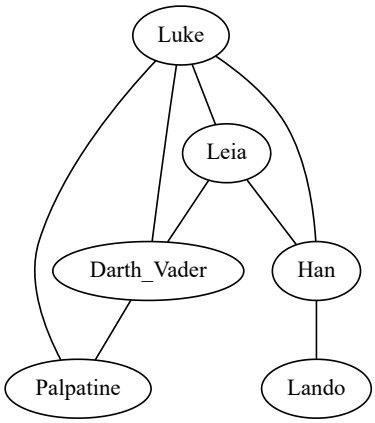
Graphs

Download Demo Code <../dsa-graphs-demo.zip>

Goals

- Learn what a graph is
- Compare and contrast different types of graphs
- Code a graph implementation
- Check if two nodes are connected

What is a Graph?



Graphs are like trees, except they can contain loops (“cycles”).

Also, the relationships can be directed or un-directed.

Terminology

Node (or Vertex)
basic unit

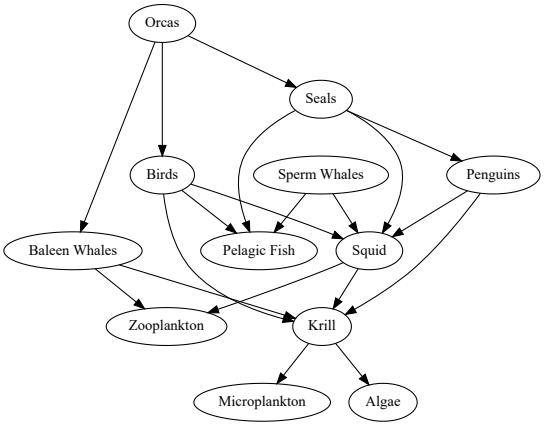
Edge (or Arc)
connects two nodes

Adjacent
two nodes are “adjacent” if they share an edge

Weight (optional)
each edge can have a weight (ex: price, or distance)

Examples

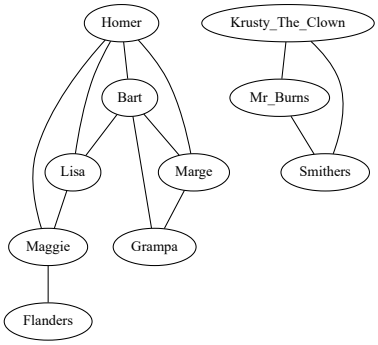
Food Chain



This graph is **directed**, showing “what eats what”

Penguins’ adjacency list: `[Squid, Krill]`

Facebook Friends (or LinkedIn)



