Article

Not peer-reviewed version

On Move with Interleaving (MwI) Implementation

Borut Žalik * , Aljaž Jeromel , Ivana Kolingerová , Niko Lukač , Blaž Repnik

Posted Date: 19 January 2024

doi: 10.20944/preprints202401.1432.v1

Keywords: string transformation; lookup table; efficiency; Java; C++

Preprints.org is a free multidiscipline platform providing preprint service that is dedicated to making early versions of research outputs permanently available and citable. Preprints posted at Preprints.org appear in Web of Science, Crossref, Google Scholar, Scilit, Europe PMC.

Copyright: This is an open access article distributed under the Creative Commons Attribution License which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 19 January 2024 doi:10.20944/preprints202401.1432.v1

Disclaimer/Publisher's Note: The statements, opinions, and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions, or products referred to in the content.

Article

On Move with Interleaving (MwI) Implementation

Borut Žalik [1,*], Aljaž Jeromel [1], Ivana Kolingerová [2], Niko Lukač [1], Blaž Repnik [1]

[1] University of Maribor, Faculty of Electrical Engineering and Computer Science, Koroška cesta 46, SI-2000 Maribor, Slovenia; aljaz.jeromel@um.si (A.J.); niko.lukac@um.si (N.L.); blaz.repnik@um.si (B.R.);
[2] University of West Bohemia, Department of Computer Science and Engineering, Technická 8, 306 14 Plzeň, Czech Republic; kolinger@kiv.zcu.cz (I.K.)
* Correspondence: borut.zalik@um.si (B.Ž.)

Abstract: Various implementations of the Move with Interleaving transform are discussed in this paper, with the transformation itself explained briefly at first. The transform has an expected time complexity of $O(n)$, where n represents the length of the sequence being transformed. The importance of implementation also grows as the sequences become longer. The paper explains three different implementations: the first uses the standard vector, the second employs a lookup table, and the third utilises a dynamic linked list with a pool of released records. The experiments were conducted on 32 greyscale raster images of various contexts and sizes. All three solutions were implemented in both C++ and Java, and then executed on three different platforms: a personal computer running MS Windows and Linux, and a Raspberry Pi 2 with Linux. The implementation based on the dynamic linked list was the fastest, and outperformed the vector-based implementation by more than 1800%. While direct comparisons between results on the Raspberry Pi and those on the modern personal computer may not be straightforward, a consistent pattern prevails: the dynamic linked list implementation outperformed the other two consistently, and the C++ code was faster than the code written in Java. The difference was most evident on the Raspberry Pi.

Keywords: string transformation; lookup table; efficiency, C++; Java.

1. Introduction

Careful implementation of algorithms was a strong focus in the past due to considerably limited computer resources [1–4]. Today, however, there is an illusion among average programmers that this problem is no longer relevant. The compilers generate efficient executable code and offer additional optimisation features [5,6]. Better personal computers are equipped today with gigabytes of on-board memory, and their CPUs can perform calculations at a rate of hundreds of megaflops [7]. However, with better computers, even more data are produced everyday [8]. Supercomputers [9], GPGPU installations [10] and cloud-based solutions [11] have been developed for processing them. Unfortunately, the majority of users do not have access to such installations; indeed, they operate modest personal computers, processing manageable amounts of data for which they expect to receive results quickly. This is why even a short segment of programming code, executed frequently, should be understood, not only arithmetically, but also from an implementation perspective. Programming code could be optimised in various aspects, such as, for example, in terms of spent CPU time [12], required memory [13], or energy consumption [14]. An approach to reduce the spent CPU time of a recently proposed string transformation technique named *Move with Interleaving* [15] is considered in this paper. This transformation reduces the information entropy [16,17] as efficiently as the combination of the Burrows-Wheeler transform [18–20] followed by Move-To-Front transform [21,22] or Inversion Frequencies transform [23]. Move with Interleaving can, therefore, be used as a replacement for these transforms in the preprocessing stage of data compression algorithms [22,24], reducing the length of the programming code. Although its theoretical time complexity is linear with respect to the length of the sequence to be transformed, it requires careful implementation to achieve an acceptable spent CPU time. The main contributions of the paper are as follows:

© 2024 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 19 January 2024 doi:10.20944/preprints202401.1432.v1

- Highlighting the awareness of computer scientists, and especially programmers, that better computers do not solve the problems of shortening CPU time without careful implementation;
- Providing information about CPU spent time using two popular programming languages, C++ and Java;
- Testing the efficiency of the implementations on various platforms using different compilers.

