Graph-theoretic Models

Lesson 5

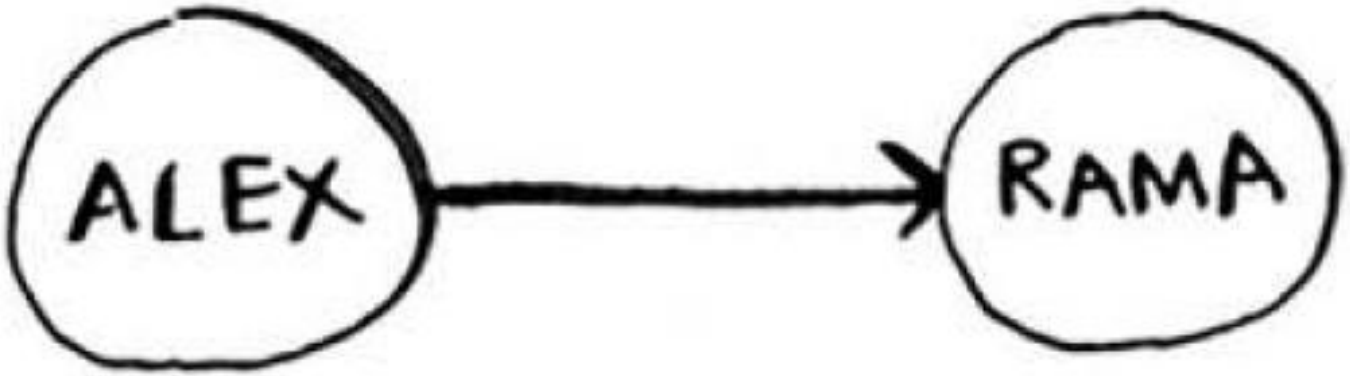
objectives

- basic of graphs
- how to implement graphs
- shortest path problem
- search
  - Breadth-first search (implementation 1)
  - Depth-first search (implementation)
  - Breadth-first search (implementation 2)

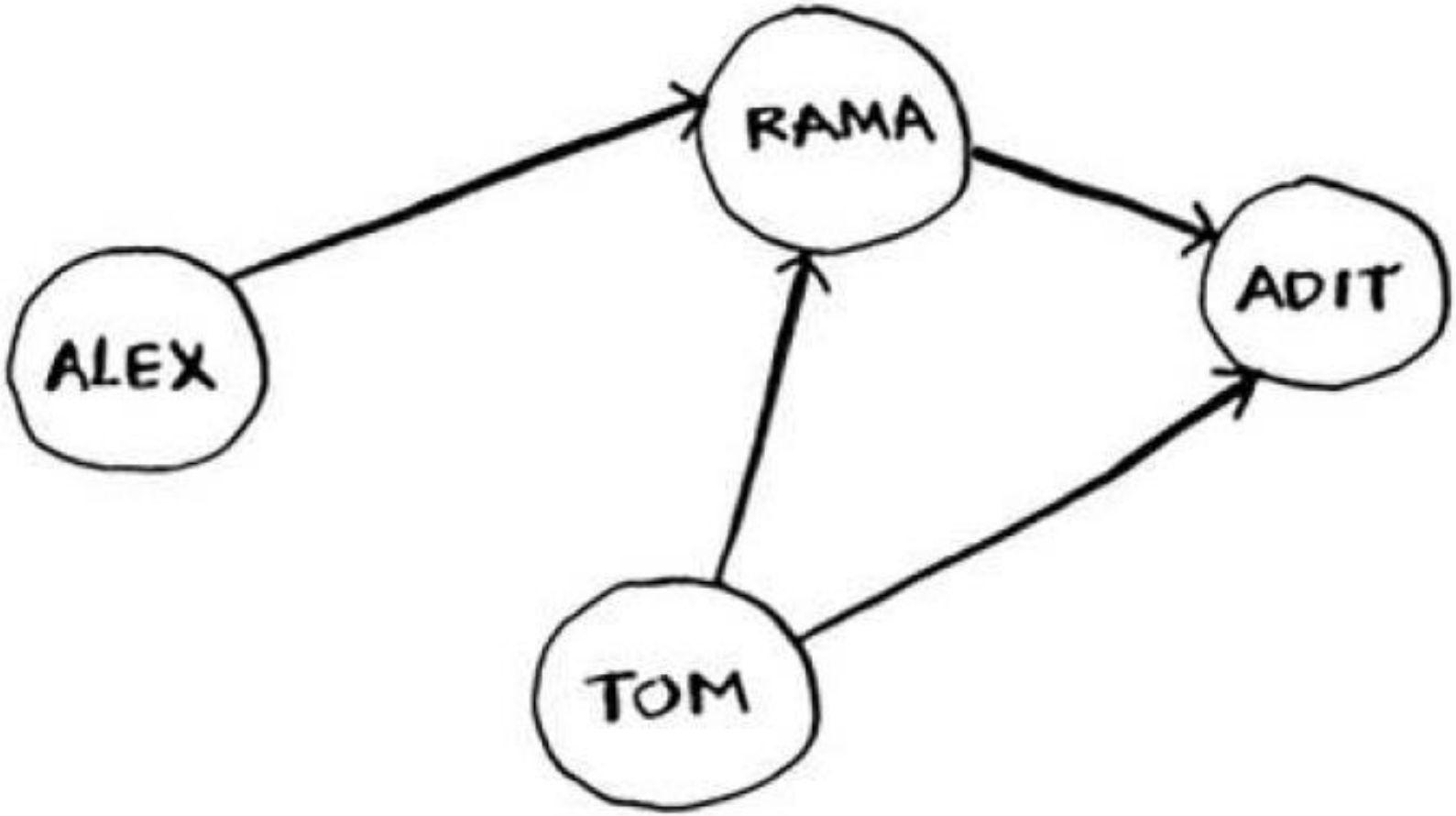
Computational Models

- Programs that help us understand the world and solve practical problems
- Saw how we could map the informal problem of choosing what to eat into an optimization problem, and how we could design a program to solve it
- Now want to look at class of models called graphs

Who owes whom money ?

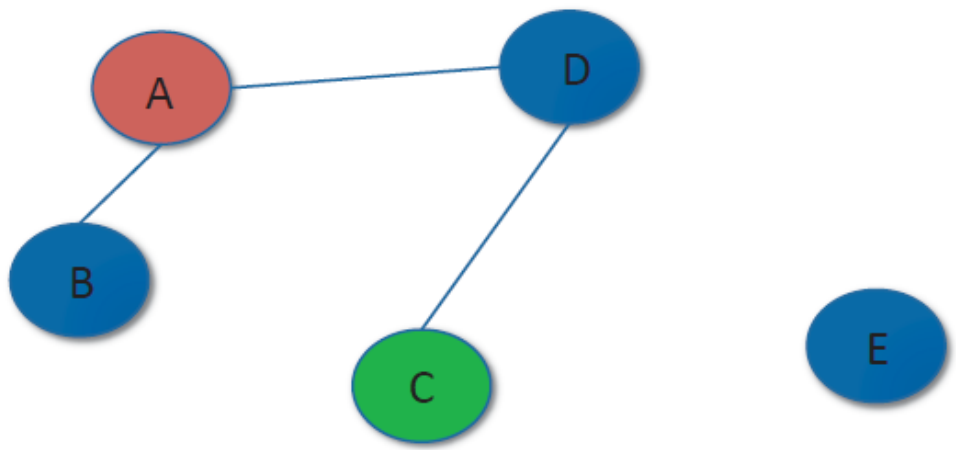


the full graph could look like this.



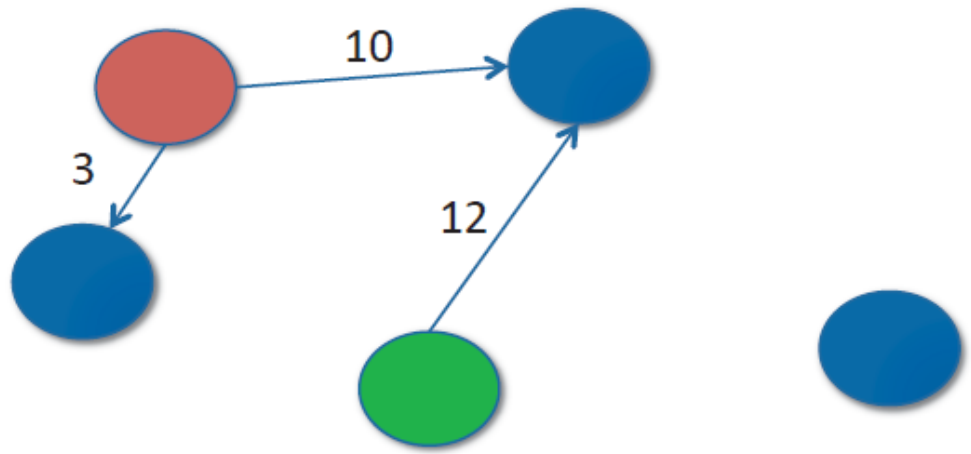
What's a Graph?

- Set of nodes (vertices)
  - Might have properties associated with them
- Set of edges (arcs) each consisting of a pair of nodes
  - Undirected (graph)
  - Directed (digraph)
    - Source (parent) and destination (child) nodes
  - Unweighted or weighted



What's a Graph?

- Set of nodes (vertices)
  - Might have properties associated with them
- Set of edges (arcs) each consisting of a pair of nodes
  - Undirected (graph)
  - Directed (digraph)
    - Source (parent) and destination (child) nodes
  - Unweighted or weighted

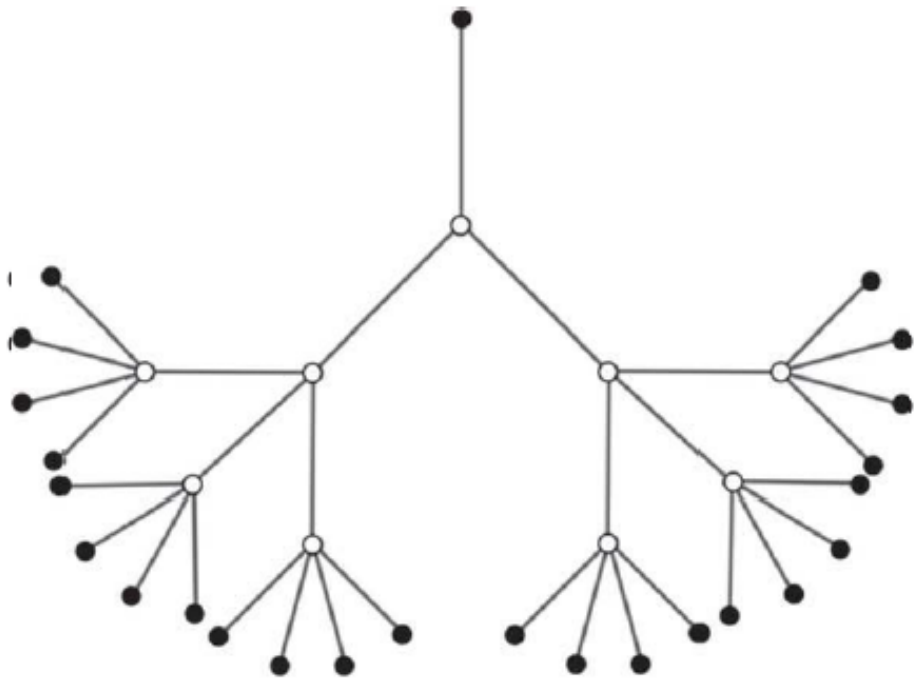


Why Graphs?

- To capture useful relationships among entities
  - Rail links between Paris and London
  - How the atoms in a molecule are related to one another
  - Ancestral relationships

Trees: An Important Special Case

- A special kind of directed graph in which any pair of nodes is connected by a single path
  - Recall the search trees we used to solve knapsack problem



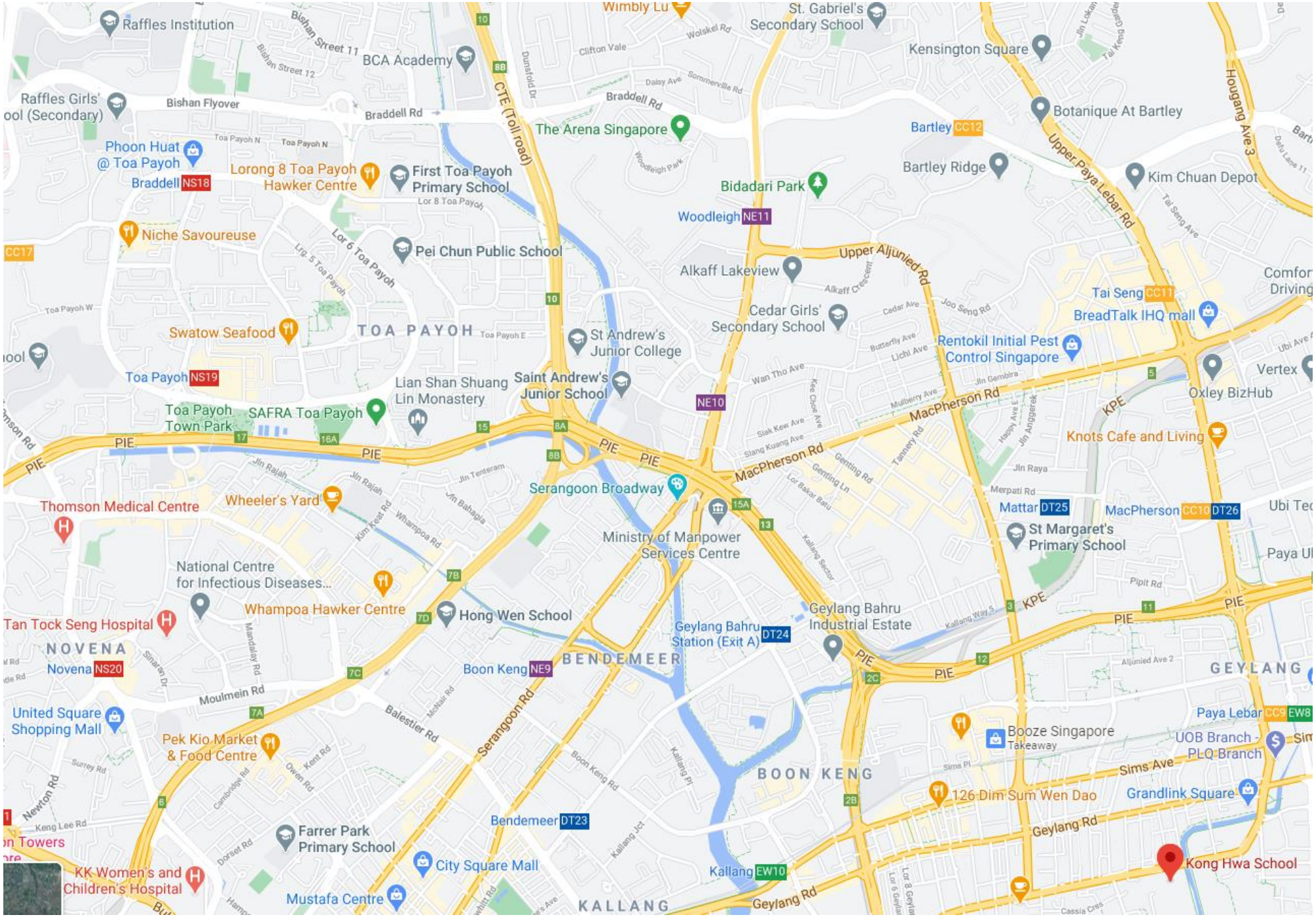
Why Graphs Are So Useful

- World is full of networks based on relationship
  - Computer networks
  - Transportation networks
  - Financial networks
  - Sewer or water networks
  - Political networks
  - Criminal networks
  - Social networks
  - etc

