Huffman Text Compression Technique

Suherman[1], Andysah Putera Utama Siahaan[2]
Faculty of Computer Science
Universitas Pembanguan Panca Budi
Jl. Jend. Gatot Subroto Km. 4,5 Sei Sikambing, 20122, Medan, Sumatera Utara, Indonesia

Abstract— *Huffman is one of the compression algorithms. It is the most famous algorithm to compress text. There are four phases in the Huffman algorithm to compress text. The first is to group the characters. The second is to build the Huffman tree. The third is the encoding, and the last one is the construction of coded bits. The Huffman algorithm principle is the character that often appears on encoding with a series of short bits and characters that rarely appeared in bit-encoding with a longer series. Huffman compression technique can provide savings of 30% from the original bits. It works based on the frequency of characters. The more the similar character reached, the higher the compression rate gained.*

Keywords— *Huffman, Compression, Algorithm, Security*

I. Introduction

A text is a collection of characters or strings into a single unit. It contains many characters in it that always cause problems in limited storage device and speed of data transmission at the particular time. Although storage can be replaced by another larger one, this in not a good solution if there is another solution. And this is making everyone try to think to find a way that can be used to compress text.

Compression is the process of changing the original data into code form to save storage and time requirements for data transmission.A loseless compression algorithm should emphasize the originality of the data during compression and decompression process[5].By using the Huffman algorithm, text compression process is done by using the principle of the encoding; each character is encoded with a series of several bits to produce a more optimal result. The purpose of writing this paper is to investigate the effectiveness and the shortest way of the Huffman algorithm in the compression of text and explain the ways of compressing text using Huffman algorithm in programming.

II. Theories

Data Compression is the process of shrinking data to smaller bits than the original representation so that it takes less storage space and less transmission time while communicating over a network [2]. Huffman algorithm was created by an MIT student named David Huffman in 1952. It is one of the oldest methods and most famous in text compression [5]. The Huffman code uses the principles similar to Morse code. Each character is encoded only with a few bits series, where characters that often appear with a coded series of short bits and characters that rarely appears is encoded with a longer set of bits. Based on the type of map, the code is used to change the initial message (the contents of the input data) into a set of codeword's. Huffman algorithm uses static methods. A static method is a method that always uses the same code map but the sequence of character appearance can be changed. This method requires two step. The first step to calculate the frequency of occurrence of each symbol and determine the map code, and the last is to convert the message into a collection of code that will be transmitted. Meanwhile, based on symbols coding technique, Huffman uses the symbol wise method. Symbol wise is a method that calculates the frequncy of the characters in every process.Tranforming text characters into symbolwise is not an easy process [3]. The symbol is more often occur will be given shorter code than the symbols that rarely appears.

A. Greedy Algorithm

Greedy Algorithm solves a problem by selecting the best distance at the particular time. Huffman problems can be solved using a greedy algorithm as well [4]. A greedy algorithm is an algorithm which follows the problem-solving metaheuristic of making the optimum choice. By calculating each step, the optimal solution is resolved. For example, applying the greedy strategy to the TSP to visit the nearest unvisited place.Greedy algorithm never finds the global solution. This algorithm is good at finding the nearest solution. Two of the examples of greedy algorithms are Kruskal's and Prim's algorithm.

B. The Relationship to Huffman Algorithm.

David Huffman encoded character by simply using an ordinary binary tree, but after that, David Huffman found that using a greedy algorithm can establish the optimal prefix code. The use of greedy algorithm on Huffman is at the election of two trees with the lowest frequency in a Huffman tree. A greedy algorithm is used to minimize the total cost. Cost is used to merge two trees at the root of frequency equal to the number two fruit trees that are combined. Therefore the total cost of the establishment of the Huffman tree is the total of the whole merger. Therefore, the Huffman algorithm is one example of compression algorithm that uses the greedy algorithm. Our goal is to calculate the total cost incurred to establish the text.

III. Implementation

Thissection tries to figure out the Huffman technique. There are four steps must be accomplished to make the Huffman technique is fully operated. These phases below explain the steps of its algorithm.

