CSCI 2270

Data Structures & Algorithms

Gabe Johnson

Lecture 41 May 1, 2013

Review for Final Exam
Part II: The Revenge

Lecture Goals

1. Debugging
2. Finite State Machines
3. Complexity (Time/Space)
4. Finite State Machines
5. Graphs: DAG, BFS, Dijkstra
6. Ability to learn and adapt
7. Special lecture next time

Upcoming Homework Assignment

HW #12 **Due: Friday, May 3**

Game AI

Write an AI for last week's game. This is entirely extra credit. Follow the instructions on GitHub. I will be *inundated* with email this week, so if you don't get the subject line there is a high probability I will not see yours.

Debugging

I'll give you some *nearly* good code and ask you to debug it. Your solution might be:

— change a variable
— change an inequality or similar operator
— rearrange code to execute in different order
— add correct flow-control (if/while/for)
— use or avoid recursion as needed
— describe in English how to re-write the code

Debugging

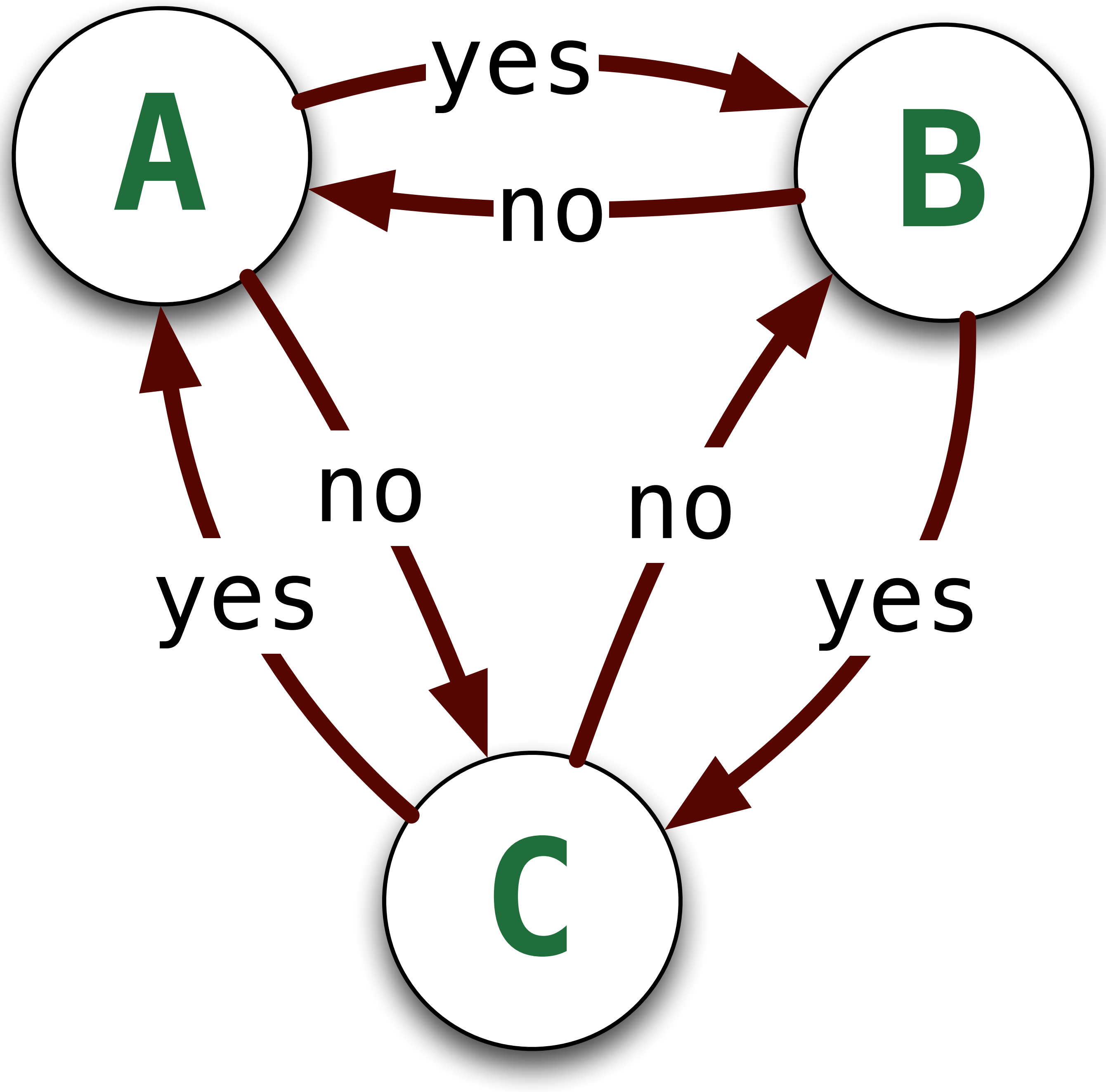
I might ask:

“*What’s wrong with this code?*” and not provide any further hint.

“*This code does X but I want it to do Y. Why*?” Here your solution is to explain, not to fix it.

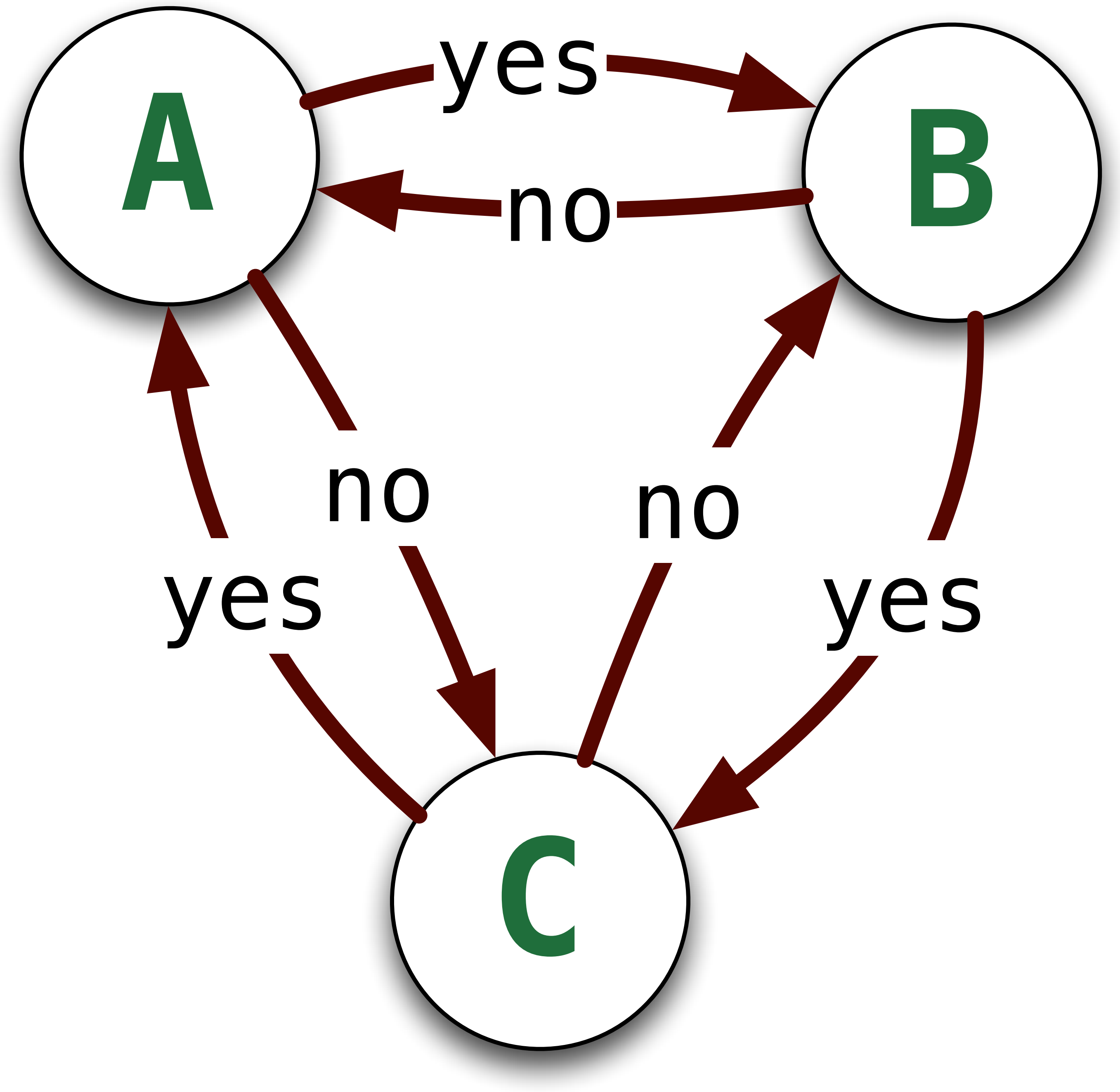
“*This code segfaults but I don’t know where*.” Your answer might be all the possible spots where a segfault can occur.

Finite State Machines



FSMs are directed graphs. Nodes represent *states*, edges represent *transitions*.

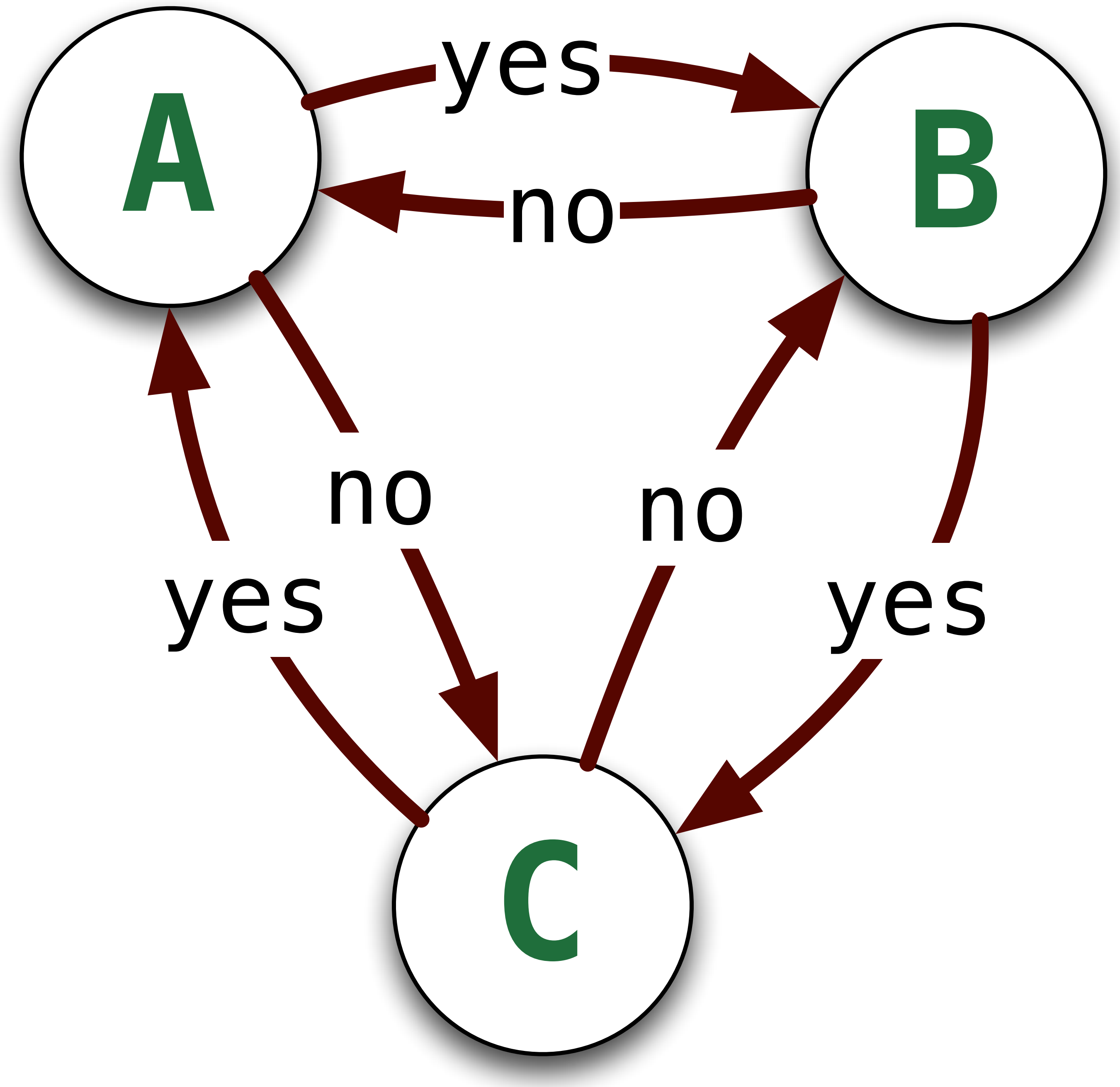
Finite State Machines



A FSM can be in a single state at a time.

