Graphs

15-110 – Wednesday 3/18

Learning Goals

- Define core concepts of **graphs**, including **nodes** and **edges**
- Use **graphs** implemented as dictionaries when reading and tracing code
- Search for values in **graphs** using **breadth-first search** and **depth-first search**

Graphs

Graphs are Like More-Connected Trees

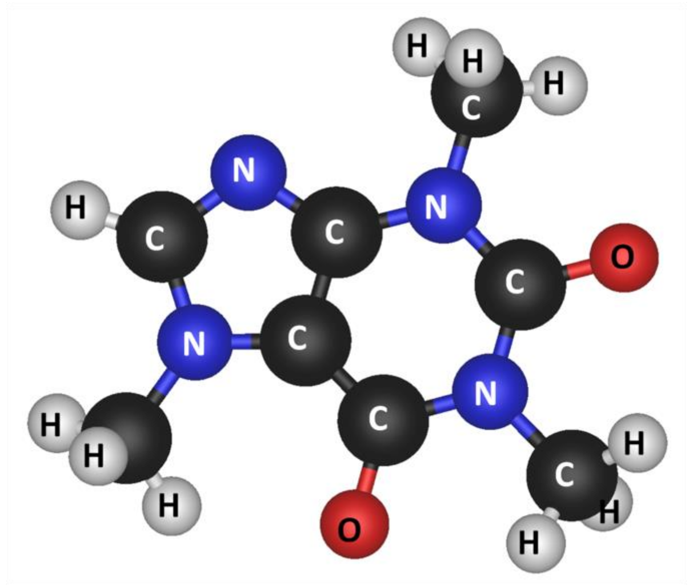
Last week we discussed trees, which let us store data by connecting nodes to each other to create a hierarchical structure.

Graphs are like trees – they use nodes, and connect those nodes together. However, they have fewer restrictions on how nodes can be connected. **Any node can be connected to any other node in the graph**.

Graphs in the Real World

Graphs show up all the time in real-world data. We can use them to represent **maps** (with locations connected by roads) and **molecules** (with atoms connected by bonds).

We also commonly use graphs in algorithms, to represent data like **social networks** (with people connected by friendships), or **recommendation engines** (with items connected if they were purchased together).



