Algorithmics I (H) : COMPSCI 4009

Assessed Exercise - Word Ladder

15th November 2022

GRANIÉ Sébastien

This document is a report about the Word Ladder project part of the Algorithmics I course. In this document, you will find a status report of the programs written, a discussion justifying my implementations decisions and the output produced by your programs for the testdata file provided.

University of Glasgow - School year 2022/2023

Table of contents

1 Introduction

First of all, this report states as written explanation about the programs that have been written to make the project successful. This project was divided in two parts.

The first program takes as **inputs** :

— a dictionnary file ;
— a beginWord ;
— an endWord.

and should **output** the length of the shortest path and its the ladder that transforms the beginWord into the endWord, or should report that no ladder is possible if any path is possible between both words using the mentioned dictionary of words. The dictionnary file is composed exclusively of five-letter words. We need to keep in mind that a word ladder is a sequence of words, where each member of the sequence differs from its predecessor in exactly one position. To move from one word to another, we can only perform a substitution of letter at the same position in both words. Here is an example : ***flour*** $\longrightarrow$ ***floor***

The second program takes the same inputs as the first program. We keep in mind the same elementary transformation between two words (one letter). Within these transformations, we now add a weight on the edge between two words, which equals the absolute distance within the alphabet of the letter change. The following transformation : ***flour*** $\longrightarrow$ ***floor*** is equal to the distance in the alphabet between letters **o** and **r**, which equals **3**. Alike the program 1, the aim is to find the shortest path in terms of weight this time and output the ladder used to perform the transformation from the beginWord to the endWord.

To compile and execute the code, a **ReadMe.md** file is available in the given archive and explains how to process.

2 First Program :

2.1 Goal - Main ideas

As shortly explained before, we take as inputs a list of five-letter words, a beginWord and an endWord, that are both assumed to be in the mentioned list of words. To achieve the goal mentioned in the introduction, it is particularly important to get a clear view of the graph built with all the words contained in the list of words. This graph will give us all the connections between the words. We study here a non-weighted and non-directed graph.

In the lectures, we learned different ways and algorithms to browse through a graph. Since this word ladder project is an algorithmic project, we will take care even more than usual to select the most appropriate data structures to all situations in order to optimize the execution time and find the best solution in less possible time.

2.2 Code review - Choices explanation

The structure is quite simple, we only have two classes : **Graph.java** and **Vertex.java**.

The **Graph** class is mainly used to build the whole graph using the list of words as parameter. A graph is build like a 2D table. The attributes **vertices** is composed of all the words of the list, known as **Vertex** by now in the code. There are a few more attributes and methods available in the code to get the size of the graph and display all its vertex amongst others.

The **Vertex** class is extremely central and important in this project as it has lots of attributes. First of all, each Vertex is mainly defined by a string which is its name, the attributes **visited** and **predecessor** are quite important as well because the attribute visited helps to avoid useless cyclic loops in the algorithm and the attribute predecessor allows each Vertex to keep in memory its best previous neighbour to build and retrieve the right ladder at the end of the algorithm.

The more interesting part is now to explain the **Main function**.

The Graph is built with a list of words in parameter. This list of words comes from the dictionary file that has been read and all the words have been added in an ArrayList. Iteratively, we add a new Vertex, named with the word as string into the first table **vertices**. After adding a new word, we check if the new word added is a neighbour of any words already added (substitution of same index letter) and if it is, we add in both of their adjacency list the respectively neighbour.

For example, if **word2** has been added into vertices, we compare **word2** with **word1** which was already in the graph. Since we have found out that **word1** and **word2** are neighbours, we add **word1** in **word2**'s adjacency list and we add **word2** in **word1**'s adjacency list.

Once our graph is well prepared, we can browse through it. Since we consider here a non-weighted and non-directed graph and we would like to find the depth of the endWord starting the research from the beginWord, we will perform a Breadth-First-Search (BFS) algorithm because it is the most appropriate algorithm to do this task. It consists in browsing every level of the graph from a starting node by discovering its neighbours and so on. We do not jump to a deeper level of neighbours without having browsing through all the neighbours of the current level of research. The most appropriate data structure to this situation is surely a **queue**. We initialise our queue to which we add the **beginWord** because it is our first level of research.

While the queue is not empty, we visit all the neighbours of all the vertices present in the queue. After looking at all the neighbours of a vertex, we remove it from the queue to avoid using it again and we set its visibility to **true** so that it will not be evaluated again. Once the endWord appears to be a neighbour of one of the graph's node, we stop because the BFS algorithm assures us by its construction that it is the optimal solution. With the help of the predecessor attribute, we are able to retrieve the corresponding optimal ladder from the beginWord to the endWord.

2.3 Status report

The program is functionnal and outputs the exact same expected output which contains the followed information :

— The size of the dictionnary used ;
— The beginning Word ;
— The ending Word ;
— The minimum path distance ;
— The corresponding optimal ladder found from beginWord to endWord ;
— The elapsed time to compute the program.

