

## Project Summary

Modern compilers can be useful, not only in detecting programming errors but by suggesting repairs for those errors using ‘error repair’. Error repair is the process of finding a repair for an error that happened as a result of compiling a piece of code written by a programmer. However, it is not always consistent with the intention of programmers. We aim in this project to enhance the error repair process in compilers. Our research finds that a typical Java compiler does not provide solutions for the most common syntax programming errors which occur as a result of spelling correction errors. Therefore, we have employed two spelling algorithms in order to find repairs to those errors. The algorithms are the Four-Way and Editex algorithms. In our work we extended the algorithms to be compatible to work with programming languages. In addition, we implemented and tested both of them to find out which is the most suitable to be used. Furthermore, we have designed a tool called ‘edit & compile’ which helps in increasing the efficiency of the algorithms. Although we aim to enhance error repair in Java compilers in particular, our approach has proven to be effective for other programming languages such as C. The study concludes that using the Editex algorithm with the edit & compile extension is the best choice in the case of finding repairs to programming errors that occur as a result of spelling errors. Our test is based on the accuracy of finding repairs for errors that are consistent with programmers’ intentions and the execution time of each algorithm.

The following points summarize our achievements and contributions in the project:

- Implement and adapt the Four-Way algorithm proposed by [18] in compiling (see pages23 - 26 and 36 - 36).
- Extend, implement and adapt the Editex algorithm proposed by [26] in compiling (see pages26 - 31 and 36 - 38).
- Design and implement a novel approach, ‘edit & compile’, which is combined with the Four-Way and Editex spelling correction algorithms in order to enhance the efficiency of the algorithms as well as other features, which are context awareness, synonyms list, learn by experience and further repairing, (see pages38 - 42).
- Test and study the results of our implementation of the algorithms. We conclude that the Editex algorithm combined with ‘edit & compile’ is the best algorithm for repairing spelling programming errors in compiling (see pages43 - 51).

# Table of Contents

Project Summary.....	1
Table of Contents .....	2
List of Tables.....	4
List of Figures .....	5
1. Introduction.....	6
1.1 Thesis Outline.....	6
2. Scope and Motivation .....	8
3. Background.....	9
3.1 Error Repair.....	9
3.2 Compilers .....	10
3.3 Error Repair Algorithms .....	12
3.4 Related work.....	14
4 Programming errors .....	17
4.1 Cannot Find Symbol Error .....	18
5 Spelling Correction Algorithms .....	21
5.1 Spelling Correction Process .....	21
5.2 Disagreement Technique .....	23
5.3 Four-Way Algorithm .....	23
5.4 Editex Algorithm .....	26
5.5 Summary .....	31
6 Methodology.....	32
6.1 Data Collection.....	32
6.2 Test Measure.....	33
6.3 Used Applications.....	34
6.4 Summary .....	34
7 Project Design.....	35
7.1 Building the Dictionary .....	35
7.2 Agreement Threshold .....	35
7.3 Four-Way Algorithm .....	36
7.4 Editex Algorithm .....	36
7.5 Edit & Compile .....	38
7.6 Context Awareness .....	39
7.7 Synonyms List.....	39
7.8 Learn by Experience.....	40
7.9 Further Repairing .....	40

7.10	Summary.....	42
8	Results and Discussion.....	43
8.1	Four-Way Algorithm .....	43
8.2	Editex Algorithm .....	44
8.3	Four-Way Vs Editex .....	45
8.4	Edit & Compile .....	46
8.5	The Approach Dependency .....	50
8.6	Summary .....	51
9	Future Work .....	52
9.1	Test Tools.....	52
9.2	BlueJ.....	52
9.3	icompile.....	53
9.4	Recommendation for further studies .....	53
9.5	Summary .....	53
10	Conclusion .....	54
11	Bibliography.....	56

## List of Tables

Table 1: Top Error Messages.....	18
Table 2: Error weight .....	24
Table 3: Error type examples .....	25
Table 4: Edit distance, first iteration .....	27
Table 5: Edit distance, finished .....	28
Table 6: Soundex Code Group .....	28
Table 7: Editex code group .....	29
Table 8: Editex example.....	31
Table 9: Test Types with Examples .....	33
Table 10: Candidate List of Four-Way Example.....	36
Table 11: Extend Editex Code Group .....	37
Table 12: Candidate List of Editex Example .....	37
Table 13: Four-Way Algorithm Results .....	43
Table 14: Editex Algorithm Results .....	44
Table 15: Four-Way with Edit & Compile Results .....	46
Table 16: Editex with Edit & Compile Results .....	47
Table 17: Edit & Compile, candidate words number =10.....	48
Table 18: Different Candidate List Size Results .....	49
Table 19: Algorithms Results .....	55

## List of Figures

Figure 1: Graphical class interaction. ....	11
Figure 2: BlueJ offers help to solve errors. ....	11
Figure 3: Burke - Fisher works with queue size 3 .....	13
Figure 4: Error message by BlueJ with extra help .....	19
Figure 5: Declaring own words.....	22
Figure 6: Spelling Correction Process .....	22
Figure 7: Four-Way Algorithm.....	25
Figure 8: Edit Distance Algorithm .....	27
Figure 9: Editex algorithm .....	30
Figure 10: Basic Algorithm.....	38
Figure 11: Edit & Compile Approach .....	38
Figure 12: The Complete Algorithm.....	41
Figure 13: The Tool Suggesting a Repair.....	41
Figure 14: Typical Compiler Error Message.....	41
Figure 15: Compile & Edit accuracy levels .....	49

## 1. Introduction

Compilers are designed to translate a piece of code written in a programming language into a machine language that a computer can understand. A compiler analyses a source of code in order to find a programming error before the translation. If a programming error is detected, then the compiler reports it to the writer of the code to let him correct it. Compilers are desired to help programmers find out about the error. Information that can be helpful is the line number in which the error is expected to be. Moreover, modern compilers are capable of suggesting repairs for these errors using the error repair method which is based on error recovery. However, novice programmers can face a challenge in understanding the error messages of compilers, according to [25], due a lack of error solutions being provided. Good compiler error messages should provide error repair [46].

Error recovery, the process of finding repair for an error that happened as a result of compiling a piece of code written by a programmer, is a well known part of compiler design in computer science. This process, in general, aims to locate the accurate position of the error [42]. Moreover, it tries to find a solution to the error to allow continued parsing of the remaining code. After finishing compiling, error messages are then produced to help programmers locate and fix listed errors. While Jacob Nielsen has recommended that error messages should suggest a solution [43], some compilers may just indicate that there is an error in a line of code without suggesting repairs. Our work is to study error recovery methods and algorithms in order to find a more workable solution for programmers.

After investigating common programming errors, we found that misspelling problems cause programming errors that a typical Java compiler cannot suggest a repair for. This type of error is solved by spelling correction algorithms. A spelling correction algorithm computes which word is the correct form of a misspelled word amongst a list of words kept in a dictionary. The aim of this work is to enhance the error repair in compilers by suggesting repairs to programming errors that are as a result of misspelling problems. We have employed two spelling correction algorithms to decide which one is the best to achieve our goal.

### 1.1 Thesis Outline

The thesis starts by expressing our motivation for doing this project. Our motivation is that enhancing the error repair in compilers can help programmers to fix their programming errors faster. By this we mean enhancing the error repair to let a compiler suggest a repair for a programming error. This repair should be consistent with programmer's intention. Our project is limited to errors that are as a result of spelling errors, because this type of error results in producing error messages that do not suggest a repair in a typical Java compiler. Although the project is limited to Java compilers, we show in section 8.5 that our approach can apply to other programming languages such as C.

Secondly, a background of error repair in compilers is discussed in the third chapter. Error repair means repairing a programming error in order to let a compiler resume searching to find other errors. There are different algorithms that try to find a solution to a programming error. In the chapter we have discussed the Burke-Fisher and Follow set algorithms. However, these repairs are not always done as a programmer wants. In fact, our work means that fixing an error repair committed by a programmer to be working as the programmer wanted. In addition to that, the chapter studies some compilers that are our targets in the project.

Then, we study programming errors in order to find what errors are not solved and how they can be solved. We have found that errors due to spelling problems are ranked as the top common errors. Spelling problems means any mistyping in writing a piece of code. This involves variables, methods, keywords, etc. However, this remains unsolved in many Java compilers. In contrast, other programming errors have repairs suggested. Therefore, we have employed a spelling correction algorithm to solve this problem.

We have discussed two spelling algorithms which are the Four-Way and Editex algorithms. The Four-Way algorithm is an algorithm used to correct spell errors in a compiler designed to compile natural languages. The algorithm compares a misspelled word with a list of words which each can be a valid repair. The comparison compares words on a letter level then assigns an error weight depending on the error type. The word with the least error weight is considered the match. The Editex algorithm is an algorithm which adapts and improves the Soundex algorithm to be in the Editex code group. This combines with edit distance method, which measures the similarity between two words. The algorithm assigns a score or error weight to each word that is compared with an erroneous word. The word with the lowest score is deemed to be a repair. Both algorithms have not been applied to this field before.

In chapter 7, shows how we have extended and implemented the spelling correction algorithms. In addition, new features have been designed to improve the application of the algorithms to the compilers. Edit & compile is a new approach which is used along with the Four-Way and Editex algorithms in order to increase their efficiency. Other features are ‘learn by experience’, ‘context awareness’, ‘synonyms list’ and ‘further repairs’. Then, we tested our approach in order to compare the algorithms and new features. Chapter 8 evaluates our test of the algorithms. We compared the Four-Way and Editex algorithms in terms of accuracy of finding repairs and the time required for execution. Chapter 9 presents and discusses the future work of more testing and enhances the method we have used, and makes some recommendations for more research.

## 2. Scope and Motivation

Computer programmers rarely write a piece of code free of error when first compiled. This assertion includes even expert programmers who might spend a long time on finding repairs for naïve errors. More obviously, novice programmers can struggle to understand the causes of programming errors and to figure out ways to solve them; especially if the error that happens is of a type the novice programmer has not been taught about. Therefore, fixing programming errors can become time-consuming because programmers might not understand why they have happened. Moreover, compilers may not be so useful in terms of clearing due to their generating ambiguous error messages. The latter refers to a message that is produced by a compiler which does not provide a fix to a programming error. We are thus motivated by the aim of suggesting a fix to an error which can help a wide range of programmers to carry on programming faster. It seems that novice programmers face a challenge in understanding the error messages of compilers [25] because some of them do not provide solutions or more [37][46]. Moreover, such help would enable novice programmers to make better progress when trying to understand programming languages. Kölling, a developer of BlueJ, states that error messages can help novice programmers significantly with regard to understanding and resolving programming errors [43].

In terms of a broader context, our aim is to develop the compilers' work, while on a more specific level, we would like to enhance error repair in compilers by making them able to suggest solutions to errors. We do not aim to design a new parser as so many are already available. However, our scope is to investigate and design a method that applies to an available Java compiler which helps programmers to find repairs to programming errors. There are different algorithms that help compilers to find repairs, for example, the Burke-Fisher algorithm. However, these algorithms repair errors enable compilers to continue parsing and do not consider the point of view of the code writer, whereas, we aim to find repairs that are consistent with the programmer's intention.

Our work is limited to programming errors that occur as a result of spelling problems. In order to solve these errors, we combined the spelling correction algorithms with the compiler's work. The target language is Java, though it is shown later that it can be applied to other programming languages. Java is commonly used in programming introduction courses; therefore, it has been chosen as a target language in order to be used in those courses to help programmers to increase their programming ability. The spelling correction algorithms are the Four-Way and Editex algorithms. Both algorithms have not been implemented in this field before. It can be argued that our contribution is very limited because it tries to enhance error repair for a specific type of programming error. Our counter argument is that a typical compiler such as JavaC can find repairs successfully for other programming errors, such as missing a semicolon. In addition, this type of programming error requires implementing spelling correction algorithms. It can be stated which algorithm is the best; therefore, we have studied, extended, implemented and tested two algorithms which are the Four-Way and Editex algorithms.



### 3. Background

#### 3.1 Error Repair

Error repair is a process of parsing which deletes or inserts symbols in order to resolve an error that has been encountered while compiling a piece of code[10]. Recently, shifting symbols have been added as way of solving the error. Error recovery aims to achieve three goals[15]: firstly, reporting the detection of an error in an accurate and clear way. Secondly, the quick repair of each error. The reason behind this is to be able to detect and report subsequent errors. Finally, it should be possible to correct a piece of code quickly. Error repair aims to let a compiler who has found a syntax error during parsing code to continue parsing the code in order to find subsequent errors. Error repair can be classified into two categories [30]: the first is local error repair which is a method of attempting to resolve an error by applying the error repair process on input symbols after detecting the position of the error. The second category is global error repair which is a method similar to local error repair but it includes input symbols before the position of the detected error as well.

It is vital to alleviate spurious error messages in compiling programs. Spurious error messages are messages of false errors. In other words, they are the result of things that are not real errors. Error correction is a step of modifying the input stream which contains an error in a way to make it correct. If the goal is to repair errors only for the purpose of continuing parsing (i.e. to detect subsequent errors) regardless of whether or not the change is what the programmer intended to do, it is called *error repair*. Error correction tries to fix errors as intended by the programmer. Therefore, any error correction is error repair but not vice versa [23]. Common error recovery techniques may include:

*Correcting technique*: this is a technique which aims to transform an incorrect input string into a syntactically correct input string by inserting, deleting or modifying the position of symbols. This technique may involve different algorithms that seek to correct an erroneous input stream, for instance the Burke – Fisher algorithm.

*Error production*: a parser generates a special token, which is called an error token, each time it encounters a syntax error. This technique requires a grammar rule in order to recognise it in the context and then take appropriate action. However, the technique may produce a group of unwanted errors that are a result of previous unsolved errors.

*Panic mode*: this is a simple technique in terms of implementation. When a parser encounters an error, it simply looks for a special symbol which is usually a delimiter (such as a semicolon in Java). This mode is applied in cases where a parser fails to find a repair for an error in an input string. Then, it skips over some parts of the input until it finds a special symbol. This method, however, is very inefficient although this is simply because it may generate spurious errors.

### 3.2 Compilers

Generally, compilers can be classified into two phases. The first phase is called the front end which is divided, as well, into three phases. First phase called lexical analysis, which means that a source code is read character by character in order to produce tokens and terminals [11]. Terminals and tokens are representations of a programming language's grammar. This step checks the grammar of the language and whether the program is written within the language boundary. The second phase is analysing, which builds the parsing tree based on the result of the previous phase [10]. It aims to produce a parsing tree in order to represent and validate language statements. The last phase is the semantic analysis which travels as the parsing tree in order to do a data type checking, object protection enforcement, etc.[11].