The paper is organised in five Sections. Move with Interleaving is, together with the Move-To-Front transform, introduced briefly in Section 2. Three various implementations of Move with Interleaving are described in Section 3. The experimental results are given in Section 4, while the last, fifth Section contains the conclusion.

2. Background

2.1. Move-To-Front Transform

Move-To-Front (MTF) is one of the self-organising list strategies [21,25–27] used in caching algorithms [28] and data compression [29,30]. Let $X = \langle x_i \rangle$, $0 \leq i < n-1$, represent the input sequence of n elements x_i from an alphabet Σ_X, where Σ_X consists of m elements. MTF reads the elements $x_i \in X$ sequentially, and maps them to the domain of natural numbers, including 0, to obtain the output sequence $Y = \langle y_i \rangle$, $0 \leq i < n$, $y_i \in \Sigma_Y = \{0, 1, 2, \cdots, m-1\}$. The list L is employed in this mapping process and it is populated with all the elements from Σ_X initially. The algorithm finds the position l, $0 \leq l < m-1$, in L, where $x_i = L_l$ and sends l to Y. After that, the positions of all the elements L_e in L between $0 \leq e < l$ are increased, while x_i is placed in front of L. The pseudocode for the MTF transform is shown in Algorithm 1. The main loop contains only three functions. The first one, on Line 7, iterates through the elements in L and stops when $L_l = x_i$. It is expected that x_i is found in the constant time $O(1)$, since MTF is a self-organising list. The function in Line 8 terminates in $O(1)$, as it just writes l to an array representing Y. The most time-consuming task is performed by the last function in Line 9, which needs to replace the first $l-1$ values in L. It can be realised in various ways. Four different implementations were proposed and analysed in [31].

MTF is used most frequently in data compression [26,27,29], as it reduces the information entropy of X when X contains local correlations. For example, a sequence of the same elements is converted into a sequence of zeros; a sequence of alternating elements gives a sequence of ones; a sequence of alternating triples results in a sequence of twos, and so on. This is the reason why MTF is used typically after BWT, which tends to group the same elements close together [19,20].

Algorithm 1 Pseudocode of MTF transform

```
1:  function MTF(X, n, Σ_X, m)             ▷ returns the transformed sequence Y
2:                                          ▷ X: input sequence with length n
3:                                          ▷ Σ_X: alphabet with length m
4:      i = 0
5:      L = Initialise(Σ_X, m)              ▷ insert all elements from Σ_X to L
6:      for i ← 0 to n do
7:          l = GetPosition(L, x_i)
8:          Y = AddToY(Y, l)
9:          L = MoveUp(L, l)    ▷ increase the positions of the elements in L between 0 and l
10:         L_0 = x_i                       ▷ move x_i to the front
11:     end for
12:     return Y                            ▷ return the transformed sequence
13: end function
```

2.2. Move with Interleaving

The *Burrows-Wheeler transform, (BWT)* should be used before MTF to enhance the local correlations of X, as mentioned in the previous Subsection. BWT was introduced in 1994 [18], and is a well-studied transform [22]. Unfortunately, it has $O(n^2 \log(n))$ time complexity, which is why X should be split into small blocks to achieve a fast enough response [18]. A direct correlation between suffix array and the BWT was determined later [19]. Straightforward implementation of the suffix array also works,

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 19 January 2024 doi:10.20944/preprints202401.1432.v1

unfortunately, in $O(n^2 \log(n))$ [32,33]. Fortunately, more algorithms were developed for constructing a suffix array of X in $O(n)$ [34–36]. This also enables obtaining the BWT in linear time. However, the implementation of these algorithms is relatively demanding. This is why it would be desirable to replace the pipeline of the three algorithms – an algorithm for constructing the suffix array, an algorithm to generate BWT from the suffix array, and the MTF algorithm – with only one compact algorithm. It was shown recently that, at least for the domain of Raster Images, this can be done by the transform named *Move with Interleaving (MwI)* [15]. Pixels from a continuous-tone image are arranged into a sequence X at first. Pixels in the close neighbourhood of the considered pixel are, in the majority of cases, similar, but not completely the same. However, this similarity is sufficient for MTF to produce small indices y_i. But, when a region of different pixel values is encountered, MTF has to generate many large indices y_i before bringing enough values from this region close to the front of L. This has, unfortunately, a negative effect on the reduction of the information entropy, and this is why BWT should be applied before MTF. Because of this, MwI monitors the value $y_i = MTF(x_i)$, and reacts when y_i exceeds the given threshold t. MwI assumes that an area with different values is found and that the values similar to x_i will follow. Because of this, it inserts, in addition to x_i, $2t$ interleaved values around x_i in front of L, i.e., the values from a generated auxiliary sequence $A = \langle x_i, x_i + 1, x_i - 1, x_i + 2, x_i - 2, \cdots, x_i + t, x_i - t \rangle$.