3 Second program :

3.1 Goal - Main ideas

As far as the second program is concerned, it takes the same number and type of inputs, which are a list of five-letter words, a beginWord and an endWord, that are both assumed to be in the mentioned list of words. We keep for granted that two words are connected in a graph if they are only from one letter substitution away from one another at the same index in both words. However, by now, the distance between two connected words will no longer be equal to one, but rather to the distance in the alphabet between the two letters substituted. As a consequence, we now study a weighted (non-directed) graph. It is non-directed because for all the nodes in the graph, if you take one of its edges to move to another node, you can come back using the same edge. In both directions, the cost will be equal to the weight of the edge.

Since we are studying a positive weighted graph, the most optimal known algorithm is called the Dijkstra's algorithm.

3.2 Code review - Choices explanation

The classes structure is pretty identical to the one in the first program and we are going to go through it in details.

The **DijkstraGraph** class is the same as in the first problem. All the changes appear in the **DijkstraVertex.java** file. We use a new attribute **distance**, which corresponds to the distance from the vertex to the beginWord. I added the **distanceLetter** method that enables to compute the distance between two vertices (i.e. two words), which are neighbours.

As far as the **Main function** is concerned, we pre-process the dictionnary file in the same way that we did it in the first part of the exercise. It is important to recall that every vertex is initialized with a distance equals to **Integer.MAXVALUE** to represent infinity. Before browsing through the graph, we need to setup the weights and distances of the beginWord and its neighbours. The distance of beginWord is surely zero and the distances of its neighbours are known using the **distanceLetter()** method.

We can now start the core of the program. We will be using a queue data structure like in the first question. This choice comes from the ease of adding and retrieving items from this list in a desired order. The main loops are the same but the algorithm is totally different compared to question one. In that case, all the nodes (vertices) of the graph will have two main attributes : **distance** and **visited**. The attribute distance of a vertex keeps in memory, the minimum distance between his position and the beginWord. The attribute visited helps the algorithm to move forward to visit new neighbours and avoid any infinity loops within the algorithm. Each time we visit a vertex, we check if the current path used might be a better one than the previously kept in memory. If it is a better path, it means a lower distance, we update both attributes of the current current vertex found.

We keep this method until we find the endWord. It is true that we do not need to visit all the nodes of the graph because of the context of word ladder we are considering in this project. The distance between two connected words is the distance in the alphabet of the unique different letter, located at the same index in both words. **This distance is positive**. Any negative weight would make the dijkstra algorithm fail. This context mathematically alows us to stop the algorithm once we have found the endWord.

3.3 Status report

The program is functionnal and outputs the exact same expected output which contains the followed information :

— The size of the dictionnary used ;
— The beginning Word ;
— The ending Word ;
— The minimum path distance ;
— The corresponding optimal ladder found from beginWord to endWord ;
— The elapsed time to compute the program.

4 Complexity - Data Structures choices

In this section, I will explain my major choices of data structures that helped me to optimize the efficiency of my programs.

— One choice is the use of an **ArrayList** called **myWords**. It contains all the words of the dictionary. It is an efficient solution to use an ArrayList because we can access any element with **constant time O(1)** with the method **.get(index)**. We mainly use it in the construction of the graph and as well in the main function to iterate over the neighbours of a vertex. To improve the robustness of the program, this data structure is efficient to check whether both the beginWord and the endWord are present in the list of words in the dictionary.
— A second choice is the use of a **LinkedList** called **adjList** for the adjacency list of all vertices of the graph. We use it to add vertices and it is done with **constant time O(1)**.
— A last interesting data structure used in the project is the **queue** in both main function. It helps us to keep a good structure in the code and it assures us that all vertices will be considered in a certain order that we decided in the algorithm. We use mainly two methods, which are **poll()** and **offer()** whose complexity is **O(log(n))**.
— More generally in the code, I paid attention to optimize the actions within each loops to avoid running them twice unnecessarily.

