Application of Graph Theory in DEFLATE Compression Algorithm

Nathaniel Evan Gunawan (13516055)
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
13516055@std.stei.itb.ac.id

***Abstract*— In 1948, Claude E. Shannon published a paper titled "A Mathematical Theory of Communication", and with it the field of information theory was established. Strides were then made to develop a way of encoding information, beginning with the Shannon-Fano coding algorithm. Today, there are at least more than 20 different compression algorithms, and this proves that data compression still maintains its relevance and has not fallen out of use in an era where data transmission speeds have increased several hundred folds during the last 2 decades. This paper discusses about the DEFLATE compression algorithm, its underlying mathematical concepts, a brief overview of its mode of operation, as well as its applications.**

***Keywords*—Graph theory, Huffman coding, LZ77, DEFLATE**

I. About Data Compression

Information encoding predates the digital age by at least a century. One of the most well-known method of information encoding is the Morse code, developed throughout the 1830s and 1840s by Samuel F. B. Morse, an inventor of the telegraph. Telegraphs were only capable of sending simple electrical pulses along wires, and thus Morse saw the need to develop an encoding scheme that translates every letter in the English language into sequences of signals in the form of dots, dashes or any combination of both, in order to enable meaningful communication via the telegraph. Morse devised the code so that the most common letters in the English language are encoded in short codes; for instance, the letter "E", which is the most common letter in English, gets only a single dot as its Morse code. The letter "I", which is also a frequently-occurring letter in the language, gets two dots as its Morse code. On the other hand, the letter "Q" gets 2 dashes and a dot followed by a dash as its Morse code, being one of the least common letters in the English language. Morse code still enjoys widespread use today especially in the field of aviation and remains popular with amateur radio operators.

Claude E. Shannon, born in 1916 and died at the age of 85 in 2001, was an American mathematician, electrical engineer and cryptographer known as the "father of information theory". The year 1948 saw the publication of his article titled "A Mathematical Theory of Communication". With it, Shannon proposed a coding technique known as the Shannon–Fano coding, a method for constructing a prefix code for a set of symbols based on the frequencies of occurrence of every symbol. Unfortunately, the Shannon–Fano algorithm does not always result in an optimal code. In 1952, David Huffman, a fellow student at the Massachusetts Institute of Technology, discovered a way to optimally construct such a code. While the Shannon–Fano algorithm works in a top-down approach, Huffman's is the opposite; it works in a bottom-up approach.

Fast forward to 1977 and 1978, two Israeli computer scientists by the name of Abraham Lempel and Yaakov Ziv developed two compression algorithms: the LZ77 and LZ78, also known as the LZ1 and LZ2 respectively. These two algorithms were the first major dictionary-based compression algorithms, and they both lay the basis for other dictionary-based compression algorithms developed after.

There are 2 classes of data compression methods: lossy and lossless. Lossy data compression is a type of data compression which, as the name may imply, involves a degree of information loss in order to compress the data. Lossy data compression is typically done when a higher compression ratio is needed, i.e. when the uncompressed data needs to be compressed as much as possible, and when the trade-off in information is worth it. This is very much the case with popular media encoding formats like MP3 and JPEG; these formats can reduce the size of the uncompressed data down to only 10% of the original size, while being able to retain a degree of quality discernible to the human senses. Lossless data compression, on the other hand, is a class of data compression in which every bit of information from the uncompressed data is retained and the original data may be reconstructed back from the compressed data. This type of data compression is necessary in cases where any deviation from the original data or loss in information could alter the data in its entirety and/or is simply unfavourable. Lossless compression is used in archive file formats such as ZIP, as well as several media encoding formats like PNG, GIF or FLAC.

DEFLATE is a lossless data compression algorithm created by Phil Katz in the early 1990s for Katz's own proprietary file archiving program named PKZIP, and it utilises a combination of the LZ77 algorithm and the Huffman encoding to achieve its purpose. Today, this algorithm finds use in gzip compressed files, PNG files and ZIP files which this algorithm was originally designed for.

II. THEORETICAL BASIS

A. Graph

In order to understand how Huffman's code (and, in turn, DEFLATE) works, a solid basic grasp on graph theory is necessary.

A ***graph*** is a structure consisting of a set of objects in which any given object in the structure may be related to another object. These objects are called ***vertices*** (singular: vertex), and the relations between any given pair of vertices are represented by ***edges*** (singular: edge).

