*Computers, Materials & Continua* CMC, vol.64, no.1, pp.17-30, 2020

# Text Compression Based on Letter's Prefix in the Word

**Majed AbuSafiya**[1, *]

**Abstract:** Huffman [Huffman (1952)] encoding is one of the most known compression algorithms. In its basic use, only one encoding is given for the same letter in text to compress. In this paper, a text compression algorithm that is based on Huffman encoding is proposed. Huffman encoding is used to give different encodings for the same letter depending on the prefix preceding it in the word. A deterministic finite automaton (DFA) that recognizes the words of the text is constructed. This DFA records the frequencies for letters that label the transitions. Every state will correspond to one of the prefixes of the words of the text. For every state, a different Huffman encoding is defined for the letters that label the transitions leaving that state. These Huffman encodings are then used to encode the letters of the words in the text. This algorithm was implemented and experimental study showed significant reduction in compression ratio over the basic Huffman encoding. However, more time is needed to construct these codes.

**Keywords:** Text compression, Huffman encoding, deterministic finite automata.

## 1 Introduction

Text compression is a well-known problem in computer science [Bell, Cleary and Witten (2018); Salomon (2007)]. Huffman encoding [Huffman (1952)] is one of the most known and efficient text compression algorithms. The idea of the Huffman encoding is to encode more frequently used letters in the text with shorter codes and less frequently used ones with longer codes. The literature is rich with variations of Huffman encodings. A recent survey paper about Huffman encoding, mechanisms and variants can be found in Moffat [Moffat (2019)]. A comparative study of a number of text compression algorithms was conducted by Shanmugasundaram et al. [Shanmugasundaram and Lourdusamy (2011)]. Dath et al. [Dath and Panicker (2017)] did a word by word compression instead of byte by byte compression. A byte by byte instead of bit by bit decoding of Huffman-encoded text was proposed by Choueka et al. [Choueka, Klein and Perl (1985)]. Chung et al. [Chung and Wu (1999)] represented a Huffman tree with an array data structure to achieve faster decoding. Enhancing compression by composing Lempel Ziv after Huffman compression was proposed by Saravanan el al. [Saravanan and Surender (2013)]. Klein et al. [Klein, Saadia and Shapira (2019)] proposed a new dynamic Huffman encoding that performs at least as good as static

[1] Al-Ahliyya Amman University, Amman, 19328, Jordan.

[*] Corresponding Author: Majed AbuSafiya. Email: majedabusafiya@gmail.com.

Received: 19 November 2019; Accepted: 19 February 2020.

Huffman encoding. Double Huffman encoding, where compression is applied on the code of the letter, can be found in Knuth et al. [Knuth (1985); Vitter (1987); Arshad, Saleem and Khan (2016)]. Javed et al. [Javed and Nadeem (2000)] proposed an adaptive Huffman encoding scheme where the codes are adjusted during the compression process. Tseng et al. [Tseng, Jiang, Pan et al. (2012)] enhanced Huffman encoding for the purposes of encryption along with compression. An improved Huffman coding method for information storage in DNA was described in Ailenberg et al. [Ailenberg and Rotstein (2009)]. Finite automata was used for compression in the work of Culik et al. [Culik and Kari (1993); Hafner, Albert, Frank et al. (1998)] and both for images and videos and not for text. It is known that lossy compression gives better compression ratios than lossless compression. However, lossy compression is suitable for images and videos but not suitable for text data. Cellular automata was used in text compression by Khan et al. [Khan, Choudhury, Dihidar et al. (1999)].

To contrast the proposed approach with related work, it can be seen that a lot of the related work is directed towards enhancing Huffman encoding. Examples of these enhancements include: more optimized implementations, composition with other compression algorithms, and adaptation. On the other hand, the proposed algorithm did not change or enhance the Huffman encoding. It just used it in a different way, and specifically for natural language text documents, to get better compression ratios. Instead of using one Huffman encoding to encode the letters of the text, multiple Huffman encodings are used. Huffman encoding, to encode a letter, depends on the prefix that precedes this letter in the word being encoded. A deterministic finite automaton (DFA) [Hopcroft and Ullman (1979)] that recognizes the words of the text document is constructed while maintaining the frequencies of letters that cause a transition. This frequency information is used to generate different Huffman encodings for every state in the DFA for the set of letters that may cause a transition from that state.

