CS:1210 Project 1

Spring 2024

Due dates: **Phase 1**: Fri 3/22 at 11:59 pm.
Phase 2: Fri 3/29 at 11:59 pm.
Phase 3: Fri 4/5 at 11:59 pm.

Introduction

This project requires you to implement a game called "word ladder", also known by other names such as "word golf" and "Word-Links". Lewis Carroll, the author of "Alice's Adventures in Wonderland", claims to have invented this game in 1877. You can see an online version of this game at `https://ceptimus.co.uk/wordladder.php`. For our purposes, implementing this game will expose you to concepts such as (i) networks and how to represent them as using data structures, (ii) efficient algorithms for traversing networks, (iii) simple visualizations of networks and their statistics, and (iv) implementing user-interaction and run-time error handling. Overall, it will give you a lot of programming practice.

A brief overview of the game. The game starts with your program prompting the user with a 5-letter *source* word s and a 5-letter *target* word t. The user is required to respond by typing a sequence of 5-letter words $s = w_0, w_1, w_2, \ldots, w_{n-1}, w_n = t$ such that (a) all the words in the sequence are valid 5-letters words in the English dictionary, and (b) every word is obtained from the previous word by changing one letter. Thus, the words sequence $w_0, w_1, w_2, \ldots, w_{n-1}, w_n$ forms a "ladder" from word s to word t. For example, your program might provide the following prompt:

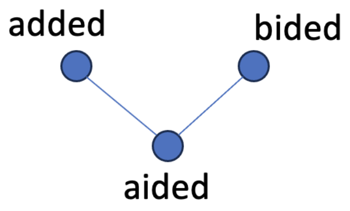
```
Source word: sport
Target word: house
```

One correct response from the user is

```
sport
spurt
spurs
sours
tours
touts
routs
route
rouse
house
```

Your program checks if the user has typed a correct response and responds appropriately. This is the basic version of the game. In the game you will implement, your program will provide the user with an "easy game" versus a "challenging game" option. The source and target word chosen by your program will depend on the option picked by the user. Further, as the user types words, your program will let the user know how far they are from the target word. Finally, your game can provide the user a small number of "lives", which the user can use at any point in the game. A "life" is simply a suggestion for the next word.

To keep you on task and to provide you with partial solutions, we have split this project into 3 phases (due dates shown above), which we now describe.



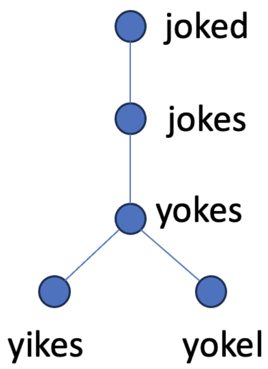


Figure 1: (left) This is the word network of the list `['added', 'aided', 'bided']`.(right) This is the word network of the list `['joked', 'jokes', 'yikes', 'yokel', 'yokes']`.

Phase 1

This phase consists of 3 main tasks:

1. Donald Knuth, a Turing award winning computer scientist created a file of 5,757 5-letter words. This is part of his book called *Stanford Graphbase*, a collection of datasets and programs which generate and manipulate graphs and networks. We have posted this file (called "`words.txt`") and your first task is to read the words from this file into a list. This list of words will form the "dictionary" for this project, i.e., it will form the set of valid 5-letter words in the English language.

2. Your second task is to write a function that takes a list L of words and builds a *word network* for the list L. The word network for the list L is a network whose nodes are the words in L and whose edges connect pairs of words that differ in exactly one letter. The figure below shows two word networks.

   You are required to represent this word network using an *adjacency list* representation, which we now define. Let L be the list $[w_0, w_1, \ldots, w_{n-1}]$ of n words. For any word w_i in L, the *neighbors* of w_i are all the words in L to which w_i is connected by an edge in the word network of L. The adjacency list representation of the word network of L is a length-n list whose first element is the list of neighbors of w_0, whose second element is the list of neighbors of w_1, and so on. Thus the adjacency list representation of the word network is a list of length n, containing a bunch of lists.

3. Your third task is to define some functions that provide basic information about the word network. Let the *degree* of a node in the word network be the number of neighbors it has.

   (a) Compute the list of neighbors of a given node in the word network.
   (b) Compute the list of all the *isolated nodes* in the word network. Isolated nodes are nodes with degree 0.
   (c) Compute the list of all the *isolated edges* in the word network. Isolated edges are pairs of nodes that are connected to each other, but not to any other node in the word network.
   (d) Compute the list of nodes (words) ordered by non-decreasing degree.
   (e) Compute the degree distribution of the word network.

