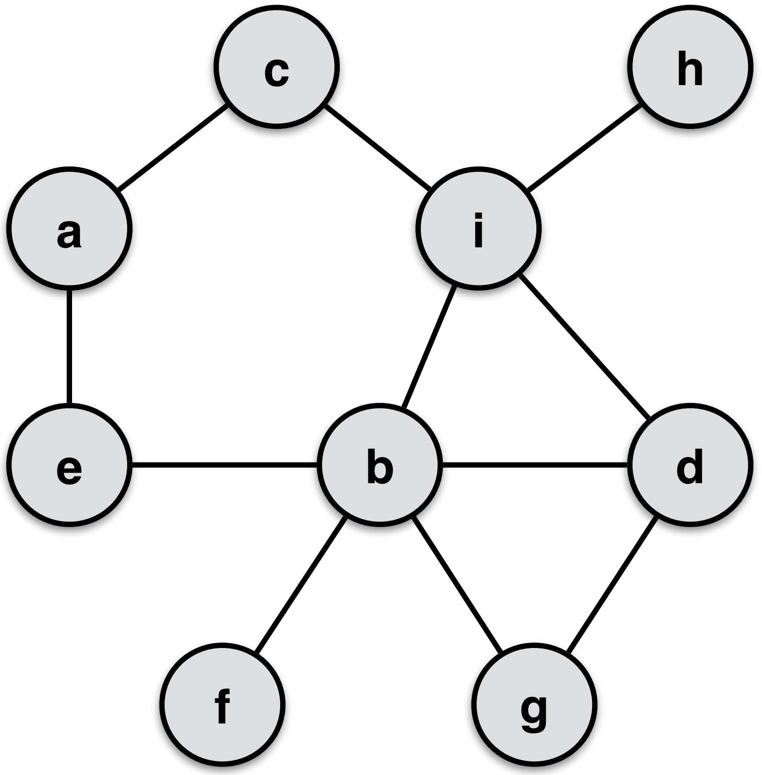
Graph Traversals

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

There are only 10 kinds of people in this world:
those who know binary and those who don't.

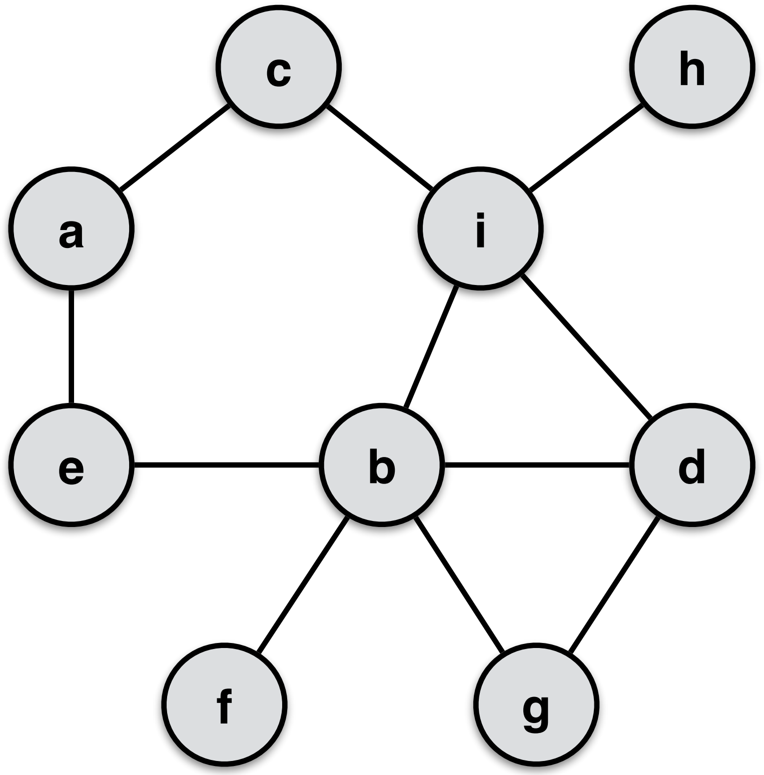
Blind Search

- Suppose we want to find a path from one vertex to another in a graph
  - ex. find a route on a map from your current location to some destination
  - ex. determine the sequence of mutual friends connecting you to someone else
- This problem is called ***graph search***.
- There are many approaches, depending on the info. we have available.
- When all we have is the graph structure itself (i.e., nothing to help us know or guess how close we are to the goal), we're essentially doing ***blind search***.
- We'll look at a couple of approaches in general (thinking of *undirected* graphs), then extend our thinking to directed graphs, along with a way to associate extra information with the vertices (or edges).



Blind Search: DFS

- **Basic Idea:** Think “maze search” — wander as far down a path as you can until you hit a “dead end.” If this happens, back up to the nearest vertex that still has incident edges that are “unvisited.” Repeat. DFS terminates when there are no more unvisited vertices.*
- **Uses:**
    - Determining if there is a path from Vertex v to Vertex u.
    - Determining if a graph is *connected*.
    - Determining if there are *cycles* in the graph.
    - etc.



