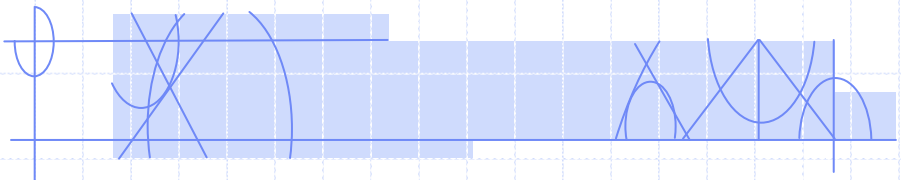
# COMP 3760

# Algorithm Analysis and Design

## Lesson 14: DFS & BFS



Rob Neilson

rneilson@bcit.ca

# Today's Agenda

- DFS (Depth First Search)
- BFS (Breadth First Search)

## Reading

- Textbook: chp 1.4, chp 5.2

## Homework exercises

The following questions are due in class at the start of lab during the week of Nov 3-7

- chapter 5.2, questions 1, 2, 3, 4, 6, 7

# Our First Graph Algo: Depth First Search

## Purpose: traverse a graph structure

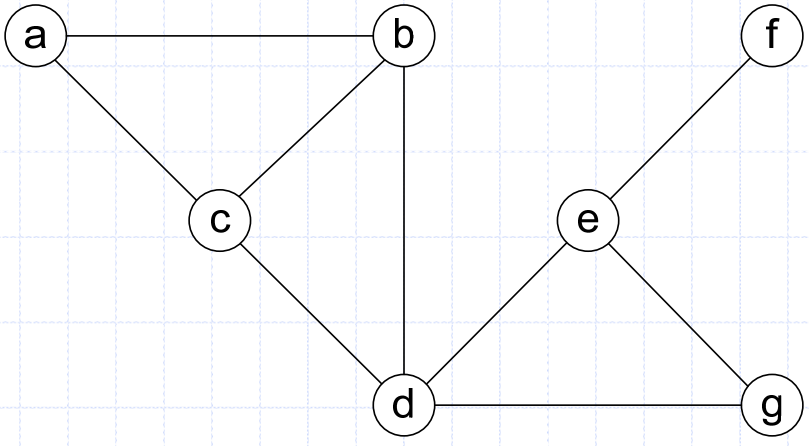
### Algorithm (informally):

*Do not assume a connected graph …*

> Idea: "visit" all the vertices, one at a time, ***marking*** them as we visit them

- for each v ∈ V
  - examine it against the "search criteria", exit if found
  - proceed to an unvisited vertex that is adjacent to v
    - when there is a choice of v's to visit, you need to decide which one to visit first
  - if there are no unvisited vertices adjacent to the current one (dead end), back up one vertex and try again
  - continue until all nodes have been visited, or, until search is successful

### Example:



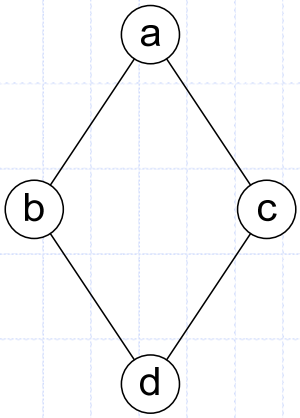
# DFS

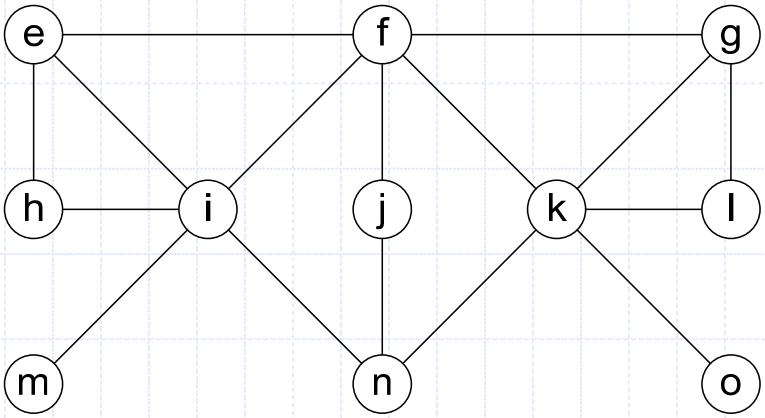
Algorithm (formally):

```
DFS(G):
   init all visited values to false
   for each vertex v in V    // where G = {V,E}
        if v has not been visited
                dfs(v)
dfs(v)
   visit node v
   for each vertex w in V adacent to v
        if w has not been visited
                dfs(w)
```