More precisely, the mathematical definition of a graph is an ordered pair G = (V, E) consisting of a set of vertices (represented by V) and a set of edges (represented by E). These edges are essentially 2-element subsets of set V, thereby connecting a vertex with another. The set of vertices (V) must not be empty, whereas E may be empty. In a case where a graph doesn't have any edges, the graph is said to be an empty graph, but a graph must have at least one vertex.

Typically, graphs may be classified based on whether:
1. there are any edges that connect the same pair of vertices,
2. the edges have directions,
3. there are any edges that connect a vertex to itself, and
4. whether the edges carry a certain "weight", or distance.

A graph which fulfils conditions:
- 1, is called a ***multigraph***
- 3, or both 1 and 3, is called a ***pseudograph***
- 2, or both 2 and 3, is called a ***simple directed graph***
- 1, 2 and 3, is called a ***multiple directed graph***
- 4, is called a ***weighted graph***.

A graph which does not fulfil conditions:
- 2, is called an ***undirected graph***
- 4, is called an ***unweighted graph***
- 1, 2, and 3, is called a ***simple graph***.

Additionally, by definition, a multigraph and a pseudograph are undirected graphs, meaning they may not fulfil condition 2. A pseudograph may not be a multigraph. A multiple directed graph is not a simple directed graph. Any graph whose edges are weighted is a weighted graph.

Consider the graph shown in Figure 1:

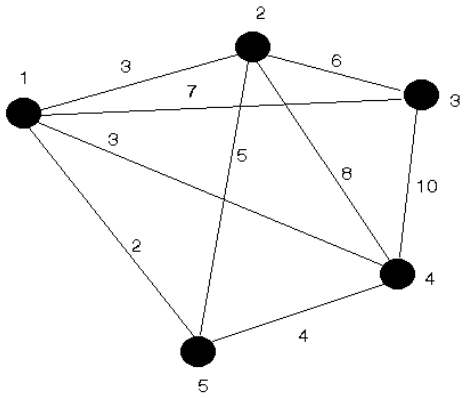

Figure 1: An example of a graph. Source: http://people.brunel.ac.uk/~mastjjb/jeb/or/graph.html (accessed Monday, 10 December 2018)

In the graph, the vertices are represented by the solid black dots; there are 5 vertices in the graph, labelled 1, 2, 3, 4 and 5. The edges are the lines connecting between a vertex and another vertex; there are 9 edges in the graph in total. This graph is an undirected graph, since the edges do not have a direction. This graph is a weighted graph, since each of the edges carry a "weight"; if the vertices denote cities on a map, the weight of the edges can be said to represent the distances between them. The graph above is also a simple graph, because all the edges connect a different pair of vertices, are undirected, and none of them connect a vertex with itself.

B. Relevant Graph Terminologies

- Connected
  A graph is said to be connected if for every possible pair of vertices, there is a pathway that connects them, directly (i.e. an edge directly connects them) or indirectly (i.e. multiple edges make up the pathway that connects them). Taking the graph in Figure 1 as an example, it is a connected graph, because every vertex in the graph is connected to one another through at least an edge.
- Cycle
  A cycle is a set or path of edges which, if traced from a certain vertex, would lead to that same vertex. A cyclic graph is therefore any graph that features such cycle. The graph from Figure 1 is an example of such graph; there are several cycles in the graph, one of which is highlighted with the red line in the image below:

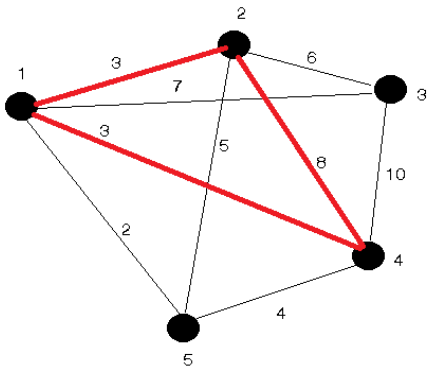

Figure 2: Figure 1, with a cycle highlighted. Source and access date: refer to Figure 1. Edited by author

