# 

# 

# **2810ICT— Software Technologies**

# **Word Ladder Assignment**

# 

*Sherief Hammad & David Saele*

*s5093523 & s5071539*

Contents

1. Problem Statement.......................................................................................... 2

2. User Requirements.......................................................................................... 2

3. Software Requirements................................................................................... 3

4. Software Design............................................................................................... 3

5. Requirement Acceptance Tests....................................................................... 9

6. Detailed Software Testing.............................................................................. 10

7. User Instructions............................................................................................ 12

## **1. Problem Statement**

A ladder-gram is a program that convert a user’s input word and converts it to a targeted end word in the least amount of steps. Each step, one letter from the previous word is replaced so that a new word is formed without changing the position of the other characters. All words in each step must be a part of the user’s provided dictionary. The program should handle all possible user input errors. The user should also be able to provide a list of words that the program cannot be used within the steps from the input word to the end word.

## **2. User Requirements**

The user requirements for the program are as follows:

* The user must input a valid dictionary text file which provide the words that the user can choose from.
* The user enters either L for longest or S for shortest route from the start word to the target.
* The user enters a start and end word, which has to be a word from the dictionary file.
* Both words must not contain any numbers or special characters.
* After the program has returned the number of steps and the path between the words, the user can try again with two other words or stop.
* The user should be able to input a list of words which the program cannot use in the path between the start and end word.

## **3. Software Requirements**

The software requirements for the program are as follows:

* The program should accept one text file as an argument for the dictionary, which provides the list of words the user can choose from.
* The program requests the user if the shortest or regular path to the word is wanted.
* If the user inputs an unknown file or an invalid argument, the program asks the user to try again with a valid file.
* The user gives the program a start word and an end word, which the program finds a path between them. The program then prints the number of steps from the start to the end word and all the words in the path in a list.
* If the start word or end word does not appear in the dictionary, the program asks the user to try other words.
* If the two words does not have a possible path between them, the program shall display: “No path found”.
* After the program has returned and displayed a path or no path found, the program asks the user a yes or no question depending on whether or not the user wish to try to other words or end the program.
* The program should also give the option to the user to choose some words that are not allowed to use in the path.

## **4. Software Design**

## High Level Design – Logical Block Diagram

# 

## List of all functions in the software

#### fileInputValidation / initialWordValidation(word:str) / targetWordValidation(word:str):

This function much like the rest of the validation based functions, is purposed with ensuring that the user inputted text is as expected and not utilising any special characters. This file affects the dictionaryFile variable that stores the opened file to then be read line by line. The function when successful at it’s try block will return the opened file on ‘read’, to whatever variable was passed in with it.  
Both the Initial and target word validation work almost as mirrors of each other, though the main difference being that the targetWordValidation compares the length of the word passed in with the starting word. Because of this it is required that the targetWordValidation function utilises a global version of the variable startWord. In both cases, whenever there’s an issue with the user input, it’ll require them to retry the input until it works. When it works it returns the word, having passed the validation.

#### ChoosePathOption(option:str):

The ChoosePathOption takes an input recorded as ‘path’ and declares another variable called ‘pathChosen’ with it being passed into the ChoosePathOption function. Due to the possibility of issues occuring when being handled, we have to, at some point before we work with the input, make the input lowercase (.lower). Once it’s passed into the function, it is tested for it’s validity much like the initial and target word tests, however unlike those, in the case of the user inputting either of the 2 passable answers, it returns either a True or a False (False = ‘L’ong, True = ‘S’hort).

#### BuildListOfPatternWords(pattern:str, wordsOfSameLength:list, seenWords:dict, potentialSolutionPath:set):

This function is one of two functions that require the passing in of four or more parameters when calling the function. This function undertakes the process of adding to the second parameter which is a list. As it utilises the regex ‘.’ character, it is able to search through the use of a for to find all other words within the word list that can be reached from the current word. For example, for word S E E K, it’s - E E K -> S - E K -> S E - K -> S E E -. This returns a List of patterned words.