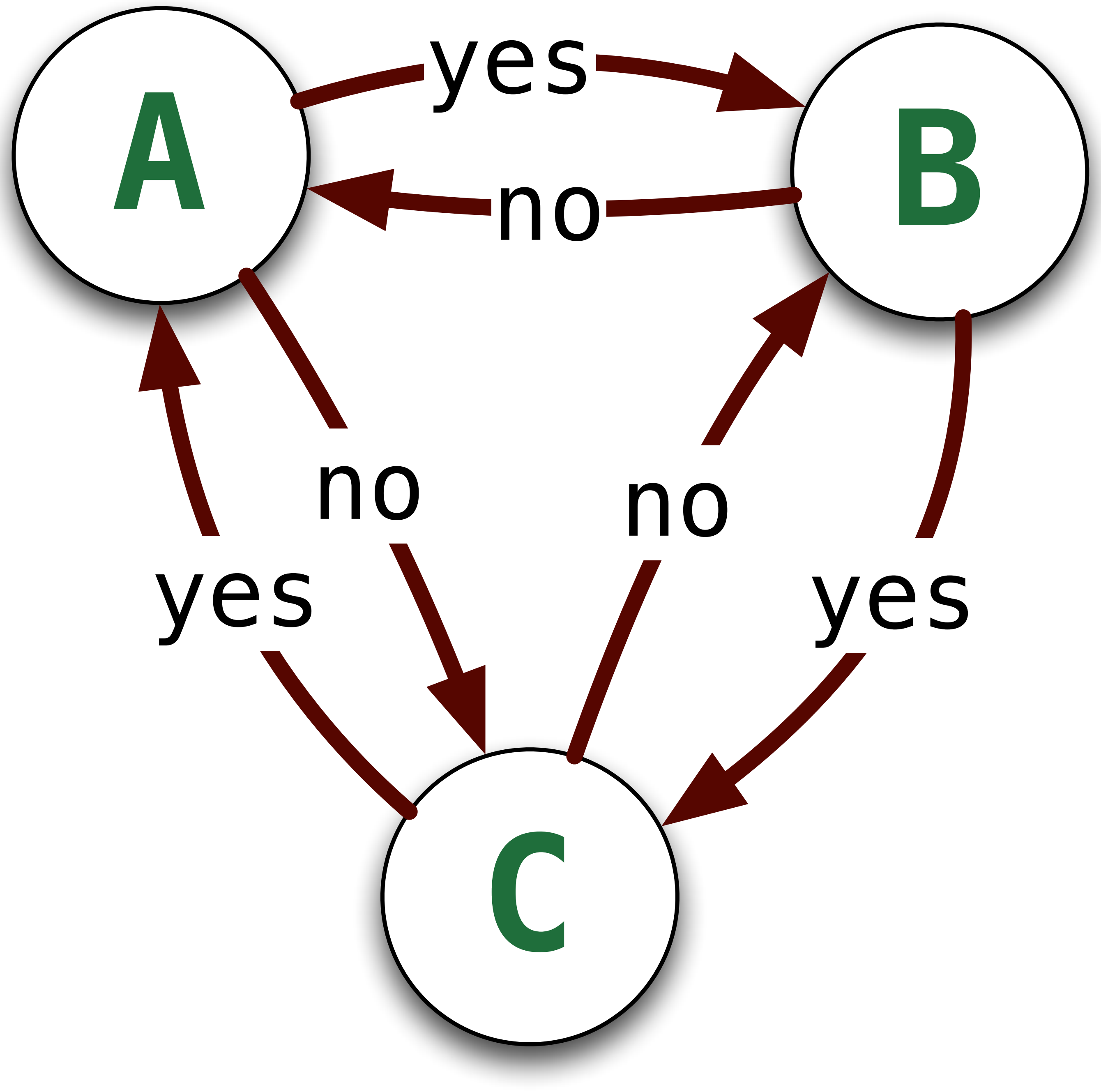
FSMs receive signals that might initiate a transition from the current state to the next.

Finite State Machines



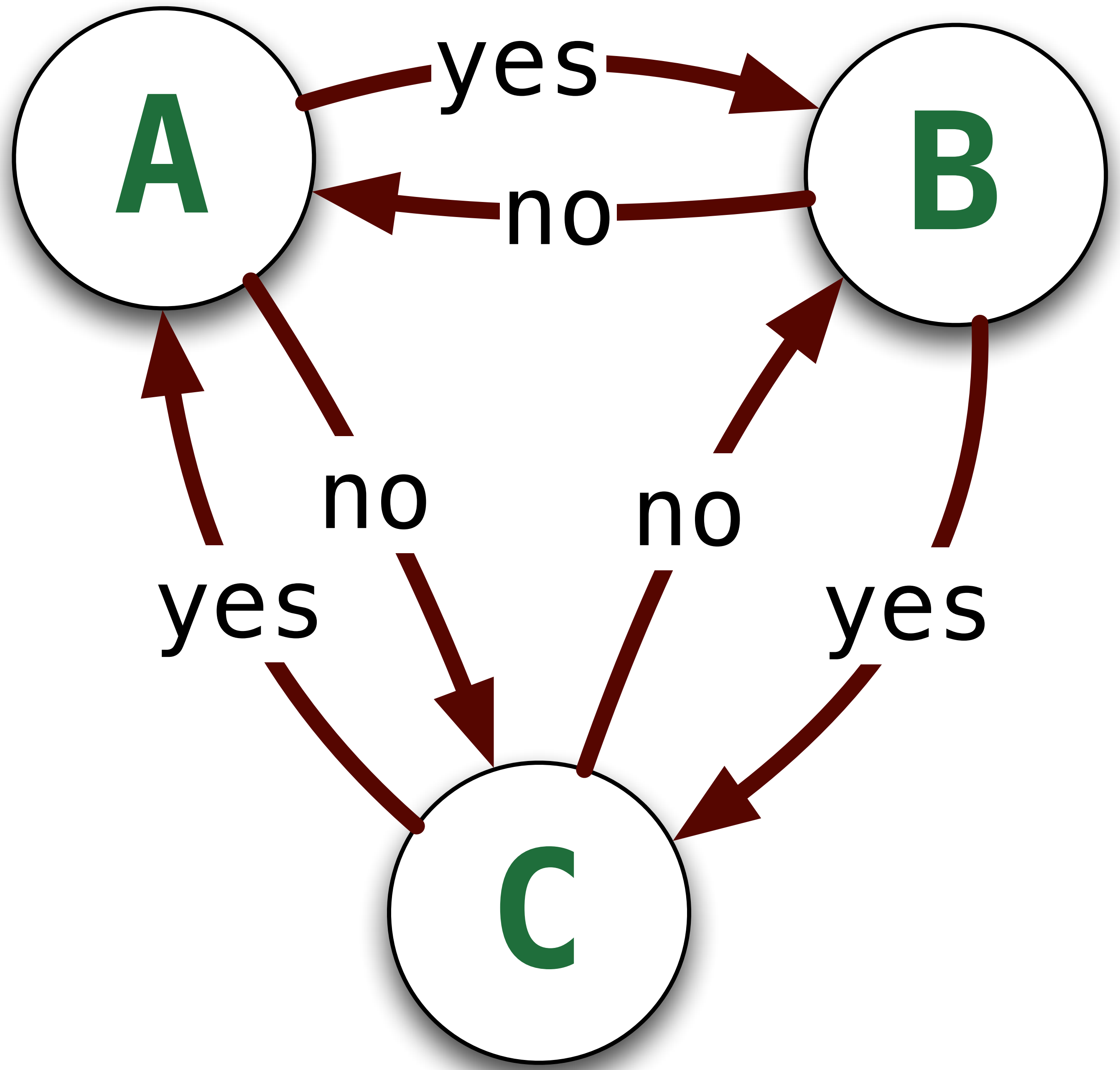
I say *might* because a transition only occurs if we get a signal that is associated with one of the outgoing transitions.

Finite State Machines



All these states accept *yes* and *no* signals, so as long as that's what we receive, the FSM will always change states from one to another.

Finite State Machines



But if the FSM receives a signal that is not associated with one of the current state's outgoing transitions, nothing happens. (Or it is an error—depends on the use case.)

Complexity

We use **big oh** notation to describe resource usage according to some input size. We write it like these:

O(1) (constant)	**O(n^2)** (quadratic)
O(n) (linear)	**O(2^n)** (exponential)
O(log n) (logarithmic)	**O(n!)** (factorial)

Notice that *we do not include constant factors* in anything other than O(1). This is because big oh denotes how resource usage grows *in proportion to* the input size.

Complexity (Time)

This refers to how many operations will be performed to handle some number of inputs.

```
k = 0
for (i=0 to n):
    for (j=0 to n):
        k = k + 1
print k
```

Complexity (Time)

This refers to how many operations will be performed to handle some number of inputs.

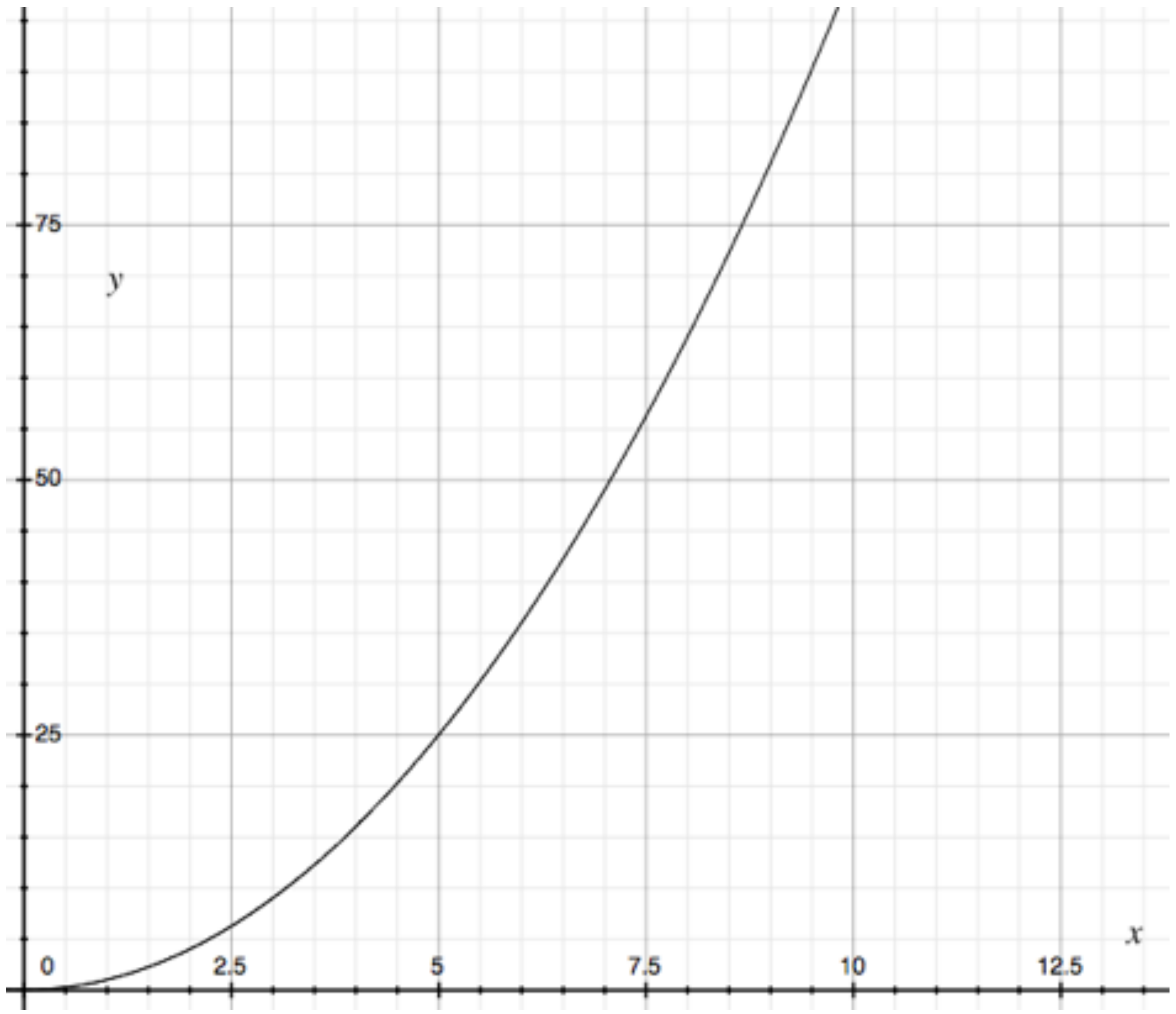
```
k = 0
for i in [0, n):
  for j in [0, n):
    k = k + 1
print k
```

n	k
1	1
2	4
3	9
4	16
5	25
6	36

Complexity (Time)

This refers to how many operations will be performed to handle some number of inputs.

