# Ladder-Gram Program Documentation

*Steven Phan - s5080436*

*Jake Attard - s5138647*

**Contents**

1. Problem Statement 1

2. User Requirements 2

3. Software Requirements 2

4. Software Design 3

5. Requirement Acceptance Tests 5

6. Detailed Software Testing 6

7. User Instructions 7

## Problem Statement

The goal of this assignment project was to repair an unfinished Python program (otherwise known as word ladder.py) that is used to generate a word ladder based on words provided in the dictionary.txt file that has been supplied. The end goal for this program is that it’ll be able to transform a source word into a target word in the least number of steps. During each step, one letter must be replaced in the previous word each time a step is performed, until that new word has been formed, without changing the positions of the other letters.

The following functionality was also added to the program:

* The program performs as it did prior to the less talented programmer
* Python unittest modules to test functions of the ladder-gram
* All program changes were added to a configuration management tool called github
* The user can add a file which excludes certain words

## User Requirements

The user requirements for the ladder-gram program include:

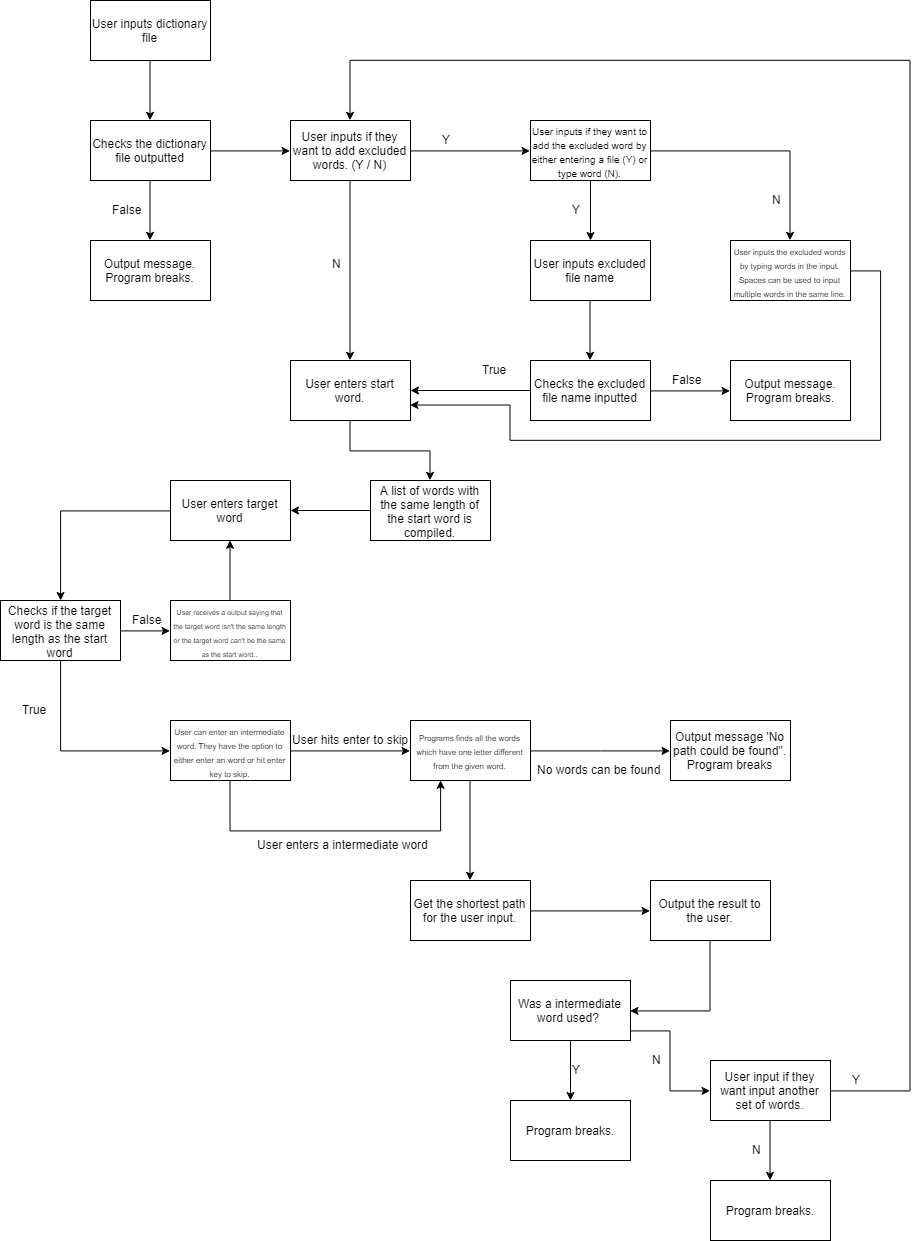
* The dictionary.txt file will be opened and read automatically saving the user time needing to manually input the dictionary.txt file name each time.
* The user will be able to specify an exclude file with a list of words which won’t be used within the start to target word steps.
* All of the files that have been provided MUST be in the .txt format and in order to run a single word must be on each line. This requirement must also be followed for the exclude words text file.
* At the beginning when you run the code, the program will be able to detect and read the dictionary.txt file. From there, the user will be able to either skip or provide the exclude word file. Users will be able to enter any word as the start word and any word for the target word, provided that the words are included in the dictionary.txt file. Numbers or non-alphanumeric characters will not be accepted as the start or target word.