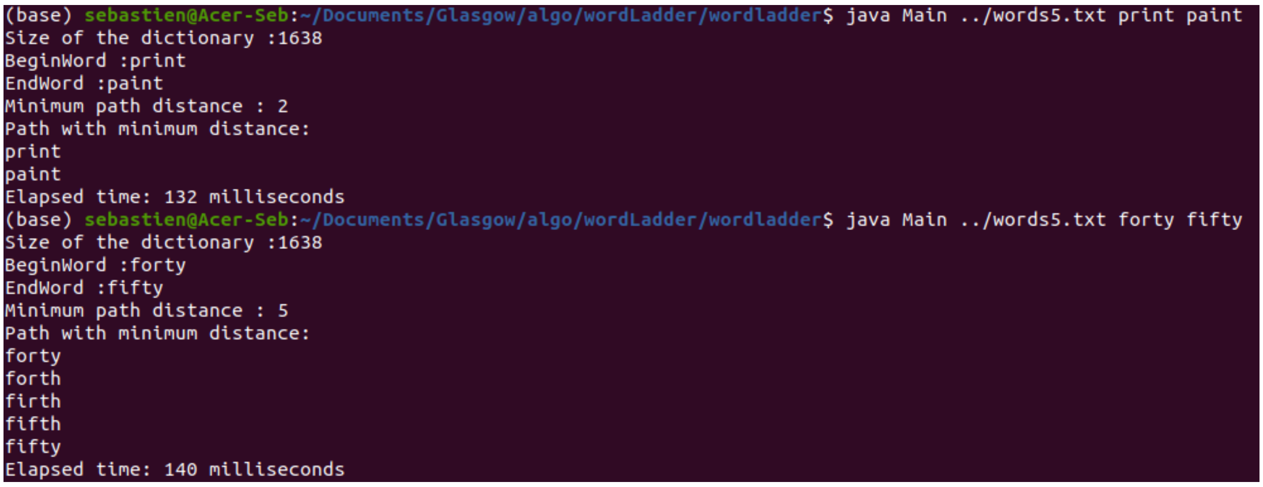
5 Conclusion - Personal retrospective

As far as the topic of the project is concerned, I really enjoyed solving both problems. It helped me better understand the concepts learned in the lectures and also to solve a practical exercise related to algorithmics. I appreciated trying to optimize the code as much as possible and questioning myself each time I used a loop to take advantage of it. Although this project was primarily an Algorithmics exercise, rather than a Software Engineering exercise, I took care about the Java best practice to use, such as adding getters and setters to classes to access private attributes, using enhanced for loops, proper commenting. I chose to write my own report to make the layout more convenient to read.

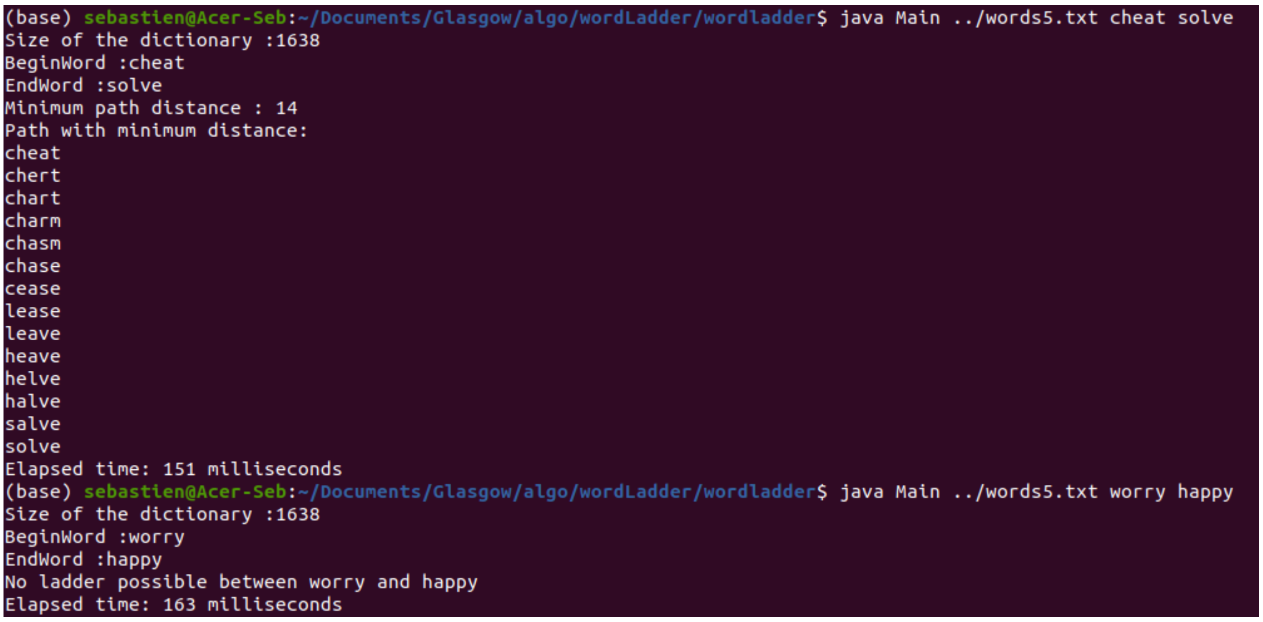
6 Programs output

In this section are listed the programs output proposed by the testdata.txt file.

