Implementation of Backtracking Algorithm in Bookworm Adventures

Helena Suzane Graciella 13515032
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
13515032@std.stei.itb.ac.id

Abstract—Today, people develop educational games to attract children education. One of world’s well-known educational games is: Bookworm Adventures. In this spelling game, players construct words from a set of 16 letters (not all available). Many approaches and methods may be implemented to play the game, including the backtracking algorithm. This paper will cover that matter. However, future improvements may be done.

Keywords—backtracking; tile score; spelling; suffix

I. INTRODUCTION

One of the most essential thing in the society today is education—education not limited to the walls of schools, but that which includes and is included in every aspect of life. People have come to realization of the importance of education, and thus have been trying to promote education to be enjoyable. Since children are to grow to be future leaders, people have come up with many great ideas to attract children to education, one of them is educational games. One of world’s well-known educational games is: Bookworm Adventures.

In the game Bookworm Adventures, a player is given 16 random letters, each with its own score. Not all from those 16 tiles can be used. Players have to construct words as best as they can to play this game. This paper will cover how to construct a word from a given set of 16 letters, with the backtracking algorithm.

II. THEORIES

A. Backtracking Algorithm [1]

Definition

Backtracking is a DFS-based algorithm to find a solution more efficiently.

General Properties

These followings are general properties of Backtracking algorithm:
1) Solution, expressed as a vector with n-tuple \( X = (x_1, x_2, \ldots, x_n) \), \( x_i \in S_i \) where \( S_i \) possible component of the vector \( X \);
2) Generator Function, expressed as \( T(k) \) to generate \( x_k \) which is a component of the solution vector;
3) Constraint Function, expressed as \( B(x_1, x_2, \ldots, x_k) \) which would convey the value of true when \( (x_1, x_2, \ldots, x_k) \) moves towards goal; if it is true, \( x_{k+1} \) will be generated. The vector will be obsolete otherwise.

Solution Space

Solution space consists of every possible solution. It is organized in a tree structure. Every node represents a state of the problem, whereas an edge is labeled \( x_i \). A path from the root to a leaf means one possible solution. A series of such paths is what we call solution space. This organization is called state space tree.

Principles of Finding Solutions with Backtracking Algorithm

These are the principles of finding solutions with backtracking algorithm:
1) Make paths from root to leaf in depth-first order.
2) Nodes generated are called live nodes.
3) A node being expanded is called the e-node (expand-node).
4) Every time an e-node is expanded, its path becomes longer.
5) If the path being constructed does not go towards solution, the e-node is “killed” to be a dead node.
6) The function used to kill an e-node is called the bounding/constraint function.
7) A dead node will not be expanded anymore.
8) If a path ends with a dead node, backtrack to the node’s parent, then proceed to generate the parent’s child (if there were any).

Figure 1. Backtracking Algorithm Illustration
Source: http://www.w3.org/2011/Talks/01-14-steven-phenotype/

procedure Backtrack(input \( k \): integer)
|
Finding all solutions with backtracking method; recursive.
Input: \( k \), index of a component in the solution vector \( x \)
Output: solution \( x = (x[1], x[2], \ldots, x[n]) \)
Algorithm:

```
for every x[k] unexpanded that ((x[k] ∈ T(k)) and B(x[1], x[2], ..., x[k]) = true) do
    if (x[1], x[2], ..., x[k]) is a solution then
        save(x)
    endif
    Backtrack(k+1)
endfor
```

Figure 2. Common Scheme of Backtracking Algorithm

B. Bookworm Adventures

Bookworm Adventures is a role-playing word game. Players help Lex finish his mission by forming words from the letter jumble. Players will have to survive three mythic storybooks and boost their power with gems, potions, and magical treasures. [2]

There are 3 game modes in the game: Adventure/Adventure Replay, Mini-games, and Arena. In the adventure mode, players follow the story and construct words to attack enemies. In the mini-games, players construct words to achieve high score and rewards that can aid them in their adventure. In the arena mode, unlocked after finishing the adventure mode, players once again have to fight the bosses they have defeated.

In every battle, be it in the adventure/adventure replay or the arena mode, players are given a set of 16 random letters. In every turn, they may choose to build a word with a minimum length of 3 and attack or to scramble the letters.

