CS201 Discussion 14

ALLWORDLADDERS AND GALAXYTRIP
Problem Statement

A word ladder is a sequence of words in which each word can be transformed into the next word by changing one letter. For example, the word ladder below changes 'lot' to 'log'.

lot dot dog log

This is not the shortest word-ladder between 'lot' and 'log' since the former can be immediately changed to the latter yielding a word ladder of length two:

lot log

The first and last words in a word ladder are the anchor rungs of the ladder. Any other words are interior rungs. For example, there are three interior rungs in the ladder below between 'smile' and 'evote'.

smile smite smote smote emote evote

In this problem you’ll write a method that has parameters representing potential interior rungs: an array of strings (these may by nonsense or English words), and the anchor rungs --- two strings. Your code must determine the shortest word ladder between the anchor rungs that uses at least one interior rung, and the number of such ladders. Return an array containing two ints: the first is the length of the shortest valid word ladder and the second is the number of shortest ladders. If there are no valid ladders return [0,0].
Comparison to WordLadder

To recap – in WordLadder, we were trying to find if some path exists.

Next word: Changing one letter
- For example, "lot" ⇔ "log" "lot" ⇔ "law"

Now, we want to find what the shortest path is, and how many paths have the same length as the shortest path.

### Example 1
- `words = {"hot", "dot", "dog"}`
- `from = "hit"`  `to = "cog"`
- Returns "ladder"

### Example 2
- `words = {"mist", "fist", "fish"}`
- `from = "lost"`  `to = "cost"`
- Returns "none"

---

What’s the algorithm?

- `isNext(String from, String to)`
  - Helper method `returns true` if there is only 1 letter difference ⇔ a directed edge

- Shortest path, not weighted ⇔ xFS
Input as a graph

Following the pattern, probably the first step you’ll want to take is to turn the input into a graph.

Here, the vertices are words. Edges should connect words which are potential adjacent rungs in a ladder – i.e. are different by one letter. You should be able to reuse code from WordLadder and other APTs to easily construct the graph.

Include from, to, and all the words in the input array in your graph.

Then, the problem is as follows: How many from-to shortest paths are there, and what is their length?
Example
e.g. if
from = “cots”
to = “mars”
words = [“bars”, “bats”, “bots,” “cats”, “mats”]
Shortest path and BFS

As we saw in ErdosNumbers, BFS is useful for finding distances. Thus, a good first step after constructing a graph may be to find the distances of each word from the start word using BFS.
Counting shortest paths

Then, we’ve found the shortest path length to the target word, so all that’s left to do is count how many paths of that length exist.

First, let’s say that trivially, the start word has one shortest path to itself, the empty path.
Counting shortest paths

Then, we can manually count the shortest paths to all nodes in this example. What is the trend here?

```
cots  bots  cats  bats  mats  mars
0  1  1  2  2  3  3  2  3  2  4  4
```
Counting shortest paths – dynamic programming

For a vertex distance $i$ away from the starting vertex, the number of shortest paths to that vertex is the sum of the number of shortest paths to its distance $i-1$ neighbors.

This makes sense – any shortest path to a distance $i$ vertex will go to some distance $i-1$ neighbor of that vertex before reaching that vertex.
Dynamic program encoding

Then, to find the number of shortest paths in code, keep a map of words to the number of shortest ladders to each word. For the pseudocode, we’ll call this shortestPaths. Start by putting the pair (from, 1) into shortestPaths.

If we reuse the BFS code from ErdosNumbers, we know when we visit a vertex, we’ve visited all the vertices which are closer to the start vertex.

Then, for any vertex \( v \), we can find the number of shortest paths to it as follows:

```java
sum = 0;
for each neighbor u of v:
    if dist[u] == dist[v] – 1:
        sum += shortestPaths.get(u)
shortestPaths.put(v, sum)
```
Full solution

Then, the full solution is as follows:

Take the input and turn it into a graph

Use the BFS from ErdosNumbers to find the shortest distance to each vertex. As an additional step, when we visit a vertex, use the pseudocode from the previous slide to compute the number of shortest paths to that vertex.

Our output is just the values for the key to in our distance and shortestPaths maps.
Problem Statement

It's the year 2059. You have some free time in May this year and want to visit an interesting galaxy. You will need some machines during your trip and you are wondering how many of them can you take.

There are a number of machines in a store. Each machine may need some other machines to work properly, and you want all the machines you take to work. Luckily for you, in the year 2053, a special rule was introduced to make customers' lives easier. The rule says that if a machine A depends on another machine B, then machine B also depends on machine A.

You want to buy K different machines such that all of them will work properly. You are given a String[] dependencies describing the dependencies between the machines. Element i of dependencies is a space delimited list of machines that machine i is dependent on. Return a int[] with all possible positive values for K in ascending order.

