南京大学

本科生毕业论文

（申请学士学位）

论文题目 **Shell** 命令错误自动修复的研究

作者姓名 陈劭源

学科、专业方向 计算机科学与技术

研究方向 程序自动修复

指导教师 蒋炎岩 博士

2020 年 **6** 月 **16** 日

学　　号：**161240004**

论文答辩日期：**2020** 年 **6** 月 **10** 日

指 导 教 师：　　（签字）

Automatic Repair of One-Line Shell Command Errors

by

Shaoyuan Chen

Supervised by

Dr. Yanyan Jiang

A dissertation submitted to
Nanjing University
in partial fulfilment of the requirements for the degree of
BACHELOR
in
Computer Science and Technology

Kuang Yaming Honors School
Nanjing University

June 16, 2020

南京大学本科生毕业论文英文摘要首页用纸

THESIS: Automatic Repair of One-Line Shell Command Errors
SPECIALIZATION: Computer Science and Technology
AUTHOR: Shaoyuan Chen
MENTOR: Dr. Yanyan Jiang

Abstract

Command-Line Interface (CLI) is a user interface where the usersinteract with the computer by entering a sequence of commands. Major applications of CLI include file management, source code compilation, and system configuration. Compared with Graphical User Interface (GUI), CLI requires the user to type commands via a keyboard, hence it is more prevalent among advanced computer users, such as system administrators and programmers. Linux Shell is a representative of CLI.

However, users often make many mistakes when entering Shell commands. These mistakes may be attributed to typos or unfamiliarity with the commands. The cryptic error message given by the commands is sometimes confusing; users also have to pay extra time to localize and fix the errors they made, thus lowering the user experience. Therefore, it is worthwhile to design a system that automatically fixes Shell command errors, both theoretically and practically.

The existing Shell command repair methods are mainly rule-based. These systems often contain a large rule library. Each rule is aimed to detect and fix a specific type of errors. However, maintaining such an extensive rule library requires much manual labor, and the rules written by the programmer may contain bugs per se. Though there exists work that automatically synthesizes fixing rules, such methods can not fix unknown bugs.

We propose a syntax-based Shell command fixing tool ShellFix, which is mainly designed to fix spelling errors in shell commands. The system contains a domain-specific language that describes the syntax of shell commands and a set of algorithms to detect and fix errors. In our system, the syntax is first compiled to a regular expression, which is then converted to a nondeterministic finite automaton (NFA). Finally,

we transform the fixing problem into a shortest path problem and use a variation of Dijkstra's algorithm to solve this problem. The experimental result shows that our tool can effectively fix spelling errors in many common shell commands, and it can fix a higher percentage of the benchmarks compared with previous works.

Keywords: Command Line Interface, Domain Specific Language, Shortest Path Problem, Program Repair

南京大学本科生毕业论文中文摘要首页用纸

毕业论文题目：Shell 命令错误自动修复的研究

计算机科学与技术 专业 2016 级学士生姓名：陈劭源

指导教师（姓名、职称）：蒋炎岩 博士

摘　要

命令行界面（CLI）主要通过用户输入一系列命令的方式完成用户与计算机之间的交互。通过 CLI 程序，用户可以方便地完成文件管理、代码编译及系统配置等任务。相比于图形界面（GUI），CLI 需要通过键盘的输入命令，因此 CLI 更适合专业用户（例如计算机管理员、程序员等）使用。Linux Shell 便是一个具有代表性的 CLI 程序。

然而，用户在输入 Shell 命令时，往往会产生很多错误。这些错误可能是因为用户对命令的使用方法不够了解，或是在输入的过程中速度过快产生了错误。一方面，很多 Shell 命令的错误提示信息具有很强的迷惑性，用户在阅读后也很难意识到错在哪里；另一方面，用户在修复错误命令时需要付出额外的时间，降低用户体验。因此，设计一套能够自动修复 Shell 命令错误的算法和系统，无论是在理论上还是在实践上都是很有价值的。

现有的 Shell 命令错误主要是基于规则的修复。修复系统通常包含一个规则库，其中的每条规则用于检测和修复某一类特定的错误。维护这样庞大的规则库需要大量的人力，且程序员编写的规则有可能本身就是错误的。尽管已有自动合成修复规则的工作，但是这类基于规则的修复方法仍然不能修复未知的错误类型。

我们提出了一个基于语法的 Shell 命令错误自动修复工具 ShellFix 。该工具主要针对 Shell 命令中含有的拼写错误。该工具包含了一套用于描述 Shell 命令语法的领域特定语言，以及用于检测和修复错误的算法。该算法的工作流程是：将 Shell 命令的语法翻译成正则表达式，再将正则表达式转换为非确定有限自动机（NFA），最后再将修复问题转换成最短路问题，并使用了 Dijkstra 算法的一个变种求解。实验结果表明，本系统能有效地修复许多常见的 Shell 命令中的拼写错误，且相比已有工作有着更高的修复成功率。

关键词： 命令行界面；领域特定语言；最短路问题；程序修复

Contents

Chapter 1 Introduction

1.1 Background

Command-Line Interfaces (CLIs) handle a series of commands in the form of texts. Although the Graphical User Interface (GUIs) are more popular for end-users nowadays, CLIs still have a certain market share among experienced users, for its relatively fast input speed via keyboards. They often use CLIs to perform various tasks like compiling source code, manipulating files, changing system settings, or even browsing web pages.

The most commonly used CLIs are the shells bundled with operating systems, which allow the users to interact with the operating system and other services interactively. The Bourne shell is the first widely used shell on Unix operating systems. The Korn shell (ksh), Bourne-again shell (bash), and Almquist shell (ash) are examples of other well-known Unix shells. Z shell (zsh), a modern Unix shell, is now the default shell on macOS. The Command Prompt (cmd.exe) is a classic command-line interpreter for Microsoft Windows, though the newer yet more powerful alternative, PowerShell, is bundled with the latest Windows versions. Many software provides application-specific CLIs as well. For example, scripting languages such as Python often provide read-eval-print loop (REPL) interactive CLIs where users can instantaneously see the execution result after entering a single line of code.

Unfortunately, compared with simple Graphical User Interfaces, CLIs are not so user-friendly. Many commands have quite complex usages, providing dozens of option flags and parameters. Users have to specify the correct values of parameters and combinations of flags to achieve the expected behavior. Even a user who uses CLIs every day may not precisely remember the correct usage of a command and sometimes makes mistakes, let alone those new to the CLIs. A more common scenario is, even when a user knows the correct commands, the occasional typos block these commands from being successfully executed. When an error occurs, the user has to localize the error,

then apply a patch to the buggy command. Sometimes, the error message returned by the shell command is confusing; users have to resort to the manuals or online Q&A forums to figure out what is actually wrong. This greatly lowers the user experience. Moreover, some shell command errors are fatal. For example, when a user simply wants to delete a subdirectory in his home directory, he may want to enter

```
rm -rf /home/foo/bar
```

but if the user erroneously types the command as

```
rm -rf / home/foo/bar
```

the shell interpreter might probably remove the entire root directory of the file system, which is catastrophic. Therefore, attentions should be paid to the problem of shell command errors, and it isworth devoting effort to address this issue automatically.

The task to automatically fix a buggy shell command basically lies in the field of automated program repair [25]. The typical workflow of automated repair is as follows: a buggy program is fed to the fixer, then the fixer repeatedly tries to modify the program until the buggy behavior is eliminated. The criteria of the correctness of the program behaviors may be documentations written in natural languages, test suites, formal specifications, etc. Genprog [11, 30, 31], a representative of program repair systems, uses the genetic algorithm to mutate the abstract syntax tree (AST) of the source code, such that it passes a set of input-output test cases. Other program repair techniques are later developed, which have better performance in some specific fields. Nonetheless, program repair remains a challenging research topic due to its natural hardness [14].