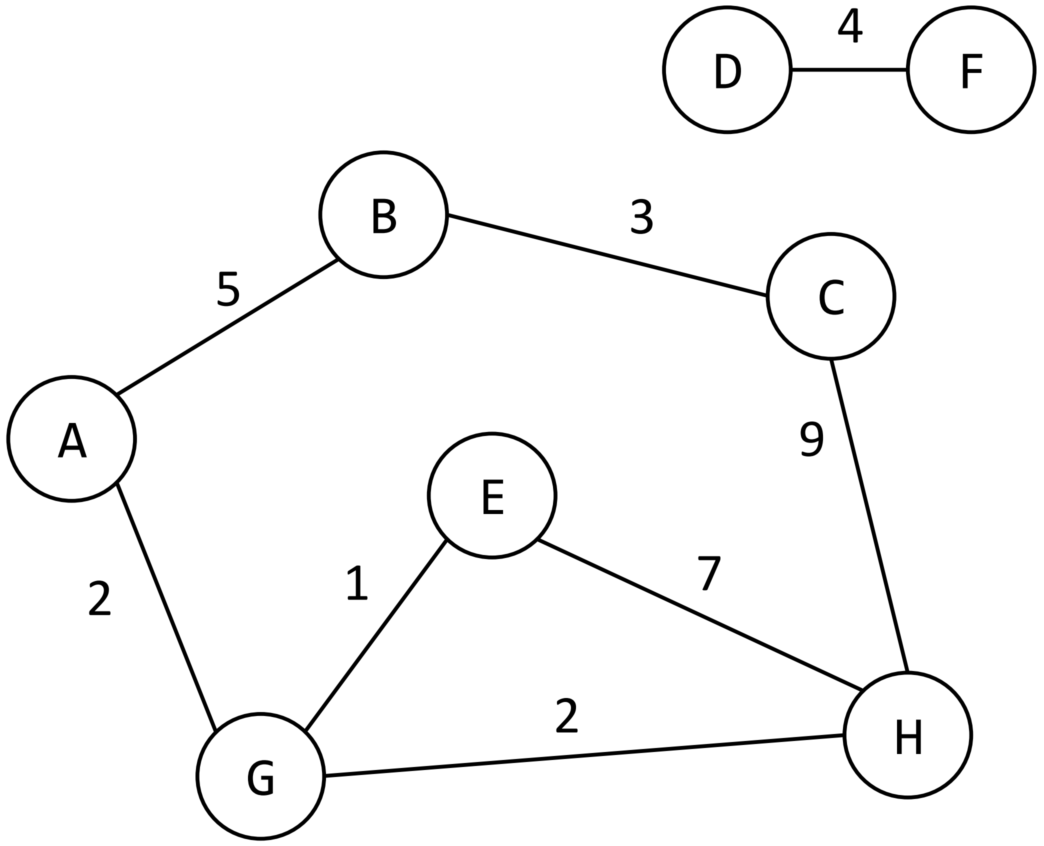
Graphs are Made of Nodes and Edges

The **nodes** in a graph are the same as the nodes in a tree – they hold the values stored in the structure.

The **edges** of a graph are the connections between nodes. Sometimes the edges can have **weights**, such as the length of a road or the cost of a flight.

Edges can be **directed** (from A to B but not from B to A unless there is another directed edge from B to A), or **undirected** (go in either direction on an edge between nodes).

The graph to the right is **weighted and undirected**; if it was directed, we'd add arrows to the lines to show directionality.



Coding with Graphs

Represent Graphs in Python with Dictionaries

Like trees, graphs are not implemented directly by Python. We need to use the built-in data structures to represent them.

Our implementation for this class will use a **dictionary** that maps node values to lists. This is commonly called an **adjacency list**.

Unlike the tree representation, graphs will not be nested dictionaries; we'll be able to access all the node values directly. That's because graphs aren't inherently recursive.

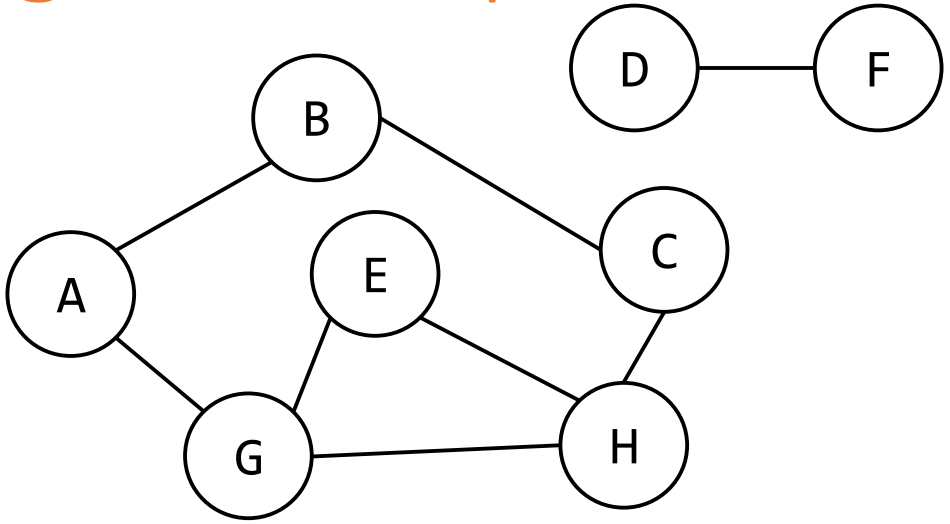
We'll need to slightly alter this representation based on whether or not the edges of the graph have weights.

Graphs in Python – Unweighted Graphs

Graphs with no values on the edges are called **unweighted graphs**.

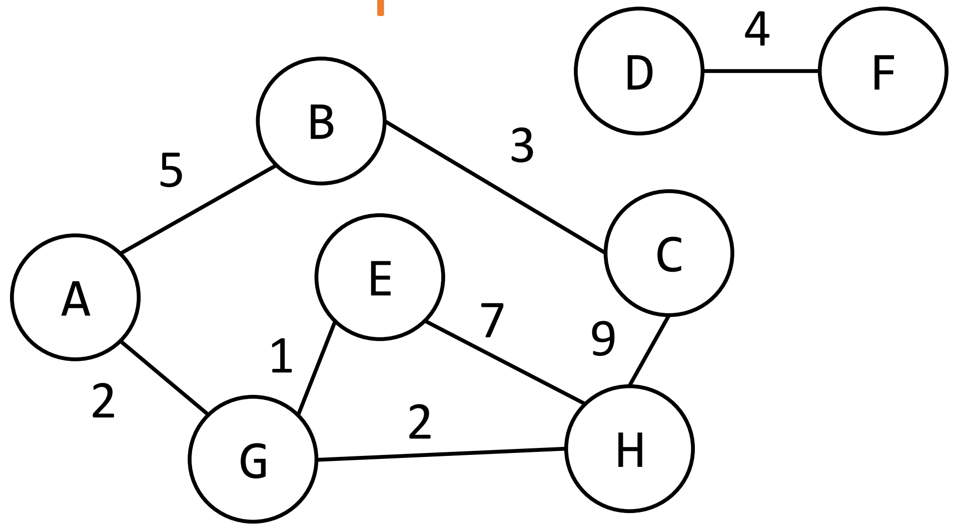
The keys of the dictionary will be the **values of the nodes**. Each node maps to a **list of its adjacent nodes (neighbors)**, the nodes it has a direct connection with.