This work may be confused with what is called adaptive Huffman encoding. The multiple Huffman encoding that is proposed in this paper is not based on adapting Huffman code while the data is streamed. The Huffman encoding that will be used to code a given letter will depend on the prefix that precedes that letter in its word.

This paper is organized as follows: Section 2 describes the proposed algorithm. Section 3 shows an example that illustrates the proposed algorithm. Section 4 presents the experimental study that was conducted to compare the proposed approach with the normal Huffman encoding. The paper ends with conclusions and a set of references.

## 2 Proposed algorithm

The proposed algorithm is shown Fig. 1. In the following subsections, these steps will be elaborated.

```
COMPRESS(T)
1 DFA=BUILD-DFA(T)
2 BUILD-FREQUENCY-LISTS(DFA)
3 BUILD-HUFFMAN-CODES(DFA)
4 ENCODE(T)
```

**Figure 1:** The proposed algorithm

### *2.1 Building deterministic finite automaton for input Text*

Before presenting the algorithm to build the DFA of the input text *T,* a number of fields are defined for the states of the DFA. These fields are shown in Tab. 1.

**Table 1:** The fields of a state *s* of the DFA

| Field Name | Initial value | Description |
|---|---|---|
| *s.string* | empty string | The string that transfers the DFA from the start state to *s*. |
| *s.transitions* | {} | The set of transitions from state *s* to some other state. |
| *s.frequency* | **0** | The number of times the state *s* is visited while scanning the words of the text to build the DFA. |
| *s.frequencyList* | < > | A list of frequency records (*s′*, *s′.frequency*) with a record for every *transition* from *s* to *s′* |
| *s.nextState(l)* | **null** | returns the state *s′* in case there is a transition from *s* to *s′* labeled with letter *l* and **null** otherwise. |
| *s.isFinal* | **false** | is **true** if *s* is a final state and **false** otherwise. |

The algorithm BUILD-DFA(*T*) is shown in Fig. 2. The input is the text to compress *T*. The DFA is initialized by creating its start state. There is a loop where the words of *T* are taken one at a time, appended by a *space* character and then added to the DFA. The algorithm for adding a word to the DFA will be explained below. The reason for adding a space is to mark the end of the word.

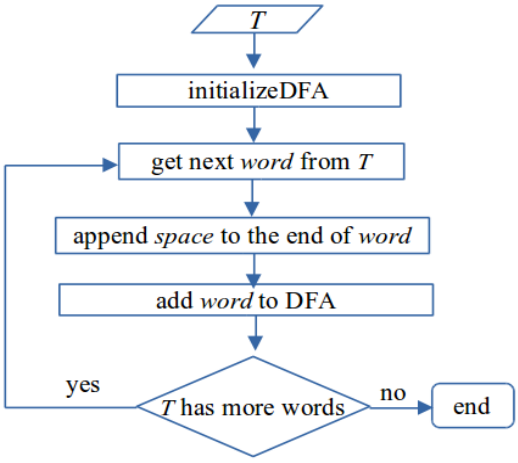


**Figure 2:** BUILD-DFA(*T*) algorithm

The algorithm in Fig. 3 shows how a *word* from *T* is added to the DFA. It starts by setting the *currentState* to be the start state of the DFA. The loop takes the letters of the *word*, one letter (*l*) every iteration. It looks for a transition from *currentState* that is labeled with (*l*). If no such transition exits, *nextState* will be **null**. This requires creating *nextState* setting its *string* field to be *string* field of the *currentState* appended with (*l*) and then adding it the DFA.*states.* Also, a new *transition* from the *currentState* to *nextState* that is labeled with (*l*) will be created and then added to the *currentState.transitions*. At the end of each iteration, the *frequency* field for the *nextState* is incremented and then

*currentState* becomes the *nextState*. This should be done in every iteration, whether *nextState* was created or found. Once the loop terminates, all the letters of the word are consumed including the appended space, and *currentState* is set to be a final state.