1.2 Related Work

Most attempts in program repair mainly focus on fixing codes written in general-purpose programming languages, such as C, C++, Java, and Python. Research on fixing shell commands is less common. We describe two popular shell command fixers: TheFxxx and NoFAQ.

1.2.1 TheFxxx

TheFxxx[1] is a rule-based shell command error fixer, which has gained great popularity in Github. It contains a large rule library (more than 300 rules in v3.30). Each fix rule contains two functions: the `match` function and the `get_new_command`, which are used to detect and fix a common kind of errors, respectively. For example, when creating and entering a new directory, people often forget to run `mkdir`, making the shell interpreter complaining that the target directory does not exist. There is a rule in TheFxxx directed against such kind of errors:

```
def match(command):
    return (
        command.script.startswith('cd ') and any((
            'no such file or directory' in command.output.lower(),
            'cd: can\'t cd to' in command.output.lower(),
            'does not exist' in command.output.lower()
        )))

def get_new_command(command):
    repl = shell.and_('mkdir -p \\1', 'cd \\1')
    return re.sub(r'^cd (.*)', repl, command.script)
```

The first function matches a command starting with `cd` and the error message contains one of the texts `"no such file or directory"`, `"cd: can't cd to"` or `"does not exist"` [2]. When the first function returns true, the second function will be run, fixing the command by adding a `mkdir` command before `cd`. A running example of this rule is shown below.

```
$ cd foo
cd: no such file or directory: foo
$ fxxx
mkdir -p foo && cd foo
```

Fig. 1–1: A running example of TheFxxx.

Since TheFxxx contains a comprehensive set of rules which cover the most com-

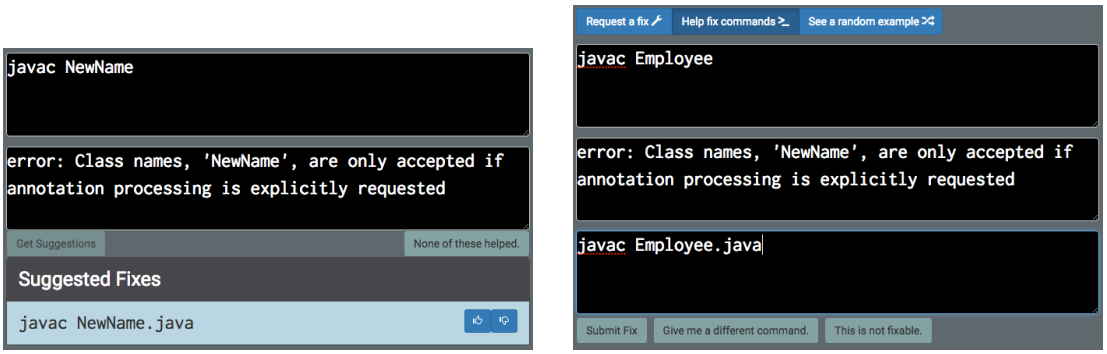
[1] The name of the tool is censored: `https://github.com/nvbn/%74%68%65%66%75%63%6b`.

[2] Since cd is a shell built-in, different shells may give different error message. For compatibility reasons, this rule checks error messages given by different shells.

monly used shell commands, it is quite effective on many shell command errors. Despite its great success in Github community, THEFXXX still requires the developers to manually hardcode the rules. This means THEFXXX lacks universality in some sense: it can not fix the errors that are not included in the rule library until new rules are added. When the developers are not familiar with the precise usage of a shell command or the framework of THEFXXX, they will probably write fix rules with bugs per se.

1.2.2 NoFAQ

Another tool, NOFAQ [9], is built to cope with the limitations of THEFXXX. Unlike THEFXXX, NOFAQ can automatically synthesis fixing rules, given a few input-output examples. Each example consists of three parts: the buggy command, the error message, and the fixed command. The first two are usually submitted by end-users, while the last part is often given by experts. Each fix rule it produces contains two parts: the `match` function and the `fix` function, which is similar to THEFXXX.



(a) Interface to get suggestions. (b) Interface to crowd-source examples of fixes.

Fig. 1–2: User interface of NOFAQ.

Theoretically, NOFAQ uses a technique called Version-Space Algebra (VSA) to synthesis rules from a given set of examples. Originated in the discipline of machine learning, version space is a classic method of concept learning [24]. Basically, the version space learning algorithm maintains a set of *hypotheses* consistent with all training samples. When a new sample is added, the algorithm filters out all hypotheses inconsistent with the newly added sample. We can use *boundary sets* to represent the set of all consistent hypotheses, and the boundary sets can be updated using the candidate-elimination algorithm [18]. This avoids the exponential memory cost when we store

all consistent hypotheses naively.

Lau et al. proposed the concept of VSA [21], which includes a set of operations on version spaces. With the help of VSA, we are able to craft complex version spaces from simple atomic version spaces, while the task of updating a compound version space can be decomposed into updating its component atomic version spaces. They developed the SMARTedit system, which can learn generalized macros for repetitive text transformations. Later, the VSA technique was applied to other Programming by Demonstration (PbD) [20, 27], and Programming by Examples (PbE) [15–17] tasks, which were quite successful.

In NoFAQ, a domain-specific language FIXIT is used for encoding rules that recognize a command and an error message as a certain type of errors, and, if so, map the command and the message to a fixed The syntax of FIXIT is shown in Fig. 1–3.

Fix rule	r	:=	**match** *cmd* **and** *err* $\rightarrow$ *fix*
Input cmd	*cmd*	:=	$[m_1, \cdots, m_l]$
Input err	*err*	:=	$[m_1, \cdots, m_k]$
Output cmd	*fix*	:=	$[f_1, \cdots, f_n]$
Match expr	m	:=	$\textsc{Str}(s) \mid \textsc{Var-Match}(i, s_l, s_r)$
Fix expr	f	:=	$\textsc{Fstr}(s) \mid \textsc{Sub}(p_L, p_R, s_l, s_r, \textsc{Var}(i))$
Pos expr	p	:=	$\textsc{Ip}(k) \mid \textsc{Cp}(c, k, \delta)$

s: string, i, k, δ: integer, c: character

Fig. 1–3: Specification of the rule description language FIXIT.

Firstly, the commands and error messages passed to NoFAQ is tokenized by white spaces. A match function is generated for input error command and error message, by unifying the corresponding tokens in all examples. Two types of match expressions can be used: $\textsc{Str}(s)$ matches the constant string s, while $\textsc{Var-Match}(i, s_l, s_r)$ matches any string with prefix s_l and suffix s_r. Then, the repair function is synthesized, which generates the fixed command token-wise. Again, two types of fix expressions are defined: $\textsc{Fstr}(s)$ produces a constant string s, while $\textsc{Sub}(p_L, p_R, s_l, s_r, \textsc{Var}(i))$ produces the substring of the result of i-th VAR-MATCH from position p_L to p_R, concatenated with prefix s_l and suffix s_r. Each position expression can be a constant value k, or of the form "the position of the k-th occurrence of character c, plus an offset δ", as in $\textsc{Cp}(c, k, \delta)$.

The positions can be either from left to right (represented by positive integers) or from right to left (represented by negative integers).