## Software Requirements

1. The user will be able to input the dictionary file.
2. If the dictionary file inputted can’t be found the user will be shown an error message and the program will terminate.
3. A prompt asking the user if they want to include an excluded text file will be shown.
4. The program will only accept the exclude file if it is saved in a .txt file format.
5. If the exclude file name inputted doesn’t exist then the user will be shown an error message and the program will terminate.
6. The user will also be able to input excluded words if they wish to not use a excluded text file.
7. When inputting the start and target words, the program will accept these inputs in the form of a series of alphabetical characters such as ‘lead’. If any type of numerical characters or punctuation such as ‘1234’ or ‘!@#$’, the program will not accept that, and it’ll produce an error
8. The user will also be able if they wish to add an intermediate word.
9. The intentions of this program was that the word ‘lead’ would be able to change to the word ‘gold’ in three steps: (*lead -> load -> goad -> gold*) and the word ‘hide’ would be able to change to the word ‘seek’ in six steps: (*hide -> side -> site -> sies -> sees -> seek*). After making changes to the program, the program will now perform as intended in the first place.
10. When the program decides on the path it takes to get from the start word to the target word, the program will try find the shortest path possible between those two points.
11. If the user enters in any input errors, those possible user input errors will be handled appropriately.

## Software Design

### High Level Design – Logical Block Diagram



### List of all functions in the software

**dictionaryListFile:**

The purpose of this function is to open and read from the dictionary.txt file provided. All relevant error handling has been provided if the dictionary.txt file can’t be opened or found. The function then iterates through all the lines in the dictionary.txt file and appends it to a list. Error handling has also been provided for if the dictionary.txt file is empty and no words are found. The fileLines are then returned.

No input parameters or side effects where used within the dictionaryListFile function.

A return value called fileLines was used to return all the words in the dictionary.txt file in a list.

**excludedListFile:**

The purpose of this function is so users can input there exclude words file. All relevant error handling has been provided if the inputted file can’t be found or opened. Just like the dictionaryListFile function, the excludedFile function then iterates through all the lines of the inputted file and appends it to a list. Error handling has been done so if the excluded file is empty then users will receive a prompt that the file inputted is empty. The excludedFileLines are then returned.

No input parameters or side effects where used within the excludedFile function.

A return value called excludedFileLines was used to return all the words in the file inputted by the user into a list.

