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).

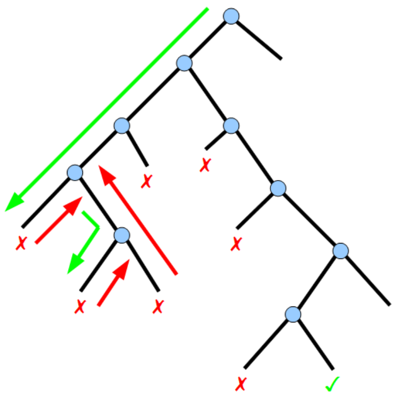

Figure 1. Backtracking Algorithm Illustration
Source: http://www.w3.org/2011/Talks/01-14-steven-phenotype/

```
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], ... ,
   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 A*rena*. 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.

Tile Weight	**Letters Included**
1 (bronze)	A, D, E, G, I, L, N, O, R, S, T, U
1.25 (silver)	B, C, F, H, M, P
1.5 (silver)	V, W, Y
1.75 (gold)	J, K
2 (gold)	X, Z
2.75 (gold)	QU

Figure 3. Tile Weight in *Bookworm Adventures* [3]

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

Gem	**Power**
Amethyst	Poisons the enemy for two turns
Emerald	Heals Lex for two hearts
Sapphire	Freezes enemy for a turn
Garnet	Weakens the enemy's attack for two turns
Ruby	Sets enemy on fire for three turns
Crystal	Purifies Lex and the grid, shields Lex from damage for one turn
Diamond	Fully heals Lex, gives one of each potion type

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]

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

Figure 5. Power downs and their descriptions

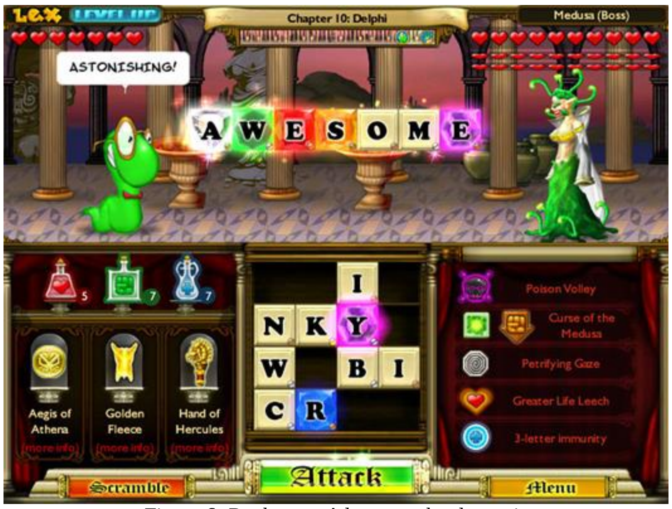

Figure 6. *Bookworm Adventures* battle session
Source: http://www.popcap.com/bookworm-adventures

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.

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 1 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 with **ue**, drop the final **e** before adding 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 **i**, **e**, or **y**. The final **c** in these words retains the **/k/** sound after the suffix has been added. If the final **c** 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 → judgment
- awe + -ful → awful
- nine + -th → ninth
- acknowledge + -ment → acknowledgment
- true + -ly → truly
- argue + -ment → argument
- whole + -ly → 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 → charitable [12]

Rules of thumb

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

bk	cx	hv	jk	jx	mq	qg	qs	vb	vq	xd	zb
bq	dx	hx	jl	jy	mx	qh	qt	vc	vt	xg	zc
bx	fk	hz	jm	jz	mz	qj	qv	vd	vw	xj	zf
bz	fq	iy	jn	kq	pq	qk	qw	vf	vx	xk	zg
cb	fv	jb	jp	kv	pv	ql	qx	vg	vz	xv	zh
cf	fx	jc	jq	kx	px	qm	qy	vh	wj	xr	zj
cg	fz	jd	jr	kz	qb	qn	qz	vj	wq	xz	zn
cj	gq	jf	js	lq	qc	qo	sx	vk	wv	yq	zq
cp	gv	jg	jt	lx	qd	qp	sz	vm	wx	yv	zr
cv	gx	jh	jv	mg	qe	qq	tq	vn	wz	yy	zs
cw	hk	jj	jw	mj	qf	qr	tx	vp	xb	yz	zx

Figure 7. Bigrams which seldom or never appear in the dictionary

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.

quc	quj	qup	quw	gqu	wqu
qud	quk	ququ	qux	jqu	yqu
quf	qul	qus	quz	kqu	
qug	qum	qut	bqu	pqu	
quh	qun	quv	fqu	vqu	

Figure 8. Trigrams which start or end with **qu** and seldom or never appear in the dictionary

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]

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

Figure 9. Suffixes that usually follow certain parts of speech

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

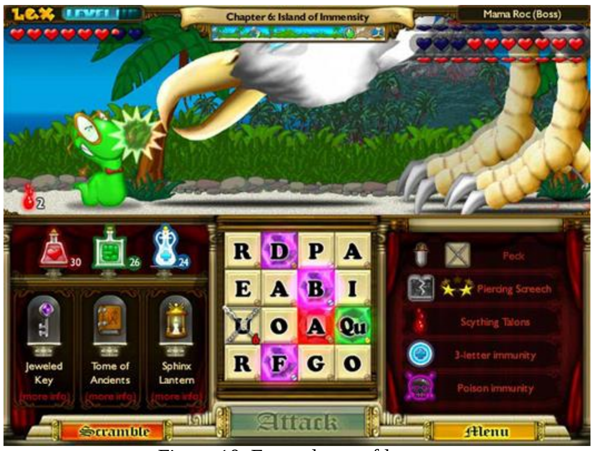

Figure 10. Example set of letters
http://www.popcap.com/bookworm-adventures

General properties:

1) Solution: $x = (x[1], x[2], \ldots, x[n])$ with $n >= 3$ that makes a valid English word, $x[i] \in S_i$, S_i = {every available tiles}
2) Generator Function: T(k) = 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[k-1], x[k]) 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*) = false, kill the node. Go back to its parent.
   c) If B(*x*) = true and *x* is not a solution, repeat step 2 to 3.
4) If no solution is found, scramble.

Let `word` = a global variable containing a found word and `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 B(path(n)) do
    generate child(ren) of n with generator
          function T(n)
    for every child of n unexpanded that do
          if (path(n)) is a solution then
          word ← path(n)
          else
          FindWord(child(n))
          endif
    endfor
  endif
```