Example 1–1 A common mistake when running a Java class is to pass the source file name to `java` instead of the class name. Given two examples of such errors, NoFAQ can produce a set of fixing rules, as shown in Fig. 1–4(c). Note that Fig. 1–4(c) effectively contains three rules, represented in a compressed symbolic form. All these three rules are reasonable. When multiple rules are consistent with the whole training set, only the *most specific* rules are preserved. Here we say a rule is most specific if no other consistent rule that recognizes a proper subset of error commands and error messages.

cmd1:	`java run.java`
err1:	`Could not find or load main class run.java`
fix1:	`java run`

(a) First example.

cmd1:	`java meta.java`
err1:	`Could not find or load main class meta.java`
fix1:	`java meta`

(b) Second example.

match $[\textsc{Str}(\texttt{java}), \textsc{Var-Match}(1, \epsilon, \texttt{.java})]$
and $[\textsc{Str}(\texttt{could}), \textsc{Str}(\texttt{not}), \textsc{Str}(\texttt{find}), \textsc{Str}(\texttt{or}), \textsc{Str}(\texttt{load}), \textsc{Str}(\texttt{main})$
$\textsc{Str}(\texttt{class}), \textsc{Var-Match}(2, \epsilon, \texttt{.java})] \rightarrow$

$$[\textsc{Fstr}(\texttt{java}), \left\{ \begin{array}{c} \textsc{Sub}(\textsc{Ip}(0), \textsc{Ip}(-5), \epsilon, \epsilon, \textsc{Ip}(0)) \\ \textsc{Sub}(\textsc{Ip}(0), \textsc{Cp}(\texttt{.}, 1, 0), \epsilon, \epsilon, \textsc{Ip}(0)) \\ \textsc{Sub}(\textsc{Ip}(0), \textsc{Cp}(\texttt{.}, -1, 0), \epsilon, \epsilon, \textsc{Ip}(0)) \end{array} \right\}]$$

(c) The symbolic representation of the synthesized rule.

Fig. 1–4: A fix rule generated by NoFAQ from two examples.

In an experiment conducted by the authors of NoFAQ, the tool is able to synthesize rules for 81 out of 92 benchmark problems in real-time, using just 2 to 5 input-output examples for each benchmark problem. The result, however, is highly questionable due to the highly biased benchmarks. The test problems used in this experiment are obviously selected that they are advantageous to this tool. Actually, in the real-world scenario, a large proportion of the errors are essentially not fixable because the infor-

mation is not enough to produce a meaningful patch. Also, the version space technique is somewhat rigid, making it a high risk of underfitting.

1.3 Overview of ShellFix

Our work primarily focuses on a specific class of errors: typing errors. Empirically, typing errors occupy a large proportion of all errors. Not only amateurs but also experienced users occasionally have typos when entering shell commands.

Unlike previous works which are rule-based, our tool, ShellFix, is a syntax-based shell command error fixer. This is inspired by a simple observation: a correct shell command must be syntactically correct, and typing errors often cause syntax errors. Since syntactically correct shell commands take a very small percentage of all text strings, even a small perturbation may make a correct command violate the syntax. For example, to install `tmux` (a popular terminal multiplexer) on Debian-based systems, one may issue the command

```
sudo apt install tmux
```

however, if one accidentally enters the command as

```
sudo apt isntall tmux
```

the shell may prompt

```
E: Invalid operation isntall
```

since the first argument of `apt` can't be `isntall`, which is a syntax error.

We use a domain-specific language to describe the syntax of a shell command. Our tool complies the syntax into a regular expression, then transforms the regular expression into a nondeterministic finite automaton (NFA). When a buggy command is given to the tool, it uses a specialized shortest path algorithm to find a syntactically correct string with minimum edit distance to the buggy command.

Example 1–2 For the `apt install` commands, we can simply write the syntax in our DSL as

$$\textsc{Command}(\texttt{apt}, [\textsc{Arg}(\textsc{Str}(\texttt{install}), \texttt{'1'}), \textsc{Arg}(\textsc{Identifier}, \texttt{'1'})])$$

the syntax is then compiled into the following regular expression

```
( )*apt( )+install[-A-Za-z0-9]+( )*
```

then transformed to the NFA shown in Fig. 1–5.

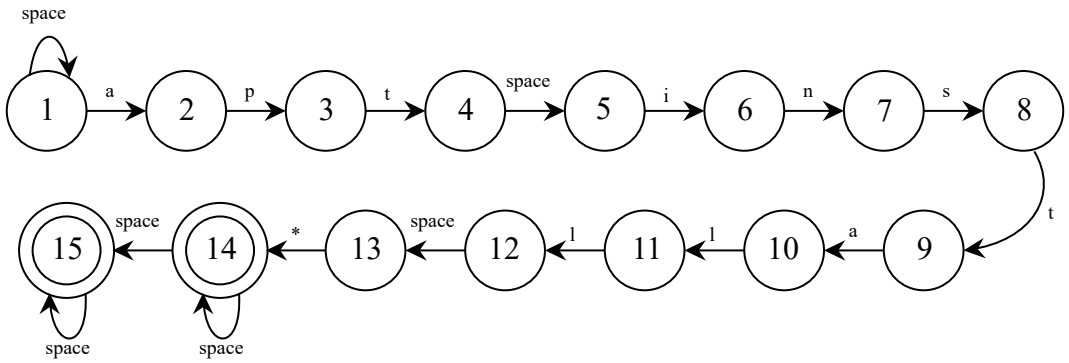


Fig. 1–5: The NFA for `apt install` commands

Given the NFA and the error command `sudo apt isntall tmux`, our tool is able to find the correct command with minimum edit distance `sudo apt install tmux` in real time.

To sum, our work makes the following contributions:

1. a domain-specific language to describe the syntax of shell commands;
2. a static polynomial-time algorithm to detect and fix a buggy command, given the syntax of correct commands;
3. a dataset of shell commands with typing errors, generated from a real-world fault model;
4. a qualitative and quantitative evaluation comparing our tool with previous works.

1.4 Organization of the Thesis

The rest of the thesis is organized as follows. We first give some background knowledge in Chapter 2. Then we describe our system, along with some possible extensions and optimizations, in Chapter 3. This follows the evaluation of this tool in Chapter 4, as well as the benchmark used for evaluation. Finally, Chapter 5 concludes the thesis.

Chapter 2 Preliminaries

In this chapter, we present a few definitions, notations and background knowledge, which are necessary for illustrating our work.

2.1 Formal Languages and Automata Theory

Here are some basic concepts in formal languages and automata theory [1].

Firstly, we give some terms about strings, which are frequently used in language theory.

Definition 2–1 (Alphabet) An *alphabet* Σ is any nonempty finite set of symbols (characters). Typical examples of alphabets include ASCII, Unicode, and the binary alphabet $\{\texttt{0}, \texttt{1}\}$.

In our work, the alphabet in shell commands is the set of all alphanumeric characters, punctuation characters, and white space. This is the default alphabet throughout this thesis.

Definition 2–2 (String) Given an alphabet Σ, any finite sequence of characters in Σ is called a *string* over Σ.

The *length* of a string s is the number of symbols in that string. We denote the length of s by $|s|$. For example, let $s = \texttt{alice}$, then $|s| = 5$. The *empty string*, denoted ϵ, is the string of length 0.

Let s_i denote the $(i+1)$-th character of s. Hence a string s of length n can be written as $s_0 s_1 \cdots s_{n-1}$. We define $s_i = s_{i+n}$ when $i < 0$.

The *concatenation* of strings s and t, denoted st, is formed by appending t to s. For example, let $s = \texttt{foo}$ and $t = \texttt{bar}$, then $st = \texttt{foobar}$. It can be seen that all strings over a fixed alphabet Σ, along with the concatenation operation, form a *monoid*. This is called the *free monoid* on Σ. The identity element of a free monoid is the empty string ϵ.

We say s is a *prefix* of t, if there exists a string w, such that $t = sw$; s is a *suffix* of t, if there exists a string w, such that $t = ws$. For example, `al` is a prefix of `alice`, `ce` is a suffix of `alice`. The empty string ϵ is a prefix or suffix of any string; also, a string is a prefix or suffix of itself.

