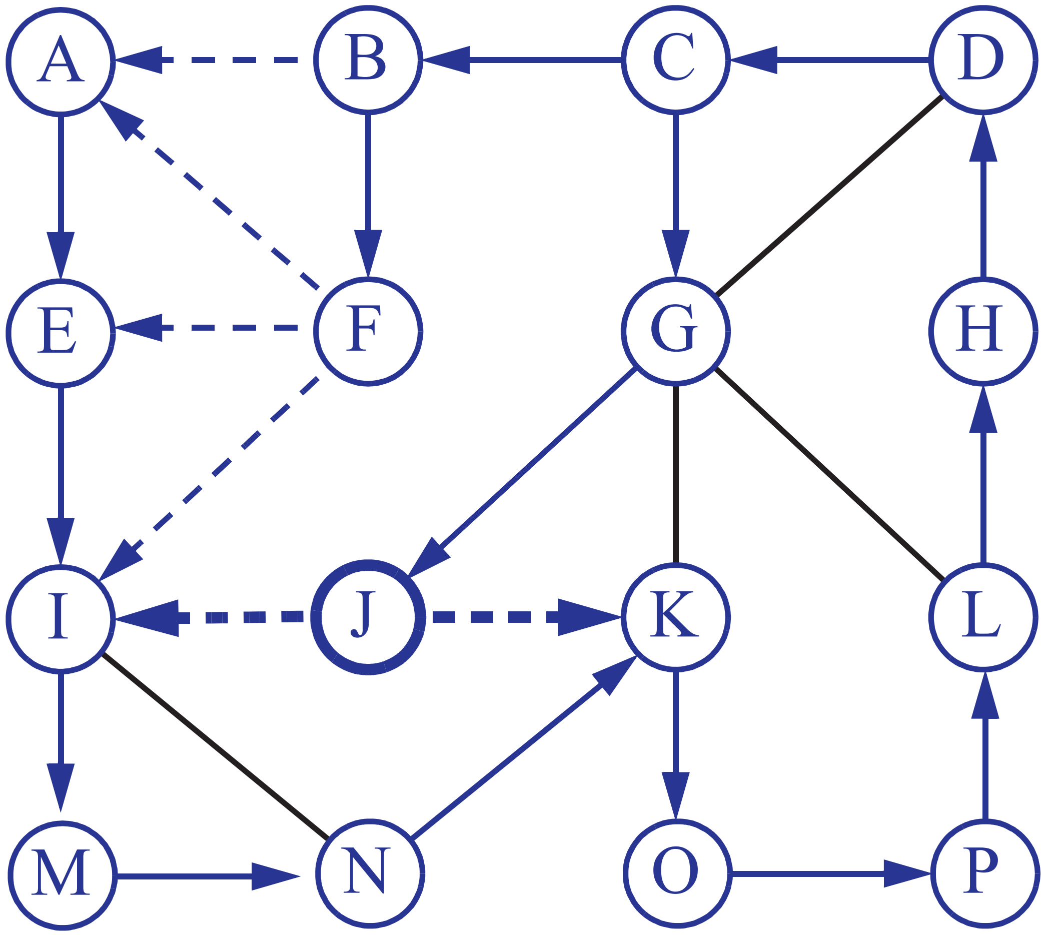


(b)

DFS Example (2)