The parsing process is a process that checks the validity of a string of tokens i.e. the string is produced by a grammar. Every programming language has syntax grammars which are described by Context Free Grammar (CFG). A CFG [15]  $G$ :

$$G = (T, N, P, S)$$

- $T$  is a finite set of terminals. Each terminal symbol represents a basic element in a language. For example, in the programming language Java: ';', 'for', '{' are considered terminals.
- $N$  is a finite set of non terminals. Each non terminal is a symbol containing a string of terminals.
- $P$  stands for productions which regulate the grammar rules of expanding non terminals. Hence each non terminal can be expanded to terminal or non terminal symbols. For instance, the 'for' statement in Java has the following form:

$$statement \rightarrow \text{for } expression \{ statement \}$$

Where 'for', '{' and '}' are terminals; and *statement* and *expression* are non terminals and can generate more terminals, non terminals or both.

- $S$  is a non terminal symbol representing the start of a sentence in a language. Therefore, in order to be a valid sentence, it must start with  $S$ .

### JavaC

JavaC is a Java compiler which was developed by Sun Microsystems [2]. The compiler seems to be a well known Java compiler. It is open source software available for developing. The compiler can be criticised as being a command line compiler without a friendly interface. However, it can be argued that it is better for novice programmers to use in order to catch the language.

## BlueJ

BlueJ is a compiler designed for teaching Java programming to novice programmers who have a basic knowledge of Java and are, mainly, studying object-oriented [33]. It was developed by both Michael Kölling and John Rosenberg [32]. BlueJ is considered to be a project between The University of Kent in the United Kingdom and Deakin University in Australia[43]. The reason behind designing the compiler is that its developers think that other compilers are either quite complicated for students or too basic for students learning object-oriented[43]. BlueJ supports programmers by UML graphics representing the interaction between classes figure 1. Furthermore, BlueJ can help programmers by presenting extra help when there is an error figure 2. The figure shows a simple error in a piece of code where the compiler tries to give a hint in order to resolve the error. It can be seen, however, that the message content in this example is not very helpful, but this might be according to personal attitude.

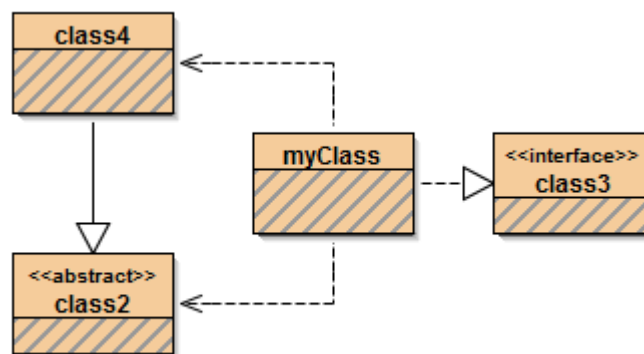


Figure 1: Graphical class interaction.

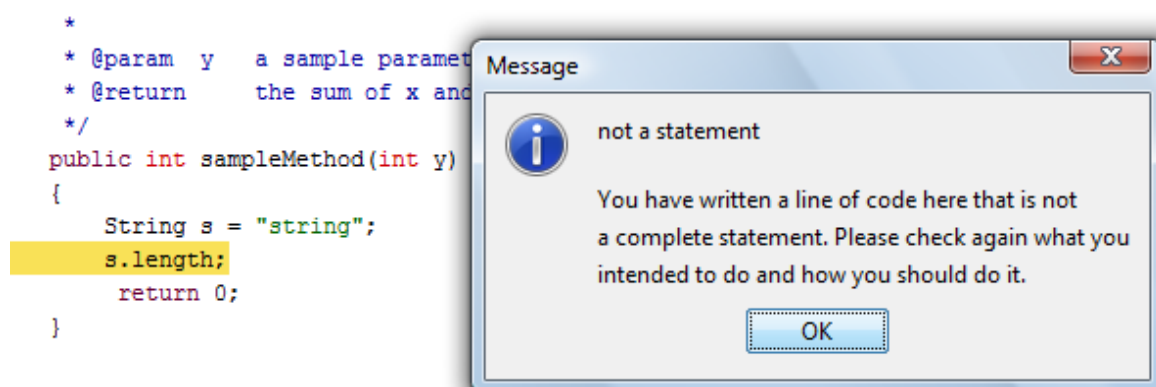


Figure 2: BlueJ offers help to solve errors.

BlueJ does not follow the error repair mechanism (section 3.1) because when the compiler encounters an error it simply reports it, without trying to continue parsing it. Some people would argue that this is not the preferred way to deal with it since it requires time and effort to fix the first

error then recompile and fix other errors in a serial way [23,49]. Others, however, argue that reporting other errors may include errors that are associated with other errors, which means solving previous ones leads to solve them by just recompiling.

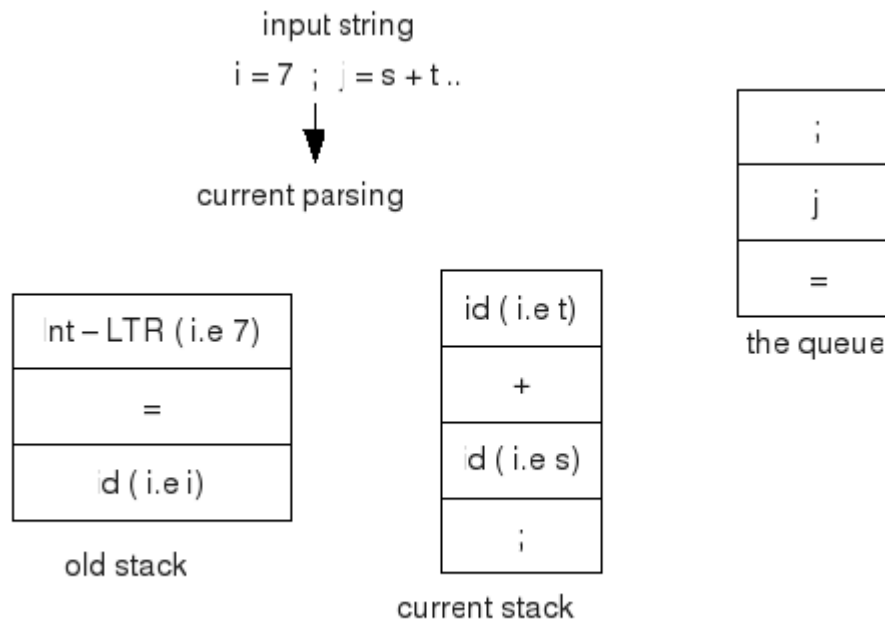
ANTLR (Another Tool for Language Recognition) [38] is a parser generator tool on which BlueJ is based. Studying ANTLR allows us to understand in more depth how BlueJ behaves with input code while compiling it. ANTLR has many features that make it more desirable for compiler related issues. ANTLR, for instance, is considered simple to use for inexpert parsing. The main feature of ANTLR in our work is that it has a manual mechanism called *parser exception handling* that allows a high quality of developing error handling [38]. In addition to the manual mechanism, it provides an automatic method of error recovery and repair [38]. Finally, ANTLR seems to be a product project as well as a research project because it was written in C language [38] with a main objective to keep it highly portable. That makes ANTLR more effectively integrated with other applications.

### 3.3 Error Repair Algorithms

This section discusses different algorithms that can be used in enhancing and designing new methods of repairing compiler errors. Burke-Fisher, McKenzie et al. and *set follow* are algorithms that have been, in general, implemented in compilers to allow parsing action to continue parsing a piece of code after detecting an error by either finding a solution or simply finding a delimiter such as a semicolon in Java. However, those algorithms cannot guarantee the repair which is employed is consistent with the programmer's intention. It can allow continuing parsing to find other errors in a piece of code. It can be effective in some cases but not always as it will describe later.

#### The Burke-Fisher Algorithm

Burke- Fisher is an algorithm which was introduced by Burke and Fisher as a PhD [49]. The algorithm aims to let a parser recover the error after detecting it by finding a group of fixes. Then, the parser can continue to find other errors. The idea of this algorithm is based on locating a group of tokens and storing them in a temporary queue. Furthermore, there are two stacks used by the algorithm: the first is called *current* and the second is called *old*. Old stack stores validated tokens (i.e. tokens that parsed successfully) while current stacks keep an arbitrary number of tokens. This action is taken before parsing action takes place. The size of the queue is  $M$ , i.e. the queue holds  $M$  of tokens each time the parsing action is executed. If an error occurs within this group of tokens, the parser is able to trace back to the error from the tokens stored in the queue. This method looks ahead where it is possible to look for more tokens to be examined. The algorithm, then, applies either insertion of a new symbol, deletion or replacement in order to solve the error. This strategy is efficient in terms of discovering errors that are symptoms of other errors but not when the error is by itself. [14]



*Figure 3: Burke - Fisher works with queue size 3*

Figure 3 shows a simple example of the Burke- Fisher error recovery algorithm where there is a temporary queue with size 3. If, after 7, the semicolon is, for example, missed, that will cause an error. The algorithm will then try to find a group of fixes for it. For instance, a group of fixes can be a semi colon, =, +, etc.

Van Der Spek et al. (2005) argue, however, that this algorithm cannot always find a repair for the error. Their study states that it can recover an error when it is a symptom, but not always. Although this study does not give reasons to the claim, the case is obvious. Furthermore, it can be argued that the algorithm has an insertion process as a way of recovering errors which can cause further errors and repeat the insertion step many times which may lead to infinite loop. However, they do not list reasons why it does not always work. In addition to that, it is claimed that the Burke – Fisher error recovery algorithm suffers from two problems. The first is that in order to insert or delete tokens, k number of token which are held in the queue should be set. Secondly, it affects the parser by delaying the semantic actions.

Another algorithm which uses the same principle is called the McKenzie et al. algorithm. This algorithm uses deterministic parsing automata (DPA) see [35] instead of using extra tables. It maintains a queue ordered by the cost of insertion or deletion symbols. The queue stores a list of configurations. Each configuration [15] is a group of four elements (S, I, D, C) where

- S is a stack which contains states.
- I and D are groups of inserted and deleted symbols.
- C is the cost of applying the change (i.e. insertion or deletion)

On one hand the algorithm, in theory, can repair any encountered error. On the other hand, to do this in practise requires time which leads to slowing down of the parsing process. That is because the algorithm invokes an intensive search to find a repair including all possible terminal tokens [29]. Other algorithms that use similar principles are: the Dain algorithm and the Fischer and Mauney algorithm [19].

### **The Set Follow Algorithm**

The idea of this algorithm is to produce a list of tokens that might occur after a token called a parser token where a parser token is generated by a parser. In other words, a parser generates a token (parser token), therefore the job of the algorithm is to design a group of possible tokens which can be placed after the parser token. This list of possible tokens is called *set follow* [49].

This algorithm can be employed in order to find a set of possible solutions to an error [29]. For example, suppose a token is followed by another token which causes an error (let this be called the error token). This error token can be examined as a possible misspelling error to one or more of the tokens in the follow set. This process, therefore, leads to the creation of another set called *possible solutions set* which contains all tokens that are possible solutions. The process of examination can be achieved by replacing the error token by each token found in the set follow and then checking whether or not the error is solved. If so, this token is considered to be a possible solution to this error, thus, it is inserted into the possible solutions set.

However, a question can be raised in employing this algorithm which is the effect of performance. For each token generated by a parser, there is a set follow that will be generated. Also, each token occurs in different places in a piece of code which means producing different sets of set follow which may or may not be similar. Furthermore, for each token in the set follow, it should be examined and should be decided whether it is a possible solution for an error token. On one hand, it can argued that there is no need to create a possible solutions set except in the case of founding errors. On the other hand, creating the set follow is effective due our target which is finding a way of suggesting a solution to programming errors.

### **3.4 Related work**

This section discusses works that are related to our work which is syntax error repair for a Java based parser which is written by [49]. This work aims to enhance the well known JavaC compiler to be able to report as many errors as possible in one compiling process. The second work is done by JECA [42] which aims to find some solutions to a specific group of errors for novice programmers.

## Syntax error repair for a java based parser

This work aims to improve the JavaC compiler by allowing the reporting of detected errors in one compilation process. JavaC originally reported errors sequentially one by one which requires a programmer to recompile its program as many times as errors are found in the code. To make it clear, suppose a piece of code has two errors. If it is compiled by JavaC, it will report the first error and then stop compiling until the programmer recompiles it again. Their motivation is that it is highly desirable that the compiler can report all errors in a single compilation instead of wasting time by recompiling the code. The method that is originally used by JavaC is called *non correcting error recovery* [23] because a parser stops parsing a piece of code when it encounters an error to let the programmer decide which to repair.

The study uses techniques in order to let the parser continue detecting other syntax errors: Firstly, this is done by implementing a variant of the Burke-Fisher algorithm. Secondly, the follow set algorithm is adapted and improved to maintain a good performance. Finally, panic mode is the last step when other steps have been followed and did not repair an error. In fact, the first two techniques are adapted and improved as a method of enhancing the performance. According to the study, the Burke-Fisher algorithm cannot be used in JavaC due to the limitation of its parser tree. In addition to this, although the work implemented follow set, it differs from the original follow set. That is because a token, e.g. X, has a different follow set which depends on its position in the input string in normal forms of follow set. In contrast, van Der Spek et al. introduce a variant of it by combining all follow sets for the token X in one big follow set [49]. The steps of their methodology are as follows when it encounters a token which causes an error and it is called an error token:

- The error token is compared to a list of expected tokens which could be the right one (i.e. the error token is a misspelling).
- If the error token resembles any of those tokens, it is added to a list of candidate tokens. This list contains a group of tokens representing a possible solution to the error.
- Then, check if each token is a real repair by inserting it before the error token. The follow set is used as way of checking as well as retrieving the follow set for the new token that is intended to be inserted.
- If the error token is considered a part of the follow set, then this new token is considered a repair token.

However, the work suffers from two drawbacks. Firstly, the improved algorithm causes less accuracy in finding a repair because the follow list for each token is basically a group of different follow sets for the same token in different positions. Therefore, a token in the follow is not necessarily a real follow for the token. Secondly, the maximum number of the temporary queue used in the Burke-Fisher algorithm is 25 [14], while the same queue in the work has 10 as a maximum number. This number is not chosen based on a theoretical study. Van Der Spek et al. argue that the chosen number shows acceptable results based on small experiments [49].



## Java error-correcting algorithm (JECA)

This work aims to get a piece of code which has some errors as an input and then tries to repair these errors in a way similar to the previous work and the error recovery algorithm. The difference, however, between this work and other works is that the modifying that is carried by JECA is invoked in a way that achieves the intention of the programmer. In contrast, in other works, for example the McKenzie et al. algorithm [35], the concept is to find a repair to let the parser parse the remaining input in order to find other subsequent errors. Consider the following code as an input to a parser:

```
For( iint x = 1; x < 7; x++)  
x =x + 3;  
int y
```

Obviously the previous code contains two errors in the first line where *For* and *iint* should be *for* and *int* respectively. According to [42], a typical compiler would produce an error message such as *)' expected* for the first two errors. In this case, the compiler continued parsing and produced a second error message which is *;' expected*. The second message is quite clear and gives a typical solution that is most likely what the programmer intended to do. In contrast, the first message is unclear since it does not suggest a solution. This is because normal compilers usually try to find a solution that allows for continuing parsing, in our example the sample technique is panic mode where it escaped to special terminal and continued parsing. The second error message is what JECA tries to do.