Overall, you are required to write and test 8 functions for Phase 1; one for Task 1, two for Task 2, and five for Task 3, corresponding to the 5 items (a)-(e) above. Precise specifications of all the functions you are required to write are provided in the posted file `project1Phase1.py`.

What you need to submit: (i) A python file called `project1Phase1.py` containing definitions of the 8 functions you are required to complete, (ii) A pdf file containing answers to the following questions about the word network constructed from the 5,757 words in the file `words.txt`.

(a) How many isolated nodes are there in this word network?

(b) List the 5 isolated nodes in this word network that appear earliest in alphabetical order.

(c) How many isolated edges are there in this word network?

(d) What is the maximum degree of a node in this word network?

(e) Output a node in this word network with highest degree; if there are multiple nodes with highest degree, list the node that appears first in alphabetical order.

Once you have implemented and tested all your functions, you can add a short main program that produces the answers to the questions above as output.

Phase 2

This phases focuses on traversing the word network you have built in Phase 1. You will be defining the following 3 functions. Detailed specifications along with definitions are provided in the starter file `project1Phase2.py`.

- `def searchWordNetwork(wordList, nbrsList, source):` This function performs a breadth first search of the word network from the `source` node, using the algorithm described in Problems 1-2 in WPS 7. This function is required to return parent and distance information obtained from the search of the word network.
- `def findPath(wordList, nbrsList, source, target):` This function returns a shortest path in the word network from the `source` word to the `target` word. It first calls the `searchWordNetwork` function and uses the parent information returned by this function to extract a shortest path.
- `def findComponents(wordList, nbrsList):` This function returns all the connected components of the word network.

After you define and test these functions, proceed to write code to answer the following questions about the word network N of the list of 5757 words in `words.txt`.

(a) Consider all words that are reachable from the word `"abode"` in N. Among these words, what is the distance of the farthest word from `"abode"`?

(b) Consider all words that are reachable from the word `"abode"` in N. Among these words, what are all the words that are farthest from `"abode"`?

(c) For each of these pairs of words, find a shortest path in N from the first word to the second word:
`"abode"` to `"house"`
`"sweet"`, `"yucky"`
`"index"`, `"third"`
`"wheel"`, `"turns"`
`"lucky"`, `"break"`

(d) How many components does N have?

(e) What is the size of the largest connected component in N?

(f) What are the first 10 words in this largest component?

(g) Print out the component of size 8 in N.

What you need to submit: (i) A python file called `project1Phase2.py` containing definitions of the 3 functions you are required to complete, (ii) a python file called `project1Phase2WithMain.py` that contains these 3 functions plus a main program whose execution produces the answers to the questions (a)-(g), and (iii) a pdf file containing answers to the questions above. Note that the python files you submit will contain all the functions you defined for Phase 1 of the project. You are welcome to replace your Phase 1 code by the solutuion we provide.

Phase 3

In this phase, you will complete the project by writing a *word ladder* game program. The basic idea of this game is that the program prompts the user with a 5-letter *source* word and a 5-letter *target* word and the user responds by typing a word "ladder" in response. This brief description makes the game seem a bit boring, but you will be adding a few bells and whistles to this basic game to make it more interesting. There are three tasks in Phase 3, *preprocessing*, *word choice*, and *user interaction*. These are described further below.

Organization of your programs

You will submit two files: `project1Phase3Preprocessing.py` and `project1Phase3GamePlay.py`. The file `project1Phase3Preprocessing.py` will contain an implementation of the preprocessing task and it is only executed once. The file `project1Phase3GamePlay.py` will contain an implementation of the word choice and user interaction tasks. This is executed whenever you want the user to play the word ladder game.

All three tasks are guided by a few parameters. A text file called `parameters.txt` will contain the values of these parameters. You can edit this text file and change the values of the parameters, if you want your programs to behave differently.

The program in `project1Phase3Preprocessing.py` will read from the files `words.txt`, `parameters.txt`, and text files downloaded from Project Gutenberg. It will write to a file called `gameInformation.txt`. The program in `project1Phase3GamePlay.py` will read from both `parameters.txt` and `gameInformation.txt` and will interact with the user. Since this program is interacting with the user, it needs to be very efficient.