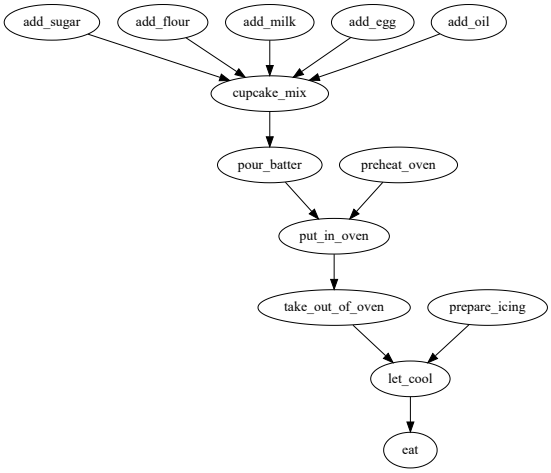
This graph is **undirected**

Homer’s adjacency list: `[Bart, Lisa, Maggie, Marge]`

Lisa’s adjacency list: `[Maggie, Bart, Homer]`

Processes

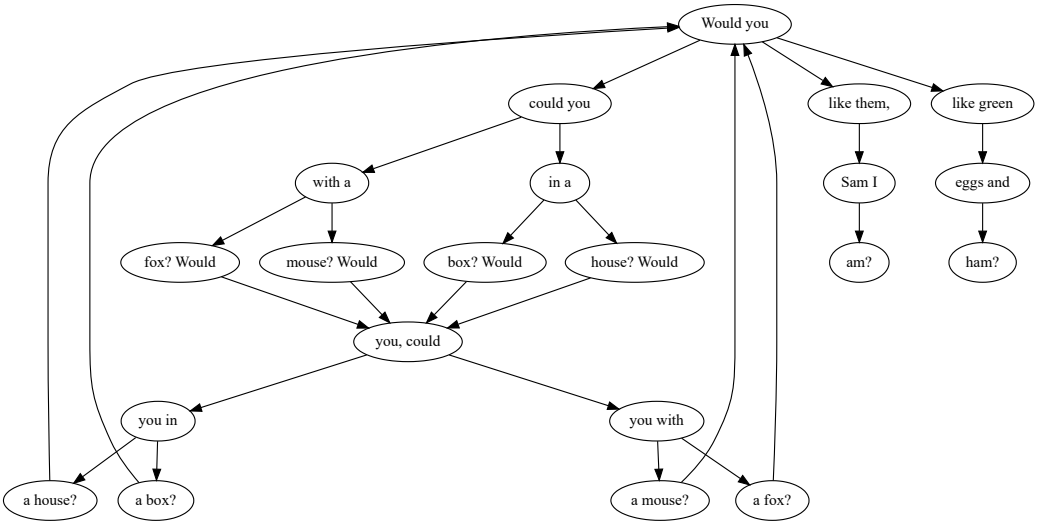
Making Cupcakes:



Don’t want to do a step until the necessary prerequisites are done!

Similar idea for manufacturing processes, supply chains, etc.

Markov Chains



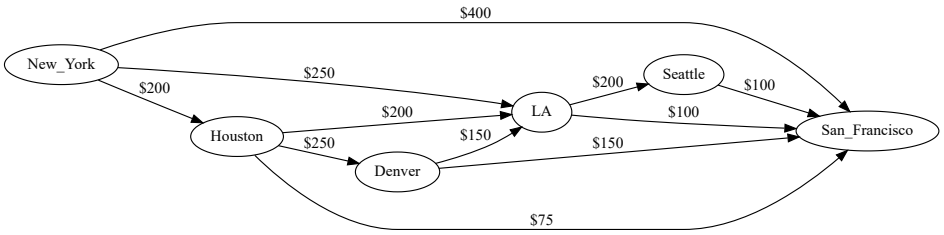
Other Markov chains: states of health and disease, finance

Airline Route Map

Each node is an airport. Each edge is a flight.

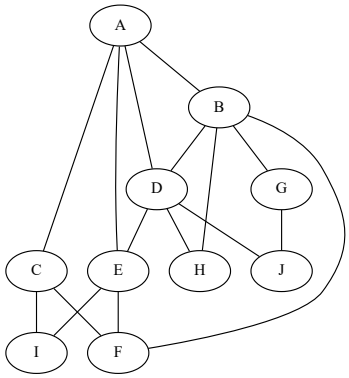
The weight of each edge is the price.

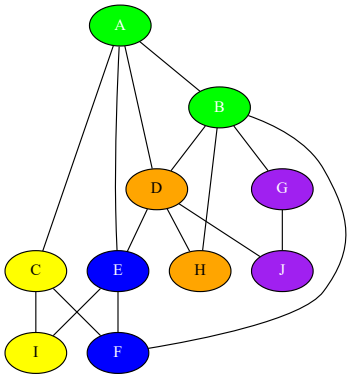
What is the cheapest way to go from New York to San Francisco?



Carpooling

Each node is a rider, and edges represent possible carpooling matches. Only two people can carpool together at a time. How can we match the maximum number of pairs of riders?





There exists a solution where everyone gets a pair. Can you find it?

Graphs

- Graphs are often used to model relationships between things
- Trees are **directed**, **acyclic** graphs
- All trees are graphs, but not all graphs are trees
- Trees have hierarchy, graphs do not

Linked Lists, Trees, and Graphs

Linked lists, trees, and graphs are all structures that have a relationship, much like squares, rectangles, and parallelograms do. A linked list is a special, more-restricted form of a tree, and a tree is a special, more-restricted form of a graph.