Each tile has its own point, indicated by a gem in the down-right corner of the tile. There are three categories of tiles based on their gems: Gold, Silver, and Bronze.

<table>
<thead>
<tr>
<th>Tile Weight</th>
<th>Letters Included</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.25 (silver)</td>
<td>B, C, F, H, M, P</td>
</tr>
<tr>
<td>1.5 (gold)</td>
<td>V, W, Y</td>
</tr>
<tr>
<td>1.75 (gold)</td>
<td>J, K</td>
</tr>
<tr>
<td>2 (gold)</td>
<td>X, Z</td>
</tr>
<tr>
<td>2.75 (gold)</td>
<td>QU</td>
</tr>
</tbody>
</table>

Figure 3. Tile Weight in Bookworm Adventures [3]

Tiles may also be powered up by gems or powered down.

<table>
<thead>
<tr>
<th>Gem</th>
<th>Power</th>
</tr>
</thead>
<tbody>
<tr>
<td>Amethyst</td>
<td>Poisons the enemy for two turns</td>
</tr>
<tr>
<td>Emerald</td>
<td>Heals Lex for two hearts</td>
</tr>
<tr>
<td>Sapphire</td>
<td>Freezes enemy for a turn</td>
</tr>
<tr>
<td>Garnet</td>
<td>Weakens the enemy’s attack for two turns</td>
</tr>
<tr>
<td>Ruby</td>
<td>Sets enemy on fire for three turns</td>
</tr>
<tr>
<td>Crystal</td>
<td>Purifies Lex and the grid, shields Lex from damage for one turn</td>
</tr>
<tr>
<td>Diamond</td>
<td>Fully heals Lex, gives one of each potion type</td>
</tr>
</tbody>
</table>

Figure 4. Gems and their powers for tiles [4]

There is also a rainbow tile, and it is not considered a gem tile. This tile acts as a wildcard, automatically giving the letter needed to form any word. [5]

<table>
<thead>
<tr>
<th>Power Down</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fire tile</td>
<td>Destroys tiles below it when not used [6]</td>
</tr>
<tr>
<td>Smashed tile</td>
<td>Causes no damage to the enemy even when forming long words [7]</td>
</tr>
<tr>
<td>Warped tile</td>
<td>Causes no damage and has the ability to change the current letters into hard letters every turn [8]</td>
</tr>
<tr>
<td>Plagued tile</td>
<td>Causes no damage and has an ability to spread to nearby tiles if not used quickly [9]</td>
</tr>
<tr>
<td>Locked tile</td>
<td>Cannot be used [10]</td>
</tr>
<tr>
<td>Alter tile</td>
<td>Switches the current letters in the grid. [11]</td>
</tr>
</tbody>
</table>

Figure 5. Power downs and their descriptions

![Bookworm Adventures battle session](http://www.popcap.com/bookworm-adventures)

Figure 6.

C. English Spelling

Up until now, there is no straight, defined rules of spelling in English language, but a number of small researches have been conducted to see if there were any general principles of word-spelling. Some rules are just generalization, to which there may be an exception. However, some rules are absolute.

**Absolute Rules**

1.) Most nouns are made plural simply by adding an -s. However, nouns ending in the letters ch, sh, s, x, or z with sibilant sounds are made plural by adding -es. Final consonant z has to be doubled before adding -es. [12]

2.) To pluralize a noun ending in a y preceded by a consonant, simply change the y to i and add -es. If the final y is preceded by a vowel, simply add -s. [12]

3.) When adding a suffix that begins with a vowel to a word, double the final consonant in the base word only if the following conditions are met:
   a. The base ends with a single consonant.
   b. The vowel sound preceding the final consonant in the base in the base is represented by a single vowel letter.

Makalah IF2211 Strategi Algoritma, Semester II Tahun 2016/2017
c. The final consonant of the base is in an accented syllable.

Do not double the final consonant before adding the suffix if any one of these three conditions is not met.

