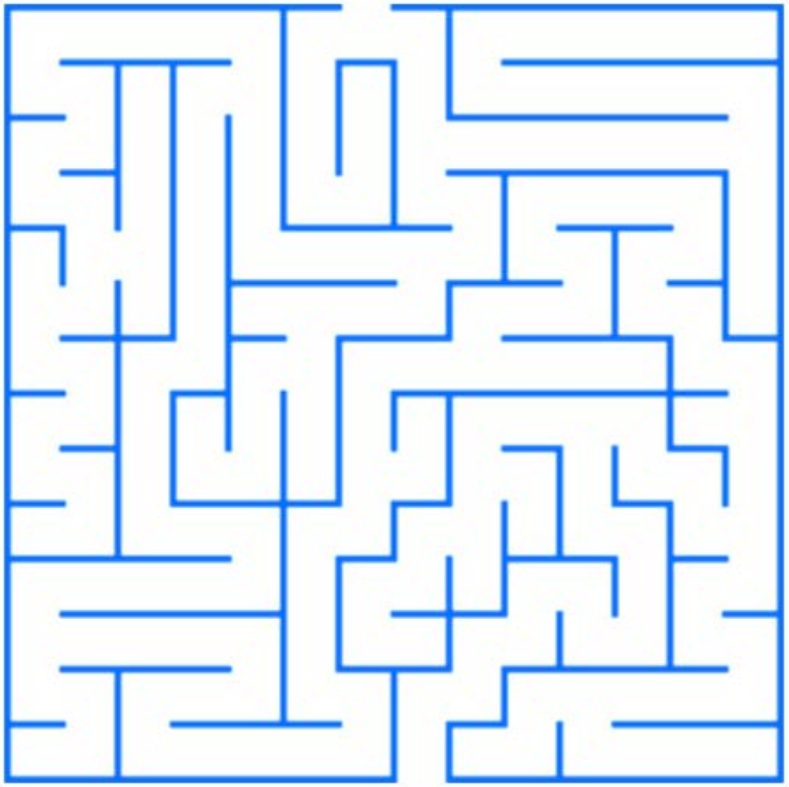
Graph Algorithms: DFS and BFS traversals

Textbook: Chapter 3.5

Depth-first search (DFS)

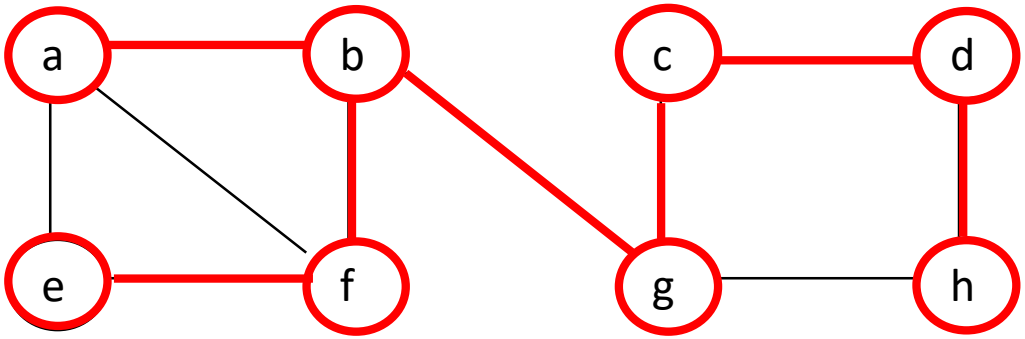
- Think about how you might try to find your way out of a maze



Depth-first search (DFS)

- Visits all vertices by always *moving away* from the last vertex visited (if possible)
  - Backtracks at “dead-ends” (no adjacent, unvisited vertices)
- Implementation often uses a stack of vertices being processed
- Follows a tree-like route throughout the graph

DFS example



DFS order: a b f e g c d h

Some notes on DFS

- To track the progress of the algorithm we use a stack
  - When we make a recursive call (e.g. (dfs(v)), we push v onto the stack
  - When v is a dead-end (i.e. no more neighbors to visit) it is popped off the stack
- Our convention: break ties for “next neighbor” by using some natural order
- Typical results from running DFS can be:
  - List of vertices in order visited
  - List of vertices in order of “dead-ends” (when popped from stack)
  - DFS Tree – tree containing all the edges that were used to visit nodes
    - Unused edges of G (edges not in DFS tree) are called “back edges”

DFS algorithm

```
Algorithm Depth_First_Search(Graph G)
// Graph G = {V,E}
   initialize visited to false for all vertices
   for each vertex v in V
      if v has not been visited
         dfs(v)

function dfs(Vertex v)
   visit node v
   for each vertex w in V adjacent to v
      if w has not been visited
         dfs(w)
```

- "Visit node v" means doing whatever you need to do at each node

Common uses of DFS

- Find a spanning tree of a graph
- Find a path between two vertices v and u
- Find a path out of a maze
- Determine if a graph has a cycle
- Find all the connected components of a graph
- Search the state-space of problems for a solution (AI)
- Many more!