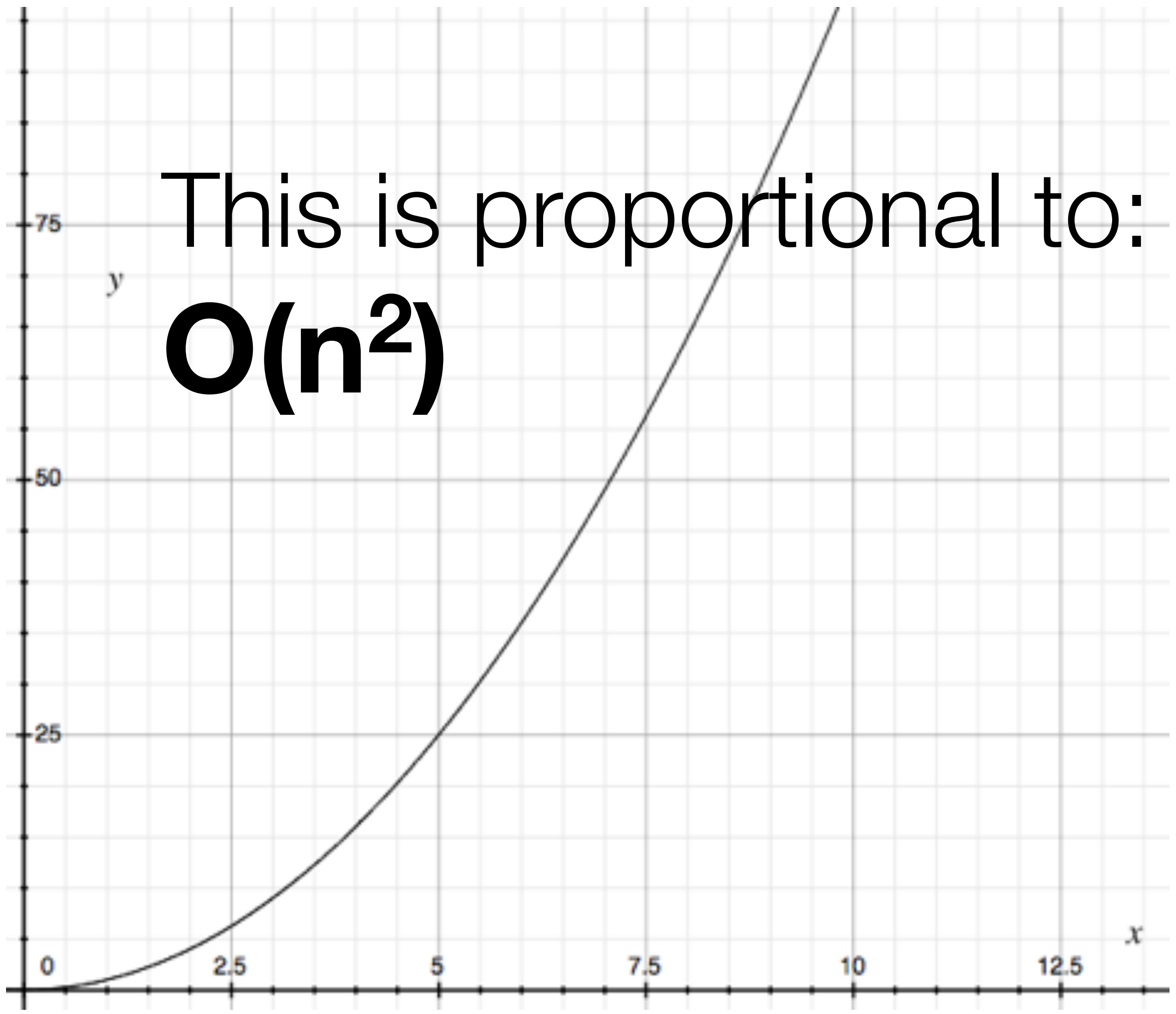
```
k = 0
for i in [0, n):
  for j in [0, n):
    k = k + 1
print k
```



Complexity (Time)

This refers to how many operations will be performed to handle some number of inputs.

```
k = 0
for i in [0, n):
  for j in [0, n):
    k = k + 1
print k
```



This is proportional to:

$O(n^2)$

Complexity (Time)

```
k = 0
for i in [0, n + 100):
  for j in [0, n*n):
    for x in [1, 5]:
      k = k + 1
print k
```

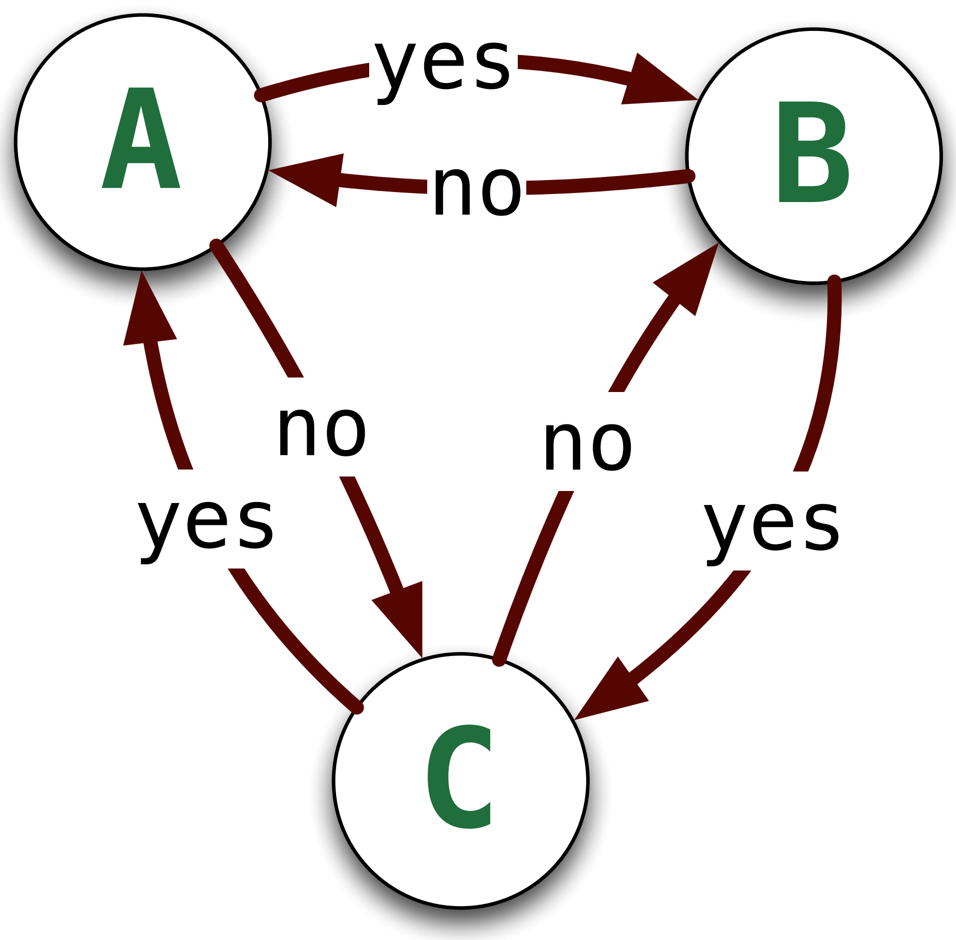
How about this? Remember, complexity notation is all about the *proportional* resource usage with respect to the input size *n*. Constant factors are not involved!

Complexity (Space)

An alternate kind of complexity measure is how much space (e.g. main memory or disk usage) an algorithm requires.

For example, if an algorithm was required to make a square matrix of size n, the space complexity is **$O(n^2)$** because it requires n*n memory elements to form the matrix.

Graphs

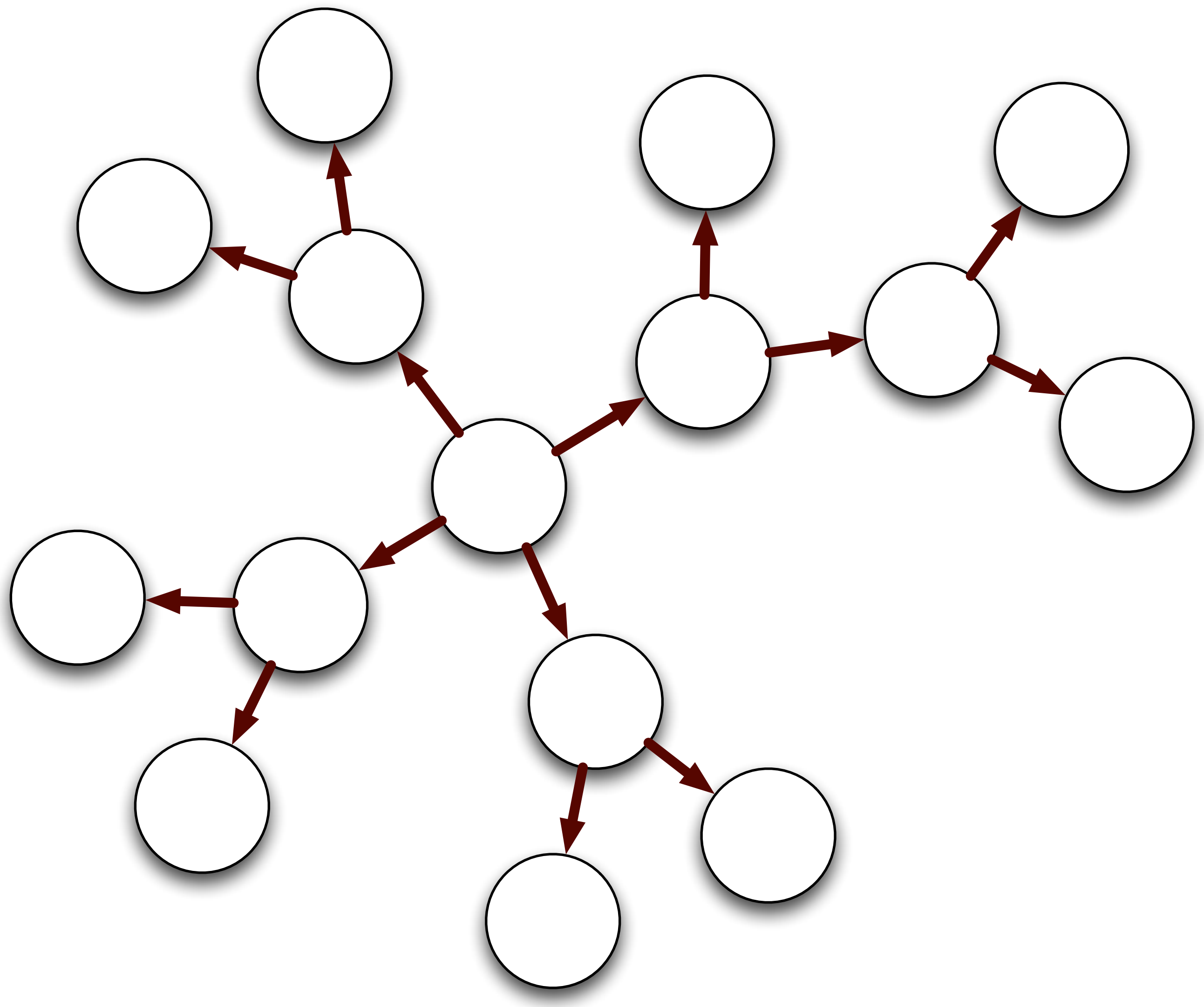


The FSM we saw earlier is a graph. Graphs are composed of nodes (often called *vertices*) and edges.

Edges may be directional (e.g. they have arrowheads) or not.

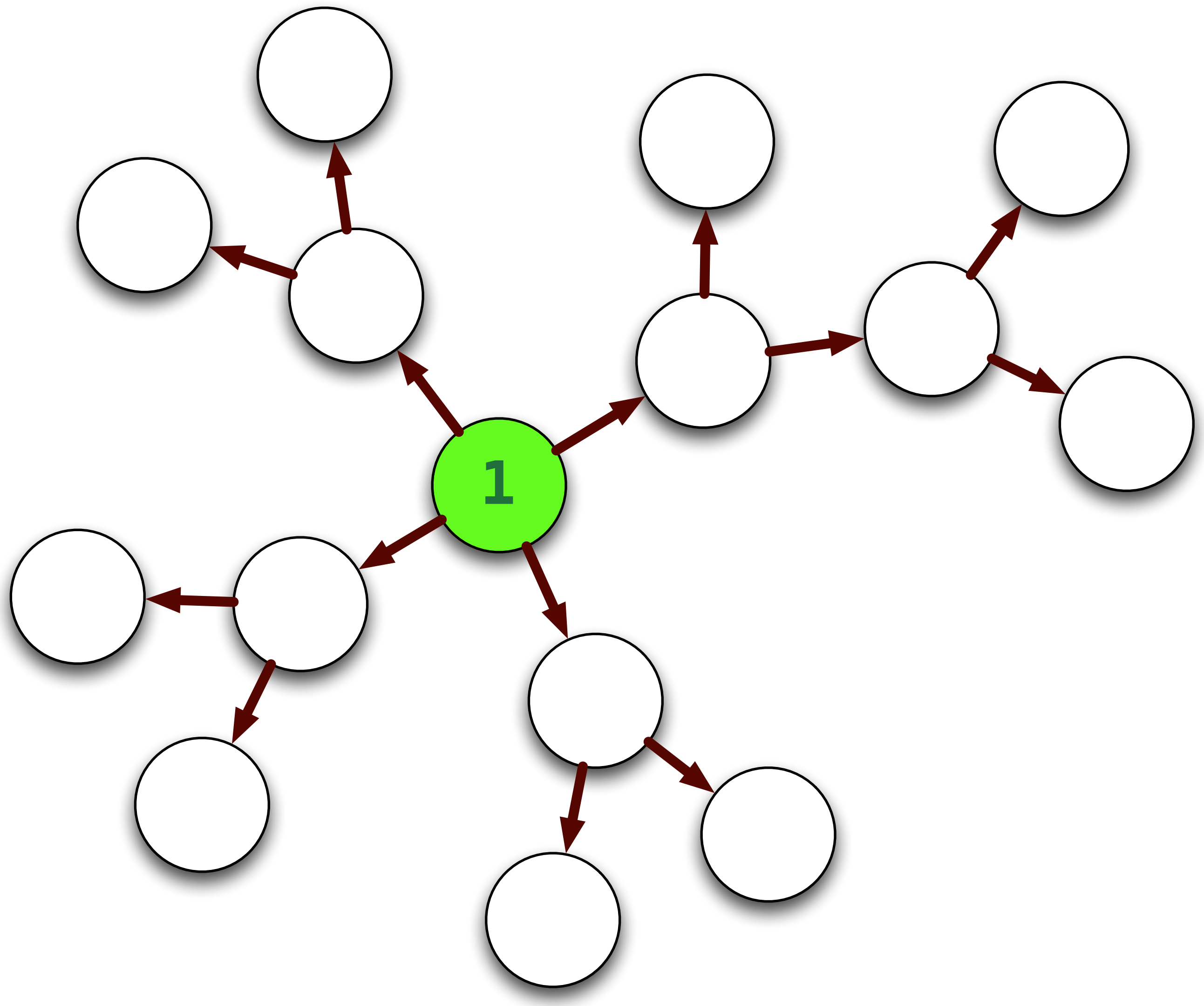
Graphs may be *directed* or *undirected*. Rarely, you might encounter a graph that has a mix of directed and undirected edges.

