String Matching in Scribblenauts Unlimited

Jordan Fernando / 13510069
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
fernandojordan.92@gmail.com

***Abstract*—Scribblenauts Unlimited is an emergent action puzzle video game where you play a role as a boy who can create or modify any object by writing words to the magic notebook that the boy carries. This paper will discuss about the string matching implementation in the game Scribblenauts Unlimited.**

***Index Terms*—About four key words or phrases in alphabetical order, separated by commas.**

1. INTRODUCTION

1.1. Scribblenauts Unlimited

Scribblenauts Unlimited is an emergent action puzzle video game developed by 5th Cell and published by Warner Bros to be played on Nintendo 3DS, Wii U, and Microsoft Windows. It is the fourth title in Scribblenauts game series.

Figure 1. Scribblenauts Unlimited Game Cover for Nintendo 3DS.

In the game you play a role as a boy, Maxwell, who can create or modify any object by writing words to the magic notebook that the boy carries. The goal of this game is to find starites that appeared after doing something good for other people. The starites will be used on Maxwell's sister, Lily in order to cure her from being a stone.

The game also supports you to create your own object with your own keyword for creation and to share your created object. The object can be made using the parts that is provided in the game.

Figure 2. Scribblenauts Unlimited Object Creation Screen.

1.2. String matching

String matching is a problem where you have a pattern and a text where you want to find the place where the pattern is found on the text.

There are many algorithms that can solve the string matching problem such as brute force, Rabin-Karp string search algorithm, Finite-state automaton based search, Knuth-Morris-Pratt algorithm, Boyer-Moore string search algorithm, and Bitap algorithm. But in this paper will only discuss about Brute Force string search algorithm, Rabin-Karp string search algorithm, Knuth-Morris-Pratt algorithm, and Boyer-Moore string search algorithm.

String matching has been used in many applications such as search engines, chat bot, DNS server, and auto-completion.

2. THEORIES

2.1. String matching algorithms

2.1.1. Brute Force string search algorithm

In Brute Force string search algorithm, first we align the start point of the pattern and the text. Then we match the pattern into the text one letter by one letter then if we find a letter that doesn't match we shift the pattern position 1 to the right and the start comparing them again. If we didn't find a letter that doesn't match until the last letter then we have already found our matching strings in the text. If we always find a letter that doesn't match until the end of text then the pattern is not within the text.

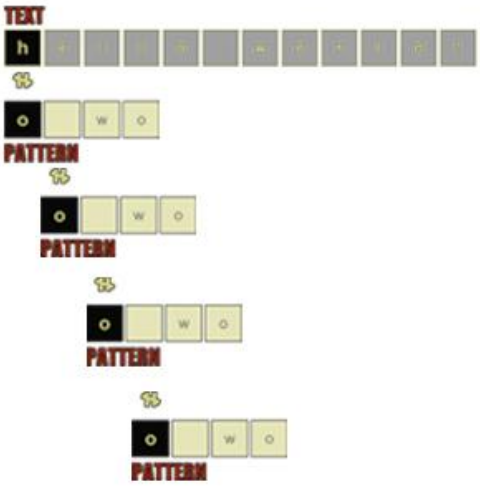

Figure 3. Brute Force string search visualization

The pseudo code of the Brute Force string search algorithm is like this in C++:

```
int BruteForce (string text, string pattern)
{
 int m = pattern.length();
 int n = text.length();
 for (int i=0;i<n-m+1;++i)
 {
  int j = 0;
  bool check = true;
  while ((check)&&(j<pattern.length()))
  {
   if (text[j]!=pattern[i+j])
    check = false;
   j++;
  }
  if (check)
   // pattern found
   return i;
 }
 return -1;
}
```

Brute Force algorithm has the complexity of:

- Preprocessing time: 0
- Matching time: $\Theta((n - m + 1)\, m)$

So, if we have a text with 1000 characters and a pattern with 5 characters, we would have to compare 4980 times.

2.1.2. Rabin-Karp string search algorithm

Rabin-Karp algorithm is a string searching algorithm created by Michael O. Rabin and Richard M. Karp in 1987 that takes the advantage of hashing to find any one of a set of pattern strings in text.

In the algorithm, we have m as the length of the pattern and you have hs as the hash result of the pattern, and hsub as hash result of substring of text by the length m. We move the pattern just like the Brute Force algorithm, but we didn't compare the characters first. Instead, we compare the hash result first. If the hash result is the same, then we compare the characters one by one just like in the Brute Force string search algorithm. In the practice, Rabin-Karp is frequently used for detecting plagiarism.