- Subgraph
  A subgraph is a graph formed from a subset of the vertices and edges of another graph. As a rule, an edge "copied over" to the subgraph must include the vertices it connects in the graph from which the subgraph is derived. Vertices, however, are not tied to such rule and therefore as many vertices as there are in the original graph may be used to form a subgraph without carrying over the edges that connect them. Figure 3 is an example of a valid subgraph from Figure 1:

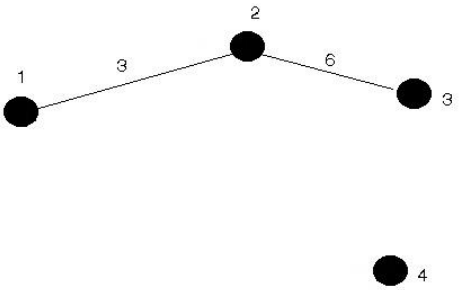

Figure 3: A sample subgraph of Figure 1. Source and access date: refer to Figure 1. Edited by author

C. Tree

A tree in graph theory is defined as an undirected graph in which for any given pair of vertices, there's only one possible pathway connecting them. In other words, a tree is an undirected, connected graph without any cycles.

Suppose there are n vertices in a tree. Mathematically speaking, the properties of a tree are as follows, and they are all equivalent:

- A tree is a connected graph with (n – 1) edges, and would become disconnected if an edge is to be removed
- A tree is acyclic, and a cycle is created if an edge is added to the tree
- Any two vertices of the tree are connected by only one possible path

A tree may be created as a subgraph of Figure 1; one such example is illustrated on Figure 4 below:

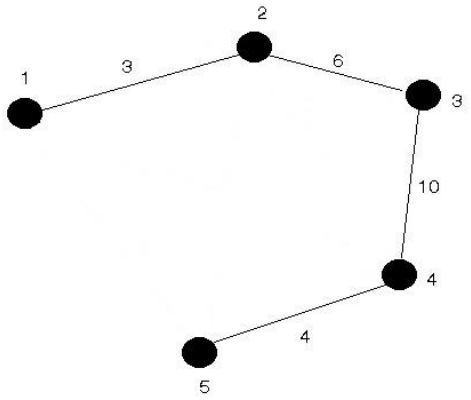

Figure 4: A subgraph from Figure 1 forming a tree. Source and access date: refer to Figure 1. Edited by author

A rooted tree is a variant of tree in which one vertex has been determined as a root and whose edges have been given a direction. For the sake of this paper, it will be assumed that the direction of these edges moves away from the root, and conventionally the arrows that display these directions may be omitted. An example of a possible rooted tree created as a subgraph from Figure 1 (with node 3 as root) is as follows:

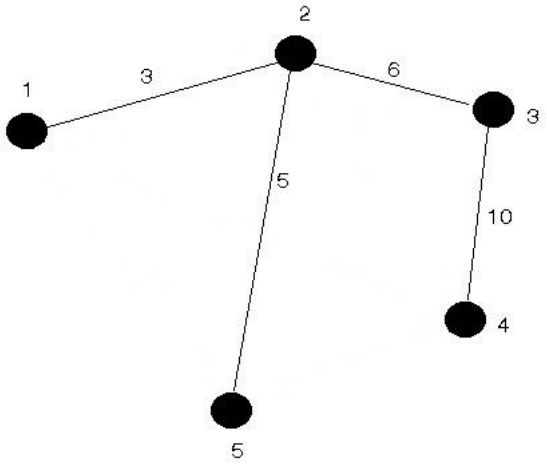

Figure 5: A rooted tree created from Figure 1. Root is assigned at node 3. Source and access date: refer to Figure 1. Edited by author

D. Relevant Rooted Tree Terminologies

- *Parent*
  A parent of a vertex *v* is the vertex directly connected to it on the path to the root. Every vertex except the root has a parent. For example, on Figure 5, the parent of vertex 5 is vertex 2, the parent of vertex 4 is vertex 3, and so on.
- *Leaf*
  A leaf of a graph is a vertex which has a degree of exactly 1; that is, the vertex which is only connected to one single edge. The leaves of the rooted graph in Figure 5 are therefore vertices 1, 5 and 4.
- *Child*
  A child of a vertex *v* is the vertex to which *v* is a parent of. In other words, a child is the opposite of a parent. In Figure 5, The child of vertex 2 is vertices 5 and 1, the child of vertex 3 is vertex 4, and so on.
- *Height*

The height of a vertex *v* is the longest downward path to a leaf from the vertex. The leaves have a height of 0, the parents of the leaves have a height of 1, *et cetera*.