The MwI pseudocode is shown in Algorithm 2. The algorithm first populates the list L in Line 5. It uses the first element from X and interleaves it with $2t$ values. These values are inserted in front of L, and the remaining values from Σ_X are filled in alphabetical order. The first element x_0 is stored immediately in the output Y in Line 6, as the decoder should perform the same initialisation of L. The algorithm then enters the main loop and processes $X = \langle x_i \rangle$, $1 \le i < n$, sequentially. The position l of x_i is determined in L (Line 8). It checks after that whether l is within the given threshold t. If so, the MTF procedure is performed in Lines 11 and 12; otherwise, the algorithm fills the auxiliary sequence A with interleaved elements around x_i in Line 14. The elements from A are then inserted in L in the MTF-style within Lines 16 and 17.

Algorithm 2 Pseudocode of MwI transform

```
1:  function MwI(X, n, Σ, m, t)                  ▷ returns the transformed sequence Y
2:                                                ▷ X: input sequence with length n
3:                                                ▷ Σ: alphabet with length m
4:                                                ▷ t: threshold
5:      L = InitialiseL(X, n, Σ, m, t)            ▷ initialisation of L according to x_{i=0}
6:      Y_0 = x_0                                 ▷ store x_{i=0}
7:      for i ← 1 to n do                         ▷ for all other x_i
8:          l = GetPosition(L, x_i)               ▷ find the position of x_i in L
9:          Y = AddToY(Y, l)                      ▷ store the position in Y
10:         if l ≤ t then                         ▷ if the position is smaller than t
11:             L = MoveUp(L, l)                  ▷ perform MTF
12:             L_0 = x_i
13:         else                                  ▷ otherwise perform MwI
14:             A = FillA(x_i, t)                 ▷ fill temporary sequence T
15:             for k ← (2 × t) downto 0 do       ▷ execute MTF 1 + 2 × t times
16:                 L = MoveUp(L, A_k)
17:                 L_0 = A_k
18:             end for
19:         end if
20:     end for
21:     return Y                                  ▷ returns the transformed sequence
22: end function
```

Finally, let us analyse the MwI time complexity. $l > t$ in each iteration for the worst-case scenario. Therefore, for each x_i, $1 \le i < n$, the algorithm fills A in time $T_A = 2t + 1$, and then performs the MTF $2t + 1$ times. MTF has the expected time complexity $O(n)$, and, as a result, the expected execution time of MwI is $T_{MwI} = (2t + 1)O(n)$. Since $t \le m << n$, it can be concluded that MwI's time complexity is the same as MTF, i.e., $O(n)$.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 19 January 2024 doi:10.20944/preprints202401.1432.v1

3. Materials and Methods

Three different implementations of MwI, denoted as **I-1**, **I-2**, and **I-3**, are considered in this Section. The programming code is presented in C++ [37], with a note that the paper does not make use of its object-oriented features.

3.1. *Implementation* I-1

List L is represented as the standard C++ vector in this implementation. Algorithm 3 shows the function that initialises L. This initialisation assumes that the alphabet represents the numbers from the range $[lBound, uBound]$. The last parameter N in function InitL_I controls the number of elements to be inserted. It is set to $uBound - lBound + 1$ initially. Variables u and l hold the incremental interleaving values around FirstEl.

Algorithm 3 Implementation **I-1**: initialization of L

```
1   vector<int> InitL_I(int lBound, int uBound, int FirstEl, int N)
2                      ▷ lBound, uBound: the values of the first and the last element from Σ
3                                              ▷ FirstEl: the value of the first element x0
4                                              ▷ N: the number of elements to be inserted
5   {
6     vector<int> L(N+1);
7     L[0] = FirstEl;                          ▷ FirstEl becomes the first element of L
8     int u = FirstEl + 1;                     ▷ incremental interleaving value
9     int l = FirstEl − 1;                     ▷ decremental interleaving value
10    int counter = 0;
11    int pos = 1;
12    while (counter++ < N/2) {
13      if (u <= uBound)
14        L[pos++] = u++;
15      if (l >= lBound)
16        L[pos++] = l−−;
17    }
18    return L;
19  }
```

The main function of implementation **I-1** is shown in Algorithm 4. The position l of each x_i in L is determined in Line 9 and then stored in Line 10. After that, l is tested (Line 11): if it is smaller than t, the MTF is executed in Lines 12-14; otherwise, more elements interleaved around x_i should be moved-to-front of L. The auxiliary vector A is filled with at most $1 + 2t$ interleaved values by calling the function InitL_I in Line 17. In Algorithm 3, this time the parameter N gets the value 2t. After that, the content of L is updated in Line 18.

