# COS 226 Lecture 17: Graph algorithms

GRAPH: a set of OBJECTS with CONNECTIONS

Interesting and useful abtract concept

Study of graph algorithms
    a challenging branch of computer science

Study of mathematical properties
    a challenging branch of discrete math

Hundreds of interesting algorithms known

Important applications abound
    * operations research
    * circuit simulation
    * software systems

# Glossary of terms

Node

Edge

Graph

Dense

Sparse

Path

Cycle

Tree

Spanning tree

Connected

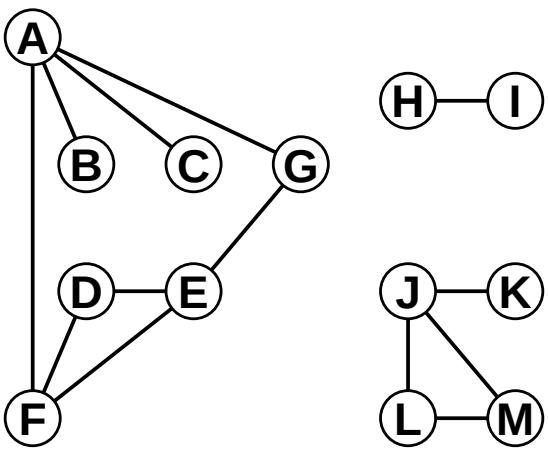
Connected component

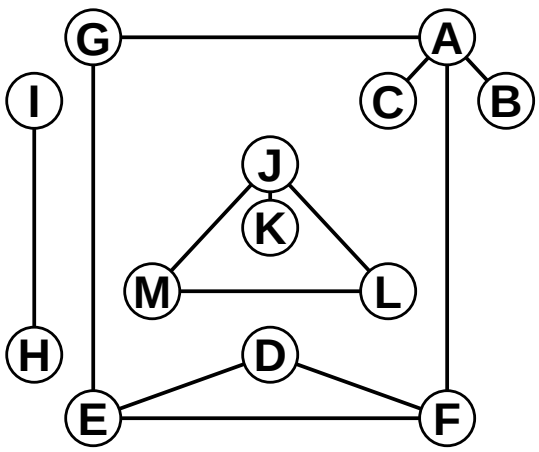
Directed

Undirected

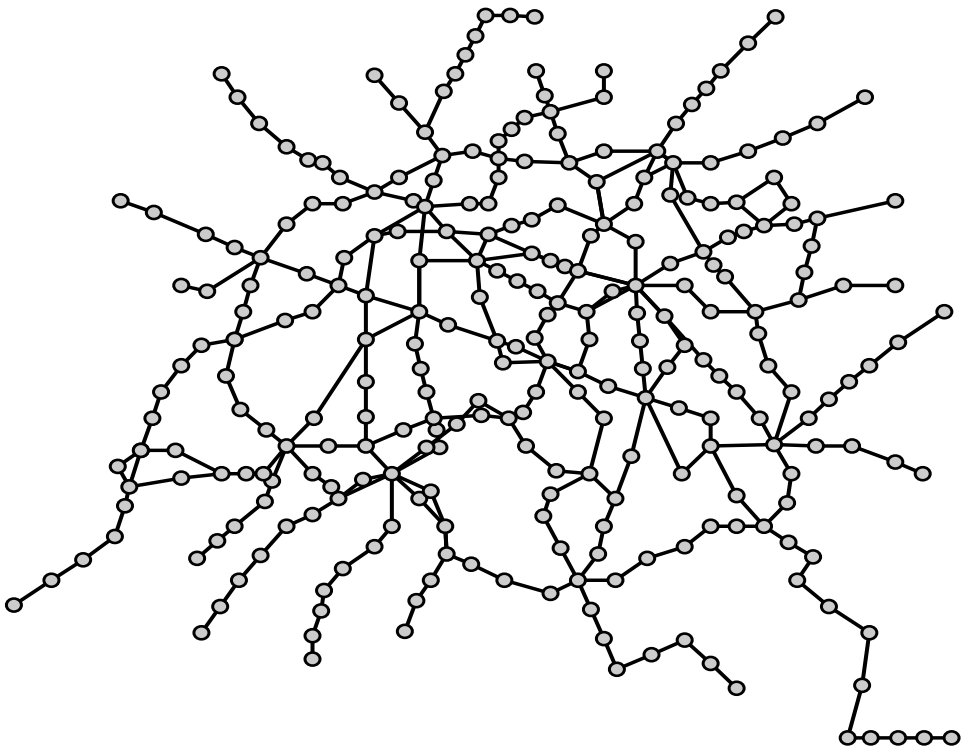
Weighted

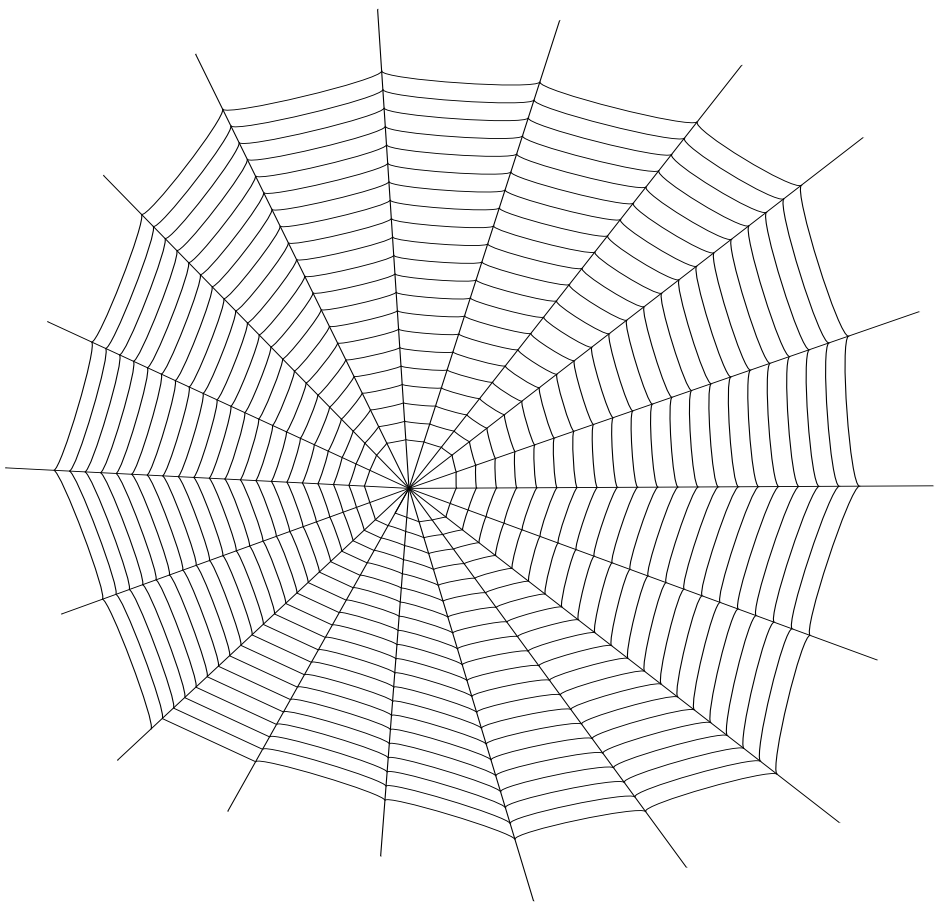
Network



# Graph examples (geometric problems)





## More graph examples

Anything involving relationships among items
can be modeled as a graph

Ex: Transportation network
Nodes: cities; Edges: roads
Ex: Electric circuit