JECA work is a part of learning the Java system which aims to teach new students how to program in Java. The concept of teaching using JECA is to help students by giving them proper feedback so they can learn better. In order to achieve this target, JECA is designed to accommodate two methods. The first is called *TokenManager* where the piece of code is fed to it and then it tries to find a repair to errors using misspelling techniques including a list of keywords to be compared with erroneous tokens. This method, however, is very restricted to only include misspelling errors in keywords such as *int*, *class* ... etc. They argue that the target of the tool is novice programmers who usually tend to encounter such mistakes at the first stages of learning. The other method is using error recovery in the parsing phase by implementing the Burke-Fisher algorithm.

However, the method has some different changes from the original algorithm. Firstly, the whole code is stored in abstract data structure since pieces of code are very short with about 50 lines each. Secondly, JECA lets the programmer himself repair errors while the algorithm itself repairs them.

However, JECA has some drawbacks. First of all, the work is aimed at a group of programmers and ignores others. In addition, the assumption that the length of code is about 50 lines seems to be improper since even novice programmers can easily write far more than that. Furthermore, the work does not cover a wide range of different types of errors. For example, the work does not cover the case when a programmer calls a method without parenthesis. Finally, semantic errors are not within the scope of the tool. However, they are deemed to be crucial for programmers, especially novice ones.



## 4 Programming errors

This section concerns the study and analysis of common programming errors by looking at other studies that centre on this issue. It seems that it is vital to do this investigation of common errors because our work aims to understand what programmers would like their code to be but they did in wrong ways. This is a result of our intention to find solutions that stem from the point of view of programmers, involving both syntax and semantic programming errors. Syntax errors are errors of syntax that are not consistent with programming language grammar and they appear during compiling time [23]. Semantic errors are those which lead programs to behave unexpectedly [23]. This chapter is limited to syntax errors.

The study by Garner *et al.* [21] describes the problems that novice programmers face. The study covers 250 students in an introduction course. It describes problems that involve using tools for writing programs, understanding Java tasks, and naming variables, etc. In addition, it involves programming errors. The study found that basic mechanics, as it is called in the work, is ranked as one of the most difficult problems for novice programmers. Basic mechanics includes braces, brackets, semi colons, typos and spelling. However, the study does not provide details about which of the basic mechanics is the most common.

In the study by Jackson *et al* [27], they identify top common Java errors of novice programmers, arguing that their study is novel because they use an online IDE which logs all common errors in a database. Their study concludes with a group of ten common errors. However, they find that academic teachers tend to identify only half of them, which gives a clear idea about the difficulty of identifying common errors without studying the programmers' errors themselves. This criticism applies to the Flowers *et al* study, where they ask programming tutors about top common mistakes [43], while Ahmadzadeh *et al* study common errors in order to improve teaching ways of programming languages [9]. The latter are motivated by how understanding student mistakes can help with regard to delivering an effective method of programming teaching. Their method is to collect error messages and sources of code that generate such errors and they analyze about 108,000 error records. Their results conclude that about a third of programming errors are syntax-based, while the majority, the other two thirds, is semantic.

Also relevant here is the study made by Thompson, which investigates common programming errors [43]. The methodology of the study is based on the use of log analyzer software to record the participant's code along with their errors. This study involves compiler and run time errors, focusing on the mean time that is required to solve programming errors, a method which can be considered a novel way of studying programming errors. It can be viewed as a valuable resource for our work because it can be seen as a sign of the need for a solution to these errors, which require a long time to be fixed. Another study was carried out by Jadudd [28], where common errors encountered by novice programmers who use BlueJ for programming are looked at. In it, the behavior of about 63 students is analyzed during lab sessions, a process which, at the same time, unearths their common mistakes. The study concludes that the most common errors, which were the five errors achieved in 58% of all committed errors, are as follows:

1. *Missing semicolons* at about 18%.
2. *Unknown symbol: variable* at about 12%.
3. *Bracket expected* at about 12%.
4. *Illegal start of expression* at about 9%.
5. *Unknown symbol: classed* at about 7%.

As shown, the top error is *missing semicolons* at about 18% out of all errors. However, the second and last seem to be because of the same reasons; either spelling errors or using undefined variables (or methods). Therefore, it can be ranked in first place because it accounted for about 19%. The study by [43] seems to agree with these findings. The study stated that *UndefinedName* is the most common error which is the same error called *unknown symbol* in the previous study. The reason for this, according to [43], is forgetting a variable declaration or misspelling the name, which can be a variable of a method.

Table 1 shows the most common error messages by a compiler, according to [27]. The top common error message is *cannot resolve symbol* by achieving about 14.6% of all messages, then, the message; *expected* comes in the second place at about 8.5%. It is worth mentioning that the message itself may differ according the compiler that is used. The first message happens as a result of spelling errors and using undefined methods or words. Spelling errors in this context cover using synonyms of variables or methods. It can be noticed that each error message in the table provides a repair for the corresponding error. For example, in the case of missing a semicolon, a compiler reports the line number in addition to the stated and clear solution; a semi colon is missing. However, the exception in this case is the most common error message where the compiler reports the line number without providing any further help. It can be argued that in case of using undeclared variables or methods whether it is impossible to give a repair while in case of spelling problem the compiler should try finding a match.

*Table 1: Top Error Messages*

Error Message	Occurrences
Cannot resolve symbol	14.6%
; Expected	8.5%
Class or interface expected	4.6%
<identifier>Expected	4.5%
) Expected	3.8%
} Expected	2.3%

#### 4.1 Cannot Find Symbol Error

As it was described before, this error is ranked the top repeated programming error. In this section, we applied a piece of code having this type of error to JavaC and BlueJ to find out how those

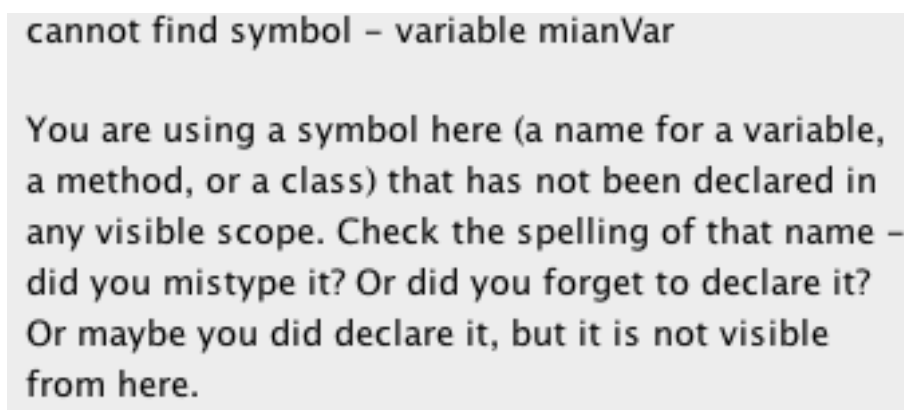
compilers would behave. The following code was used.

```
public class test
{
    public static void main(String[] args)
    {
        intmainVar;
        mianVar=7;
    }
}
```

Firstly, the error in this simple code can be confusing as the eye can easily skip to the difference between *mainVar* and *mianVar* which is swapping a and i. This may lead a programmer to spend some time figuring out what is the problem, especially if the source code has ten of hundreds of lines and thousands of variables. JavaC generated the following message:

```
test.java:11: cannot find symbol
symbol : variable mianVar
location: class test
    mianVar=7;
      ^
1 error
```

As it can be seen, the compiler reported the line number and the variable name but it, however, did not provide the programmer with a repair for this error which can be simply the correct variable's name. By compiling the same example using BlueJ, it generates a message with extra help provided by the compiler shown in Figure 4. The message seems informative and suggests a repair by declaring the variable. However, this might help in solving this problem but not always, as in our case. It mentioned a spelling error, but without giving a suggested word for it. It can be thought that if the compiler provided a correct form of word, then it would be considered a helpful message.

The image shows a screenshot of an error message from the BlueJ IDE. The message is displayed in a light gray box with a dark gray border. The text is as follows:

**cannot find symbol – variable mianVar**

**You are using a symbol here (a name for a variable, a method, or a class) that has not been declared in any visible scope. Check the spelling of that name – did you mistype it? Or did you forget to declare it? Or maybe you did declare it, but it is not visible from here.**

*Figure 4: Error message by BlueJ with extra help*

The reason behind this error is forgetting to declare a variable before using it in the first place.

Obviously, in this case a compiler cannot provide help to programmers to solve this problem. Secondly, the lifetime of a variable can be a reason behind this error. For example, a variable might be declared in a method, then used in another which is out of the scope of the previous method which requires a new declaration. Thirdly, spelling errors play a key role in this error. This includes misspelling variable names and methods as well. It may involve using synonyms for variables or methods. A programmer may call, for instance, a method *size* to get a size, while it is defined as *length*. This quite often happens when dealing with arrays, arraylists and strings. Therefore, we have employed spelling correction algorithms in compilers in order to find matches for this type of error.

## 5 Spelling Correction Algorithms

Misspelling words is a common problem that can be encountered while writing; either with hand writing or using a word processor. A study done by Damerau revealed that about 80% of misspelled words happen as a result of a single error [17]. This study is one of the first studies in the field of detecting spelling errors in computers. This section describes the processes of correcting a misspelled word that are used in general, regardless of an application's field. Then, it describes two algorithms that can be employed in order to find a match for misspelled words.

The problem seems to also happen with programmers as they write programs. Several reasons might be behind the problem. Firstly, a programmer may commit a misspelling mistake due to pressing the wrong; for example, a programmer writes *id* instead of *if* due to the closeness of the *F* and *D* keys. Moreover, fast typers usually make silly mistakes due to their fast typing [44]. Furthermore, a lack of experience or knowledge can be seen as another reason as well. This involves the experience of the programming language itself; for example, a method called *cmd* instead of a command in a class. Also, a lack of knowledge in the English language itself could be a problem for programmers whose first language is not English as programming languages, mainly, written, in English such as Java.

Fixing misspelled words manually might be considered an easy task, though it can be a time consuming and boring one. However, detecting the error itself is not always a smooth job. It is well known that the eye can read a word such as *knowledge* correctly without noticing the swapping of characters as a mistake. Thus, a compiler might report the word as an undefined symbol while the programmer himself does not figure out what the mistake is. Therefore, designing a method to catch a misspelled word regardless whether it is a keyword or a defined variable as long the suggested word is consistent with the programmer intention.

This section concerns different algorithms to be included in our experiments. The first known work of designing a spelling correction method was done by Glantz [22] in 1957 [24]. The domain of the study simply considered a way to recognise misspelled words in an English plaintext. Then, each misspelled word is compared with the correct word in the dictionary [36]. Other works were designed for different domains; for example, for correcting misspelled names, search queries, word processors, etc. Freeman's work [20] was considered to be the first study which focused on correcting programming spelling errors in compilers [36].

### 5.1 Spelling Correction Process

There are two basic steps in the process of correcting misspelled words [18]. Firstly, the misspelled word should be detected. Before that, a misspelled word needs to be defined. The definition can be different depending on the domain of the problem. In word processing applications, for example, the definition of misspelled words, obviously, is writing a word in the wrong format. Detecting is achieved simply by comparing it with a list of predefined words to decide whether the word is

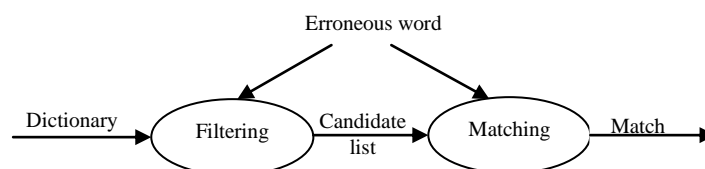
recognised or it is a misspelled error. In case of compilers and programming languages, however, this is not always true because a computer programmer has the ability to define variables on his own and they are not necessarily in the correct form. Consider the following example:

```
intxVarCord;  
.....  
xVarCod = 6;
```

*Figure 5: Declaring own words*

In the previous figure, although *xVarCord* is in the wrong format compared to a word processor, it remains the right choice in programming languages. In contrast, *xVarCod* is seen as a programming error. In our work, it is assumed to be a misspelling programming error. In our work, detecting a misspelled word is not our target as any person can recognise undefined words. However, we are still focusing on analysing the erroneous word to decide if it is really a misspelled word or another error. For example, in the Java programming language, using a variable without defining it is a syntax error. Our work is to find out whether the intention was to use a predefined variable or just to use a variable without defining it. This could be seen as the next step in the field.

The second basic step of correction is to correct a detected erroneous word. This step can also be classified into different steps. The first is to find a group of candidate words which are kept in a dictionary. The number of candidate words is limited to a reasonable number due the efficiency of finding a solution. This depends on filtering process. This process aims to discard irrelevant words as soon as possible to produce a smaller list to be compared with the erroneous word (see later). Then, a word should be chosen from amongst this group of words to be considered as a repair for the misspelled word in a process called matching process. The transformation may involve one or more added, deleted or substituted characters. Figure 6 shows how the process is achieved.



*Figure 6: Spelling Correction Process*

In figure 6, spelling correction process is achieved by comparing the erroneous word with the dictionary in filtering process. Filtering is used to increase the efficiency of the process by decreasing the number of words that will be compared in the matching process. Disagreement technique can be used as a filtering method (explained later). After producing a list of words that can be considered as repair, each word is compared with the matching algorithm in order to find the best match. This can be achieved using two algorithms: Four-way algorithm and Editex algorithm.

The match result can be either a single word or a list of words depending on the context of an application. For example, in word processing applications, the application prints out a list of possible correct words and the user choose among them. In contrast, our work aims to find the repair by suggesting a single correct word only.

## 5.2 Disagreement Technique

The comparison algorithm of two strings: the erroneous word and a candidate word, seems to be the most important process in spelling correction [18] and require time to finish. Therefore, comparing each word in a dictionary might be time consuming especially if the dictionary has a huge size of words involving relevant words which can be repairs and irrelevant words. To limit the number of unwanted words, a disagreement threshold is proposed. It involves counting the difference between two strings in a percentage [31]. For example, the disagreement of “details” and “dmteils” is about 28% ( $2/7 \times 100$ ), while “the” and “jhe” is 33 %, because the erroneous word “jhe” differs by one letter out of three in the candidate word.

The aim of a disagreement threshold is to find good candidate words and discard or limit the unwanted words. However, it might ignore good words if the threshold is too low or it could involve unwanted words. Therefore, an adaptive disagreement threshold is proposed [18], which aims to find a candidate word without including unwanted words. It simply starts with a relatively high threshold, say 50%. Then, after recognising an N number of candidate words, the threshold dynamically sets to a lower value, say 30%.