Why Graphs Are So Useful

- We will see that not only do graphs capture relationships in connected networks of elements, they also support inference on those structures
  - Finding sequences of links between elements – is there a path from A to B
  - Finding the least expensive path between elements (aka shortest path problem)
  - Partitioning the graph into sets of connected elements (aka graph partition problem)
  - Finding the most efficient way to separate sets of connected elements (aka the min-cut/max-flow problem)

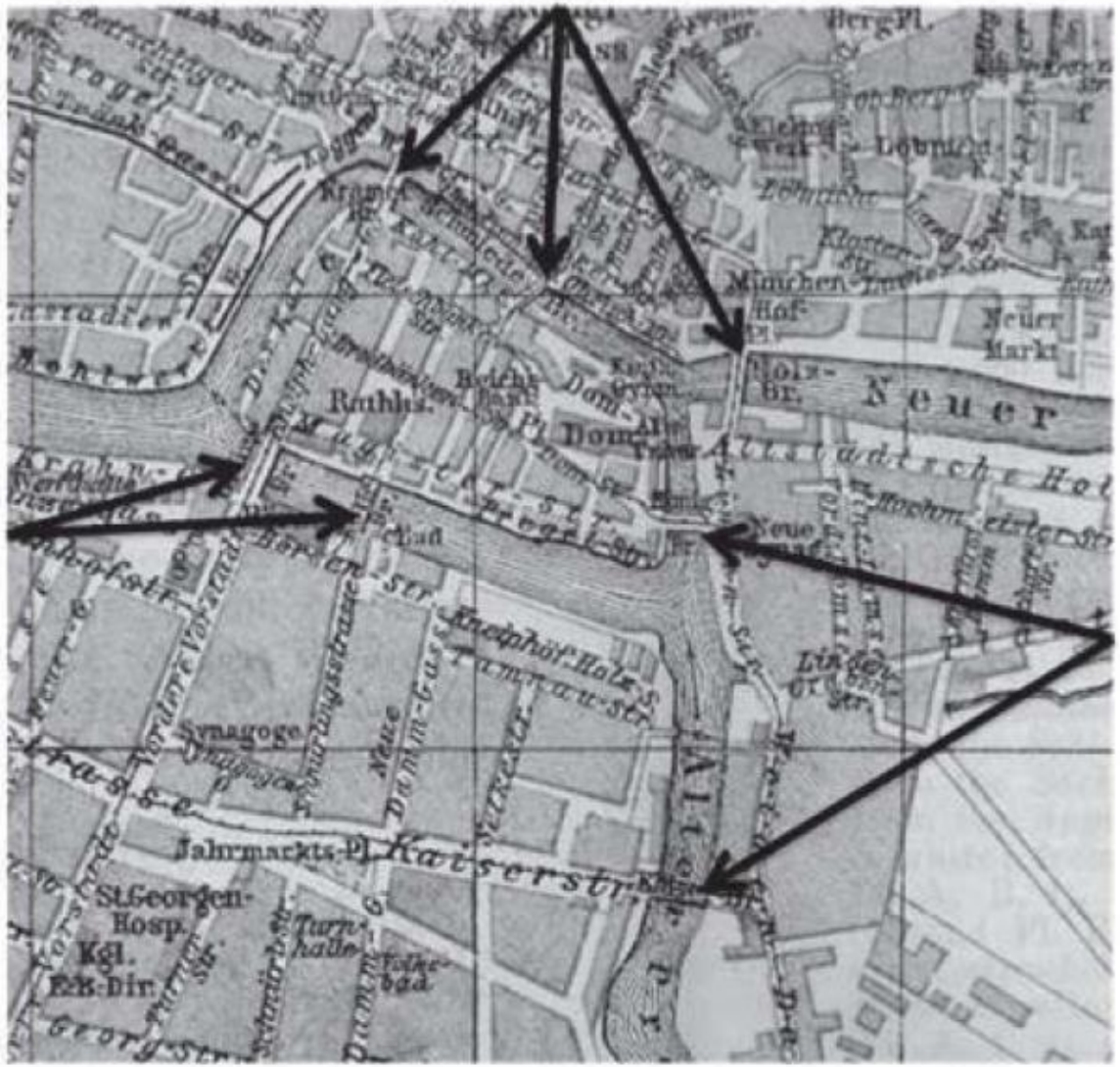
Graph Theory Saves Me Time Every Day


Raffles Institution
Bishan Street 11
Bishan Street 12
BCA Academy
Wimbly Lu
St. Gabriel's Secondary School
Kensington Square
Raffles Girls' ool (Secondary)
Bishan Flyover
Braddell Rd
CTE (Toll road)
Clifton Vale
Wolskel Rd
Dunsfold Dr
Daisy Ave
Sommerville Rd
Botanique At Bartley
Hougang Ave 3
Phoon Huat @ Toa Payoh
Braddell NS18
Toa Payoh N
Lorong 8 Toa Payoh Hawker Centre
First Toa Payoh Primary School
Lor 8 Toa Payoh
The Arena Singapore
Woodleigh Park
Bartley CC12
Bartley Ridge
Upper Paya Lebar Rd
Kim Chuan Depot
Tai Seng Ave
Bidadari Park
Woodleigh NE11
Niche Savoureuse
CC17
Lor 6 Toa Payoh
Lrg 5 Toa Payoh
Pei Chun Public School
Upper Aljunied Rd
Alkaff Lakeview
Alkaff Crescent
Cedar Girls' Secondary School
Cedar Ave
Joo Seng Rd
Tai Seng CC11
BreadTalk IHQ mall
Comfor Driving
Toa Payoh W
Swatow Seafood
TOA PAYOH
Toa Payoh E
St Andrew's Junior College
Rentokil Initial Pest Control Singapore
Butterfly Ave
Lichi Ave
Toa Payoh NS19
Lian Shan Shuang Lin Monastery
Saint Andrew's Junior School
Wan Tho Ave
Kee Chow Ave
Jln Gembira
Vertex
Oxley BizHub
NE10
Toa Payoh Town Park
SAFRA Toa Payoh
MacPherson Rd
KPE
PIE
Siak Kew Ave
Siang Kuang Ave
Tannery Rd
Happy Ave
Jln Anggerak
Knots Cafe and Living
Jln Raja
Jln Tenteram
Jln Bahagia
Serangoon Broadway
Jln Raya
Merpati Rd
Thomson Medical Centre
Wheeler's Yard
Kim Keat Rd
Whampoa Rd
Mattar DT25
MacPherson CC10 DT26
Ubi Tec
Ministry of Manpower Services Centre
St Margaret's Primary School
Paya Ub
National Centre for Infectious Diseases...
Whampoa Hawker Centre
Hong Wen School
Pipit Rd
Geylang Bahru Industrial Estate
Tan Tock Seng Hospital
Geylang Bahru Station (Exit A) DT24
NOVENA
BENDEMEER
GEYLANG
Novena NS20
Boon Keng NE9
Moulmein Rd
Aljunied Ave 2
United Square Shopping Mall
Balestier Rd
Serangoon Rd
Paya Lebar CC9 EW8
Booze Singapore Takeaway
UOB Branch - PLQ Branch
Pek Kio Market & Food Centre
Sims Ave
Sims Pl
BOON KENG
126 Dim Sum Wen Dao
Grandlink Square
Newton Rd
Keng Lee Rd
Boon Keng Rd
Kallang Pl
Bendemeer DT23
Geylang Rd
Farrer Park Primary School
Kong Hwa School
KK Women's and Children's Hospital
City Square Mall
Kallang EW10
Mustafa Centre
KALLANG
Kallang Jct
Cassia Cres

- Model road system using a digraph
  - Nodes: points where roads end or meet
  - Edges: connections between points
    - Each edge has a weight
      - Expected time to get from source node to destination node for that edge
      - Distance between source and destination nodes
      - Average speed of travel between source and destination nodes
- Solve a graph optimization problem
  - Shortest weighted path between my house and my office

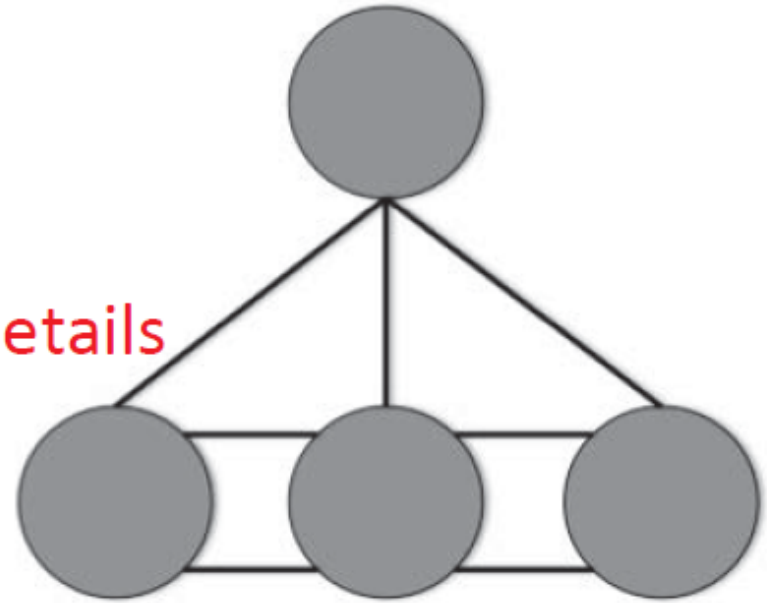
First Reported Use of Graph Theory

- Bridges of Königsberg (1735)
- Possible to take a walk that traverses each of the 7 bridges exactly once?



Leonhard Euler's Model

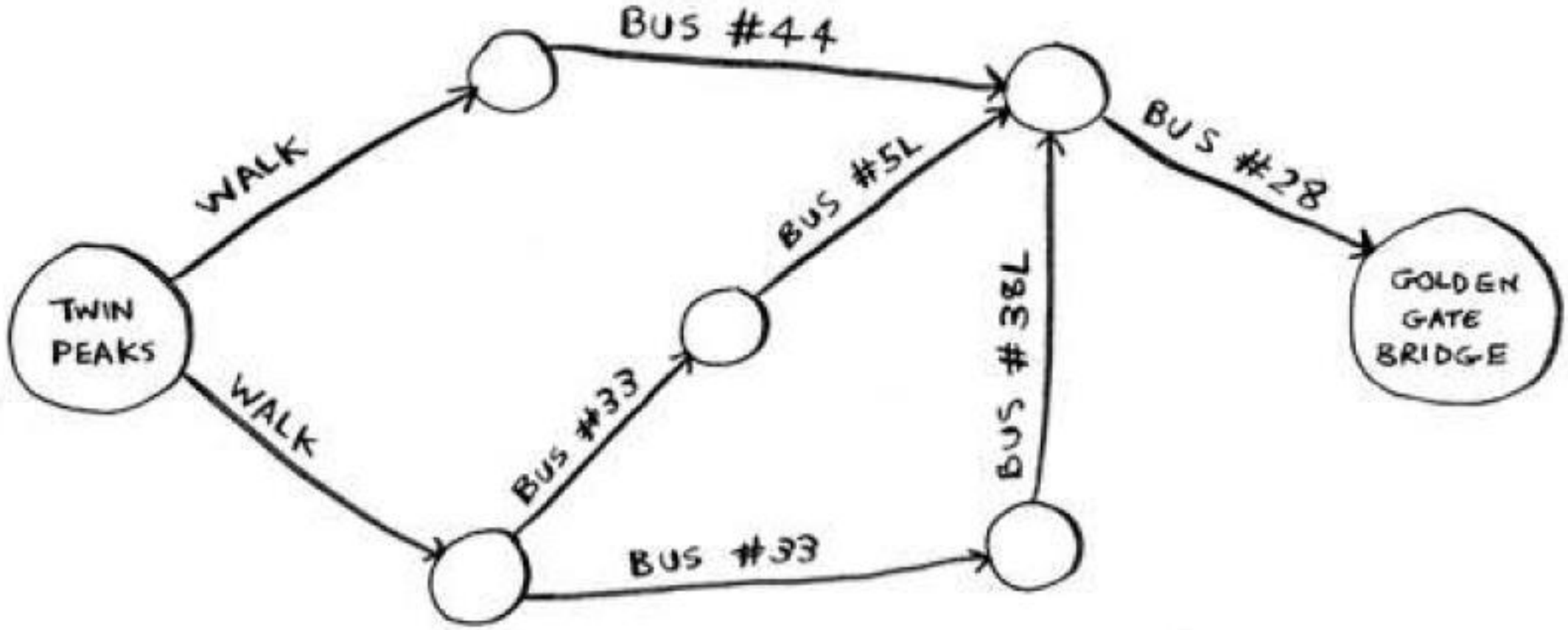
- Each island a node
- Each bridge an undirected edge
- Model abstracts away irrelevant details
  - Size of islands
  - Length of bridges



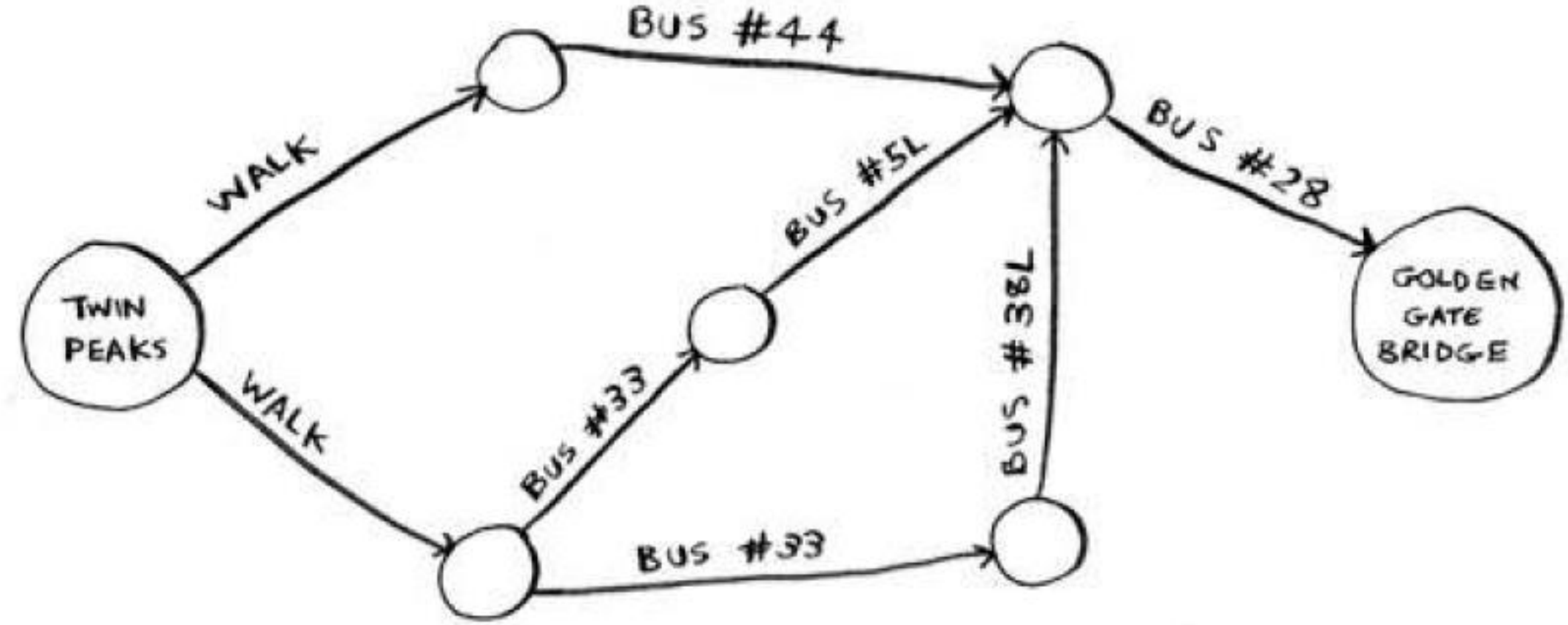
- Is there a path that contains each edge exactly once?
  - **No!**