A. *Phase One.*

Assume the sentence is “LIKA-LIKU LAKI-LAKI TAK LAKU-LAKU”. The text above is 33 length. It must be categorized base on the character frequency. The *Character_Set* function is to determine how many times each character appears.

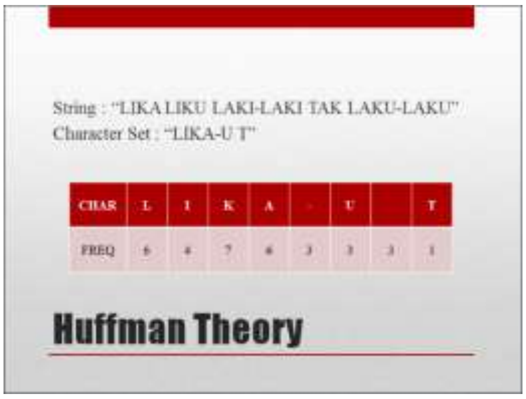
```
  function  Character_Set(text  :  string)  :
string;
  var
   temp  : string;
   result : string;

  begin
   temp := text;
   result := '';

   for i := 1 to length(temp) do
    for j := i + 1 to length(temp) do
     if  temp[i]  =  temp[j]  then  temp[j]  :=
'#';

   j := 1;

   for i := 1 to length(temp) do
    if temp[i] <> '#' then
    begin
     result := concat(result, temp[i]);
     inc(j);
    end;

   Character_Set := result;
  end;
```

The first loop which is done by “for” is to replace the duplicate character into ‘#’. The second loop removes the original text that consists of ‘#’. The illustration is showed below.

```
   Original  Text  :  LIKA-LIKU  LAKI-LAKI  TAK
LAKU-LAKU
   Replaced        Text        :        LIKA-###U
#########T#############
   Character Set : LIKA-U T
```

```
   procedure Character_Freq(text : string);
   var
     temp  : string;
     freq  : byte;

   begin
     temp := Character_Set(text);
     for i := 1 to length(temp) do
     begin
       freq := 0;
       for j := 1 to length(text) do
         if temp[i] = text[j] then inc(freq);
       AddNode(Head, Tail, temp[i], freq);
     end;
   end;
```

The above procedure calculates the frequency of the character occurrence. First, the character set function must be run to obtain a series of a single character used; then the first character until the last character must be compared to the original text to get the frequency. The result will be sent to the node after getting incremented. Figure 1 illustrates the characters and their frequency obtain from Phase One.



String : “LIKA LIKU LAKI-LAKI TAK LAKU-LAKU”
Character Set : “LIKA-U T”

CHAR	L	I	K	A	-	U		T
FREQ	6	4	7	6	3	3	3	1

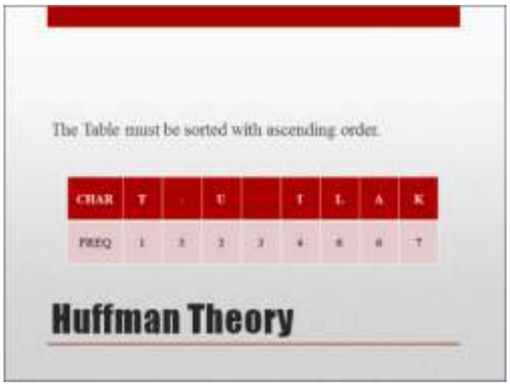
Huffman Theory

Fig. 1 Unsorted character and frequency

The table must be sorted in ascending order and the primary key is the frequency.

```
procedure Tree_Sorting;
var
 tASCII : char;
 tFreq  : byte;

begin
 for i := length(cs) - 1 downto 1 do
 begin
  Current := Head;

for j := 1 to i do
  begin
   if  Current^.Freq  >  Current^.Next^.Freq
then
   begin
    tASCII := Current^.ASCII;
    tFreq := Current^.Freq;
    Current^.ASCII := Current^.Next^.ASCII;
    Current^.Freq := Current^.Next^.Freq;
    Current^.Next^.ASCII := tASCII;
    Current^.Next^.Freq := tFreq;
   end;
   Current := Current^.Next;
    end;

 end;
end;
```

The procedure above is to sort the unsorted list by using bubble sort algorithm. The first node will be compared to the next node, and the larger value will be swapped and moved to the right. Figure 2 demonstrates the result after the sort.