## 5.3 Four-Way Algorithm

This algorithm is based on a previous algorithm called three ways match approach that is presented by Lee for spelling correction [45]. The method is designed to correct spelling mistakes in a parser that used to understand and compile natural languages. Then, the approach is extended in four-ways [18]. Both algorithms aim to find a repair for erroneous words in natural languages. Elmi proposed a work on designing a parser for natural language which involves an algorithm for correcting misspelled words [18]. The steps of correcting an erroneous word are as follows. First, the erroneous word is compared with a list of correct words kept in a lexicon. Then, words with a minimum edit distance, i.e. minimum number of letters required to be added, deleted or substituted, are considered to be candidate words. Thirdly, the candidate words are ordered by the weight of each. The weight counting will be explained later. Finally, the syntax and semantic of the context can be employed in order to find the best match.

### Error Weights Assignment

Error weight is used to find the best match for an erroneous word amongst a group of candidate words. The comparison is done by comparing two strings, letter by letter, and if there is an error, it will be counted as Elmi proposed in [18]. However, we have changed the categories to be applied in compilers as follows:

1. The case of a character: This type of error is considered as a character substitution error.
2. Missing character: the study differs when the missing character is at the end or not. In our case, we do not distinguish.
3. Adding a character: in this case, the study differs as well when the character should be added at the end or not. In our case we just consider them the same.
4. Changing characters' order: it is called reversed order, when a writer writes the correct letters but in wrong order. For example, "totoh" for "tooth".

The table below shows examples of each case with the considered weight for it.

*Table 2: Error weight*

Error Type	Weight
Reversed order	60
Character substitution	70
Missing character	80
Adding character	80

### **The Comparison Algorithm**

The algorithm was designed by Lee and was extended by Elmi. It is called the four-way approach because the approach compares four different letters: two letters from each string. The first string, S1, is the erroneous word, whereas the second, S2, is the word from the lexicon. The following figure 7 explains how it works, where n means the position n of a letter in the string S1, and m is the position of a letter in the string S2.



```

if n != m then
  if n + 1 == m && n == m + 1 then
    error "reversed order"
  else if n + 1 != m && n == m + 1 then
    if n + 1 == m + 1 then
      error "character substitution"
    else
      error "missing character"
    end if
  else if n + 1 == m && n != m + 1 then
    if n + 1 == m + 1 then
      error "character substitution"
    else
      error "adding character"
    end if
  else
    error "character substitution*"
  end if
end if

```

*Figure 7: Four-Way Algorithm*

The algorithm compares two words letter by letter and assigns an error weight accordingly. A word can have one or more of error types listed earlier. The following table shows an example of each error type with its error weight. Notice that the last example *actrv* is a misspelled word of *active* which it has two error types: character missing (80) and character substitution (70). Therefore, the total error weight is 150.

*Table 3: Error type examples*

Erroneous Word	Correct Word	Error Type	Weight
osme	some	Reversed order	60
royel	royal	Character substitution	70
th	the	Missing character	80
exxample	example	Adding character	80
actrv	active	Substitution & missing	150

## Four-Way in Compilers

The algorithm is designed to figure out misspelling problems in natural languages. The aim in this section is to employ the algorithm to work with programming languages. The main difference between human languages and programming languages is that misspelled words are clearly defined in natural languages, which is any word that is not recognised in a lexicon. In contrast, some words

can be seen as correct words from the point view of a compiler even though they are misspelled word in terms of a human language. This is because a programmer is allowed to define variables or methods using his own words, as long as he is bounded by the programming language rules. In this case, the comparison in a piece of code has a misspelling error that differs in different situations. The lexicon that is used to compare the erroneous word in the compiler contains the keywords that are used in a programming language; for example, *int* and *if* in Java. Therefore, the size of the lexicon would be very short compared with the lexicon of a natural language.

In addition to that, as the programmer has the ability to write variables or methods of his own, he probably writes a variable then uses it again, but in the wrong format which is a syntax error as it is shown in the below example. The variable *firstCharacter* is considered a correct word in Java compilers, whereas *firstCharater* is considered an undefined symbol by any typical Java compiler.

```
void fun1()
{
    intfirstCharacter;
    firstCharater = 7;
}
```

Therefore, we amend the dictionary to involve the variable and the methods that have been used by the programmer prior the occurrence of the error. In this case, special characters, such as an underscore or number, should be taken into consideration. This seems to be missed by the original algorithm since it checks the spelling of normal words that are defined in a natural language. Our method involves giving words that contain special characters more priority to be checked.

## 5.4 Editex Algorithm

We have developed a new method in checking misspelled words in a piece of code based on the Editex algorithm, designed by Zobel and Dart [48]. The algorithm is based on the idea of combining Soundex coding (explained later) with an edit distance algorithm.

### Edit Distance

Edit distance is an algorithm that is used to measure the distance between two strings [40] in order to find the minimum number of letters required to transform a string into another string [12]. It was designed by a Russian scientist called Vladimir Levenshtein in 1965. Therefore, the algorithm is called the Levenshtein distance algorithm. The goal of the algorithm is to measure the similarity between two strings [47]. The mechanism of the algorithm is to construct an array of two dimensions representing the letters of the two compared strings. Figure 8 shows the step required to get the distance between the two strings.

```

n = length of S
m = length of T
for i=1 to n
    for j=1 to m
        distance[i,j] = min(
            distance[i-1, j] + 1,
            distance[i, j-1] + 1,
            distance[i-1, j-1] + S[i] == T[j]? 0:1
        )
distance(S,T) = distance[n,m]

```

Figure 8: Edit Distance Algorithm

Figure 8 summarizes the steps that are required to compute the distance between two strings, S and T. They start by initializing an array called the *distance* of two dimensions  $n \times m$  where  $n$  represents the length of S, whereas  $m$  represents the length of T. Cells from 0 to  $n$  and from 0 to  $m$  are initialized from 0 to  $n$  and from 0 to  $m$  respectively. Each cell on position  $[i,j]$ , then, is assigned a value that is a minimum of either:

- The value of the cell  $[i-1,j] + 1$ ,
- The value of the cell  $[i,j-1] + 1$ , or
- The value of the cell  $[i-1,j-1] + \text{cost}$ .

Table 4: Edit distance, first iteration

		<b>m</b>	<b>a</b>	<b>x</b>	<b>l</b>	<b>n</b>	<b>u</b>	<b>m</b>
	0	1	2	3	4	5	6	7
<b>m</b>	1	0						
<b>a</b>	2	1						
<b>x</b>	3	2						
<b>i</b>	4	3						
<b>m</b>	5	4						
<b>u</b>	6	5						
<b>m</b>	7	6						

Cost means if the letter at position  $i$  of the string S is equal to the letter at position  $j$  of the string T, then the cost is 0. Otherwise, it is 1. For example, *maxlnum* is a misspelled word for *maximum* and as shown in table 4 after finishing the iteration  $i = 1$ . Table 5 shows the complete matrix after finishing assigning the value. As shown in table 5 the minimum number of letters required to transform the erroneous word into the correct form is 2.

*Table 5: Edit distance, finished*

		<b>m</b>	<b>a</b>	<b>x</b>	<b>l</b>	<b>n</b>	<b>u</b>	<b>m</b>
	0	1	2	3	4	5	6	7
<b>m</b>	1	0	1	2	3	4	5	6
<b>a</b>	2	1	0	1	2	3	4	5
<b>x</b>	3	2	1	0	1	2	3	4
<b>i</b>	4	3	2	1	1	2	3	4
<b>m</b>	5	4	3	2	2	2	3	3
<b>u</b>	6	5	4	3	3	3	2	3
<b>m</b>	7	6	5	4	4	4	3	<b>2</b>

## The Soundex Algorithm

Soundex is a phonetic encoding algorithm developed as a method of representing names of individuals in the United States in 1880 [13]. The methodology of the algorithm is to represent English words in a code following these rules [4,16]:-

- The first character in a Soundex code is the first letter in the corresponding English word.
- Remove the following letters from the word: A, E, H, I, O, U, W and Y, except when one of them is a first letter.
- Converting the remaining letters from the corresponding word after converting the previous letters into digits as described in table 6. Note, that the converting is only for the second, third, and fourth letters and the remaining letters are just discarded. If there are less than 3 letters, put zeros.

*Table 6: Soundex Code Group*

<b>Letters</b>	<b>Soundex digit</b>
B, F, P, V	1
C, G, J, K, Q, S, X, Z	2
D, T	3
L	4
M, N	5
R	6

For example, to encode James:

- Take J as the first character.
- Then, remove a and e.
- converting the remaining letters to digits, m: 5 and s: 2.
- Soundex encoding of James is J520. Note James has the same Soundex encoding.

The Soundex algorithm is widely used in retrieving data in databases where names are commonly misspelled. The algorithm, however, suffers from some limitations. Firstly, the algorithm depends on the order of the word [13]. This can be considered a drawback when it comes to finding a misspelled word where a permutation is required in order to restore the correct word. For example, *collect* has the encoding C442. In contrast, the misspelled version of it, *colletc*, has a different encoding which is C443. In the second place, the Soundex code consists of four characters: a letter with three digits which can have the same Soundex code for two different English words.

However, the Soundex representation of words is claimed to be poor at matching strings according to [26]. Therefore, Editex employs derived code groups. Table 7 lists each group of letters with its code. Compared to the Soundex algorithm, Editex uses 10 different codes instead of 7 for Soundex which can be seen as a way of widening the code required to represents words.

*Table 7: Editex code group*

Letters Group	Code
A, E, I, O, U, Y, H, W	0
B, P	1
C, G, J, K, Q	2
D, T	3
L	4
M, N	5
R	6
F, V	7
S, X, Z	8

## Editex Steps

Figure 9 surmises the steps required to accomplish the algorithm. *edit* is a function that computes the distance between two words: the erroneous word ‘s’ and a word from the lexicon where ‘i’ and ‘j’ are the positions of each letter in both words as ‘s1...si’ represent letters if word s and the same is true for the word t which is represented by letters t1 to tj. The algorithm considers the Editex code

through the function  $r$ . The function takes two letters, say 'a' and 'b', then returns 0 if they are identical, 1 if they are within the same group, or 2 if not. For example, if 'a' is 'f' and 'b' 'y',  $r$  function returns 2 because they are from different groups. Each candidate word from the lexicon has its score which is computed by the function *edit*. The word with the lowest score is chosen to be the best match.

$$\begin{aligned}
 \text{edit}(0,0) &= 0 \\
 \text{edit}(i,0) &= \text{edit}(i-1,0) + r(s_{i-1}, s_i) \\
 \text{edit}(0,j) &= \text{edit}(0,j-1) + r(t_{j-1}, t_j) \\
 \text{edit}(i,j) &= \min [ \\
 &\quad \text{edit}(i-1,j) + r(s_{i-1}, s_i), \\
 &\quad \text{edit}(i,j-1) + r(t_{j-1}, t_j), \\
 &\quad \text{edit}(i-1,j-1) + r(s_i, t_j) ]
 \end{aligned}$$

Figure 9: Editex algorithm

In figure 9, an example that has been described earlier in this section is reused but using the Editex algorithm. The figure is shown after finishing constructing the matrix. The result of the distance between the erroneous word *maxlnum* and the correct word *maximum* is 3. In comparison to the previous example, using the Edit distance gives a result of 2, because the difference between *maxlnum* and *maximum* is two letters and because  $i$  and  $l$  are from a different Editex group, then it is counted as 2 while  $n$  and  $m$  are from the same Editex group, then it is counted as 1. Therefore, the total is 3.

The algorithm is claimed to be poor in terms of performance due to the fact that it requires  $\mathcal{O}(m \times n)$  where  $n$  is the number of letters in the erroneous word and  $m$  is the number of letter of a word from candidate words from the lexicon[26]. This claim is alleviated in our work by two reasons: the first is that the lexicon's size is relatively small compared with the one used in word processors; secondly, we have a limited the number of candidate words by ignoring irrelevant words by using a disagreement threshold, which allows the discarding of unwanted words in the early stages.

*Table 8: Editex example*

		<b>m</b>	<b>a</b>	<b>x</b>	<b>l</b>	<b>n</b>	<b>u</b>	<b>m</b>
	0	1	2	3	4	5	6	7
<b>m</b>	1	0	2	4	6	7	9	9
<b>a</b>	2	2	0	1	3	5	6	8
<b>x</b>	3	4	1	0	2	4	6	7
<b>i</b>	4	6	2	2	2	4	5	7
<b>m</b>	5	6	4	4	4	3	5	7
<b>u</b>	6	7	5	6	6	5	3	5
<b>m</b>	7	8	7	7	8	6	5	<b>3</b>

## 5.5 Summary

In summary, this section presented the spelling correction process that consists of the following. Firstly, it compares the erroneous word with a predefined dictionary which contains all possible correct words. This comparison aims to come up with a candidate list of possible solutions in order to save the required time in matching. Then, the matching algorithm is an algorithm which compares the erroneous word with each word in the candidate list to find the best match. In this section, two algorithms have been discussed which are the Four-Way and Editex algorithms.

## 6 Methodology

This section discusses our method in this project. The section starts by explaining the data collection. This part describes the way of building the dictionary which contains possible solutions for an erroneous word. In addition to that, it discusses how we collect and build the test data. Moreover, it discusses our methodology of running the test and which of our test measures to judge an algorithm is better. The second part of the section concerns the application that has been used to write and implement our project.

### 6.1 Data Collection

#### The Dictionary

As described earlier, the misspelling correction process requires a dictionary to compare the erroneous word in order to find a match. In most cases, the dictionary usually contains a list of proper words. Proper words mean words that are defined in a particular natural language. For example, a dictionary that is used in an English word processing application should contain a list of English words. However, our project requires a dictionary containing words that are not considered proper words. The dictionary should contain Java keywords, predefined methods, predefined variables, abstracts and packages. In addition, a user can define variables and methods on his own and these should be included in the dictionary. These differ from one user to another; therefore, they are included only during the running of a particular source code.

In order to build the dictionary, we have designed two tools. The first is to read and analyse Java source code files from websites [34][1] to get some variables, methods and keywords. Our project aims to enhance Java compilers in particular; therefore, we collected about 1,000 Java source code files. The second tool was used to filter the collected words and keep them in a file called *dictionary.txt*. About 200,000 words were collected by the first tool whereas the second tool filtered it to make it just above 10,000. The filtering process involves deleting repeated and unrelated words. Furthermore, we have collected predefined methods and classes from [7]. In addition, we posted a question on a Java formal forum [8] regarding more resources about Java that can be useful in our work.