https://en.wikipedia.org/wiki/Seven_Bridges_of_K%C3%B6nigsberg

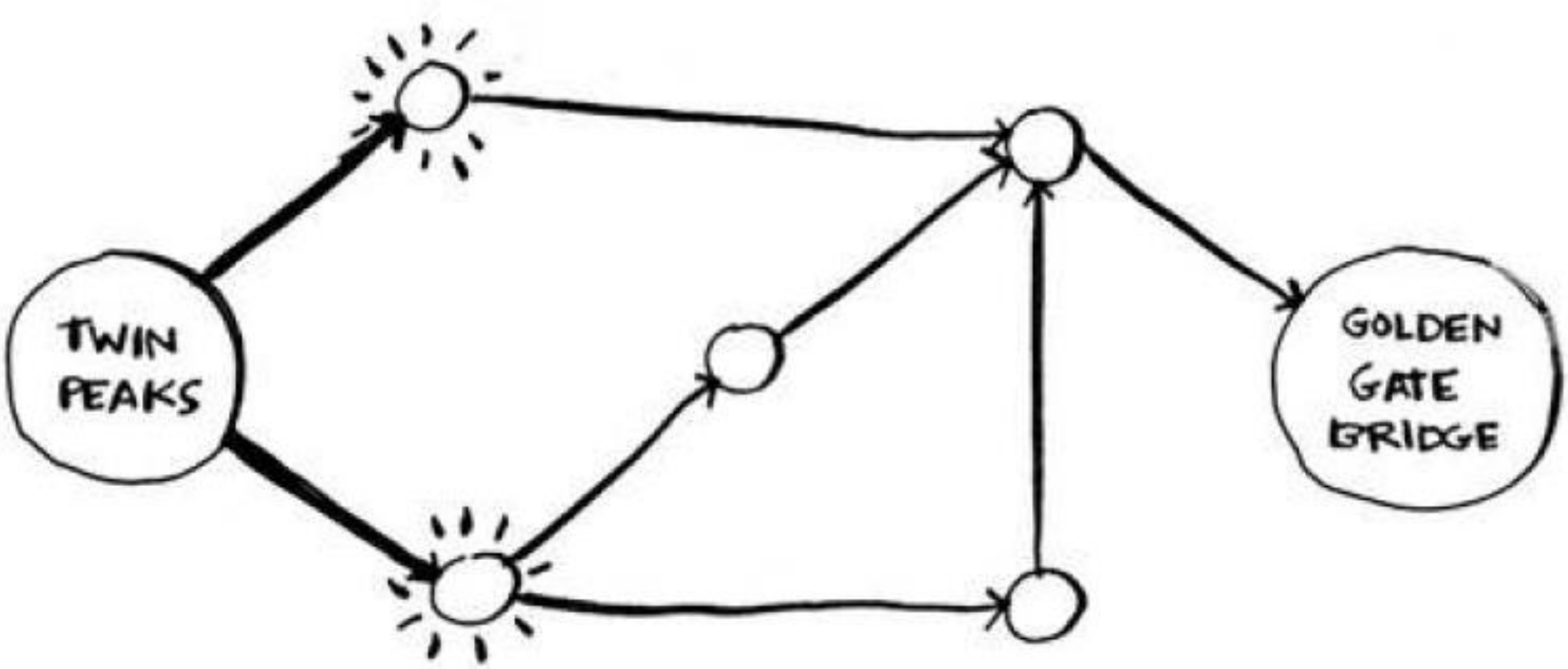
Suppose you want to travel to Golden Gate Bridge from Twin Peaks.



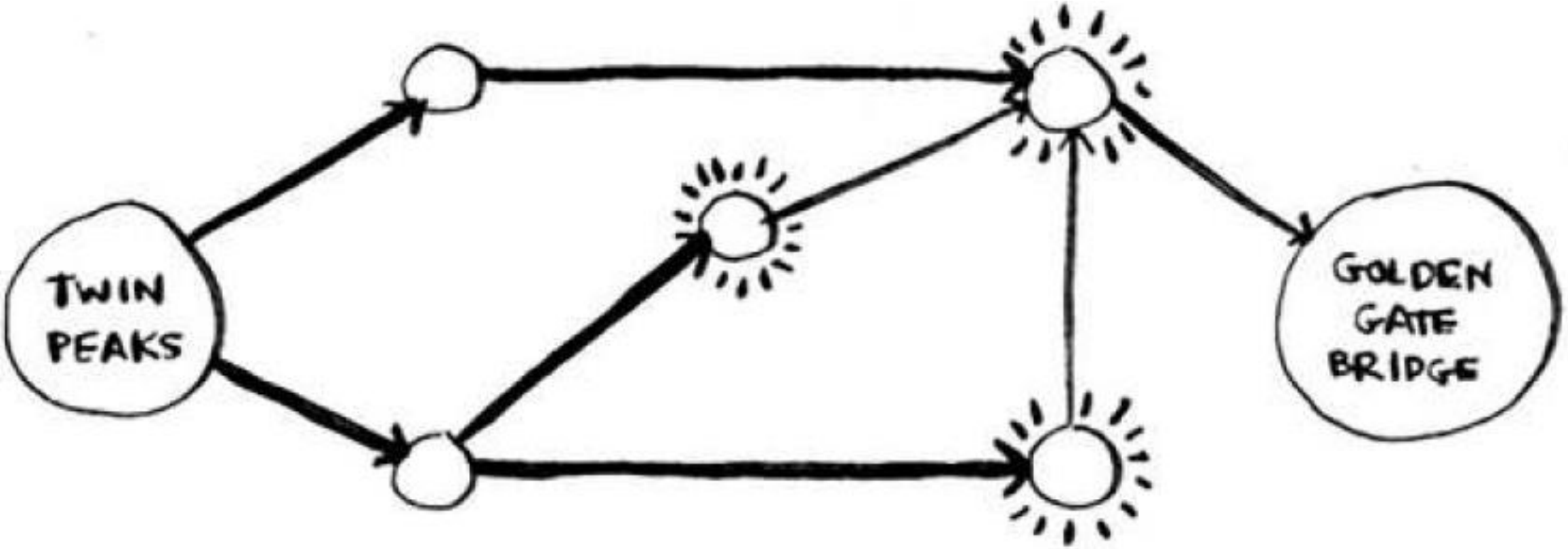
What's your algorithm to find the path with the fewest steps?



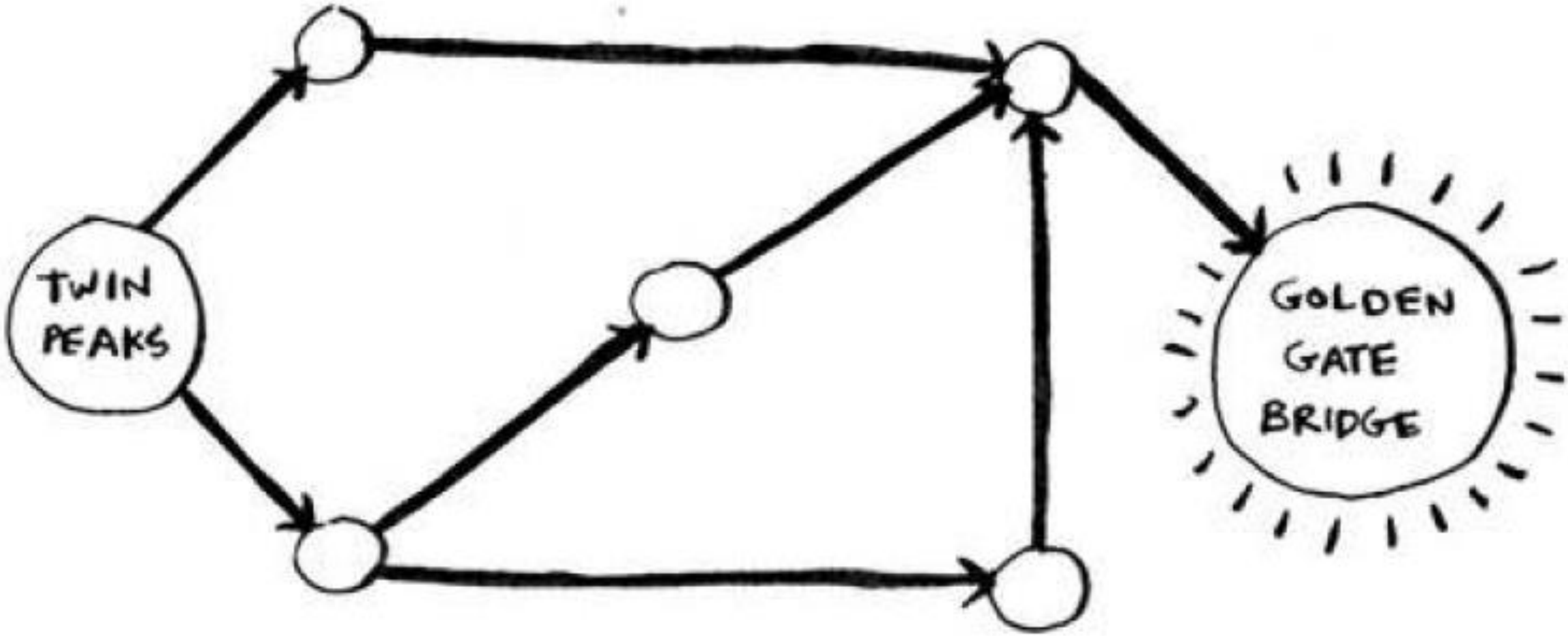
can you get there in one step?



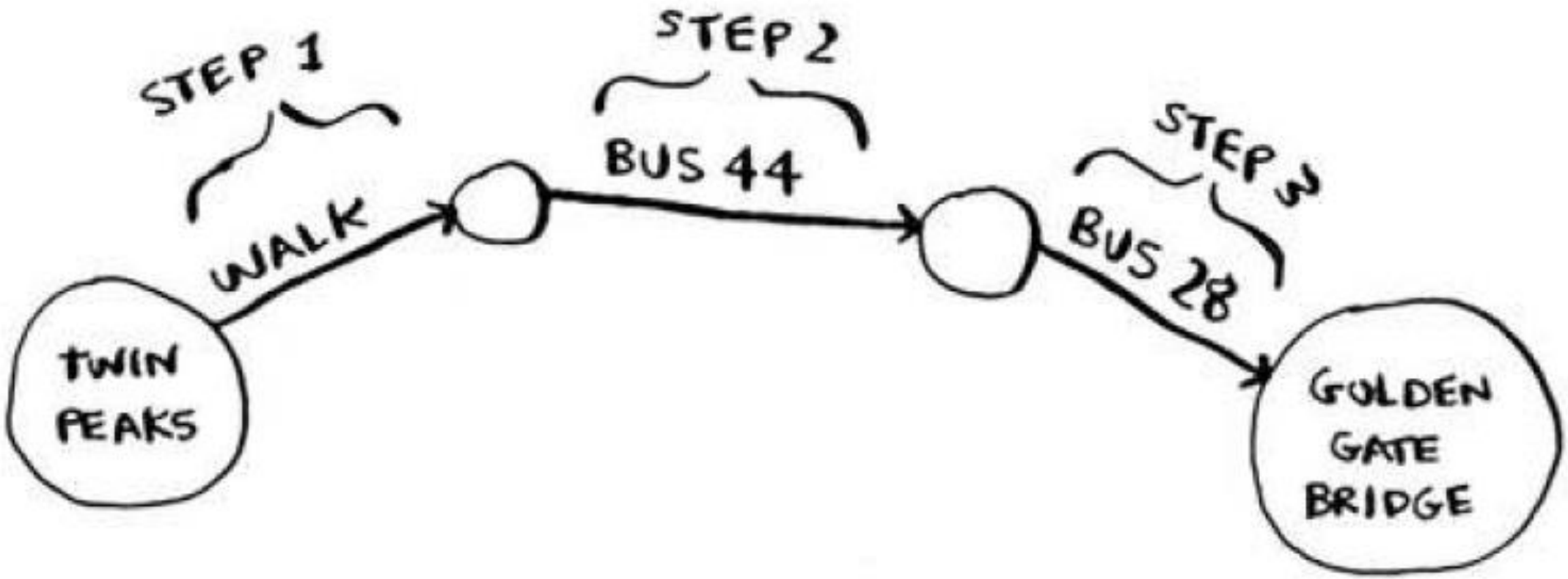
if no, can you get there in two steps?



if no, can you get there in three steps?



it takes three steps to get there.



summary.

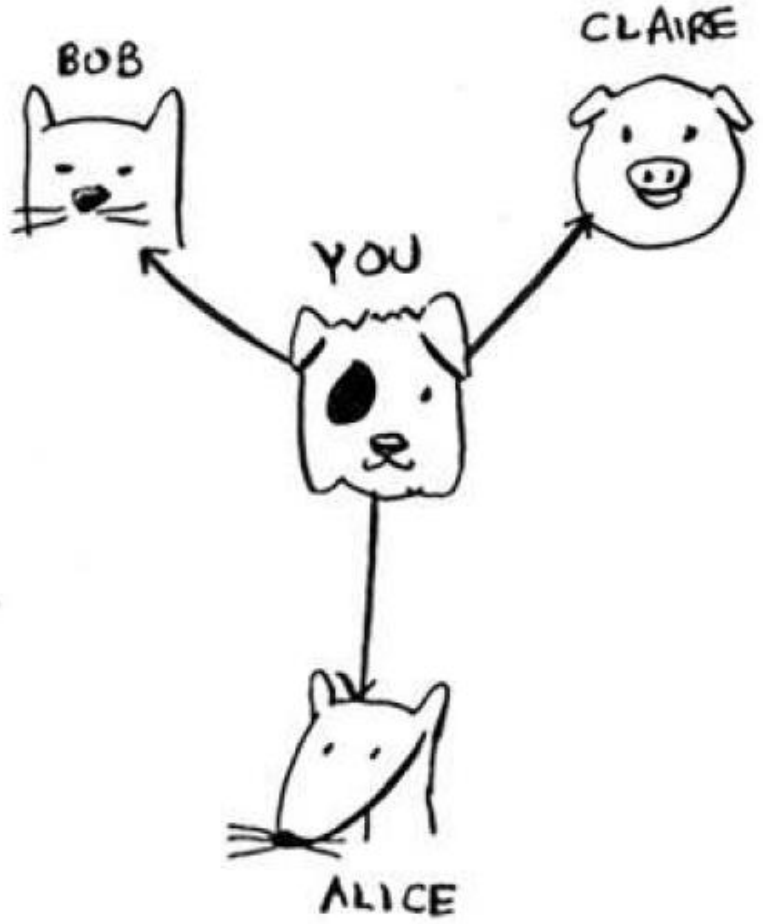
- there are other routes that will get you there
- they are at least three steps long
- the algorithm found that the shortest route to the bridge is three steps long
- this type of problem is called a shortest-path problem