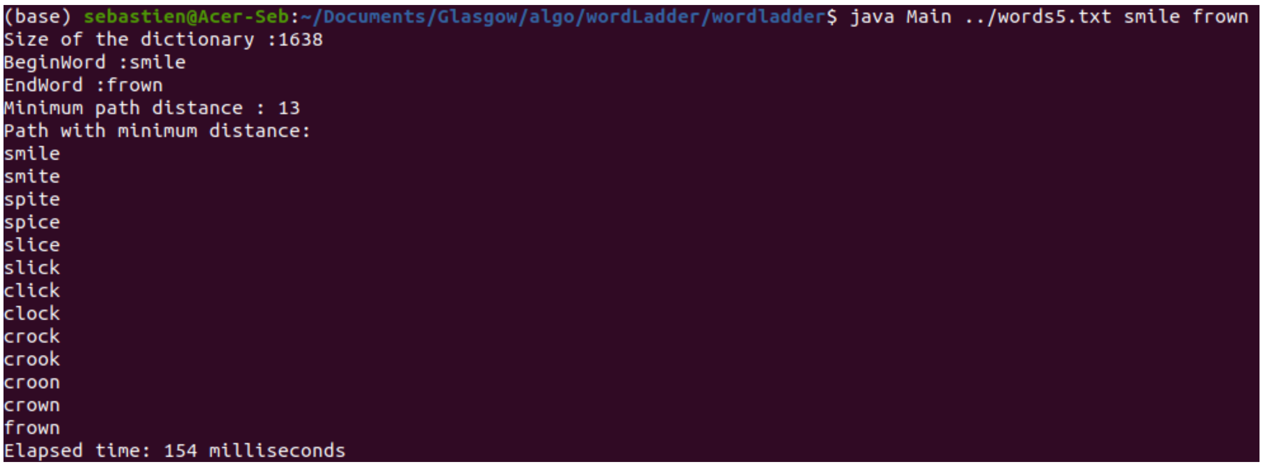
6.1 First program :