#### Test Data

In order to examine our project, we need a list of Java source code files that have programming errors. There are several studies which have studied programming errors in general. However, none of these provide an archive of programming error samples. Thus, we have used another way by collecting a list of words that are considered top misspelled English words, according to [6][5][3]. These resources provide common misspelled words with the correct form of each. This gives us the ability to use it as real test data. The test data is divided into several categories in order to investigate the behavior of the algorithms that have been used. Table 9 presents those categories.



Table 9: Test Types with Examples

Test Type	Example	Correct Form
Missing single letter	special	specal
Missing many letters	according	ccoring
Adding single letter	exam	exram
Adding many letters	description	deescription
Substitution single letter	between	vetween
Substitution many letters	place	zkace
Special characters single letters	max_int_value	max_in_value
Special characters many letters	min_double_Val	min_ydouble_Val

The single letter in Table 9 means that the misspelled word is deferent from the correct word by only one letter, regardless of whether this difference is a result of adding, missing or substituting a letter. Substituting a letter in our case means replacing a letter incorrectly by another. In other words, it can be seen as combining a missing a letter and adding it instead of another. In contrast, many letters means the distance between the misspelled word and the correct is more than one letter. A special character means any word that is not considered a proper word as a result of adding numbers or special characters, such as an underscore character.

## 6.2 Test Measure

In order to compare the Four-Way algorithm with the Editex algorithm, we considered two test measures which are accuracy and time. Accuracy means the number of correctly recognised misspelled words divided by the total words. The equation is shown below:

$$\frac{\text{number of correct misspelled words}}{\text{total number of misspelled words}}$$

A recognised word means the word that a programmer originally meant but wrote it in the wrong format. The other test measure is the time that is required to accomplish the task, regardless of whether the result is correct or not. The two measures are fundamental in the spell checking process according to the study by Hogde and Austin [26]. The study focused on the accuracy of retrieval names or what is called recall, in addition to the speed of finding a repair. Their study examined misspelled names, whereas our approach examines programming errors.

The test data is divided into categories, as shown in Table 9. Each is tested by each algorithm in order to compare each type of test among the algorithms. Furthermore, in the final stage all the data is tested in one step in order to find out the accuracy and required time for each algorithm.

However, it can be claimed that the best test is done by applying our work to a group of programmers in order to study the effectiveness of the project. This claim is certainly true, but cannot be carried out due to the time limit of this project. In addition to the time limit, because the project was written and implemented during the summer, hence almost all undergraduate students in the Computer Science department were away on vacation. Therefore, finding voluntary programmers is difficult. In summary, this test is discussed in the future work section in more detail and is considered to be an aspect of future work.

### **6.3 Used Applications**

Several applications have been used in implementing our project and others have been used in order to write this thesis. This project is written in Java using Gedit editor software version 2.26.3. The source is compiled under JavaC compiler version 7. Both JavaC and BlueJ version 3.0.2 compiler have been used in order to get the error messages of java source files. Our aim is to study those messages as described in section2. Furthermore, Microsoft Word 2007 and Excel 2007 have been used to record result data and write this thesis.

### **6.4 Summary**

In this section, we have presented our method of collecting data in order to be used as a dictionary in our project, although the project requires adding a list of words each time it runs. In addition to collecting data for the dictionary, we have discussed the collection of test data. We have mentioned that due to the lack of archive files containing programming errors, we have collected a list of common English misspelled words. Furthermore, an explanation of our method of testing our tool is described. Finally, we have listed the applications that have been used to write and implement our work.

## 7 Project Design

This section demonstrates the design of our project. It starts with a general overview to our work. Then, it moves on to describe the process of building the dictionary which is used in the project. Moreover, a new way of filtering is proposed in this section. After that, the designing of a process of detecting the errors and finding the erroneous words is discussed. We present the design of the Four-Way and Editex algorithms. Despite both algorithms showing a reasonable level of results, we have designed a group of features that can improve the algorithms. The features are Edit & Compile, Synonyms List, Context and Learn by Experience. The features are discussed in this section. A new feature is designed, as well, which is called Further Repairing. The feature considers continuing to find and repair other programming errors in order to save the time of a programmer. Finally, the final design is shown and discussed. It is our final approach that we follow in running our experiments in the next section.

### Overview

In our project, we have combined compiling with a spelling correction process. Therefore, the tool starts by getting a source code as an input. Then, it uses a compiler such as JavaC in order to get the error message. A tool gets this message and analyses it in order to get the erroneous word. A tool analyses the source code in order to get a list of words that will be added to the dictionary. Then, a matching algorithm accepts the dictionary and the erroneous word then produces a match based on the algorithm. However, later in this section, some features will be added to enhance the efficiency of the work.

### 7.1 Building the Dictionary

The dictionary in our work consists of two parts: the first is a predefined text file containing words that can be useful for any Java source code file being processed in the project. The second is by analysing the input source code in order to get words that can be useful such as variables or methods that have been defined by the writer of the source code. As a result, this process outputs a list of all possible repairs for this particular file. It is particular for this file because it involves the words generated by the programmer which can be of no benefit for other Java files.

### 7.2 Agreement Threshold

Section 5.2 discusses the role of the disagreement threshold in filtering the dictionary. The filtering process is employed in spelling correction in order to increase the efficiency by decreasing the number of words in the dictionary. We have designed a new technique which is based on the disagreement threshold. The basic idea of the disagreement technique is to discard unwanted words as early as possible. In contrast, the agreement technique aims to involve preferable words by setting a threshold (40%). Once a word from the dictionary is similar to the erroneous word, it is considered as a candidate word, then it is added to the candidate table. The aim of this technique is

to involve words that can be repaired in order to increase the accuracy of the algorithms. In contrast, the disagreement technique aims to discard unwanted words to preserve the performance

However, it can be argued that this technique will not increase the efficiency of the misspelling correction process the same as the disagreement technique. That is true to a certain extent. Firstly, the technique maintains accuracy in the first place and then the speed. Secondly, the agreement approach might work like the disagreement technique when the threshold is 60%.

### 7.3 Four-Way Algorithm

We have implemented the algorithm as described in section 5.3. On one hand, the Four-Way algorithm can be applied to our work without extending the algorithm which can be seen as a positive point compared to Editex algorithm, which requires alteration in order to be suitable. On the other hand, the accuracy of the algorithm is lower compared to the Editex algorithm. This will be discussed in the next chapter. We have added character case to Table 2: Error weight as Java is a case sensitive language. In fact, the weight of the character case is very light in terms of if the word is the same but requires a change of case from a capital to small letter or vice versa.

Table 10 shows an example of a misspelled word, *recogniseFla*, of which the correct form is *recogniseFlag*. The table demonstrates a list of ten candidate words ordered by weight in ascending order. The word with the least weight is considered the repair.

Table 10: Candidate List of Four-Way Example

Candidate Word	Error Weight
recogniseFlag	70
recogniseflag	140
RecogniseFlag	140
recogniserFlag	150
recognise_Flag	150
Recognize	280
Recognized	280
Recognizes	280
Recursively	410
inconsistent	420

### 7.4 Editex Algorithm

The Editex algorithm has been extended in our work because the Editex group in Table 7 does not involve special characters or numbers. The reason behind that is the algorithm is made for proper

words. Therefore, we have added two new group codes which are special characters (code 9) and numbers (code 10) (see Table 11).

*Table 11: Extend Editex Code Group*

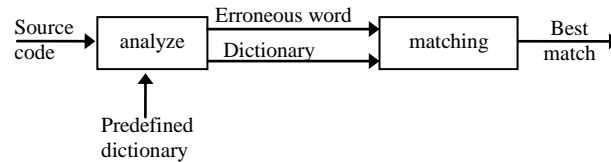
Letters Group	Code
A, E, I, O, U, Y, H, W	0
B, P	1
C, G, J, K, Q	2
D, T	3
L	4
M, N	5
R	6
F, V	7
S, X, Z	8
Special chars	9
0 to 9	10

The Editex algorithm shows a better accuracy level compared to the Four-Way algorithm but it, however, requires a longer time to find a best match and this will be described in the results and discussion section. Consider the example of the misspelled word *recogniseFla* and the correct word *recogniseFlag*. Table 12 shows ten correct words that can be a repair. It is ordered by the score in ascending order. The word with the minimum weight is chosen to be the best match. The example is processed as well by the Four-Way algorithm (see the above section).

*Table 12: Candidate List of Editex Example*

Candidate Word	Score
recogniseFlag	2
recogniseflag	4
RecogniseFlag	4
recogniserFlag	4
recognise_Flag	4
recognize	7
recognized	7
recognizes	7
recursively	10
inconsistent	10

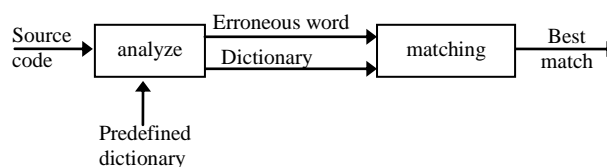
Figure 10 presents the correction spelling combined with a compiler work. This is called a basic algorithm because in the results section we need to distinguish between it and the improved algorithm, which involves features that are described later. The match algorithm is either the Four-Way or Editex Algorithm. The result is a single best match.



*Figure 10: Basic Algorithm*

## 7.5 Edit & Compile

We have added a feature which involves altering the basic algorithm to produce a list of matching words instead of a single word. The list of candidate words is called the candidate list. It has been noticed that despite the failure in finding a repair, the repair is still able to find the wanted repair although it is not ranked the best match. The best match means the candidate word that has a minimum error weight. Therefore, a function is implemented which is called edit which replaces the erroneous word with a word from the candidate list. This is achieved by editing the original source code that contains an erroneous word. Then, a function called 'compile' compiles the source code to check whether the error is still or not. If the error is still, then it repeats all the steps using another word from the candidate list. This approach tries a certain number of words in order to save the time as each time the source code is altered then compiled. Our approach considers two figures of the list, 7 and 10. These figures will be discussed later.



*Figure 11: Edit & Compile Approach*

Figure 11 shows our approach process. It starts by taking words one by one from the candidate list based on the error weight. The word with a minimum error weight is more likely to be a repair. In fact, in our case we altered, as well as the spelling correction process, by removing the filtering process because our work uses a small dictionary.

## 7.6 Context Awareness

In word processing applications, employing a context of a misspelled word can be effective to get a suitable repair. For instance, it can be figured out whether a misspelled word is a verb, adverb, etc. In fact, context awareness is limited in terms of dealing with programming errors. The context may include the lifetime of a variable, type checking, calling for a method and assigning variables. Regarding the lifetime of variables, this can involve building a dictionary by discarding a variable that cannot be used as a repair due to the fact it is out of context. In addition, the type of misspelled error can lead to figuring out which variable it is meant to be. Consider the following example:

```
int var1;  
  
String var2;  
  
var = 4;
```

In the above example, *var* is a misspelled error. Depending on the context, which is assigning a variable to an integer value, it is high likely the repair is *var1*. However, the case is not always true. Suppose both candidate words have the same type, therefore, this cannot be helpful. In our work context awareness is included in analysing the source code. In addition, it is also involved in the ‘compile & edit’ process because if it is correct, then the context would suggest the same way.

## 7.7 Synonyms List

Furthermore, in programming language, it is common to get confused between using variables instead of calling methods or vice versa. That includes forgetting writing parentheses or adding them wrongly. In Java, for instance, it can be confusing whether it should be a method or variable when it deals with classes, for example:

```
ArraylistarVar;  
  
String strVar;  
  
int length = strVar.length();
```

The above example shows two common built classes in Java: *ArrayList* and *String*. In *String* class, to get the length of a string *length()* method should be called while it is called *size* in *ArrayList* which serves the same goal. Furthermore, the length in arrays can be got using a built variable called *length*. This variant of names for almost the same purpose may lead to the committing of errors. Therefore, we think that using a list of synonyms can be useful in finding repairs that spelling correction algorithms cannot find. There are plenty of lists of synonyms for proper English words that can be used in our work.

However, in programming languages there are combined words which are new words consisting of two words or some letters from them, etc. There is no list of similar words for this. For example, suppose a method called *getResult* is defined in a class; a programmer may call it *returnResult*. This

is a programming error that cannot be solved by spelling correction algorithms but can be solved using the synonyms list, but there is no list of synonyms for these combined words. In our work, we have designed this feature to be employed in case edit & compile cannot find a repair.

## 7.8 Learn by Experience

This feature is employed when a repair is found. It stores the repair and the error in a file called experience in order to be used in the future analysis of other errors. This tool is designed to be programmer oriented which means each programmer has his own style of programming as well as making errors; for example, a programmer usually misspells *indexOf* as *indexof*. So, it can be repaired next time from experience based on this error. Similar work is described by [25] which proposes a web tool to store and collect fixes of error messages and pieces of code. It involves human helping that is offered by programming expertises.

In the future work section, we explain how to extend this feature to get compiler programmer oriented. However, this feature cannot be tested because it requires voluntary programmers to store their errors then use them in the future as it is discussed in chapter 9.

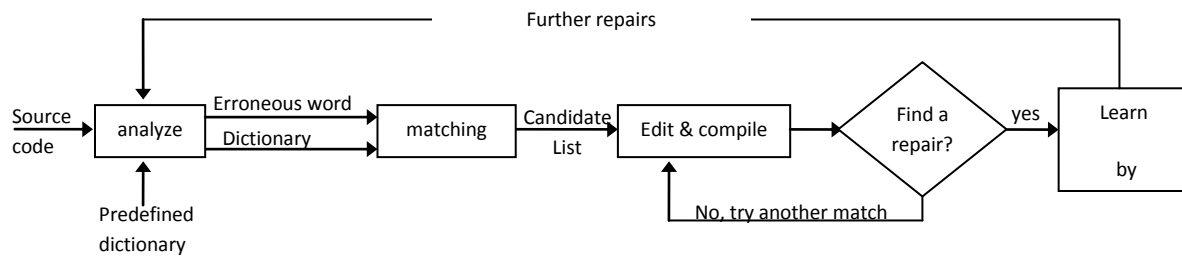
## 7.9 Further Repairing

This function continues repairing further programming errors. Suppose a programmer misspells a variable name more than once in a piece of code, when he compiles his work in JavaC, for example, he gets as many programming errors as the number of times he misspells the variable, apart from other errors. Our tool helps in solving all those errors in two ways: the first is by solving all those errors just by replacing the erroneous word with the repair; the second is if there is another programming error as a result of mistyping problem. Then, we start the process again from the matching algorithm in order to find a list of candidates and so on. However, if the errors are other than misspelling problems or there are no more errors left, the process terminates.

## The Complete Algorithm

Figure 12 demonstrate our approach, including the matching algorithms and other features. It starts by analysing a source code which contains a programming error. It gets the erroneous word and produces the completed dictionary after adding words from the source code to the predefined dictionary. Then, both are applied to a matching process in order to generate a list of candidate words. The matching algorithm can be Four-Way or Editex. Then, the edit & compile process replaces the erroneous word in the source code with a word from the list which is ranked the best match (sorted in an ascended ordered by error weight or score). Then, it compiles to check if the error is solved. If so, then learn by experience adds the erroneous word and the repair to a file to be used later. Then, the further repairs process repeats the analysis again in order to fix other errors.