The Table must be sorted with ascending order.

CHAR	T	-	U		I	L	A	K
FREQ	1	3	3	3	4	6	6	7

Huffman Theory

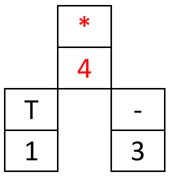
Fig. 2 Character and frequency after sorted

B. *Phase Two.*

After the table is fully sorted, it is time to face the most difficult step; that is to make the Huffman tree. Each node must be categorized and put it on the linked list. The Greedy algorithm takes apart at this section. It needs to combine two nodes and make a new node and make it as a parent of those earlier nodes. Let's see the illustration below:

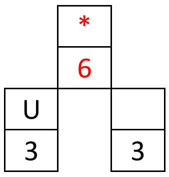
T	-	U		I	L	A	K
1	3	3	3	4	6	6	7

Draw the first two nodes, and release from the table. Make a new node which will be their parent.

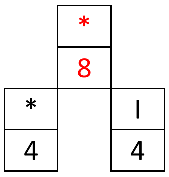


U		*	I	L	A	K
3	3	4	4	6	6	7

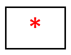
The parent node will be inserted to the table in ascending order using insertion algorithm. The first node is now replaced by the third node before. Moreover, It has to do the same way again until the table consists of one node.

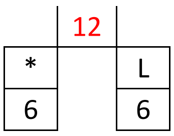


*	I	*	L	A	K
4	4	6	6	6	7

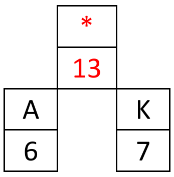


*	L	A	K	*
6	6	6	7	8

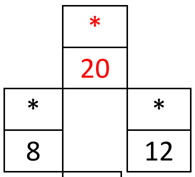




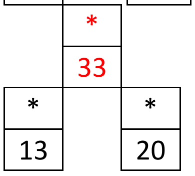
A	K	*	*
6	7	8	12



*	*	*
8	12	13



*	*
13	20



*
33

T	-	U		*	I	L	A	K						
1	3	3	3	4	4	6	6	7						
T	-	U		*	I	*	L	A	K					
1	3	3	3	4	4	6	6	6	7					
T	-	U		*	I	*	L	A	K	*				
1	3	3	3	4	4	6	6	6	7	8				
T	-	U		*	I	*	L	A	K	*	*			
1	3	3	3	4	4	6	6	6	7	8	12			
T	-	U		*	I	*	L	A	K	*	*	*		
1	3	3	3	4	4	6	6	6	7	8	12	13		
T	-	U		*	I	*	L	A	K	*	*	*	*	
1	3	3	3	4	4	6	6	6	7	8	12	13	20	
T	-	U		*	I	*	L	A	K	*	*	*	*	*
1	3	3	3	4	4	6	6	6	7	8	12	13	20	33

Finally, the array consists of 15 nodes.

In this step, there are two modelsof tree.

- Double Linked List.
- Binary Tree Linked List.

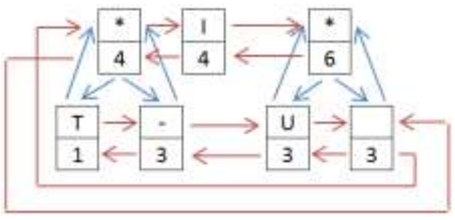


Fig. 3Double & Binary Tree Linked List

Figure 3 shows the hierarchy of the array. The * indicates the node has a parent. Figure 4 is the complete diagram of the Huffman tree.