Linked List
: Nodes have 0 or 1 child; acyclic and directed

Tree
: Nodes have 0+ children; acyclic and directed; only one designated root node

Graphs
: Nodes have 0+ connections; cyclic or acyclic; directed or undirected; disconnected or connected; optional weights

There are other possibilities, including:

- there are “circular linked lists,” where the linked list can contain a cycle (A points to B points to C which points to B). These do not have tails, as there’s no single end-point.
- there are “forests,” which are collections of directed, acyclic graphs but without a single root node. This essentially is a set of trees, hence a “forest.”

Code

Representing a Graph

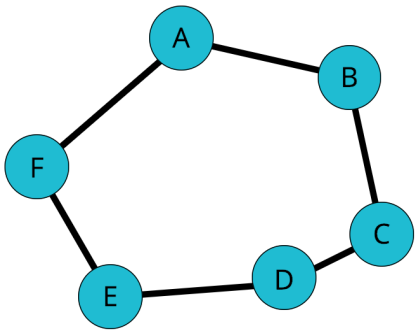
Adjacency List

for node, a list of every node it is directly connected to

Adjacency Matrix

a matrix of every pair of nodes, with a 1 if that pair is connected (otherwise 0)

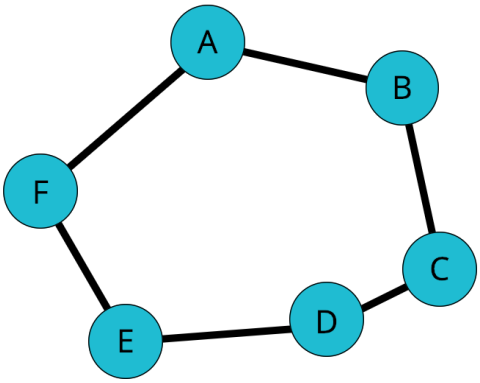
Representing a Graph - Adjacency Lists




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

<_images/adj-list.png>

Representing a Graph - Adjacency Matrix



-	A	B	C	D	E	F
A	0	**1**	0	0	0	**1**
B	**1**	0	**1**	0	0	0
C	0	**1**	0	**1**	0	0
D	0	0	**1**	0	**1**	0
E	0	0	0	**1**	0	**1**
F	**1**	0	0	0	**1**	0

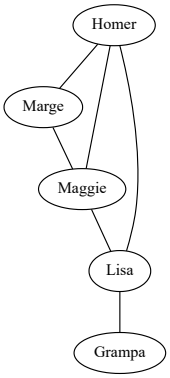
<_images/adj-matrix.png>

We're going to use Adjacency Lists!

They're more common.

Adjacency matrices can be preferred for graphs that are highly connected.

Friend Graph



Node and Graph Class

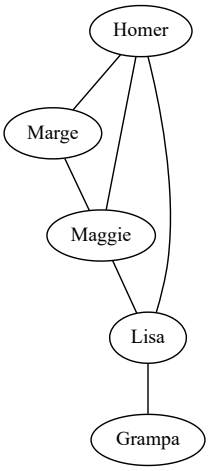
demo/friends.js

```
class PersonNode {
  constructor(name, adjacent = new Set()) {
    // Create a person node with friends adjacent
    this.name = name;
    this.adjacent = adjacent;
  }
}
```

demo/friends.js

```
class FriendGraph {
  // Graph holding people and their friendships.
  constructor() {
    this.nodes = new Set();
  }

  addPerson(person) {
    // Add a person to our graph
    this.nodes.add(person);
  }

  setFriends(person1, person2) {
    // Set two people as friends
    person1.adjacent.add(person2);
    person2.adjacent.add(person1);
  }

  addPeople(people_list) {
    // Add a list of people to our graph
    for (let person of people_list) {
      this.addPerson(person);
    }
  }
}
```