*Figure 12: The Complete Algorithm*

If the problem is not solved, then the best match is removed and the process uses the second match until either the problem is solved or there are no more candidate words. If there are no candidate words left, it tries synonym words for the error. If the error is solved it goes to learn by experience, but if the error is not solved then it prints out a message that the tool has failed to find a repair.

```

recogniseFla' in line:43 is misspelled,
did you mean 'recogniseFlag'

```

*Figure 13: The Tool Suggesting a Repair*

Finally, figure 13 shows that the message our tool generates when it finds a repair while figure 14 shows that how a typical compiler (in our example is JavaC) prints an error message for the same error. As it is shown, our tool suggests a repair while the other tool does not help programmers although the message seems so informative.

```

Java/file.java:43: cannot find symbol
symbol  : variable recogniseFla
location: class file
recogniseFla = 7;

```

*Figure 14: Typical Compiler Error Message*

## 7.10 Summary

In this section, we discussed the process of our project. Firstly, an explanation of building the dictionary was given. Secondly, a new filtering method was proposed. Then, the implementation of the matching process was explained, which consists of two different algorithms: Four-Way and Editex. Then, several features were discussed. The main feature which enhances the matching algorithms is edit & compile which is based on the fact that matching algorithms fail sometimes to recognise the correct word as the best match. However, they are still able to rank it among the best candidates. Other features were context awareness, learn by experience, the synonyms list and further repairing.

## 8 Results and Discussion

This chapter demonstrates and discusses our results of the experiments on running the implementation of our tool. The test took place as the section 6.2 describes it. We ran the same test set under four different kinds of algorithms. The first was the Four-Way algorithm in a basic format that was proposed by [18]. The second was the Editex algorithm and it was implemented as described in [26]. Then, the edit & compile feature was tested as an extension to both algorithms. Finally, we examined the size of the candidate words. This feature determines whether it can increase the accuracy of both algorithms.

Both algorithms, Four-Way and Editex, were compared to each other in order to find which one is the best according to our test. The comparison was done one with basic algorithms as well as with the edit & compile feature. In addition, the Editex algorithm results were compared with previous studies that examined it. Finally, the dependency of our approach is discussed.

### 8.1 Four-Way Algorithm

The Four-Way algorithm has been implemented as described in [18]. Table 13 shows the result of our experiments of applying 1,214 misspelled words with the algorithm. The test was divided into 9 categories and each contained 100 misspelled words. However, the last type had four-fold the size of other test data at just above 400 words. The *time* column shows the total time required to run each test set. The last row shows the average accuracy of running the algorithm and the average time required for finding a repair for a single misspelled word in a piece of code. The average accuracy of the algorithm was 72% and the required time to find a repair for a misspelled word was .02 seconds.

*Table 13: Four-Way Algorithm Results*

Test Type	Accuracy	Time (seconds)
Missing single letter	83%	1.9
Missing many letters	37%	2.6
Adding single letter	81%	2
Adding many letters	44%	2.1
Substitution single letter	91%	1.9
Substitution many letters	68%	3.3
Special characters single letter	96%	2.4
Special characters many letters	78%	2.3
Top misspelled words	87%	7.2
Average	72.75%	0.02

The table above shows the accuracy of the algorithm is high in finding a repair for substitution single letter and a special character single letter at 91% and 96% respectively. For words that contain special characters, the algorithm can't find a repair because it is a word generated by the

user himself; therefore, the error weight will be consequently very low. In contrast, the algorithm shows a low level of accuracy in finding repairs for words missing many letters at only 37% and for those with many added letters. This is because missing more than one letter, in general, confuses the algorithm with other correct words. For example, *however* is a correct word for *wever* which is closer to *ever* which leads to choosing a wrong word.

Overall, the algorithm is accurate in terms of finding repairs for single letter misspelled words at about 88%, while this percentage sharply drops to be about 56% for many letters. The main reason is, as mentioned earlier, missing more than one letter leads to confusion with other correct words.

The algorithm is executed in a very reasonable time which is estimated at about 0.02 seconds. The reason behind that is the algorithm requires  $\mathcal{O}(n)$ , where  $n$  is the length of an erroneous word. Figure 7 shows the maximum number of comparisons required to classify a letter. It requires only 6 maximum. Then, it is incremented by one or two depending on the error type.

## 8.2 Editex Algorithm

The Editex algorithm was implemented as described in [26], but it required the extension of the Editex code group as was mentioned in section 5.4. The same test set with the Four-Way algorithm was tested under the Editex algorithm. The size of the test was 1,400 misspelled words. The average accuracy was about 84% and on average about 0.08 seconds was the time taken to process a file.

Table 14: Editex Algorithm Results

Test Type	Accuracy	Time (seconds)
Missing single letter	87%	8.9
Missing many letters	49%	8.7
Adding single letter	94%	9.3
Adding many letters	90%	10.2
Substitution single letter	89%	9
Substitution many letters	67%	10.7
Special characters single letters	95%	13.8
Special characters many letters	85%	8
Top misspelled words	97%	31.6
Average	83.67%	0.08

Table 14 demonstrates the accuracy of each test type along with the total time of running the whole test type. It seems from the table that the algorithm is not very accurate in repairing misspelled words that miss more than one letter. Apart from that, the algorithm in general has a high accuracy in finding correct words. It is highly accurate in dealing with the top misspelled words, special

characters words that miss a single character, and adding single letters. According to [17], 80% of misspelled words are just single letter misspelled; thus, the majority of top misspelled words are single letter misspelled.

The algorithm achieves a high level of accuracy for single letter misspelled words at about 92%. For many missing letters, the level is satisfactory at about 72%. For words containing special characters, the algorithm gains a very high accuracy in single misspelled at 95% and almost 85% for many missing letters.

The algorithm was slow compared to the Four-Way algorithm at about 0.08 seconds, which is four times slower than the Four-Way algorithm. The main reason for this is that it requires  $\vartheta(m \times n)$  where  $n$  is the length of the erroneous word and  $m$  is the length of a word from the dictionary. This is because the algorithm runs a nested loop of  $n \times m$  representing the array of two dimensions which was discussed in section 5.4. Overall, the algorithm shows better accuracy than the Four-Way algorithm.

### 8.3 Four-Way Vs Editex

Editex has a better accuracy than the Four-Way. It seems that this is because Editex takes into consideration the similarity of letters by dividing them into different groups called Editex code groups, whereas the Four-Way divides each error depending on the error case. However, the Four-Way algorithm spends about a quarter of the time of Editex processing a misspelled word. Therefore, we can conclude that there is a direct relationship between accuracy and time. That means in order to increase the accuracy of finding a repair for a misspelled word, the time required to achieve this task increases as well.

The Four-Way algorithm is implemented without needing to change the algorithm because the algorithm does not involve any language specification to it. For example, the same algorithm can be employed in English or Arabic without any change. This seems to be a good point in terms of implementation. However, the specification of a language, such as a similar phonetic letter, plays a key role in increasing the efficiency of an algorithm aiming to correct spelling problems. In our work there is the partial use of English words; thus, employing some language features can help to improve the accuracy. The algorithm is poor in finding a repair of misspelled word with more than one incorrect letter due to the fact that it can classify other correct words as a repair.

The Editex algorithm is extended to involve numbers and special characters by altering the Editex code group. Editex maintains some properties of a language. In fact, the Editex code group reflects this because it is based on the phonetic similarity between each letter within the same group. This is not, however, a positive aspect because it may lead to choosing a wrong word based on the similarity between letters. For example, *same* and *name* both are candidate words for a misspelled word *mame*. If the programmer's intention is *same* then the algorithm fails in finding the correct repair because of the phonetic similarity between *n* and *m*. Also, the algorithm deals with misspelled words in a case sensitive manner in the same way as if they are completely different words. It is

obvious that it should be from the same code group. The algorithm is better than the Four-Way algorithm in terms of accuracy, but in terms of time, however, is it worse.

Both algorithms fail if a programmer mistypes a word by making the misspelled word closer to a word other than what he intended to do, because both algorithms use computation to find the best match, which sometimes maybe misleading. Editex shows an acceptable level of accuracy at about 84% but still requires a better level because our project aims to find the best match without letting the user choose among a list of candidate words, as is the way in word processing applications. Therefore, we have developed a new model called edit & compile which combines the algorithms with a compiler in order to gain better efficiency.

## 8.4 Edit & Compile

We have modified both algorithms to produce a list of candidates of a particular number of words. We do not consider the match with the least error weight as the best match until the erroneous word is replaced with it; then, it is compiled correctly. If not, then it uses the second candidate word, and so on. We have used two different numbers each time, as explained later.

*Table 15: Four-Way with Edit & Compile Results*

Test Type	Accuracy	Time (seconds)
Missing single letter	99%	1.9
Missing many letters	53%	1.9
Adding single letter	98%	2.6
Adding many letters	71%	2.2
Substitution single letter	99%	2
Substitution many letters	92%	2
Special characters single letters	100%	2.1
Special characters many letters	91%	2.3
Top misspelled words	97%	7.3
Average	88.89%	0.22

Table 15 presents the accuracy and time of the execution of edit & compile with the Four-Way algorithm. The new approach shows an improvement in the overall accuracy at about 88.89%, compared with 72% without the new feature. The reason is that the algorithm successfully finds the corresponding repair but it does not rank it as the best match. The time, however, is increased about ten-fold because the approach compiles each time it replaces the erroneous word with a word from the candidate list. Compilation plays a key role in increasing the time required to find a repair. On one hand, the algorithm is still poor in dealing with missing many letters at about 53%, despite that fact that it improves by about 16%. On the other hand, the algorithm shows a very high accuracy in finding a repair for misspelled words containing special characters with a single incorrect letter,

missing single letter, added single letter, and top misspelled words at 100%, 99%, 99% and 97% respectively.

*Table 16: Editex with Edit & Compile Results*

Test Type	Accuracy	Time (seconds)
Missing single letter	99%	9.2
Missing many letters	80%	7.2
Adding single letter	99%	12.4
Adding many letters	99%	11
Substitution single letter	100%	8.3
Substitution many letters	95%	8.6
Special characters single letters	100%	12
Special characters many letters	99%	12
Top misspelled words	99%	31.5
Average	96.67%	0.89

Table 16 presents the results of our test approach with the Editex algorithm. As shown, the algorithm has the highest accuracy level amongst all algorithms with an accuracy of above 96%. According to Table 15 and Table 16, the algorithm is improved by about 16% compared to the original algorithm. The algorithm achieves a very accepted level of accuracy in dealing with words that miss more than one letter. It is considered the best by 80%. However, this percentage is considered the least accurate amongst other test types which all were above 90%. The high accuracy comes at a price. The time of processing is relatively high. In fact, the time of processing a word, on average, is 0.9 seconds and that includes the time of compiling an edited source code.

In comparison, between the algorithms with the edit & compile feature, Editex has a higher accuracy as a positive, whereas the Four-Way is faster in terms of execution time. Therefore, the comparison is the same with the edit & compile feature apart from the figures themselves. This proves our claim that the relationship between accuracy and execution time is a positive relationship.

Tables 14 and 15 present the results when the candidate number is 7; i.e. the approach tries seven words from the candidate list as a repair and if one is compiled without errors, then it is the repair. The approach in the worst case, i.e. no repair is found, requires the following:

$$totaltime = d * \vartheta(n) + c * compiletime \dots (1)$$

$$totaltime = d * \vartheta(n * m) + c * compiletime \dots (2)$$

The first equation is for the Four-Way algorithm and the second is for Editex; both with the edit & compile approach.  $d$  means the number of words in the dictionary.  $n$  is the length of the erroneous word.  $c$  is the size of the candidate words list and  $m$  is the length of the compared word with the

erroneous word. *Compile time* is the time of compiling a piece of work which varies depending on the compiler itself.

In order to examine whether increasing the number of candidate words can lead to an increase in the accuracy, we changed the candidate list number to ten. Table 17 shows the results for both algorithms: the Four-Way and Editex along with the edit & compile approach with the candidate word number at ten.

*Table 17: Edit & Compile, candidate words number =10*

Test Type	Four-Way Algorithm		Editex Algorithm	
	Accuracy	Time (seconds)	Accuracy	Time (seconds)
Missing single letter	99%	1.9	100%	9.3
Missing many letters	57%	1.9	85%	7.9
Adding single letter	98%	2.0	99%	9.5
Adding many letters	76%	2.3	99%	10.7
Substitution single letter	100%	2.0	100%	8.7
Substitution many letters	95%	1.9	98%	8.7
Special characters single letters	100%	2.2	100%	12
Special characters many letters	92%	2.2	99%	12.2
Top misspelled words	98%	7.5	99%	37.3
Average	90.56%	0.25	97.67%	0.96

Table 17 shows that the overall accuracy of the Four-Way is 90.56% compared to 88.89% for the same algorithm, but the candidate number is 10. The accuracy is increased as well under Editex which achieves 97.67% of accuracy. However, both show an increase in the time of processing. From this, we can conclude that increasing the size of the candidate list leads to an increase in the overall accuracy. The drawback of it is that it requires more time because it needs to compile as many times as the number of candidate words in the worst case. Therefore, table 18 presents the overall accuracy of each algorithm with different numbers in the candidate list. The table also proves our claim that increasing the number of candidate words increases the accuracy and time as well.



Table 18: Different Candidate List Size Results

Candidate words number	Four-Way Algorithm		Editex Algorithm	
	Accuracy	Time (seconds)	Accuracy	Time (seconds)
7	88.89%	0.22	96.67%	0.89
10	90.56%	0.25	97.67%	0.96
13	90.92%	1.01	97.84%	1.51
16	91.33%	1.21	97.91%	1.73
19	91.74%	1.34	98.33%	1.81
22	91.90%	1.51	98.41%	2.19

Figure 18 shows that the accuracy of the Four-Ways increases relatively sharply when the size of the candidate words is changed from 7 to 10, then it increases gradually. For Editex, almost the same happened, except that after size 10 it remains steady. Therefore, we conclude that the best number of candidate words according to our results is 10, which shows a good level of accuracy with a reasonable amount of time for both algorithms.