Graphs: BFS



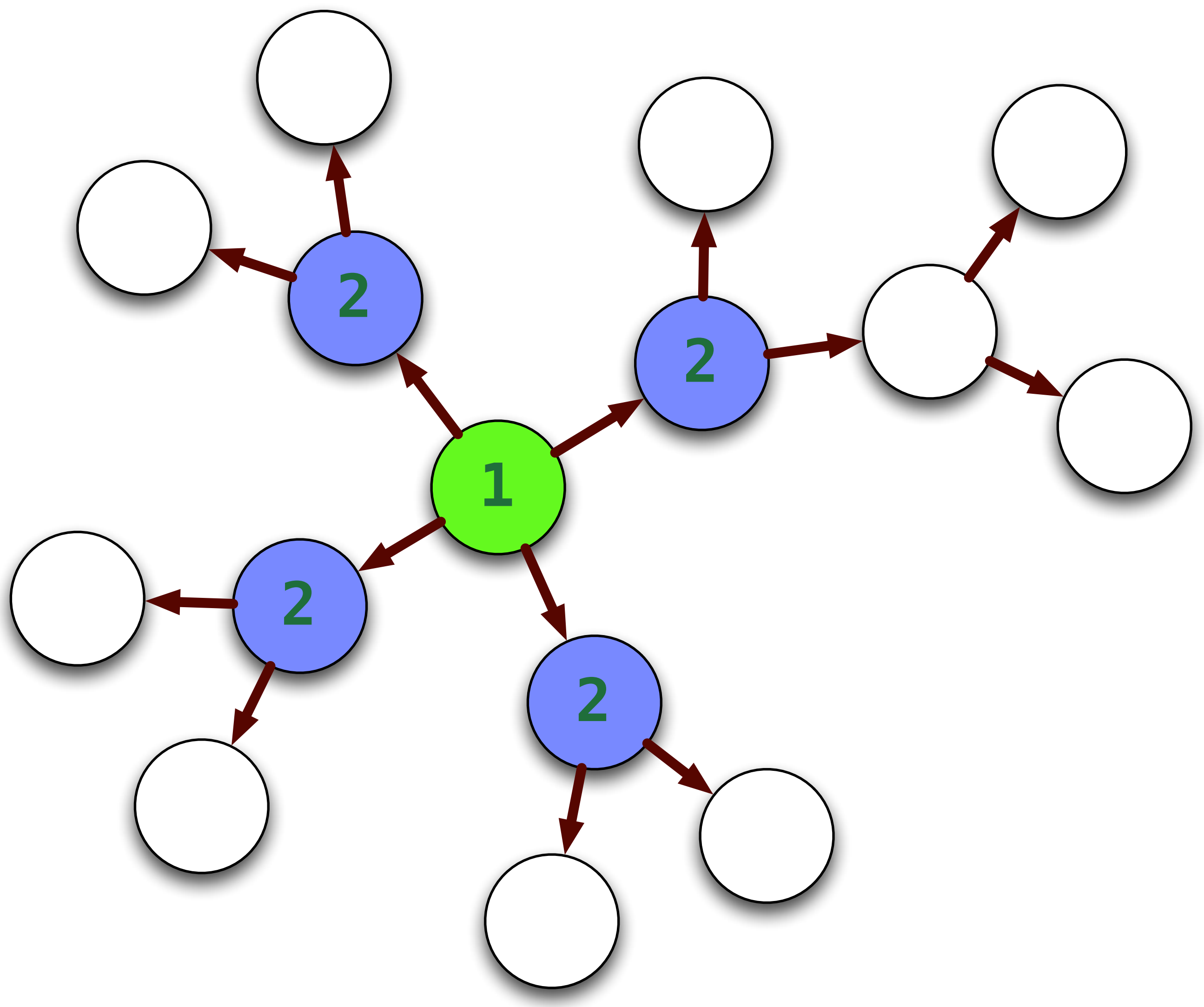
A *breadth-first search* is a broad category of graph algorithm where we explore the graph one 'layer' of depth at a time.

Graphs: BFS



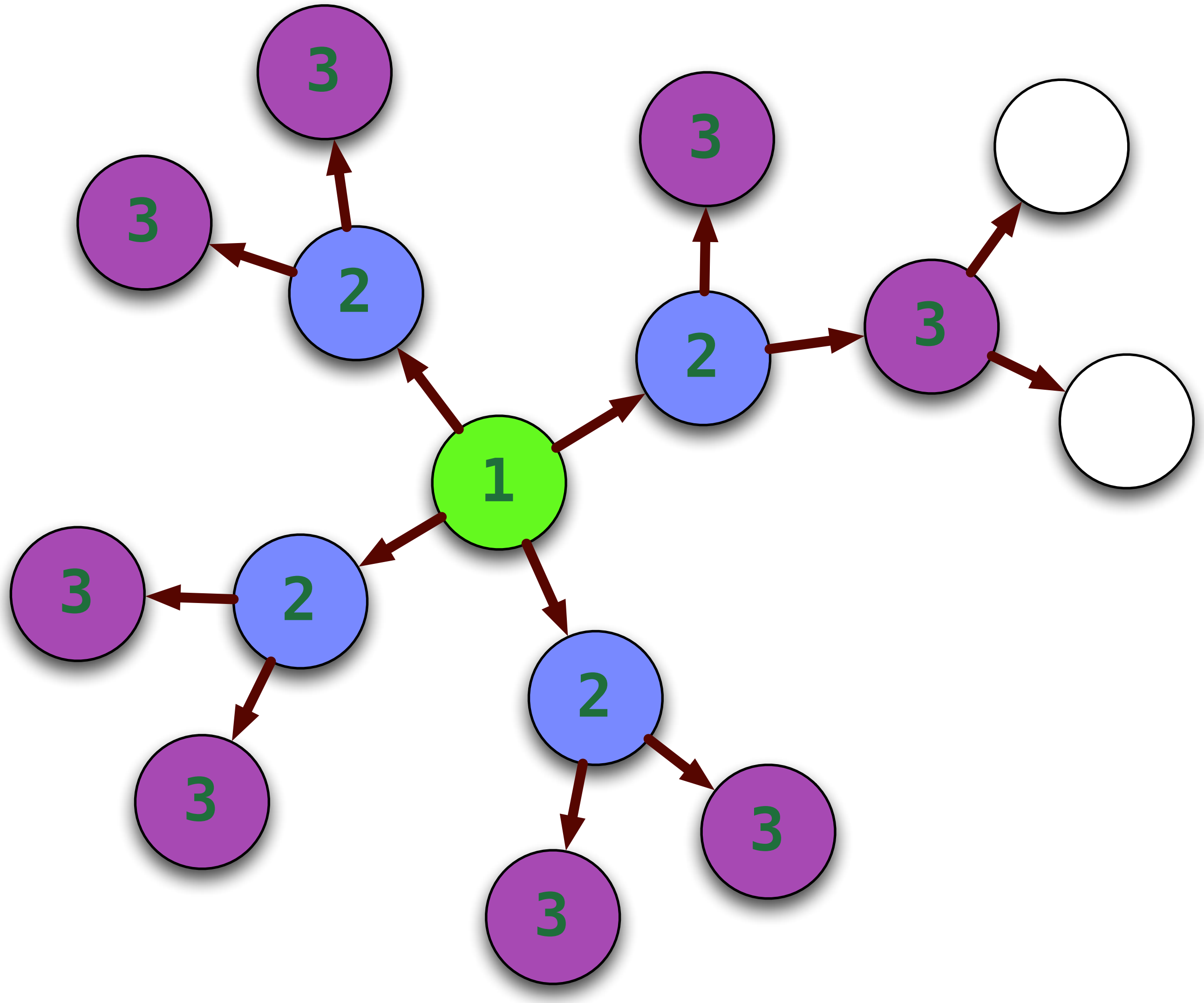
We discover nodes, and add them to a queue. So if we start in the middle here, we'll add all four of the neighboring nodes to a queue, and visit them all in turn.

Graphs: BFS



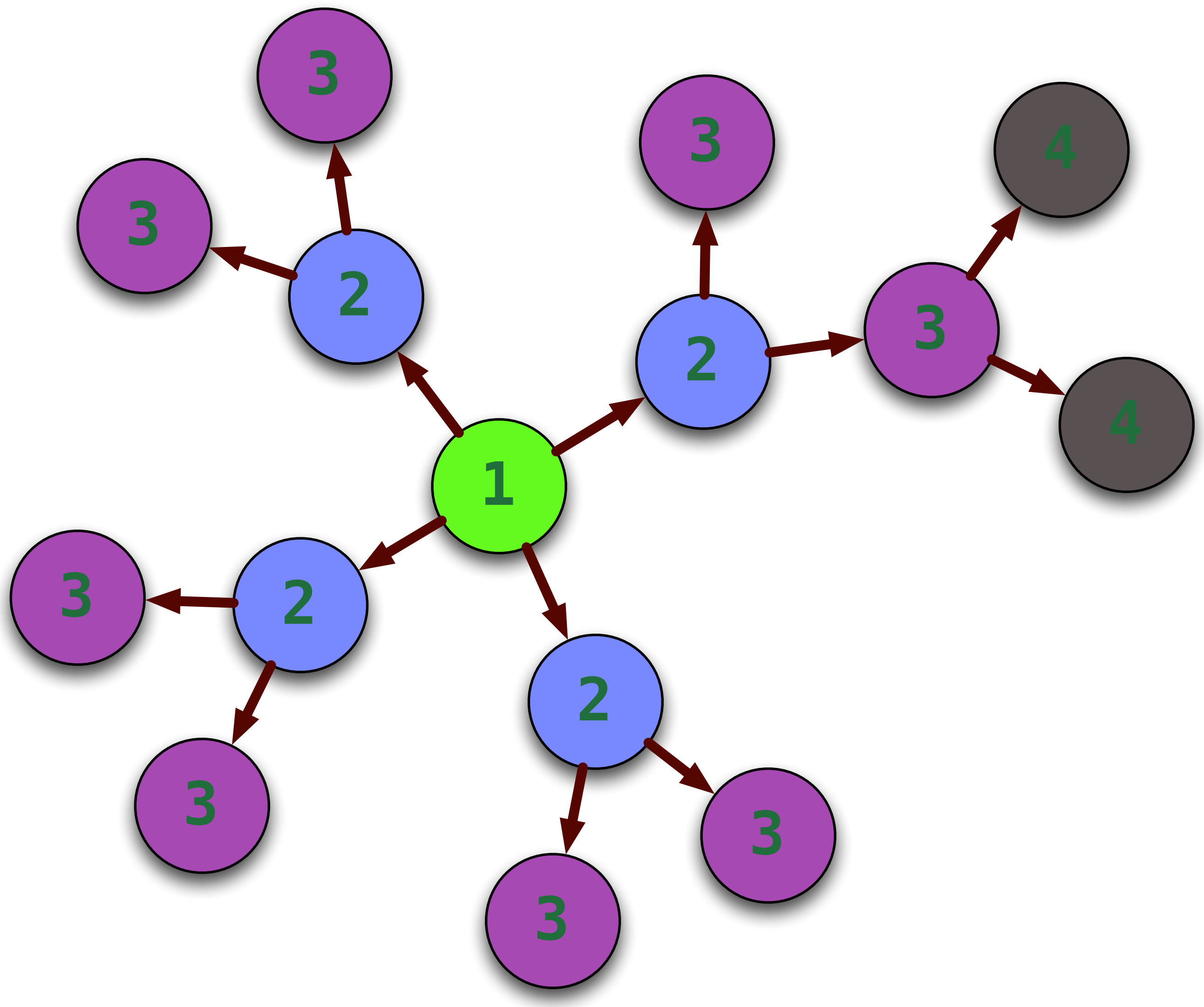
Each time we visit a node, we add its children to the same queue—which means we attend to nodes in the same order we discovered them.

