Joonyoung Kim (s5120372) & Krister Blanch (s2680202)

Scripting, Testing and Version Control

7810ICT – SOFTWARE TECHNOLOGIES

Contents

[1. Problem Statement 2](#_Toc523112481)

[2. User Requirements 2](#_Toc523112482)

[3. Software Requirements 2](#_Toc523112483)

[4. Software Design 3](#_Toc523112484)

[High Level Design – Logical Block Diagram 3](#_Toc523112485)

[List of all functions in the software 4](#_Toc523112486)

[List of all data structures in the software (eg linked lists, trees, arrays etc) 5](#_Toc523112487)

[Detailed Design 7](#_Toc523112488)

[Configuration management and version control 8](#_Toc523112489)

[5. Unit Tests 9](#_Toc523112490)

[6. Requirement Acceptance Test 10](#_Toc523112491)

[7. User Instructions 12](#_Toc523112492)

[8. Amendment 1. Changelog for 3.1 to 3.2 12](#_Toc523112493)

[Pseudocode for this function 12](#_Toc523112494)

## Problem Statement

The purpose of the assessment is to create a program that allows a user to find the shortest possible path between two words, by changing a single letter at a time. A word could only be changed into a word that was included within a supplied dictionary. A user could also include a number of words that a word could not change into. Lastly, a user could ask for all possible paths, not just the shortest.

To achieve this, a breadth-first-search algorithm was used. The nature of the algorithm requires the iterations to go through siblings before children. For example, if a user wanted to go from ‘lead’ to ‘gold’, it would iterate through all the words that ‘lead’ could change into, before moving onto the next layer.

## User Requirements

* The user can only enter one dictionary file
* User shall enter text file only
* User shall enter without file extension
* User shall know exact directory of the text file
* User shall enter alphabets to word inputs except dictionary file input (e.g. !@#%% not allowed on start, target, blacklisted words but dictionary file can have them)
* Users shall enter less number of blacklisted words than the number of filtered words that match the length of start word.
* User shall check a list of blacklisted words based on their inputs
* Users can skip entering blacklisted words by putting empty string (e.g. user can hit enter key to skip)
* User inputs must be in a dictionary file
* User shall put the number of blacklisted words they want to remove before processing path

## Software Requirements

* The program shall accept single file name
* The program shall display error message when word input is either numeric or empty or has punctuations
* The program shall display an error when value from user is invalid
* The program shall fill file extension to user input automatically
* The program shall only accept text file.
* The program should display blacklisted words
* The program shall enable users to skip to put blacklisted words if they desire
* The program shall ask users appropriate start, target and blacklisted word repeatedly until they input expected values (by while statement)

## Software Design

### High Level Design – Logical Block Diagram

neither

2

1

False

True

2

1

False

False

True

True

True

False

Output error message

Print shortest paths

Print possible unique paths

User input between 1 and 2

Output error message

User input for blacklisted words

Validate the input

Display a list of the words

User input for blacklisted words

1. The number of blacklisted words

2. Enter to skip

Output error message

Validate the input

User input for target word

Output error message

Validate the input

User input for start word

Output error message

Validate the input

User input for filename

### List of all functions in the software

Class buildsame:

1. def \_\_init\_\_(self, item, target, words)

* This function is to initialize variables given to this class
* Item: string and used as start word, target: string and used as target word, words: a list and used to find possible paths from dictionary file
* Calling with different number of arguments may raise an error
* No values returned

1. def get\_item(self)

* It returns the current iteration

1. def hit\_target(self, target)

* This function is to figure out whether current word matches the target word
* Target is string data type and is an element of lists
* This returns boolean type value. If target word is found it returns True. If not returns False

1. def get\_successors (self, current\_item)

* This function is to build a list of items that match a pattern defined by n-1 (n being the number of letters). This is done by taking each letter of a word and seeing if the remaining letters match any word in the dictionary. Lead = \_ead, l\_ad, le\_d, lea\_. (‘bead’, ‘load’, ‘lend’ & ‘leaf’ are all words that would match this criteria).
* It only has current\_item as input parameter which is string type. It is used when it comes to comparing if it matches the target worde
* It returns a list of any possible words from dictionary based on the pattern

Function bfs\_shortfind(buildsame)

* This function uses a bredth first search algorithm, which will return the shortest path upon the first successful hit of the target.
* This is to find the shortest path possible from start to target word
* This function receives an object created by buildsame class that will be used as a reference to start, target and words parameters created by user
* The iteration is first checked against the target – if a match, then the path is geneated using the current\_path\_rebuild function.
* If not, using the iteration, a series of successors are built using the buildsame function.
* If there are siblings for the current iteration, it will go through those before moving to the successors.
* It loops until target word matches to current\_item variable

Function bfs\_allfind(buildsame)

* This function uses the same structure as the bfs\_shortfind function.
* The only difference is that it uses a forward scanning method – searching the successors for a match. This is similar to the match == n-1 (n being the number of letters) in the supplied word\_ladder. If the current iteration is match == n-1, then it would be a viable path.
* The viable path is printed, without closing the loop.
* After each batch of succesors is processed, (and there are no further successors), the iteration is popped. Once that list is entirely empty (and can no longer be popped), the function ends.