A *substring* of s is a contiguous subsequence of s. For example, `al`, `lic`, `ce`, `alice`, and the empty string ϵ, are all substrings of `alice`. Let $s[i : j]$ denote the substring $s_i s_{i+1} \cdots s_{j-1}$.

Definition 2–3 (Language) A *language* is any set of strings over a given alphabet Σ.

Particularly, the set of all strings on Σ is a language, denoted by Σ^*.

Note that a language does not require that every string in that language is associated with a real-world meaning.

Definition 2–4 (Regular language) Given an alphabet Σ,

1. $\{\epsilon\}$ is a regular language;
2. For every $a \in \Sigma$, $\{a\}$ is a regular language;
3. If L_1, L_2 are regular languages on Σ, then the *concatenation* of L_1 and L_2, $\{uv | u \in L_1, v \in L_2\}$, is a regular language on Σ, denoted $L_1 L_2$;
   For the sake of convenience, we may define the exponentiation of a language as follows:
   - $L^0 = \{\epsilon\}$;
   - $L^n = L^{n-1} L$ $(n \geq 1)$.
4. If L_1, L_2 are two regular languages, then $L_1 | L_2 = L_1 \cup L_2$ is a regular language, called the *union* of L_1 and L_2.
5. If L is a regular language, then $L^* = \bigcup_{n=0}^{\infty} L^n$ is a regular language, called the *Kleene closure* of L.

Also, we make the following definition:

6. If L is a regular language, then $L^+ = \bigcup_{n=1}^{\infty} L^n = LL^*$ is called the *positive closure* of L. Obviously, L^+ is a regular language as well.

Definition 2–5 (Regular expression) A *regular expression* is the symbolic expression of a regular language. Let $L(r)$ denote the language the regular expression r represents.

The syntax and semantics of regular expressions are defined as follows.

1. ϵ is a regular expression with $L(\epsilon) = \{\epsilon\}$;
2. For every $a \in \Sigma$, a is a regular expression with $L(a) = \{a\}$;
3. If s, t are two regular expressions, then st is a regular language with $L(st) = L(s)L(t)$;
4. If s, t are two regular expressions, then $s|t$ is a regular language with $L(s|t) = L(s)|L(t)$;
5. If s is a regular expression, then $s*$ is a regular language with $L(s*) = L(s)^*$;
6. If s is a regular expression, then $s+$ is a regular expression with $L(s+) = L(s)^+$.

Definition 2–6 (Nondeterministic Finite Automaton) A *nondeterministic finite automaton* (NFA) is formally defined as a 5-tuple $\langle Q, \Sigma, \Delta, S, F\rangle$ where

- Q is a finite set of *states*;
- Σ is a finite set of input symbols, i.e., an alphabet;
- $\Delta : Q \times \Sigma \to 2^Q$ is a transition function. It maps each pair of state and symbol into a set of states;
- $S \subseteq Q$ is the set of *initial* states;
- $F \subseteq Q$ is the set of *accepting* states.

We say a string $s = s_0 s_1 \cdots s_{n-1}$ is *accepted* by an NFA $M(Q, \Sigma, \Delta, S, F)$, if these exists a sequence of states $r_0, r_1, \cdots, r_n$, such that

- $r_0 \in S$;
- $r_{i+1} \in \Delta(r_i, s_i)$, for $0 \leq i < n$;
- $r_n \in F$.

The set of all strings accepted by NFA M is the language *recognized* by M, denoted by $L(M)$.

For each regular language, there exists an NFA that accepts the regular language. There are many algorithms that convert a given regular language to an NFA, of which the most well-known one is the Thompson's construction [1]. In Thompson's construction, the regular language is first converted to an ϵ-NFA, which is a variation of NFA. Then the ϵ-NFA is transformed to NFA. Here we present another algorithm, the Glushkov's construction [13], that converts the regular language directly into NFA. The

steps of Glushkov's construction is as follows:

1. Append the regular expression with a special character `$` which is not in the alphabet.
2. For each alphabet symbol a in the regular expression, assign a unique NFA state, denoted s_a.
3. For each subexpression r, compute the following three functions:
   - nullable(r) is true if and only if r accepts the empty string ϵ;
   - firstpos(r) is the set of states corresponding to the first symbol of some string recognized by r;
   - lastpos(r) is the set of states corresponding to the last symbol of some string recognized by r.

   For each alphabet symbol a, compute followpos(a) which is the set of states that may follow s_a.
4. Let the firstpos(r) be initial states, and the state of `$` be the only accepting state.
5. For each alphabet symbol a and each $s' \in$ followpos(a), add a transition from s_a to s' with character a.
6. The resulting NFA is one that accepts the language represented by the given regular expression.

2.2 Graph Theory

Graphs are a kind of useful structures in discrete mathematics. In this thesis, we mainly use this tool to characterize the states and the transitions between states.

Definition 2–7 (Directed graph) A *directed graph* G is a 3-tuple $\langle V, E, \phi \rangle$, where

- V is the *vertex set* of G;
- E is the *edge set* of G;
- $\phi : E \to V \times V$ is the incidence function that maps every edge to an ordered pair of vertices.

An edge e directed from vertex u to v (i.e., $\phi(e) = (u, v)$) can be written uv.

An *incoming edge* of vertex t is an edge directed to t; an *outgoing edge* of vertex s is an edge directed from s.

Sometimes, we may assign some information to every edge in a graph. The resulting graph is called a *weighted graph*.

We may represent an NFA as a directed graph, where every vertex represents a state, and every edge st weighted with symbol a represents a transition from state s to t when consuming the symbol a, or, $t \in \Delta(s, a)$.

Definition 2–8 (Path) Given a directed graph $G(V, E, \phi)$, a *path* is a sequence of edges $v_0v_1, v_1v_2, \cdots, v_2v_n$.

When every edge is associated with a real number, the *length* of a path $e_1, e_2, \cdots e_n$ is $\sum_{i=1}^{n} w(e_i)$, where $w : E \to \mathbb{R}$ is the weight function of the graph.

The shortest path problem is a fundamental problem in graph theory. It is formally defined as follows:

> *Given a real-weighted directed graph $G(V, E, \phi, w)$ and two vertices $s, t \in V$, find a path from s to t such that the length is minimized.*

When all edge weights are nonnegative, we have Dijkstra's algorithm [6] to compute the shortest path between two vertices:

Algorithm 1 Dijkstra's shortest path algorithm

Input: A weighted directed graph $G(V, E, \phi, w)$ and two vertices $s, t \in V$.
Output: A shortest path $e_1, e_2, \cdots, e_n$ from s to t.

```
1: mark all vertices unvisited
2: d(s) := 0
3: d(u) := +∞ for all u ∈ V \ {s}
4: prev(u) := NIL for all u ∈ V
5: while there exists an unvisited vertex do
6:     let u be any unvisited vertex with minimum d(u)
7:     mark u visited
8:     for each edge e = uv directed from u do
9:         if d(v) > d(u) + w(e) then
10:            d(v) := d(u) + w(e)
11:            prev(v) := e
12:        end if
13:    end for
14: end while
15: return e_1 e_2 ··· e_n (e_i = u_i v_i) where e_n = prev(t), e_i = prev(u_{i+1}) for i < n and
    u_1 = s.
```

In Dijkstra's algorithm, we often use a priority queue to maintain the $d(\cdot)$ of unvisited vertices. Hence, the exact running time of Dijkstra's algorithm depends on the implementation of the priority queue. If we use the Fibonacci heap, the Dijkstra's algorithm runs in $O(|E| + |V| \log |V|)$ [12]. Also, when all edge weights are either 0 or 1, we may use a double-ended queue (deque) to achieve $O(|E| + |V|)$ time complexity.