Figure 11. Pseudocode for finding a valid word

Adding a suffix

Noun		Verb		Adjective	
Suffix	Score	Suffix	Score	Suffix	Score
-ology	5.5	-able	4.25	-ize	4
-logy	4.5	-ance	4.25	-ness	4
-ship	4.5	-ence	4.25	-ity	3.5
-hood	4.25	-ible	4.25	-ate	3
-less	4	-ment	4.25	-est	3
-ify	3.75	-tion	4	-ise	3
-ful	3.25	-ive	3.5	-ly	2.5
-ish	3.25	-ful	3.25	-ty	2.5
-ism	3.25	-ing	3	-en	2
-age	3	-ion	3	-er	2
-ese	3	-ed	2		
-ess	3	-ee	2		
-ian	3	-er	2		
-ist	3	-es	2		
-ous	3	-or	2		
-fy	2.75	-al	1		
-ly	2.5	-d	1		
-ic	2.25	-s	1		
-al	2				
-es	2				
-y	1.5				
-i	1				
-s	1				

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 make 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 `nounSuffix` = a global variable that is an array of suffixes that follow nouns; `verbSuffix` = a global variable that is an array of suffixes that follow verbs; `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:
    Output: a new string for the global variable
    word (if there were any)
}
Algorithm
    i: integer
    i ← 0
    found: boolean
    found ← false
    if word is a noun then
      while not found and i < nounSuffix element
        number do
        if nounSuffix[i] is available then
          found ← word + nounSuffix[i]) is a valid
              word
        endif
        if not found then
          i ← i + 1
        else
          word ← word + nounSuffix[i]
          Suffix()
        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 ← word + verbSuffix[i]) is a valid
              word
        endif
        if not found then
          i ← i + 1
        else
          word ← word + verbSuffix[i]
          Suffix()
        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 ← word + adjSuffix[i]) is a valid word
        endif
        if not found then
          i ← i + 1
        else
          word ← word + adjSuffix[i]
          Suffix()
        endif
      endwhile
    endif
```

Figure 13. Finding a suffix that can be appended to a word found previously

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()
```

Figure 14. Main program

Example

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

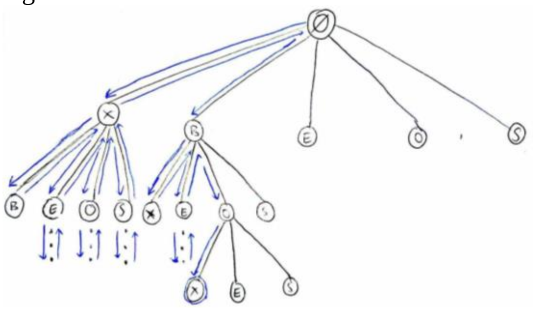

Figure 15. The problem's corresponding state space tree

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.