Finally, let us explain the updating procedure for L, when more elements should be moved to the front (see Algorithm 5). In Line 4, $1 + 2t$ interleaved values from vector A are first copied to a temporary vector Temp. The remaining values of L are added incrementally within the Lines 6– 17 to Temp if they have not yet been added in Line 4. When the for-loop terminates, vector Temp contains the updated status of L, which is then returned in Line 18 for use in the next iteration of Algorithm 4.

3.2. *Implementation* I-2

In implementation **I-2**, the data structure used to store L is once again a standard vector. Consequently, the initialisation process remains identical to that given in Algorithm 3. The primary drawback of implementation **I-1** is in the update function outlined in Algorithm 5, which contains two nested loops. The first loop iterates through all the elements of L. The nested **while** loop scans the auxiliary vector A to verify whether the element L[i] also exists in A. If not, the element is added to the temporary vector Temp. Although the UpdateL function is not called for each element being

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 19 January 2024 doi:10.20944/preprints202401.1432.v1

Algorithm 4 Implementation **I-1**: MwI main function

```
1   vector<int> MwI_I(vector<int>& X, int t, int lBound, int uBound, int vector<int>& L)
2   {
3     int i, j, l;
4     vector<int> A;
5     vector<int> Y(X.size());                           ▷ allocate space for Y
6     Y[0] = X[0];                                        ▷ first element in Y is X[0]
7     for (i = 1; i < (int)X.size(); i++){                ▷ for all elements of X except the first
8       l = 0;
9       while (X[i] != L[l]) l++;                         ▷ find the position l of x_i in L
10      Y[i] = l;                                         ▷ store position l in the result
11      if (l <= t) {
12        for (j = l − 1; j >= 0; j−−)                    ▷ move-to-front
13          L[j + 1] = L[j];
14        L[0] = X[i];
15      }
16      else {                                            ▷ move-to-front more elements
17        A = InitL_I(lBound, uBound, X[i], 2 ∗ t);       ▷ fill the auxiliary array
18        L = UpdateL(A, L);                              ▷ update L with elements from A
19      }
20    }
21    return Y;
22  }
```

Algorithm 5 Implementation **I-1**: L update function

```
1   vector<int> UpdateL(vector<int>& A, vector<int>& L)
2   {
3     int NotExistInA, i, j;
4     vector<int> Temp = A;                ▷ the first elements of Temp are equal to those from A
5     Temp.reserve(L.size());                          ▷ allocate space for all elements from L
6     for (i = 0; i < (int)L.size(); i++) {            ▷ for all elements of the alphabet
7       NotExistInA = 1;
8       j = 0;
9       while (NotExistInA && (j < (int)A.size())) {
10        if (L[i] == A[j])                            ▷ check if an element from L exists in A
11          NotExistInA = 0;                           ▷ mark that it exists
12        else
13          j++;                                       ▷ otherwise check the next element from A
14      }
15      if (NotExistInA)
16        Temp.push_back(L[i]);                        ▷ if the element does not exist, add it
17    }
18    return Temp;                                     ▷ Temp becomes the new L
19  }
```

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 19 January 2024 doi:10.20944/preprints202401.1432.v1

transformed, and the length of the vector A is expected to be small, $1 + 2t << m$, it opens up room for an improvement. Namely, the nested **while** loop can be omitted by introducing a lookup table LUT. Its purpose is similar to the lookup table in the Counting-sort algorithm [39]. In our case, LUT consists of m flags set to FALSE initially. The flags in LUT are set during the creation of the auxiliary array A, i.e. each flag is raised at the position LUT_{A_i}. During the updating of L, for each element in L it checks the status of the flag at the location LUT_{L_i}. The element is added into the temporary vector if the flag is FALSE. Details, including the construction of vector A and the updating of L, are presented in Algorithm 6. An array of flags LUT is allocated in Line 5. It consists of m elements, i.e. the size of