Chapter 3 The ShellFix System

In this chapter, we illustrate the ShellFix, which contains the Syntax Description Language and the command fixing algorithm. Our system is designed to be static, which means that it does not actually run the buggy shell command. This avoids the potential risk that a buggy command causes catastrophic damage to the entire system.

3.1 The Syntax Description Language

We designed the syntax description language (SDL), which is a domain-specific language to describe the syntax of shell commands. The syntax of SDL is shown in Fig. 3-1.

Commands	*cmds*	:=	$cmd, cmds \mid cmd$
Command	*cmd*	:=	$\textsc{Command}(s, [args])$
Argument list	*args*	:=	$arg, args \mid arg \mid \epsilon$
Argument	*arg*	:=	$\textsc{Arg}(expr, mult)$
Argument expression	*expr*	:=	$\textsc{Str}(s) \mid \textsc{Identifier} \mid \textsc{Integer} \mid \textsc{Path} \mid \textsc{Glob} \mid \cdots$
Multiplicity	*mult*	:=	`'1'`|`'?'`|`'+'`|`'*'`
String	*s*	:=	arbitrary string

Fig. 3-1: Syntax of SDL.

Generally, the SDL describes the syntax of a set of commands. A *command expression* in SDL consists of two parts: the command name and the argument list. The command name, e.g., `ls`, `mkdir` or `git`, is generally the first token of a shell command. The argument list specifies the syntax of all remaining tokens. In the fixing algorithm, we use the specification of arguments in the argument list to match and correct the actual arguments in the input error command.

Each *argument* in the argument list has the form $\textsc{Arg}(expr, mult)$, where *expr* is the *argument expression*, denoting the syntax of this specific argument, and *mult* is the

multiplicity of this argument, which means the number of repetitions of this argument that can be accepted.

The argument expression can either be a constant string s, denoted by STR(s), or one of the following predefined token classes:

- IDENTIFIER: any string that consists of alphanumeric characters and `-`; this can be used to match, for example, a branch name in `git` command, or a package name in `apt` command.
- INTEGER: any string that consists of decimal digits; this can be used to recognize a process identification, or any argument which is required to be a number.
- PATH: any string that represents a pathname in the file system.
- GLOB: any string that represents a glob expression, which is a pathname that consisting of wildcard matchings. There are three types of wildcard matchings supported by glob expressions:
  - `?`: matches any single character;
  - `*`: matches any sequence of characters, possibly empty;
  - `[...]`: matches any single character in the character group.

  Glob expressions are beneficial for specifying multiple pathnames. They are automatically expanded by the shell command interpreter.

Note that PATH and GLOB represent all syntactically correct path names and glob expressions, and the semantics of the actual parameters are never taken into consideration. For example, a valid pathname that does not exist in the current file system will be successfully recognized as an argument specified in PATH.

The multiplicity of an argument can be `1`, `+`, `*`, or `?`. The semantics of each value is shown in Table 3–1. The `1` is often used in any required argument, for example, the target path name of a `move` command; `?` is used in an optional argument, such as a flag that controls some feature of the command; `+` and `*` also mean required and optional argument, respectively, but they allow multiple occurrences of the argument. One may use `+` to mark the source path names of the `mv` command, which supports moving multiple files within a single line of command.

The above definition of SDL is sufficient to describe the syntax of most commonly used shell commands. The ShellFix system also allows the user to extend the SDL by adding user-defined token classes.

Table 3–1: Semantics of the multiplicity values.

Multiplicity	Semantics
1	exactly one token is recognized as the argument
?	the argument is optional: zero or one token is recognized
+	one or more tokens are recognized as the argument
*	zero or more tokens are recognized as the argument

3.2 The Fixing Algorithm

Our algorithm for fixing shell command errors consists of three phases: syntax compilation, NFA generation, and bug fixing.

3.2.1 The Syntax Compilation Phase

In the syntax compilation phase, the syntax specified in SDL is compiled to a regular expression. The compilation algorithm is shown in Algorithm 2.

3.2.2 The NFA Generation Phase

The NFA generation phase converts regular expression of the compiled SDL to a nondeterministic finite automaton. This step is standard, as described in Chapter 2. Here we use Glushkov's construction to convert the regular expression to NFA.

3.2.3 The Bug Fixing Phase

In this phase, our goal is to correct the buggy command to a syntactically correct one. Since there are enormous syntactically correct shell commands, we need to find the possible fixed command, given the buggy command. This is crucial to the effectiveness of the fixing algorithm. Many successful spelling error correction algorithms [3, 7, 19, 26, 28, 29] incorporate well-designed models that find the most possible corrections.

One commonly used class of metrics to measure the similarity between two strings is the *edit distance*, which is defined as the minimum number of atomic operations required to transform one string to the other. There are different versions of the edit

Algorithm 2 SDL compilation algorithm

Input: Command syntax specified in SDL.
Output: Regular expression that represents the syntax.

```
1:  function COMPILE-COMMANDS(cmds)
2:      return Union(map(COMPILE-COMMAND, cmds))
3:  end function
4:
5:  function COMPILE-COMMAND(s, args)
6:      rargs := map(COMPILE-ARGUMENT, args)
7:      return Concat(KleeneClosure(' '), s, rargs, KleeneClosure(' '))
8:  end function
9:
10: function COMPILE-ARGUMENT(arg, mult)
11:     if arg = STR(s) then
12:         re := s
13:     else if arg = IDENTIFIER then
14:         re := KleeneClosure([-a-zA-Z0-9])
15:     else if arg = NUMBER then
16:         re := KleeneClosure([0-9])
17:     else if arg = PATH then
18:         re := KleeneClosure([/-a-zA-Z0-9])
19:     else if arg = GLOB then
20:         re := KleeneClosure(Union([/-a-zA-Z0-9], '[', ']'))
21:     else
22:         invoke the custom handler to get the regular expression
23:     end if
24:     re = Concat(PositiveClosure(' '), re)
25:     if mult = '?' then
26:         re := Union(re, ϵ)
27:     else if mult = '+' then
28:         re := PositiveClosure(re)
29:     else if mult = '*' then
30:         re := KleeneClosure(re)
31:     end if
32:     return re
33: end function
```

Notes:

1. map$(\cdot,\cdot)$ is a higher-order function that applies a given function to each element of a given list, returning a new list as the result, i.e., map$(f, [a_1, a_2, \cdots a_n]) = [f(a_1), f(a_2), \cdots, f(a_n)]$.
2. Union$(\cdot)$, Concat$(\cdot)$, KleeneClosure$(\cdot)$, and PositiveClosure$(\cdot)$ are regular expression constructors. In the above pseudocode, a list is automatically unpacked when passed to Union$(\cdot)$ or Concat$(\cdot)$.

distance depending on the exact definition of the set of atomic operations. Here we use Levenshtein distance as the string distance metric [23], where the atomic operations are insertion, deletion, or substitution of any single character. The Levenshtein distance of two strings s, t, denoted by lev(s, t), can be computed in $O(|s||t|)$ time, which is shown in Algorithm 3.

Algorithm 3 Computing Levenshtein distance of two strings

Input: Two strings s, t.
Output: The levenshtein distance lev(s, t).

```
1:  f[i, j] := +∞ for all 0 ≤ i ≤ |s| and 0 ≤ j ≤ |t|
2:  f[0, 0] := 0
3:  for i = 0 to |s| do
4:      for j = 0 to |t| do
5:          if i < |s| and j < |t| then
6:              if s_i = t_j then
7:                  f[i + 1, j + 1] = min(f[i + 1, j + 1], f[i, j])
8:              else
9:                  f[i + 1, j + 1] = min(f[i + 1, j + 1], f[i, j] + 1)  ▷ change s_i to t_j
10:             end if
11:         end if
12:         if i < |s| then
13:             f[i + 1, j] = min(f[i + 1, j], f[i, j] + 1)              ▷ delete s_i
14:         end if
15:         if j < |t| then
16:             f[i, j + 1] = min(f[i, j + 1], f[i, j] + 1)              ▷ insert t_j after s_i
17:         end if
18:     end for
19: end forreturn f[|s|, |t|]
```

The main idea of Algorithm 3 is dynamic programming. In page 19, $f[i, j]$ denotes the Levenshtein distance between the first i characters of s and the first j characters of t. We may choose the action to perform at the s_i or t_j, and update the distance of the next state.