On the right, we show our example graph in its dictionary implementation.



```
g = {
       "A" : [ "B", "G" ],
       "B" : [ "A", "C" ],
       "C" : [ "B", "H" ],
       "D" : [ "F" ],
       "E" : [ "G", "H" ],
       "F" : [ "D" ],
       "G" : [ "A", "E", "H" ],
       "H" : [ "C", "E", "G" ]
     }
```

Graphs in Python – Weighted Graphs



Weighted graphs have values associated with the edges. We need to store these values in the dictionary also.

We'll do this by changing the list of adjacent nodes to be a 2D list. Each of the inner lists represents a node/edge pair, so it has two values – the adjacent node's value and the weight of the edge.

On the right, we show our updated example graph in this format.

```
g = {
    "A" : [ ["B", 5], ["G", 2] ],
    "B" : [ ["A", 5], ["C", 3] ],
    "C" : [ ["B", 3], ["H", 9] ],
    "D" : [ ["F", 4] ],
    "E" : [ ["G", 1], ["H", 7] ],
    "F" : [ ["D", 4] ],
    "G" : [ ["A", 2], ["E", 1], ["H", 2] ],
    "H" : [ ["C", 9], ["E", 7], ["G", 2] ]
   }
```

Example: Most Popular Person

Let's write a function that takes a social network as a graph and returns the person in the network who has the most friends.

This is just our typical find-largest-property algorithm, but applied to a graph.

```
def findMostPopular(g):
    biggestCount = 0
    mostPopular = None
    for person in g:
        if len(g[person]) > biggestCount:
            biggestCount = len(g[person])
            mostPopular = person
    return mostPopular
```

Example: Make Invite List

Now let's say that popular person wants to make even more friends, so they're holding a party. They want to invite their own friends, but also anyone who is a friend of one of their friends.

Now we have to loop over each of the person's friends, to access that node's own list of friends.

```
def makeInviteList(g, person):
    invite = g[person] + [ ] # break alias
    for friend in g[person]:
        for theirFriend in g[friend]:
            if theirFriend not in invite and \
               theirFriend != person:
                invite.append(theirFriend)
    return invite
```

Searching Graphs

Searching a graph

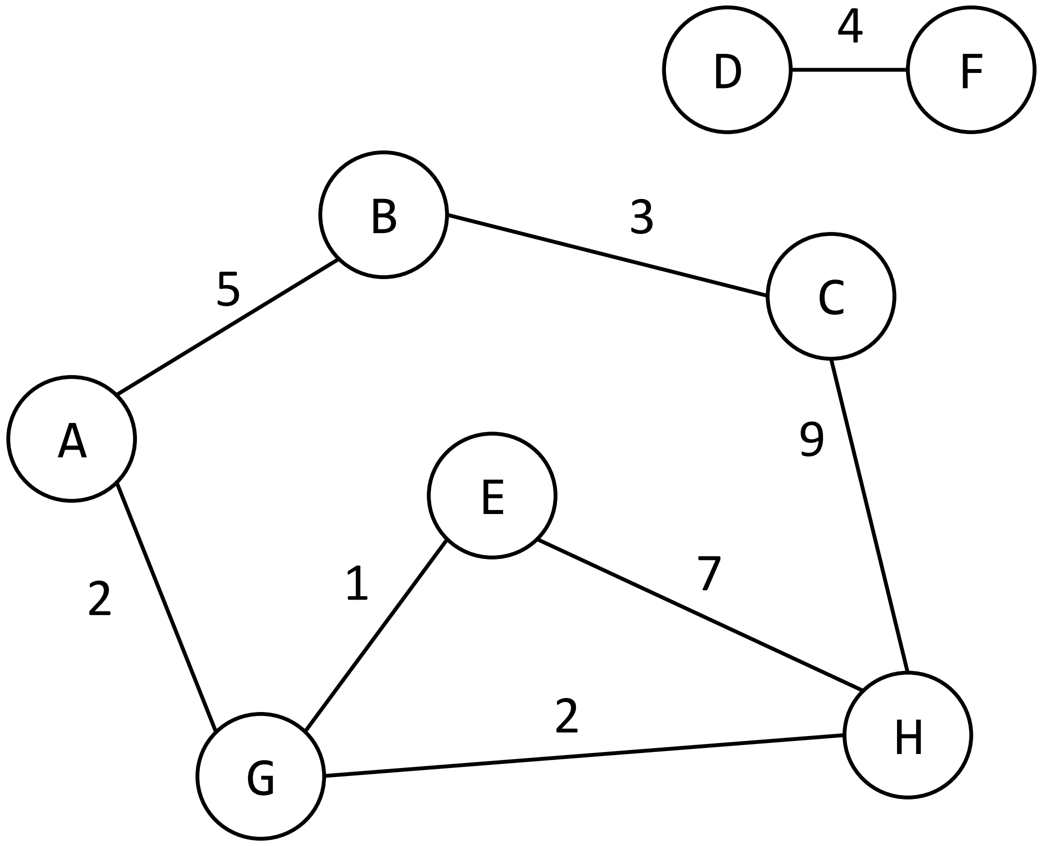
Now let's attempt a familiar but more complicated problem: search.