- *Depth*
  The depth of a vertex is the opposite of its height; that is, the depth of a vertex is its distance to the root vertex. Vertices 2 and 4 in Figure 5, for instance, would have a depth of 1, while vertex 3 would have a depth of 0 since it is the root itself.

III. HUFFMAN CODING

A. Overview

Huffman coding is a lossless encoding/compression algorithm which reads a string of symbols or characters, recognises what symbols are there and how frequently they appear in the string. Huffman's coding then constructs a rooted tree based on the obtained information, starting from the least frequently occurring characters or symbols to the most frequently occurring characters or symbols. This is where it sets itself apart from its "predecessor", the Shannon–Fano algorithm, which attempts to accomplish the same task but starts from the most occurring characters or symbols to the least frequently occurring ones. The idea behind Huffman coding is that the most frequently occurring characters will be represented with the smallest code while the least occurring ones get the largest code, potentially resulting in a very good compression ratio depending on how many recurring characters are there in the string. These codes are called prefix codes, meaning that none of the resulting Huffman codes for each character will be ambiguous or conflicting.

For example, assume a 12-character string "abcbcdcdedef". Simply from looking at the string above, we know that there are 6 different characters: a, b, c, d, e and f. The table below shows what characters are there in the string, and their frequencies and probabilities:

Character	Frequency	Probability of occurrence
a	1	1/12
b	2	2/12
c	3	3/12
d	3	3/12
e	2	2/12
f	1	1/12

The table above will be sorted based on each character's probability of occurrence in descending order:

Character	Frequency	Probability of occurrence
a	1	1/12
f	1	1/12
b	2	2/12
e	2	2/12
c	3	3/12
d	3	3/12

The steps to construct a Huffman tree are, in order:

- Pick 2 characters of the string with the lowest frequency or probability of occurrence.
- Make 2 nodes representing each of the 2 characters.
- Give both nodes one parent representing the total occurrence of both characters.
- Assign a bit, 0 or 1, to each of the 2 edges which connect the parent with the children.
- Pick the next character(s) with lowest frequency from the remaining "pool" of characters, disregarding the previous characters which have been represented with their respective nodes.
- Make a node representing the character, then pair it with an already existing parent of two children.
- Give both nodes one parent representing the total occurrence of all the descendant characters.
- Assign a bit, 0 or 1, to each of the edges which connect the parent with the children.
- Repeat steps 5-8 until the character with the most occurrence has been reached.

The resulting Huffman tree is as follows:

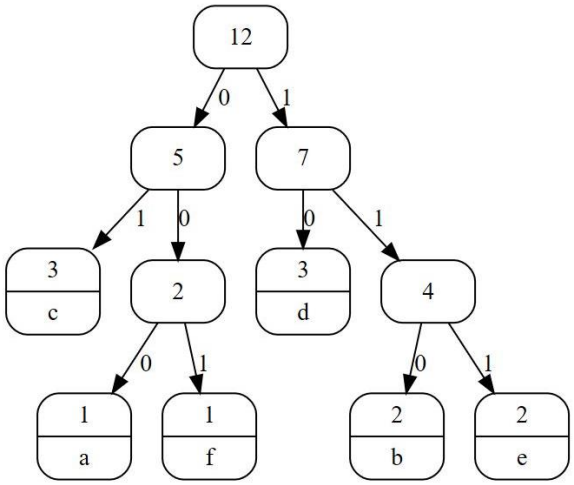

Figure 6: Huffman tree of string "abcbcdcdedef". Graph drawn with assistance of the following website: http://www.csfieldguide.org.nz/en/interactives/huffman-tree/index.html

The resulting bytecode for every character can be obtained by tracing the edges from the root to the vertex which contains the respective character. For instance, to obtain the Huffman code for character f, observe that the route from the root to the vertex passes the 12 → 5 edge, 5 → 2 edge, and finally 2 → 1(f) edge, hence resulting in a Huffman code of 001. Below is a table for the resulting Huffman codes for each character in the string:

Character	Huffman code
a	000
b	110
c	01
d	10

e	111
f	001

IV. DEFLATE COMPRESSION ALGORITHM

DEFLATE Compression Algorithm is a lossless data compression algorithm first conceived by Phil Katz for his proprietary archive packaging tool PKZIP. The algorithm uses a combination of LZ77 algorithm and Huffman coding to achieve its purpose.