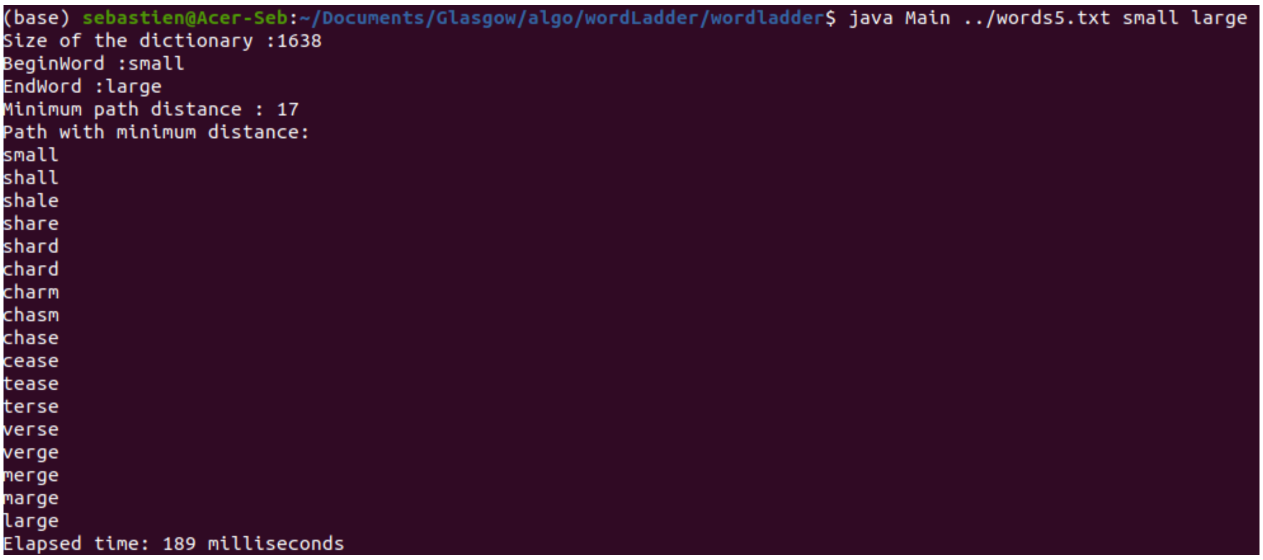


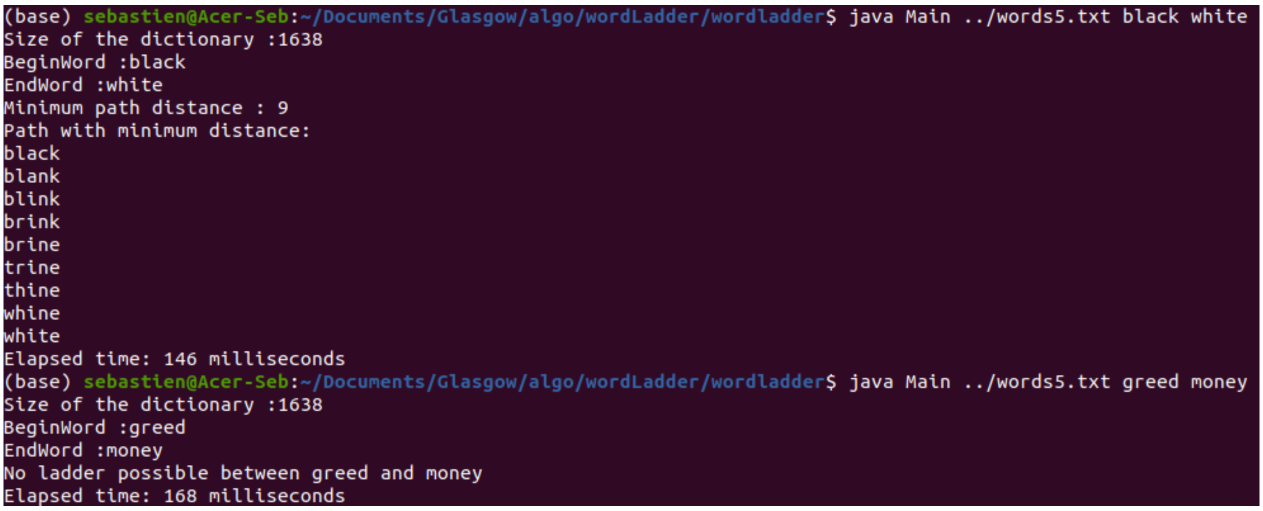
```
(base) sebastien@Acer-Seb:~/Documents/Glasgow/algo/wordLadder/wordladder$ java Main ../words5.txt print paint
Size of the dictionary :1638
BeginWord :print
EndWord :paint
Minimum path distance : 2
Path with minimum distance:
print
paint
Elapsed time: 132 milliseconds
(base) sebastien@Acer-Seb:~/Documents/Glasgow/algo/wordLadder/wordladder$ java Main ../words5.txt forty fifty
Size of the dictionary :1638
BeginWord :forty
EndWord :fifty
Minimum path distance : 5
Path with minimum distance:
forty
forth
firth
fifth
fifty
Elapsed time: 140 milliseconds
```



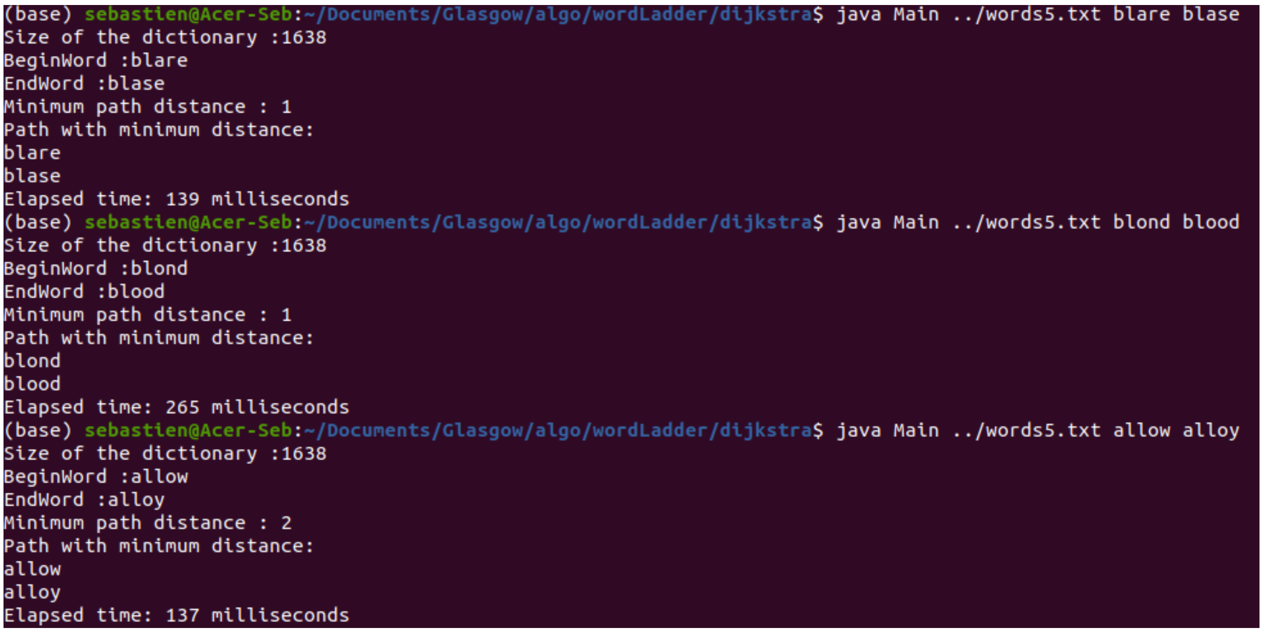
```
(base) sebastien@Acer-Seb:~/Documents/Glasgow/algo/wordLadder/wordladder$ java Main ../words5.txt cheat solve
Size of the dictionary :1638
BeginWord :cheat
EndWord :solve
Minimum path distance : 14
Path with minimum distance:
cheat
chert
chart
charm
chasm
chase
cease
lease
leave
heave
helve
halve
salve
solve
Elapsed time: 151 milliseconds
(base) sebastien@Acer-Seb:~/Documents/Glasgow/algo/wordLadder/wordladder$ java Main ../words5.txt worry happy
Size of the dictionary :1638
BeginWord :worry
EndWord :happy
No ladder possible between worry and happy
Elapsed time: 163 milliseconds
```


```
(base) sebastien@Acer-Seb:~/Documents/Glasgow/algo/wordLadder/wordladder$ java Main ../words5.txt smile frown
Size of the dictionary :1638
BeginWord :smile
EndWord :frown
Minimum path distance : 13
Path with minimum distance:
smile
smite
spite
spice
slice
slick
click
clock
crock
crook
croon
crown
frown
Elapsed time: 154 milliseconds
```