Nodes: devices; Edges: wires

Warning: geometric intuition may mislead
Ex: Airline fares
(triangle equality might not hold)

Graphs are ABSTRACT objects, and model
other abstractions

Ex: Scheduling
Nodes: tasks
Edges: must do task A before task B
Ex: Programming system
Nodes: subroutine; Edges: A can call B
Ex: Finite state automaton
Ex: CFG
Nodes: symbols; Edges: productions
Ex: Game graphs
Nodes: board positions; Edges: moves

# Representing graphs

Graph are abstract mathematical objects

Algorithms have to work with
concrete representations

Many different representations possible

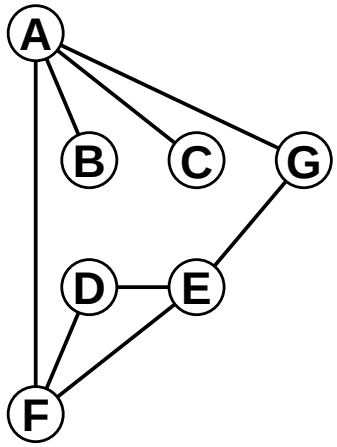
Efficiency depends on matching
representation with algorithm

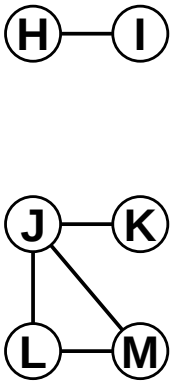
Nothing new here:
standard first step in algorithm design
is to pick representation (data structure)
* array vs. linked list
* integers
* points, lines, polygons
* trees

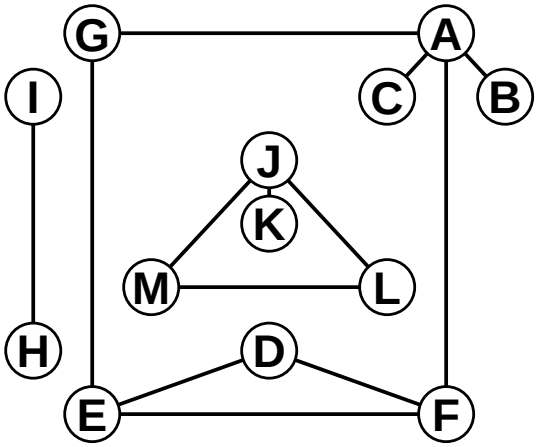
# Representing graphs (examples)