Graphs: BFS

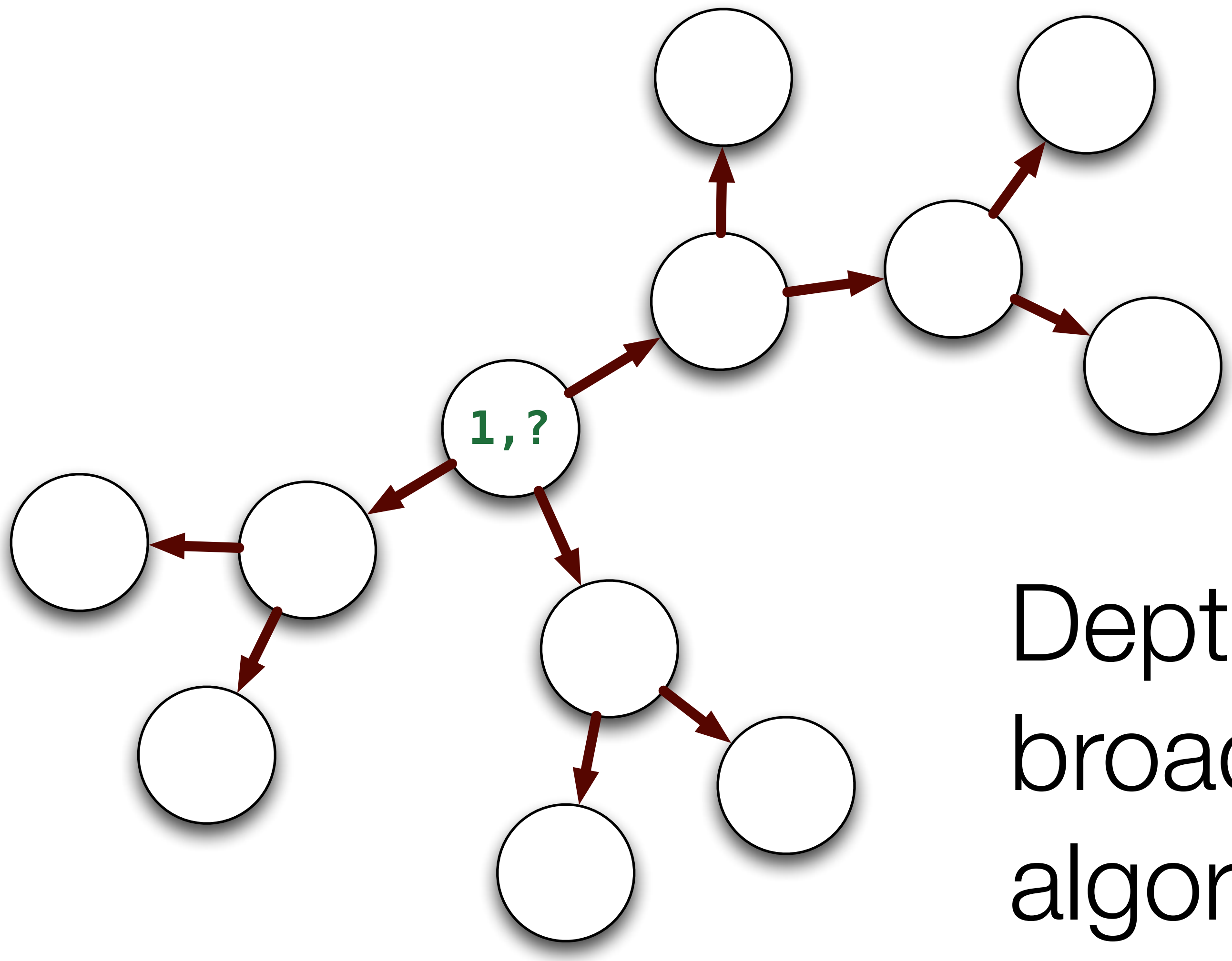


As you probably see, this algorithm proceeds slowly away from the start node. We can measure the minimum number of edges it takes to reach a node easily.

Graphs: BFS

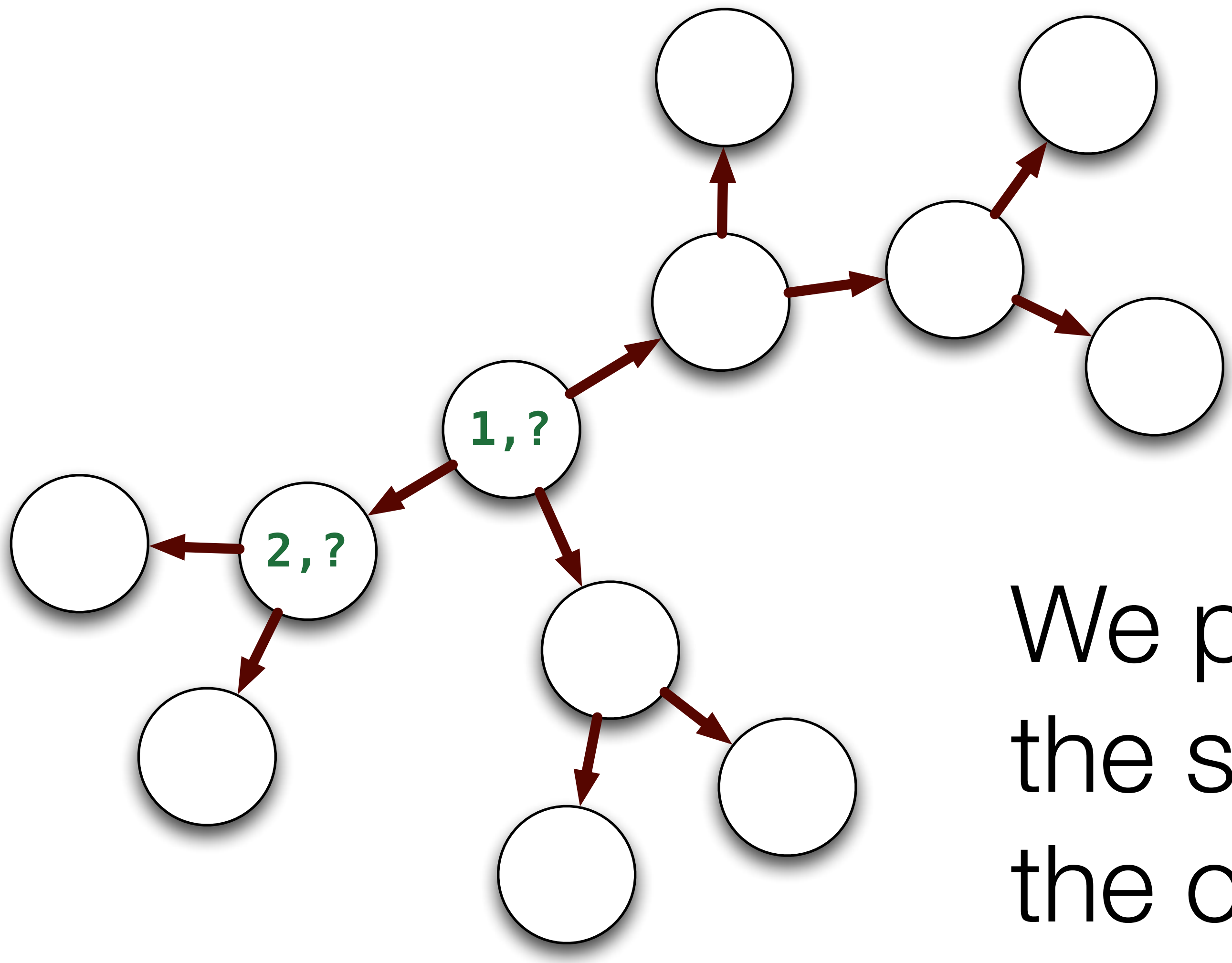


Graphs: DFS



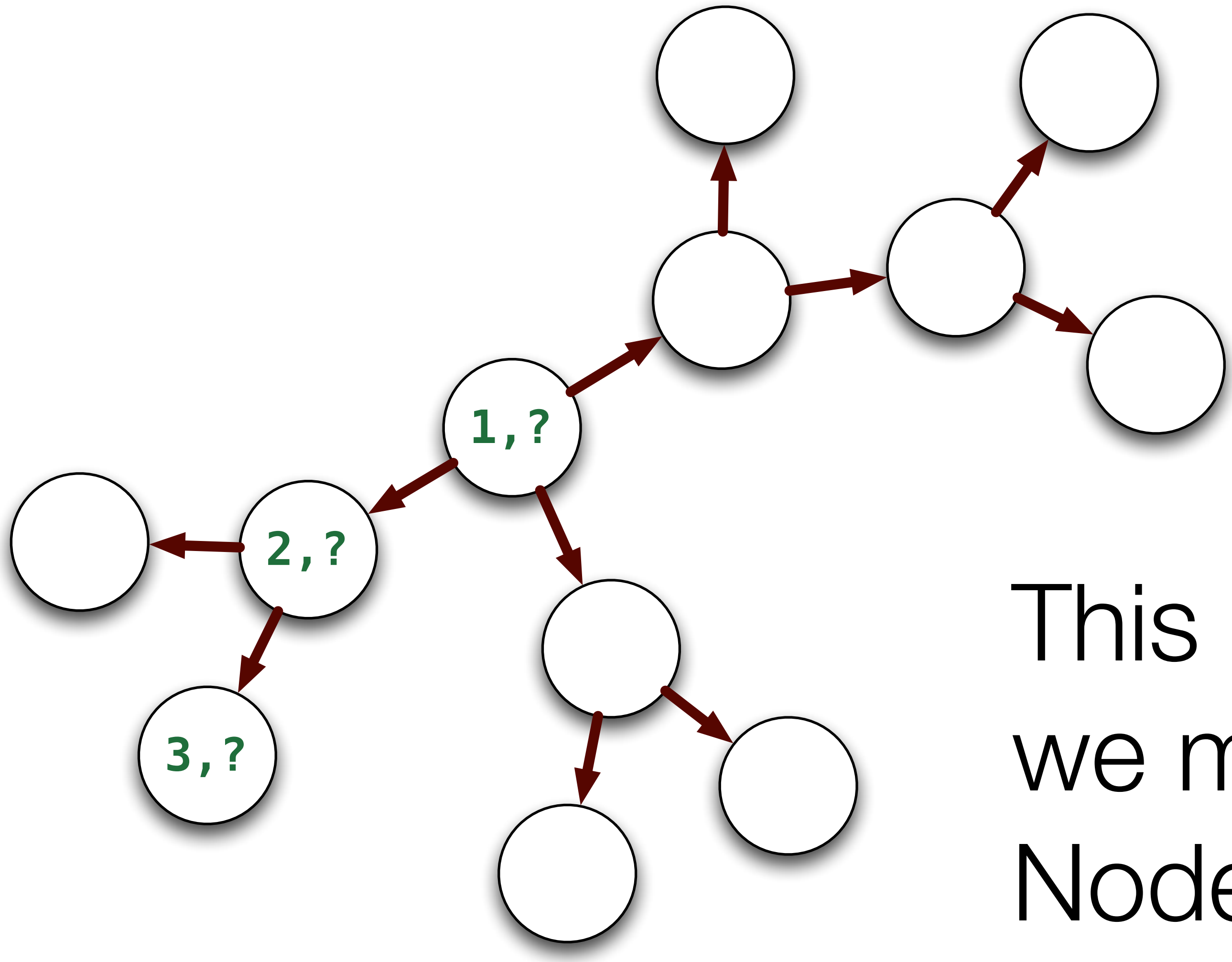
Depth-first search is another broad category of graph algorithms. Here we use a *stack* to explore as deeply as we can.

Graphs: DFS



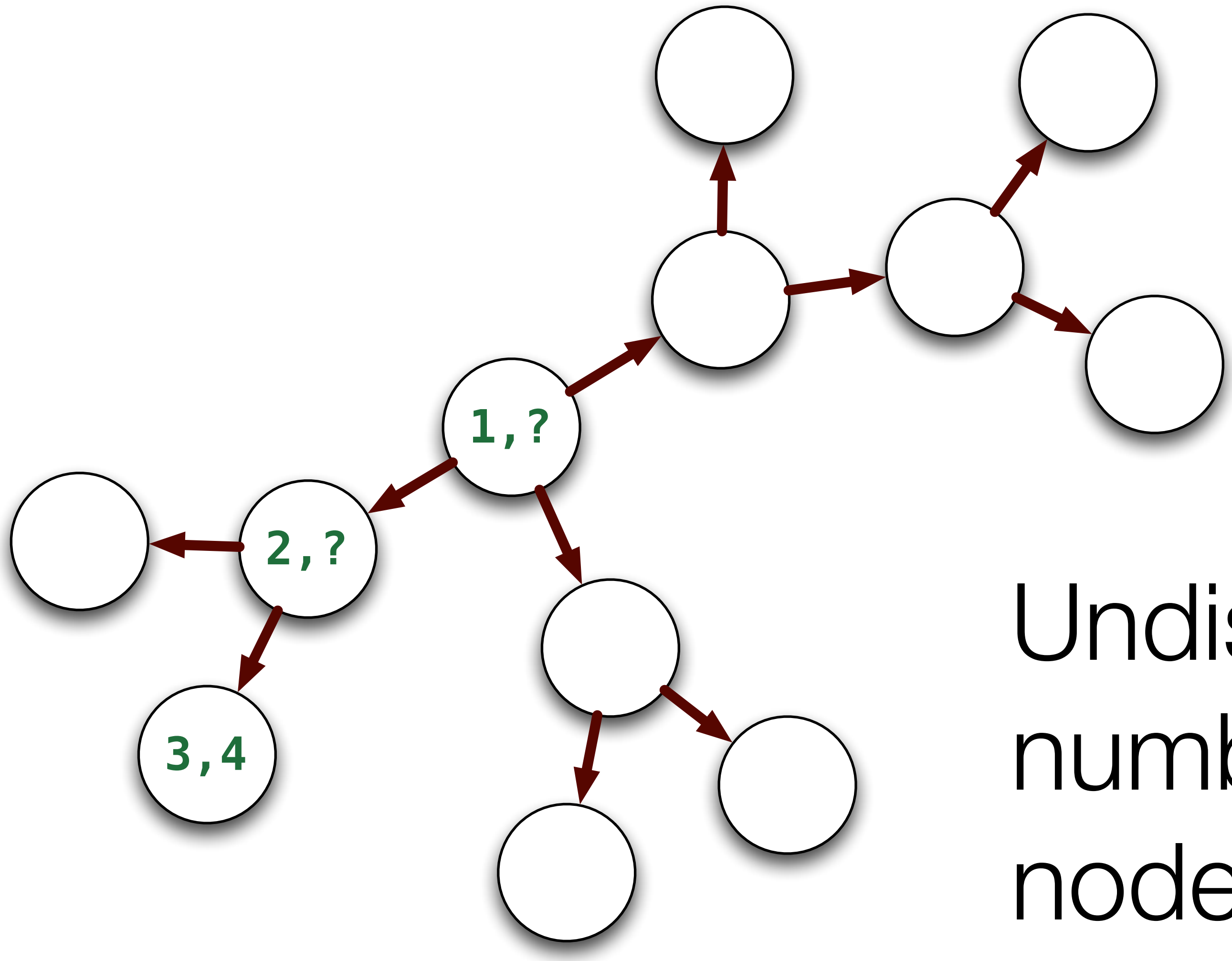
We push a discovered node onto the stack, and remove them in the opposite order they were added. (Remember how stacks work?)