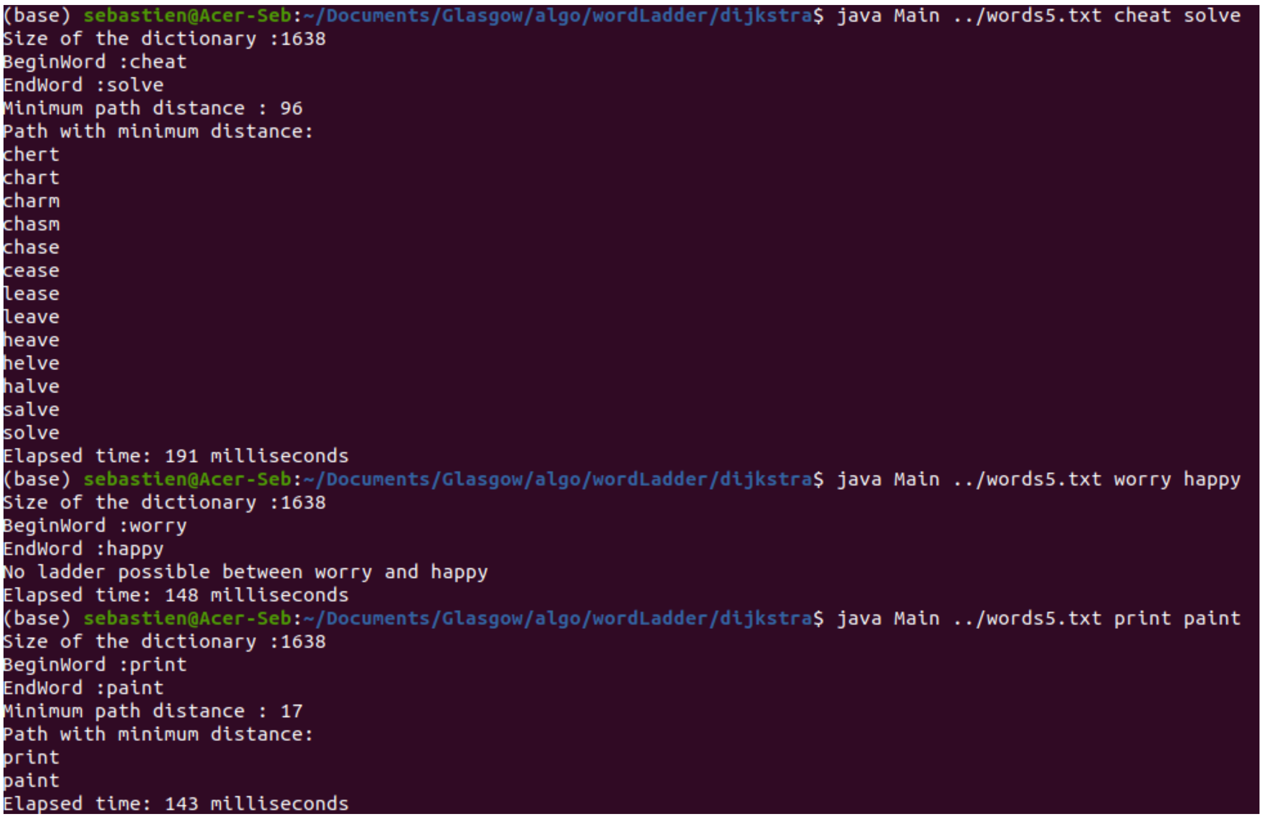

```
(base) sebastien@Acer-Seb:~/Documents/Glasgow/algo/wordLadder/wordladder$ java Main ../words5.txt small large
Size of the dictionary :1638
BeginWord :small
EndWord :large
Minimum path distance : 17
Path with minimum distance:
small
shall
shale
share
shard
chard
charm
chasm
chase
cease
tease
terse
verse
verge
merge
marge
large
Elapsed time: 189 milliseconds
```


```
(base) sebastien@Acer-Seb:~/Documents/Glasgow/algo/wordLadder/wordladder$ java Main ../words5.txt black white
Size of the dictionary :1638
BeginWord :black
EndWord :white
Minimum path distance : 9
Path with minimum distance:
black
blank
blink
brink
brine
trine
thine
whine
white
Elapsed time: 146 milliseconds
(base) sebastien@Acer-Seb:~/Documents/Glasgow/algo/wordLadder/wordladder$ java Main ../words5.txt greed money
Size of the dictionary :1638
BeginWord :greed
EndWord :money
No ladder possible between greed and money
Elapsed time: 168 milliseconds
```