Example:

{"1 2", "0", "0", ""}  

- Machine 0 needs machines 1 and 2 to work  
- Machine 1 needs machine 0 to work  
- Machine 2 needs machine 0 to work  
- Machine 3 does not need any other machine to work

The allowed K values are 1 (you take only machine 3), 3 (you take machines 0,1,2) and 4 (you take all the machines). So you should return

{1,3,4}
GalaxyTrip as a graph problem

Clearly, we can represent the set of machines as a graph:

The vertices represent the machines

Two machines \( i, j \) share an edge if \( i \) requires \( j \) to work vice-versa. e.g.:

\{
"1 2", "0", "0", ""
\}

- Machine 0 needs machines 1 and 2 to work
- Machine 1 needs machine 0 to work
- Machine 2 needs machine 0 to work
- Machine 3 does not need any other machine to work
Another graph example

```
{"4 2", "3", "0 4", "1", "0 2", "6", "5"}
```

Returns: `{2, 3, 4, 5, 7}`
Components

For this APT, converting the input into a graph is pretty simple – vertex i is adjacent to all the values that appear in dependencies[i]. It will probably be a useful first step to turn this into a Map from Integers to Sets of Integers, but it’s not necessary to solve the APT.

Then, we just have to think about how to solve the problem using the graph.

Notice that when we select a machine, we have to select all machines it’s connected to in the graph. Then, the sizes of the groups of machines we can pick depend on the sizes of components in the graph.

```
{"4 2", "3", "0 4", "1", "0 2", "6", "5"}
Returns: {2, 3, 4, 5, 7 }
```
Finding component sizes

We know that both DFS and BFS will visit all vertices in a graph connected to their start point.

Then, DFS and BFS are useful in finding component sizes – if we run DFS or BFS on a vertex $v$, it will visit a number of vertices equal to the size of $v$’s component.

So, we can run DFS or BFS on every vertex to find all the component sizes with one catch – we won’t run DFS or BFS on a vertex whose component we’ve visited before.

So, we’ll keep track of the set of vertices whose components we’ve visited, and then union that set with our DFS/BFS’s visited set each time, to avoid revisiting the same component.
Pseudocode for finding component sizes

V* = empty set
Sizes = list of integers
For each vertex v:
    If v not in V*:
        V = empty set
        Run DFS/BFS on v, using V as the visited set for DFS/BFS
        Add V.size to sizes
        Add all elements in V to V*
List of sizes to final answer

Once we have a list of sizes, finding the answers simply requires finding all possible additions of the sizes. This can be done using recursion – for each size, we either include it in our sum or not.

0+0+0 = 0 0+0+3 = 3
2+0+0 = 2 2+0+3 = 5
0+2+0 = 2 0+2+3 = 5
2+2+0 = 4 2+2+3 = 7

The problem is this can take exponential time and time out the tester.

{"4", "2", "3", "0 4", "1", "0 2", "6", "5"}

Returns: {2, 3, 4, 5, 7}
Improving the runtime

Rather than maintain a list of all component sizes, we’ll maintain an array of frequencies of sizes – i.e. if freq is our array, freq[i] is the number of components with size i. Turning our list of sizes into this array is fairly simple.

Then, rather than choose whether to include each component or not, we choose how many components of size 1 to include, then how many components of size 2 to include, etc. In the case where there are many same-sized components, this reduces the amount of work we have to do.

This still can take exponential time, but will at least finish within the APT tester’s timer.
How this reduces work

Consider the example where our components had size 2, 2, and 3. Before, our computations were:

\[
\begin{align*}
0+0+0 &= 0 \\
0+0+3 &= 3 \\
2+0+0 &= 2 \\
2+0+3 &= 5 \\
0+2+0 &= 2 \\
0+2+3 &= 5 \\
2+2+0 &= 4 \\
2+2+3 &= 7
\end{align*}
\]

But if we account for duplicate 2s, our computations become:

\[
\begin{align*}
0 \times 2 + 0 \times 3 &= 0 \\
0 \times 2 + 1 \times 3 &= 3 \\
1 \times 2 + 0 \times 3 &= 2 \\
1 \times 2 + 1 \times 3 &= 5 \\
2 \times 2 + 0 \times 3 &= 4 \\
2 \times 2 + 1 \times 3 &= 7
\end{align*}
\]

Then, we only do 6 computations rather than 8. This is not a drastic improvement, but in the case where all the components are the same size, this takes us from exponential \(2^n\) to linear \(n\) computations.
Recursive pseudocode

S = set of answers (global variable)
count(freq, 0, 0)

count(freq, total, pos):
    if total != 0, add total to S //add one possible answer
    if pos >= sizes’ length, return //we’ve iterated through whole
        //list, we’re done
    for i from 0 to freq[pos]:
        count(freq, total+i*pos, pos+1) //try adding a different
            //number of components of each size
Finishing the APT

Once you use the code on the previous slide to turn a list of component sizes into a set of possible machine set sizes, you just have to turn it into a sorted int array to finish the APT.

Some useful data structures:

```java
ArrayList<Integer> sizes = new ArrayList<Integer>();
    HashSet<Integer> allVisited = new HashSet<Integer>();
HashSet<Integer> combinations = new HashSet<Integer>();
//Turn the result into a sorted array and return it
ArrayList<Integer> combinationList =
    new ArrayList<Integer>(combinations);
```

Some useful helper methods:

```java
public void search(String[] dependencies, int current, HashSet<Integer> visited) {

public void computeSizes(int[] freqs, HashSet<Integer> combos, int current, int pos) {
```