Here are some notable exceptions to this rule:

a. Always double the final m of base words that end in the syllable -gram before adding the suffix.

b. Do not double a final l before the suffix -ize, -ism, or -ity.

c. Never double a final x. It is pronounced as two consonants, /ks/, and so violates condition I of this rule. [12]

4.) When adding a suffix that begins with a or o to a word that ends with ce or ge, do not drop the final e from the base word. If the suffix begins with e, i, or y, however, drop the final e from the base. [12]

5.) If words end in ye, ee, or oe, drop the final e before adding a suffix that starts with e. If the suffix does not begin with the letter e, do not drop the final e from the word. [12]

6.) When adding suffix that begins with a vowel to a word that ends in a vowel, change the ie to y, and then add the suffix. [12]

7.) When adding the suffix -ing to a word that ends in ie, change the ie to y, and then add the suffix. [12]

8.) For most words that end in c, insert a k before a suffix that begins with k, c, or y. The final e in these words retains the /k/ sound after the suffix has been added. If the final e does not retain the /k/ sound, do not add a k. [12]

9.) It is sometimes difficult to determine when to use the suffix -ible and when to use the suffix -able at the end of a word. The following general rules will show you the way.

a. If the base word ends in a hard c or a hard g, use the suffix -able.

b. If the base itself is a complete English word, use the suffix -able.

c. If you can add the suffix -ion to the base to make a legitimate English word, then you should use -ible. [12]

Rules with exceptions

1.) Do not drop the final e before adding a suffix that begins with a consonant. However, there are some unusual and unexplainable exceptions to this rule:

- judge + -ment \( \rightarrow \) judgment
- awe + -ful \( \rightarrow \) awful
- nine + -th \( \rightarrow \) ninth
- acknowledge + -ment \( \rightarrow \) acknowledgment
- true + -ly \( \rightarrow \) truly
- argue + -ment \( \rightarrow \) argument
- whole + -ly \( \rightarrow \) wholly [12]

2.) If a word ends in y, change the y to i before adding any of the suffixes -able, -ance, or -ant. Exception: charity + -able \( \rightarrow \) charitable [12]

Rules of thumb

1.) Bigrams (combination of two letters) which seldom or never appear in the dictionary are: [13] [14]

<table>
<thead>
<tr>
<th>Suffixes</th>
</tr>
</thead>
<tbody>
<tr>
<td>-ible</td>
</tr>
<tr>
<td>-able</td>
</tr>
<tr>
<td>-ing</td>
</tr>
<tr>
<td>-esse</td>
</tr>
<tr>
<td>-esse</td>
</tr>
<tr>
<td>-esse</td>
</tr>
</tbody>
</table>

2.) q often comes followed by a u. Although not all q comes with a u, in Bookworm Adventures, there is no q tile, but qu tile.

Therefore, we need to list trigrams which start or end with qu and seldom or never appear in the dictionary.

<table>
<thead>
<tr>
<th>Suffixes</th>
</tr>
</thead>
<tbody>
<tr>
<td>qu</td>
</tr>
<tr>
<td>qun</td>
</tr>
<tr>
<td>qux</td>
</tr>
</tbody>
</table>

Common Suffixes

A suffix is a morpheme added at the end of a word to form a derivative. The table below shows what suffix usually follow a word, grouped by parts of speech. [15] [16]

<table>
<thead>
<tr>
<th>Part of Speech</th>
<th>Suffixes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nouns</td>
<td>-s, -es, -ful, -hood, -fy, -ess, -less, -ism, -ist, -al, -ish, -logy, -ology, -age, -ship, -ese, -i, -ic, -ian, -ly, -ous, -y, -ify</td>
</tr>
<tr>
<td>Verbs</td>
<td>-ing, -s, -es, -able, -ible, -ment, -tion, -ful, -ion, -al, -ance, -ence, -ee, -er, -or, -ive, -d, -ed</td>
</tr>
<tr>
<td>Adjectives</td>
<td>-ly, -er, -est, -ity, -ty, -ness, -ise, -ize, -ate, -en</td>
</tr>
</tbody>
</table>

III. BACKTRACKING ALGORITHM IN PLAYING BOOKWORM

In playing bookworm, the backtracking algorithm will be implemented to find a valid word. To maximize the points attained, check if the word found can be prolonged with a suffix.