#### same(item:str, targetWord:str):

The same function uses two strings, item and targetWord that need to be passed in (though their names don’t have to be that) it simply tests what characters in the two words are the same. Basically, checking for common letters. This has no effect on any global values though does return an integer of how many letters the two passed in words have in common.

#### FindSolutionPath(word:str, words:list, seenWords:dict, targetWord:str, instancePath:list)

This function is the largest out of any of them and also requires the BuildListOfPatternWords function to be used at certain points. The FindSolutionPath actively builds a list utilising the patternWords function to have access to all variations within the supplied dictionary. Much like other functions, this also utilises

the seenWords dictionary to ensure no double ups occur through the program running. Depending on the result of the ChoosePathOption (whether True or False), this will adjust depending and give the results for either the long form or the shortest possible path from startingWord to targetWord. on completion, depending on if the the lists words had a path to each other, the function will pass back a True along with appending the targetWord to the end of the newly created and filled out path. This function declares both pathChosen and rareLetters as global variables prior to the functions process, to ensure that both the outside of the function and in are able to fully manipulate and view the aforementioned variables.

## List of all data structures in the software

#### wordsOfSameLength:

This data structure is a list and it’s, as the name suggests, purposed with storing all words within the dictionary of the same length as both the target/initial word. This filters all other useless words out to not make the process convoluted. This list is utilised within both the FindSolutionPath and the BuildListOfPatternWords at times though is only passed into the BuildListOfPatternWords through the called within the FindSolutionPath.

#### dictionaryLines:

This list is a breakdown line by line of the previously declared dictionaryFile textIOWrapper that handles the user inputted files opening. Once this list is created and filled up, it’s used to then iterate through each word and in the next steps of filtering, and any words that fit the validation are then appended to the wordsOfSameLength list.

#### rareLetters:

The rareLetters List is used to contain any unlikely letters that would appear in words, with those filtered out it ensures the path is a lot shorter. As to not rely on any words with uncommon overlaps. The rare letters used are not programmatically added and are defined at the bottom. This list is utilised within the main FindSolutionPath function to filter rareLetters and depending on if the user chose ‘L’ong or ‘S’hort form, if they chose ‘L’, it’ll remove the words from the filter.

#### SeenWords:

This dictionary uses each word previously added as the key whilst the value is a boolean of whether or not it’s been seen before. This can then be programmatically checked to ensure no double ups occur. It’s used multiple times within both FindSolutionPath and then handed into BuildListOfPatternWords by the previous function.

#### finalPath:

This list is used as the final storing place of the decided upon path. Once all things are said and done. It’s utilised within the call and use of the FindSolutionPath function, in which returns a True or False depending on the current word being assessed. If the current word meets the criteria, it gets added to the finalPath to then be printed once it’s completed.

## Detailed Design

create Function(fileInputValidation()):

**try if** possible:

assign fileName = the input handed **in** by user(**"Please type the file name of your dictionary (default = dictionary.txt) : "**)

**return** the opened file back to the assigning variable(**'read'**)

**except if** FileNotFoundError occurs:

print to console(**"File is missing, double check your spelling or directory.\n"**)

restart input validation

create Function **and pass in** set variables **with** datatypes(targetWordValidation(word)):

make **global** variable startWord

**if** the length(word) **is not** = len(startWord) :

word variable **is** = the user input **for**(**"Both the initial word and the target word must be the same length, please try again with said parameters"**)

**else if** the word **is** a digit():

word variable **is** = the error, **try** again message(**"Target word must not be numerical, please type a new target word : "**)

**else if** the word **is** only letters():

replace the word(**" "**, **""**)

**return** the current word

create Fucntion(FindSolutionPath **and pass in** set variables **with** datatypes(word, words, seenWords, targetWord, instancePath)) :

make **global** variable pathChosen

make **global** variable rareLetters

variable tempList **is** = list

**for** looping **from** 0 to the length of the word:

tempList adds the result of function(BuildListOfPatternWords(currentWord **in** loop[:up to the loops current point(i)]