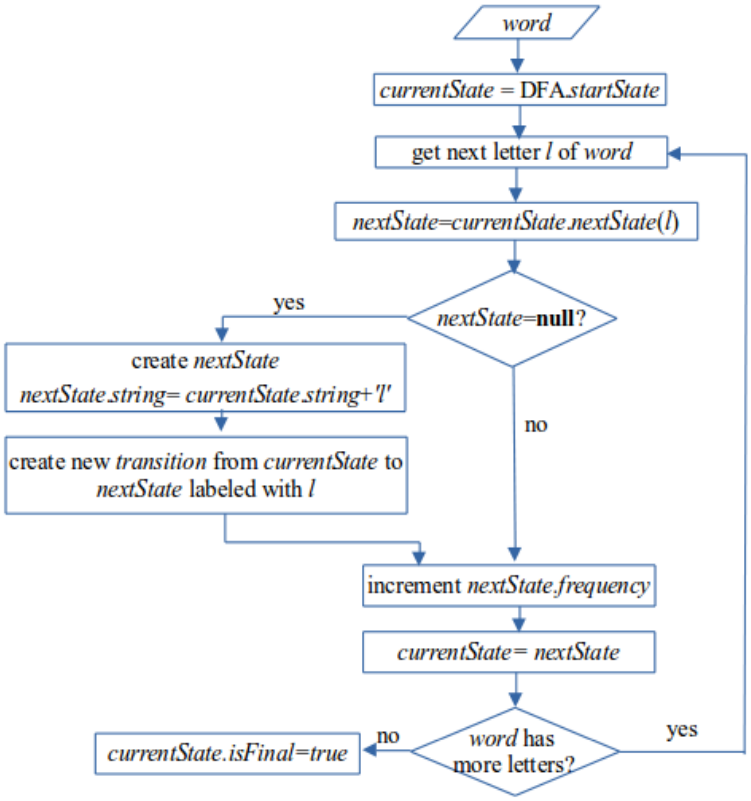


**Figure 3:** ADD-TO-DFA (*word*) Algorithm

To illustrate this algorithm with an example, let *T*=<*ab ac*>. Fig. 4 shows the DFA after adding the word *ab*. Note the appended space. The *string* and *frequency* fields are shown for all states. For example, $s_2$.*string* is "*ab*" which transfers the DFA from the start state $s_0$ to $s_2$. The *string* field of the start state ($s_0$) is the empty string. Final states are distinguished with a different color. The *frequency* fields are all set to 1 because each of these states was visited once.

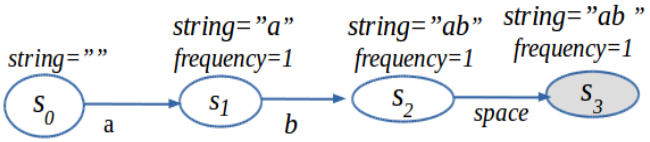


**Figure 4:** DFA after adding "*ab*"

Next word to add is "*ac*" with appended space. The *currentState* is set to $s_0$. The word length is 3. So the loop will iterate three times. In the first iteration, next letter will be '*a*'. The algorithm finds a transition from $s_0$ with '*a*'. A *nextState* is found which is $s_1$. The *frequency* field for $s_1$ is incremented and *currentState* becomes $s_1$. In the second iteration,

no transition from $s_1$ with letter '*c*' is found. So, a new state will be created which is $s_4$. The frequency field is set to *1*. Then, the space letter will be processed and $s_5$ will be created. When the loop terminates, $s_5$ is set to be a final state. The DFA will look as shown in Fig. 5.

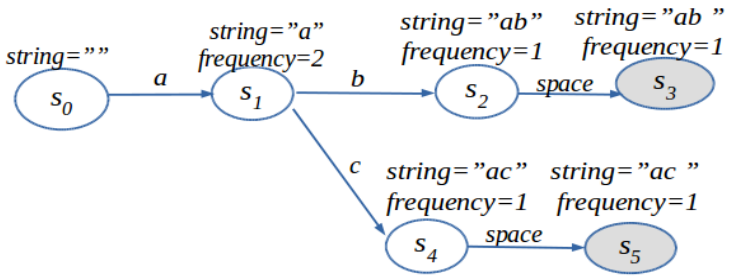


**Figure 5:** DFA after adding the word "*ac*"

### *2.2 Building frequency lists*

The second step of the main algorithm (Fig. 1) is to build the *frequencyList* field for every non-final state *s*. One frequency record is created for every transitions from *s*. This record is composed of a letter (*l*) and the frequency of the next state that is reachable from *s* with (*l*). For example, the frequency lists for the states of the DFA in Fig. 5 are shown in Tab. 2. It can be noticed that final states have no frequency lists.