Node names

A B C D E F G H I




Adjacency lists

A: F C B G

B: A

C: A

D: F E

E: G F D

F: A E D

G: E A

H: I

I: H

J: K L M

K: J

L: J M

M: J L

## Representing graphs (more examples)

Set of edges

AB AG AC LM JM JL JK ED FD HI FE AF GE

Adjacency matrix

| | A | B | C | D | E | F | G | H | I | J | K | L | M |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **A** | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| **B** | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| **C** | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| **D** | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| **E** | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| **F** | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| **G** | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| **H** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 |
| **I** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 |
| **J** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| **K** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 |
| **L** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 1 |
| **M** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 1 |

Implicit edges

"calculate" edges only when needed

## Summary of costs

E edges, V vertices

Space requirements:

- Adjacency lists: V+E
- Adjacency matrix: V^2
- Set of edges: E (+V)

Choice of representation affects algorithm efficiency

Trivial examples:

- Is there an edge from A to B?
  - lists: O(E)
  - matrix: O(1)
- Is there an edge from A to anywhere?
  - lists: O(1)
  - matrix: O(E)

## Basic graph problems (short list)

Is there a path from A to B?

CYCLES

Does the graph contain a cycle?

CONNECTIVITY (SPANNING TREE)

Is there a way connect the vertices?

BICONNECTIVITY

Will the graph become disconnected if one vertex is removed?

PLANARITY

Is there a way to draw the graph without edges crossing?

SHORTEST PATH

What is the shortest way from A to B?

LONGEST PATH

What is the longest way from A to B?

MINIMAL SPANNING TREE

What is the best way connect the vertices?

HAMILTONIAN CIRCUIT

Is there a way to visit all the vertices without visiting the same vertex twice?

TRAVELING SALESMAN

What is the shortest route to connect the vert without visiting the same vertex twice?

ISOMORPHISM

Do two given representations represent the same abstract graph?

# Traversing graphs

Goal: "visit" every node on the graph

Depth-first search (DFS)

- Recursive traversal algorithm
- * Mark all nodes as "unvisited"
- * To visit a node k
  - * mark it
  - * (recursively) visit all unmarked nodes connected to k by an edge
- * Visit node 1, then visit any unvisited node, continue until all nodes visited

Solves some simple graph problems

- * connectivity
- * cycles

Basis for solving difficult graph problems

- * biconnectivity
- * planarity

## DFS implementation (recursive)

```
int val[maxV]; int id = 0;
dfs()
   { int k;
      for (k = 1; k <= V; k++) val[k] = 0;
      for (k = 1; k <= V; k++)
         if (val[k] == 0) visit(k);
   }
```

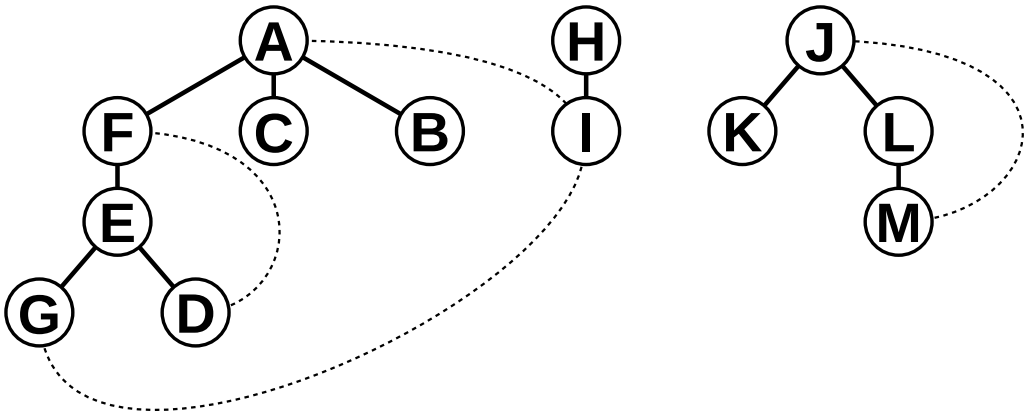
Adjacency matrix

```
visit(int k)
   { int t;
      val[k] = ++id;
      for (t = 1; t <= V; t++)
         if (a[k][t] != 0)
            if (val[t] == 0) visit(t);
   }
```

Adjacency list

```
visit(int k)
   { struct node *t;
      val[k] = ++id;
      for (t = adj[k]; t != z; t = t->next)
         if (val[t->v] == 0) visit(t->v);
   }
```

