# Word Ladder Assignment

*Ben Xu* - *s5057948*

*Jack Leyden -* s*5052939*

**Contents**

[Ladder-Gram Assignment 1](#_Toc491066310)

[1. Problem Statement 2](#_Toc491066311)

[2. User Requirements 2](#_Toc491066312)

[3. Software Requirements 2](#_Toc491066313)

[4. Software Design 1](#_Toc491066314)

[High Level Design – Logical Block Diagram 1](#_Toc491066315)

[List of all functions in the software. 1](#_Toc491066316)

[List of all data structures in the software. (eg linked lists, trees, arrays etc) 1](#_Toc491066317)

[Detailed Design – Pseudocode for all non-standard and non-trivial algorithms that operate on data structures 2](#_Toc491066318)

[Configuration management and version control 5](#_Toc491066319)

[5. Unit Tests 1](#_Toc491066320)

[6. Requirement Acceptance Test 2](#_Toc491066321)

[7. User Instructions 4](#_Toc491066322)

## Problem Statement

The goal of this project was to create a program with Python 3 that could find the shortest path between two words chosen by the user. The path is found by iteratively changing the original word one character at a time to identify the shortest path. The path with the least number of letters changed before arriving at the final word is chosen.

## User Requirements

The following outlines the user requirements for the program:

* The user shall be able to enter the dictionary file name.
* The user should be able to enter start and target words
* The user should be able to enter a list of words that they want to have exempted from the search.
* User should be able to choose to find the shortest path for the program to navigate to the final word.

## Software Requirements

The following outlines the software requirements for the program:

1. The program shall accept a file with the condition that it is a valid dictionary.
2. The program shall display an appropriate message if a file does not exist or if a file name is invalid.
3. The program shall display a message if the word inputs do not match.
4. The program must display the path between the source and target words.
5. The program must have input fail safes that check and validate whether the user’s input can be run.

## Software Design

### High Level Design – Logical Block Diagram



Validate input (regular expr)

Get User input (dictionary)

Get User input (start)

Get User input (end)

Get User input (banned words)

Output Message:

Validate input (regular expr)

Validate input (regular expr)

Check if it is a file or directory

True True

Validate input (regular expr)

Valid Valid



false neither not valid not valid valid

Get User input (shortest path)

Output Message:

Output Message:

Output Message:

Output Message:

Output Message:



Run:

Search words for path

Output

Result:

Validate input (regular expr)

valid

### List of all functions in the software.

For each function in the list the following information is provided:

Find():

Describe: this function identifies the words in the path and finds the shortest path to the destination.

Inputs: word, words, seen, target, path

Side effects: words are selected and appended to path

Return value: true or false

Same():

Describe: identifies what characters in the current word is the same as the target word.

Inputs: item, target

Side effects: return int value of number or same characters.

Return value: return int value of number or same characters.

Build():

Describe: is part of a build process that builds a list of words that are related to the current word/item.

Inputs: pattern, words, seen, list

Side effects:

Return value: returns a single word that is related to seen word but is not the same as the previously seen word.

Main():

Describe: All the code that runs outside the function is unit tested in here. All inputs and links to other functions for the overall running of the software are stored in here.

Inputs: fname, start, target, forbidInput, short, i (match another word)

Side effects:

Return value: path

### List of all data structures in the software.

dictionary.txt:

* Type: file
* Description: contains a full dictionary
* Members: fname
* Functions: main()

Words:

* Type: List
* Description: Builds a list of the words from the dictionary file
* Members: words
* Functions: find(), and build()

forbiddenList:

* Type: List
* Description: Builds a list of the words that must not be used in the path
* Members: none
* Functions: main()

list:

* Type: List
* Description: finds the words in words that are related to the current word selected
* Members: list
* Functions: find(), and build()

path:

* Type: List
* Description: contains the final path from start to target
* Member: path
* Functions: find()

seen:

* Type: dictionary
* Description: stores all the words the program has looked through so that the program doesn’t loop through and check the same word.
* Members: seen
* Functions: find(), and build()

### Detailed Design – Pseudocode for all non-standard and non-trivial algorithms that operate on data structures

#create fail safes try except

while True:

{

While True:

{

try:

{

Ask user for dictionary location

Open that location

Read the file

break

}

Except:

{

Print Please enter a valid file

Continue

}

}

#Ask for words

while True:

{

Ask user for start word and convert to lowercase

Create a blank list called words

for line in lines: #Regenerate word list

{

Clear blank spaces from the right side of the words in dictionary

if (word in dictionary is same length as the start word length):

{

Add word to list words

}

Ask user for target word and convert it to lowercase

if (length of target is not the same as the length of start) or (length of target is 0 or length of start is 0) or (target word == start word):

{

Continue

}

Break

}

}

#Ask for forbidden words

Create forbiddenList as a blank list

while True:

{

Ask user for a word they would like the path to not include

if (start or target words is equal to forbidInput):

{

print("Cannot equal the target or start word")

Continue

}

elif (forbidInput not equal to nothing): #will always run until user inputs empty string

{

forbiddenList.append(forbidInput)

}

Else:

{

print("Forbidden words: "+the list of accumulated forbidden words))

break

}

#Remove forbidden words from list words

for s in forbiddenList: #For every forbidden word…

{

for d in words: #check every word in the dictionary…

{

if (d == s): #and if it's the same as a forbidden word, remove it

{

Remove word in words list that matches the word in the forbiddenlist

}

}

}

}

#Ask if user wants shortest path

while True:

{

Ask user if they want the shortest path

if (input == yes) or (input == "n"):

{

Break

}

Else:

{

print('Invalid input!')

}

path = list containing start

seen = dictionary containing start:true

if find(start, words, seen, target, path):

{

Add target word to path

print(path length - 1, and path)

}

Else:

{

print("No path found")

}

Ask user if they want to find another path #Ask if user wishes to restart program

if (input == yes):

{

print('\n') #newline for neatness

continue #continue main loop

}

Else:

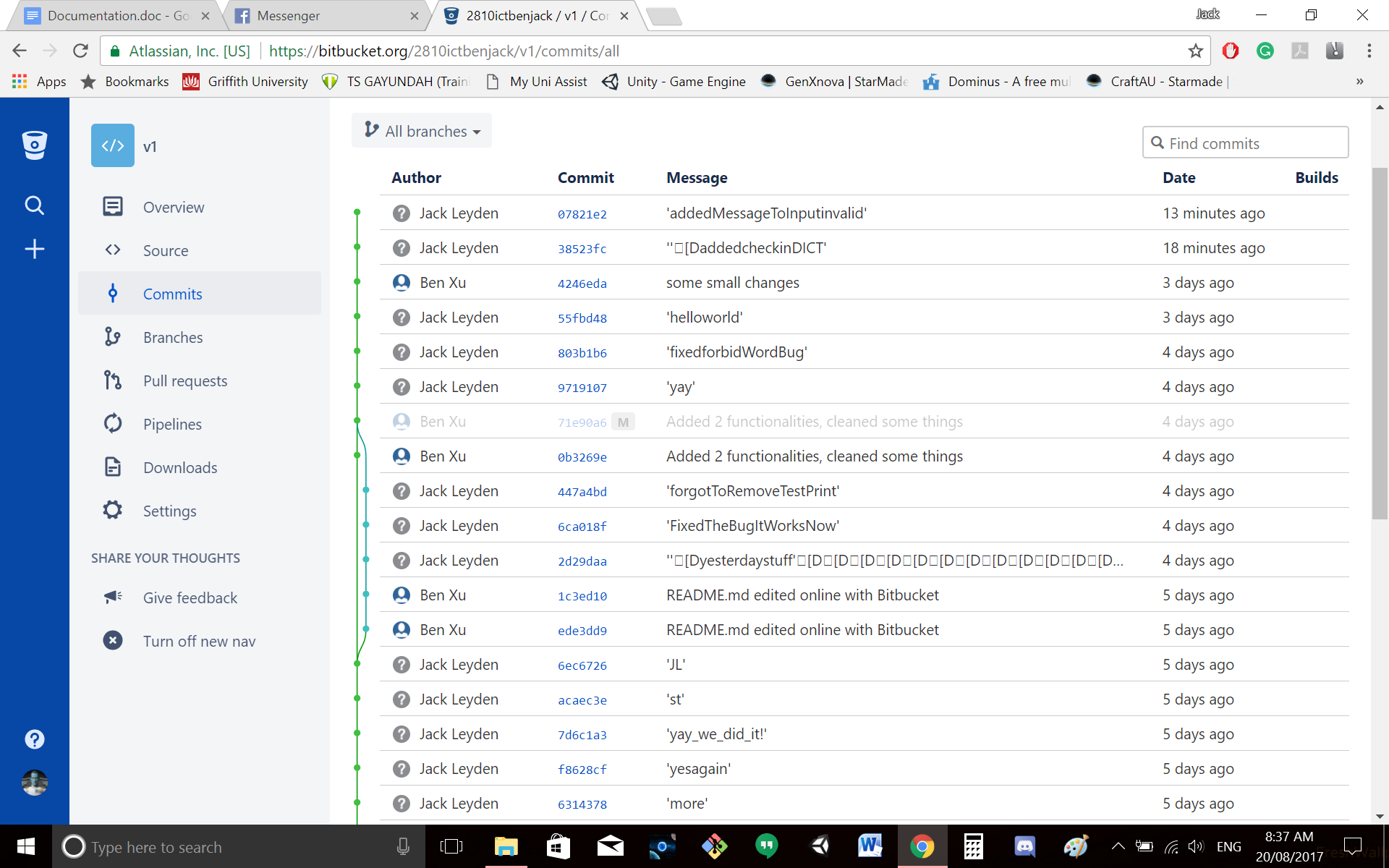
{

exit

}

### Configuration management and version control

During the creation of this project bitbucket was utilised to allow multiple users to have access to the latest version. The files were updated when major changes were created. These changes would be labelled as a new file with the version on the title. This was to protect the original code in the event of a crash or break in the new version.



## Unit Tests

To test the code, the *unittest* module was used.

The unit test code is separate from the code to be tested. The code to be tested is a copy of the original code, modified to allow testing.

|  |  |  |  |
| --- | --- | --- | --- |
| **No** | **Test Case** | **Expected Results** | **Actual Results** |
| **1.0** | **File functions** |  |  |
| 1.1 | Inputted file does not exist | Exception handled, error displayed and looped | As expected |
| **2.0** | **Word inputs** |  |  |
| 2.1 | Target word input is same as start word | Exception handled, error displayed and looped | As expected |
| 2.2 | The y/n input is upper case | Program reads lower-case version of the letter and continues | As expected |
| 2.3 | The y/n input is lower case | Program continues as usual | As expected |
| **3.0** | **Same() function** |  |  |
| 3.1 | Test Same(‘hide,’seek’) | Returns amount of matching letters (0) | Returns 0 as expected |
| 3.2 | Test Same(‘lead,’gold’) | Returns amount of matching letters (1) | Returns 1 as expected |
| 3.3 | Test Same(‘seek,’seal’) | Returns amount of matching letters (2) | Returns 2 as expected |
| 3.4 | Test Same(‘presumptuous’, ‘unfamiliarly’) | Returns amount of matching letters (0) | Returns 0 as expected |

## Requirement Acceptance Test

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **User  Requirement No** | **Test** | **Implemented (Full /Partial/ None)** | **Test Results (Pass/ Fail)** | **Comments (for partial implementation or failed test results)** |
| 1 | The user shall be able to enter the dictionary file name. | Full | Pass |  |
| 2 | The user should be able to enter start and target words | Full | Pass |  |
| 3 | The user should be able to enter a list of words that they want to have exempt from the search. | Full | Pass |  |
| 4 | User should be able to choose to find the shortest path for the program to navigate to the final word. | Full | Pass |  |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Software  Requirement No** | **Test** | **Implemented (Full /Partial/ None)** | **Test Results (Pass/ Fail)** | **Comments (for partial implementation or failed test results)** |
| 1 | The program shall accept a file with the condition that it is a valid dictionary. | Full | Pass |  |
| 2 | The program shall display an appropriate message if a file does not exist or if a file name is invalid. | Partial | Pass | Only shows error if the file doesn’t exist |
| 3 | The program shall display a message if the word input lengths do not match. | Full | Pass |  |
| 4 | The program must display the path between the source and target words. | Full | Pass |  |
| 5 | The program must have input failsafes that check and validate whether the user’s input can be run. | Full | Pass |  |

## User Instructions

* Open the file V2(word\_ladder).py in python 3.6
* Run the program with python 3.6
* Enter the valid location of the directory.
* For existing files details of each file will be displayed.
* For invalid file names an error message will be displayed.
* For files that do not exist an error message will be displayed.
* Insert the start word
* Insert target word
* Error message when word length is not equal
* Enter banned words
* Error if banned words is equal to start or target.
* Insert y for yes or n for no on whether you wish to have the shortest path.
* Error if not one of these is entered
* The output will then display on the shortest path between the two words.
* Insert y if you wish to find another path or anything else if you wish to quit.