Demo: *friends.js*



demo/friends.js

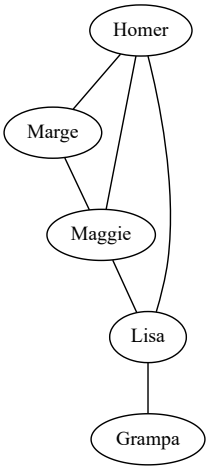
```
let homer = new PersonNode("Homer");
let marge = new PersonNode("Marge");
let maggie = new PersonNode("Maggie");
let lisa = new PersonNode("Lisa");
let grampa = new PersonNode("Grampa");

let friends = new FriendGraph();
friends.addPeople([homer,marge,maggie,lisa,grampa]);

friends.setFriends(homer, marge);
friends.setFriends(homer, maggie);
friends.setFriends(homer, lisa);
friends.setFriends(marge, maggie);
friends.setFriends(lisa, grampa);
```

Graph Traversal

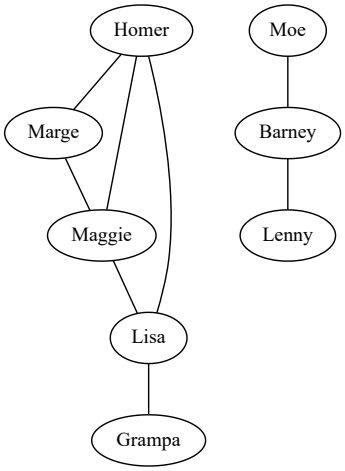
Problem:

Write a function that checks if two people are connected.

Is Marge connected to Grampa?



Is Marge connected to Moe?



How do we figure this out?

- We need to traverse through the graph
- We want to make sure we only visit each vertex once
- But how do we search through it?
  - *BFS* - go to all closest neighbors and work your way outwards
  - *DFS* - continue on a path until it's exhausted

Not like your tree traversal

- This one is a bit different!
- Since graphs can have cycles, we need to be sure not visit same node again!
- How can we mark a node as visited?

Graph Breadth First Search

Solution

demo/friends.js

```
areConnectedBFS(person1, person2) {
  let toVisitQueue = [person1];
  let seen = new Set(toVisitQueue);

  while (toVisitQueue.length > 0) {
    let currPerson = toVisitQueue.shift();

    if (currPerson === person2) return true

    for (let neighbor of currPerson.adjacent) {
      if (!seen.has(neighbor)) {
        toVisitQueue.push(neighbor);
        seen.add(neighbor);
      }
```

```
    }
  }

  return false;
}
```

This is a *breadth-first* search (would be *depth-first* if we used a stack)

Graph Depth First Search

Another Iterative Approach

demo/friends.js

```
areConnectedDFS(person1, person2) {
  let toVisitStack = [person1];
  let seen = new Set(toVisitStack);

  while (toVisitStack.length > 0) {
    let currPerson = toVisitStack.pop();

    if (currPerson === person2) return true;

    for (let neighbor of currPerson.adjacent) {
      if (!seen.has(neighbor)) {
        toVisitStack.push(neighbor);
        seen.add(neighbor);
      }
    }
  }

  return false;
}
```

Recursive Solution

demo/friends.js

```
areConnectedRecursive(person1, person2, seen=new Set([person1])) {
  if (person1 === person2) return true;

  for (let neighbor of person1.adjacent) {
    if (!seen.has(neighbor)) {
      seen.add(neighbor);
      if (this.areConnectedRecursive(neighbor, person2, seen)) {
        return true;
      }
    }
  }
```

```
  return false;
}
```

This is a recursive *depth-first* search

Further Study

Gentle Introduction to Graph Theory <https://medium.com/basecs/a-gentle-introduction-to-graph-theory-77969829ead8>

BFS Graph Traversal <https://medium.com/basecs/going-broad-in-a-graph-bfs-traversal-959bd1a09255>

From Theory to Practice: Representing Graphs <https://medium.com/basecs/from-theory-to-practice-representing-graphs-cfd782c5be38>

- Visualizations: Visualgo.net <https://visualgo.net/en>
- Problem Solving with Algorithms and Data Structures <http://interactivepython.org/courselib/static/pythonds/index.html> (awesome FREE book!)
- Graph Database: Neo4j
- Joe Celko, *SQL for Smarties* (graphs and trees in SQL)