# DFS example (adjacency lists)




A: F C B G

B: A

C: A

D: F E

E: G F D

F: A E D

G: E A

H: I

I: H

J: K L M

K: J

L: J M

M: J L

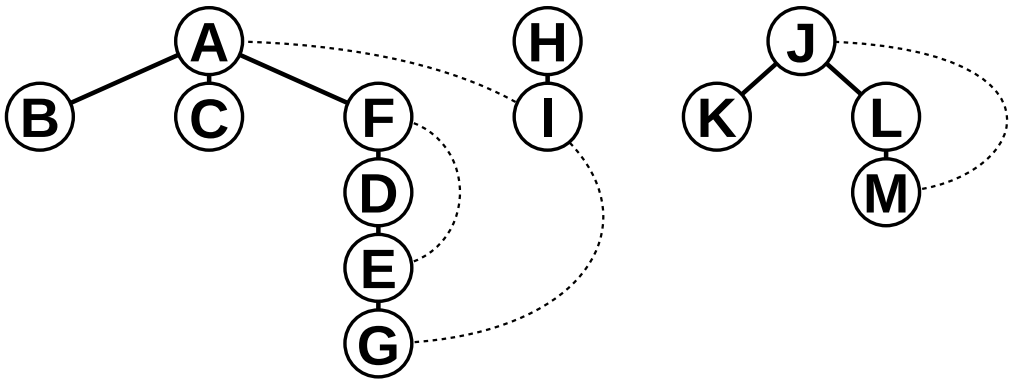
## DFS example (adjacency lists)

```
visit A
    visit F (first on A's list)
        check A on F's list (been there)
        visit E (second on F's list)
            visit G (first on E's list)
                check E on G's list (been there)
                check A on G's list (been there)
            check F on E's list (been there)
            visit D (third on E's list)
                check F on D's list (been there)
                check E on D's list (been there)
        check D on F's list (done that)
    visit C (second on A's list)
    visit B (third on A's list)
    check G on F's list (done that)
    ...
```

"been there": currently working on it

"done that": totally finished dealing with it

# DFS example (adjacency matrix)




Adjacency matrix

| | A | B | C | D | E | F | G | H | I | J | K | L | M |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| A | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| B | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| C | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| D | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| E | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| F | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| G | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| H | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 |
| I | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 |
| J | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| K | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 |
| L | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 1 |
| M | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 1 |

## DFS example (adjacency matrix)

```
visit A
    check A on A's list (been there)
    visit B (second on A's list)
        check A on B's list (been there)
        check B on B's list (been there)
    visit C (third on A's list)
        check A on C's list (been there)
        check C on C's list (been there)
    visit F (fourth on A's list)
        check A on F's list (been there)
        visit D (second on F's list)
            check D on D's list (been there)
            visit E (second on D's list)
                check D on E's list (been there)
                check E on E's list (been there)
                check F on E's list (been there)
                visit G (fourth on E's list)
                    check A (been there)
                    check E (been there)
                    check G (been there)
            check F on D's list (been there)
        check E on F's list (done that)
        check F on F's list (been there)
    check G on A's list (done that)
    ...
```

## DFS implementation (stack-based)

Use explicit stack instead of recursive calls

```
int val[maxV]; int id = 0;
visit(int k)
   { struct node *t;
      push(k);
      while (!stackempty())
         {
            k = pop(); val[k] = ++id;
            for (t = adj[k]; t != z; t = t->next)
               if (val[t->v] == 0)
                  { push(t->v); val[t->v] = -1; }
         }
   }
listdfs()
   { int k;
      stackinit();
      for (k = 1; k <= V; k++) val[k] = 0;
      for (k = 1; k <= V; k++)
         if (val[k] == 0) visit(k);
   }
```

## DFS example (adjacency lists, stack-based)

```
visit A
    push F, push C, push B, push G
    visit G
        push E, been to A
        visit E
            been to G, been to F, push D
            visit D
                been to F, been to E
    visit B
        been to A
    visit C
        been to A
    visit F
        been to A, done with E, done with D
```

## Stack-based search

NOT the same as recursive DFS. Why?

Algs differ in treatement of nodes adjacent to partially visited nodes

- DFS: visits such a node
- stack-based: treats it as "visited"

Automatic recursion removal would stack

- local variable (next node on adj. list)

Can get same effect by deleting (if present)

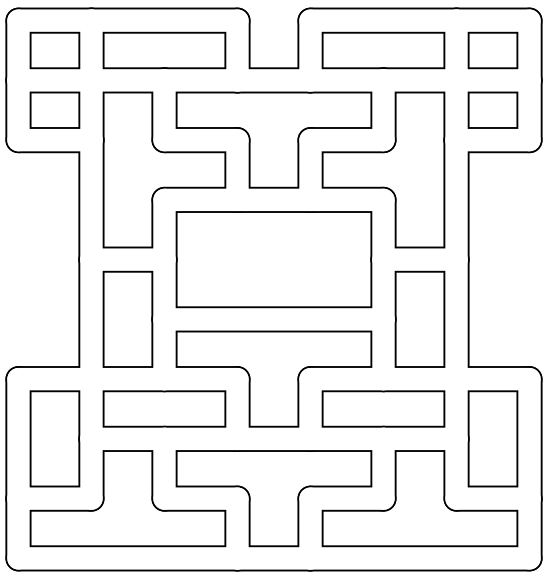
- before pushing (more general ADT than stack)