A. What is the LZ77 algorithm?

The LZ77 algorithm is a lossless data compression algorithm published in papers by two Israeli computer scientists by the name of Abraham Lempel and Yaakuv Ziv. The LZ77 compression algorithm, along with its other variant, the LZ78, are both very influential compression algorithms in the realm of computer science that it has formed the basis of several ubiquitous compression formats, including GIF, PNG and ZIP, and up to this day no other algorithm has ever replaced these two algorithms, further strengthening its prominent role.

The LZ77 algorithm is a dictionary coder. A dictionary coder is a class of lossless compression algorithms which reads a string (in this case, the text to be compressed) and matches it between an already existing set of strings or characters stored by the decoder/encoder in a data structure called the "dictionary". During encoding, the algorithm maintains a *sliding window* (also called the *search buffer*) which stores an amount of the most recent data; this window may range in size from 2 kilobytes to 32 kilobytes. The program also maintains a *look-ahead buffer* which reads a certain number of characters or symbols from the string simultaneously. These characters will then be compared with the *search buffer*, and the encoder will check if there is any part of the look-*ahead buffer* that matches part of the *search buffer*. If it does, the encoder will return a tuple consisting of 3 elements: the distance to matching string, the length of matching string, and the next character after the matching string. If it doesn't, the encoder will still return a zero tuple, with the last element in the tuple representing the current character.

For example, let us the string "Blah blah blah blah blah!", with a *search buffer* of 5 characters and a *look-ahead buffer* of 5 characters. The characters "B", "l", a", "h", " " and "b" – which compose the first 6 characters in the string – aren't found anywhere in the string yet. The encoded will return the tuples <0,0,B>, <0,0,l>, <0,0,a>, <0,0,h>, <0,0, >, and <0,0,b> for each of the first 6 characters. However, upon reading the next 5 characters, which are, in order, "l", "a", "h", " ", "b", the encoder detects this very sequence of characters in the *search buffer*. According to the algorithm, then, these 5 characters will automatically be encoded as <5, 5, l>. However the same pattern is again picked up two more times, and the previous tuple that was returned by the algorithm eventually becomes <15, 5, l>. Reading the next 4 characters in the string, the encoder can see that there are still 3 more characters belonging to the pattern, before it sees an exclamation mark character "!" which doesn't fit in the previous pattern. The previous tuple now becomes <18, 5, !> and the final exclamation mark is encoded as <0,0,!>. The resulting encoding of the string by the LZ77 encoding algorithm, therefore, becomes <0,0,B><0,0,l><0,0,a><0,0,h><0,0, ><0,0,b><18,5,!><0,0,!>.

After the string has been compressed by the LZ77 algorithm, the DEFLATE compressor will then either construct a Huffman tree based on the compression result by the LZ77 or use an already pre-existing Huffman encoding table. The DEFLATE compressor will then proceed to further compress the data, resulting in a compressed archive with a ratio possibly as low as 25% or even less, depending on the files being compressed and the content of the files themselves.

Let us look at the DEFLATE64 algorithm (a slightly different variant of DEFLATE algorithm with negligible differences) in practice. A folder named "random" containing 223 images with a total size of 339 megabytes, as shown below:

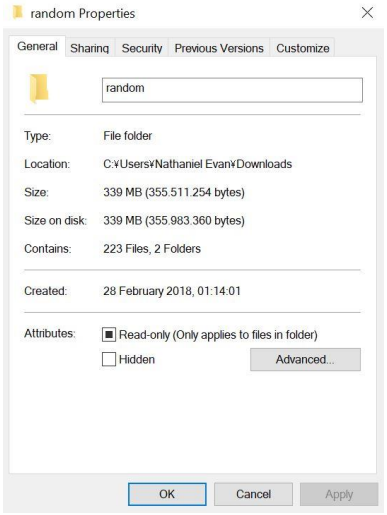

Figure 7: "random" folder before compression using DEFLATE algorithm.

Applying the DEFLATE64 algorithm to the folder results in a .zip archive with 324 megabytes in size, only managing to save 15 megabytes of memory off the original uncompressed data.