Preprocessing

This task is executed only once; not each time the game is played. Think of it as something that needs to happen just once, when your program is "installed". Since this task is executed only once, it is okay for this task to take a few seconds, i.e., not be as efficient as the remaining two tasks. As mentioned above, to complete this task, you will read from a file called `parameters.txt` and write into a file called `gameInformation.txt`.

The preprocessing tasks consists of the following steps. Let N denote the word network of the 5,757 words in the file `word.txt`.

1. Find the list of words P, in the largest connected component of N. From the answer to question (e) in Project 1 Phase 2, you know that there are 4493 words in P. For the rest of this description, assume that P is sorted in alphabetical order.

2. Partition P into a sublist of "easy" words and a sublist of "hard" words. To find which words in P are "easy" and which are "hard", you will need to read and extract words from 6 text files I have posted[1]. Specifically, you will compute the frequency of each word in P in the 6 texts. Then, for a

[1]The texts I downloaded are classics from among the most popular titles at Project Gutenberg (`https://www.gutenberg.org`). These are "Pride and Prejudice" by Jane Austen, "A Room with a View" by E.

parameter p (between 0 and 100), you will take p % of the words in P that are most frequent and denote them the "easy" words. The remaining words will be denoted "hard" words. In the rest of this description, I will use P_E and P_H to denote these two lists of words, respectively. For example, if we set $p = 30$, then 30% of the words in P that are most frequent – this is 30% of 4493 = 1347.9 ≈ 1348 (rounding up) – are considered "easy". The intuition here is simple; words that are more frequently used are considered "easy" and words that are less frequently used are considered "hard". The specific value of the parameter p will be read from the file `parameters.txt`.

3. Write all of this information into a file called `gameInformation.txt`. The specific information you will write into this file are the list P_E of "easy" words, the list P_H of "hard" words, and the adjacency list representation of the word network of P. An illustration of the format of the file `gameInformation.txt` is as follows:

```
1348
easyWord1
easyWord2
...
...
easyWord1348
3145
hardWord1
hardWord2
...
...
hardWord3145
neighbors of word1
neighbors of word2
...
...
neighbors of word4493
```

   This illustration assumes that $p = 30$. This first line of this file contains the number of "easy" words. This is followed by all the "easy" words in alphabetical order, one word per line. This is followed by the number of "hards" words and then all the "hard" words in alphabetical order, one word per line. Finally comes the word network of all the words in P. It is assumed that the words in P are in alphabetical order and neighbors of each word in P are comma separated and occur in a single line.

Note: We are computing the word network of P and writing it into a file in the preprocessing stage because computing the word network is a relatively time consuming task and we don't want a user, who is playing the game to be waiting on the word network to be computed. Since we are writing all this into a file, the program that plays the game (`project1Phase3GamePlay.py`) can just read it.

Word choice

Now suppose that the preprocessing task is complete and the file `gameInformation.txt` has been correctly written into. Then we can execute the program that plays the game. Recall that this program is in the file `project1Phase3GamePlay.py`. The first thing that this program does is read from the files `parameters.txt` and `gameInformation.txt`. Then, after setting up some data structures, this program asks the user if they want to play the "easy" version of the game or the "hard" version. Depending on

M. Forster, "Moby Dick; Or, The Whale" by Herman Melville, "Little Women; Or, Meg, Jo, Beth, and Amy" by Louisa May Alcott, "The Great Gatsby" by F. Scott Fitzgerald, and "Frankenstein; Or, The Modern Prometheus" by Mary Wollstonecraft Shelley.

the user's input, the program either picks source and target words that are relatively easy to connect by a word ladder or relatively hard. Below, we describe how your program should do that.

The high level idea for how to do this is simple. If the user wants to play the "easy" version of the game, then your program should pick two words between which there is a *short* path that contains very few "hard" words. Otherwise, if the user wants to play the "hard" version of the game, then your program is allowed to pick two words that are farther apart from each other and there are no restrictions on which words appear in the ladder.

- **Easy version:** In this case, your program should first pick a target word at random, from the list of all "easy" words P_E. Then for two given non-negative integer parameters ed_1, ed_2, where $ed_1 \leq ed_2$, your program should consider all words that are at distance between ed_1 and ed_2 from the target word and pick one of these words at random as the source word. For example, if $ed_1 = 5$, $ed_2 = 10$, then the source word is chosen at random from all the words that are between 5 units and 10 units distance from the target word. The values ed_1 and ed_2 are parameters and their values will appear in `parameters.txt`.