+ **"."** + currentWord **in** loop[**from** the loops current point (i) + 1: onward], listOfAllSameSizeWords, PreviouslyseenWords, tempList))

**if** the length of (tempList) **is** == 0 :

**return** boolean **False**

**if** the pathChosen **is** == **True**:

tempList **is** = sort the result of([(Function same(w, targetWord), w) **for** w **in** tempList], make it reversed = **True**)

**else if not**:

empty rareLetters list

tempList list = the sorted result of([(Function same(w, targetWord), w) **for** w **in** tempList])

**for** each (varName, item) **in** tempList list:

**for** each of the letters **in** the rareLetters list:

**if** the letter **is in** item:

remove **from** tempList list((the varName, **and** the item))

**if** the varName **is** >= the length(of the targetWord) - 1:

**if** the varName == length (of the targetWord) - 1:

add the item to the instancePath List

**return** boolean **True**

add the item to seenWords[item] = **and** give it value **True**

**for** each(varName, item) **in** tempList list:

add the item to instancePath list

**if** the Function FindSolutionPath(item, words, seenWords, targetWord, instancePath) returns **True**:

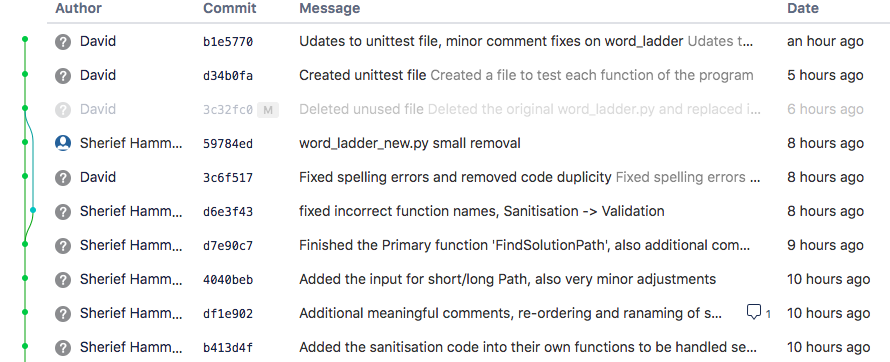
**return** boolean **True**

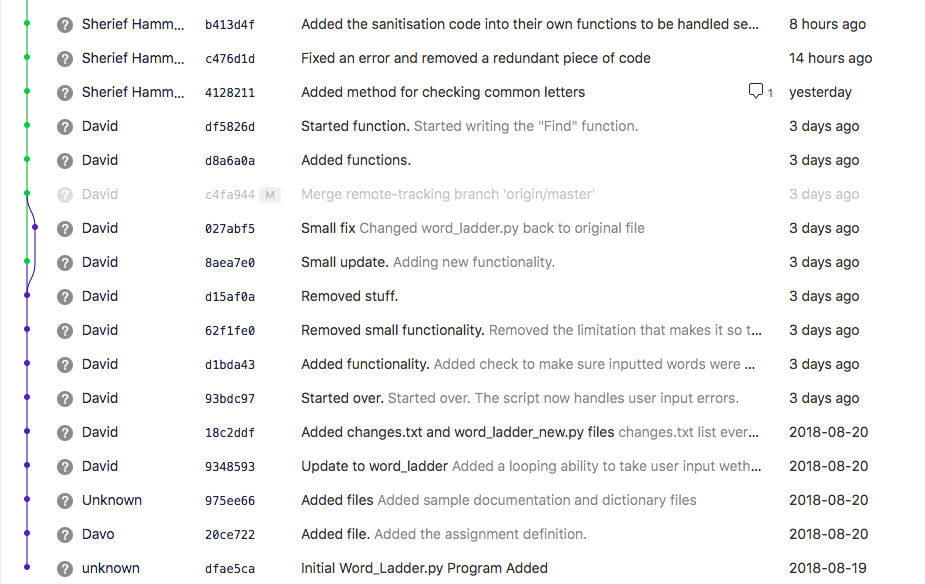
remove **and return** the final item **in** instancePath list (pop())