Blind Search: DFS

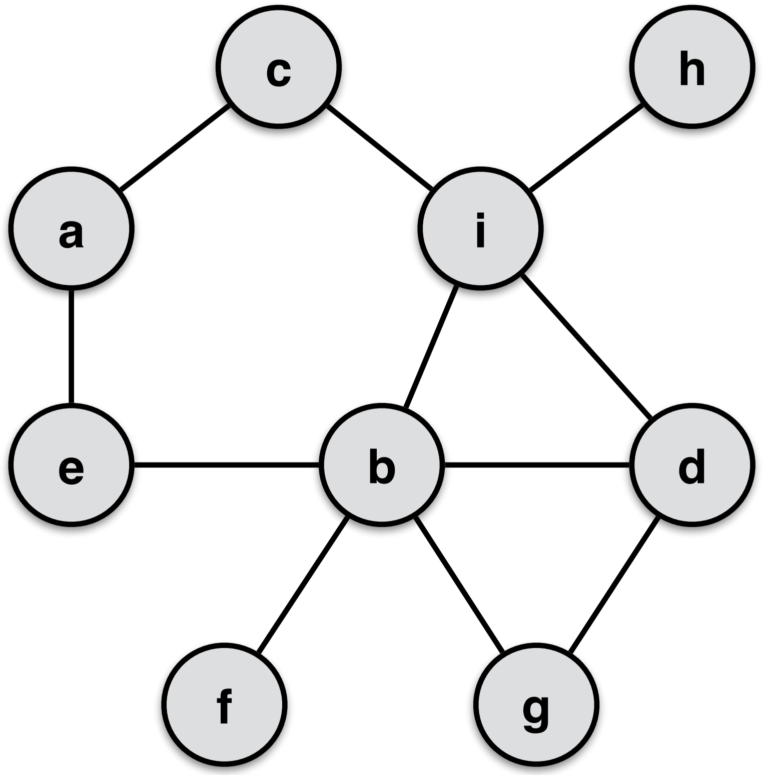
```
Push the start vertex v onto the stack
Repeat until we find the goal vertex or the stack is empty:
  Pop the next vertex v from the stack
  If v has not been visited
    Mark v as visited (and maybe do some processing)
    for all vertices v' that are adjacent to v
      If we haven't already visited v':
        push v' onto the stack
```

The textbook provides a sample recursive implementation of DFS — check it out.

Blind Search: DFS

```
Push the start vertex v onto the stack
Repeat until we find the goal vertex or the stack is empty:
  Pop the next vertex v from the stack
  If v has not been visited
    Mark v as visited (and maybe do some processing)
    for all vertices v' that are adjacent to v
      If we haven't already visited v':
        push v' onto the stack
```

[Demo In Class]



Blind Search: BFS

- **Basic Idea:** Think of BFS like a "layered" search — send out many “explorers” that collectively traverse a graph in a coordinated fashion. Explorers proceed by “levels”, starting at some vertex (level 0), then working out to all vertices distance 1 away (level 1), … distance 2 away (level 2), …
- Of course we can’t actually send out a multitude of “explorers” every time, so instead we do things one at a time, collecting vertices and “visiting” them *in the order that we discover them*.
- Similar idea to DFS — the only minor changes needed to go from DFS to BFS…
    - Change **stack** to **queue**.
- **Uses:**
    - Finding ***shortest path*** between Vertex s (start) and Vertex u.
    - Determining if a graph is *connected*.
    - etc.