Finding a valid word
General properties:
1) Solution: \( x = (x[1], x[2], \ldots, x[n]) \) with \( n \geq 3 \) that makes a valid English word, \( x[i] \in S_i \), \( S_i = \{ \text{every available tiles} \} \)
2) Generator Function: \( T(k) = \text{available tiles which are not yet taken for } x \) so far, ordered by their score; if there were more than one tiles with the same letter(s), only one will be generated
3) Constraint Function: \( B(x) = (x[i-1], x[i]) \) is not included in the bigrams or trigrams mentioned in Figure 5 and Figure 6

These are the steps to find a valid word with backtracking algorithm:
1) Start with level 0 at the tree (Ø).
2) Generate nodes of the next level with the generator function \( T(k) \).
3) Check the live node(s) of the next level with the order given by the generator function. When checking a live node:
a) If \( x \) so far is a solution, stop.
b) If \( B(x) = \text{false} \), kill the node. Go back to its parent.
c) If \( B(x) = \text{true} \) and \( x \) is not a solution, repeat step 2 to 3.
4) If no solution is found, scramble.

Let \( \text{word} \) = a global variable containing a found word and \( \text{path}(n) \) = a function that returns a string which is the characters contained in a treeNode, the pseudocode will be:

```
procedure FindWord(input n: treeNode) {
    // Finding all solutions with backtracking method; recursive.
    Input: n, a treeNode which consists a possible component of the solution vector Output: path from root to solution node
}
Algorithm
    if \( \text{B}(\text{path}(n)) \) do
        generate child(ren) of \( n \) with generator function \( T(n) \)
    for every child of \( n \) unexpanded that do
        if \( \text{path}(n) \) is a solution then
            word \( \leftarrow \) \( \text{path}(n) \)
            \text{else}
            \text{FindWord(child(n))}
        endif
    endfor
}
```

Figure 11. Pseudocode for finding a valid word

Adding a suffix

<table>
<thead>
<tr>
<th>Noun</th>
<th>Verb</th>
<th>Adjective</th>
</tr>
</thead>
<tbody>
<tr>
<td>Suffix</td>
<td>Score</td>
<td>Suffix</td>
</tr>
<tr>
<td>-logy</td>
<td>5.5</td>
<td>-able</td>
</tr>
<tr>
<td>-logy</td>
<td>4.5</td>
<td>-ance</td>
</tr>
<tr>
<td>-ship</td>
<td>4.5</td>
<td>-ence</td>
</tr>
<tr>
<td>-hood</td>
<td>4.25</td>
<td>-ible</td>
</tr>
<tr>
<td>-less</td>
<td>4</td>
<td>-ment</td>
</tr>
<tr>
<td>-ify</td>
<td>3.75</td>
<td>-tion</td>
</tr>
<tr>
<td>-ful</td>
<td>3.25</td>
<td>-ive</td>
</tr>
<tr>
<td>-ish</td>
<td>3.25</td>
<td>-ful</td>
</tr>
<tr>
<td>-ism</td>
<td>3.25</td>
<td>-ing</td>
</tr>
<tr>
<td>-age</td>
<td>3</td>
<td>-ion</td>
</tr>
<tr>
<td>-ese</td>
<td>3</td>
<td>-ed</td>
</tr>
<tr>
<td>-ess</td>
<td>3</td>
<td>-ee</td>
</tr>
<tr>
<td>-ian</td>
<td>3</td>
<td>-er</td>
</tr>
<tr>
<td>-ist</td>
<td>3</td>
<td>-es</td>
</tr>
<tr>
<td>-ous</td>
<td>3</td>
<td>-or</td>
</tr>
<tr>
<td>-ly</td>
<td>2.75</td>
<td>-al</td>
</tr>
<tr>
<td>-ly</td>
<td>2.5</td>
<td>-d</td>
</tr>
<tr>
<td>-ic</td>
<td>2.25</td>
<td>-s</td>
</tr>
<tr>
<td>-al</td>
<td>2</td>
<td>-e</td>
</tr>
<tr>
<td>-es</td>
<td>2</td>
<td>-y</td>
</tr>
<tr>
<td>-i</td>
<td>1</td>
<td>-s</td>
</tr>
</tbody>
</table>

Figure 12 List of suffixes according to the parts of speech they may follow, ordered by their scores according to Bookworm Adventure rule

The process will proceed to this part if and only if a valid word has been found.