Breadth – first search

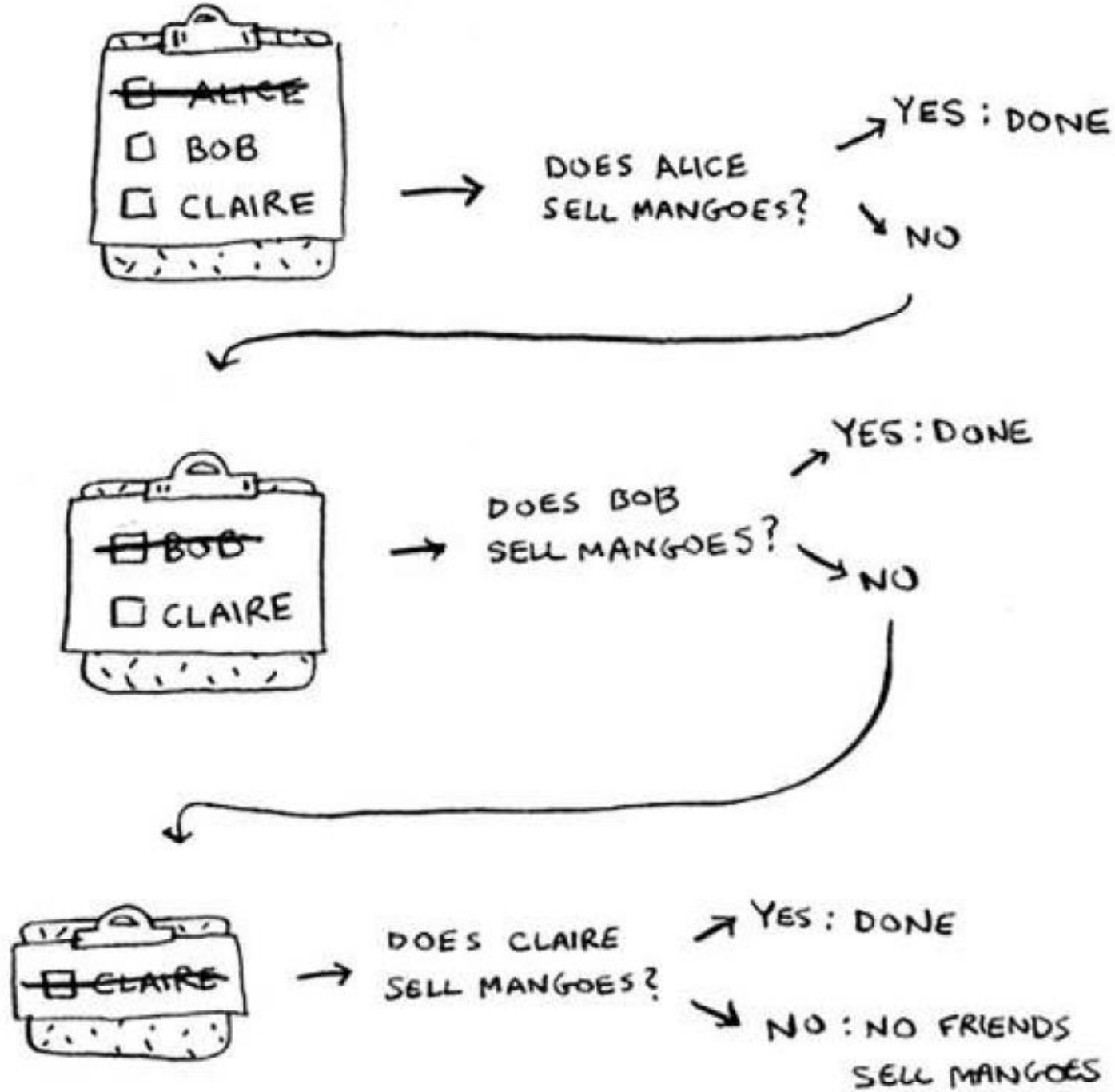
- we have discussed linear search and binary search
- BFS is a different kind of search algorithm: one that runs on graphs
- it can help answer two types of questions:
  1. Is there a path from node A to node B?
  2. What is the shortest path from node A to node B?

Suppose you're the proud owner of a mango farm. Looking for mango seller to sell yours.

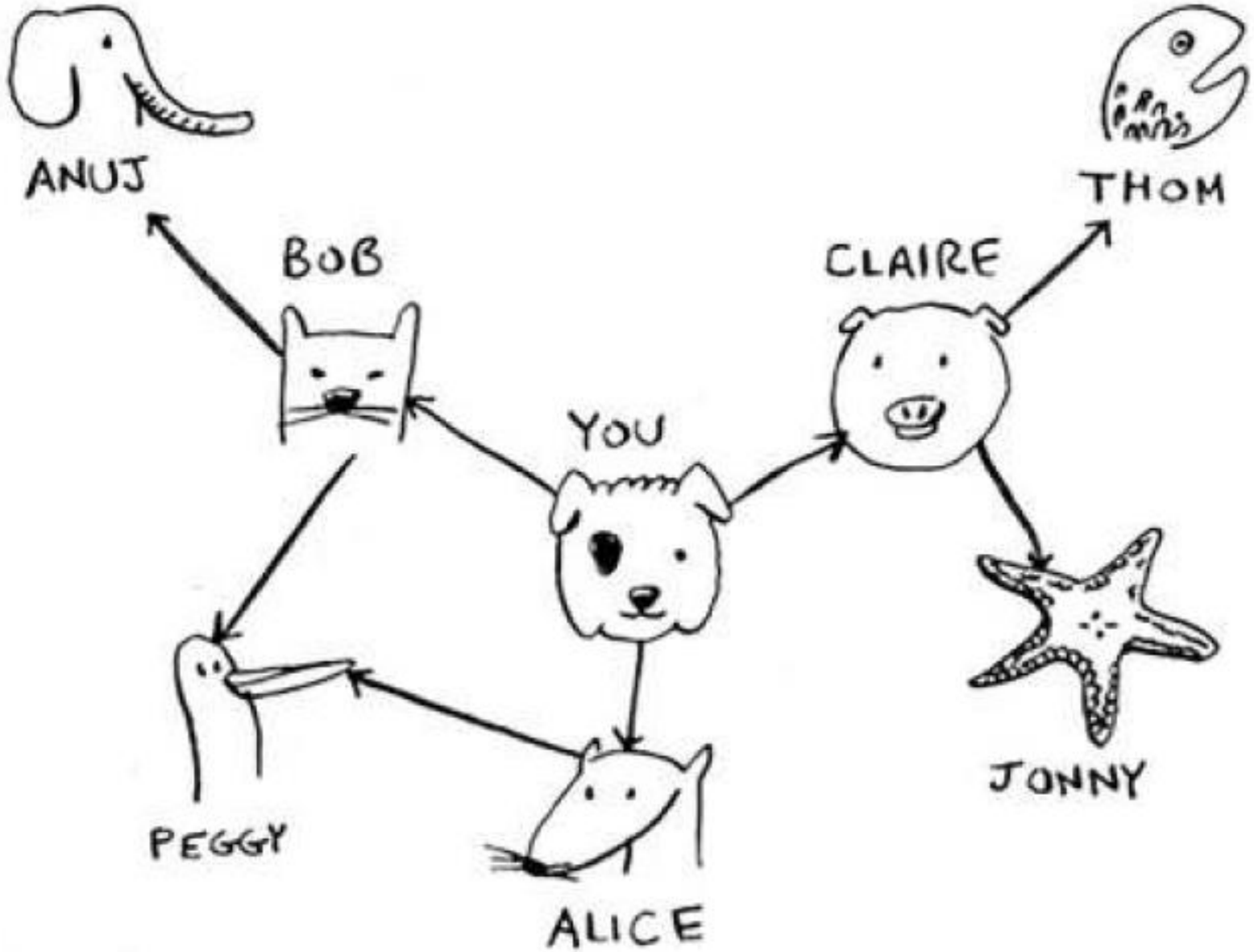
- this search is pretty straightforward
- first, make a list of friends to search



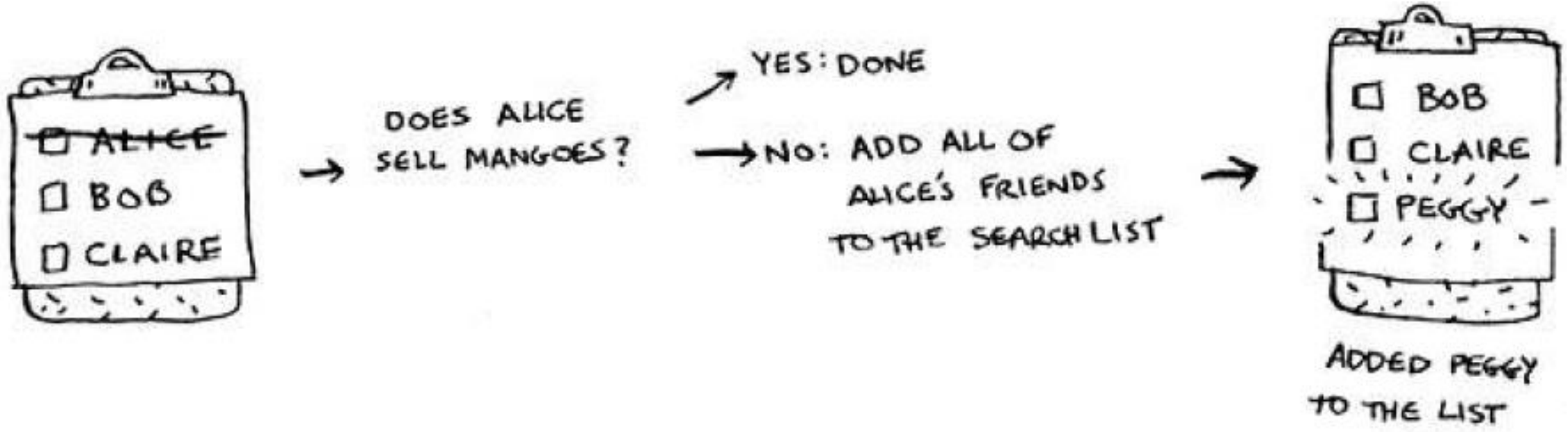
now go to each person in the list and check whether that person sells mangoes.


ALICE
BOB
CLAIRE
DOES ALICE SELL MANGOES?
YES : DONE
NO
BOB
CLAIRE
DOES BOB SELL MANGOES?
YES : DONE
NO
CLAIRE
DOES CLAIRE SELL MANGOES?
YES : DONE
NO : NO FRIENDS SELL MANGOES

suppose none of your friends are mango sellers. now you have to search through your friends' friends.


ANUJ
BOB
YOU
CLAIRE
THOM
PEGGY
ALICE
JONNY

each time you search for someone from the list, add all of their friends to the list.



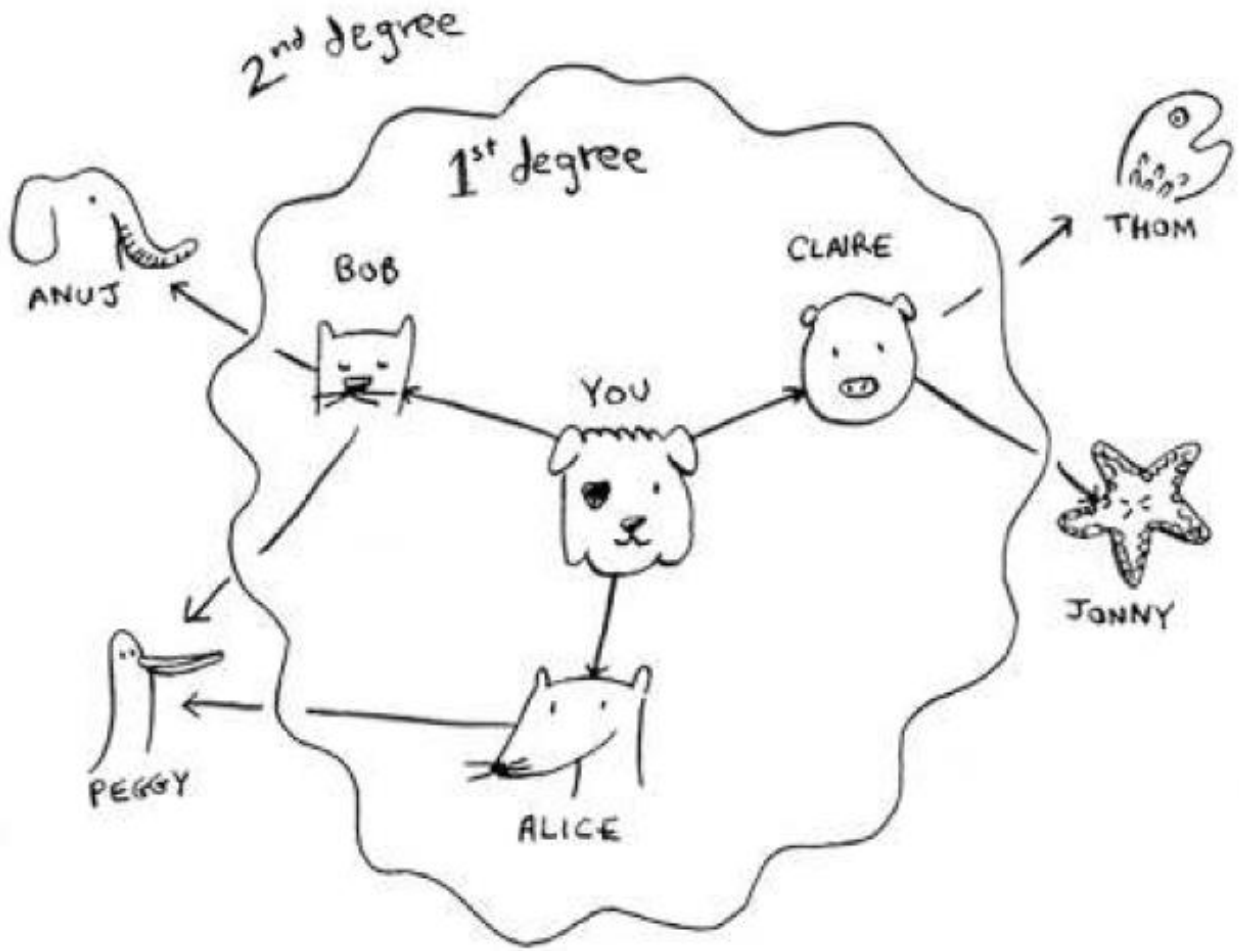
- you not only search your friends, but you search their friends too
- the goal is to find one mago seller in your network
- if Alice isn't a mango seller, you add her friends to the list, too
- that means you'll eventually search her friends – and then their friends
- with this algorithm, you'll search your entire network until you come across a mango seller. This algorithm is breadth-first search.

relook.

- Question 1: is there a path from node A to node B?
  - Is there a mango seller in your network?
- Question 2: what is the shortest path from node A to node B?
  - Who is the closest mango seller?

look deeper to question 2.

- can you find the closest mango seller?



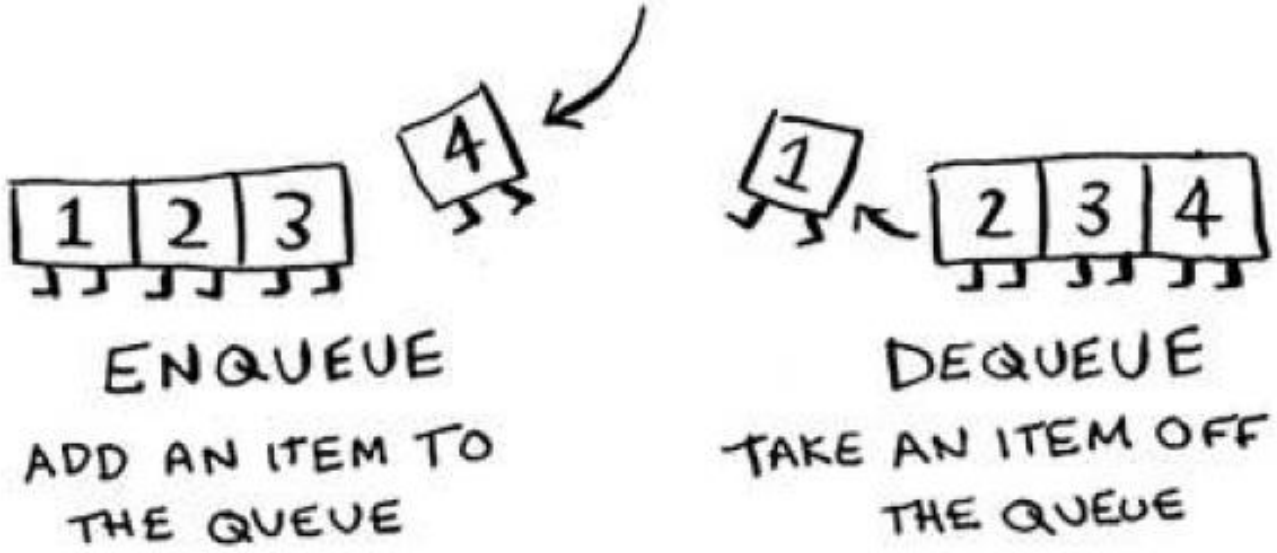
look deeper.