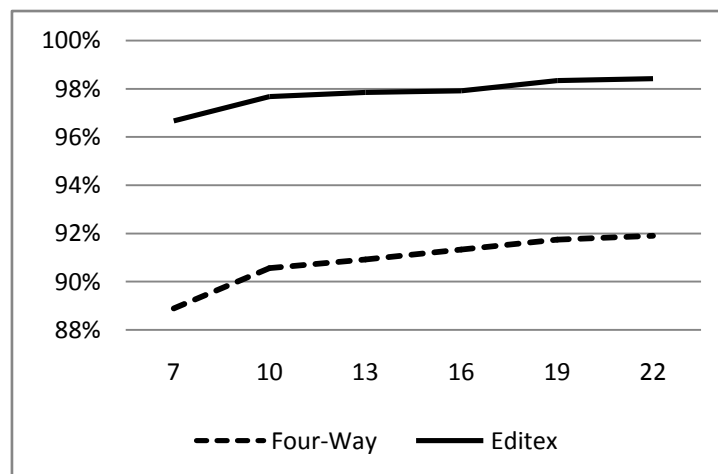


Figure 15: Compile & Edit accuracy levels

In [26], Editex shows an accuracy of 93% for counting correct words (335 out of 360). However, the study claims that it is ranked as the third among other spelling corrections. We can argue that our method enhances the accuracy to about 98% which makes it the best compared to other algorithms such as Phonetex, which achieves an accuracy of 98%. The other argument is that the study does not describe the test set, whether it contains spelled words missing many letters, as this can vary the accuracy accordingly.

## 8.5 The Approach Dependency

Our method seems to be language and compiler independent. For the Four-Way algorithm we have shown it can be applied to any natural language such as Arabic or English, as well as any programming language such as Java or C. The matter is not the same for Editex, as the Editex code group should be changed according to the natural language that it is used for. However, for a programming language we can say it requires adding two new code groups (number and special characters) in order to be able to use the algorithm under any programming language that uses English as it is keywords. Therefore, both algorithms are language independent, which leads to the fact that it is compiler independent as well.

However, in order to be used under another compiler, the tool that gets the error message to get the erroneous word should be altered. For example, we applied our work to the gcc compiler to study whether it is able to work with it. We only changed the method required to deal with the message generated by the compiler. For a programming language, it may involve adding and removing some words from the dictionary which vary depending on the language itself.

Our method has some drawbacks. The first is that it requires some time in order to find the best match. This is more obvious when using the Editex algorithm. Secondly, edit & compile can be misleading in terms of choosing a match because it assumes that the first word in the candidate list which compiles the source code correctly is the best match and terminates. However, this repair may not be consistent with the intention of a programmer. This seems to be true to a certain extent but we can argue that we employ two principles. The first is computation, i.e. the list is created using calculation; thus, choosing the candidate list based on similarities between the erroneous word and the candidate words. Secondly, the repair our approach finds leads to compiling the code correctly.

Furthermore, although our work achieves a high accuracy in finding repairs, it still cannot always find a repair in some cases. The first is when a programmer mistypes a word which can have more than a correct repair (i.e. a word which can compile the code without errors); *xchange*, for instance, is a misspelled word while both *exchange* and *change* are repairs for it. Both can correctly compile the code, then it can be said that the algorithm fails because there is no way to know which one the programmer wanted. Secondly, if a programmer uses a variable with declaration, the algorithm obviously fails. Lastly, if the programmer mistypes a word, by making the correct form is very far more, the algorithm fails to find the repair. However, this depends on the size of the candidate list. Moreover, we have designed the learn by experience feature to add the erroneous word and its repair without consulting the programmer. However, this repair may not be what the programmer intended to use. So, it can be argued that it is better to consult the programmer before adding to the experience file.

## 8.6 Summary

In summary, this chapter presented and discussed our experiments on testing sets of test data. The test set was divided into different categories: adding, missing and substitution letters. Also, some words contained special characters, which included numbers as well. Each was divided into two groups: single letters and many letters. If an erroneous word differed from a correct form by a single word, then it was considered a single letter; otherwise, if it differed by more than one letter, it was placed in the many letters group. In addition, we collected a set of top misspelled words to be a test set as well.

The test sets were applied to the Four-Way algorithm, the Editex algorithm and edit & compile. Edit & compile is an improved approach to both the Four-Way and Editex algorithms. Our experiments showed that the Editex algorithm achieved better accuracy compared with the Four-Way algorithm. However, the Four-Way algorithm could be executed in less time.

We conclude that Editex is the best choice for our work using edit & compile. The size of the candidate number can increase the accuracy but the time as well. We cannot decide which size is the best choice because it depends on the goal of the tool itself. Our goal is to have a very high accuracy in a reasonable time; therefore, the best number for the candidate list is 7. Our work seems to be independent of a compiler and a programming language. However, it still needs to be modified in terms of the way it deals with the error message produced by the used compiler.

## 9 Future Work

This chapter presents our thoughts regarding the future work of the project. As it was described earlier, in order to devise a good argument for our work, the project should be applied to a real study. This section starts by describing the process of testing out the project on a group of programmers and studying their behavior accordingly. Then, it discusses our recommendations for the further studying of programming errors. It finally demonstrates extending of the learn by experience tool to be a complete tool that adapts to the behavior of programmers individually by involving some machine learning techniques such as the Bayesian algorithm.

### 9.1 Test Tools

In section 6.2, we explained our way of testing the data based on accuracy and time in order to compare the algorithms. However, it seems that it is not enough to consider that the algorithms are highly efficient in enhancing the error repair in compilers. We think that the test should be taken further and divided into two stages. The first is applying the Four-Way algorithm to a group of novice programmers who learn Java. The same applies with Editex. In addition, a third group uses any common compiler such as JavaC. The behaviour and the time required to accomplish Java tasks and assignments should be recorded for a semester or two. At the end of the course, there should be three sets of data for each group. Based on these sets, it will be possible to figure out whether our project is effective or not. Furthermore, the students in each group will participate in a questionnaire studying their thoughts about the tools at the end of the term.

The second stage is to have three different groups of novice students and examine each group by a set of programs each with its own program error. As in the first stage, each group has a different algorithm. Then, we can measure the number of correct programs for each student and the total number of correct programs out of the total number of programs for each student in addition to the time. Out of this exam can show whether the tools are effective or not, as well as which is the best.

This test can involve, as well, the test tool learn by experience, because we could not test it because there were no students to apply it on. However, it can be argued that not all errors that each group encounters are considered as the errors we aim to solve, which may influence our results. This seems to be a good argument but we can alleviate it by filtering the errors that they have faced in order to limit the study to our goal.

In fact, we could not achieve this task due to the fact that the time is limited on our work. In addition, the project was carried out during the summer when most of the undergraduate students were away on vacation.

### 9.2 BlueJ

Design an integrated tool, BlueJ, to allow a programmer, after compiling a piece of code, to click on a plug-in in order to give him a solution to a particular programming error. BlueJ, as discussed in 3.2, is a compiler for novice programmers. Designing this tool can help in accomplishing the test task in this chapter.

### 9.3 icompiler

The learn by experience tool which has been described earlier is based on the idea that a programmer usually repeats his errors. Therefore, when our tool finds a repair for a particular programming error, it saves the error and the repair in case a similar error happens again. Then, it is more likely the repair is the same one. In this section we propose future work to extend this feature to involve other programming errors. The idea is to design a compiler which can be different from one programmer to another. The aim is to design a programmer oriented compiler which means a compiler that learns from a programmer's errors. Also, the programmer can alter the error message to become easier for him to figure out the problem. In addition, a repair for an error made by a programmer is different in another programmer's compiler for the same error because of what each of them usually commits in terms of errors. For example, for a programmer that often writes *lengthaslength*, icompiler can figure it out this error easily for this particular programmer but not for others. The name of icompiler is derived from the iGoogle service which allows users to personalize their Google webpages.

In addition, naïve Bayes classifier is an algorithm which classifies test data based on a training data reference [41]. It is based on computing the maximum likelihood [39]. For icompiler, it classifies a programming error based on a training set of files. Each file has its own programming error along with its repair, then it simply computes the likelihood of each file. Finally, it gives a solution which is a repair of a programming error in a file in the training set which has the maximum likelihood.

### 9.4 Recommendation for further studies

Our work reveals the need for more studying regarding the common errors in programming languages. There are plenty of works about common misspelling problems in languages in general. However, in the case of programming language, the resources are limited. It would be helpful to find a list of reasons behind misspelling problems. In addition, more studies about the synonyms of combined words that commonly confuse programmers such as *size* instead of *length* and more preferably *getColor* instead of *returnColor* and so on would be useful. This could be effective in our proposed tool list of synonyms to be implemented.

### 9.5 Summary

In summary, this chapter presents future work that can be effective in backing up our work. It discusses, in the first place, further steps for testing our tools. It focuses mainly on dividing a group of students who learn Java into three groups, then examining their interactions with the tools by recording and comparing the time spent on lab tasks and the ability to find repairs for their errors. Then, it shows our recommendations for more research. Finally, it demonstrates that the learn by experience feature can be extended to be a compiler aspect that can be personalized to each programmer's skills.

## 10 Conclusion

In conclusion, error repair means repairing programming errors in order to continue parsing. However, the error repair is not always consistent with how the programmer wants to do it. Our aim in this project was to study how to enhance the error repair to be able to find solutions that are consistent with intentions of programmers. We limited our goal to Java compilers in trying to repair spelling programming errors using spelling correction algorithms. The study started by looking at compilers and error recover algorithms. These algorithms aim to find repairs when encountering a programming error in order to find other subsequent errors. Then, some related works which try to enhance compiling were discussed.

We studied common programming errors to find out what the common errors are and how a typical Java compiler would respond to them. The study was based on previous studies which show that *cannot resolve symbol* is ranked as the top common error. In fact, a typical Java compiler such as JavaC cannot suggest solutions for such errors. The errors occur through either using undeclared variables or spelling errors. In our work, we combined the Four-Way and Editex spelling correction algorithms with compiling to solve this type of programming error.

The Four-Way algorithm is an algorithm designed to resolve spelling errors in a compiler that is used to understand natural languages. The basic idea is to compare an erroneous word with a list of words, of which each can be a repair. Then, it assigns each word a particular error weight depending on classifying each letter's deference into four categories: adding, missing, substitution and reversing errors. The word with the lowest error weight is deemed to be the best match. The second algorithm is Editex which is an algorithm that consists of combing the Edit distance algorithm with the Editex code group. Edit distance is a way of measuring the similarity between two words. The Editex code group is based on the Soundex algorithm of representing a group of English letters as numbers. Each letter is classified in a group which has a corresponding number. Then each candidate word is assigned a score according to its similarity to the erroneous word. Then, the word with the lowest score is considered the best match. We expanded this group to include special characters and numbers.

In order to test the algorithms, we collected the top misspelled words to examine the accuracy of each algorithm as well as the execution time of it. In addition, we designed the 'edit & compile' method to be applied to each algorithm in order to enhance the accuracy of each algorithm. Although it helped in increasing the accuracy, it increased the execution time of the algorithms. There seems to be a direct relationship between the accuracy of an algorithm and its execution time. We showed as well that our method can be language independent by using it with C language. Also, it is compiler independent because we applied it to two compilers: JavaC and gcc.

*Table 19: Algorithms Results*

Algorithm	Four-Way Algorithm		Editex Algorithm	
	Accuracy	Time (seconds)	Accuracy	Time (seconds)
<b>Baseline algorithm</b>	72.75%	0.02	83.67%	0.08
<b>Edit &amp; compile, 7</b>	88.89%	0.22	96.67%	0.89
<b>Edit &amp; compile, 10</b>	90.56%	0.25	97.67%	0.96

Table 19 summarizes the results of our tests on the algorithms. Baseline algorithm means the algorithm as it is described without any enhancing. Edit & compile was the feature that was added to both algorithms in the second row of the table when the algorithm maintained a candidate list of 7 words and 10 words in the last row. Our experiments showed that the Editex algorithm is better than the Four-Way in all different kinds of test, regardless of whether it is implemented as a baseline algorithm or with edit & compile. In addition, it can be concluded that the Four-Way algorithm has an advantage in terms of execution time because it is almost four times as fast as Editex. Moreover, a significant enhancement in accuracy is achieved when using edit & compile with both algorithms. The choice of candidate word number size may differ according to the purpose of the tool.

## 11 Bibliography

1. CodeProject . <http://www.codeproject.com/?cat=10>. [last access: 13/07/2010]
2. JavaC. <http://openjdk.java.net/groups/compiler/>. [last access: 1/09/2010]
3. Words at Dumbtationary.com. <http://www.dumbtationary.com/word/index.shtml>. [last access: 11/07/2010]
4. The Soundex Indexing System. 2007. <http://www.archives.gov/publications/general-info-leaflets/55.html>. [last access: 12/05/2010]
5. 150 More Often Misspelled Words in English. 2010. <http://www.yourdictionary.com/library/150more.html>. [last access: 11/07/2010]
6. Common misspellings : Oxford Dictionaries Online. 2010. <http://www.oxforddictionaries.com/page/spellingcommonmisp/common-misspellings;jsessionid=082070F581446958B284847E7564A28E>. [last access: 12/07/2010]
7. Java™ 2 SDK, Standard Edition Documentation. 2010. <http://download.oracle.com/javase/1.4.2/docs/index.html>. [last access: 01/07/2010]
8. Oracle Forums. 2010. <http://forums.sun.com/index.jspa>. [last access: 23/07/2010]
9. Ahmadzadeh, M., Elliman, D., and Higgins, C. An analysis of patterns of debugging among novice computer science students. *ACM SIGCSE Bulletin*, (2005), 84-88.
10. Appel, A. and Palsberg, J. *Modern Compiler Implementation in Java*. Cambridge UP, Cambridge, UK, 2002.
11. Bigrigg, M., Bortz, R., Chandra, S., Reed, D., Sheehan, J., and Smith, S. *An Evaluation of the Usefulness of Compiler Error Messages*. Carnegie Mellon University. Available from: [www.ices.cmu.edu/reports/040903.pdf](http://www.ices.cmu.edu/reports/040903.pdf), 2003.
12. Bille, P. A survey on tree edit distance and related problems ☆ *Theoretical Computer Science* 337, 1-3 (2005), 217-239.
13. Branting, L.K. A comparative evaluation of name-matching algorithms. *Proceedings of the 9th international conference on Artificial intelligence and law - ICAIL '03*, (2003), 224.
14. Burke, M.G., Fisher, G.A., and A. practical method for LR and LI syntactic error diagnosis and recovery. *ACM Transactions on Programming Languages and Systems* 9, 2 (1987), 164-197.
15. Cerecke, C. Repairingsyntax errors in L R - based parsers. *Reproduction* 4, (2001).
16. Cha, B., Vaidya, B., and Han, S. Anomaly Intrusion Detection for System Call Using the Soundex Algorithm and Neural Networks. *ISCC*, (2005).