Algorithm 6 Implementation **I-2**: MwI with LUT

```
1   vector<int> MwIwithLUT(vector<int>& X, int t, int lBound, int uBound, int vector<int>& L)
2   {
3     vector<int> Y(X.size());                                  ▷ allocate space for Y
4     int LUTSize = (int)L.size();
5     vector<int> LUT(LUTSize);                                 ▷ allocate space for LUT
6     Y[0] = X[0];                                              ▷ the first element in Y is X[0]
7     for (int i = 1; i < (int)X.size(); i++){                  ▷ for all elements of X ecept the first
8       int l = 0;
9       while (X[i] != L[l]) l++;                               ▷ find the position l of x_i in L
10      Y[i] = l;                                               ▷ store position l in the result
11      if (l <= t) {
12        for (j = l − 1; j >= 0; j−−)                          ▷ move-to-front
13          L[j + 1] = L[j];
14        L[0] = X[i];
15      }
16      else {                  ▷ construct interleaved elements around L[l] and store them into A
17        vector<int> A(L.size());
18        A[0] = L[l];                                          ▷ the first element in A is L[l]
19        fill(LUT.begin(), LUT.end(), 0);                      ▷ set all flags in LUT to 0
20        LUT[l] = 1;                                           ▷ set the flag in LUT
21        int uv = L[l] + 1;
22        int lv = L[l] − 1;
23        int c = 1;
24        for (int j = 0; j < t; j++) {                         ▷ insert at most 2t elements in A
25          if (uv <= uBound) {
26            LUT[uv] = 1;                                      ▷ set the flag in LUT
27            A[c++] = uv++;
28          }
29          if (lv >= lBound) {
30            LUT[lv] = 1;                                      ▷ set the flag in LUT
31            A[c++] = lv−−;
32          }
33        }
34        for (int k = 0; k < LUTSize; k++)                     ▷ for all elements of the alphabet
35          if (!LUT[L[k]])                                     ▷ If the flag in LUT is not set, add an element
36            A[c++] = L[k];
37        L = A;                                                ▷ A is copied to the new L
38      }
39    }
40    return Y;
41  }
```

the alphabet Σ_X. In this implementation it is assumed that the alphabet Σ contains elements from the interval $[0, m-1]$. If this is not the case, a simple mapping of the alphabet's element into this interval should be done before calling the function MwIwithLUT. When the position l of the considered element $x_i > t$, all flags in LUT are set to FALSE in Line 19. The flag belonging to l is set to TRUE after that in Line 20. The same is applied for all the remaining interleaved values in Lines 26 and 30. The update of

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 19 January 2024 doi:10.20944/preprints202401.1432.v1

L is started in Line 34 after A has been prepared. The **for** loop operates for all elements of Σ_X. For each element of L, it is checked whether its flag in LUT is set (Line 35). If it is not, the considered element is inserted into A. Finally, A is copied into the new L in Line 37.

3.3. *Implementation* **I-3**

Implementation **I-3** utilises a dynamically linked list of records to represent L. Each record has only two components: el, containing the value, and a pointer next that points to the subsequent record. Concerning MTF, it seems that this data structure could offer benefits, as there is no requirement to move the elements at positions 0 through $l-1$.

Let us consider an example of L shown in Figure 1a, containing 10 elements. Suppose the element that should be moved to the front of L has the value 7. The record containing this value has already been found and is pointed to by pointer w; its preceding pointer is denoted as wPrev. The new status of L is established in constant time by repointing just three pointers, as illustrated in Figure 1b. The details are provided in Algorithm 7.

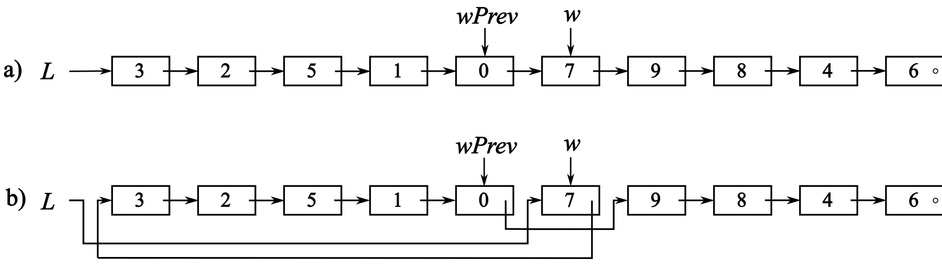

Figure 1. L reconfiguration when a dynamic linked list is used.