**same:**

The same function takes two string parameters called the item and the target. A list is generated with each individual characters which occur in the same position in the string. The length of the list called len is then returned. This compares how similar the current word is to the target word.

The same function takes in two input parameters called item and target. Both of these parameters are a String with the item being the current word in the updated step. The target word is inputted by the user.

A return value called len was used to return the number of characters in both the item and target word. The len data type is an integer.

**build:**

The purpose of the build function is that it generates a list of words from the dictionary. The dictionary matches a pattern given to it in the form of a string. In order to denote a wild card, which means it can be any character, the ‘.’ character is used for this example to denote a wildcard.

These are the input parameters that have been used:

**findingWords -** ‘findingWords’ is a list. It’s a list of all the potential words that have already been identified previously, for the next step in the word ladder.

**words -** ‘words’ is a list. Like the name and definition itself, it’s a list of words that can be attained from the dictionary file.

**pattern -** ‘pattern’ is a string. The string can potentially be matched by the function, against the contents of ‘words’. Like mentioned earlier, the ‘.’ character is used whenever you want to denote a wildcard.

**seen -** ‘seen’ is a dictionary. Within this dictionary, there are words whose values are boolean; meaning True or False. With each word is a key. Each key that is a word that has already been seen by the program will not be able to be used again.

In terms of the side effects, there are none.

Return value:

**List -** The list is a list of possible words that is found in the provided dictionary. Essentially, all of these possible words will try to match the pattern specified.

**Find:**

The find function discovers the words in the path and finds the shortest path.

The find function has five input parametres:

-word

-words

-seen

-target

-path

The side effects include selecting the words and path, appending them.

The return value can either be True or False.

### List of all data structures in the software (eg linked lists, trees, arrays etc)

For each data structure in the list the following information is provided:

1. Type of structure (tree, list etc),
2. Description of where and how it is used
3. List of data members, and what each one is for do
4. List of functions that use it

**path:** The data structure for the path is the current semi-tree of the word ladder.py and it consists of a list of strings. The function that is involved in this is the ‘find()’ function. It’s used when the final item is being used for the up-to-date recursion of the ‘find()’ function. If no path can be found while using this function, then it’ll be removed.

**seen:** The data structure for seen is the dictionary and it consists of strings as boolean values and keys. All of the words of the working dictionary that the program has already encountered in the past will be contained within the dictionary. This happens each time the find function and all of its recursions is called.

**startWord:** The data structure for startWord is the dictionary and it also consists of strings as boolean values and keys. This dictionary also contains blacklisted words that have already been seen previously in another path determined. The dictionary representing the original state of ‘seen’ before the find function is called, the ‘seen’ dictionary above is actually copied from this variable. Because ‘seen’ needs to be reset before each path can be found, this is important to note when it comes to finding multiple paths. As for the functions, it’s only used in the main code.

**words:** The data structure for words is a list and it’s consisted of a list of strings. The dictionary that has provided the list of words have provided words that has a length equal to the length of the start of the target word. This is also the dictionary from where valid words are appended to ‘findingWords’. As for the functions, the functions that are involved are: build() and find()

**findingWords:** The data structure for findingWords is a list that consists of two states: a list of strings and a list of tuples. Firstly the list starts off as a list of potential words that can be next in line in the path (e.g. different to the target word in the path by just one letter). Eventually, it becomes a list of tuples because the first value has similarities of the next potential word of target word, which is an integer. The second value being the potential next word, which is a string. As for the functions, the functions that are involved are: build(), find(), and same()

**fixedIndexes:** The data structure for fixedIndexes is a list and it consists of a list of integers. It’s a list of indexes where essentially, for both the current final word in the path and the target word, there is a matching letter for both. As for the functions, only the find() function is involved.

### Detailed Design

**dictionaryListFile:**

try:

dictionaryFileName = input(“Enter dictionary file name: “)

file = open(dictionaryFileName)

This opens the dictionary file inputted by the user

except:

print(“The dictionary file can’t be found. Please check if you have the correct .txt file.”)

exit(“Please re-run the program and try again!”)