Since so far there is no decisive rule of spelling English words, this process will just plug suffixes to the word found according to its part of speech—no attempt to modify the base word to fit the suffixes. The combination of the base word and the suffix is then checked whether they make a valid word or not.

If a valid word constructed from a base word and a suffix is found, recheck if the new word can be prolonged with yet another suffix.

This is how process (2) is undergone:
1. Determine to what part of speech the word belongs.
2. If it is a noun, a verb, or an adjective, check what suffixes may generally follow it, according to its part of speech.
3. Search the suffixes, ordered by score. Plug them to the word and see if it makes a new valid word.
4. If a new valid word is found, repeat steps 1 to 4. Otherwise, finish.

The suffixes will be checked ordered by scores. Suffixes of the same score will be ordered alphabetically (Figure 12).

Let \( \text{nounSuffix} \) = a global variable that is an array of suffixes that follow nouns; \( \text{verbSuffix} \) = a global variable that is an array of suffixes that follow verbs; \( \text{adjSuffix} \) = a global variable that is an array of suffixes that follow adjectives, the pseudocode will be (a suffix is available when the remaining tiles can construct it):

```
procedure Suffix() {
    // Finding a word that may be constructed by the global variable word + available suffix from Figure 10.
    Input:
}
```

Makalah IF2211 Strategi Algoritma, Semester II Tahun 2016/2017
Output: a new string for the global variable word (if there were any)

Algorithm

\[
i: integer
i \leftarrow 0
found: boolean
found \leftarrow false
\]

if word is a noun then

\[
while not found and i < nounSuffix element number do
\]

if nounSuffix[i] is available then

found \leftarrow word + nounSuffix[i]) is a valid word

endif

if not found then

i \leftarrow i + 1

else

word \leftarrow word + nounSuffix[i]

Endif

endwhile

if word is a noun then

\[
while not found and i < nounSuffix element number do
\]

if nounSuffix[i] is available then

found \leftarrow word + nounSuffix[i]) is a valid word

endif

if not found then

i \leftarrow i + 1

else

word \leftarrow word + nounSuffix[i]

Endif

endwhile

if word is a noun then

\[
while not found and i < verbSuffix element number do
\]

if verbSuffix[i] is available then

found \leftarrow word + verbSuffix[i]) is a valid word

endif

if not found then

i \leftarrow i + 1

else

word \leftarrow word + verbSuffix[i]

Endif

endwhile

if word is a noun then

\[
while not found and i < adjSuffix element number do
\]

if adjSuffix[i] is available then

found \leftarrow word + adjSuffix[i]) is a valid word

endif

if not found then

i \leftarrow i + 1

else

word \leftarrow word + adjSuffix[i]

Endif

endwhile

endif

endwhile

endif

if word is a verb then

\[
while not found and i < verbSuffix element number do
\]

if verbSuffix[i] is available then

found \leftarrow word + verbSuffix[i]) is a valid word

endif

if not found then

i \leftarrow i + 1

else

word \leftarrow word + verbSuffix[i]

Endif

endwhile

endif

if word is an adjective then

\[
while not found and i < adjSuffix element number do
\]

if adjSuffix[i] is available then

found \leftarrow word + adjSuffix[i]) is a valid word

endif

if not found then

i \leftarrow i + 1

else

word \leftarrow word + adjSuffix[i]

Endif

endwhile

endif

endwhile

endif

endwhile

endif

endif

endif

Main Program

Let stateSpace = the state space tree; root(t) = a function that returns the root of a tree; Initialize(t) = a procedure that empties tree t and; Scramble() is a function to scramble the letters, the main program will be:

Global Variables

word: string
nounSuffix, verbSuffix, adjSuffix: array of string
stateSpace: tree

Algorithm

initialize(stateSpace)
FindWord(root(stateSpace))
while word is empty do
Scramble()
Initialize(t)
FindWord(root(stateSpace))
Endwhile
Suffix()

Example

Let at a given time, \(S_i = (O, X, B, S, E, E)\), this is how the algorithm works.