Algorithm 7 Implementation **I-3**: MTF

```
1   ListType* UpdateMTF(ListType* L, int x, int& l)
2   {              ▷ x should be moved to the front of L; function returns position l and updated L
3     if (L−>el == x) {                                   ▷ x is already on the top
4       l = 0;                                            ▷ set its position
5       return L;                                         ▷ return the existing L
6     }
7     ListType *w, *wPrev;
8     w = wPrev = L;
9     l = 0;
10    while (w−>el != x) {                                ▷ work until x is not found in L
11      l++;                                              ▷ increment the position
12      wPrev = w;                                        ▷ move pointers for one record
13      w = w−>next;
14    }
15    wPrev−>next = w−>next;                              ▷ relink the pointers in L
16    w−>next = L;
17    L = w;
18    return L;                                           ▷ return new L
19  }
```

The concept of how the MwI transformation can be realised using a dynamic linked list is shown in Figure 2. Let us suppose that $x_i = 7$ is again the considered value in L, which is shown in 2a. In addition, let us suppose that $t = 2$. The position of $x_i = 7$ is at $l = 5$, and as $l > t$, the auxiliary list A of $1 + 2t$ values should be appended in front of L. The auxiliary list A is shown in Figure 2b. L is inspected, and all its records holding values within the range $[x_i - t, x_i + t]$ are omitted (see Figure 2c). Finally, L and A are linked together using the pointer $ATail$; the result is shown in Figure 2d. We can observe that $1 + 2t$ new records have been created, and an equal number of them have been deleted.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 19 January 2024 doi:10.20944/preprints202401.1432.v1

Memory allocation and release are, unfortunately, computationally expensive operations, and should be avoided whenever possible [38]. Therefore, records that should be released, are placed in a stack named *Pool* in the continuation. Whenever a new record is needed during the process of constructing A, it is taken from the *Pool*. Details of the implementation can be found in Algorithm 8. In Line 8, the function PreparePool inserts $1 + 2t$ dynamically allocated records into the stack pointed to by Pool. Function GetPosition (Line 10) performs the same tasks as the programming code in Lines 8 to 14 of Algorithm 7. After that, the records containing the values from the interval $[x_i - t, x_i + t]$ are inserted into Pool by the function in Line 14. The function ConstructA (Line 15) constructs list A with interleaved values around the value x_i using the records from the Pool. It returns pointers A and $ATail$. In Lines 16 and 17, lists L and A are merged into a new list L. In Lines 20, 21, and 22, the record is moved in front as explained earlier.
All implementations are accessible at [40].

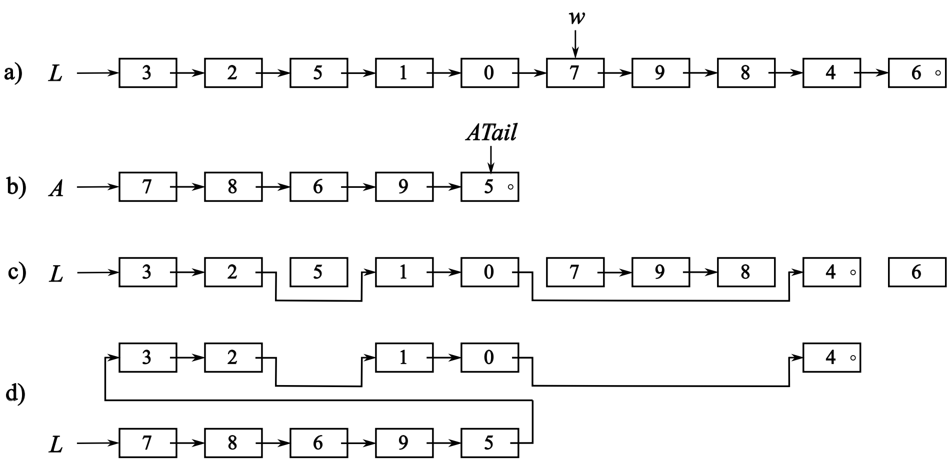

Figure 2. L reconfiguration when a dynamic linked list is used.

4. Experiments

All three proposed implementations of MwI were tested on 32 benchmark 8-bits greyscale images. The images can be seen in [15], and, in this paper, they are denoted in the same way. The values of pixels were arranged into the sequence X in the scan-line order. All three implementations were coded in both C++ and Java, as the most popular programming languages, and they were executed on three different platforms:

P1: A personal computer with AMD Ryzen 9 7950X3D 16-Core Processor clocked at 4.20 GHz, equipped with 64 GB of RAM, running the Windows 11 operating system. The code was compiled using MS Visual Studio, Version 17.4.2. OpenJDK-8-JRE was used for Java.