Base	Suffix	Suffix Available?	Valid?
box +	-ology	No	
	-logy	No	
	-ship	No	
	-hood	No	
	-less	No	
	-ify	No	
	-ful	No	
	-ish	No	
	-ism	No	
	-age	No	
	-ese	Yes	No
	-ess	No	
	-ian	No	
	-ist	No	
	-ous	No	
	-fy	No	
	-ly	No	
	-ic	No	
	-al	No	
	-es	Yes	Yes

Figure 16. Illustration of procedure Suffix() before calling Suffix() after finding a new valid word

After finding a new word, the procedure `Suffix()` will call itself again.

The word **boxes** also may be a noun or a verb.

Base	Suffix	Suffix Available?	Valid?
boxes +	-ology	No	
	-logy	No	
	-ship	No	
	-hood	No	
	-less	No	
	-ify	No	
	-ful	No	
	-ish	No	
	-ism	No	
	-age	No	
	-ese	No	
	-ess	No	
	-ian	No	
	-ist	No	
	-ous	No	
	-fy	No	
	-ly	No	
	-ic	No	
	-al	No	
	-es	No	
	-y	No	
	-i	No	
	-s	No	

Figure 17. Illustration of procedure Suffix() after finding a new word, checking nounSuffix

Base	Suffix	Suffix Available?	Valid?
boxes +	-able	No	
	-ance	No	
	-ence	No	
	-ible	No	
	-ment	No	
	-tion	No	
	-ive	No	
	-ful	No	
	-ing	No	
	-ion	No	
	-ed	No	
	-ee	No	
	-er	No	
	-es	No	
	-or	No	
	-al	No	
	-d	No	
	-s	No	

Figure 18. Illustration of procedure Suffix() after finding a new word, checking verbSuffix

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 **-ie** in **die** to **y** before adding **-ing**. This will avoid constructing **dieing** 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 `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 to Mrs. Masayu Leylia Khodra, Mr. Rinaldi Munir, and Mrs. Nur Ulfa Maulidevi for their guidance in teaching students Algorithm Strategies.

References

[1] R. Munir, "Algoritma Runut-Balik," in *Diktat Kuliah IF2211 Strategi Algoritma*. Bandung, Indonesia: Penerbit Informatika ITB, 2009, ch. 7, sec. 1-4, pp. 125-129.

[2] Available: http://www.popcap.com/bookworm-adventures-2

[3] Various Authors. *Tile*. Available: http://bookwormadvs.wikia.com/wiki/Tile

[4] Various Authors. *Gems*. Available: http://bookwormadvs.wikia.com/wiki/Gems

[5] Various Authors. *Rainbow Tiles*. Available: http://bookwormadventures.wikia.com/wiki/Rainbow_Tile

[6] Various Authors. *Fire Tiles*. Available: http://bookwormadventures.wikia.com/wiki/Fire_Tiles

[7] Various Authors. *Tile* Smash. Available: http://bookwormadventures.wikia.com/wiki/Tile_Smash

[8] Various Authors. *Warp Tile*. Available: http://bookwormadventures.wikia.com/wiki/Warp_Tile

[9] Various Authors. *Plagued Tile*. Available: http://bookwormadventures.wikia.com/wiki/Plague_Tile

[10] Various Authors. *Tile Lock*. Available: http://bookwormadventures.wikia.com/wiki/Tile_Lock

[11] Various Authors. *Alter Tiles*. Available: http://bookwormadventures.wikia.com/wiki/Alter_Tiles

[12] M. Strumpf, A. Douglas, “Spelling,” in *The Complete Grammar*. New Delhi, India: Goodwill Publishing House, 2008, ch. 14, sec. 1, pp. 385-391.

[13] Prash. *Impossible Bigrams in the English Language*. Available: https://linguistics.stackexchange.com/questions/4082/impossible-bigrams-in-the-english-language

[14] P. Remaker, H. Brown, J. Pepersack, J. Lin. *What are all of the two-letter combinations that never occur in an English dictionary?* Available: https://www.quora.com/What-are-all-of-the-two-letter-combinations-that-never-occur-in-an-English-dictionary

[15] M. Strumpf, A. Douglas, “Vocabulary,” in *The Complete Grammar*. New Delhi, India: Goodwill Publishing House, 2008, ch. 15, sec. 2, pp. 460-461.

[16] R. Carter, M. McCarthy, G. Mark, and A. O’Keeffe, “Word Formation,” in *English Grammar Today*. Cambridge, United Kingdom: Cambridge University Press, 2011, ch 50, pp. 182-185.

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

[illegible]

Helena Suzane Graciella 13515032