Graphs

Graphs

- Graphs are used to represent data where connections between data points are complex and important
- They consist of a set of **nodes** and **edges** between them
- Graphs are more general like trees and can represent relationships that aren't hierarchical
- If we can represent our data as some sort of graph, we can use a bunch of existing algorithms that operate on graphs

Terminology

- Graphs can be `directed` or `undirected`
- In a directed graph (aka a digraph) edges have a direction (an arrow with a point)
- In undirected graphs, edges are undirected (duh), meaning that there is no orientation and the order you list the nodes being connected doesn't matter
- In some graphs, edges are `weighted` meaning there's a number associated with each edge representing some sort of “cost” of moving along that edge
- A node's `neighbors` are the nodes that are connected to it via edges (nodes that are 1 edge away)

Simple Graphs

- We'll work with “simple” graphs
- Simple graphs only allow 0 or 1 edges between a pair of nodes (you can't have multiple edges between the same nodes)
- self edges are also disallowed (an edge that connects a node to itself)
- These are sometimes useful things to add, and won't require major changes to the algorithms we discuss to handle

Paths

- Lots of graph analysis is related to finding “paths” on graphs
- A path is a sequence of connected edges ie, put your pen down on a node and follow whatever edges you want without lifting your pen (you must start/end at nodes)
- A `cycle` is a path that starts + ends at the same node
- An undirected graph is `connected` if there is a path between every pair of nodes
- A directed graph is `strongly connected` if there is a path between every pair of nodes. We get a new term because since the edges have a direction there may be a path from A → B but not B → A

Special Graphs

- Trees: No loops
- “Directed Acyclic Graphs”: directed analog of trees. We can draw these like a tree but lower nodes may have multiple parents. But, there are no directed edges that point “up”
- Bipartite Graphs — 2 “groups” of nodes and all edges are between nodes in different groups.
- Complete Graph — Graph with all possible edges included (undirected)

Graphs in code

- Link base representation (similar to our tree representation):

```
class graph:
    class Node:
        data
        Set<node> edges/neighbors

    Set<nodes> nodes
```

Pathfinding

- A common problem in graphs is finding paths between pairs of nodes
- The algorithms vary based on whether the graph is directed/undirected and whether edges have weights
- Examples are finding a flight between SLC and Toledo or from U of U to Ikea via public transit
- Often we want to find the shortest or minimum cost (minimum weight) path

Pathfinding algorithms

- All pathfinding algorithms follow the same structure:

```
path(start, end)
    toExplore = someDatastructure(start)
    while !toExplore.empty:
        n = toExplore.removeSomeNode()
        if n is end, reconstruct path to start and return //easy
        add neighbors of n to toExplore
        maybe update some per node data
```

Pathfinding algorithms

- The algorithms differ based on which data structure they use to store nodes which affects the order we visit the nodes
- We'll start with the 2 simplest: depth first search and breadth first search

Depth First Search (DFS)

- Recursive version: pseudocode

```
DFS(Node n, target):
    n.visited = true
    foreach neighbor of n:
        if(!neighbor.visited):
            pathFromHere = dfs(neighbor, target)
            if pathFromHere is not null:
                return  n + pathFromHere
    return null // no path from here
```

- Which data structure are we using to store nodes to visit?

Iterative DFS

- Explicitly store toExplore nodes in a stack
- Add a “cameFrom” field to each node and set it to null intially. Follow the cameFrom fields from end to start to reconstruct the path
- Are these good paths?

Breadth first search: finding shortest paths

- Can we change the data structure to guarantee we get the shortest path?
- What order should we explore nodes for that to work?

BFS Pseuducode:

```
BFS(start, end)
    start.visited = true
    Q = queue(start)
    while !Q.empty
        n = dequeue(Q)

        if n is end:
            follow cameFrom links back to start, return
        for neighbor of n:
            if !neighbor.visited:
                neighbor.visited = true
                neighbor.cameFrom = n
                enqueue(neighbor)
    return no path
```