The exit quits the program.

dictionaryFileLines = file.readlines()

This reads the dictionary file inputted by the user and generates a list of lines.

for dictionaryFileLine in range(length of the dictionaryFileLines):

strip is used to remove the character in the new line.

if the line is blank:

remove the blank line from the dictionaryFileLines.

if the length of the dictionaryFileLines is empty or 0:

print(“The dictionary file you entered is empty.”)

exit(“Please re-run the program and try again!”)

The exit quits the program.

return dictionaryFileLines

**excludedListFile:**

try:

excludedFileName = input(“Enter excluded file name: “)

file = open(excludedFileName)

This opens the excluded file inputted by the user

except:

print(“The excluded file can’t be found. Please check if you have the correct .txt file.”)

exit(“Please re-run the program and try again!”)

The exit quits the program.

excludedFileLines= file.readlines()

This reads the excluded file inputted by the user and generates a list of lines.

for excludedFileLine in range(length of the excludedFileLines):

strip is used to remove the character in the new line.

if the line is blank:

remove the blank line from the excludedFileLines.

if the length of the excludedFileLines is empty or 0:

print(“The excluded file you entered is empty.”)

exit(“Please re-run the program and try again!”)

The exit quits the program.

return excludedFileLines

**find:**

findingWords = list()

A list of indexes are created based on matching letters between the final and target word

for x in the index of the range of the word length:

if the index of the word is not in the fixedIndexes:

Create a list of dictionary words differing on the index word.

if there is not any more potential next words:

return False

findingWords is generating a new list of words which are similar to the target word. reverse = True means all words being sorted in a reversed order.

for match, item in findingWords:

if the match >= to the length of the targetWord -1:

if the match is == to the length of the target word -1:

path.append(item)

return True

add the seen item to the dictionary = True

for match, item in findingWords:

the word is then appended to the path

if find is True:

return True

The word is then removed from the path.

### Configuration management and version control

Provide detailed description on how version control tools are used during the development process. In addition to your description, you could append a log or screenshot of your version control activities.

## Unit Tests

|  |  |  |  |
| --- | --- | --- | --- |
| **No** | **Test Case** | **Expected Results** | **Actual Results** |
| **1.0** | **DictionaryListFile() Function** |  |  |
| 1.1 | Testing if the dictionary file inputted does not exist. | SystemExit and FileNotFound exceptions are raised. | Exception Raised |
| 1.2 | Test if the dictionary file inputted is a empty string. | SystemExit exception is raised. | Exception Raised |
| **2.0** | **ExcludedListFile() function** |  |  |
| 2.1 | Testing if the excluded file inputted does not exist. | SystemExit and FileNotFound exceptions are raised. | Exception Raised |
| 2.2 | Test if the excluded file inputted is a empty string. | SystemExit exception is raised. | Exception Raised |
| **3.0** | **Same() Function** |  |  |
| 3.1 | Testing words (‘lead’, ‘gold’). | Returns ‘ok’ if the matching letters is 1. | Returns ok – the test was succesful. |
| 3.2 | Testing words (‘hide’, ‘seek’). | Returns ‘ok’ if the matching letters is 0. | Returns ok – the test was succesful. |
| 3.3 | Testing words (‘goal’, ‘load’). | Returns ‘ok’ if the matching letters is 2. | Returns ok – the test was succesful. |
| 3.4 | Testing unsuccessful words (‘bike’, ‘car’) | Returns ‘ok’ if the word lengths aren’t the same. | Returns ok |
| 3.5 | Testing unsuccessful words (‘truck’, ‘run’) | Returns ‘ok’ if the word lengths aren’t the same. | Returns ok |
| **4.0** | **Build() function** |  |  |
| 4.1 | Testing words (‘hide’, ‘seek’) | Returns ‘ok’ with the words building successfully. | As expected |
| 4.2 | Testing unsuccessful words | Returns ‘ok’ due to it being false. | As expected |
| **5.0** | **Find() function** |  |  |
| 5.1 |  |  |  |
| 5.2 |  |  |  |
| 5.3 |  |  |  |