- the stmts "visit node v" should be replaced by whatever you are doing
- use a temp 2D matrix of size nxn to store the visits, or maintain a visit attribute in each node
- the output is typically a "***DFS Tree***", which is a tree containing all the edges that are used to visit node
- edges that are in G, but not in the DFS Tree are called "***back edges***"

# DFS Example (using the algo)




Notes: To trace the operation algorithm we use a stack.

When we make a recursive call (eg (dfs(v)), we push v onto the stack.

When v becomes a dead-end (ie: no more adjacent unvisited neighbours) it is popped off the stack.

Typically we break ties for next unvisited neighbour by using alphabetical order.

# Uses of DFS

DFS is commonly used to:

- find a spanning tree
- find a path from v to u (ie: get out of a maze)
- find a cycle
- find all connected components

# Efficiency of DFS

- the basic operation is:

```
for each vertex w in V adacent to v
    if w has not been visited
        dfs(w)
```

- we can see that this operation will be performed once for each vertex that occurs in the underlying graph structure
  - therefore the #basic ops depends on the size of the structure used to implement the graph

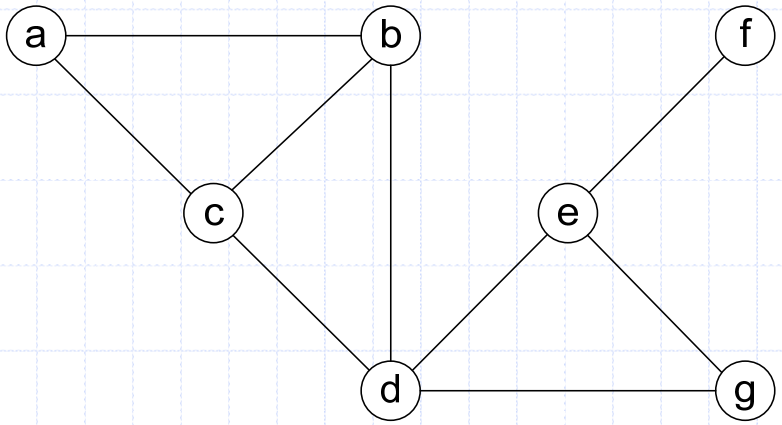
- basically we need to visit each element of the data structure exactly once. so the efficiency must be:
  - $O(|V|^2)$ - for adjacency matrix
  - $O(|V|+|E|)$ for adjacency lists

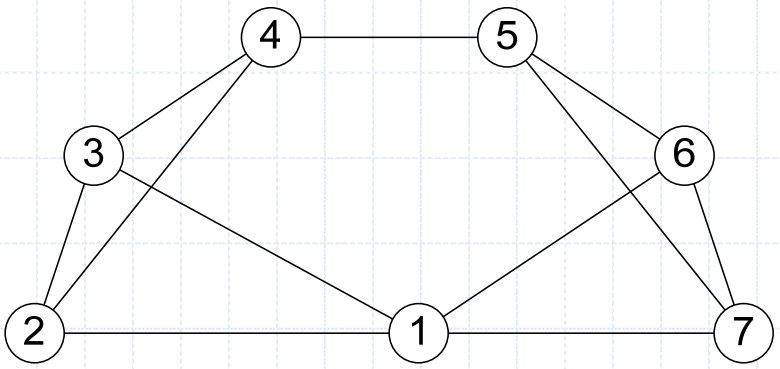
# Breadth First Search

Informally:

- for each vertex v in V
- visit all vertices adjacent to v
- when all vertices have been visited, visit all vertices 2 hops away
- continue in this way until all have been visited

Examples:



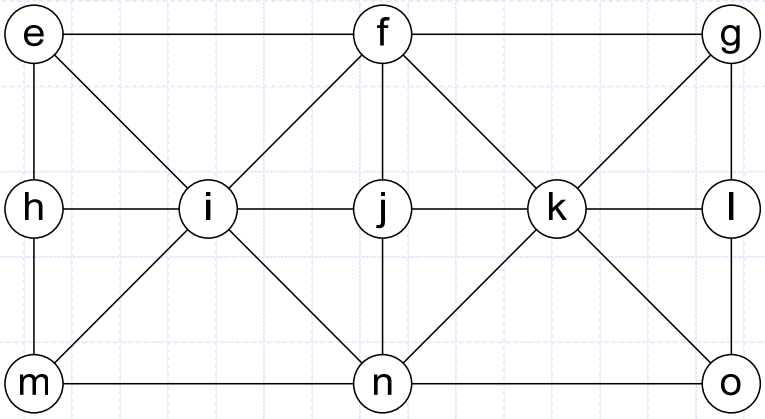



# BFS Algorithm

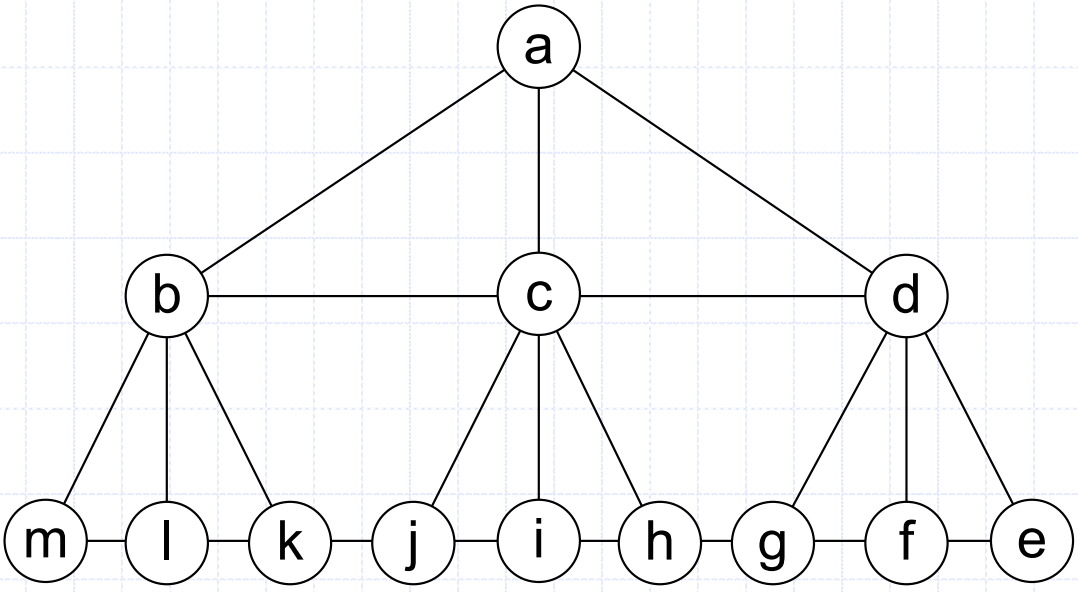
```
BFS(G):
   init all visited flags to false
   for each v in V
        if v has not been visited
                bfs(v)
bfs(v)
   visit node v
   initialize a queue Q
   add v to Q
   while Q is not empty
        for each w adjacent to Q.head
                if w has not been visited
                        visit node w
                        add w to Q
        remove Q.head from Q
```