Determining whether a node exists in a graph is too easy (just look at the keys). We'll ask a more difficult question: can we build a path from a specific **starting node** to the node we're looking for?

Discuss: How to Search?

How would you systematically search the graph shown here to see if there's a path between A and C?

Alternatively, how would you systematically check if there's a path between A and D?



Two Search Algorithms: BFS and DFS

We'll need to start at the start node and follow the edges to find all the other nodes it's connected to. There are two common approaches for determining in which order to visit the connected nodes.

In **Breadth-First Search (BFS)**, we slowly move outwards in the graph from the start node. We visit all the neighbors of start, then visit all the neighbors of the already visited nodes, etc., until we've checked all the nodes that were connected to the start node the graph.

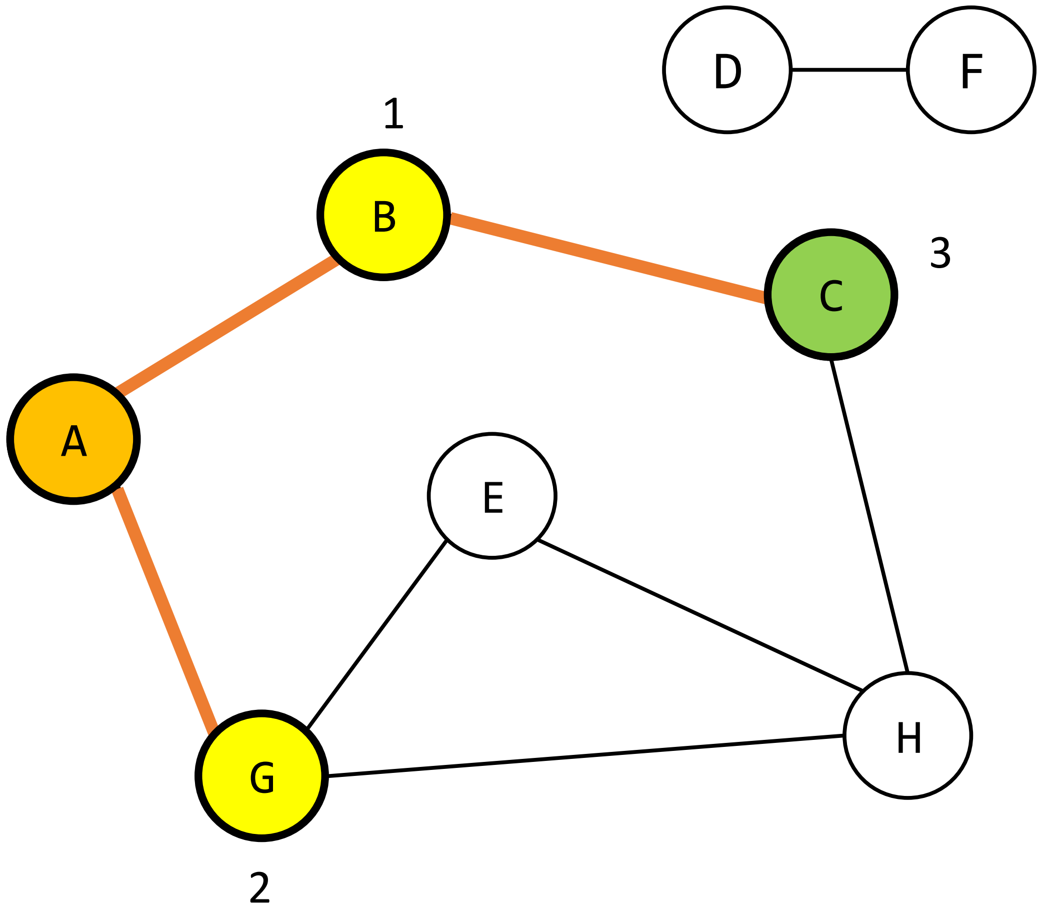
In **Depth-First Search (DFS)**, we go all the way down one potential path, then backtrack and try other possible paths. So we choose one neighbor, then choose one of its neighbors, etc., until there are no unvisited neighbors left.