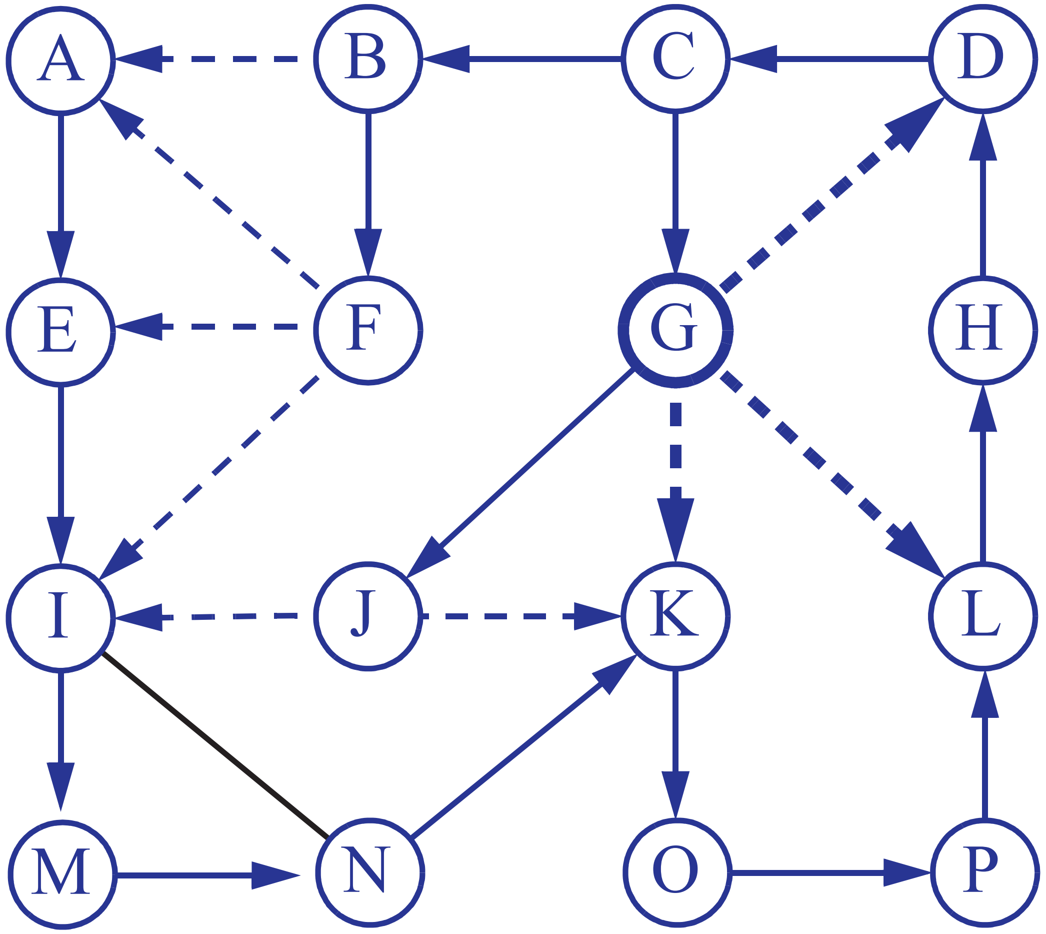


(c)