- we will prefer a first degree connection to a second degree connection, and so on
- we shouldn't search any second degree connections before we make sure we don't have a first degree connection who is a mango seller.
- breadth first search already does this!
  - the way breadth-first search works, the search radiates out from the starting point
  - notice that this only works if we search people in the same order in which they are added.
  - we need to search people in the order that they are added.
    - there's a data structure for this: it's called a **queue**

Queues

- same concept as real life
- suppose you and your friend are queuing up at the bus stop. if you are before him in the queue, you get on the bus first.
- two operations in queue
  - enqueue
  - dequeue

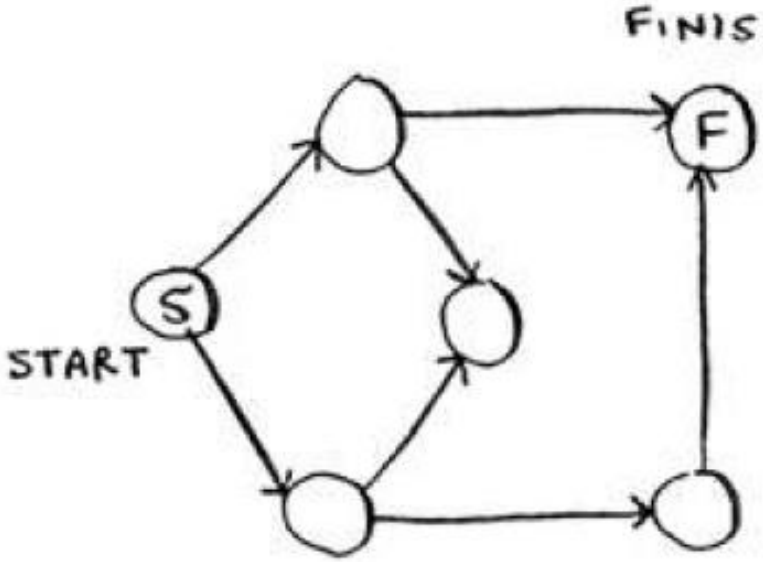
Queues



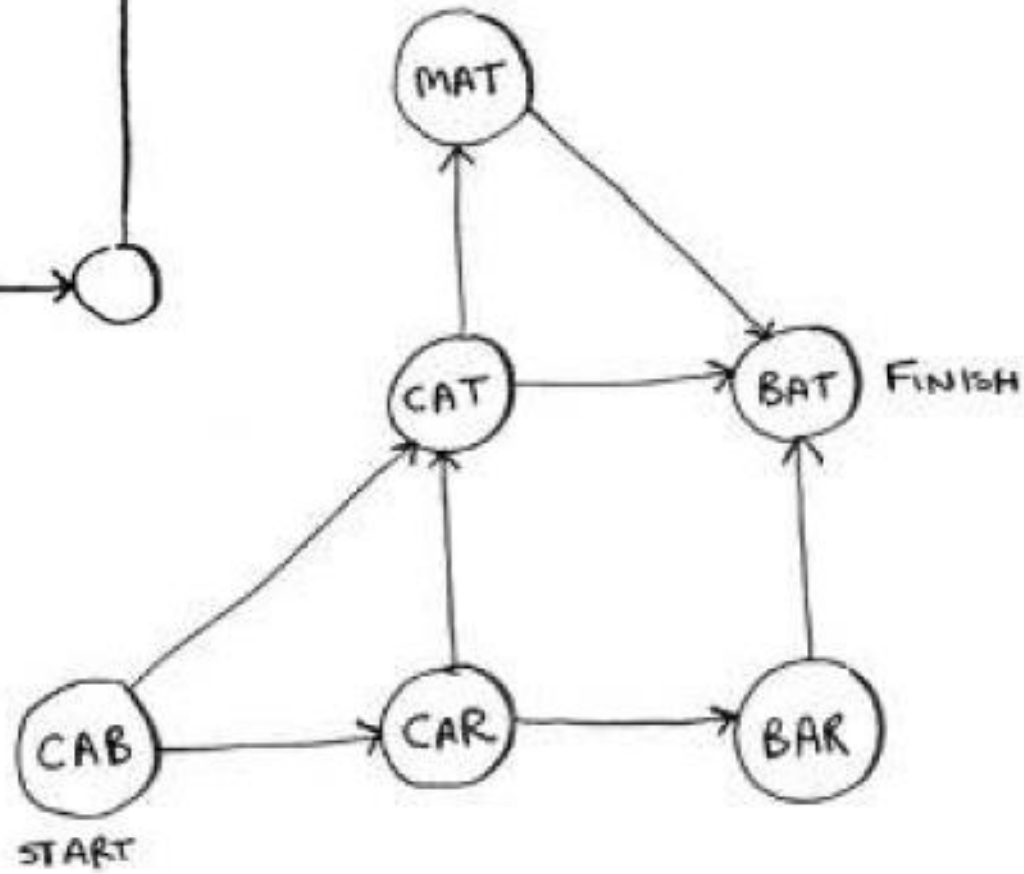
- if we enqueue two items to the list, the first item we add will be dequeued before the second item
- we can use this for our search list
- the queue is called a FIFO data structure: First in, First Out.

let’s try this simple graph.

- Find the length of the shortest path from start to finish.

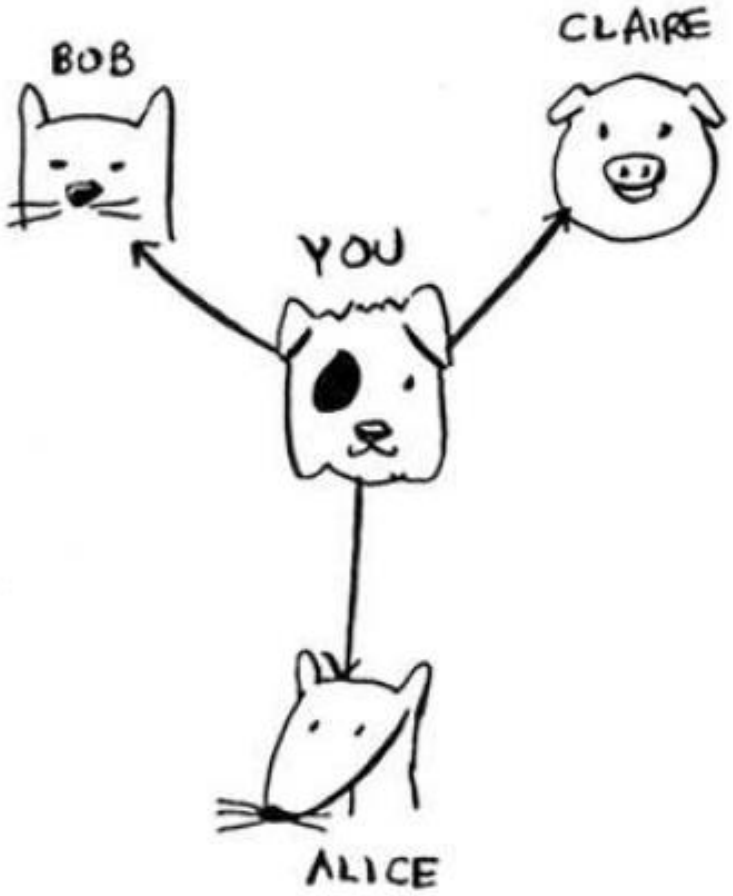


- Find the length of the shortest path from “cab” to “bat”

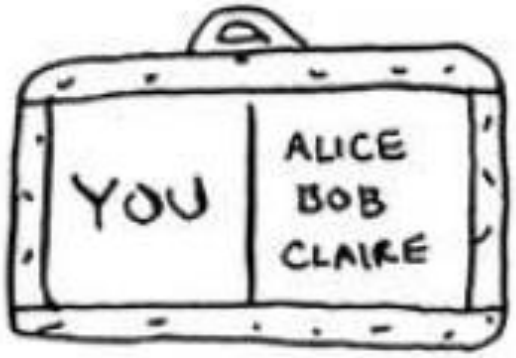


Implementing the graph.

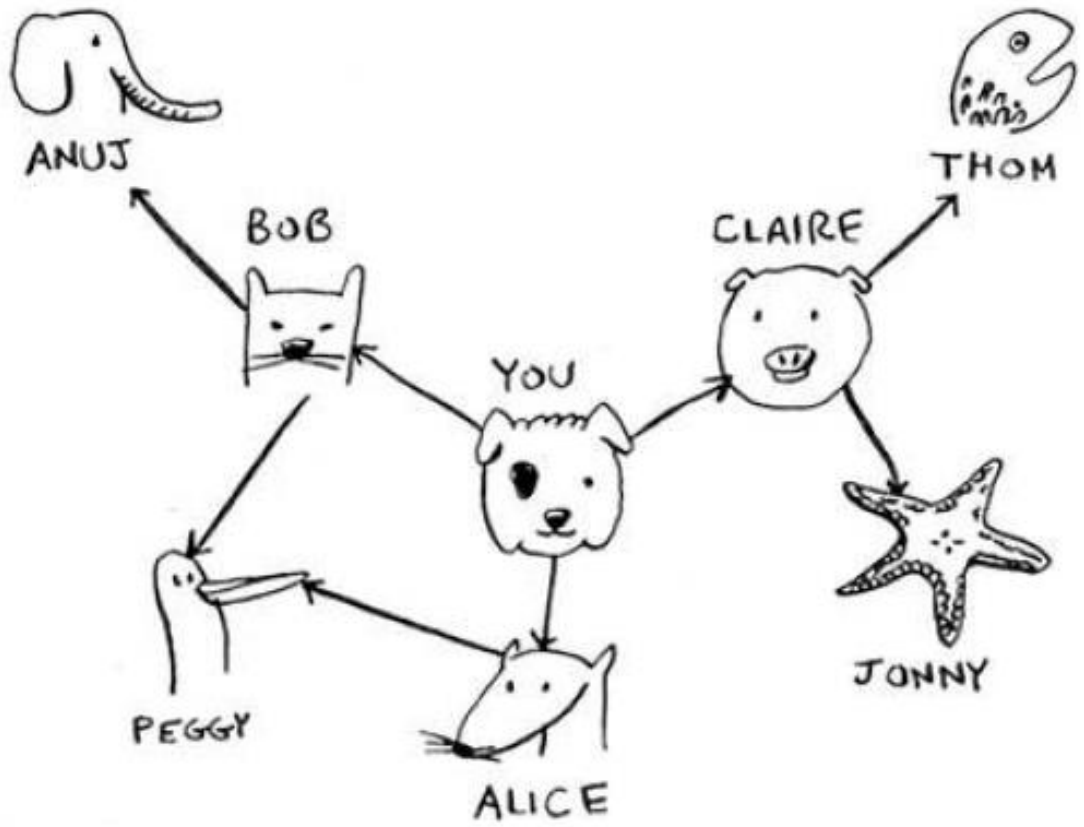
- use a dictionary
  - key, value pair
- we want to map a node to all of its neighbours




```
graph = {}
graph["you"] = ["alice", "bob", "claire"]
```



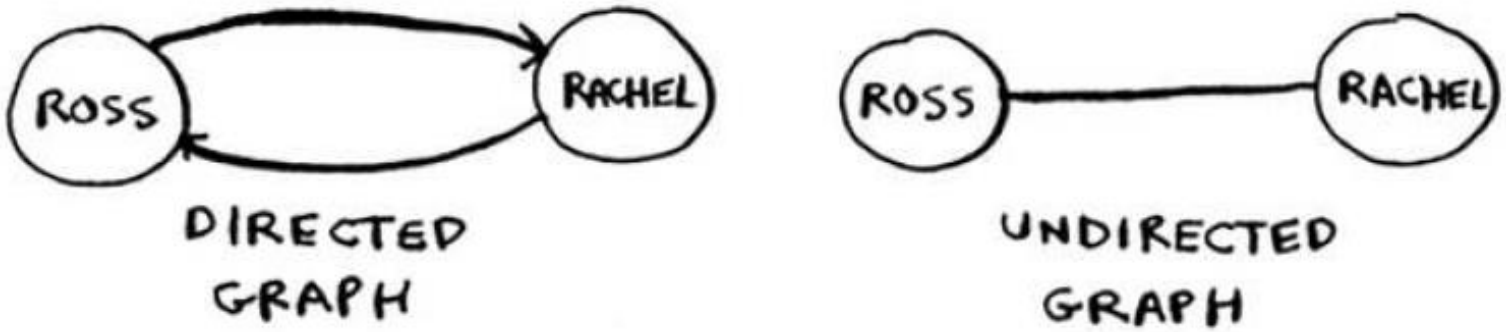
a bigger graph.



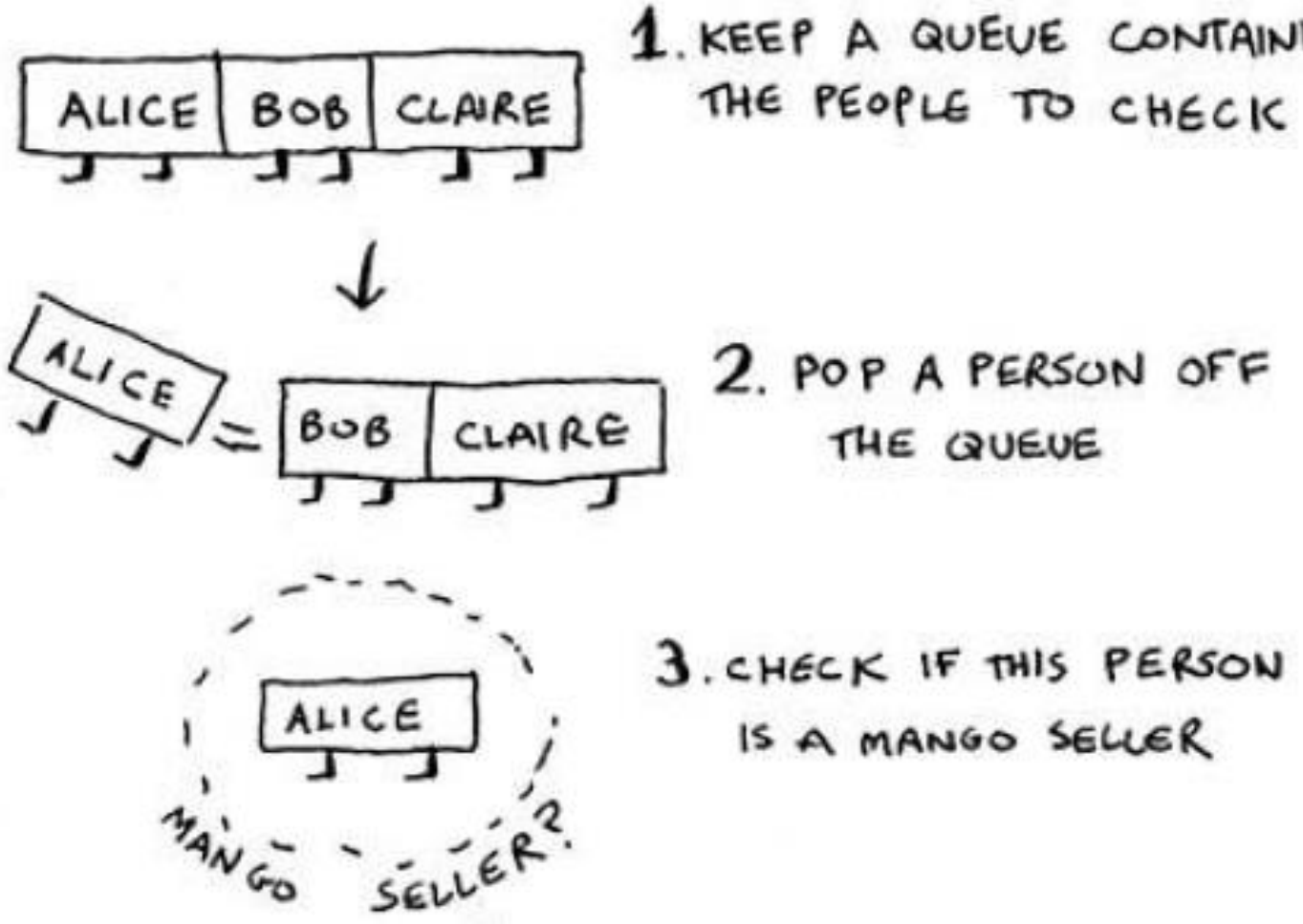

```
graph = {}
graph[“you”] = [“alice”, “bob”, “claire”]
graph[“bob”] = [“anuj”, “peggy”]
graph[“alice”] = [“peggy”]
graph[“claire”] = [“thom”, “jonny”]
graph[“anuj”] = []
graph[“peggy”] = []
graph[“thom”] = []
graph[“jonny”] = []
```

directed graph vs undirected graph

- Anuj, Peggy, Thom and Jonny don't have any neighbors.
- they have arrows pointing to them, but no arrows from them to someone else
- this is called a directed graph
- an undirected graph doesn't have any arrows, and both nodes are each other's neighbours.
  - For eg, both of these graphs are equal