  Furthermore, in the "easy" version of the game, we don't want the user to have to use too many "hard" words. In order to pick a source and target words that are largely connected by "easy" words, we assign to each node corresponding to a "hard" word a weight w, where w is a non-negative integer parameter. In a network where nodes have weights, the distance between a pair of nodes along a path is defined as the sum of the number of edges in the path plus the weights of the nodes in the path. See the figure below for an illustration of this notion. For example, if we set $w = 3$, then any path that includes 2 "hard" words in it has length at least 6. There is a simple modification to the breadth first search algorithm that can non-negative integer accommodate node weights. This modification is described later.

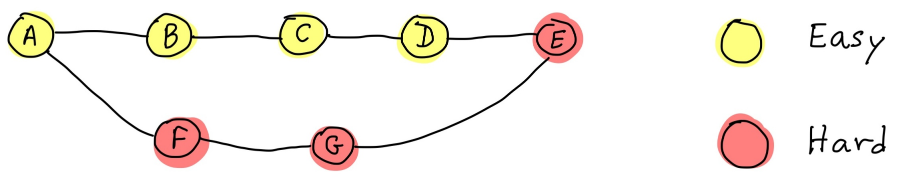


Figure 2: Suppose we assign a weight $w = 3$ to every "hard"node (shown in red). Then the path A-B-C-D-E has length $4 + 3 = 7$ because this path includes 4 edges and one node of weight 3. On the other hand, the path A-F-G-E has length $3 + 9 = 12$. So the shorter path from A to E is the one that includes fewer "hard" nodes.

- **Hard version:** In this case, your program should pick a target word at random, from the list of all words P. Then for two given non-negative integer parameters hd_1, hd_2, where $hd_1 \leq hd_2$, your program should consider all words that are at distance between hd_1 and hd_2 from the target word and pick one of these words at random as the source word. In the hard version of the game, we want the distance between the source word and target word to typically be larger than in the easy version. So if we pick the distance thresholds to be $ed_1 = 5$ and $ed_2 = 10$ for the easy version, we might (for example) pick the distance thresholds for the the hard version to be $hd_1 = 8$ and $hd_2 = 13$. Furthermore, in this case we make no distinction between easy words and hard words. So all words will have weight 0, as in traditional breadth first search.

User interaction [Under construction]

[I will make some small tweaks to this part of the project over the next two days (Mon 4/1).]

Once the program has output the source word s and target word t, it is the user's turn to respond. Ideally, the user will type a sequence of words $w_1, w_2, \ldots, w_{n-1}$, one word per line, such that $s = w_0, w_1, \ldots, w_{n-1}, w_n = t$ forms a ladder from s to t. However, the user may make a mistake and type something that is not a neighbor of the previous word in the ladder. Your program needs to respond to this mistake with an appropriate message.

The program also provides two types of help to the user:

- **Word hints:** In the "easy" version of the game, for some non-negative integer parameter eh, the user can avail of eh number of next word hints. In other words, at any point in the game, and up to eh number of times, the user can ask the program for help with the next word. This is possible in the "hard" version of the game as well, but for a different non-negative integer parameter $hh \leq eh$.
- **Distance hints:** From time to time, the program informs the user how far they are from the target word. This is done randomly at a rate r, which is a parameter, with value being a real number between 0 and 1. For example, if $r = 0.2$, then your program tosses a biased coin that produces heads with probability 0.2 and tails with probability 0.8. This coin is tossed just before the user is about to type a word. If the coin produces heads, then your program informs the user how far (i.e., how many words) they are from the target.

The format of the file `parameter.txt`

In the description above, we see that there are 8 parameters that control the behavior of your programs. Values are provided for these 8 parameters in the file `parameters.txt` in the following format.

```
p = value1
ed1 = value2, ed2 = value3
hd1 = value4, hd2 = value5
eh = value6, hh = value7
r = value8
```

Handling Exceptions

We would like to make sure that any run-time error your program encounters, is handled gracefully. This includes errors that occur when the files your program is expecting to read, e.g., `parameters.txt`, are not found or the user types something totally unexpected. Ideally, your program should try to manage without the files or should exit with a very clear and detailed message about the error. You can use the Python `try except` feature to handle exceptions gracefully.