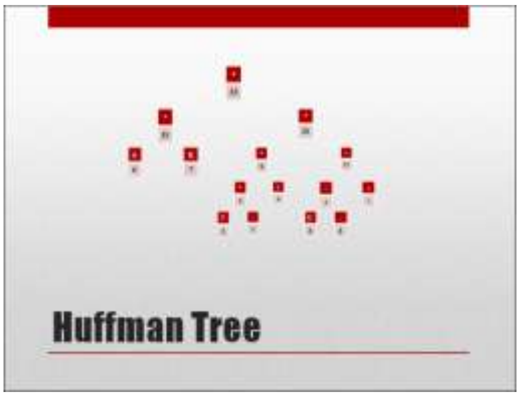


Fig.4Huffman Tree

The aim using two models of linked list is to avoid the searching procedures. The search can be either breadth-first or depth-first search to form the bit code, but it takes time and advanced programming technique. However, a list can be used to replace the backtracking procedure. The procedure track what the parent is.

```
  procedure Huffman_Tree;
  var
   tFreq  : byte;
   tASCII : char;
   InsertN : NodeP;
   NewNode : NodeP;

  begin
   tASCII := '*';
   Current := Head;

   while Current^.Next <> NIL do
   begin
    InsertN := Head;
    tFreq      :=      Current^.Freq      +
Current^.Next^.Freq;
    Current^.Bit   := 0;
    Current^.Next^.Bit := 1;

    while InsertN <> NIL do
    begin
     if tFreq <= InsertN^.Freq then
     begin
      new(NewNode);
      NewNode^.Freq := tFreq;
      NewNode^.ASCII := tASCII;
      NewNode^.Next := InsertN;
      NewNode^.Prev := InsertN^.Prev;
      InsertN^.Prev^.Next := NewNode;
      InsertN^.Prev := NewNode;

      NewNode^.Left  := Current;
      NewNode^.Right := Current^.Next;
      Current^.Parent := NewNode;
      Current^.Next^.Parent := NewNode;
      break;
     end
     else if tFreq > Tail^.Freq then
     begin
      new(NewNode);
      NewNode^.Freq := tFreq;
      NewNode^.ASCII := tASCII;
      Tail^.Next := NewNode;
      NewNode^.Prev := Tail;
      Tail:=NewNode;
      Tail^.Next := NIL;

      NewNode^.Left  := Current;
      NewNode^.Right := Current^.Next;
      Current^.Parent := NewNode;
      Current^.Next^.Parent := NewNode;
      break;
     end;
     InsertN := InsertN^.Next;

    end;

    Current := Current^.Next;
    Current := Current^.Next;
   end;
  end;
```

Huffman Tree procedure is used to form the Huffman tree by processing the earlier linear tree.The nodes must be marked "what is on the left" and "what is on the right". It is done by adding a sign 0 or 1 to the node field.

```
type
 NodeP   = ^Node;
 Node   = record
        ASCII : char;
        Bit  : byte;
        Code  : string;
        Dec  : byte;
        freq  : byte;
        Prev,
        Next  : NodeP;
        Parent,
        Left,
        Right : NodeP;
       end;
```

ASCII	:	where character is memorized.
Bit	:	node sign. (left or right)
Code	:	Huffman code.
Dec	:	decimal code.
Freq	:	occurence of the character.
Next, Prev, Parent, Left, Right	:	represent the connected nodes.

The first two nodes must be combined. The first node will be marked as '0' since it is on the left and the second node will be marked as '1' since it is one the right. Moreover, both nodes have the same parent, and the parent has two children, the nodes. After the parent node is created, it must be inserted into the linked list by combining the value of the node and the value of the linked list. The node must be added to the left of the larger or same value. However, if the parent node is greater than every node, it has to be inserted after the last node.

C. Phase Three.

In this step, the tree is already structured. It is time to retrieve the node sign by making a loop until the parent of the last node is empty (null).

```
procedure Write_Huffman;
var
 result : string;
 bit  : string[1];

begin
 Current := Head;

 repeat
  result := '';
  Cursor := Current;
  Current^.Dec := 0;
  Biner := 1;

  if Cursor^.ASCII <> '*' then
  begin
   repeat
    if (Cursor^.Bit = 0) or
(Cursor^.Bit = 1) then

    begin
     Current^.Dec := Current^.Dec +
(Cursor^.Bit * Biner);
     Biner := Biner * 2;
     str(Cursor^.Bit, Bit);
     insert(Bit, result, 1);
    end;
    Cursor := Cursor^.Parent;
   until Cursor^.Parent = NIL;
  end;

  Current^.Code := result;
  Current := Current^.Next;
 until Current^.Next = NIL;
end;
```

The nodes which are not parents are retrieved. Moreover, the node sign from the node will be inserted into a single string.