**Table 2:** Frequency List for DFA states

| State | State's string | Frequency List |
|---|---|---|
| $s_0$ | "" | <(a, 2)> |
| $s_1$ | *a* | <(b, 1),(c, 1)> |
| $s_2$ | *ab* | <(space, 1)> |
| $s_4$ | *ac* | <(space, 1)> |

The algorithm that builds frequency lists (Fig. 6) goes through all the non-final states of the DFA and build the frequency list for each state. Final states have no transitions.

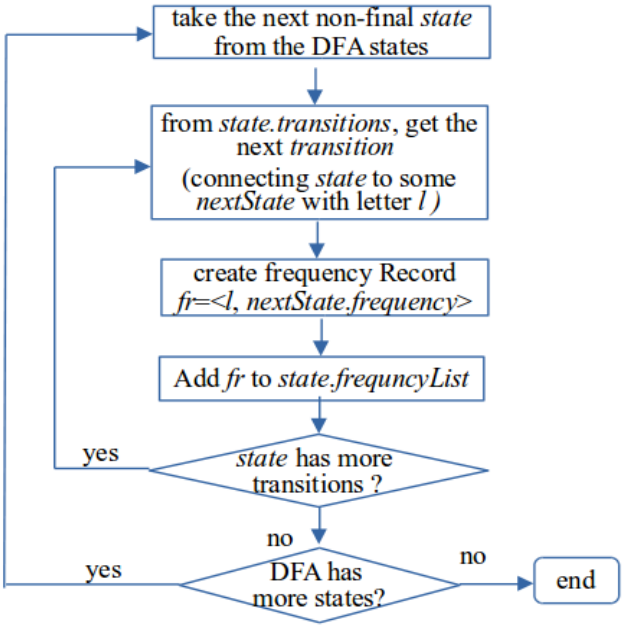


**Figure 6:** BUILD-FREQUENCY-LISTS for the DFA states

The loop iterates through the states of the DFA. For every non-final *state* of the DFA, it iterates through all the *transitions* of *state*. For every *transition,* a frequency record is created and added to the *frequency list* of *state*. This record is composed of two fields: the *letter* that labels the transition and the *frequency* of the *nextState* that is reachable through this *transition*.

### *2.3 Building a Huffman encoding for every non-final state*

The third step of the main algorithm is to define Huffman codes for every non-final *state* of the DFA (Fig. 7). The algorithm iterates through all the non-final states of the DFA. It builds *treeNodeVector* for the current *state.* This vector is the basis to build the Huffman encoding for the current *state*. The *treeNodeVector* is transformed to a Huffman tree by the BUILD-HUFFMAN-TREE algorithm. Finally, *treeNodeVector* is used to generate the encodings of the letters of *state* (the letters that label transitions from the current *state*).

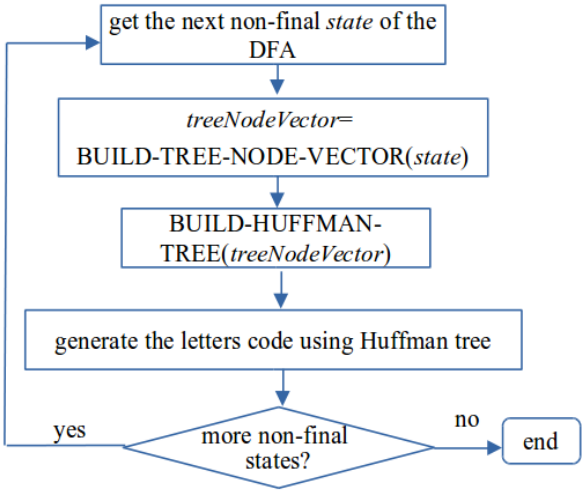


**Figure 7:** BUILD-HUFFMAN-CODES for the DFA states

This algorithm uses the *TreeNode* data structure. This data structures is needed to build the Huffman tree and its fields are shown in Tab. 3.

