CSEC BFS/DFS

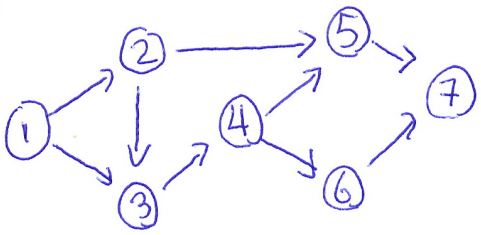
CSEC

February 3, 2017

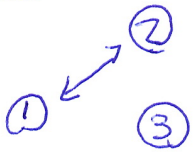
1 Brief Primer on Graphs

What is a graph? A graph is formally defined as a collection of vertices and edges, $G = \{V, E\}$. There are a couple of ways we can represent these graphs, below we will show a visual representation of a graph.

Ex1



Ex 2



1.1 Edge List

An edge lists represents a graph, solely through edges linking one node to another.

Ex1 $V = \{1, 2, 3, 4, 5, 6, 7\}$

$E = \{(1,2), (1,3), (2,3), (2,5), (3,4), (4,5), (4,6), (5,7), (6,7)\}$

Ex2 $V = \{1, 2, 3\}$

$E = \{(1,2), (2,1)\}$

1.2 Adjacency Matrix

Given N nodes, we will represent a graph using an $N \times N$ matrix, $M_{N \times N}$, where each $M_{i,j}$ represents an edge from the i^{th} node to the j^{th} node.

Ex 1

	1	2	3	4	5	6	7
1	0	1	1	0	0	0	0
2	0	0	1	0	1	0	0
3	0	0	0	1	0	0	0
4	0	0	0	0	1	1	0
5	0	0	0	0	0	0	1
6	0	0	0	0	0	0	1
7	0	0	0	0	0	0	0

Ex 2

	1	2	3
1	0	1	0
2	1	0	0
3	0	0	0

1.3 Adjacency List

Given N nodes, we will reprsent a graph using N resizable lists.

Ex 1

1: 2 3
2: 3 5
3: 4
4: 5 6
5: 7
6: 7
7:

Ex 2

1: 2
2: 1
3:

2 Depth First Search

Idea: Search as deep as possible

Pros:
- Generally uses less memory
- Easier to implement

Cons:
- Quite slow on graphs were you must traverse every state to acheive an answer, and the "recursive depth" is quite steep

Pseudo code:
- There exists an iterative and recusive algorithm, which are both lagically the same.

Recursive

```
def DFS(U):
    label U as a visited-node
    for all V in adj[U] do
        if not visited[V] then
            DFS(V)
```

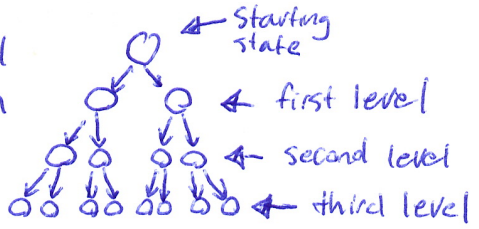
Iterative

```
def DFS(U):
    S := stack()
    S.push(U)
    while S is non-empty do
        n := S.pop()
        for all V in adj[n] do
            if not visited[n] then
                visited[n] := true
                S.push(n)
```

notes:
- As long as you are properly keeping track of states your program will terminate
- Good for problems where you need to find 'a solution', with some leniancy on time constraints
- Can be adopted to other problems i.e. back tracking

3 Breadth First Search

Idea: - Search 'level-first', until you reach a valid solution



Pros - ability to solve shortest path problem quite easily
- Quite fast for finding the "shortest path" compared to a DFS solution

Cons - Requires a lot more memory generally to represent states, compared to DFS
- Can't be adapted too too many problems unlike DFS. I.e. problems solved should be able to be interpretted as a 'shortest path' problem

Pseudo code: - There only exists an iterative algorithm involving a queue

Iterative

```
def BFS(U):
    Q := Queue()
    Q.push(U)
    while Q is non-empty do
        n := Q.pop()
        for all V in adj[n] do
            if not visited[V] then
                visited[V] := true
                Q.push(V)
```

notes
- you can generally represent states as a collection of tuples/pairs, however if you need to keep track of more elements/attributes it's easier to represent with a class

4 Sample Problem

Torn To Pieces - 2015 ICPC North American Qualifier
Author: Nathan Backman
Problem Statement

4.1 Description

You have arrived in The Big City but your journey is not yet complete. You must still navigate the subway and get to your final destination. The information booth in the subway station is unattended and fresh out of maps of the subway system. On the floor you notice fragments of a map. Can you piece together enough of the map to figure out how to get to your final destination?