With the powerful tool of Levenshtein distance, our problem can be rephrased as follows: given a language L specified by an NFA and an error string s, find a string $s' \in L$ such that lev(s, s') is minimized. This can be transformed into a graph-theoretical problem and solved by the shortest path algorithm, which is described in Chapter 2. The procedure is detailed in Algorithm 4.

We may construct the fixed command string character-by-character, by walking

Algorithm 4 Command fixing algorithm

Input: An NFA $M(Q, \Sigma, \Delta, S, F)$ specifying the syntax of commands, and the error command s, given as a string.

Output: A syntactically correct command s' with minimum Levenshtein distance to the error command string.

create an empty graph with vertices $(Q \times \{0, 1, \cdots, |s|\}) \cup \{f, t\}$
for each state $q \in Q$ **do**
    for $i := 0$ to $|s| - 1$ **do**
        for each symbol $a \in \Sigma$ **do**
            for each state $q' \in \Delta(q, a)$ **do**
                if $a = s_i$ **then**
                    add an edge from (q, i) to $(q', i + 1)$ with weight 0 and label a
                else
                    add an edge from (q, i) to $(q', i + 1)$ with weight 1 and label a
                end if
                add an edge from (q, i) to (q', i) with weight 1 and label a
            end for
        end for
        add an edge from (q, i) to $(q, i + 1)$ with weight 1 and label ϵ
    end for
end for
for each state $q \in S$ **do**
    add an edge from f to $(0, q)$ with weight 0 and label ϵ
end for
for each state $q \in F$ **do**
    add an edge from $(|s|, q)$ to f with weight 0 and label ϵ
end for
compute the shortest path P from f to s
return the concatenation of all labels of edges in P

along the edges of the NFA. Similar to the dynamic program computing the Levenshtein distance in Algorithm 4, we pair up the positions in the input string and the states of the NFA. According to what action we can perform at the given position and the state of the NFA, we may add edges from between the pairs of input string positions and NFA states, weighted with the cost of the action. Starting from any initial state of the NFA, our target is to reach any accepting state with the minimum total cost, which corresponds to the shortest length of the path from any initial state to any accepting state. Finally, we obtain the fixed command by concatenating the labels of the edges in the shortest path. The number of edges in the built graph is $O(|M||s|)$, where $|M| = \sum_{q \in Q} \sum_{a \in \Sigma} |\Delta(q, a)|$ is the total number of automaton state transitions. Since all edge weights are either 0

or 1, we may use deque to implement Dijkstra's algorithm, achieving an overall time complexity $O(|M||s|)$, linear to both the complexity of syntax (measured in the size of the NFA) and the length of the input buggy command.

3.3 Extensions and Optimizations

Our system may be extended and improved in the following aspects. These extensions can be either carried out alone or combined with others.

3.3.1 The Multiple Candidates Extension

Our algorithm terminates upon finding the fix with minimum cost. However, it is often the case that the minimum cost fix is not the intended fix. Hence we may let the algorithm continue running until the k best solutions are generated and let the user choose which candidate solution is intended.

To find the k best solutions, we need to find the k shortest paths in the graph built in Algorithm 4. This is known as the k shortest path problem [2, 4], which can be solved in $O(m+n\log n+k)$ time [10], where n is the number of vertices and m is the number of edges.

Example 3-1 Given the buggy command `apt upgate` and syntax specifications

$$\textsc{Command}(\texttt{apt}, \textsc{Arg}(\textsc{Str}(\texttt{update})))$$
$$\textsc{Command}(\texttt{apt}, \textsc{Arg}(\textsc{Str}(\texttt{upgrade})))$$

our algorithm is able to find one possible fix, `sudo apt update`, which has Levenshtein distance 1 to the buggy command. However, with the multiple candidates extension, we can find another possible fix, `sudo apt upgrade`, though this one has Levenshtein distance 2.

3.3.2 The Probabilistic Model Extension

Our algorithm uses the Levenshtein distance to measure the closeness between the fixed command and the error command. However, the edit distance is a coarse measurement

because by using the edit distance, we assume that all typing errors are equally likely to happen. This is not the case when a real man types with a real keyboard. For example, it is more likely that one mistypes letter `t` as `r` than mistypes `t` as `q` , since `t` and `r` are adjacent on a QWERTY keyboard.

We may use a probabilistic model to take this difference into consideration. The probabilistic model is essential to some spelling correction systems [5] and program synthesis algorithms [22], though in our system, the simple edit distance metric works fine due to the sparsity of syntactically correct commands. In our problem, given a buggy command string s, we use the *a posteriori probability* $\Pr[s'|s]$ to measure the likelihood of a fix s'. We may further assume that the probabilities of the actions are independent, decomposing the overall probability into the product of probabilities of the actions. Hence we may replace the costs of the edges in Algorithm 4 with the negative log-likelihoods of the corresponding actions, and find a sequence of actions with minimum total negative log-likelihoods actions. In this case, the graph is no longer binary-weighted, so we have to resort to the original version of Dijkstra's algorithm, resulting in an overhead of a log factor in the total time complexity.

3.3.3 The Environment Oracle Extension

In our original algorithm, we only ensure the syntactic correctness of the fixed shell command. However, this is a relatively weak criterion. Consider that a user that mistypes the name of a file, our system can not detect and fix such type of errors because a misspelled file name is still syntactically correct, even though the file name does not exist in the current file system.

To tackle this problem, we may introduce *environment oracles* to make full use of run-time information. An environment oracle is an oracle that, given a potentially buggy string with some environment-related semantics, e.g., file names, process identifications, or network interface names, returns the most similar semantically correct string under the current run-time environment. For example, a file name oracle may search for files whose names are similar to the given string.

We may embed environment oracles into our system, by extending our syntax description language with environment-dependent token classes. We may first replace these token classes with special characters, then run the first two phases of our system

normally. In the bug fixing phase, we may augment the graph with fixes given by environmental oracles, by adding edges between two automaton states with the fixed strings and corresponding costs returned by environmental oracles. The algorithm is described in Algorithm 5.

Algorithm 5 Command fixing algorithm with embedded environmental oracles

Input: An NFA $M(Q, \Sigma, \Delta, S, F)$ specifying the syntax of commands, the error command s, and a set of environmental oracles $\{\mathcal{M}\}$.

Output: A syntactically correct command s' with minimum Levenshtein distance to the error command string.

```
1: create an empty graph with vertices (Q × {0, 1, ··· , |s|}) ∪ {f, t}
2: for each state q ∈ Q do
3:     for i := 0 to |s| − 1 do
4:         ···                                  ▷ the same as line 4 to line 14 in Algorithm 4
5:         for each special symbol a and corresponding environment oracle M_a do
6:             for j := i to |s| do
7:                 cost, fix := M_a(s[i : j])
8:                 for each q' ∈ Δ(q, a) do
9:                     add an edge from (q, i) to (q', j) with weight cost and label fix
10:                end for
11:            end for
12:        end for
13:    end for
14: end for
15: ···                                          ▷ the same as line 17 to line 24 in Algorithm 4
```

Example 3–2 Consider the `kill` command, which sends a signal (default SIGTERM) to the specified processes. The arguments of `kill` is the identifications of processes to kill, which should exist in the current operating system.