Blind Search: BFS

DFS

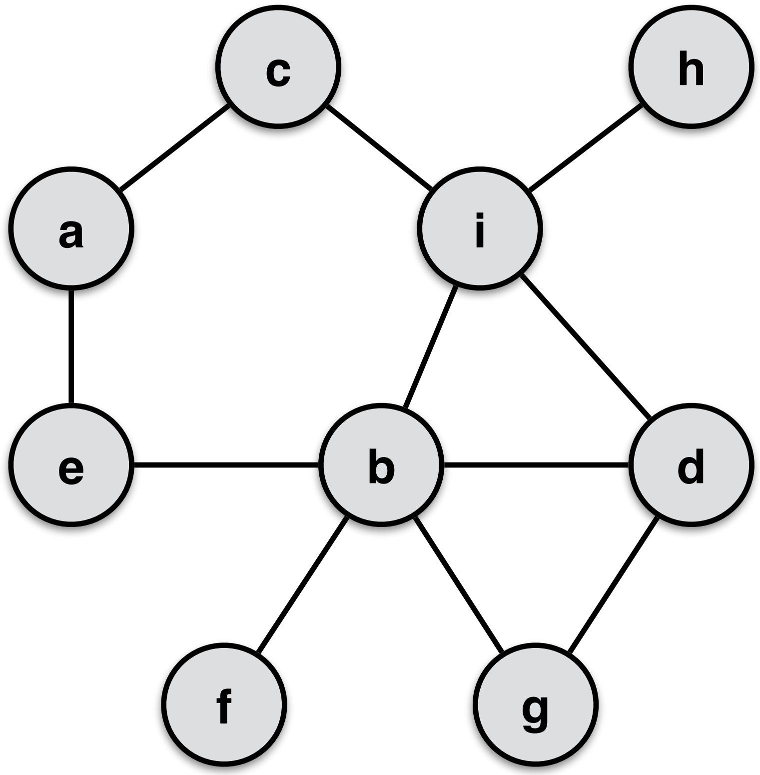
```
Push the start vertex v onto the stack
Repeat until we find the goal vertex or the stack is empty:
  Pop the next vertex v from the stack
  If v has not been visited
    Mark v as visited (and maybe do some processing)
    for all vertices v' that are adjacent to v
      If we haven't already visited v':
        push v' onto the stack
```

```
Enqueue the start vertex v onto the queue
Repeat until we find the goal vertex or the queue is empty:
  Dequeue the next vertex v from the queue
  If v has not been visited
    Mark v as visited (and maybe do some processing)
    for all vertices v' that are adjacent to v
      If we haven't already visited v':
        enqueue v' onto the queue
```

Blind Search: BFS

```
Enqueue the start vertex v onto the queue
Repeat until we find the goal vertex or the queue is empty:
  Dequeue the next vertex v from the queue
  If v has not been visited
    Mark v as visited (and maybe do some processing)
    for all vertices v' that are adjacent to v
      If we haven't already visited v':
        enqueue v' onto the queue
```

[Demo In Class]



Blind Search: BFS (optimized)

```
Enqueue the start vertex v onto the queue and mark it
Repeat until we find the goal vertex or the queue is empty:
  Dequeue the next vertex v from the queue
    (Maybe do some processing)
    for all vertices v' that are adjacent to v
      If we haven't already visited v':
        mark v' (and maybe do some processing)
        enqueue v' onto the queue
```

NOTE: *visit the node when we first discover it and avoid enqueuing the same vertex multiple times.*

Directed Graphs

- Recall: directed graphs distinguish an edge **from A to B** vs. the one **from B to A**.
- We'll use directed graphs to store **BFS** trees.
- Additional methods needed for *Directed Graphs*:
  - **isDirected(e)**: Test if e is a directed edge.
  - **insertDirectedEdge(v, w, label)**: Create a directed edge from v to w with value label
- Also — I find it useful to add four additional methods:
  - **incidentEdgesIn(w)**: Return an iterable collection of the edges directed into w.
  - **incidentEdgesOut(w)**: Return an iterable collection of the edges directed out of w.
  - **inDegree(v)**: Return in-degree of v.
  - **outDegree(v)**: Return the out-degree of v.
- Changes to the undirected graph implementation:
  - **endVertices(e)**: *must* return array A w/ A[0] = src and A[1] = dest (to make direction clear…)
  - SA9: implement DirectedAdjListMap (extends AdjacencyListGraphMap).

Decoration

- **SA9 presents one problem:** the undirected graph provides an **insertEdge** method that doesn't distinguish edge direction!
  - Q: How do we mark an edge as "directed"?
    - Go in and modify the representation of an Edge…?
    - …no!
  - This is a rather common problem with a well known solution: **decoration**.
- **Basic Idea:**
  - Given a fixed representation (e.g., Edge, Vertex), "decorate" it with additional information
    - ex. mark Edge as DIRECTED
    - ex. mark Vertex as VISITED
- **Decoration Implementation Notes:**
  - Add a *small map* to MyPosition — have it implement Map + Position interfaces.
  - The book uses "DecorablePosition"
  - MyPosition extends HashTableMap & implements DecorablePosition
    - HashTableMap is the authors' implementation of open addressing w/ linear probing.
  - Default table is size 3 (small!!!) — b/c not many decorations are expected.

Decoration

- Q: How do we use the map to decorate a position?
  - Create an **Object** to represent the key/value.
  - ex. Key —> Object STATUS;
    Value —> Object VISITED; Object UNVISITED.
  - ex. Key —> Object EDGE_TYPE;
    Value —> Object DIRECTED;
  - etc.
- Mark a position p as visited
  - p.put(STATUS, VISITED)
- Mark a position p as unvisited
  - p.put(STATUS, UNVISITED)
- Test if position p is visited/unvisited
  - p.get(STATUS) == VISITED
- You'll use this in SA9… :)

NOTE: *don't really need both VISITED and UNVISITED… if something is VISITED, by definition it is UNVISITED…*

One hundred little bugs in the code
One hundred little bugs.
Fix a bug, compile again,
One hundred little bugs in the code.