Function construct\_path

* Once a target is hit, this function looks back through the actions taken to reach this path. This ignores all other siblings that weren’t part of the path. This continues until the starting word is reached.
* The rebuild is in reverse, so it is then reversed and returned.

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

Successors

* Set
* This was used in get\_successors method as a list of words that match the search curriculum
* It is a data member itself in this case
* Get\_successors is the only function using this data structure

Return\_list

* List
* This is just to return data set of successors as a list
* It is itself
* Get\_successors is where this data structure is used

Fifo\_q

* List
* Holds a list of all viable items based on the match n-1 function, except goes is used from the first item (i.e, a sibling), rather than the last (a child).

Visited\_path

* List
* This holds all visited items.
* It is itself
* Bfs\_shortfind and bfs\_allfind use this data structure

Current\_path

* Dictionary
* Holds a list of all visited items of the current iteration.
* It is itself
* Bfs\_shortfind and bfs\_allfind use this data structure

Return\_path

* List
* This is to contain each child of the current tree process. This is printed every time target word is matched
* It is itself
* bfs\_allfind applies this data structure

Words\_list

* List
* This contains a list of words that has the same length to the start word
* Used by buildsame to generate a list of successors to analyse

B\_list

* List
* This is only to create a list of blacklisted words based on user inputs
* Used globally to remove items from the global ‘Words’

### Detailed Design

### Configuration management and version control

Version control was maintained using the built in git commands in the Pycharm environment and BitBucket. User 1 (Kris Blanch), created a BitBucket repository, which User 2(Joonyoung Kim) was then given read/write access.

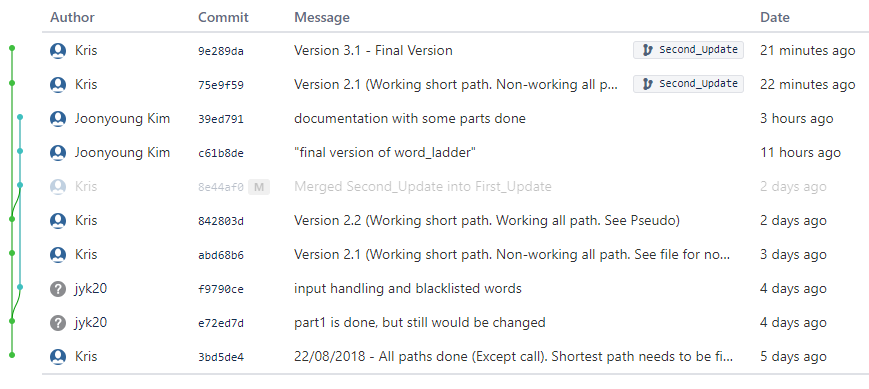
Versions were numbered with an x.y convention. The x numeral was changed when a task had been completed. The y numeral was changed when an alteration had been made to the file towards a task completion but had not yet been completed or tested.

The versions were then pushed from the Pycharm environment via Git to BitBucket. Until versions had been tested, they were pushed in their own development branch. After testing, branches were then merged with the main branch.

Both users were unfamiliar with version management, so some errors have occurred during this process, however the intent behind the version control has been fulfilled.

UnitTesting was performed on version 3.1 – this version does not correctly display the results of the allpath function. That version did not consider that the user was looking for paths of a specific length, instead processing all available paths. Subsequently, most of the documentation is built on version 3.1.

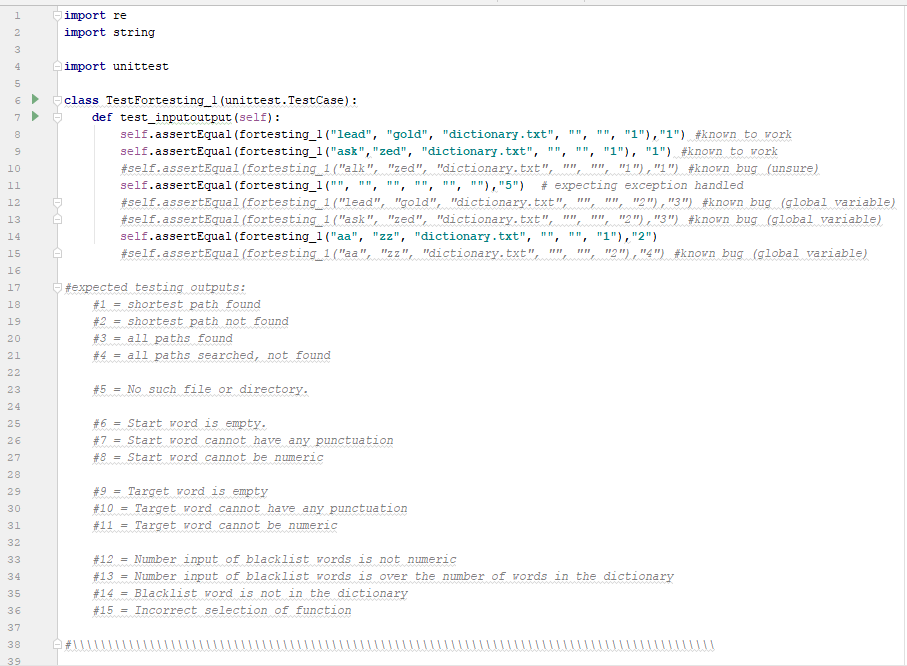
**Final version is version 3.2. This meets the requirements. Amendment 1 discloses the changes between 3.1 and 3.2.**