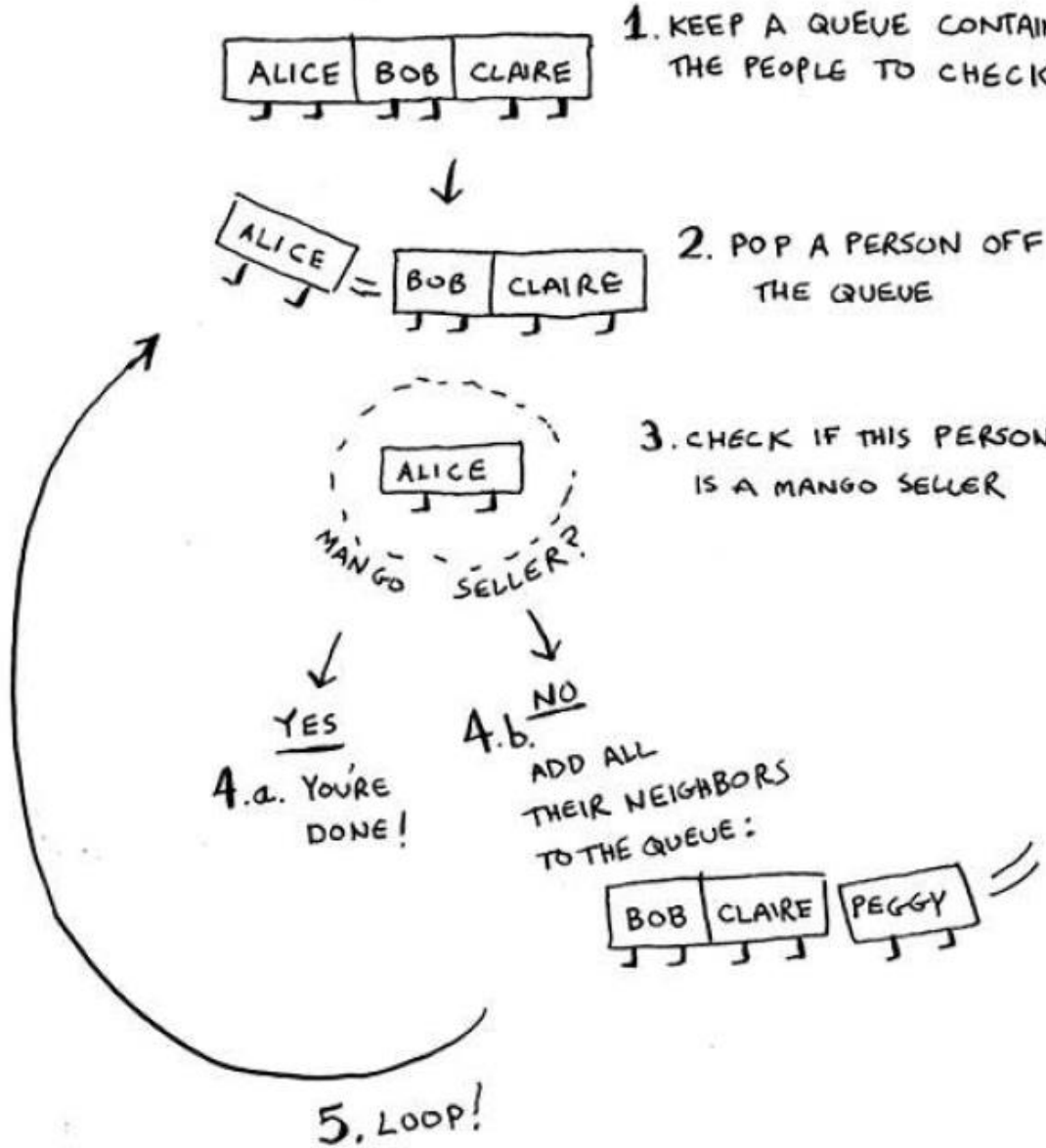


implementing the algorithm



Note

When updating queues, I use the terms *enqueue* and *dequeue*. You'll also encounter the terms *push* and *pop*. *Push* is almost always the same thing as *enqueue*, and *pop* is almost always the same thing as *dequeue*.



6. IF THE QUEUE IS EMPTY, THERE ARE NO MANGO SELLERS IN YOUR NETWORK

implementation.

- make a queue to start. in python, use the double ended queue (deque) function for this:

```
from collections import deque
search_queue = deque()          <-- Creates a new queue
search_queue += graph["you"]    <-- Adds all of your neighbors to the search queue
```

- remember, graph[“you”] will give you a list of all your neighbours, like [“alice’, “bob”, “Claire”]. those all get added to the search queue.

implementation.

```
while search_queue:
    person = search_queue.popleft()
    if person_is_seller(person):
        print person + " is a mango seller!"
        return True
    else:
        search_queue += graph[person]
return False
```

While the queue isn't empty ...

... grabs the first person off the queue

Checks whether the person is a mango seller

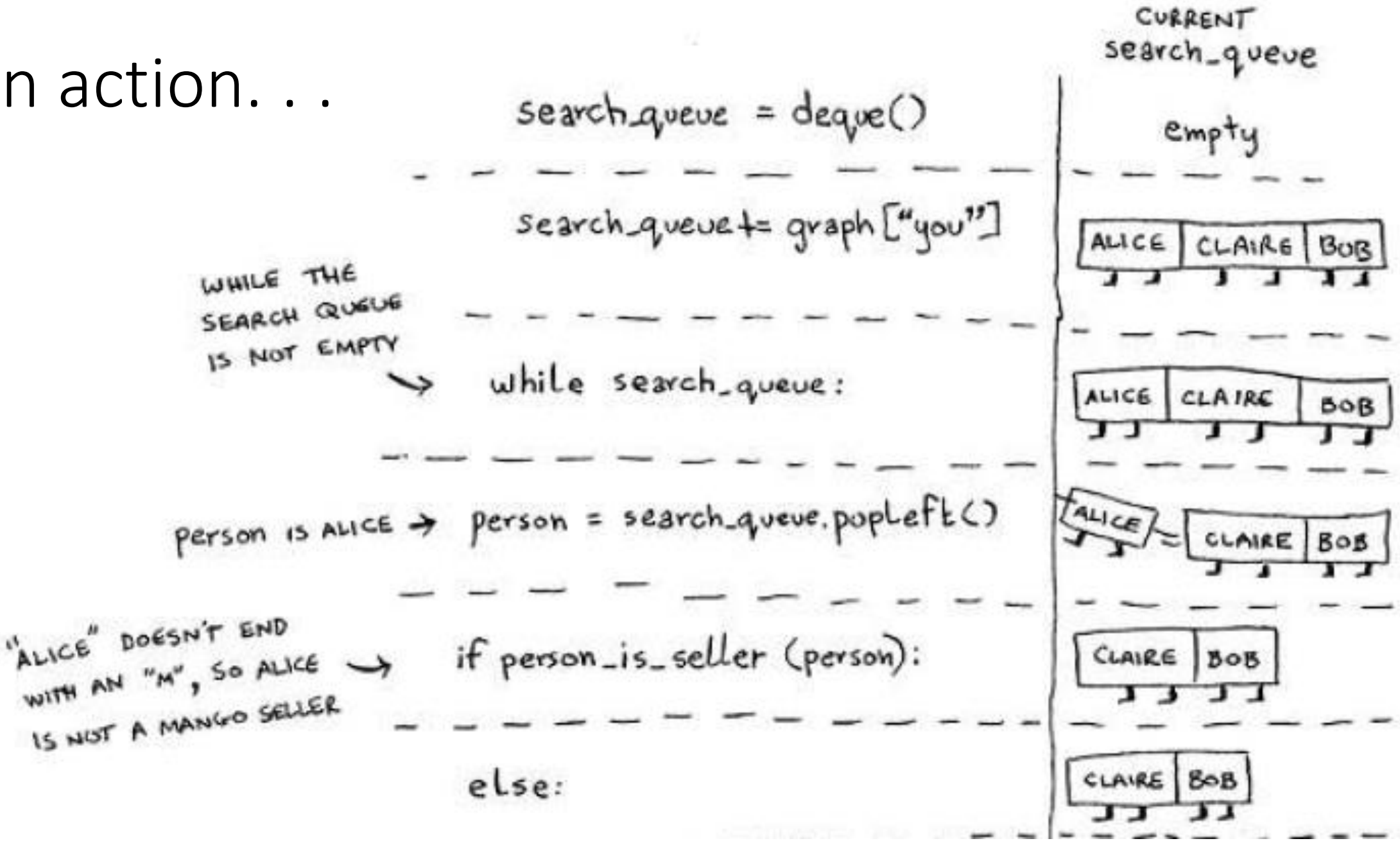
Yes, they're a mango seller.

No, they aren't. Add all of this person's friends to the search queue.

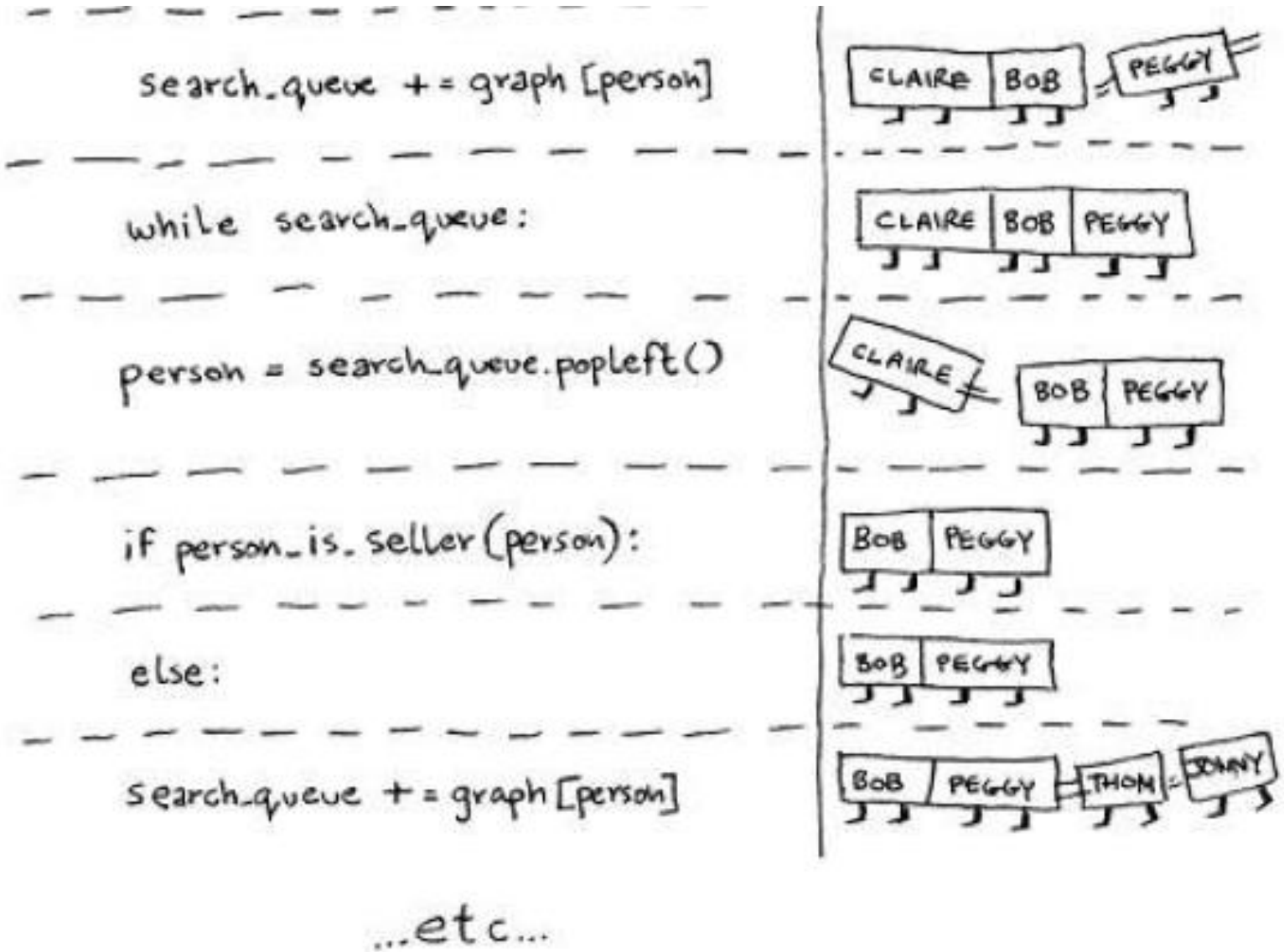
If you reached here, no one in the queue was a mango seller.

```
def person_is_seller(name):
    return name[-1] == 'm'
```

in action. . .


CURRENT
search_queue
search_queue = deque()
empty
search_queue += graph["you"]
ALICE
CLAIRE
BOB
WHILE THE SEARCH QUEUE IS NOT EMPTY
while search_queue:
ALICE
CLAIRE
BOB
person IS ALICE
person = search_queue.popleft()
ALICE
CLAIRE
BOB
"ALICE" DOESN'T END WITH AN "M", SO ALICE IS NOT A MANGO SELLER
if person_is_seller(person):
CLAIRE
BOB
else:
CLAIRE
BOB

in action . . .