P2: The same personal computer as in P1, but using Linux Kernel 6.6.8. with the Manjaro distribution. The code was compiled with G++ compiler Version 13.2.1. for C++. OpenJDK-8-JRE was used for Java again.

P3: The Raspberry Pi 2 Model B running Raspbian GNU/Linux 11 (bullseye). The C++ code was compiled with gcc version 10.2.1 20210110 (Raspbian 10.2.1-6+rpi1), while OpenJDK-8-JRE was used for Java.

Table 1 illustrates the CPU time spent for all three implementations across the three platforms for code written in C++.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 19 January 2024 doi:10.20944/preprints202401.1432.v1

Algorithm 8 Implementation **I-3**: MwI

```
1  vector<int> MwIwithList(vector<int>& X, int t, int lBound, int uBound, ListType* L)
2  {                          ▷ parameters X, t, lBound, uBound are the same as in Algorithm 4
3          ▷ L is the pointer to the elements of Σ_X with interleaved values in regards to X[0]
4    ListType *A, *ATail, *w, *wPrev, *Pool;
5    int i, l;
6    vector<int> Y(X.size());                                    ▷ allocate space for Y
7    Y[0] = X[0];                                           ▷ the first element in Y is X[0]
8    Pool = PreparePool(t);                                      ▷ allocate 1+2t records
9    for (i = 1; i < (int)X.size(); i++){                ▷ for all elements of X except the first
10     l = GetPosition(X[i], &wPrev, &w);         ▷ return l and get *w and predecessor *wPrev
11     Y[i] = l;                                                               ▷ store l
12     if (l > 0) {                                               ▷ if l==0, nothing to do
13       if (l > t) {
14         L = RemoveFromL(X[i], t, L, &Pool)     ▷ Records from [X[i]-t, X[i]+t] are sent to Pool
15         ConstructA(X[i], &Pool, &A, &ATail);         ▷ A consists of records from the Pool
16         ATail->next = L;                                                ▷ Link L and A
17         L = A;                                              ▷ set L on the top of the list
18       }
19       else {                                                ▷ MTF part of rearranging L
20         wPrev->next = w->next;
21         w->next = L;
22         L = w;
23       }
24     }
25   }
26   return Y;                                                            ▷ return new L
27 }
```

Table 1. CPU time spent [s] when the C++ programming language was used.

X	n	P1			P2			P3		
		I-1	**I-2**	**I-3**	**I-1**	**I-2**	**I-3**	**I-1**	**I-2**	**I-3**
(1)	262,144	0.454	0.081	0.027	0.435	0.058	0.027	7.382	1.753	0.373
(2)	414,720	0.177	0.033	0.013	0.171	0.025	0.013	2.907	0.726	0.177
(3)	414,720	0.528	0.095	0.036	0.505	0.068	0.033	8.735	2.160	0.447
(4)	262,144	0.392	0.068	0.030	0.370	0.048	0.027	6.420	1.514	0.356
(5)	262,144	0.504	0.091	0.038	0.479	0.063	0.035	8.262	2.053	0.467
(6)	414,720	0.196	0.037	0.015	0.189	0.027	0.014	3.210	0.816	0.200
(7)	414,720	0.289	0.055	0.019	0.276	0.038	0.018	4.744	1.240	0.265
(8)	65,536	0.065	0.012	0.005	0.060	0.008	0.004	1.041	0.256	0.057
(9)	262,144	0.235	0.046	0.017	0.221	0.031	0.016	3.751	0.888	0.217
(10)	245,760	0.162	0.031	0.011	0.153	0.020	0.009	2.614	0.638	0.139
(11)	181,000	0.205	0.036	0.014	0.195	0.024	0.012	3.328	0.764	0.162
(12)	245,760	0.177	0.034	0.011	0.167	0.022	0.009	2.861	0.711	0.142
(13)	414,720	0.339	0.064	0.020	0.321	0.045	0.019	5.551	1.407	0.274
(14)	414,720	0.387	0.070	0.021	0.368	0.049	0.020	6.355	1.576	0.289
(15)	262,144	0.214	0.040	0.014	0.202	0.027	0.013	3.469	0.881	0.189
(16)	1,745,280	1.970	0.342	0.105	1.880	0.240	0.100	32.338	7.732	1.387
(17)	5,474,148	1.801	0.337	0.114	1.715	0.245	0.107	29.313	7.425	1.534
(18)	1,048,576	1.210	0.211	0.066	1.150	0.144	0.060	19.654	4.615	0.805
(19)	995,520	0.474	0.086	0.030	0.454	0.060	0.026	7.717	1.830	0.356
(20)	387,228	0.268	0.047	0.014	0.256	0.032	0.011	4.371	1.022	0.177
(21)	393,216	0.261	0.048	0.019	0.249	0.034	0.016	4.260	1.057	0.222
(22)	262,144	0.010	0.004	0.001	0.008	0.001	0.001	0.162	0.059	0.027
(23)	154,401	0.200	0.037	0.016	0.192	0.024	0.014	3.263	0.771	0.191
(24)	393,216	0.235	0.041	0.015	0.225	0.029	0.013	3.826	0.921	0.181
(25)	245,760	0.228	0.041	0.015	0.216	0.028	0.013	3.721	0.910	0.188
(26)	262,144	0.238	0.044	0.017	0.227	0.030	0.014	3.865	0.952	0.195
(27)	2,073,600	1.959	0.351	0.138	1.872	0.236	0.161	31.632	7.489	1.707
(28)	393,216	0.273	0.049	0.017	0.264	0.033	0.015	4.445	1.070	0.208
(29)	4,271,400	1.507	0.286	0.091	1.450	0.204	0.084	24.726	6.070	1.204
(30)	17,448,000	12.131	2.091	0.481	11.802	1.477	0.457	201.470	47.227	6.975
(31)	245,760	0.188	0.035	0.015	0.180	0.024	0.013	3.052	0.750	0.169
(32)	414,720	0.266	0.053	0.017	0.252	0.035	0.015	4.339	1.137	0.216
Sum		27.543	4.896	1.462	26.504	3.429	1.389	452.784	108.420	19.496