Graphs: DFS



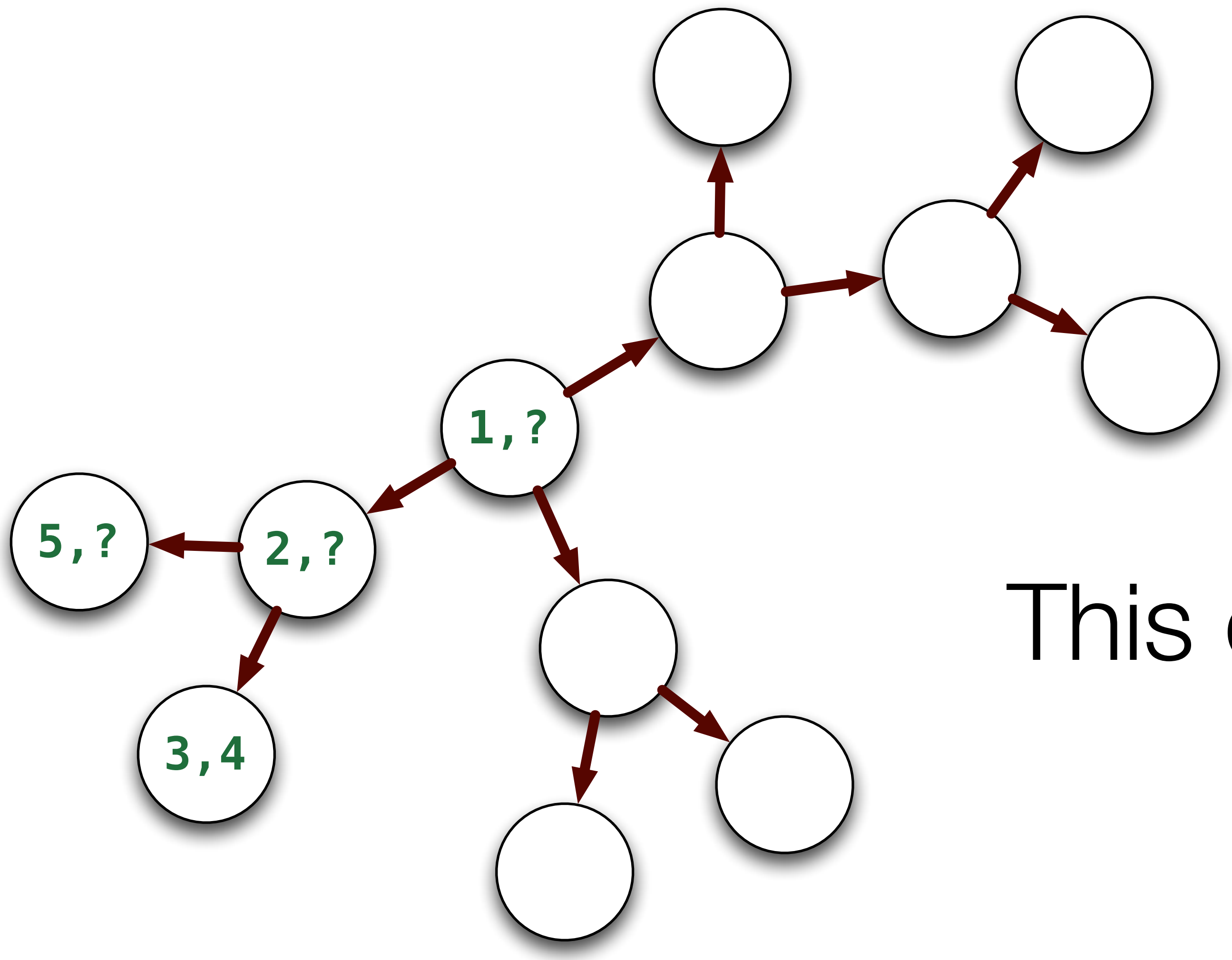
This means we explore the nodes we most recently discovered. Nodes that were pushed onto the stack first are dealt with last.

Graphs: DFS



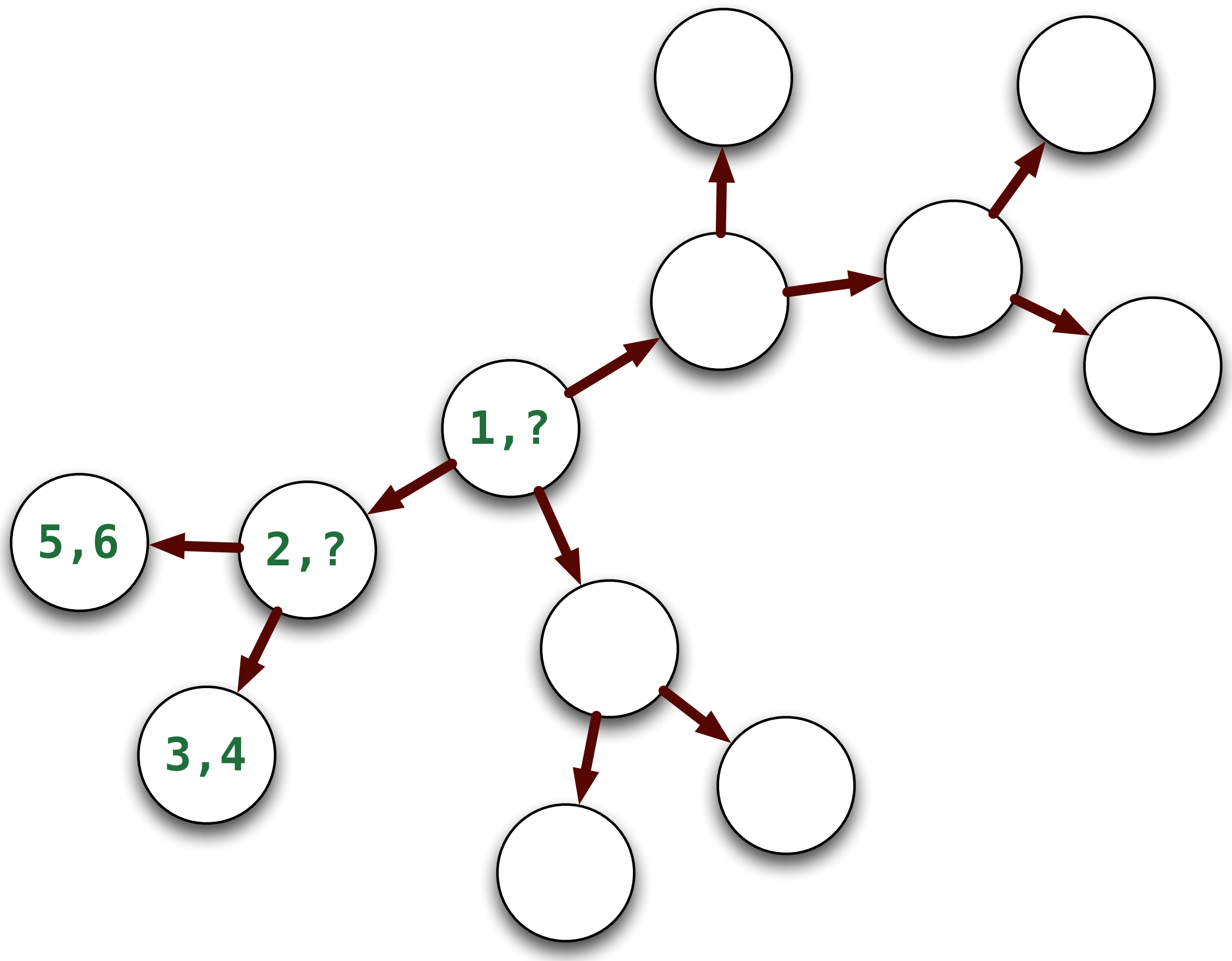
Undiscovered nodes (without numbers) are *white*; discovered nodes (number and question mark) are *gray*; finished nodes (two numbers) are *black*.

Graphs: DFS

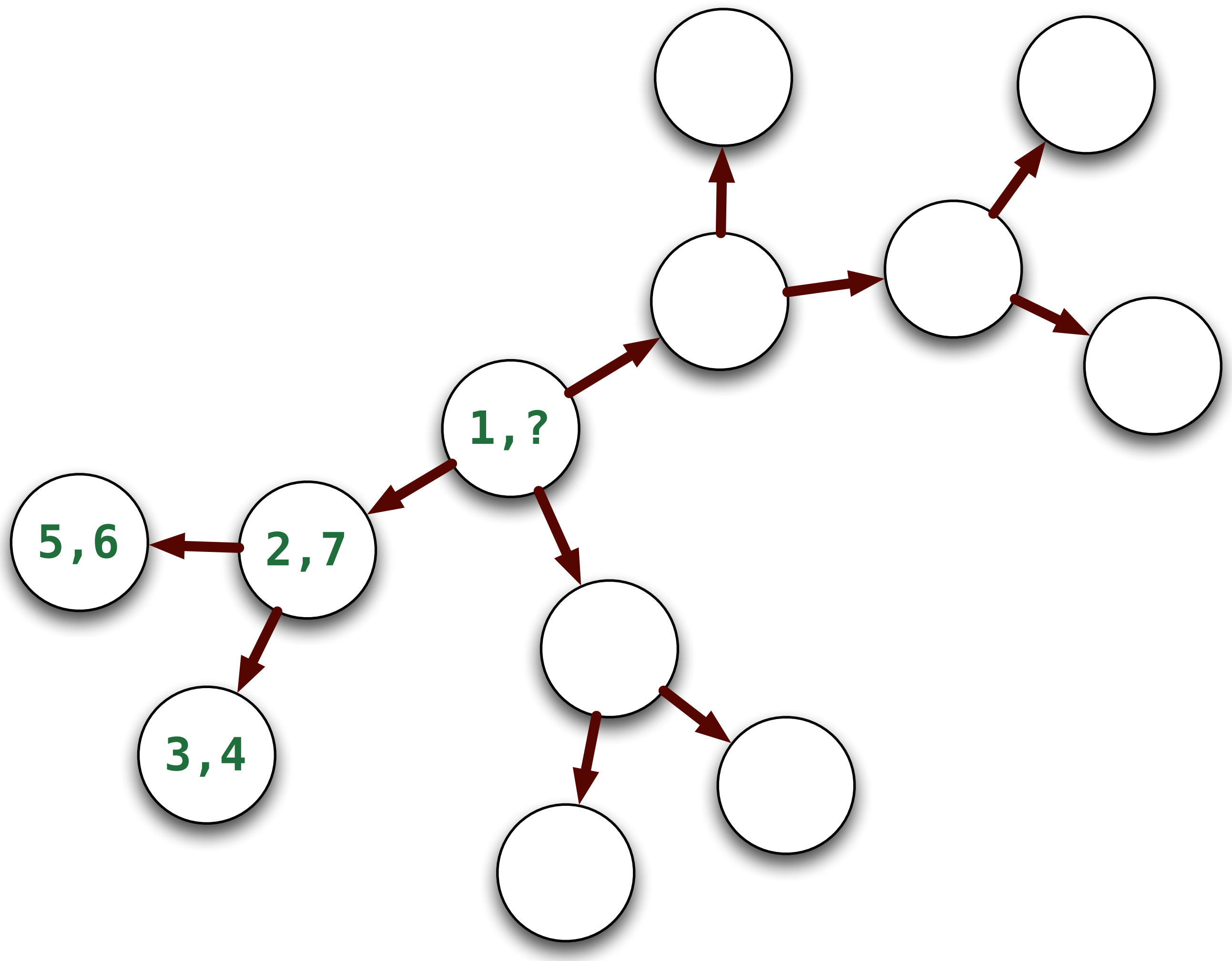


This continues for a while...