Efficiency of DFS

- The basic operation is the if statement inside the loop in the dfs() function:

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

- Each time dfs() is called, this loop examines all the neighbors of some vertex v … eventually it looks at ALL the neighbors of ALL the vertices
  - Therefore the number of basic operations depends on the data structure used to implement the graph

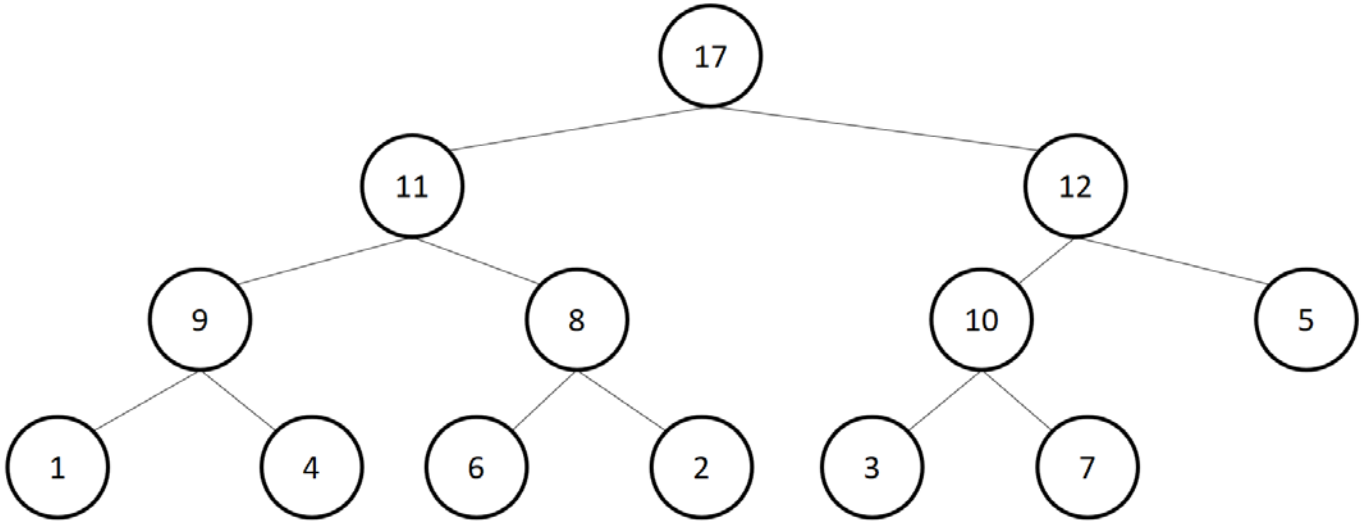
- Basically we need to visit each element of the data structure exactly once. So the efficiency must be:
  - $O(|V|^2)$ for adjacency matrix
  - $O(|V|+|E|)$ for adjacency lists

Which is worse?

- $O(|V|^2)$
- $O(|V|+|E|)$

Breadth-first search (BFS)

- Recall the way we use an array to store a (very specific type of) binary tree being used as a heap.



17 11 12 9 8 10 5 1 4 6 2 3 7

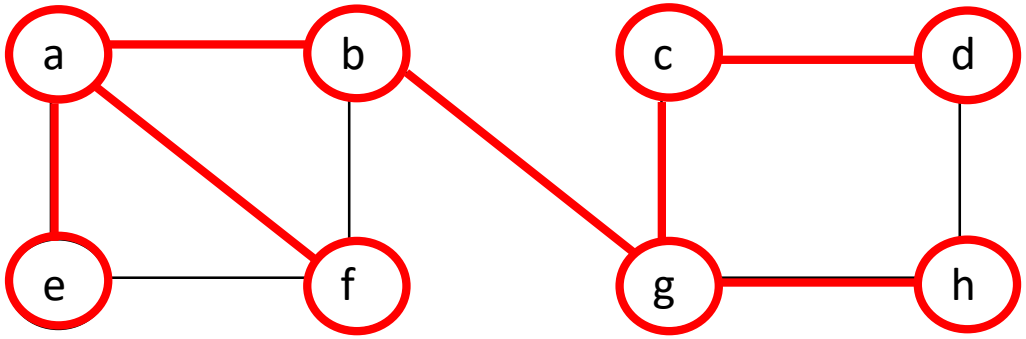
Breadth-first search (BFS)