## Configuration management and version control

For version control, Bitbucket was used as a tool during the development process. Firstly, the original source code was uploaded as a starting point for both programmers. When modifying and editing the program to fulfill the requirements, a new python file was created to work on. The reason for that was to easily go back and compare the new program to the old one. For every new and updates to old function, we committed the work so that the other programmer could view changes and work from there. For every commit, a brief description of the updates was provided so that it was clear what had been done. Both programmers worked on the same branch and same file, which can be problematic when working simultaneously. But for this project, we worked mostly at different times, which eliminated that problem. The benefit of working on the same file on the same branch is that we avoided potential problems when merging the branches.

The images below provide an overview of every commit made, with a brief description of what it does, in the development process of the project.





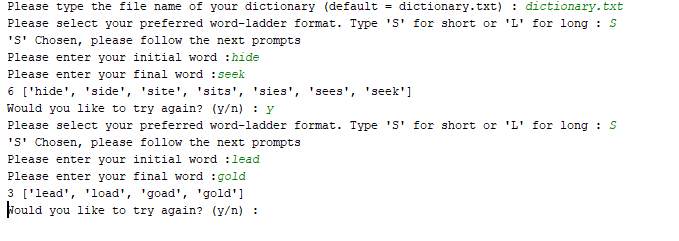
## **5. Unit Tests**

|  |  |  |  |
| --- | --- | --- | --- |
| No | Test Case | Expected Results | Actual Results |
| **1.0** | **File Input Validation** |  |  |
| 1.1 | Wrong filename input | Error message asking the user to try another file name input. | Error message asking the user to try another file name input. |
| **2.0** | **Initial Word Validation** |  |  |
| 2.1 | Numerical character in the word | Error message asking the user to input a word without numerical values. | Error message asking the user to input a word without numerical values. |
| **3.0** | **Target Word Validation** |  |  |
| 3.1 | Target word is of same length as start word | Error message stating that the words have to be of same length, try another input. | Error message stating that the words have to be of same length, try another input. |
| **4.0** | **ChoosePathOption** |  |  |
| 4.1 | User input longer than 1 character | Error message stating that the user must input either S or L for length of path. | Error message stating that the user must input either S or L for length of path. |

## **6. Requirement Acceptance Test**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Software**  **Requirement No** | **Test** | **Implemented (Full /Partial/ None)** | **Test Results (Pass/ Fail)** | **Comments (for partial implementation or failed test results)** |
| 1 | Accept one text file as an argument for the dictionary, which provides the list of words the user can choose from. | Full | Pass |  |
| 2 | The program request the user if the shortest or regular path to the word is wanted. | Full | Pass |  |
| 3 | Asks user for to try another file name if an invalid one is inputted. | Full | Pass |  |
| 4 | Accept a start and an end word and finds a path between them. Prints the number of steps between them and all the words in the path as a list. | Full | Pass |  |
| 5 | If the start word or end word does not appear in the dictionary, the program asks the user to try other words. | Full | Pass |  |
| 6 | If the two words does not have a possible path between them, the program shall display: “No path found”. | Full | Pass |  |
| 7 | After the program has returned and displayed a path or no path found, the program asks the user a yes or no question depending on whether or not the user wish to try to other words or end the program. | Full | Pass |  |
| 8 | The program should also give the option to the user to choose some words that are not allowed to use in the path. | None | Fail | Was not implemented in the program. |

## Below is an image which shows that the program runs as it should with lead--> gold and hide→ seek as examples:



## **7. User Instructions**

* Run the program in Anaconda prompt with python.
* Enter either dictionary.txt(default) or any other text files as a file name
* Choose either ‘S’ for short or ‘L’ for long to decide whether you want the shortest or original path between the words.
* Enter start word
* Enter target word
* If every input is entered correctly, the program will find the path between the words, if there is not a path, “No path found” will be displayed.
* Then enter Y if you want to continue with other words or enter N if you want to exit the program.