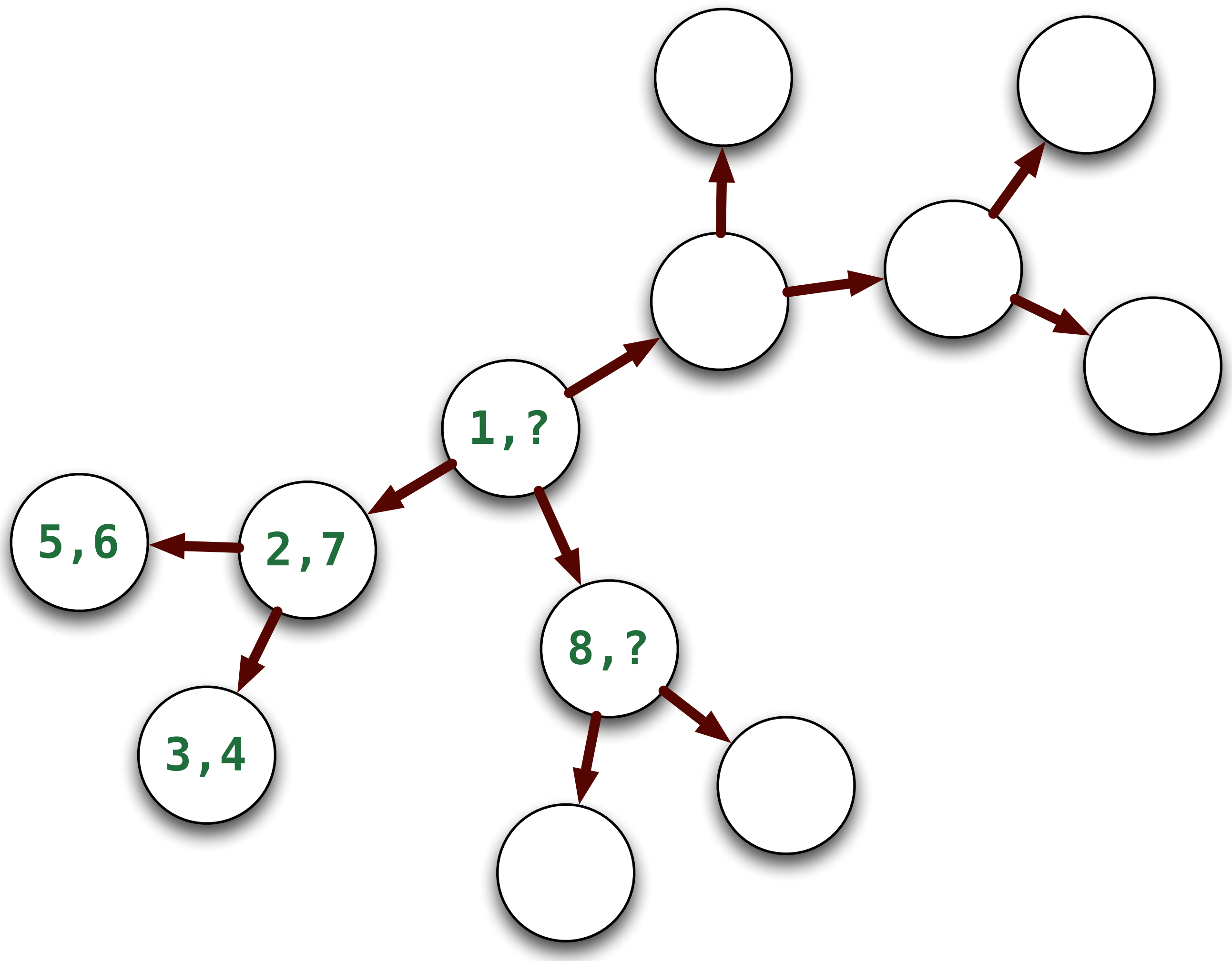
Graphs: DFS



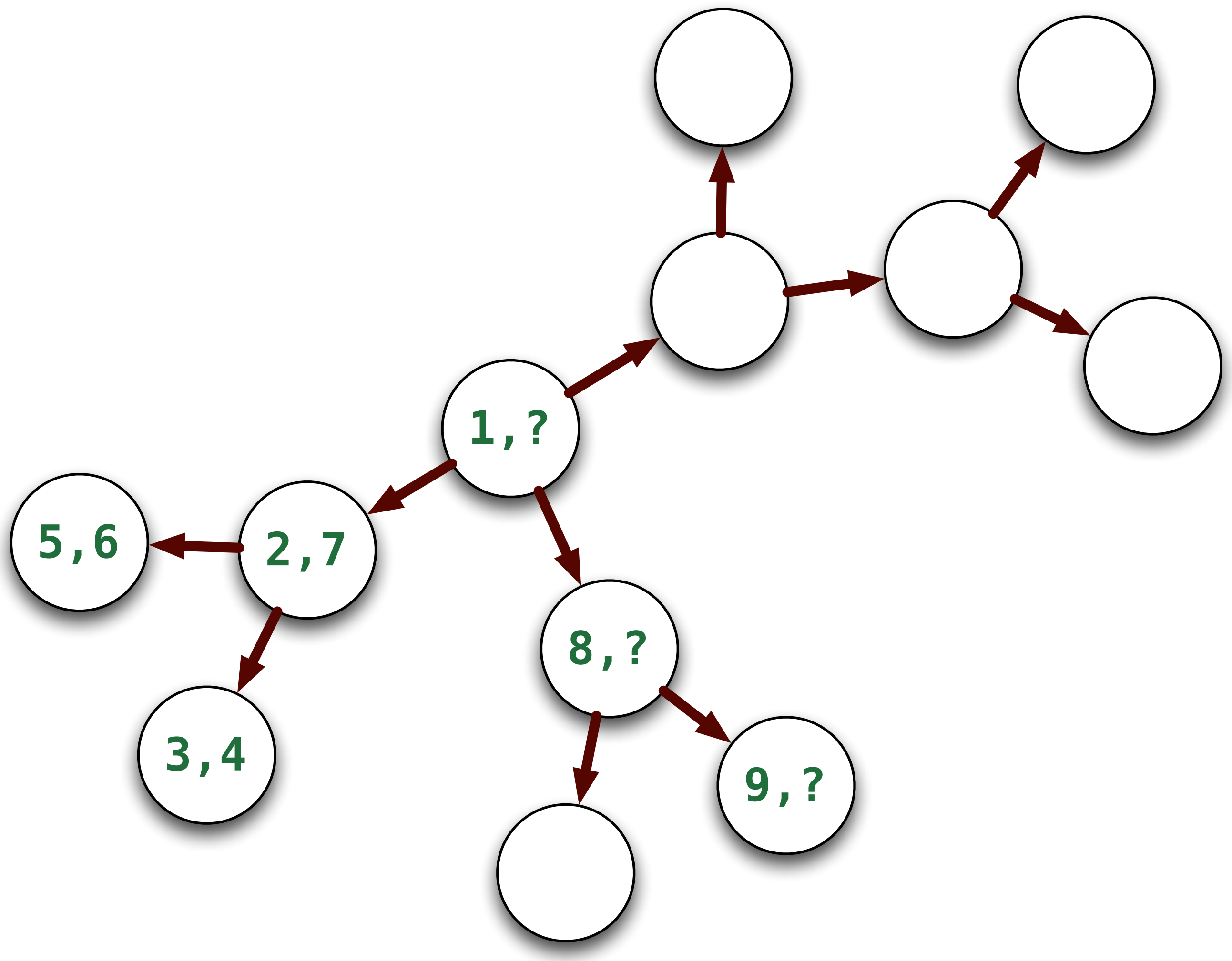
Graphs: DFS



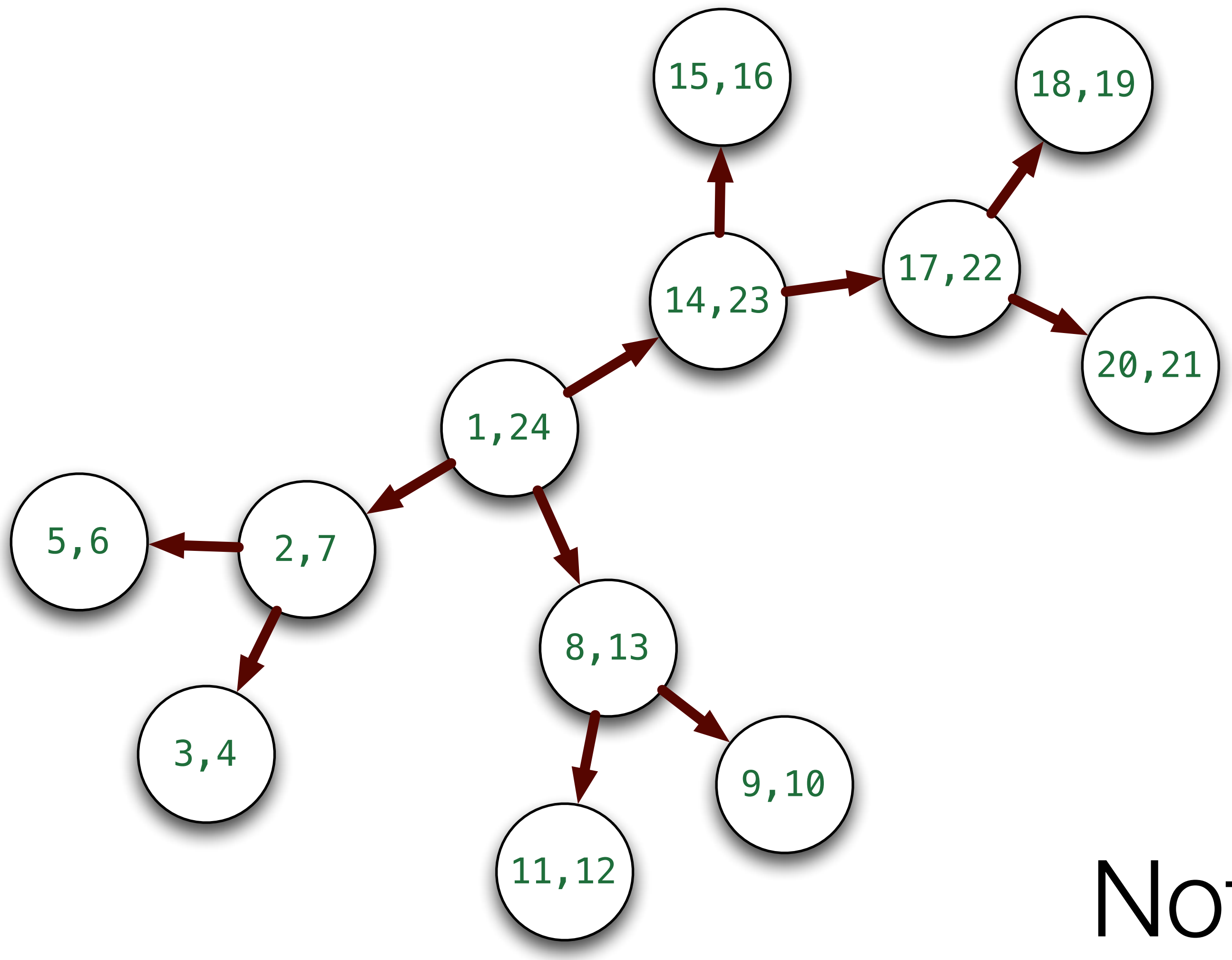
Graphs: DFS



Graphs: DFS



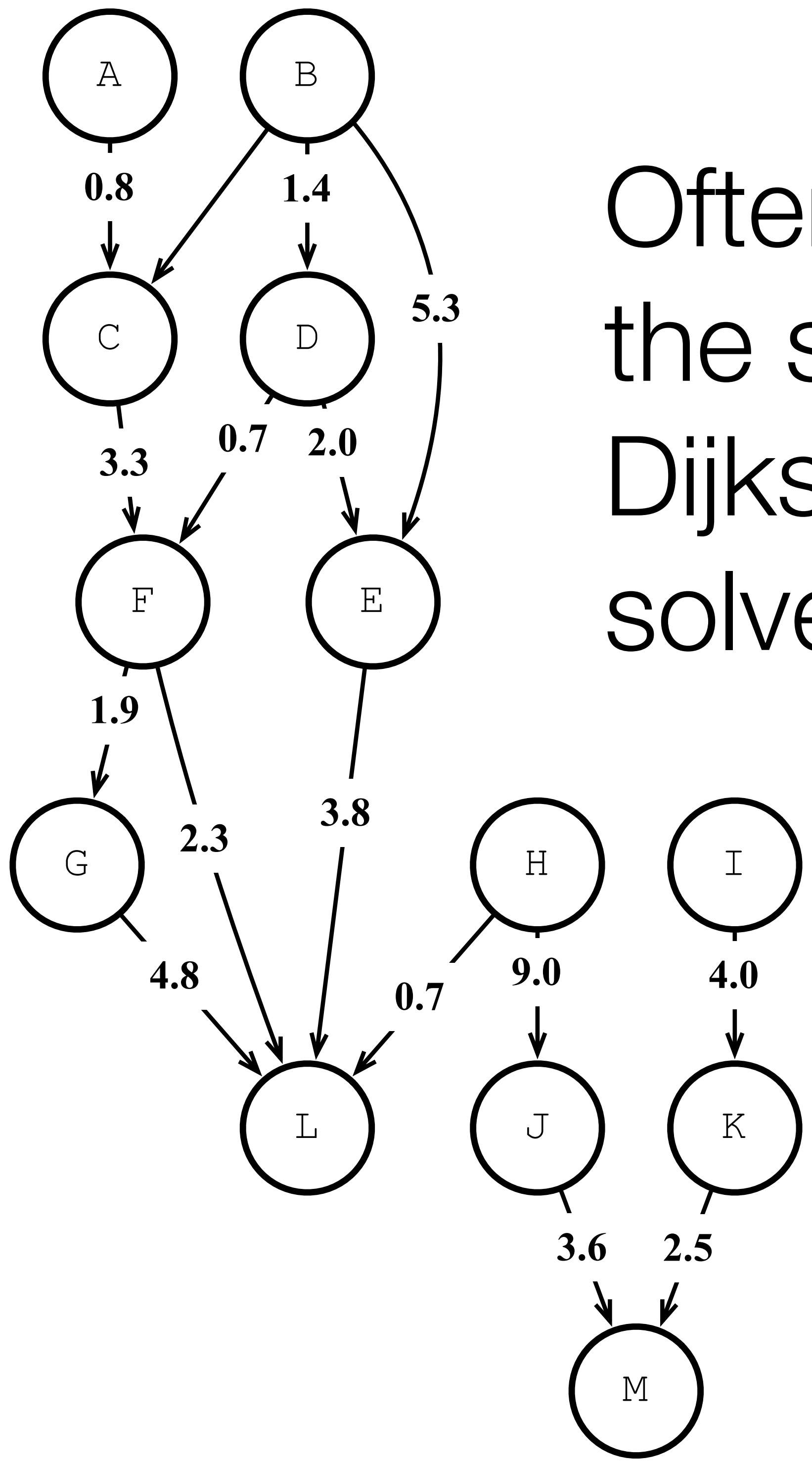
Graphs: DFS



At the end of the DFS each node is annotated with the discovery and finish time.