## Requirement Acceptance Test

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Software  Requirement No** | **Test** | **Implemented (Full /Partial/ None)** | **Test Results (Pass/ Fail)** | **Comments (for partial implementation or failed test results)** |
| 1 | Users can input the dictionary file name. | Full | Pass | Complete |
| 2 | Users can input the excluded file name. | Full | Pass | Complete |
| 3 | Users can input the excluded words in without a file needed. | Full | Pass | Complete |
| 4 | Error handling has been done so if the input files don’t exist the program will end. | Full | Pass | Complete |
| 5 | If the start word and target word do not match in length the user will receive a error message saying that the target word needs to be the same length as the start word. | Full | Pass | Complete |
| 6 | Users are able to add an intermediate word for example ‘lear’ within ‘hide’ and ‘seek’. | Full | Pass | Complete |
| 7 | Ladder-gram program changes ‘lead’ to ‘gold’ in 3 steps (lead->load->goad->gold. ‘hide’ changes to ‘seek’ in 6 steps (hide->side->site->sits->sies->sees->seek) | Full | Pass | Complete |
| 8 | Ladder-gram program displays the path between the start, intermediate (if added by the user) and target word. | Full | Pass | Complete |
| 9 | If the target word can’t be found the program will display to the user “No path was found”. | Full | Pass | Complete |
| 10 | If the user doesn’t add a intermediate word and only puts in a start and target word. Once the path has been displayed the user will have the option to input another set of words. | Full | Pass | Complete |

## User Instructions

1. Open up the file ‘word\_ladder.py’
2. Run the program in pyCharm or Visual Studio Code in terminal
3. When prompted for the dictionary file name, enter in the dictionary file name you’d like to use and then press enter or return on the keyboard
4. Once the dictionary file name has been entered, the blacklist will appear and ask if you want to exclude any words. You now have the option to enter ‘y’ or ‘n’ with ‘y’ meaning Yes and ‘n’ meaning No.
5. If you enter ‘y’, it’ll prompt you and ask if you would prefer to exclude the word/s via a file or without a file. If you would like to exclude the word/s via a file, you enter in ‘y’, or if you would like to exclude the word/s without the use a file, you would enter in ‘n’
6. If you enter ‘n’, it’ll ask you to enter in a start word. Once the start word has been chosen, please press enter or return. The start word must be a word from the dictionary.
7. If you selected ‘n’ because you **didn’t** want to exclude any words, when prompted, it’ll ask you to enter in a target word. Once the target word has been chosen, please press enter or return. The target word must be a word from the dictionary.
8. When prompted, it’ll ask you to enter in an intermediate word. If you don’t wish to enter in an intermediate word, you can press enter or return to skip.
9. This will then output the amount of steps and the steps it took to get from your start word to your target word.
10. If you selected ‘y’ because you **wanted** to exclude words, it’ll prompt you and ask if you would like to exclude words via a file by entering in ‘y’, or if you would like to exclude words without the use of a file by entering in ‘n’.
11. If you selected ‘y’, it’ll ask for your excluded file name. If you selected ‘n’, it’ll ask you to input the word/s you would like to exclude. There would have to be a space in between each word, and it all has to be on the same line
12. When prompted, it’ll ask you to enter in a start word. Once the start word has been chosen, please press enter or return. The start word must be a word from the dictionary.
13. When prompted, it’ll ask you to enter in a target word. Once the target word has been chosen, please press enter or return. The target word must be a word from the dictionary.
14. When prompted, it’ll ask you to enter in an intermediate word. If you don’t wish to enter in an intermediate word, you can press enter or return to skip.
15. This will then output the amount of steps and the steps it took to get from your start word to your target word.