Breadth-First Search Example

Let's consider Breadth-First Search on our example graph, starting from A and searching for C.

A has two neighbors, B and G. We can visit B and then G, or G and then B.

Once both have been visited, we visit B and G's neighbors – C, E, and H. (A is a neighbor as well, but we don't visit it because it's been visited before.) As soon as we reach C, we've found the node, and we're done!



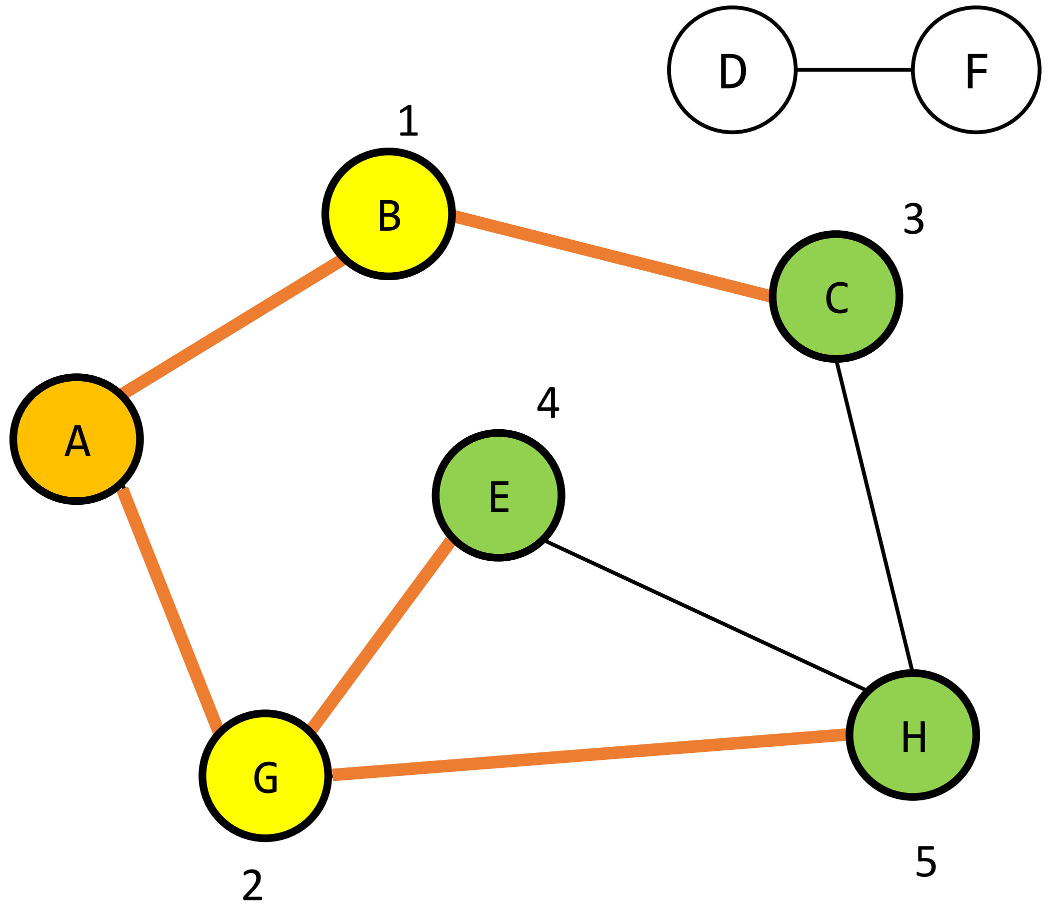
Breadth-First Search Example

Now let's run Breadth-First Search starting from A and searching for a value not connected to it, D.

A has two neighbors – B and G. As before, we can visit B and then G, or G and then B.

Once both have been visited, we visit B and G's neighbors – C, E, and H. Again, these can be visited in any order (CEH, CHE, ECH, EHC, HEC, HCE). We don't revisit A.