Each fragment of the map happens to perfectly contain a single subway station while also identifying all of the other stations that it connects to. Each connection between stations is bi-directional such that it can be travelled going either direction. Using all of the available fragments, your task is to determine the sequence of stations you must pass through in order to reach your final destination or state that there is no route if you don't have enough information to complete your journey.

4.2 Input

The first line of input has an integer, $2 \leq N \leq 32$, that identifies the number of pieces of the map that were found.

The following N lines each describe a station depicted on one of those pieces. Each of these lines starts with the name of the station they describe and is followed by a space-separated list of all of the station names that are directly connected to that station (there may be as many as $N - 1$).

The final line identifies a starting station and a destination station. The destination station is guaranteed to be different than the starting station.

Each station name is a string of up to 20 characters using only letters a–z and A–Z. It is guaranteed that there is at most one simple route (without revisiting stations) from the starting station to the destination station.

4.3 Editorial

For this problem, we'd like to construct an undirected unweighted graph from the input. Note that during the construction we have to take care with the specification of the nodes, specifically we may be queried for nodes which weren't already specified in the adjacency list. The goal is to find **a path** from the start node to the end node. So we can use any of the algorithms previously discussed, in this case we use a solution for both breadth first search and depth first search.

4.4 Breadth First Search Solution

```
import queue

MAXN = 1000
N = 0
node_id = 0
nodes = {}
names = {}
adj = [[] for i in range(MAXN)]

def bfs(start_node, end_node):
    q = queue.Queue()
    v = [False for i in range(MAXN)]
    q.put((start_node, [start_node])) # current node, list of visited nodes
    v[start_node] = True
    while not q.empty():
        state = q.get()
        node = state[0]
        path = state[1]
        if node == end_node:
            return path
        for i in adj[node]:
            if not v[i]:
                q.put((i, path+[i]))
                v[i] = True
    return None


def apply_id(entry):
    global nodes, node_id
    if entry not in nodes:
        nodes[entry] = node_id
        names[node_id] = entry
        node_id += 1

if __name__ == '__main__':
    N = int(input())
    for t in range(N):
        data = str(input()).strip().split(' ')
        for i in data:
            apply_id(i)
        for i in data[1:]:
            adj[nodes[data[0]]].append(nodes[i])
            adj[nodes[i]].append(nodes[data[0]])
    data = str(input()).strip().split(' ')
    apply_id(data[0])
    apply_id(data[1])
    start_node = nodes[data[0]]
    end_node = nodes[data[1]]
    ans = bfs(start_node, end_node)
    if ans:
        for i in ans:
            print(names[i],end=' ')
        print('')
    else:
        print('no route found')
```

4.5 Depth First Search Solution

```
MAXN = 1000
N = 0
node_id = 0
nodes = {}
names = {}
adj = [[] for i in range(MAXN)]
v = [False for i in range(MAXN)]


def dfs(node, end_node, path):
    path.append(node)
    if node == end_node:
        return path
    ret = None
    for i in adj[node]:
        if not v[i] and not ret:
            v[i] = True
            ret = dfs(i, end_node, path[:])
    return ret



def apply_id(entry):
    global nodes, node_id
    if entry not in nodes:
        nodes[entry] = node_id
        names[node_id] = entry
        node_id += 1

if __name__ == '__main__':
    N = int(input())
    for t in range(N):
        data = str(input()).strip().split(' ')
        for i in data:
            apply_id(i)
        for i in data[1:]:
            adj[nodes[data[0]]].append(nodes[i])
            adj[nodes[i]].append(nodes[data[0]])
    data = str(input()).strip().split(' ')
    apply_id(data[0])
    apply_id(data[1])
    start_node = nodes[data[0]]
    end_node = nodes[data[1]]
    v[start_node] = True
    ans = dfs(start_node, end_node, [])
    if ans:
        for i in ans:
            print(names[i],end=' ')
        print('')
    else:
        print('no route found')
```