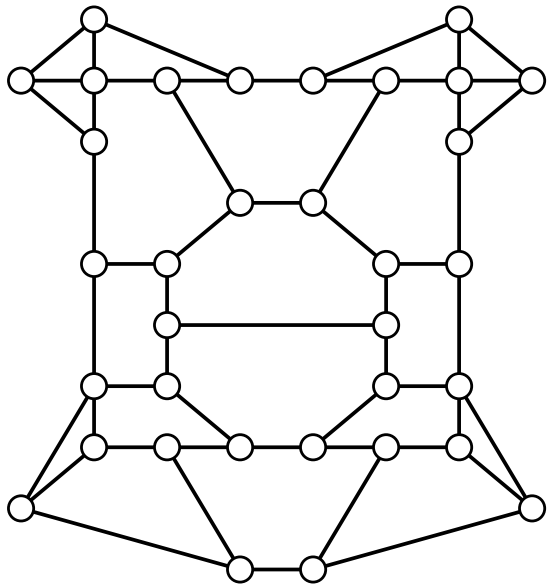
No particular reason to use stack

- other ADTs work as well (stay tuned)

Analogous to traversing a maze

# Graphs and mazes





Nodes: intersections

Edges: hallways

DFS

mark ENTRY and EXIT halls at each vertex

leave by ENTRY when no unmarked halls

Stack-based?

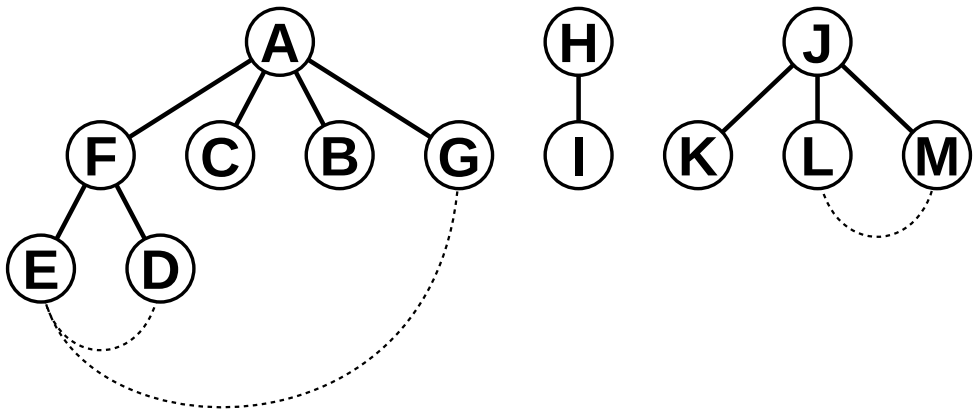
light each intersection

if you can see a light when arriving at an intersection, turn back

leave by ENTRY when no unmarked halls

What characterizes a "recursive" method??