**Table 3:** The fields of *TreeNode t*

| Field Name | Type | Description |
|---|---|---|
| *t.leftChild* | TreeNode | Points to the left child tree node of *t.* |
| *t.rightChild* | TreeNode | Points to the right child tree node of *t.* |
| *t.frequency* | int | contains a letter frequency if *t* is a leaf or the sum of the frequencies of its direct children if it is an internal node. |
| *t.letter* | string | contains a letter if *t* is a leaf node or the concatenation of its left and right children *letter* fields if it is an internal node. |
| *t.bit* | char | The Huffman code which will be *0* or *1*. It will be *0* if *t* is a left child of its parent and *1* if *t* is a right child of its parent |

BUILD-TREE-NODE-VECTOR algorithm is shown in Fig. 8. It builds *treeNodeVector* for a *state* and this vector will become latter the Huffman tree of this *state*. Initially this vector is empty. The algorithm goes through the records in the state's frequency list. It will create *newTreeNode* for every frequency record and copy the *letter* and the *frequency* fields to *newTreeNode*. The *newTreeNode* will be inserted into the *treeNodeVector* such that *treeNodeVector* stays sorted according to the *frequency* field.

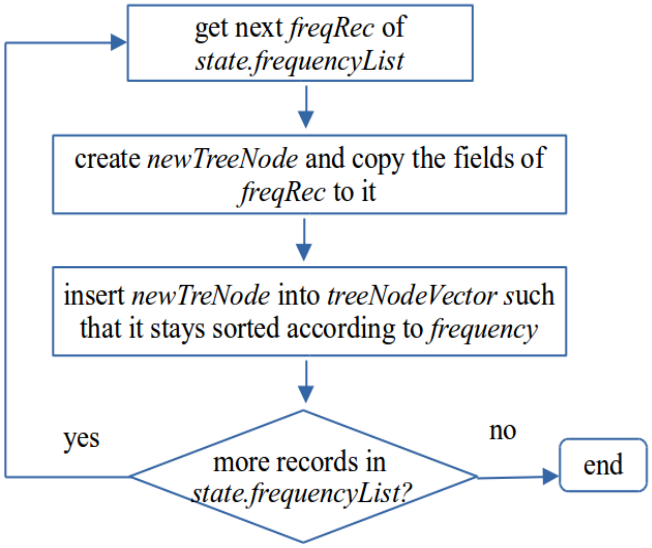


**Figure 8:** BUILD-TREE-NODE-VECTOR

Returning to the BUILD-HUFFMAN-CODES algorithm in Fig. 7. After building the *treeNodeVector* for the current non-final state, this *treeNodeVector* is transformed to a Huffman tree by BUILD-HUFFMAN-TREE algorithm (Fig. 9). The *treeNodeVector* is already sorted according to the frequency field. So, the first two tree nodes of this vector have the least *frequency* values. We will refer to them as *minTreeNode* and *nextMinTreeNode*. The *bit* field of these two tree nodes will be set to 0 and 1 respectively. A *newTreeNode* will be created with *leftChild* field is set to be *minTreeNode* and *rightChild* field is set to be *nextMinTreeNode*. The *frequency* field for the *newTreeNode* is set to be the sum of the *frequency* fields of these two nodes. They are then removed from *treeNodeVector* and *newTreeNode* is inserted such that *treeNodeVector* stays sorted. The loop will iterate until the *treeNodeVector* contains one tree node only (the root of the Huffman tree of the corresponding state).

The last step of building the Huffman codes for the states, is to define the codes for the letters with transitions from the current state (Fig. 7). This is done exactly as done in the known Huffman algorithm and will not be presented here.