D. Phase Four.

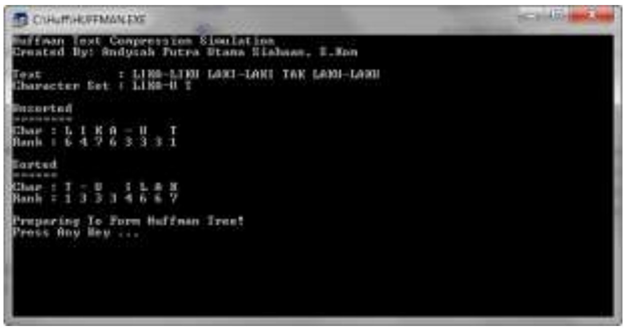
Each node has consisted of code. This phase, everything that has been coded is converted to a Huffman table. Table 1 shows the priority of characters. The character "K" has the most appearance while the character "T" has the less one.Characters with the highest emergence is having the shortest binary code.

TABLE IHUFFMAN TABLE

Char	Freq.	Code	Bit Len.	Code Len.
T	1	1000	4	4
-	3	1001	4	12
U	3	1100	4	12
	3	1101	4	12
I	4	101	3	12
L	6	111	3	18
A	6	00	2	12
K	7	01	2	14
				96

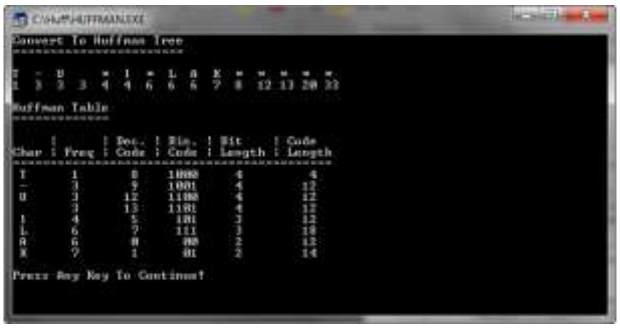
Each code represents the character. It consists of a few digit of the binary string. The previous text will be transformed into the new binary set. The old eight-bit binary is replaced by the new one. Let's see the example below:

```
   Original Text  :  LIKA-LIKU  LAKI-LAKI  TAK
LAKU-LAKU
   Original Code :

   111  101  01  00  1001  111  101  01  1100  1101
   111  00  01  101  1001  111  00  01  101  1101  1000
   00  01  1101  111  00  01  1100  1001  111  00  01
   1100

   Bit Code :

   11110101 00100111110101110011011110001101
   10011110001101110110000 0111011110001110
   01001111 00011100

   Decimal Code :

   254, 39, 215, 55, 141, 158,
   55, 96, 119, 142, 79, 28
```



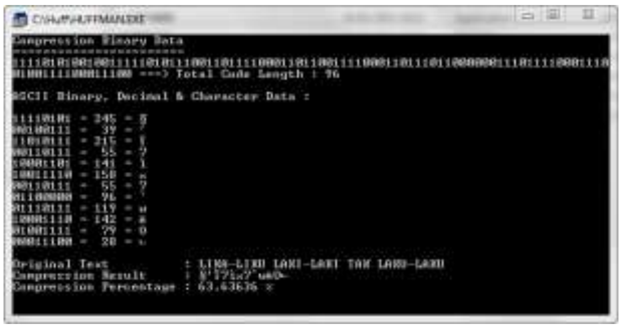
```
C:\Huff\HUFFMAN.EXE
Huffman Text Compression Simulation
Created By: Andysah Putra Utama Siahaan, S.Kom

Text          : LIKA-LIKU LAKI-LAKI TAK LAKU-LAKU
Character Set : LIKA-U T

Unsorted
========
Char : L I K A - U   T
Rank : 6 4 7 6 3 3 3 1

Sorted
======
Char : T - U   I L A K
Rank : 1 3 3 3 4 6 6 7

Preparing To Form Huffman Tree!
Press Any Key ...
```