At this point, there are no nodes left that are neighbors of C, E, and H and have not been visited. We conclude there is no path from A to D.

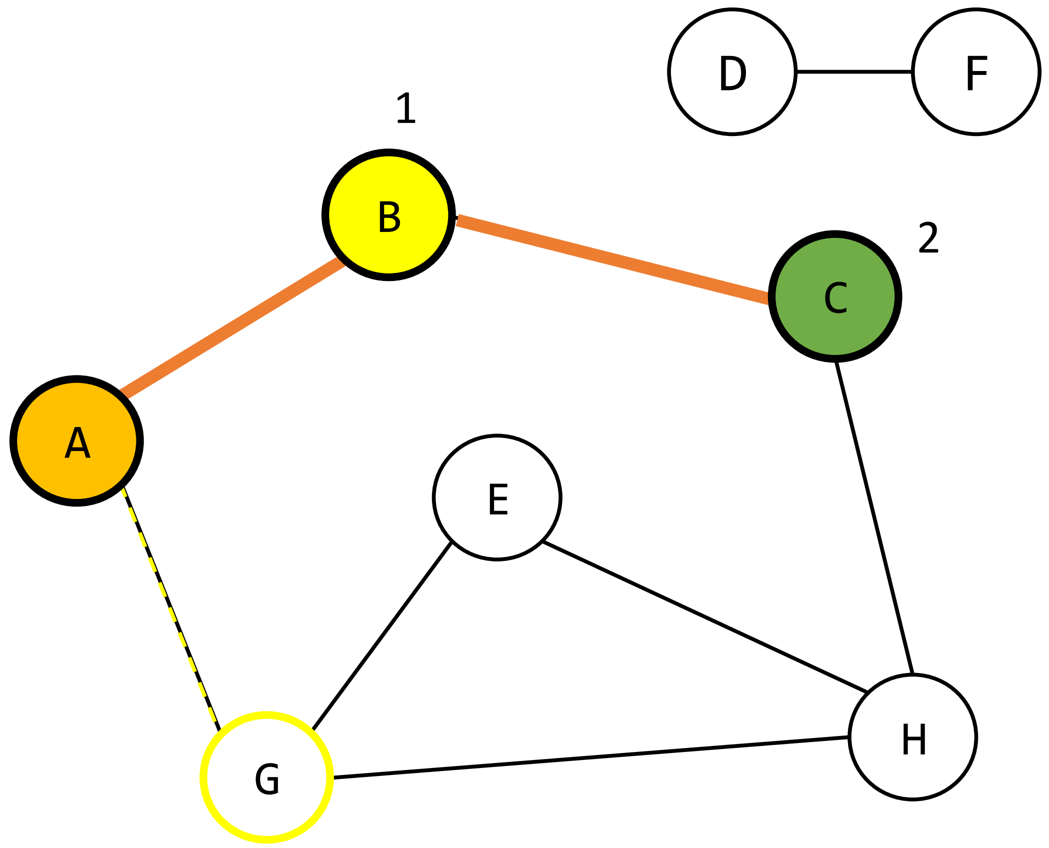


Depth-First Search Example

Now let's search the example graph starting from A with depth-first search, searching for C.

There are two possible starting routes: B or G. Let's choose B. We'll store G as a backup option, in case we run into a dead end.

From B, we only have one unvisited neighbor: C. We've found the node we're looking for, so we're done!