We may add an environmental-dependent token class, PID, which indicates a valid process identification. Hence the syntax of `kill` can be written in the extended SDL as (optional flags are omitted)

$$\text{COMMAND}(\texttt{kill}, \text{ARG}(\text{PID}, \texttt{+})).$$

This is then converted to NFA after phases 1 and 2 as in Fig. 3–2, in which the token class PID is represented by a special character `pid`:

Let the input error command be `kill 1297`, and there is a process with identifica-

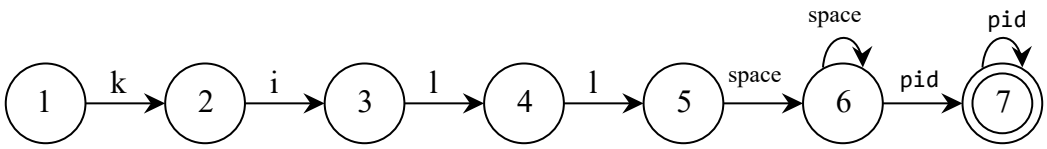


Fig. 3–2: The NFA representing the syntex of `kill`.

tion 1296 in the current running system. Then Algorithm 5 may add edges like

- $(6, 5)$ to $(7, 9)$ with weight 1 and label `1296`;
- $(7, 5)$ to $(7, 9)$ with weight 1 and label `1296`;
- $(6, 5)$ to $(6, 8)$ with weight 1 and label `1296`;
- $(7, 5)$ to $(7, 8)$ with weight 1 and label `1296`;
- $(6, 6)$ to $(7, 9)$ with weight 2 and label `1296`;
- .....

With these edges, our algorithm can successfully find the correct fix `kill 1296`.

Chapter 4 Evaluation

In this chapter, we present a dataset for evaluating our tool. Then we describe the experimental settings and the evaluation results.

4.1 Dataset

Since it is tough to collect enough real-world shell command errors, we crafted a set of shell commands with typing errors. The buggy commands are generated by a fault model, which is derived from real-world typing logs. We believe that the dataset generated in this way is fair and can be used to assess the effectiveness and efficiency of our work.

4.1.1 Fault Model

We learn a probabilistic fault model from 136 million keystrokes, which is collected from an online study involving 168 000 volunteers [8].

Our fault model makes the following assumptions. The user may take one of the following actions at any moment during typing:

1. correctly entering the next character to type;
2. mistyping the next character;
3. omitting the next character;
4. inserting an extra character before the next character of type.

In addition, we assume that the probability of any action that a user takes depends only on the last and the next character he wants to type.

Now we formalize our model. Let s denote the string to type, then the state of typing can be represented by a pair $\langle i, s' \rangle$, where i the the number of characters that the user thinks he've typed, and s' is the string that the user actually types. Each action

maps one state to another:

$$\begin{aligned}\mathsf{Enter}(a) &: \langle i, s' \rangle \mapsto \langle i+1, s'a \rangle \\ \mathsf{Insert}(a) &: \langle i, s' \rangle \mapsto \langle i, s'a \rangle \\ \mathsf{Omit} &: \langle i, s' \rangle \mapsto \langle i+1, s' \rangle\end{aligned}$$

Let Π denote the set of actions, i.e.,

$$\Pi = \{\mathsf{Enter}(a) | a \in \Sigma\} \cup \{\mathsf{Insert}(a) | a \in \Sigma\} \cup \{\mathsf{Omit}\}$$

then we have

$$\Pr\left[\alpha | \langle i, s' \rangle\right] = \Pr\left[\alpha | s_{i-1} s_i\right], \alpha \in \Pi.$$

Since we only consider two characters, there are $O(|\Pi||\Sigma|^2) = O(|\Sigma|^3)$ conditional probabilities in our model. This is a trade-off between the size and the precision of the model.

We learned the fault model in a supervised fashion. The dataset in [8] contains the sentences the volunteers were required to enter and the sentences that the volunteers actually entered. We computed the Levenshtein distance between the two strings by Algorithm 3 and counted the frequency of each action in the action sequence corresponding to the minimum Levenshtein distance. In case that there were multiple such action sequences, we chose one such sequence uniformly at random.

4.1.2 Data Generation

The dataset used for our experiment is generated by the above fault model. Basically, our data generation algorithm is to randomly walk on the state transition graph. The pseudocode is shown in Algorithm 6.

We collected 36 shell command examples from a Linux command line tutorial website[1]. These commands can be classified into 18 programs. For each shell command, we generated 10 buggy commands by Algorithm 6. The overview of our dataset is shown in Table 4–1.

[1] `https://ss64.com/bash/`

Algorithm 6 Data generation algorithm

Input: The fault model $\mathcal{F}$, and the correct command s.
Output: The buggy command s'.

prepend s with a special character ^
append s with a special character $
$i := 1$
$s' :=$ ^
while $i < |s|$ **do**
    $\alpha := \mathcal{F}(s_{i-1}, s_i)$
    if $\alpha = \mathsf{Enter}(a)$ **then**
        $i := i + 1$
        $s' := s'a$
    else if $\alpha = \mathsf{Insert}(a)$ **then**
        $s' := s'a$
    else if $\alpha =$ Omit **then**
        $i := i + 1$
    end if
end while
return s'

4.2 Experiment

We conducted an experiment to show how many shell command errors can ShellFix detect and fix.

4.2.1 Experimental Settings

We implemented the ShellFix in Python. The implementation is composed of the following parts:

- a fixing system that implements the algorithms described in Chapter 3;
- a set of syntax specification written in SDL that encodes the usage of 18 Unix shell commands;
- miscellaneous scripts for data transformation, batch processing, etc.

In the experiment, we run the ShellFix on the dataset described in Section 4.1. All tests were conducted on a server with 48 cores of 2.0GHz Intel Xeon processors and 128GB of RAM, with a timeout of 600 seconds. Also, we compare our tool with TheFxxx version 3.29, run in Python 3.6.9 and zsh 5.4.2.

Table 4–1: Overview of the dataset.

Program	# correct	# buggy	Example
apt-get	4	40	apt-get update
cat	2	20	cat myfile.txt
cd	2	20	cd sybase
chmod	4	40	chmod a-x file
echo	1	10	echo "Hello World"
expr	6	60	expr ss64 : ss6
kill	1	10	kill 1293
ln	4	40	ln file1.txt link1
ls	3	30	ls -al
man	3	30	man intro
mkdir	1	10	mkdir foo
mv	1	10	mv apple orange.doc
rmdir	1	10	rmdir myfolder
tail	1	10	tail -85 file.txt
touch	2	20	touch sample.txt
Σ	36	360	

4.2.2 Results

The experimental results is shown in 4–2. As the results indicate, our tool outperforms TheFxxx in 5 out of all 15 programs, though TheFxxx is better in 3 programs. The experiment suggests that our algorithm is able to fix more shell command typing errors than previous works.

Table 4–2: Experimental results.

	ShellFix		TheFxxx	
Program	**#**	**%**	**#**	**%**
apt-get	**31/40**	**77.50%**	10/40	25.00%
cat	**5/20**	**25.00%**	1/20	5.00%
cd	0/20	0.00%	0/20	0.00%
chmod	**17/40**	**42.50%**	15/40	37.50%
echo	**1/10**	**10.00%**	0/10	0.00%
expr	**13/60**	**21.67%**	10/60	16.67%
kill	3/10	30.00%	3/10	30.00%
ln	0/40	0.00%	0/40	0.00%
ls	0/30	0.00%	0/30	0.00%
man	1/30	3.33%	**5/30**	**16.67%**
mkdir	6/10	60.00%	6/10	60.00%
mv	0/10	0.00%	0/10	0.00%
rmdir	3/10	30.00%	**4/10**	**40.00%**
tail	1/10	10.00%	1/10	10.00%
touch	1/20	5.00%	**4/20**	**20.00%**
Σ	**82/360**	**22.78%**	59/360	16.39%

Chapter 5 Conclusion

This thesis has presented a novel algorithm to fix buggy shell commands with typing errors. The algorithm overcomes the shortcomings of the previous methods by adopting a syntax-based approach. Unlike rule-based methods, a syntax-based method does not require a broad set of rules, and it is still effective to those previously unknown errors. Our method is static: we can fix shell command errors without actually running the buggy shell commands. Evaluation results show that our algorithm is more effective and can successfully fix a larger proportion of benchmarks than previous works.