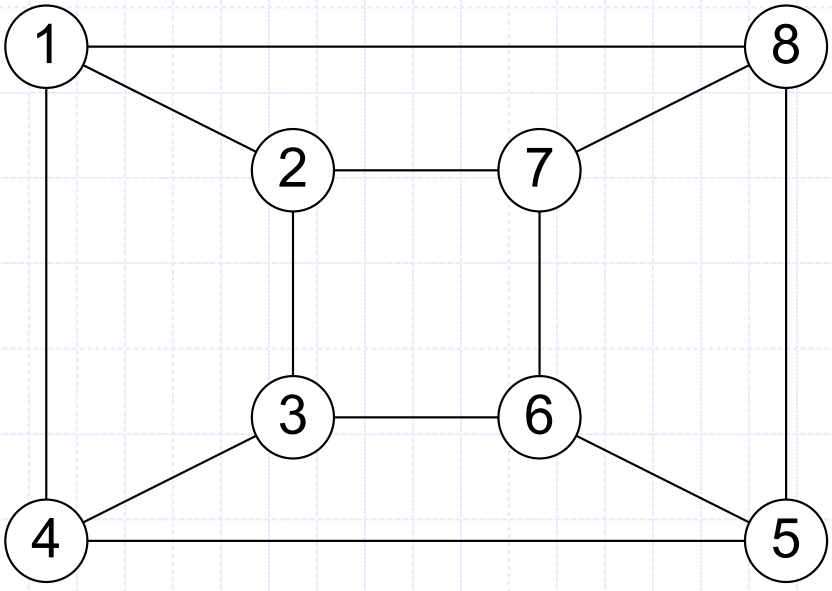
- the stmts "visit node v|w" can be replaced by whatever you are doing
- use a queue (FIFO) to determine which vertex to visit next
- edges that are in G, but not in the resulting BFS tree are called ***cross-edges***

# More examples (using the algorithm)



# BFS vs DFS



# BFS: Efficiency & Application

- Efficiency is the same as DFS

Uses of BFS:

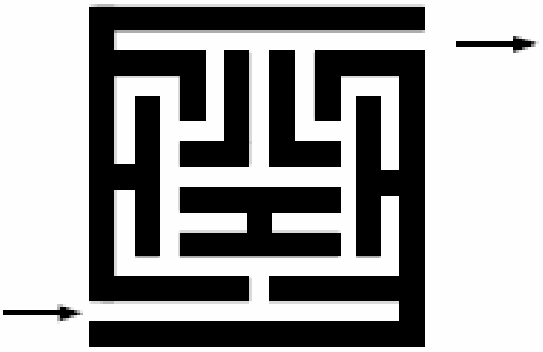
- Finding all connected components in a graph
- Traversing all nodes within one connected component
- Finding the shortest path (number hops) between two connected vertices

# Problem 1: Spanning Tree

- Given a connected graph G, use BFS or DFS to construct a spanning tree of G.
  - use BFS so that we get “shorter” paths between vertices
  - this is a straight-up application of BFS, just build a new graph (the spanner) as we go

# Problem 2: Maze Solving

- Model the following maze as a graph. Use DFS to find a path through the maze
  - use DFS because its tree is constructed by moving along existing edges (in contrast, BFS keeps back-tracking to the parent node, so you would have to walk further)
  - modify the algorithm so that it discards edges if we back-track along them (ie: we just want to keep edges that will be in the final solution – which is a path)
    - do this by looking at what is on the stack when we find the end of the maze



# Problem 3: Shortest Path

- Use BFS to find the shortest path between two connected vertices, u and w
  - use BFS because it will find a shortest path (DFS will find “a path” – not always the shortest one)

Step 1: run bfs(u) to create a spanning tree T rooted at u (all paths from in T, starting at root, are shortest… why?)

Step 2: extract the path from T

- use DFS on T, to find any path (as in the previous problem), OR,
- let BFS maintain a queue of paths, rather than a queue of vertices, OR,
- maintain a hash of references to parents, and reconstruct after

# Problem 4: Find Cycles

- Explain how you can use BFS or DFS to determine if a graph is acyclic
  - either will work
  - need to quit when we have a cycle (we attempt to visit a neighbor, but it has already been visited)

# Problem 5: Determine Connectivity

- Explain how you can use BFS or DFS to determine if a graph is connected
  - either will work
  - modify the first loop so that it calls dfs|bfs on any vertex. If there are any unvisited vertices when it returns, the graph is not connected

# Problem 6: Checking Bipartite

- Explain how you can use BFS or DFS to determine if a connected graph is bipartite
  - either will work
  - a two-coloring is given by the resulting dfs/bfs tree, such that vertices on alternate levels of the tree are colored differently
  - as long as there are no back/cross edges that connect vertices from adjacent levels, the graph is bipartite
  - the algorithm should traverse the graph, marking vertices as even or odd at each level change. If we encounter an edge between two vertices of the same color, the graph is not bipartite

# The End