Depth-First Search Example

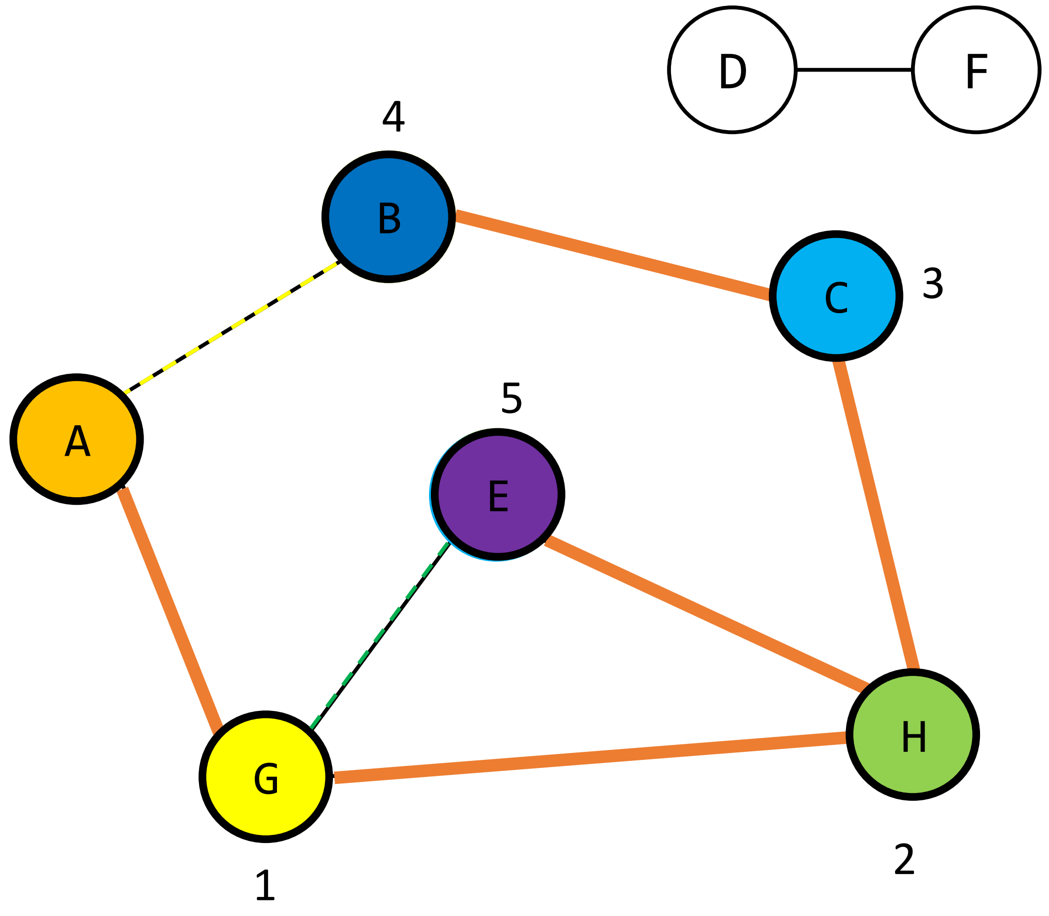
What if we search the example graph starting from A with depth-first search, now looking for D?

There are two possible starting routes: B or G. Choose G, and place B in the **backup list**.

From G, we have two possible routes, E or H; choose H and mark E as backup. Note that A is not a valid choice, as it's already been visited.

From H, we have two more possible routes: E or C (G is not valid). We'll choose C. C's only remaining neighbor is B (H is not valid), so we must visit it.

Now B has no unvisited neighbors remaining (A and C are both visited), so we must **backtrack** to the last node that had an unvisited neighbor. If we check our backup list, the only unvisited node remaining is E (which was G and H's neighbor). We visit E, and we're done.