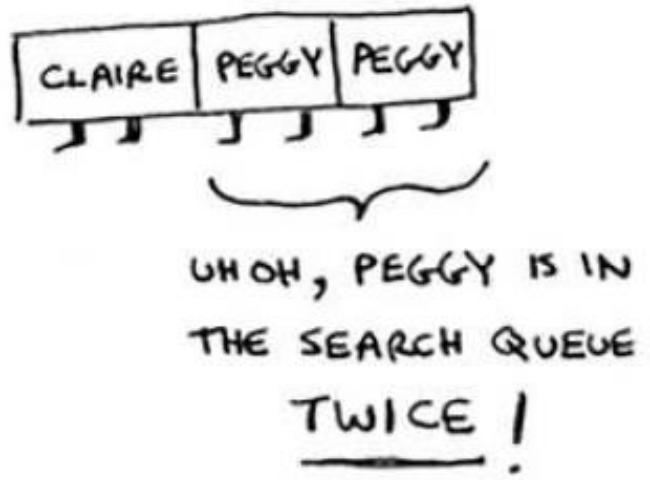

search_queue += graph[person]
CLAIRE
BOB
PEGGY
while search_queue:
CLAIRE
BOB
PEGGY
person = search_queue.popleft()
CLAIRE
BOB
PEGGY
if person_is_seller(person):
BOB
PEGGY
else:
BOB
PEGGY
search_queue += graph[person]
BOB
PEGGY
THOM
JONNY
...etc...

when will it ends.

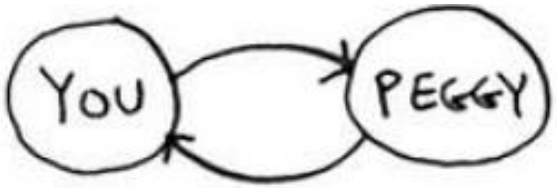
- the algorithm will keep going until either:
  - a mango seller is found, or
  - the queue becomes empty, in which case there is no mango seller.

potential problem.

- alice and bob share a friend Peggy. So Peggy will be added to the queue twice:
  - once when you add Alice's friends and
  - again when you add Bob's friends
- if you check her twice, you're doing unnecessary, extra work. So once you search a person, you should mark that person as searched and not search them again
- if you don't do this, you could end up in an infinite loop

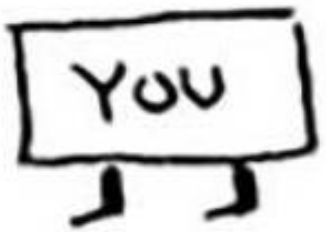


Suppose the mango seller graph looked like this



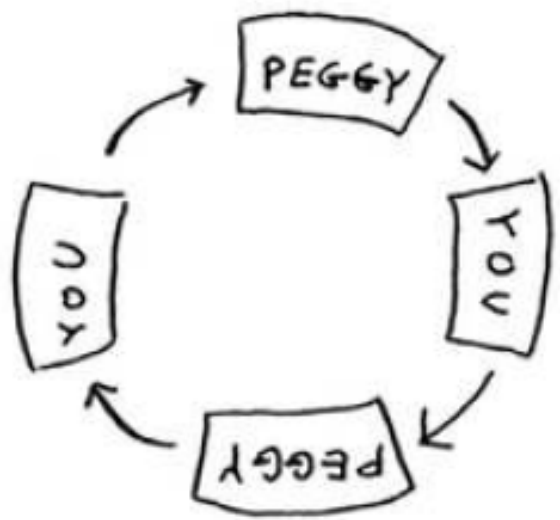
To start, the search queue contains all of your neighbors.

Now you check Peggy. She isn't a mango seller, so you add all of her neighbors to the search queue.



Next, you check yourself. You're not a mango seller, so you add all of your neighbors to the search queue.

And so on. This will be an infinite loop, because the search queue will keep going from you to Peggy.



final code.

```
def search(name):
    search_queue = deque()
    search_queue += graph[name]
    searched = []
    while search_queue:
        person = search_queue.popleft()
        if not person in searched:
            if person_is_seller(person):
                print person + " is a mango seller!"
                return True
            else:
                search_queue += graph[person]
                searched.append(person)
    return False
```

This array is how you keep track of which people you've searched before.

Only search this person if you haven't already searched them.

Marks this person as searched

Implementing and using graphs

- Building graphs
  - Nodes
  - Edges
  - Stitching together to make graphs
- Using graphs
  - Searching for paths between nodes
  - Searching for optimal paths between nodes

Class Node

```
class Node(object):
    def __init__(self, name):
        """Assumes name is a string"""
        self.name = name
    def getName(self):
        return self.name
    def __str__(self):
        return self.name
```

Class Edge

```
class Edge(object):
    def __init__(self, src, dest):
        """Assumes src and dest are nodes"""
        self.src = src
        self.dest = dest
    def getSource(self):
        return self.src
    def getDestination(self):
        return self.dest
    def __str__(self):
        return self.src.getName() + '->'\
               + self.dest.getName()
```

Common Representations of Digraphs

- Digraph is a directed graph
  - Edges pass in one direction only
- Adjacency matrix
  - Rows: source nodes
  - Columns: destination nodes
  - Cell[s, d] = 1 if there is an edge from s to d
    = 0 otherwise
  - Note that in digraph, matrix is **not** symmetric
- Adjacency list
  - Associate with each node a list of destination nodes

Class Digraph, part 1

```
class Digraph(object):
    """edges is a dict mapping each node to a list of
    its children"""

    def __init__(self):
        self.edges = {}

    def addNode(self, node):
        if node in self.edges:
            raise ValueError('Duplicate node')
        else:
            self.edges[node] = []

    def addEdge(self, edge):
        src = edge.getSource()
        dest = edge.getDestination()
        if not (src in self.edges and dest in self.edges):
            raise ValueError('Node not in graph')
        self.edges[src].append(dest)
```

Nodes are represented as keys in dictionary

Edges are represented by destinations as values in list associated with a source key

Class Digraph, part 2

```
    def childrenOf(self, node):
        return self.edges[node]

    def hasNode(self, node):
        return node in self.edges

    def getNode(self, name):
        for n in self.edges:
            if n.getName() == name:
                return n
        raise NameError(name)

    def __str__(self):
        result = ''
        for src in self.edges:
            for dest in self.edges[src]:
                result = result + src.getName() + '->'\
                         + dest.getName() + '\n'
        return result[:-1] #omit final newline
```

Class Graph

```
class Graph(Digraph):
    def addEdge(self, edge):
        Digraph.addEdge(self, edge)
        rev = Edge(edge.getDestination(), edge.getSource())
        Digraph.addEdge(self, rev)
```

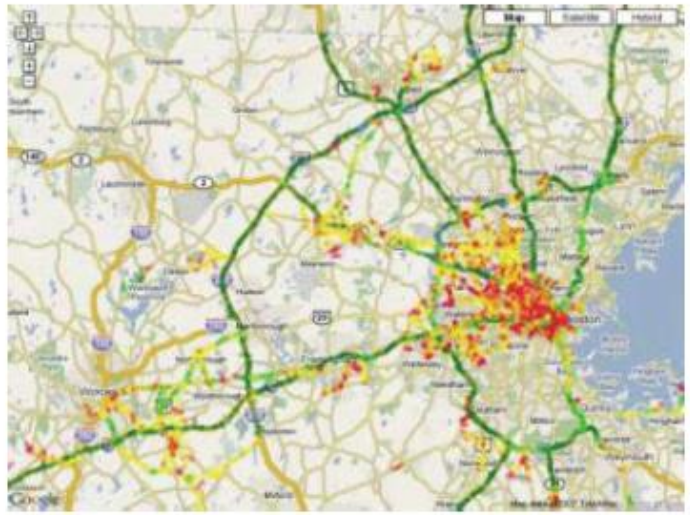
- Graph does not have directionality associated with an edge
  - Edges allow passage in either direction
- Why is Graph a subclass of Digraph?
- Remember the substitution rule?
  - If client code works correctly using an instance of the supertype, it should also work correctly when an instance of the subtype is substituted for the instance of the supertype
- Any program that works with a Digraph will also work with a Graph (but not *vice versa*)

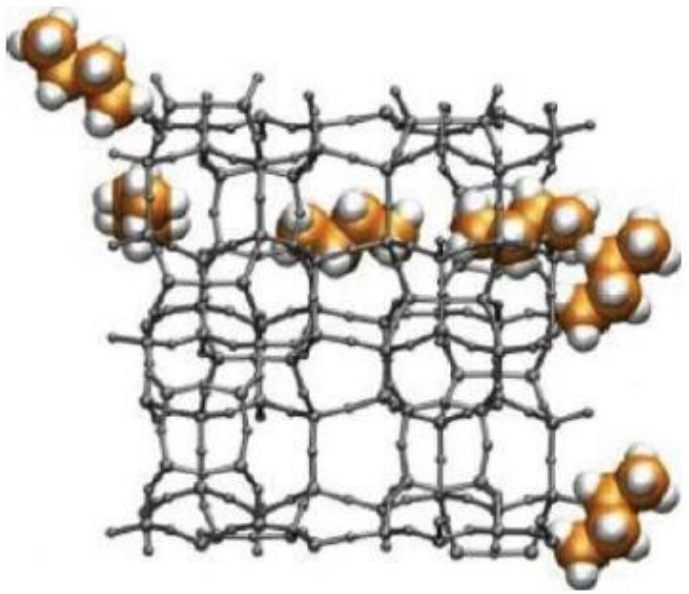
A Classic Graph Optimization Problem

- Shortest path from n1 to n2
  - Shortest sequence of edges such that
    - Source node of first edge is n1
    - Destination of last edge is n2
    - For edges, e1 and e2, in the sequence, if e2 follows e1 in the sequence, the source of e2 is the destination of e1
- Shortest weighted path
  - Minimize the sum of the weights of the edges in the path

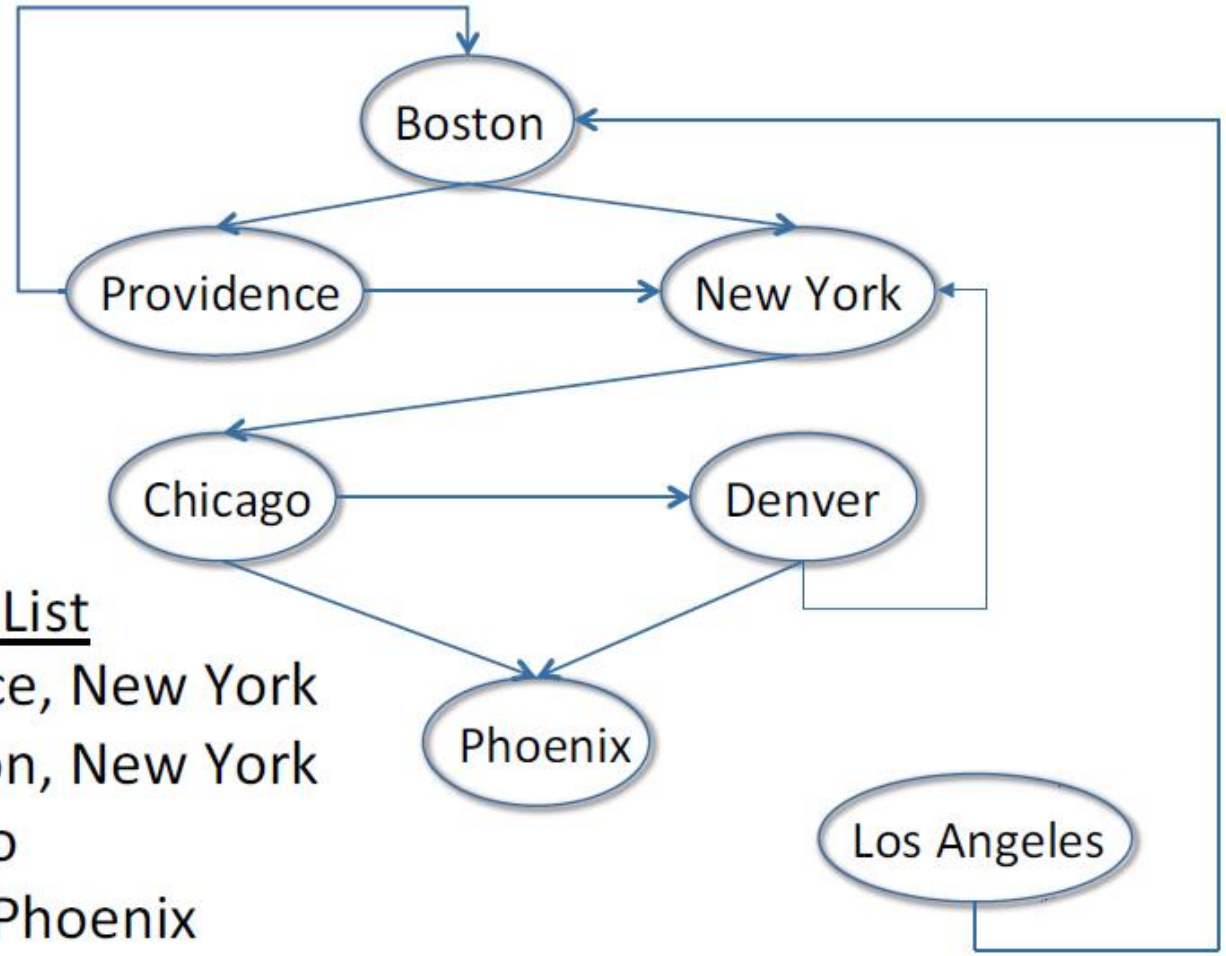
Some Shortest Path Problems

- Finding a route from one city to another
- Designing communication networks
- Finding a path for a molecule through a chemical labyrinth
- ...





An Example



Adjacency List

Boston: Providence, New York
Providence: Boston, New York
New York: Chicago
Chicago: Denver, Phoenix
Denver: Phoenix, New York
Los Angeles: Boston
Phoenix:

Build the Graph

```
def buildCityGraph(graphType):
    g = graphType()
    for name in ('Boston', 'Providence', 'New York', 'Chicago',
                 'Denver', 'Phoenix', 'Los Angeles'): #Create 7 nodes
        g.addNode(Node(name))
    g.addEdge(Edge(g.getNode('Boston'), g.getNode('Providence')))
    g.addEdge(Edge(g.getNode('Boston'), g.getNode('New York')))
    g.addEdge(Edge(g.getNode('Providence'), g.getNode('Boston')))
    g.addEdge(Edge(g.getNode('Providence'), g.getNode('New York')))
    g.addEdge(Edge(g.getNode('New York'), g.getNode('Chicago')))
    g.addEdge(Edge(g.getNode('Chicago'), g.getNode('Denver')))
    g.addEdge(Edge(g.getNode('Chicago'), g.getNode('Phoenix')))
    g.addEdge(Edge(g.getNode('Denver'), g.getNode('Phoenix')))
    g.addEdge(Edge(g.getNode('Denver'), g.getNode('New York')))
    g.addEdge(Edge(g.getNode('Los Angeles'), g.getNode('Boston')))
```

Finding the Shortest Path

- Algorithm 1, depth-first search (DFS)
- Similar to left-first depth-first method of enumerating a search tree (Lecture 2)
- Main difference is that graph might have cycles, so we must keep track of what nodes we have visited to avoid going in infinite loops

Note that we are using divide-and-conquer: if we can find a path from a source to an intermediate node, and a path from the intermediate node to the destination, the combination is a path from source to destination

Depth First Search

- Start at an initial node
- Consider all the edges that leave that node, in some order
- Follow the first edge, and check to see if at goal node
- If not, repeat the process from new node
- Continue until either find goal node, or run out of options
  - When run out of options, backtrack to the previous node and try the next edge, repeating this process

Depth First Search (DFS)

```
def DFS(graph, start, end, path, shortest, toPrint = False):
    path = path + [start]
    if toPrint:
        print('Current DFS path:', printPath(path))
    if start == end:
        return path
    for node in graph.childrenOf(start):
        if node not in path: #avoid cycles
            if shortest == None or len(path) < len(shortest):
                newPath = DFS(graph, node, end, path, shortest, toPrint)
                if newPath != None:
                    shortest = newPath
        elif toPrint:
            print('Already visited', node)
    return shortest


def shortestPath(graph, start, end, toPrint = False):
    return DFS(graph, start, end, [], None, toPrint)
```