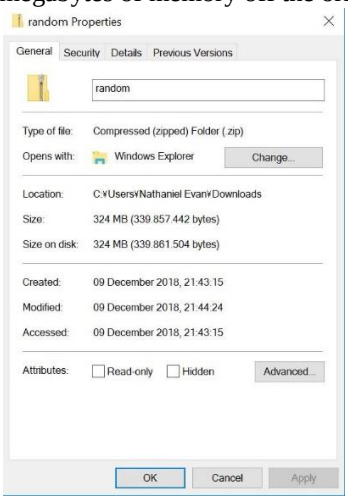

Figure 8: Folder "random", after compression with DEFLATE64 into .zip file

Another folder named “large” consists of 3 large text files totalling up to 10.6 megabytes of data, shown below.

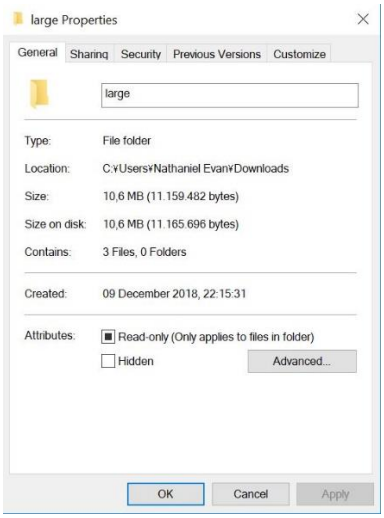

Figure 9: A folder named "large", uncompressed.

After applying DEFLATE64 algorithm to the folder, the size of the resulting archive shrinks greatly in size, amounting to a meagre 2.79 megabytes.

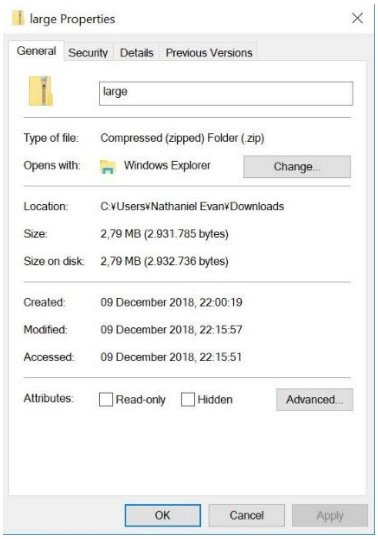

Figure 10: Folder "large", after compressed with DEFLATE.

V. CONCLUSION

Discrete mathematics continues to prove to be useful in the rapid development of computer science; graph theory, in particular, ultimately gave birth to Shannon–Fano’s algorithm, Huffman’s coding, and eventually the DEFLATE compression algorithm, which even today finds ubiquitous use in a number of areas especially file encoding in the form of GIF, PNG and ZIP. It is also apparent that the DEFLATE compression algorithm works best when compressing text files than image files.

VII. ACKNOWLEDGMENT

The author would like to send his utmost gratitude to Dr. Rinaldi Munir as the instructor/lecturer of IF2120 for providing the author with an opportunity to write this paper and apply the acquired discrete mathematics knowledge to use. The author would also like to thank the authors of the references that have been used or cited in this paper for their generosity of allowing the author to utilise these resources and ultimately giving the author more, fresh insight into the realm of computer science.

REFERENCES

[1] Feldspar, Antaeus. *An Explanation of the DEFLATE Algorithm.* https://zlib.net/feldspar.html, accessed December 9, 2018.

[2] Calmarius. *ZLIB + Deflate file format*. http://calmarius.net/?lang=en&page=programming%2Fzlib_deflate_quick_reference, accessed December 9, 2018.

[3] Fraile, Raul. *How GZip Compression Works*. https://2014.jsconf.eu/speakers/raul-fraile-how-gzip-compression-works.html#transcript, accessed December 10, 2018.

[4] The Royal Society Publishing. *Claude Elwood Shannon 30 April 1916 — 24 February 2001*. https://royalsocietypublishing.org/doi/abs/10.1098/rsbm.2009.0015, accessed December 10, 2018.

[5] http://www.csfieldguide.org.nz/en/interactives/huffman-tree/index.html, accessed December 10, 2018.

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, 10 Desember 2018

Ttd (scan atau foto ttd)

Nathaniel Evan Gunawan (13516055)