# Breadth-first search

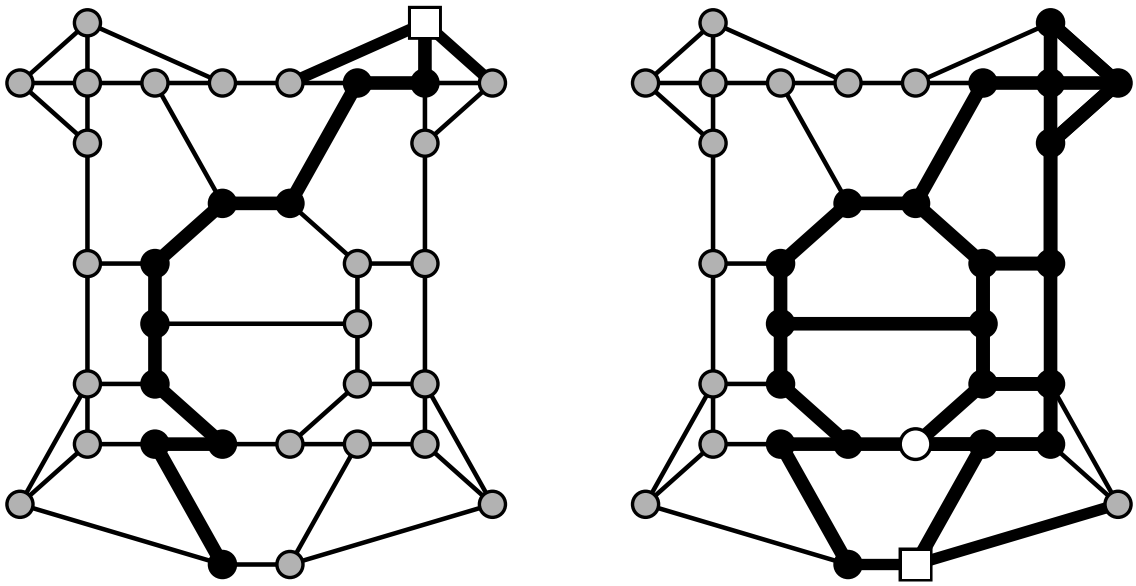



Put unvisited nodes on a QUEUE, not a stack

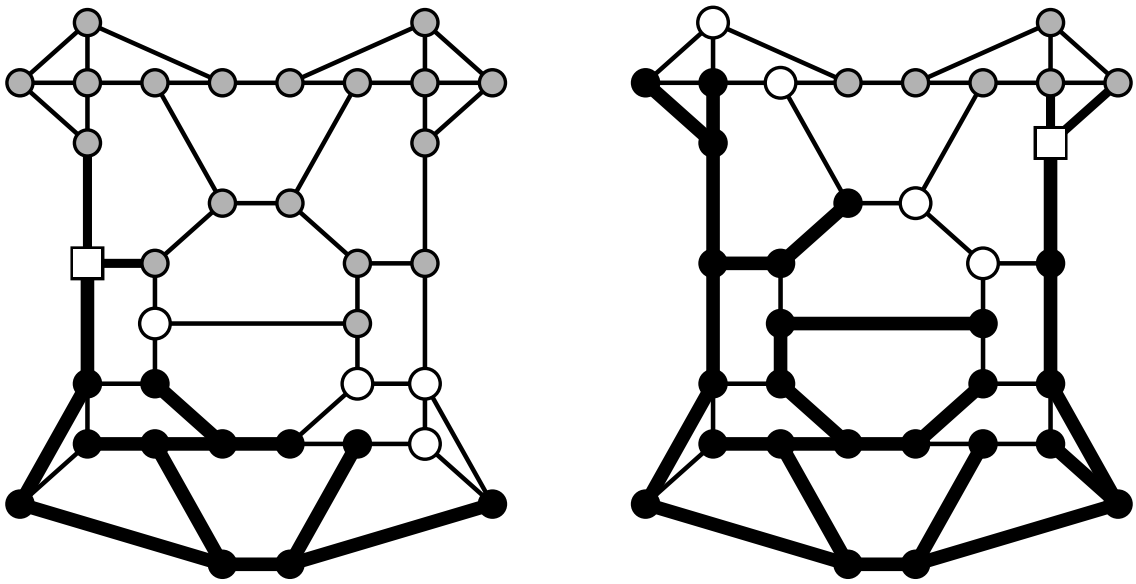
```
visit(int k)
   { struct node *t;
      put(k);
      while (!queueempty())
         {
            k = get(); val[k] = ++id;
            for (t = adj[k]; t != z; t = t->next)
               if (val[t->v] == 0)
                  { put(t->v); val[t->v] = -1; }
         }
   }
bfs()
   { int k;
      queueinit();
      for (k = 1; k <= V; k++) val[k] = 0;
      for (k = 1; k <= V; k++)
         if (val[k] == 0) visit(k);
   }
```