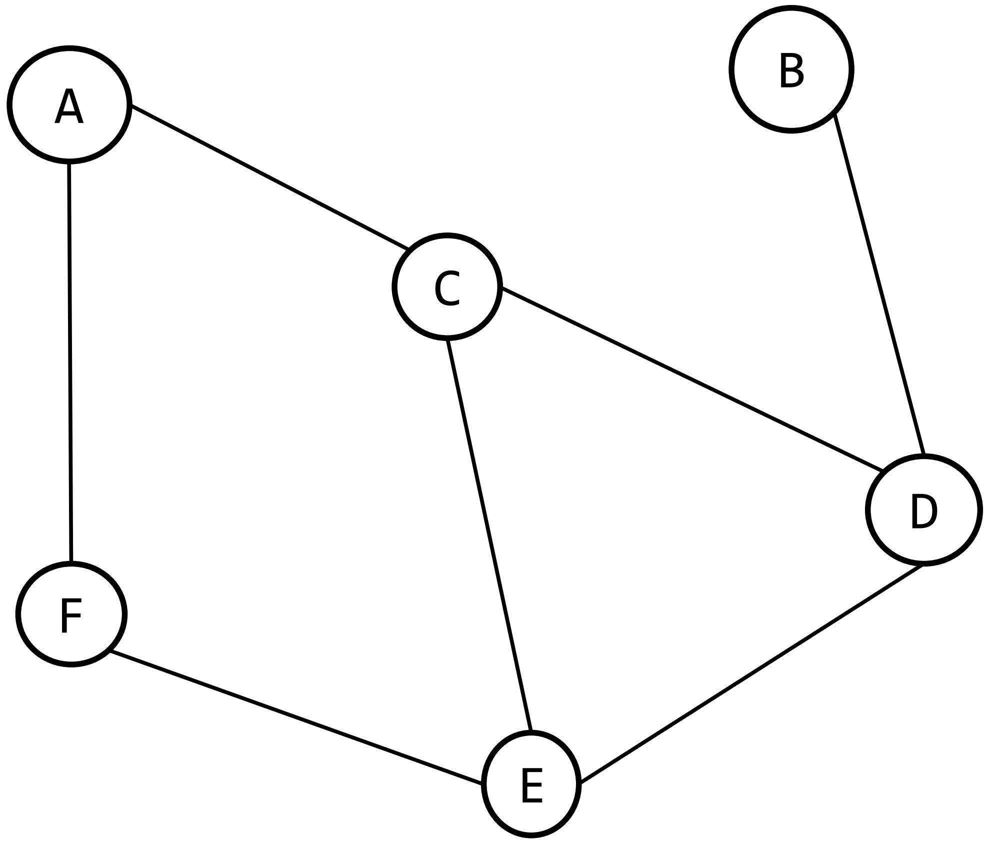


Activity: BFS and DFS Tracing

Given the graph to the right and starting from A, which potential trace through the graph is a valid trace for **Breadth-First Search,** and then for **Depth-First Search**?

Choose your answer on Piazza.

Note that not all possible answers are included. Just visit neighbors **alphabetically** (which following the search rules) to make things simpler.



Coding BFS and DFS

To code these search algorithms, we'll need to keep track of two pieces of data. One is the **nodes we need to search next**. The other is **the nodes we've already visited**. It's important to keep track of what we've visited so far, to avoid cycling back to nodes we've seen before and looping forever!

We'll use a **while loop** to iterate over the nodes we need to search, since we'll update the list as we go. Each iteration will check the next node that hasn't been visited yet on the to-search list, to see if it's the one we're looking for.

If we find the node, we'll return True right away. If we don't, we'll add all the node's neighbors to the to-visit list. **How we add the nodes changes based on whether we implement BFS or DFS**.

Breadth-First Search Code

Note that in the BFS code, we add neighbors of each node we visit to the **end** of the to-visit list. This prioritizes neighbors that are connected earlier in the graph.

```
def breadthFirstSearch(g, start, item):
    # Set up two lists for visited nodes and to-visit nodes
    visited = [ ]
    nextNodes = [ start ]

    # Repeat while there are nodes to visit
    while len(nextNodes) > 0:
        next = nextNodes[0]
        nextNodes.pop(0)

        # Only check this node if we haven't visited it before, to avoid repeats
        if next not in visited:
            visited.append(next)

            if next == item: # If it's what we're looking for- we're done!
                return True
            else: # Otherwise, add the neighbors to the back of the to-visit list
                nextNodes = nextNodes + g[next]
    return False
```

Depth-First Search Code

In the DFS code, we add neighbors of each node we visit to the **start** of the to-visit list. This prioritizes neighbors that are connected deeper inside the graph. Otherwise, the algorithm is the same.

```
def depthFirstSearch(g, start, item):
    # Set up two lists for visited nodes and to-visit nodes
    visited = [ ]
    nextNodes = [ start ]

    # Repeat while there are nodes to visit
    while len(nextNodes) > 0:
        next = nextNodes[0]
        nextNodes.pop(0)

        # Only check this node if we haven't visited it before, to avoid repeats
        if next not in visited:
            visited.append(next)

            if next == item: # If it's what we're looking for- we're done!
                return True
            else: # Otherwise, add the neighbors to the front of the to-visit list
                nextNodes = g[next] + nextNodes
    return False
```

Testing BFS and DFS

We only change one line of code between BFS and DFS, but it makes a big difference in the way the algorithms work. The test code to the right demonstrates how the algorithm moves through the nodes of an example graph for two different nodes.

When we search for `"D"`, BFS is more efficient- it only needs to check three nodes, compared to DFS's four. But when we search for `"E"`, DFS only takes three moves, compared to BFS's five!

Both algorithms are **O(n)**, where n is the number of nodes in the graph, because both must check every node in the graph in the case that the sought node doesn't exist and the graph is fully connected.

```
g = { "A" : [ "B", "D", "F" ],
      "B" : [ "A", "E" ],
      "C" : [ ],
      "D" : [ "A", "E" ],
      "E" : [ "A", "B", "D" ],
      "F" : [ "A" ]
    }

breadthFirstSearch(g, "A", "D")
breadthFirstSearch(g, "A", "E")

depthFirstSearch(g, "A", "D")
depthFirstSearch(g, "A", "E")
```

Learning Goals

- Define core concepts of **graphs**, including **nodes** and **edges**
- Use **graphs** implemented as dictionaries when reading and tracing code
- Search for values in **graphs** using **breadth-first search** and **depth-first search**