... returning to this point in the recursion to try next node

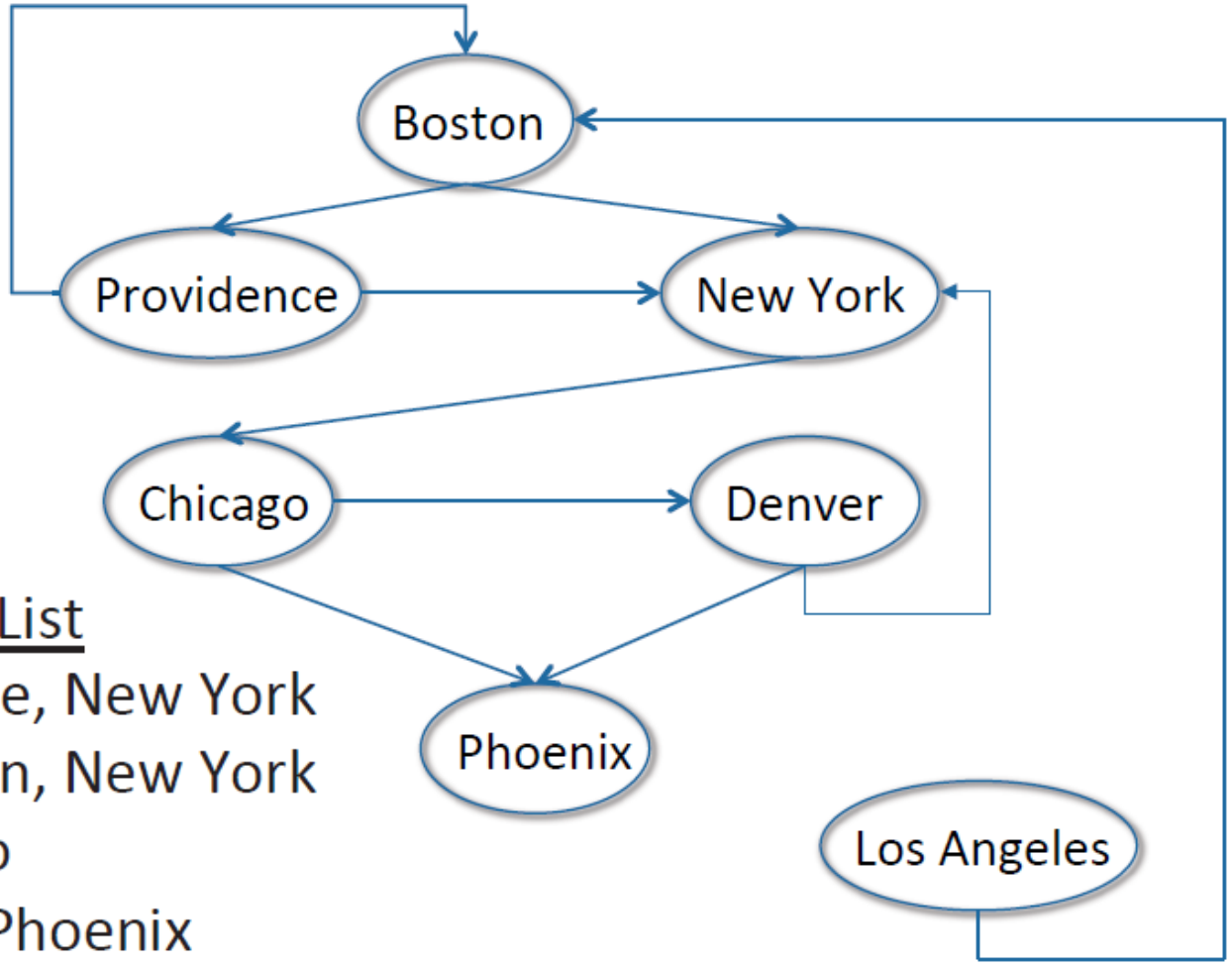
Note how will explore all paths through first node, before ...

Test DFS

```
def testSP(source, destination):
    g = buildCityGraph(DiGraph)
    sp = shortestPath(g, g.getNode(source), g.getNode(destination)
                      toPrint = True)
    if sp != None:
        print('Shortest path from', source, 'to',
              destination, 'is', printPath(sp))
    else:
        print('There is no path from', source, 'to', destination)

testSP('Boston', 'Chicago')
```

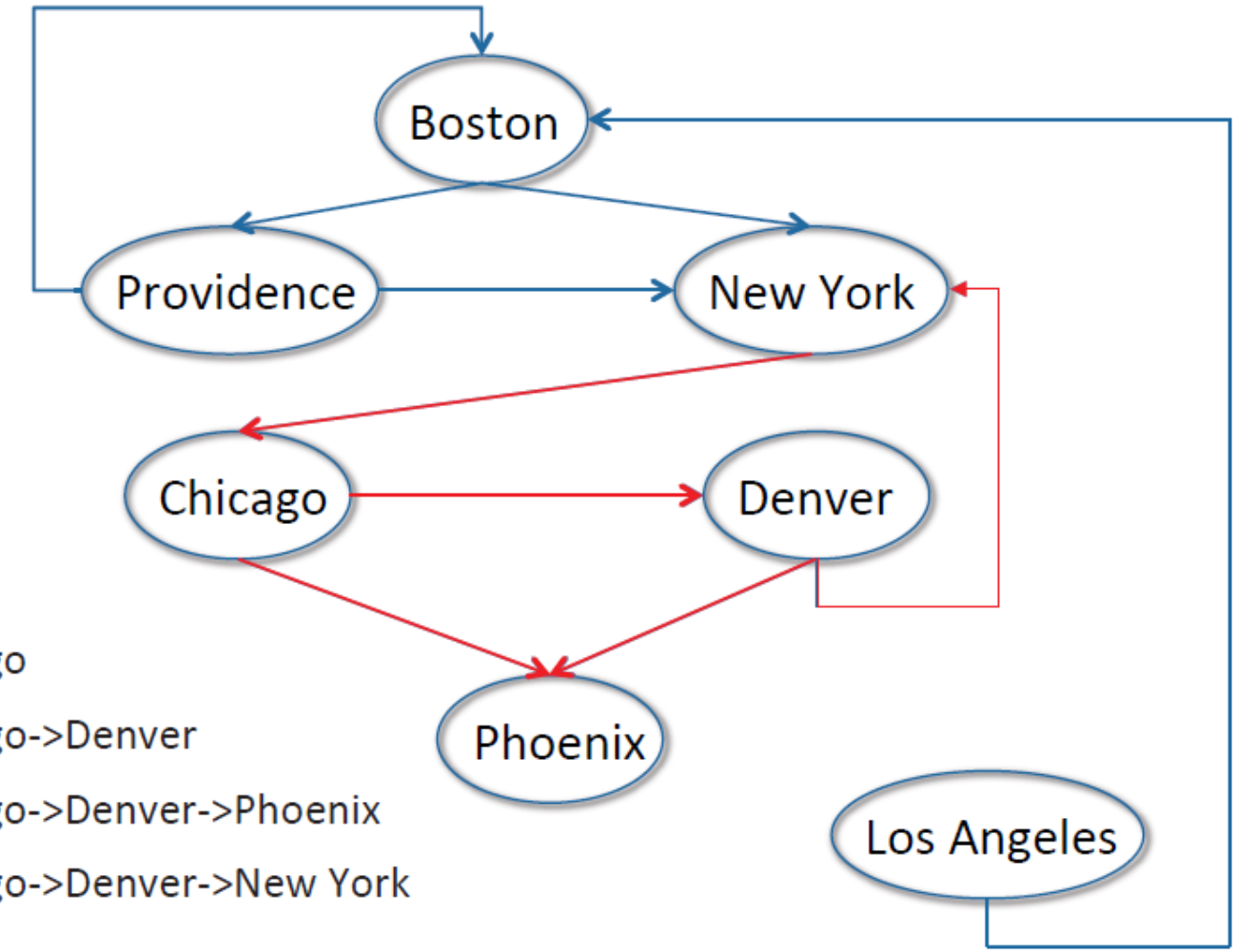
An Example



Adjacency List

Boston: Providence, New York
Providence: Boston, New York
New York: Chicago
Chicago: Denver, Phoenix
Denver: Phoenix, New York
Los Angeles: Boston
Phoenix:

Output (Chicago to Boston)



Current DFS path: Chicago

Current DFS path: Chicago->Denver

Current DFS path: Chicago->Denver->Phoenix

Current DFS path: Chicago->Denver->New York

Already visited Chicago

Current DFS path: Chicago->Phoenix

There is no path from Chicago to Boston

Output (Boston to Phoenix)

Current DFS path: Boston

Current DFS path: Boston->Providence

Already visited Boston

Current DFS path: Boston->Providence->New York

Current DFS path: Boston->Providence->New York->Chicago

Current DFS path: Boston->Providence->New York->Chicago->Denver

Current DFS path: Boston->Providence->New York->Chicago->Denver->Phoenix Found path

Already visited New York

Current DFS path: Boston->Providence->New York->Chicago->Phoenix Found a shorter path

Current DFS path: Boston->New York

Current DFS path: Boston->New York->Chicago

Current DFS path: Boston->New York->Chicago->Denver

Current DFS path: Boston->New York->Chicago->Denver->Phoenix Found a “shorter” path

Already visited New York

Current DFS path: Boston->New York->Chicago->Phoenix Found a shorter path

Shortest path from Boston to Phoenix is Boston->New York->Chicago->Denver->Phoenix

Breadth First Search

- Start at an initial node
- Consider all the edges that leave that node, in some order
- Follow the first edge, and check to see if at goal node
- If not, try the next edge from the current node
- Continue until either find goal node, or run out of options
  - When run out of edge options, move to next node at same distance from start, and repeat
  - When run out of node options, move to next level in the graph (all nodes one step further from start), and repeat

Algorithm 2: Breadth-first Search (BFS)

```
def BFS(graph, start, end, toPrint = False):
    initPath = [start]
    pathQueue = [initPath]
    while len(pathQueue) != 0:
        #Get and remove oldest element in pathQueue
        tmpPath = pathQueue.pop(0)
        if toPrint:
            print('Current BFS path:', printPath(tmpPath))
        lastNode = tmpPath[-1]
        if lastNode == end:
            return tmpPath
        for nextNode in graph.childrenOf(lastNode):
            if nextNode not in tmpPath:
                newPath = tmpPath + [nextNode]
                pathQueue.append(newPath)
    return None
```

Output (Boston to Phoenix)

Current BFS path: Boston

Current BFS path: Boston->Providence

Current BFS path: Boston->New York

Current BFS path: Boston->Providence->New York

Current BFS path: Boston->New York->Chicago

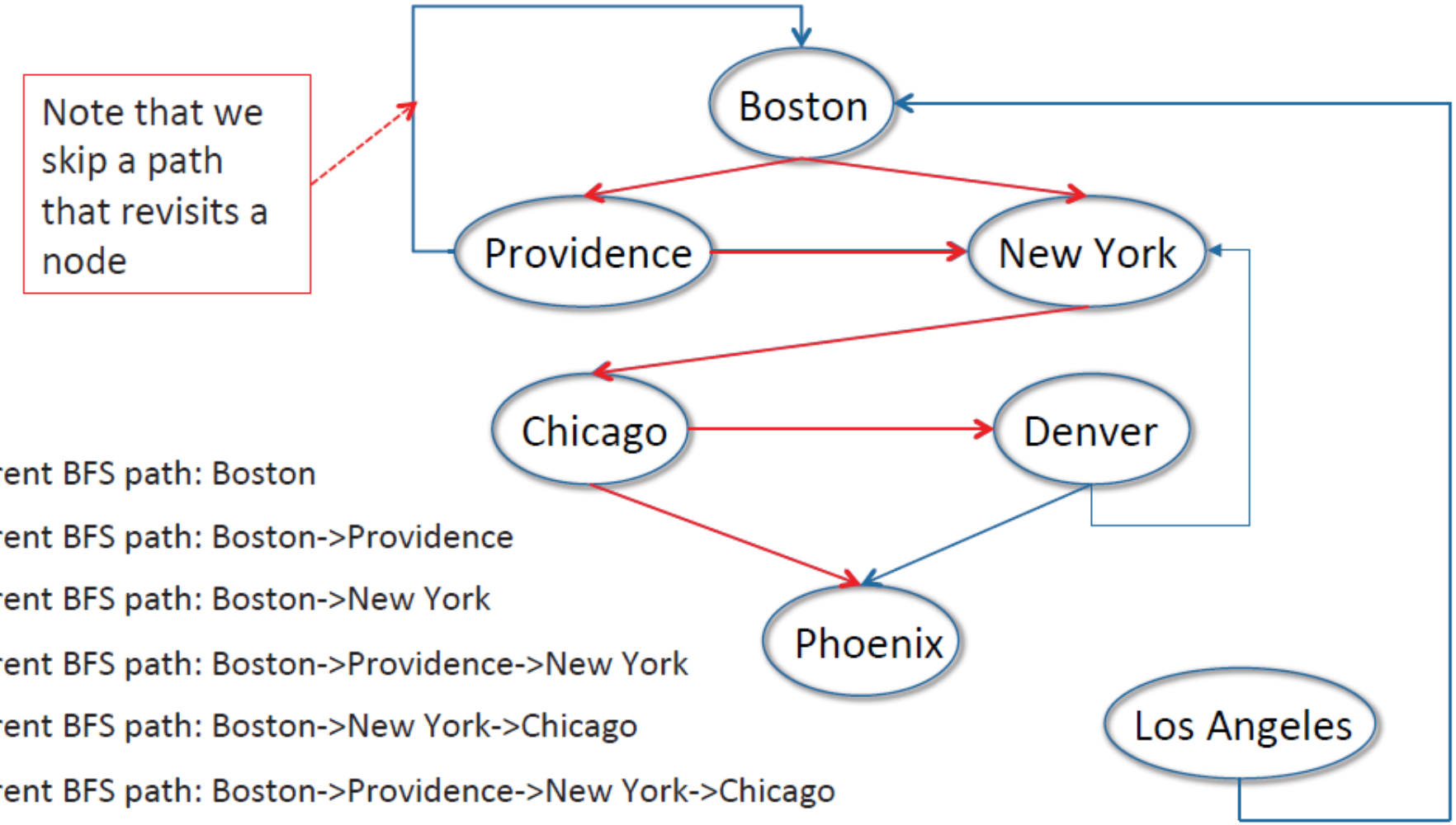
Current BFS path: Boston->Providence->New York->Chicago

Current BFS path: Boston->New York->Chicago->Denver

Current BFS path: Boston->New York->Chicago->Phoenix

Shortest path from Boston to Phoenix is Boston->New York->Chicago->Phoenix

Output (Boston to Pheonix)



Current BFS path: Boston

Current BFS path: Boston->Providence

Current BFS path: Boston->New York

Current BFS path: Boston->Providence->New York

Current BFS path: Boston->New York->Chicago

Current BFS path: Boston->Providence->New York->Chicago

Current BFS path: Boston->New York->Chicago->Denver

Current BFS path: Boston->New York->Chicago->Phoenix

Shortest path from Boston to Phoenix is Boston->New York->Chicago->Phoenix

What About a Weighted Shortest Path

- Want to minimize the sum of the weights of the edges, not the number of edges
- DFS can be easily modified to do this
- BFS cannot, since shortest weighted path may have more than the minimum number of hops

Recap

- Graphs are cool
  - Best way to create a model of many things
    - Capture relationships among objects
  - Many important problems can be posed as graph optimization problems we already know how to solve
- Depth-first and breadth-first search are important algorithms
  - Can be used to solve many problems

terms to find out more.

- weighted graph
- cycles
- tree
- complete graph
- clique
- bipartite graph

- Dijkstra’s algorithm

Finger Exercises

Modify the DFS algorithm to find a path that minimizes the sum of the weights. Assume that all weights are positive integers.

Consider a digraph with weighted edges. Is the first path found by BFS guaranteed to minimize the sum of the weights of the edges?