References

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., USA, 2006.

[2] J. Azevedo, Maria Costa, Joaquim Madeira, and Ernesto de. An algorithm for the ranking of shortest paths. *Eur. J. Operational Research*, 69:97–106, 08 1993.

[3] Eric Brill and Robert C. Moore. An improved error model for noisy channel spelling correction. In *Proceedings of the 38th Annual Meeting on Association for Computational Linguistics*, ACL ’00, page 286–293, USA, 2000. Association for Computational Linguistics.

[4] Eugene Chong, S. Maddila, and S. Morley. On finding single-source single-destination k shortest paths. *Journal of Computing and Information*, Vol.1, No.2, 11 1995.

[5] Kenneth Church and William Gale. Probability scoring for spelling correction. *Statistics and Computing*, 1:93–103, 01 1991.

[6] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

[7] Sebastian Deorowicz and Marcin Ciura. Correcting spelling errors by modelling their causes. *Int. J. Appl. Math. Comput. Sci*, 15:275–285, 01 2005.

[8] Vivek Dhakal, Anna Maria Feit, Per Ola Kristensson, and Antti Oulasvirta. Observations on typing from 136 million keystrokes. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, CHI ’18, New York, NY, USA, 2018. Association for Computing Machinery.

[9] Loris D’Antoni, Rishabh Singh, and Michael Vaughn. Nofaq: Synthesizing command repairs from examples. In *Proceedings of the 2017 11th Joint Meeting*

on Foundations of Software Engineering, ESEC/FSE 2017, page 582 – 592, New York, NY, USA, 2017. Association for Computing Machinery.

[10] David Eppstein. Finding the k shortest paths. *SIAM J. Comput.*, 28(2):652 – 673, February 1999.

[11] Stephanie Forrest, ThanhVu Nguyen, Westley Weimer, and Claire Le Goues. A genetic programming approach to automated software repair. In Franz Rothlauf, editor, *Genetic and Evolutionary Computation Conference, GECCO 2009, Proceedings, Montreal, Québec, Canada, July 8-12, 2009*, pages 947–954. ACM, 2009.

[12] Michael L. Fredman and Robert Endre Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *J. ACM*, 34(3):596 – 615, July 1987.

[13] V M Glushkov. The abstract construction of automata. *Russian Mathematical Surveys*, 16(5):1–53, 1961.

[14] Claire Goues, Stephanie Forrest, and Westley Weimer. Current challenges in automatic software repair. *Software Quality Journal*, 21(3):421 – 443, September 2013.

[15] Sumit Gulwani. Automating string processing in spreadsheets using input-output examples. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL ’ 11, page 317 – 330, New York, NY, USA, 2011. Association for Computing Machinery.

[16] Sumit Gulwani, William R. Harris, and Rishabh Singh. Spreadsheet data manipulation using examples. *Commun. ACM*, 55(8):97 – 105, August 2012.

[17] William R. Harris and Sumit Gulwani. Spreadsheet table transformations from examples. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI ’ 11, page 317 – 328, New York, NY, USA, 2011. Association for Computing Machinery.

[18] Haym Hirsh. Theoretical underpinnings of version spaces. In *Proceedings of the 12th International Joint Conference on Artificial Intelligence - Volume 2*, IJ-

CAI’91, page 665 – 670, San Francisco, CA, USA, 1991. Morgan Kaufmann Publishers Inc.

[19] Mark D. Kernighan, Kenneth W. Church, and William A. Gale. A spelling correction program based on a noisy channel model. In *Proceedings of the 13th Conference on Computational Linguistics - Volume 2*, COLING ’90, page 205 – 210, USA, 1990. Association for Computational Linguistics.

[20] Tessa Lau, Pedro Domingos, and Daniel S. Weld. Learning programs from traces using version space algebra. In *Proceedings of the 2nd International Conference on Knowledge Capture*, K-CAP’03, page 36 – 43, New York, NY, USA, 2003. Association for Computing Machinery.

[21] Tessa A. Lau, Pedro Domingos, and Daniel S. Weld. Version space algebra and its application to programming by demonstration. In *Proceedings of the Seventeenth International Conference on Machine Learning*, ICML’00, page 527 – 534, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc.

[22] Woosuk Lee, Kihong Heo, Rajeev Alur, and Mayur Naik. Accelerating search-based program synthesis using learned probabilistic models. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2018, page 436 – 449, New York, NY, USA, 2018. Association for Computing Machinery.

[23] Vladimir Iosifovich Levenshtein. Binary codes capable of correcting deletions, insertions and reversals. *Soviet Physics Doklady*, 10(8):707–710, Feb 1966.

[24] Tom M. Mitchell. Generalization as search. *Artificial Intelligence*, 18(2):203 – 226, 1982.

[25] Martin Monperrus. Automatic software repair: A bibliography. *ACM Comput. Surv.*, 51(1), January 2018.

[26] Howard L. Morgan. Spelling correction in systems programs. *Commun. ACM*, 13(2):90 – 94, February 1970.

[27] Michael Pardowitz, Bernhard Glaser, and Rüdiger Dillmann. Learning repetitive robot programs from demonstrations using version space algebra. In *Proceedings of the 13th IASTED International Conference on Robotics and Applications*, RA’07, page 394 – 399, USA, 2007. ACTA Press.

[28] James L. Peterson. Computer programs for detecting and correcting spelling errors. *Commun. ACM*, 23(12):676 – 687, December 1980.

[29] Kristina Toutanova and Robert C. Moore. Pronunciation modeling for improved spelling correction. In *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics*, ACL ’02, page 144 – 151, USA, 2002. Association for Computational Linguistics.

[30] Westley Weimer, Stephanie Forrest, Claire Le Goues, and ThanhVu Nguyen. Automatic program repair with evolutionary computation. *Commun. ACM*, 53(5):109 – 116, May 2010.

[31] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. Automatically finding patches using genetic programming. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE’09, page 364 – 374, USA, 2009. IEEE Computer Society.

Acknowledgements

I would like to express my gratitude to all those who helped me during the thesis writing.

Firstly, I sincerely acknowledge the help of my supervisor, Dr. Yanyan Jiang, who gave me professional instructions and useful materials, and, most importantly, the inspirations of the ideas.

Secondly, I would like to thank the members of the SPAR group, including Zelin Zhao, Da Li, Zhao Gang, Yicheng Huang, Xinyi Mao, Yifan Pei, Enmeng Liu, and Hantong Liu, who proposed valuable suggestions during our group meetings.

Thirdly, I am eternally grateful to my teammates, Chunyang Wang and Dongyang Wang, our coach, Prof. Jun Ma, and all other members of the ICPC training team of Nanjing University, who accompanied me during the last four years. There were smiles, there were tears; there were successes, there were failures. We trained in the rooms of laboratories, we fought in the venues of contests. These unforgettable moments and traces long live in my soul.

Also, it's my pleasure to meet the friends in Kuang Yaming Honors School and the teachers of Nanjing University. It is you that taught me a lot, helped me a lot and supported me a lot.

Finally, my gratitude extends to my family members as well, who have always been supporting me in all aspects of my life.