17. Damerau, F. A technique for computer detection and correction of spelling errors. *Communications of the ACM* 7, 3 (1964), 171–176.
18. Elmi, M. A natural language parser with interleaved spelling correction supporting lexical functional grammar and ill-formed input. 1994.  
<http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.22.417&rep=rep1&type=pdf>.
19. Fischer, C. and Mauney, J. A simple, fast, and effective LL (1) error repair algorithm. *Acta Informatica* 120, (1992).
20. Freeman, D. Error correction in CORC: The Cornell computing language. 27-29, 1964, fall joint computer conference, part I, (1964), 15.
21. Garner, S., Haden, P., and Robins, A. My program is correct but it doesn't run: a preliminary investigation of novice programmers' problems. *Proceedings of the 7th Australasian conference on Computing education-Volume 42*, Australian Computer Society, Inc. (2005), 180.
22. Glantz, H. On the recognition of information with a digital computer. *Proceedings of the 1956 11th ACM national meeting*, (1956).
23. Grune, D. and Jacobs, C. *Parsing techniques: a practical guide*. Springer-Verlag New York Inc, 2008.
24. Hamrouni, B. Logic compression of dictionaries for multilingual spelling checkers. *Proceedings of the 15th conference on*, (1994), 292.
25. Hartmann, B., Macdougall, D., Brandt, J., and Klemmer, S.R. What Would Other Programmers Do ? Suggesting Solutions to Error Messages. *Human Factors*, (2010).
26. Hodge, V., Austin, J., and Dd, Y. An Evaluation of Phonetic Spell Checkers. (2001).
27. Jackson, J., Cobb, M., and Carver, C. Identifying top Java errors for novice programmers. *Proceedings 35th Annual Conference Frontiers in Education, 2005. FIE'05*, (2005), T4C–T4C.
28. Jadud, M. A First Look at Novice Compilation Behaviour Using BlueJ. *Computer Science Education* 15, 1 (2005), 25-40.
29. Kim, I. and Choe, K. Error Repair with Validation in LR-based Parsing. 23, 4 (2001), 451-471.
30. Kim, I. and Yi, K. LR error repair using the A\* algorithm. *Acta Informatica* 47, 3 (2010), 179-207.
31. Kukich, K. Techniques for Automatically Correcting Words in Text. *Computing* 24, 4 (1992).
32. Kölling, M. and Rosenberg, J. An object-oriented program development environment for the first programming course. *SIGSE Bulletin* 28, 1 (1996), 83-87.
33. Kölling, M., Quig, B., Patterson, A., and Rosenberg, J. The BlueJ System and its Pedagogy. *Computer Science Education* 13, 4 (2003), 249-268.
34. Kölling, M. BlueJ - Source Downloads. <http://www.bluej.org/download/source-download.html>.

35. McKenzie, B., Yeatman, C., and Vere, L.D. Error repair in shift-reduce parsers. *ACM Transactions on* 17, 4 (1995), 672-689.
36. Morgan, H. Spelling correction in systems programs. *Communications of the ACM* 13, 2 (1970), 90-94.
37. Nienaltowski, M., Pedroni, M., and Meyer, B. Compiler Error Messages : What Can Help Novices ? *Proceedings of the 39th SIGCSE technical symposium on Computer science education*, ACM (2008), 168–172.
38. PARR, T. and QUONG, R. ANTLR: A Predicated-LL(k) Parser Generator. *SOFTWARE—PRACTICE AND EXPERIENCE* 25, June 1994 (1995), 789-810.
39. Peng, F. and Schuurmans, D. Combining naive Bayes and n-gram language models for text classification. *Advances in Information Retrieval*, (2003), 547–547.
40. Pighizzini, G. How Hard Is Computing the Edit Distance? *Information and Computation* 165, 1 (2001), 1-13.
41. Rish, I. An empirical study of the naive Bayes classifier. *IJCAI 2001 Workshop on Empirical Methods in Artificial Intelligence*, (2001), 41–46.
42. Sykes, E. and Franek, F. Presenting JECA: A java error correcting algorithm for the java intelligent tutoring system. *IASTED International Conference on Advances in*, (2004).
43. Thompson, S.M. An Exploratory Study of Novice Programming Experiences and Errors. *Most*, 2006. [www.cs.uvic.ca/~chisel/thesis/SuzanneThompsonThesis.pdf](http://www.cs.uvic.ca/~chisel/thesis/SuzanneThompsonThesis.pdf).
44. Veronis, J. Morphosyntactic correction in natural language interfaces. *Proceedings of the 12th conference on Computational linguistics-Volume 2*, Association for Computational Linguistics (1988), 708–713.
45. Woo, C.W., Evens, M.W., Freedman, R., et al. An intelligent tutoring system that generates a natural language dialogue using dynamic multi-level planning. *Artificial intelligence in medicine* 38, 1 (2006), 25-46.
46. Yeum, Y., Jang, H., Kwon, D., Yoo, S., Kanemune, S., and Lee, W. Dolittle : A Heuristic Approach to Improving Error Messaging Module Based on Error Feedback Strategy. *Journal of Computer Science* 6, 7 (2006), 135-140.
47. Yujian, L. and Bo, L. A normalized levenshtein distance metric. *IEEE TRANSACTIONS ON PATTERN ANALYSIS AND MACHINE INTELLIGENCE* 29, 6 (2007), 1091 - 1095.
48. Zobel, J. and Dart, P. Phonetic string matching: Lessons from information retrieval. *Research and development in information retrieval*, (1996).
49. van Der Spek, P., Plat, N., and Pronk, C. Syntax error repair for a Java-based parser generator. *ACM SIGPLAN Notices* 40, 4 (2005), 47.

## Appendix A: Source Code

```
invokeFourWay(String erroneousWord, int lineNumber)
{
    System.out.println("----- Four way -----");
    candidateList = new ArrayList<String>();
    fourWaysList = new ArrayList<Integer>();
    startTime = System.currentTimeMillis();
    //Four Way algorithm in baseline
    buildFourWaysList(erroneousWord);
    getMinFourWaysList();
    printFourWayCandidateList();
    //using our feature
    fourWay_editAndCompile(erroneousWord, lineNumber);
}
fourWay_editAndCompile(String erroneousWord, int lineNumber)
{
    String match = "";
    String line = "";
    BufferedReader errorMessage;
    Process p;

    //foundFlag = 1 means a repair has been found
    int foundFlag = 0;
    String tmpFile;
    File file = new File(sourceCodeFN);
    String fileName = file.getName();
    tmpFile = "./tmp/" + fileName;

    for( int i=0; i<candidateList.size() ;i++)
    {
        match = candidateList.get(i);
```

```
String code = edit(sourceCodeFN, erroneousWord, match,
lineNumber);
    writeFile(tmpFile, code);
    p = Runtime.getRuntime().exec(cmpMsg + tmpFile);
    errorMessage = new BufferedReader(new
InputStreamReader(p.getErrorStream()));
    line = errorMessage.readLine();
    int msgStatus = checkMsg(erroneousWord, line);
    if(msgStatus != 0) // the error is repaired
    {
        System.out.println(erroneousWord + " in line:" +
lineNumber + " is misspelled, did you mean " + "" + match + "");
        learnByExperiance(erroneousWord, match);
        foundFlag = 1;
        break;
    }
    }
    if (foundFlag == 0)
        System.out.println("can't find a repair for " + erroneousWord
+ " in line:" + lineNumber);
    }
    computeFourWays(String erroneous, String word)
    {
        int errLen = erroneous.length();
        int wordLen = word.length();
        if (erroneous.length() > word.length())
            for(int i=0; i< errLen - wordLen; i++)
                word = word + " ";
        if (word.length() > erroneous.length())
            for(int i=0; i< wordLen - errLen; i++)
                erroneous = erroneous + " ";
        char erroneousAr[] = erroneous.toCharArray();
```

```

char wordAr[] = word.toCharArray();
int weight =0;
int n=0, m=0;
while(n < erroneousAr.length && m < wordAr.length)
{
    if(erroneousAr[n] == wordAr[m])
    {
        //do nothing
        n++;
        m++;
        continue;
    }
    if(n+1 >= erroneousAr.length || m+1 >= wordAr.length)
    {
        n++;
        m++;
        weight = weight + 70;
        continue;
    }
    else if(Character.toString(wordAr[m]).toLowerCase() ==
Character.toString(erroneousAr[n]).toLowerCase())
    {
        //case sensitive error
        n++;
        m++;
        weight = weight + 3;
    }
    else if (erroneousAr[n+1] == wordAr[m])
    {
        //reversed order B, weight = 60
        n+=2;
        m+=2;
        weight = weight + 60;
    }
    else if (erroneousAr[n+1] != wordAr[m] && erroneousAr[n] ==
wordAr[m+1])

```

```

{
    if(erroneousAr[n+1] == wordAr[m+1])
    {
        //character substitution four ways, C, weight =
60, both f and c are the same
        n++;
        m++;
        weight = weight + 70;
    }
    else
    {
        //missing character, D, 80
        n++;
        m+=2;
        weight = weight + 80;
    }
}
else if (erroneousAr[n+1] == wordAr[m] && erroneousAr[n] !=
wordAr[m+1])
{
    if(erroneousAr[n+1] == wordAr[m+1])
    {
        //character substitution four ways,C, weight = 60
        weight = weight + 70;
        n++;
        m++;
    }
    else
    {
        //added character, E, 80
        n+=2;
        m++;
        weight = weight + 80;
    }
}
else

```

```

        {
            //character substitution,F 70
            weight = weight + 70;
            n++;
            m++;
        }
    }
    return weight;
}
invokeEditex(String erroneousWord, int lineNumber)
{
    candidateList = new ArrayList<String>();
    editexList = new ArrayList<Integer>();
    startTime = System.currentTimeMillis();
    //Editex algorithm in baseline
    buildEditexList(erroneousWord);
    getMinEditexList();
    printEditexCandidateList();
    //using our feature
    editex_editAndCompile(erroneousWord, lineNumber);
}
retEditexCode(char c)
{
    int ret=0;
    if (c=='a' || c=='e' || c=='i' || c=='o' || c=='u' ||c=='y')
        ret =0;
    else if (c=='b' || c=='p')
        ret =1;
    else if (c=='c' || c=='k' || c=='q')
        ret =2;
    else if (c=='d' || c=='t')
        ret =3;
    else if (c=='l' || c=='r')
        ret =4;
    else if (c=='m' || c=='n')
        ret =5;

```

```

    else if (c=='g' || c=='j')
        ret =6;
    else if (c=='f' || c=='p' || c=='v')
        ret =7;
    else if (c=='s' || c=='x' || c=='z')
        ret =8;
    else if (c=='s' || c=='c' || c=='z')
        ret =9;
    else if (Character.isDigit(c))
        ret =10;
    else
        ret =11;
    return ret;
}
rFunction(char s, char t)
{
    int ret =0;
    if ( s == t)
        ret =0;
    else if (retEditexCode(s) == retEditexCode(t))
        ret =1;
    else
        ret = 2;
    if ( (s == 'c' && t == 'x') || (s == 'x' && t == 'c'))
        ret =2;
    return ret;
}
computeEditex(String erroneous, String word)
{
    int[][] editex = new int[erroneous.length() + 1][word.length() + 1];
    char s_i, t_j;
    int ret =0;
    for (int i = 0; i <= erroneous.length(); i++)
        editex[i][0] = i;
    for (int j = 0; j <= word.length(); j++)

```

```

        editex[0][j] = j;
    for (int i = 1; i <= erroneous.length(); i++)
    {
        for (int j = 1; j <= word.length(); j++)
        {
            if ( i == 1 && j == 1)
                editex[i][j] = 0;
            else if ( i > 1 && j == 1)
                editex[i][j] = editex[i-1][j] +
rFunction(erroneous.charAt(i - 1), erroneous.charAt(i-2));
            else if ( i == 1 && j > 1)
                editex[i][j] = editex[i][j-1] +
rFunction(word.charAt(j - 1), word.charAt(j-2));

            else if ( i > 1 && j > 1)
                editex[i][j] = min(
                    editex[i - 1][j] +
rFunction(erroneous.charAt(i - 1), erroneous.charAt(i - 2)),
                    editex[i][j - 1] + rFunction(word.charAt(j -
1), word.charAt(j - 2)),
                    editex[i - 1][j - 1] +
rFunction(erroneous.charAt(i - 1), word.charAt(j-1)));
        }
    }
    ret= editex[erroneous.length()][word.length()];
    ret = ret + rFunction(erroneous.charAt(0), word.charAt(0));
    return ret;
}
learnByExperiance(String erroneousWord, String match)
{
    String text = erroneousWord + ":" + match;
    writeFile("files/experiance.txt", text);
}
furtherRepairing(int msgStatus, String tmpFile, int algorithm)
{
    if (msgStatus == 2)//spelling error

```

```

    {
        String erroneousWord;
        erroneousWord = getErroneousWord(tmpFile);
        buildDictionary(erroneousWord);
        sourceCodeFN = tmpFile;
        if(algorithm == 1) //using Four-Way
            invokeFourWay(erroneousWord, lineNumber);
        else //using Editex
            invokeEditex(erroneousWord, lineNumber);
    }
}
editex_editAndCompile(String erroneousWord, int lineNumber)
{
    String match = "";
    String line = "";
    BufferedReader errorMessage;
    Process p;

    int foundFlag =0;
    String tmpFile;
    File file = new File(sourceCodeFN);
    String fileName= file.getName();
    tmpFile = "./tmp/" + fileName;

    for( int i=0; i<candidateList.size();i++)
    {
        match = candidateList.get(i);
        String code = edit(sourceCodeFN, erroneousWord, match,
lineNumber);
        writeFile(tmpFile, code);
        p = Runtime.getRuntime().exec(cmpMsg + tmpFile);
        errorMessage = new BufferedReader(new
InputStreamReader(p.getErrorStream()));
        line = errorMessage.readLine();
        int msgStatus = checkMsg(erroneousWord, line);
    }
}

```

```

        if(msgStatus != 0) // the error is repaired
        {
            System.out.println(erroneousWord + " in line:" +
lineNumber + " is misspelled, did you mean " + "" + match + "");
            learnByExperiance(erroneousWord, match);
            foundFlag = 1;
            break;
        }
    }
    if (foundFlag == 0)
        System.out.println("can't find a repair for " + erroneousWord
+ " in line:" + lineNumber);
}
buildFourWaysList(String erroneous)
{
    for(int i=0; i< dictionaryList.size(); i++)
    {
        int errorWeight = computeFourWays(erroneous,
dictionaryList.get(i));
        fourWaysList.add(errorWeight);
    }
}
buildEditexList(String erroneous)
{
    for(int i=0; i< dictionaryList.size(); i++)
    {
        int editex = computeEditex(erroneous, dictionaryList.get(i));
        editexList.add(editex);
    }
}
equal(char c1, char c2)
{
    if (c1 == c2)
        return 0;
    else

```

```

        return 1;
    }
    checkWord(String word)
    {
        for( int i=0; i<candidateList.size();i++)
            if (word.equals(candidateList.get(i)))
                return true;
        return false;
    }
    buildDisagreementList(String erroneousWord)
    {
        String word = ""; //a word from the dictionary

        for(int i=0; i<dictionaryList.size();i++)
        {
            word = dictionaryList.get(i);
            double d = compareWords(erroneousWord, word);
            if(d <= disagreement)
                disagreementList.add(word);
        }
        copyDisagreementToDictionaryLists();
    }
}

```