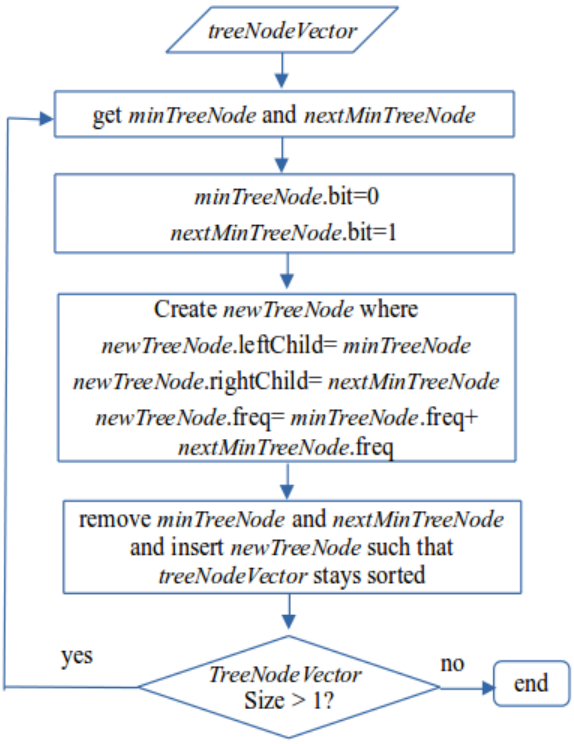


**Figure 9:** BUILD-HUFFMAN-TREE algorithm

### *2.4 Encoding the text*

The last step in the main algorithm is to encode the text using the calculated Huffman codes (Fig. 1). The ENCODE algorithm (Fig. 10) takes as input the text to compress. The output will be the code string *textCode.* Initially, *textCode* is the empty string. It goes through the words in *T* a word by word and appends a space character to the end of the current *word*. Always at the beginning of the outer loop, *currentState* is set to be the start state of the DFA. The inner loop will generate the code for *word*. It iterates through the letters of *word,* use the Huffman tree of the *currentState* to get the code of the current letter, append the code of the letter to *textCode* and finally update *currentState* to be the next state reachable from *currentState* with the current letter.

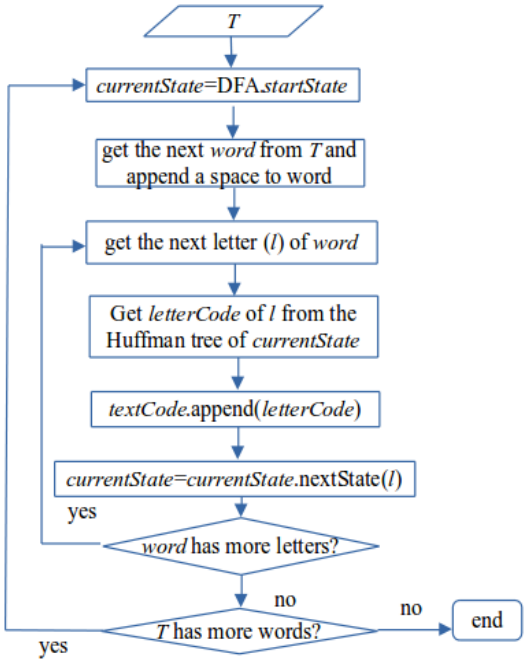


**Figure 10**: ENCODE(*T*) algorithm

### *3 Example*

Let *T*=<ab bd ac ab be bd ac>, in the following example, *T* is compressed using Huffman encoding, *T* is compressed using the proposed algorithm and the two compressions are compared.

### *3.1 Huffman encoding*

The first step is to calculate the frequency of the letters in *T*. The frequencies of the letters are: freq(a)= 4, freq(b)=5, freq(c)=2, freq(d)=2, freq(e)=1 and freq(space)=6. The space is defined as a letter and will be encoded as well.

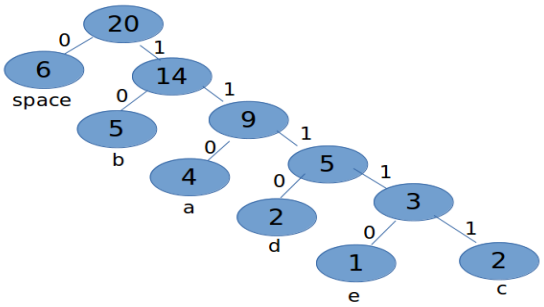


**Figure 11:** Huffman tree

The second step is to build the Huffman tree (Fig. 11). The internal nodes contain the accumulative frequencies. The Huffman encoding for the letters are defined as follows:

code(*a*)=011, code(*b*)=01, code(*c*)=11111, code(*d*)=0111, code(*e*)=01111, code(*space*)=0. So, the encoding of *T* will be 011010-0101110-011111110-011010-01011110-0101110-011111110. Hyphens are used to separate the code of the words for illustration.

The compression ratio can be calculated as follows. The size of the original file is 20 character. The size of one character is 2 bytes assuming the Unicode encoding. So the total size of the document is 40 bytes. The Huffman code length is 52 bits which equals 6.5 bytes. So the compression ratio is 16%.