# BFS vs DFS example

Depth-first search



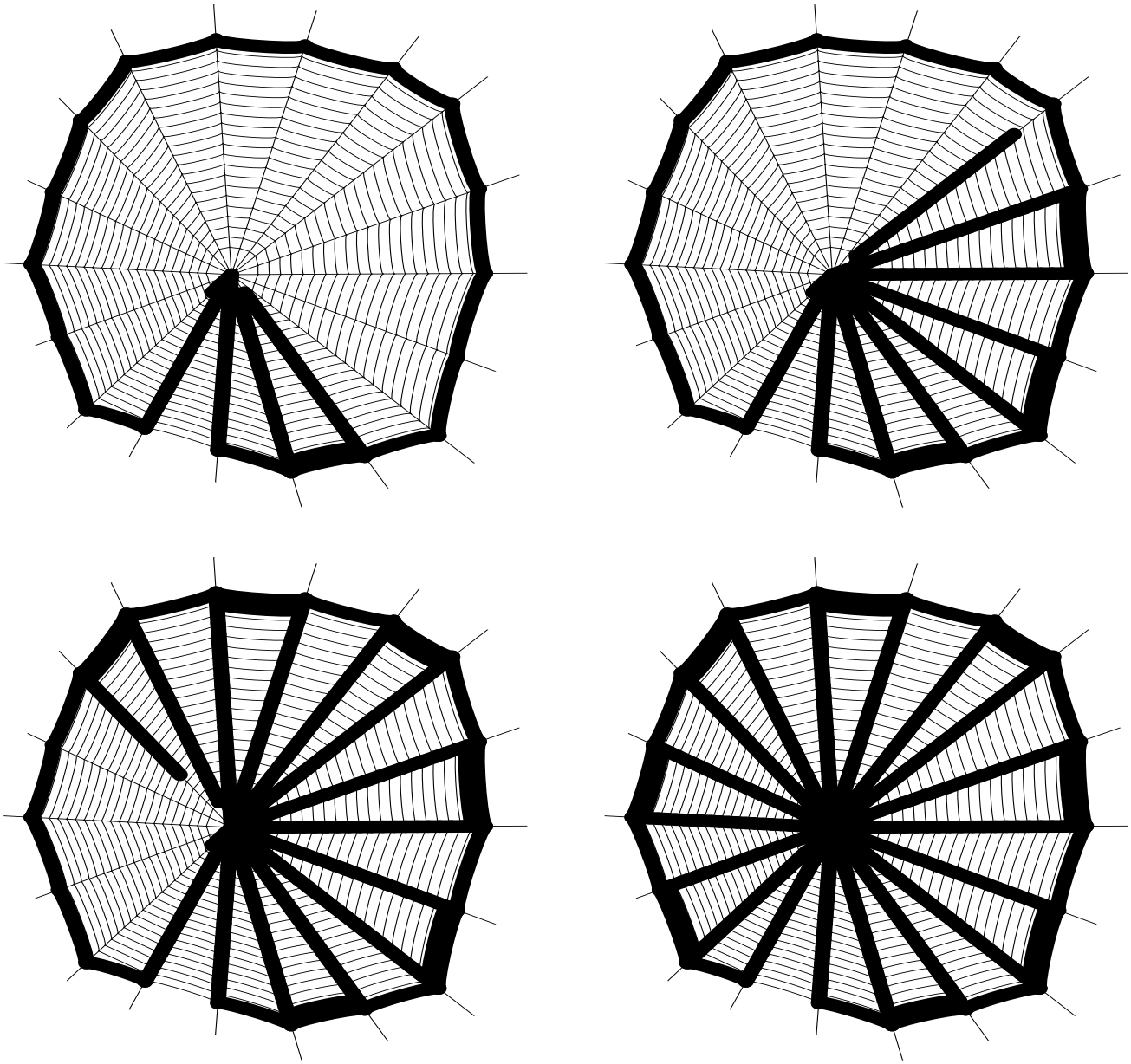
Breadth-first search



DFS: "sequential"

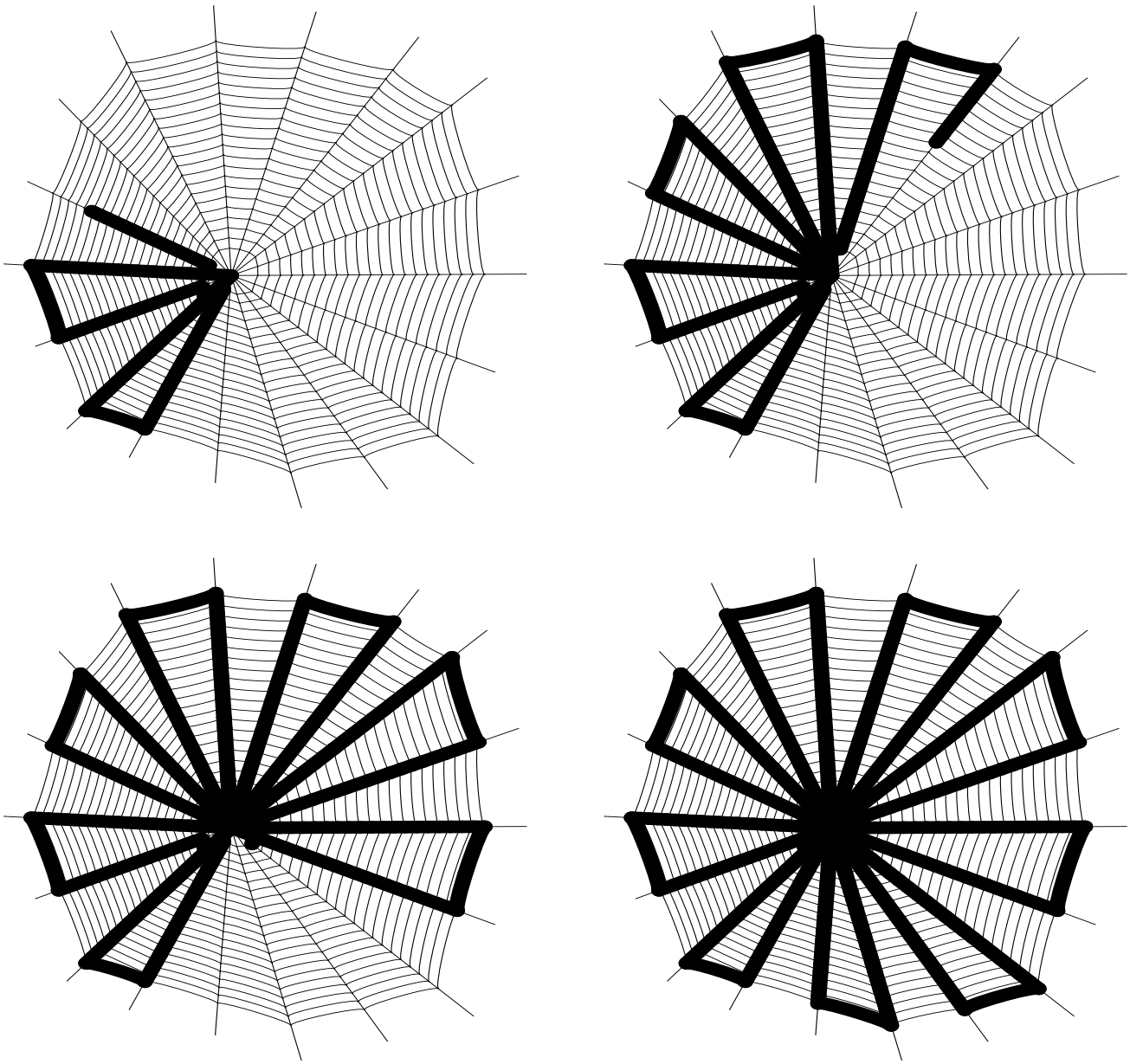
BFS: "parallel"