Fig. 5Running Program (Part. 1)



```
C:\Huff\HUFFMAN.EXE
Convert To Huffman Tree
=======================

T - U   * I * L A K * * * * *
1 3 3 3 4 4 6 6 6 7 8 12 13 20 33

Huffman Table
=============

     |      | Dec. | Bin. | Bit    | Code
Char | Freq | Code | Code | Length | Length
==========================================
 T      1      8   1000     4        4
 -      3      9   1001     4       12
 U      3     12   1100     4       12
        3     13   1101     4       12
 I      4      5    101     3       12
 L      6      7    111     3       18
 A      6      0     00     2       12
 K      7      1     01     2       14

Press Any Key To Continue!
```

Fig. 6Running Program (Part. 2)



```
C:\Huff\HUFFMAN.EXE
Compression Binary Data
=======================
1111010100100111110101110011011110001101100111100011011101100000011101111000111
001001111000011100 ---> Total Code Length : 96

ASCII Binary, Decimal & Character Data :

11110101 - 245 - [illegible]
00100111 -  39 - '
11010111 - 215 - [illegible]
00110111 -  55 - 7
10001101 - 141 - [illegible]
10011110 - 158 - [illegible]
00110111 -  55 - 7
01100000 -  96 - `
01110111 - 119 - w
10001110 - 142 - [illegible]
01001111 -  79 - O
00011100 -  28 - [illegible]

Original Text          : LIKA-LIKU LAKI-LAKI TAK LAKU-LAKU
Compression Result     : [illegible]
Compression Percentage : 63.63636 %
```

Fig. 7Running Program (Part. 3)

Figure 5 to Figure 7 are the Huffman running processes. The original text before compression takes 33 characters length. Sooner after being compressed, the string only takes 12 characters. It saved 21 characters.

The illustration:

```
   Original Text Length   :  33
   Coded Text Length      :  12
   Saving Rate            :  (33 - 12) / 33 *
100 %
                             63.63636 %
```

By doing this algorithm, the compression rate achieves 63% of the original message. Of course, it will save the storage capacity.

IV. Conclusion

This research should explain the basic technique on how to implement the compression algorithm. Huffman algorithm can combine with the greedy algorithm which always find the easiest ways from the nearest node. There are four phases in the Huffman algorithm to compress a text; the first is the phase to manage and get the frequency of the character. The second is the formation of the Huffman tree; the third is to form the code from the node sign. Moreover, the last phase is the encoding process. However, in this paper, the author only perform the encoding method, the decoding is under a project. Applying Huffman in compression will again more space in the storage. Huffman encoding is very powerful to the text message which has similar character occurrence.

References

[1] A. Malik, N. Goyat and V. Saroha, "Greedy Algorithm: Huffman Algorithm," International Journal of Advanced Research in Computer Science and Software Engineering, vol. 3, no. 7, pp. 296-303, 2013.

[2] A. S. Sidhu and M. Garg, "Research Paper on Text Data Compression Algorithm using Hybrid Approach," IJCSMC, vol. 3, no. 12, pp. 1-10, 2014.

[3] H. Al-Bahadili and S. M. Hussain, "A Bit-level Text Compression Scheme Based on the ACW Algorithm," International Journal of Automation and Computing, pp. 123-131, 2010.

[4] I. Akman, H. Bayindir, S. Ozleme, Z. Akin and a. S. Misra, "Lossless Text Compression Technique Using Syllable Based Morphology," International Arab Journal of Information Technology, vol. 8, no. 1, pp. 66-74, 2011.

[5] M. Schindler, "Practical Huffman coding," 1998. [Online]. Available: http://www.compressconsult.com/huffman/.