- Visit all neighbors “the same distance” from starting vertex
  - Visit immediate neighbors first
  - Then the neighbors of all those vertices
  - Etc.

- Instead of a stack, BFS uses a queue

- Follows a tree-like route throughout the graph, but perhaps a different tree than DFS

BFS example



BFS order: a b e f g c h d

BFS algorithm

```
Algorithm Breadth_First_Search(Graph G)
// Graph G = {V,E}
   initialize visited to false for all vertices
   for each vertex v in V
      if v has not been visited
         bfs(v)

function bfs(Vertex 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
      Q.dequeue()
```

- Uses 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”

Notes on BFS

- Same efficiency as DFS:
  - Adjacency matrix: $O(|V|^2)$
  - Adjacency list: $O(|V|+|E|)$

- Yields just one ordering of vertices (order added/deleted from queue is the same)
  - Whereas with DFS, the order that vertices are *visited* may be different from the order they get *finished* (become dead-ends)

BFS applications

- Really the same as DFS
- Sometimes one or the other may be better for specific problems

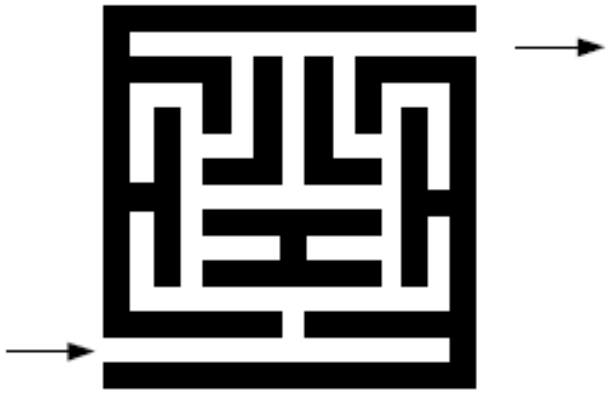
Solving problems using graph traversals

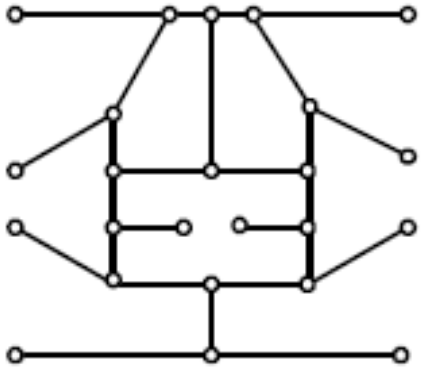
Problem 1: Spanning Tree

- Given a connected graph G, find a spanning tree T
  - This is a straight-up application of BFS (or DFS)
  - Build a new graph (the spanning tree) as we go
    - Initialize a new graph T
    - Each time we visit a vertex, add the edge we used to T
- BFS or DFS?
  - BFS usually gives shorter paths between vertices

Problem 2: Solving a Maze

- Represent maze as a graph
  - Nodes for start, finish, intersections, and dead-ends
  - Find a path from *start* to *finish*
- BFS or DFS?
  - If interested in end-result:
    - BFS will find the shortest total path
  - If actually walking while solving:
    - DFS tends to result in less actual walking
    - BFS backtracks to parent nodes too often





Problem 3: Shortest Path

- Find the shortest path between two vertices u and w
- BFS or DFS?
  - BFS will find a shortest path
  - DFS will find ***a*** path – maybe not the shortest one

- Idea of algorithm (and why it works):
  1. Use bfs(u) to create a spanning tree T with u as the root. Note that all paths that appear in T are the shortest paths from u to their respective vertices
  2. Use DFS on T to find a path from u to w (as in the maze problem)

Problem 4: Determine Connectivity

- Determine if a graph is connected
- BFS or DFS?
  - Either will work
  - Think about this yourself!
  - What modification(s) do you need to make to the algorithm to answer this question?

Practice problems

1. Chapter 3.5, page 128, questions 1,2,4,10