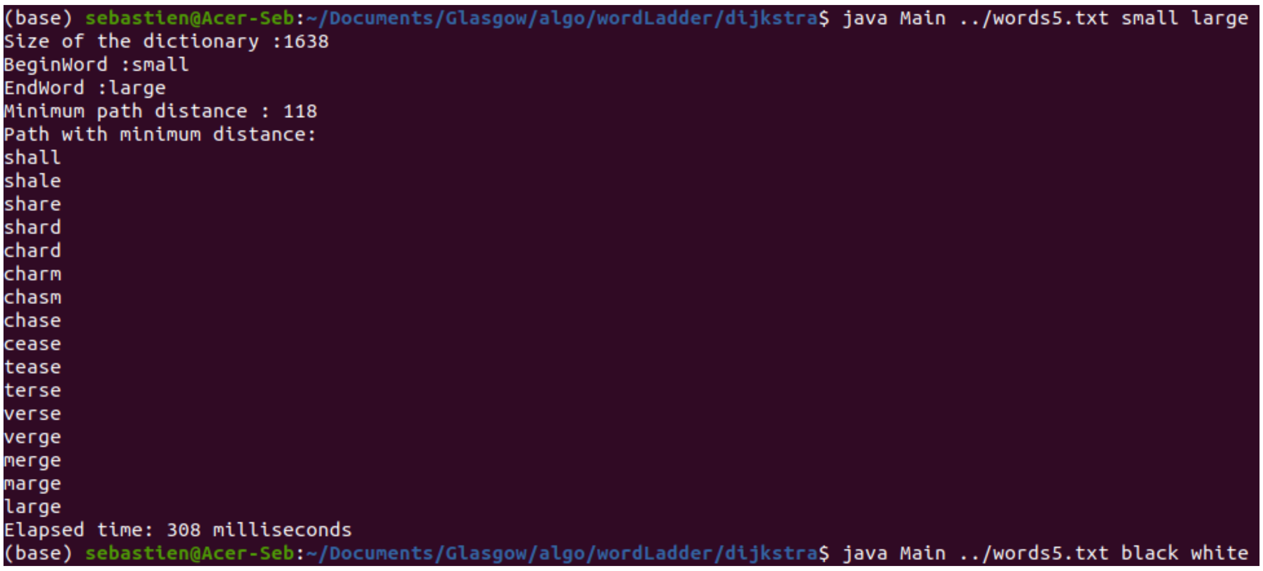
6.2 Second program :



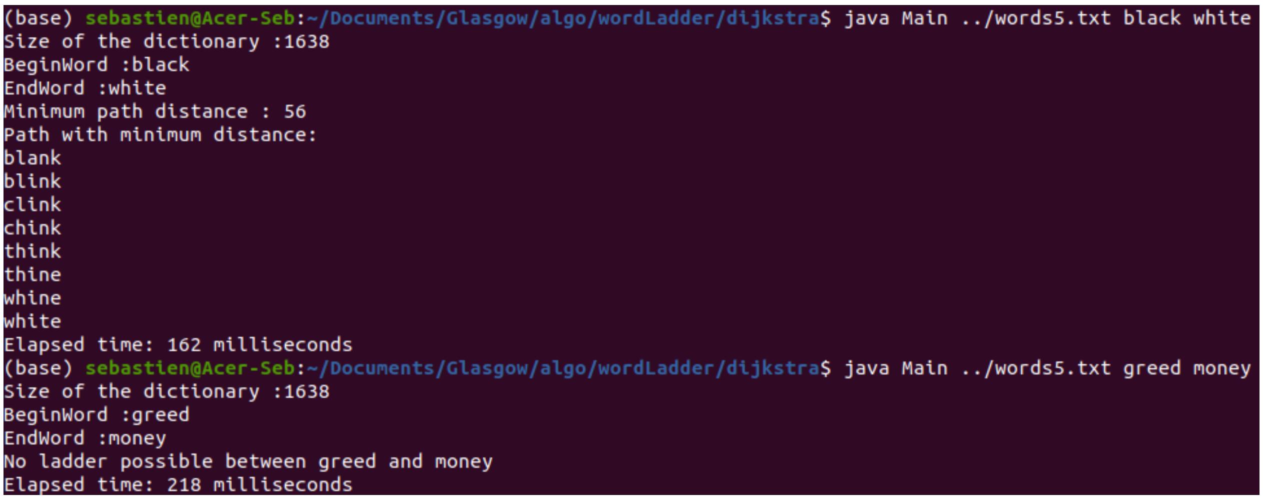
```
(base) sebastien@Acer-Seb:~/Documents/Glasgow/algo/wordLadder/dijkstra$ java Main ../words5.txt blare blase
Size of the dictionary :1638
BeginWord :blare
EndWord :blase
Minimum path distance : 1
Path with minimum distance:
blare
blase
Elapsed time: 139 milliseconds
(base) sebastien@Acer-Seb:~/Documents/Glasgow/algo/wordLadder/dijkstra$ java Main ../words5.txt blond blood
Size of the dictionary :1638
BeginWord :blond
EndWord :blood
Minimum path distance : 1
Path with minimum distance:
blond
blood
Elapsed time: 265 milliseconds
(base) sebastien@Acer-Seb:~/Documents/Glasgow/algo/wordLadder/dijkstra$ java Main ../words5.txt allow alloy
Size of the dictionary :1638
BeginWord :allow
EndWord :alloy
Minimum path distance : 2
Path with minimum distance:
allow
alloy
Elapsed time: 137 milliseconds
```