Notice the first node started first and finished last.

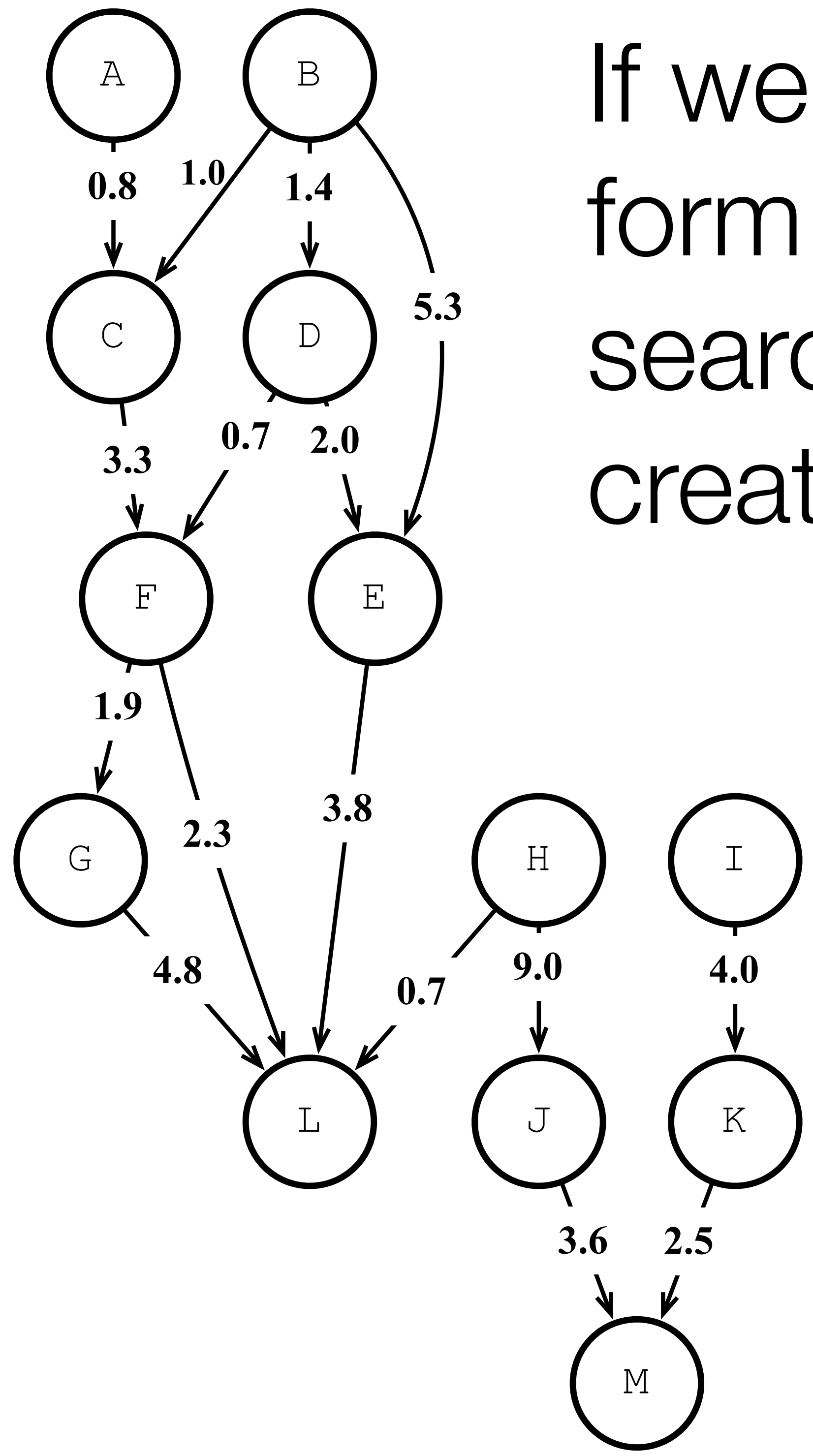
Graphs: Dijkstra’s Algo



Often we’re interested in knowing what the shortest path between locations. Dijkstra’s algorithm is one of many that solves this problem.

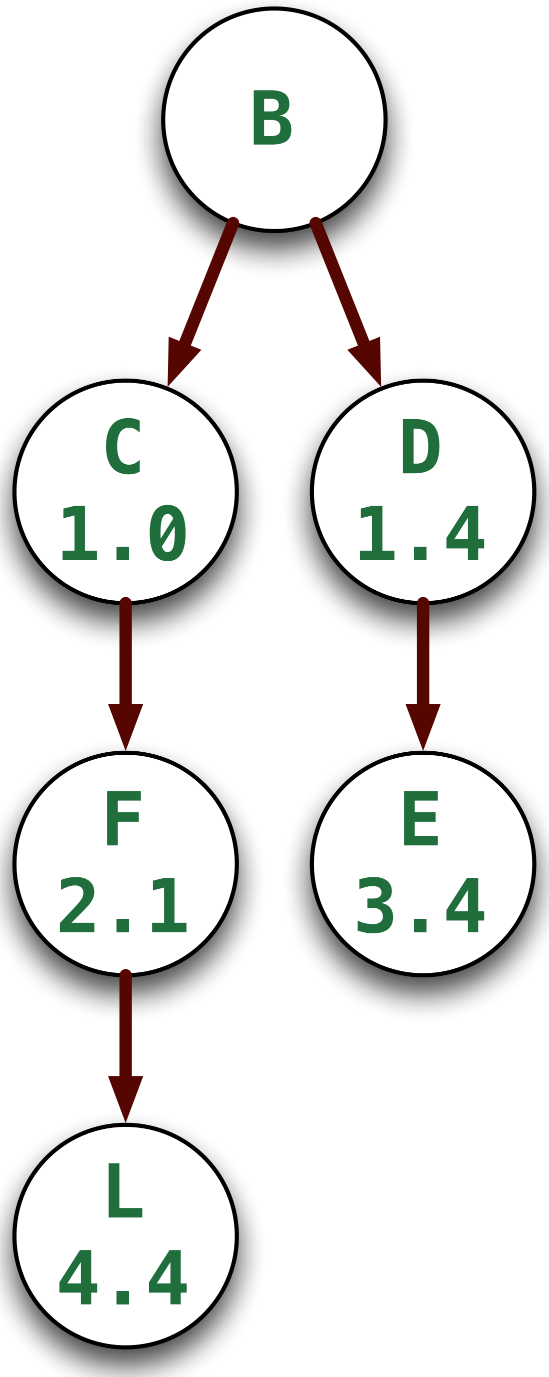
This builds a *minimum spanning tree* that describes the shortest path between some start node and all other nodes.

Graphs: Dijkstra's Algo

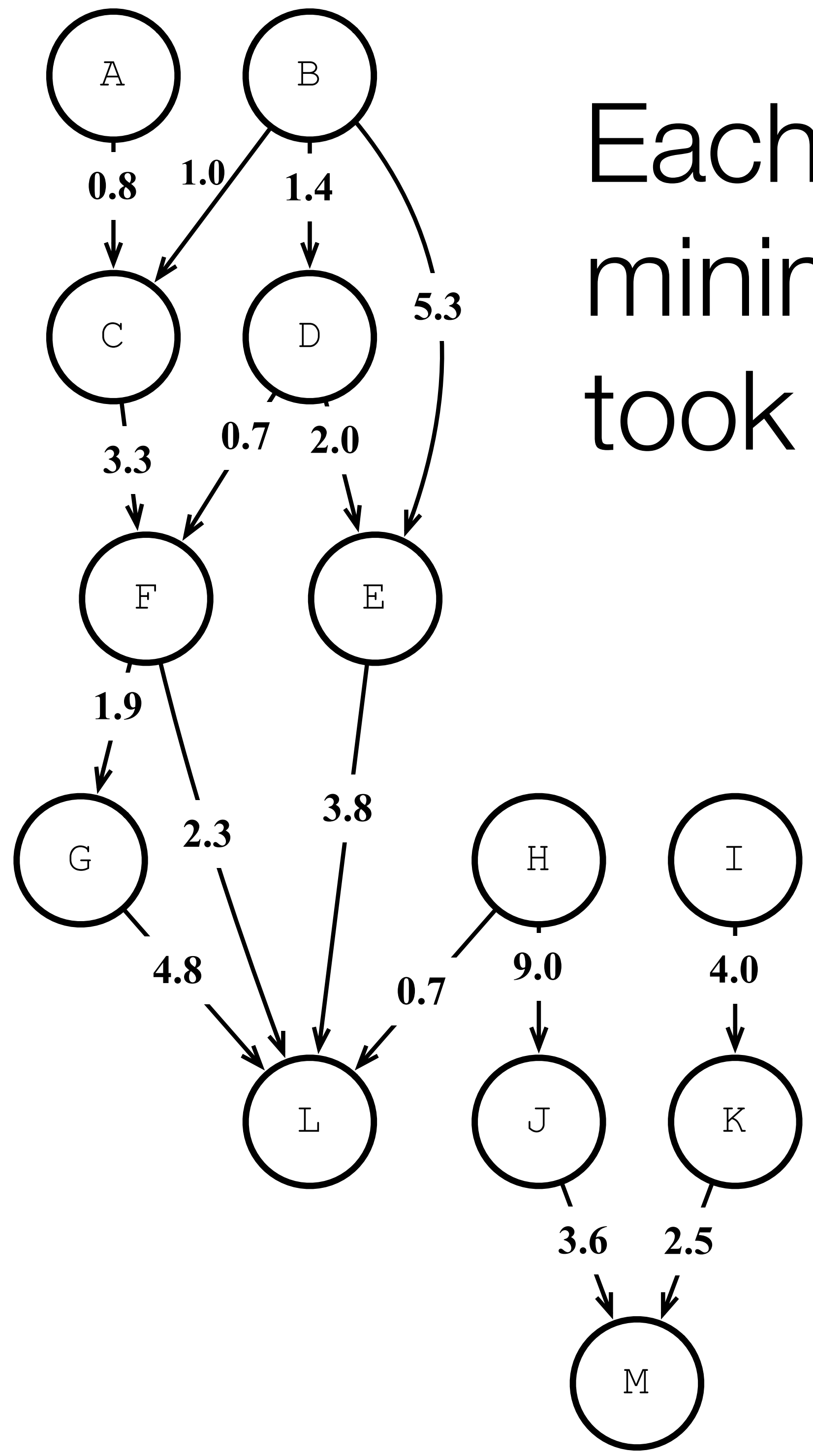


If we start at node **B** and form an MST while searching for node **L**, we create the tree at right.

Each node shows the minimum path cost it took to get there.

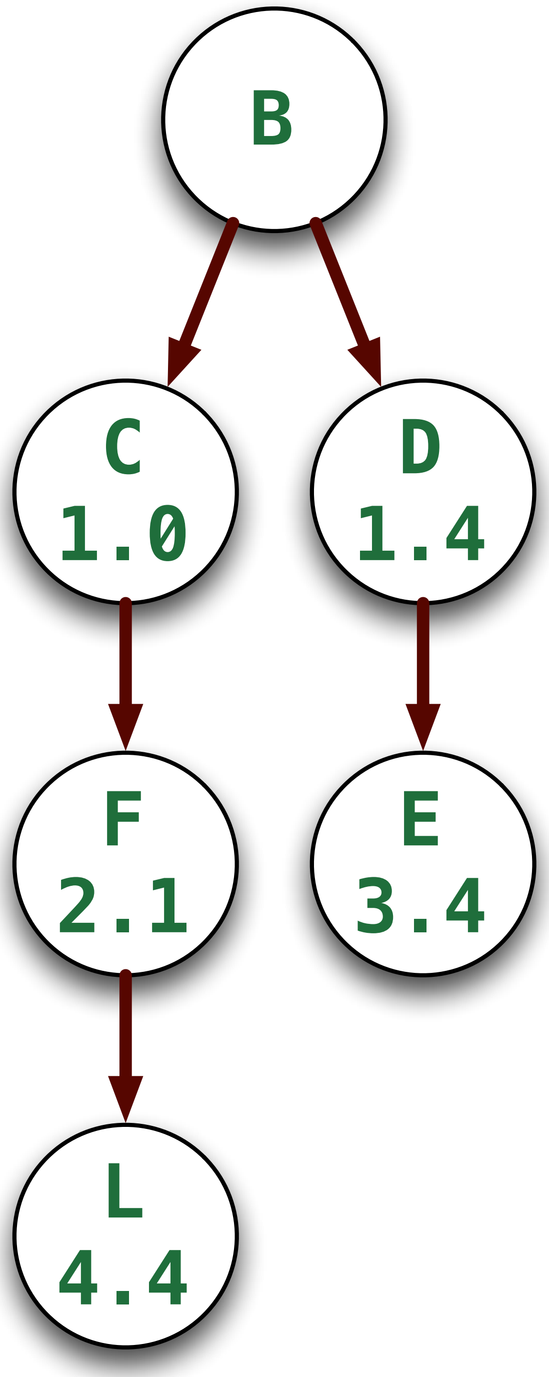


Graphs: Dijkstra's Algo



Each node shows the minimum path cost it took to get there.

The parent/child relation shows the minimum cost path.



Ability to Learn + Adapt

There will be one *evil question* (and it is labeled as such) that asks you to write an algorithm for a data structure that you have not seen before. You'll need to use your understanding of linked lists and graphs in order to answer it completely.

(This question will also let you get extra credit.)

Next Time

Friday is the last lecture. It is *optional*, but don’t let that turn you away.

I just found out that **my company will receive seed money**, so this is probably my last lecture at CU for the foreseeable future. I will make it count!

“HOW TO WIN AT SOFTWARE”