The above outlines the commits from Pycharm to BitBucket. This is a snapshot of the 3.1 version before testing. It was sitting in the second branch, where it was tested for validity before being migrated to the main branch.

## Unit Tests

The following is a snapshot of the testing conducted within the program file. Each exception handle is represented as a number (as defined in the testing outputs comments). The assertEqual function of the unittest module allows for accurate testing of expectation against actuality.



The following table outlines the tests conducted with the python unittest module, and the expected outcomes. In the case of the Allpath function, it relies on a global variable that was moved to a closed variable with the changing of the program for the testing. This resulted in a failure within the testing process.

The use of the word ‘alk’ causes a crash within the program if used as a starting word. This was discovered during random word testing, but does not impact the tasked start/target words (‘lead’ to ‘gold’ & ‘hide’ to ‘seek’)



|  |  |  |  |
| --- | --- | --- | --- |
| No | Test Case | Expected Results | Actual Results |
| 1.0 | Correct Inputs (lead, gold). No Blacklist. Shortpath | 1. | 1. |
| 1.1 | Different Inputs (ask, zed). No Blacklist. Shortpath | 1. | 1. |
| 1.2 | ‘alk’ as start word. No Blacklist. Shortpath | 1. | Display error message and exit |
| 2.0 | No Inputs | 5. | 5. |
| 3.0 | Correct Inputs (lead, gold). No Blacklist. Allpath | 3. | Display error message and exit |
| 3.1 | Different Inputs(ask, zed). No Blacklist. Allpath | 3. | Display error message and exit |
| 1.3 | Different Inputs (aa, zz). No Blacklist. Shortpath | 2. | 2. |
| 3.2 | Different Inputs (aa, zz), No Blacklist. Allpath | 4. | Display error message and exit |

The final unittesting file supplied with the program runs 4 of the 8 tests listed above, resulting in an okay. The remaining have been converted to comments.

## 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 single file | Full | Pass |  |
| 2 | Error message raises when file does not exist or there is no such directory | Full | Pass |  |
| 3 | File name without file extension works as long as they are text file. (automatically attached) | Full | Pass |  |
| 4 | If one of start, target and blacklisted words is not alphabets, it outputs error message | Full | Pass |  |
| 5 | If one of start, target and blacklisted words contains one of punctuation, numeric values and empty value it raises an error message according to the type of error. E.g. empty value raises an error message saying value cannot be empty. | Full | Pass |  |
| 6 | If the number of blacklisted words are greater than the number of a list of words it raises an error message. | Full | Pass |  |
| 7 | User can skip putting blacklisted values by hit “enter” key | Full | Pass |  |
| 8 | The number of blacklisted words are asked and user gets asked that number of times for those words | Full | Pass |  |
| 9 | When user inputs for start, target and blacklisted words are not expected values it keeps asking until receiving suitable values | Full | Pass |  |

## 

## User Instructions

* Files that is not under same directory or does not exist will raise an error message
* File extension is automatically filled (only text file allowed)
* Every user inputs only expect alphabets but input asking the number of blacklisted words only allows numeric values as valid ones.
* Run the program in python to minimize errors when running it
* Only single file is allowed as valid input
* Invalid values will raise an error message according to what they are. e.g. punctuation into user input raise the message that says “Punctuation is now allowed. Please try again”
* Start, target and blacklisted words would be asked until they get valid values.

## Amendment 1. Changelog for 3.1 to 3.2

* Added functionality for users to search for a desired length of the path in the allpath function (function 2).
* Input added for user to choose length of path.
* Input sanitation and validation added for desired pathlength.
* Amended comments to reflect changes.

### Pseudocode for this function

#(upon selection function 2)

pathlengthnum = input()

while pathlengthnum not number:

pathlengthnum = input()

#in bfs\_allpath

Count = int()

If next item == target:

print the rebuilt path.

Add to count

If no more successors (end iteration)

If count > 0:

Print(“All paths found”)

Else:

Print(“No Paths of this length found”)