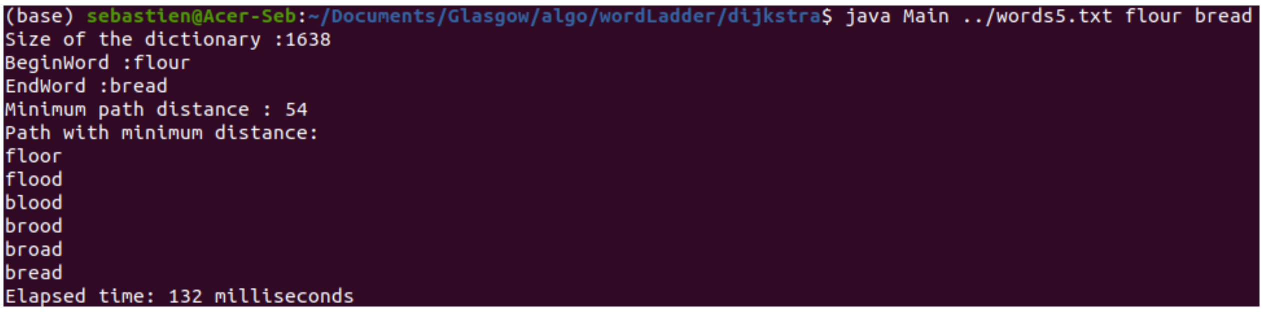
```
(base) sebastien@Acer-Seb:~/Documents/Glasgow/algo/wordLadder/dijkstra$ java Main ../words5.txt cheat solve
Size of the dictionary :1638
BeginWord :cheat
EndWord :solve
Minimum path distance : 96
Path with minimum distance:
chert
chart
charm
chasm
chase
cease
lease
leave
heave
helve
halve
salve
solve
Elapsed time: 191 milliseconds
(base) sebastien@Acer-Seb:~/Documents/Glasgow/algo/wordLadder/dijkstra$ java Main ../words5.txt worry happy
Size of the dictionary :1638
BeginWord :worry
EndWord :happy
No ladder possible between worry and happy
Elapsed time: 148 milliseconds
(base) sebastien@Acer-Seb:~/Documents/Glasgow/algo/wordLadder/dijkstra$ java Main ../words5.txt print paint
Size of the dictionary :1638
BeginWord :print
EndWord :paint
Minimum path distance : 17
Path with minimum distance:
print
paint
Elapsed time: 143 milliseconds
```



```
(base) sebastien@Acer-Seb:~/Documents/Glasgow/algo/wordLadder/dijkstra$ java Main ../words5.txt small large
Size of the dictionary :1638
BeginWord :small
EndWord :large
Minimum path distance : 118
Path with minimum distance:
shall
shale
share
shard
chard
charm
chasm
chase
cease
tease
terse
verse
verge
merge
marge
large
Elapsed time: 308 milliseconds
(base) sebastien@Acer-Seb:~/Documents/Glasgow/algo/wordLadder/dijkstra$ java Main ../words5.txt black white
```



```
(base) sebastien@Acer-Seb:~/Documents/Glasgow/algo/wordLadder/dijkstra$ java Main ../words5.txt black white
Size of the dictionary :1638
BeginWord :black
EndWord :white
Minimum path distance : 56
Path with minimum distance:
blank
blink
clink
chink
think
thine
whine
white
Elapsed time: 162 milliseconds
(base) sebastien@Acer-Seb:~/Documents/Glasgow/algo/wordLadder/dijkstra$ java Main ../words5.txt greed money
Size of the dictionary :1638
BeginWord :greed
EndWord :money
No ladder possible between greed and money
Elapsed time: 218 milliseconds
```



```
(base) sebastien@Acer-Seb:~/Documents/Glasgow/algo/wordLadder/dijkstra$ java Main ../words5.txt flour bread
Size of the dictionary :1638
BeginWord :flour
EndWord :bread
Minimum path distance : 54
Path with minimum distance:
floor
flood
blood
brood
broad
bread
Elapsed time: 132 milliseconds
```