The pseudo code of the Rabin-Karp string search algorithm is like this in C++:

```
int RabinKarp (string text, string pattern)
{
 int m = pattern.length();
 int n = text.length();
 int hsub = hash(pattern,0,m);
 int hs = hash(text,0,m);
 for (int i=0;i<n-m+1;++i)
 {
  if (hs==hsub)
  {
   int j = 0;
   bool check = false;
   while ((check)&&(j<pattern.length()))
   {
    if (text[j]!=pattern[i+j])
     check = false;
    j++;
   }
   if (check)
    return i;
  }
 }
 return -1;
}
```

Rabin-Karp algorithm has the complexity of:

- Preprocessing time: $\Theta(m)$
- Matching time: average $\Theta(n+m)$, worst $\Theta((n-m+1)\, m)$

So, if we have a text with 1000 characters and a pattern with 5 characters, we would have compare 1005 times in average cases and 4980 times in worst cases.

2.1.3. Knuth-Morris-Pratt algorithm

Knuth-Morris-Pratt algorithm also known as KMP algorithm is a string searching algorithm where we search a pattern within text and when mismatch occurs, the pattern has the information to determine where the next match could begin. The algorithm was conceived in 1974 by Donald Knuth and Vaughan Pratt, and independently by James H. Morris. The three published it jointly in 1977.

In the algorithm, we have an array with the same size of the pattern that contains the number of suffix that matches the prefix. We precompute the table first and then when the character is not matching at some point we could shift the string pattern based on the position – the value of the table at that index.

Here is some visualization:

We have the text "abacabaccabacdabaabb" and the pattern "abacab". First we need to precompute the table. And then use the KMP string search algorithm to find the pattern at the text.

Table 1. Precomputation table for KMP algorithm

k	1	2	3	4	5	6
b(k)	0	0	1	0	1	2

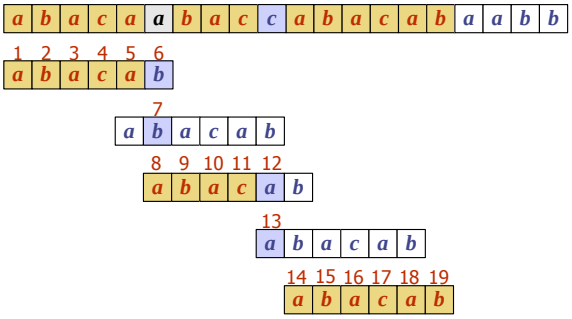

Figure 4. Visualization of KMP algorithm

The pseudo code of the KMP string search algorithm is like this in C++:

```
int KMP (string text, string pattern)
{
 int m = pattern.length();
 int n = text.length();

 // creating precompute table
 int table[m];
 table[0] = 0;
 int cnd = 0;
 int i = 1;
 while (i < m)
 {
  if (pattern[i]==pattern[cnd])
  {
   table[i] = cnd + 1;
   cnd = cnd + 1;
   i++;
  }
  else if (cnd > 0)
   cnd = table [cnd -1];
  else if (cnd ==0)
  {
   table[i] = 0;
   ++i;
  }
 }
 i = 0;
 int j = 0;
 bool check = true;
 while ((check)&&(i<n))
 {
  if (pattern[j]==text[i])
  {
   i++;
   j++;
  }
  if (j==m)
   return i;
  else if ((i<n)&&(pattern[j]!=text[i]))
  {
   if (j!=0)
    j = table[j-1];
   else
    i++;
```

```
   }
  }
 }
```

KMP algorithm has the complexity of:

- Preprocessing time: Θ(m)
- Matching time: Θ(n)

So, if we have a text with 1000 characters and a pattern with 5 characters, we would have compare 1000 times in average.

2.1.4. Boyer-Moore string search algorithm

Boyer-Moore string search algorithm is a famous efficient string searching algorithm in the field of computer science. This algorithm was developed by Robert S. Boyer and J Strother Moore in 1977. Boyer Moore works by preprocessing the pattern first. This algorithm uses information information gathered during the preprocess step to skip sections of the text.

Boyer-Moore compares the pattern with the text from the end of the pattern to the front. So, it is checking from the behind and shifts the pattern if a character mismatch.

In Boyer-Moore there are two shift rule, the first is the bad character rule, by assuming the comparison at position X is failed, then we check the character that is being compared at the text, if that character has appear in the pattern search before then we will only shift by one character, if that character has not appeared and is in the front of the pattern then we shift the pattern so that the character is at position X, and the last case, if the character is not in the pattern then we shift so that the first character of the pattern is at position X+1.

The second shift rule is the good suffix rule, the reason the comparisons begin at the end of the pattern than the beginning is because of this rule. In this rule, we find that a substring t of the text matches with the suffix of the pattern, and a mismatch occur at the next comparison. We need to find if exist the right-most copy t' of t in the pattern such that t' is not a suffix of the pattern and the character at the left of t' in the pattern differs from the character to the left of t in P. If it exists, shift the pattern to the right so that the substring t' is at the same position of the substring t before, if it doesn't exist, shift such that the first character of the pattern started after the position of t in the text. This rule requires two tables to be preprocessed.

Here is some visualization:

We have the text "a pattern matching algorithm" and the pattern "rithm".