### *3.2 The proposed approach*

The first step is to build a DFA that recognizes the words of *T*. It is shown in Fig. 12. The number of the final states equals the number of distinct words in *T*. The space is considered as part of the word preceding it. While building the DFA, the frequencies for every transition that was visited is recorded in frequency list of the state. The state with string field "*a*" has frequency=4 because there are four words from *T* that transfers the DFA to the state named *a* from the start state. However, the state whose string is *ab* has frequency 2 because there are two words in *T* that start with *ab.*

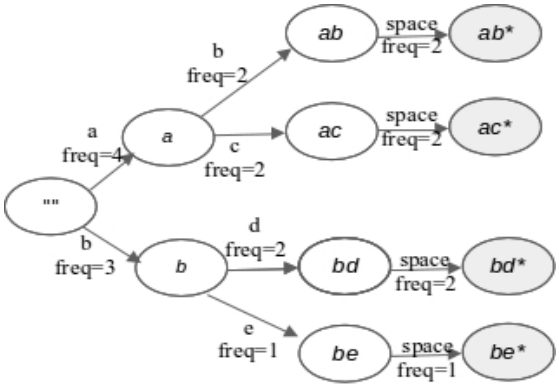


**Figure 12:** DFA that recognizes *T*=<*ab bd ac ab be bd ac*>

The second step is to build the frequency lists for every non final state. Tab. 4 shows the frequency lists for the states of the DFA.

**Table 4:** Frequency lists for the DFA states in Fig. 12

| State string | Frequency List | Letter Encoding |
|---|---|---|
| "" | < (b,3), (a,4) > | **code(b)=0** |
| | | code(a)=1 |
| *a* | < (c,2), (b,2) > | code(c)=0 |
| | | **code(b)=1** |
| *b* | < (e,1), (d,2) > | code(e)=0 |
| | | code(d)=1 |
| *ab* | < (space), 2> | code(space)=0 |
| *ac* | < (space), 2> | code(space)=0 |
| *bd* | < (space), 2> | code(space)=0 |
| *be* | < (space), 1> | code(space)=0 |

The third step is to build different Huffman encodings for the letters that label the transitions for each state. In this example, Huffman trees are composed at most of three nodes because *T* is too simple. That is why all letters are encoded with one bit only. It can be noticed that the same letter may have different encodings depending on the prefix that precedes it in the word. For example, the code of *b* when it is the first letter in the word is 0 (Tab. 4: row 1). However, its code when it is the second letter and an *a* precedes it in the word will be 1 (Tab. 4: row 2). These codes are shown in bold in Tab. 4. Using these encodings, the text is encoded as follows: 110-010-100-110-000-010-100. For example, the word *bd* is encoded as follows: the encoding of letter *b* when it is the first letter of the word is 0, the encoding of *d* when it is preceded by the prefix *b* is 1, the encoding of the space when it is preceded by *bd* is 0, so the encoding of *bd* is 010.

The compression ratio can be calculated as follows. The total size of the document is 40 bytes as explained above. The text code length is 21 bits which equals about 3 bytes. So the compression ratio is 7.5%.

## 4 Experimental study

We have implemented this algorithm and applied its compression on the text of the Quran. The size of *T* is 411082 characters, each character is two bytes (Unicode encoding). So the size of the text in bytes is 822164. The frequencies of the different Arabic letters in the text were found and then used to generate Huffman encoding. The Huffman encoding was found in a form of a string of 0's and 1's. To find the size of the compressed text in bytes, the binary encoding string length was divided by 8. The compression ratio was calculated by finding the ratio of the size of the compressed text to the size of the original text in bytes. The results are summarized in Tab. 5.

**Table 5:** Experimental results for a *T* of 822164 characters

| | **Huffman Encoding** | **Proposed Encoding** |
|---|---|---|
| **Size (bytes)** | 184,798 | 114580 |
| **Compression Rate** | 22.5% | 13.9% |
| **Compression Time (milliseconds)** | 115 | 617 |

It is obvious that the compression ratio of the proposed encoding is much better than the ratio of the normal Huffman encoding. However, we found the time needed to encode, using the proposed approach, took significantly longer time than the normal Huffman encoding time.

It is widely known that Huffman encoding gives an optimal or very close to optimal compression. Does this contradict our results? The answer is no. Huffman encoding gives the optimal compression under the assumption of a very generic context where no constraints are assumed about the letters. The only constraint is that they are randomly located in a string with varying frequencies. However, in natural language text, there are many implicit and complex constraints that are completely ignored by the Huffman encoding in its native generic form. A natural language text contains words with lengths

that follow a statistical distribution. The natural language text is not composed of randomly selected letters (from some alphabet) that are randomly placed in a string. There are many constraints regarding the length of the words and the likelihood that a letter may exist in a given location within a given sequence of surrounding letters. Our proposed approach has considered some of these constraints when using Huffman encoding and this is the reason behind these better results.