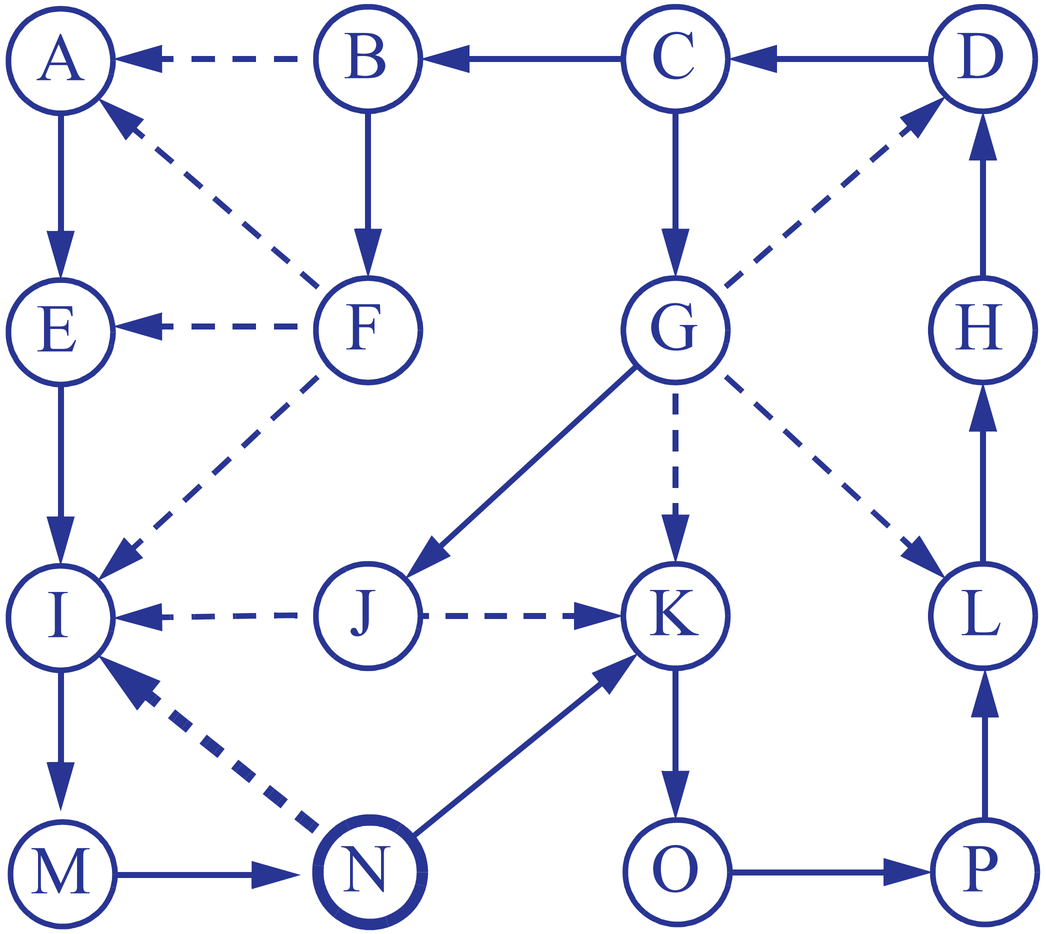


(d)

DFS Example (3)



(e)



(f)

DFS’s Complexity

- With adjacency-list representation, DFS takes time *O*(*n*+*m*) to traverse a graph with *n* vertices and *m* edges

- *Analysis*: `DFS–visit` gets called with a vertex `s`only once (before `parent[s]` is set)

- The time taken by `DFS–visit` in worst case can be

$$\sum_{s \in V} deg(s) = O(m)$$

- The time taken by `DFS–explore` just adds $O(n)$

DFS Implementation in C++ (1)

```
// C++ program to print DFS traversal from a given vertex in a  given graph
#include<iostream>
#include<list>
using namespace std;

// Graph class represents a undirected graph using adjacency list
representation
class Graph
{
    int V;    // No. of vertices
    list<int> *adj; // Pointer to an array containing adjacency lists

    void DFSVisit(int s, bool visited[]);  // A recursive function used by DFS

public:
    Graph(int V);   // Constructor
    void addEdge(int v, int w); // Function to add an edge to graph
    void DFSInit(int s); // DFS traversal of the vertices reachable from s
};
```

DFS Implementation in C++ (2)

```
Graph::Graph(int V)
{
    this->V = V;
    adj = new list<int>[V];
}

void Graph::addEdge(int v, int w)
{
    adj[v].push_back(w); // Add w to v’s list
    adj[w].push_back(v); // Add v to w’s list
}

void Graph::DFSVisit(int s, bool visited[])
{
    // Mark the current node as visited and print it
    visited[s] = true;
    cout << s << " ";

    // Recurse for all the vertices adjacent to this vertex
    list<int>::iterator i;
    for (i = adj[s].begin(); i != adj[s].end(); ++i)
        if (!visited[*i])
            DFSVisit(*i, visited);
}
```

DFS Implementation in C++ (3)

```
// DFS traversal of the vertices reachable from v
void Graph::DFSInit(int s)
{
    // Mark all the vertices as not visited
    bool *visited = new bool[V];
    for (int i = 0; i < V; i++)
        visited[i] = false;

    // Call the recursive helper function to print DFS traversal
    DFSVisit(s, visited);
}

int main()
{
    Graph g(4);
    g.addEdge(0, 1);
    g.addEdge(0, 2);
    g.addEdge(1, 2);
    g.addEdge(2, 0);
    g.addEdge(2, 3);

    cout << "Following is Depth First Traversal (starting from vertex 2) \n";
    g.DFSInit(2);
    return 0;
}
```

Output:

```
Following is Depth First Traversal (starting
from vertex 2)
2 0 1 3
```

Exercise

- As we have seen, the C++ implementation of DFS uses the technique of recursion.

- *Questions*:

    - a) How can we implement DFS in iterative version?

    - b) How can we modify DFS to check whether a graph is connected?

    - c) How can we modify DFS to check whether a graph has a cycle?