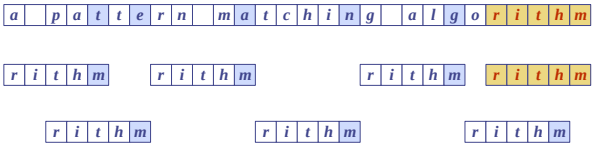

Figure 5. Visualization of Booyer-Moore algorithm

The pseudo code of the Boyer-Moore string search algorithm is like this in C++:

```
int BoyerMoore (string text, string pattern)
{
 int m = pattern.length();
 int n = text.length();
 int table1[256];

 // creating the first char table
 for (int i=0;i<256;++i)
  table[i] = m;
 for (int i=0;i<m-1;++i)
  table[pattern[i]] = m -1 - i;

 // creating the second offset table
 int table2[m];
 int last_prefix = m;
 for (int i=m-1;i>=0;--i)
 {
  bool check = true;
  int j = i+1, k = 0;
  while ((check)&&(j<m))
  {
   if (pattern[j]!=pattern[k])
    check = false;
   ++j;
   ++k;
  }
  if (check)
   last_prefix = i+1;
  table2[m-1-i] = last_prefix-j+m-1;
 }
 for (int i=0;i<m-1;++i)
 {
  int slen = 0;
  for (int j=i, k=m-1;j>=0 && pattern[j]==pattern[k]; --j, --k)
   slen +=1;
  table2[slen] = m - 1 - j +slen;
 }

 // searching
 bool check = true;
 int i = m - 1, j = 0;
 while ((check)&&(i<text.length())
 {
  j = m -1;
  while ((check) && (pattern[j] == pattern[i])
  {
   if (j==0)
   // found
    check = false;
   --i;
   --j;
  }
  if (check)
   j+=max(table2[m-1-j], table[text[i]]);
 }
 if (!check)
  return i;
}
```

Boyer-Moore algorithm has the complexity of:

- Preprocessing time: $\Theta(m + |\Sigma|)$
- Matching time: $\Omega(n/m)$, $O(n)$

So, if we have a text with 1000 characters and a pattern with 5 characters, we would have compare 1000 times or less in average.

2.2. Scribblenauts Unlimited object creation

In the Scribblenauts Unlimited game, we can create any object we want by typing the name of the object to the magic notebook. If we write the unrecognized words, the game recommends some correction. Here are the screenshots:

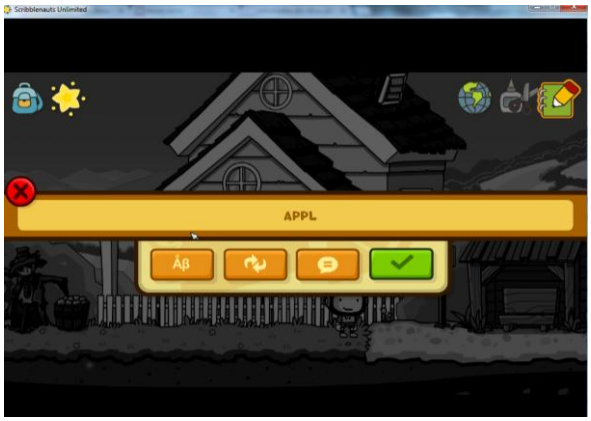

Figure 6. Text box to create object in Scribblenauts Unlimited

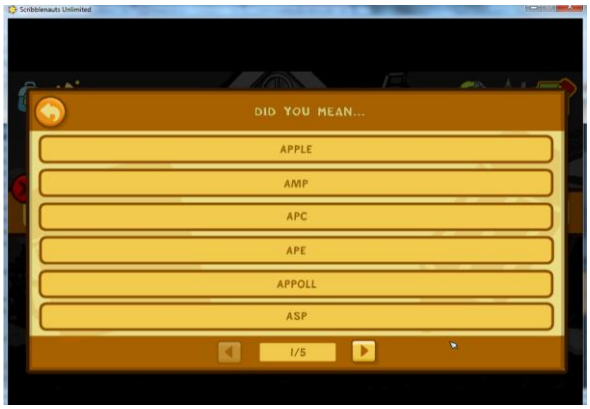

Figure 7. Correction suggestion for Scribblenauts Unlimited

The string matching in the algorithm use one of the string matching algorithms which I don't know. But the implementation will be discussed in the next chapter.

3. IMPLEMENTATION

In the game Scribblenauts unlimited, first the program split the input into words by spaces and then search the word from the list of available text such that all the words is contained in the text. After finding the exact text, it then creates the object in the game and the object can also be manipulated further in the game.

The implementation is just as simple as that but it will require a powerful string matching algorithm to make sure the game run fast.

4. CONCLUSION

From this paper, we can conclude that many things uses string matching algorithm, such as games, word processor, search engine, and many others. Also, there are many strings matching algorithm that differs in the complexity and best cases.

REFERENCES

[1] http://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/stmik.htm
accessed at 19 December 2012

[2] http://www.cplusplus.com/reference/
accessed at 20 December 2012

[3] http://games.kidswb.com/official-site/scribblenauts/unlimited/
accessed at 19 December 2012

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, 21 December 2012

Jordan Fernando / 13510069