## 5 Conclusions

In this paper, a text compression algorithm is proposed that is based on Huffman encoding. Instead of having one encoding for a text letter, a letter will have different encodings depending on the prefix that precedes it in the word. The proposed approach showed significant improvement in the compression ratio compared to the normal Huffman encoding. This is because letters have varying probabilities of existence in a word after a given prefix of that word. Also some letters are unlikely to appear after a given prefix in natural language words. One important lesson we learned from this work is that considering the constraints of natural languages, it may open the door for improvements on string algorithms in general. As for future work, we look forward to optimize the implementation to give better compression time.

**Conflicts of Interest:** The authors declare that they have no conflicts of interest to report regarding the present study.

## References

**Ailenberg, M; Rotstein, O.** (2009): An improved Huffman coding method for archiving text, images, and music characters in DNA. *Biotechniques*, vol. 47, no. 3, pp. 747-754.

**Arshad, R.; Saleem, A.; Khan, D.** (2016): Performance comparison of Huffman coding and double Huffman coding. *Sixth International Conference on Innovative Computing Technology*.

**Bell, T.; Cleary, J.; Witten, I.** (1990): *Text Compression.* Prentice Hall.

**Choueka, Y.; Klein, S.; Perl, Y.** (1985): Efficient variants of Huffman codes in high level languages. *Proceedings of ACM SIGIR Conference on Research and Development in Information Retrieval*, pp. 122-130.

**Chung, K.; Wu, J.** (1999): Level-compressed Huffman decoding. *IEEE Transactions on Communications*, vol. 47, no. 10, pp. 1455-1457.

**Culik, K.; Kari, J.** (1993): Image compression using weighted finite automata. *Computer & Graphics*, vol. 17, no. 3, pp. 305-313.

**Dath, D.; Panicker, V.** (2017): Enhancing adaptive Huffman coding through word by word compression for textual data. *International Conference on Communication and Signal Processing*, pp. 1048-1051.

**Hafner, U.; Albert, J.; Frank, S.; Unger, M.** (1998): Weighted finite automata for video compression. *IEEE Journal on Selected Areas in Communications*, vol. 16, no. 1, pp. 108-119.

**Hopcroft, J.; Ullman, J.** (1979): *Introduction to Automata Theory, Languages and Computation.* Edison Wesley.

**Huffman, D.** (1952): A method for the construction of minimum redundancy codes. *Proceedings of the Institute of Radio Engineers*, vol. 40, no. 9, pp. 1098-1101.

**Javed, M.; Nadeem, M.** (2000): Data compression through adaptive Huffman coding schemes. *Proceedings of Intelligent Systems and Technologies for the New Millennium*, pp. 187-190.

**Khan, A.; Choudhury, P.; Dihidar, K.; Verma, R.** (1999): Text compression using two-dimensional cellular automata. *Computers & Mathematic with Applications*, vol. 37, no. 6, pp. 115-127.

**Klein, S.; Saadia, S.; Shapira, D.** (2019): Better than optimal Huffman coding? *Proceedings of Data Compression Conference*, pp. 582-582.

**Knuth, D.** (1985): Dynamic Huffman coding. *Journal of Algorithms*, vol. 6, no. 2, pp. 163-180.

**Moffat, A.** (2019): Huffman coding. *ACM Computing Surveys*, vol. 52, no. 4, pp. 1-35.

**Salomon, D.** (2007): *Data Compression.* Springer, London.

**Saravanan, C.; Surender, M.** (2013): Enhancing efficiency of Huffman coding using Lempel Ziv Coding for image compression. *International Journal of Soft Computing and Engineering*, vol. 2, no. 6, pp. 38-41.

**Shanmugasundaram, S.; Lourdusamy, R.** (2011): A comparative study of text compression algorithms. *International Journal of Wisdom Based Computing*, vol. 1, no. 3, pp. 68-76.

**Tseng, K.; Jiang, J.; Pan, J.; Tang, L.; Hsu, C. et al** (2012): Enhanced Huffman coding with encryption for wireless data broadcasting system. *Proceedings of International Symposium on Computer, Consumer and Control*. pp. 622-625.

**Vitter, J.** (1987): Design and analysis of dynamic Huffman codes. *Journal of the ACM*, vol. 34, no. 4, pp. 825-845.