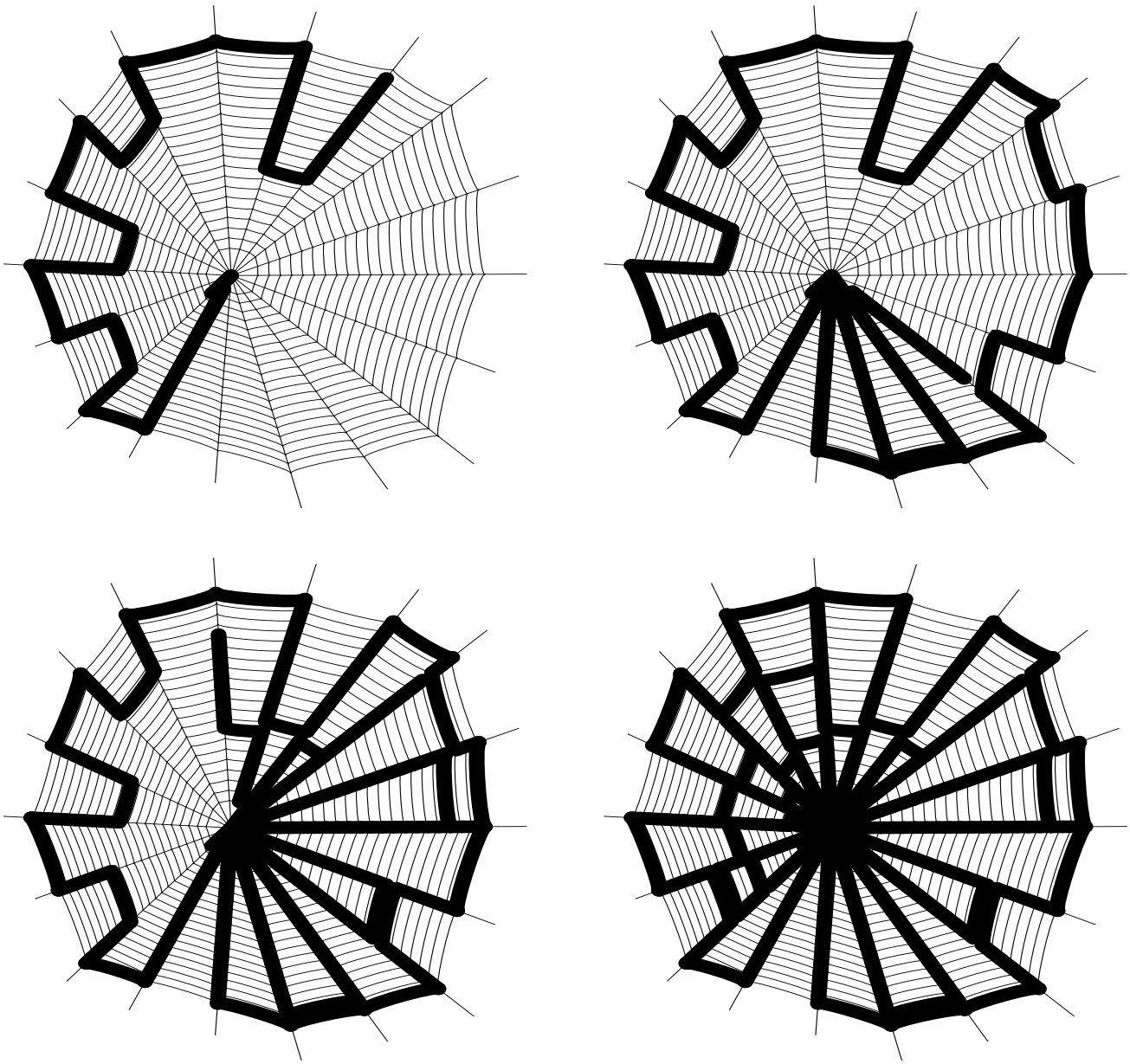
# DFS example



# DFS example (continued)

Same graph, different order of edges on adjacency lists



# DFS example (continued)

Same graph, "randomized" choice of edges on adjacency lists