The picture above illustrates how to find a valid word from \(S_i\) with the procedure from Figure 11. At the end of this process, \(word = box\). After finding a valid word, suffixes are plugged and checked with the procedure from Figure 13. The word \(box\) may be a noun or a verb. It will first enter the noun if-statement.

![Figure 15. The problem’s corresponding state space tree](image)

After finding a new word, the procedure Suffix() will call itself again. The word \(boxes\) also may be a noun or a verb.

![Figure 16. Illustration of procedure Suffix() before calling Suffix() after finding a new valid word](image)
and dying, the algorithm avoids using a tile powered by an amethyst gem; or when Lex is dying, seek to use a tile powered by the emerald gem.

4. Handle cases where a word is of more than one part of speech, for example: *light* may be a noun or a verb. Further researches may be conducted to see which part of speech is more likely and easily to be prolonged by a suffix. This is to optimize the process of prolonging words with suffix in procedure $\text{Suffix}()$.

V. CONCLUSION

*Bookworm Adventures* is a role-playing word game. Players help Lex finish his mission by forming words from the letter jumble. Players will have to survive three mythic storybooks and boost their power with gems, potions, and magical treasures.

This spelling game uses the English language—a language with no clear or decisive rules of spelling. A number researches have been conducted, and the world have come with a discovery of fact that a certain bigrams or trigrams seldom or never appear in the English language.

With this heuristic, we can develop a backtracking algorithm in solving the random letters, coming up with a decent word, then attacking the enemies. However, certain improvements may and should be done.

ACKNOWLEDGMENT

Praise be unto God Who is bountiful in grace and providence, without Whom the writer is of no capability. The writer is also grateful for the never-ending support from parents, siblings, family, and friends. Last but not least, the writer also thanks Mrs. Masayu Leylia Khodra, Mr. Rinaldi Munir, and Mrs. Nur Ulfa Maulidevi for their guidance in teaching students Algorithm Strategies.

REFERENCES


IV. FUTURE POSSIBLE IMPROVEMENTS

The writer realizes that this paper is imperfect and has yet to be improved. Here are the suggestions of improvements that may be developed in the future:

1. More researches in English language may be conducted to make a better, more decisive, and more reliable general abstraction of its spelling rules.

2. When adding a suffix, try to apply the rules included in chapter 2 of this paper. For example, changing the _-ic in die_ to y before adding _-ing_. This will avoid constructing _dying_ from _die_ and the program will rather come up with _dying_.

3. More intellect approach may be implemented where the algorithm also takes care of the tiles power when making a decision. For example, when the enemy is already poisoned and dying, the algorithm avoids using a tile powered by an amethyst gem; or when Lex is dying, seek to use a tile powered by the emerald gem.

4. Handle cases where a word is of more than one part of speech, for example: *light* may be a noun or a verb. Further researches may be conducted to see which part of speech is more likely and easily to be prolonged by a suffix. This is to optimize the process of prolonging words with suffix in procedure $\text{Suffix}()$. 

Figure 17. Illustration of procedure $\text{Suffix()}$ after finding a new word, checking noun $\text{Suffix}$

<table>
<thead>
<tr>
<th>Base</th>
<th>Suffix</th>
<th>Suffix Available?</th>
<th>Valid?</th>
</tr>
</thead>
<tbody>
<tr>
<td>-ology</td>
<td>No</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-logy</td>
<td>No</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-ship</td>
<td>No</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-hood</td>
<td>No</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-less</td>
<td>No</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-ify</td>
<td>No</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-ful</td>
<td>No</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-ish</td>
<td>No</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-ism</td>
<td>No</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-age</td>
<td>No</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-ese</td>
<td>No</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-ian</td>
<td>No</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-ist</td>
<td>No</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-ous</td>
<td>No</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-ly</td>
<td>No</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-y</td>
<td>No</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-ic</td>
<td>No</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-al</td>
<td>No</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-es</td>
<td>No</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-y</td>
<td>No</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-i</td>
<td>No</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-s</td>
<td>No</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 18. Illustration of procedure $\text{Suffix()}$ after finding a new word, checking verb $\text{Suffix}$


PERNYATAAN
Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 19 Mei 2017

Helena Suzane Graciella 13515032