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 19 January 2024 doi:10.20944/preprints202401.1432.v1

The results were indeed stunning. Implementation **I-2** was 562% faster than **I-1**. However, Implementation **I-3** surpassed **I-2** by 334% and outperformed **I-1** by an astonishing 1883% on platform P1. Similarly, on platform P2, **I-3** was 1908% faster than **I-1** and surpassed **I-2** by 246%. Interestingly, despite using the same computer, platform P2 outperformed platform P1 consistently. While **I-1** and **I-3** differed by 4% and 5%, respectively, platform P2 surpassed P1 for **I-2** significantly by 42%. The pattern repeated on platform P3, but with a more pronounced effect. **I-3** was, this time, faster than **I-1** by more than 2300%, and by more than 550% from **I-2**. However, in this case, the absolute spent CPU time also becomes important. The user should wait for 7 and a half minutes to transform all 32 images using implementation **I-1**, while **I-3** requires only 19.5 seconds.

The results for the Java implementation are provided in Table 2. Implementation **I-3** again surpassed **I-1** and **I-2** by more than 1400% and 680%, respectively, on platform P1. Similarly, on platform P2, **I-3** outperformed **I-1** by over 1270% and **I-2** by 580%. Platform P2 was faster than platform P1 for **I-1** and **I-2**, but not for **I-3**. On platform P3, implementation **I-3** outperformed **I-1** by more than 6700%, and **I-2** by almost 1000 %.

Table 2. CPU time spent [s] when the Java programming language was used.

X	n	P1			P2			P3		
		I-1	**I-2**	**I-3**	**I-1**	**I-2**	**I-3**	**I-1**	**I-2**	**I-3**
(1)	262,144	0.659	0.739	0.071	0.696	0.358	0.079	103.426	15.059	1.397
(2)	414,720	0.242	0.284	0.029	0.259	0.138	0.038	39.905	6.434	1.001
(3)	414,720	0.672	0.791	0.062	0.687	0.333	0.061	115.944	17.137	1.500
(4)	262,144	0.506	0.526	0.054	0.497	0.230	0.051	85.874	12.795	1.173
(5)	262,144	0.678	0.327	0.058	0.648	0.304	0.062	110.942	16.434	1.768
(6)	414,720	0.255	0.114	0.029	0.259	0.120	0.025	43.032	6.766	0.809
(7)	414,720	0.377	0.170	0.038	0.382	0.178	0.034	62.491	9.475	1.044
(8)	65,536	0.087	0.032	0.010	0.083	0.037	0.008	14.074	2.111	0.222
(9)	262,144	0.286	0.116	0.014	0.303	0.141	0.029	50.448	7.548	0.792
(10)	245,760	0.212	0.100	0.028	0.210	0.094	0.019	35.155	5.276	0.609
(11)	181,000	0.262	0.149	0.020	0.267	0.121	0.022	45.329	6.583	0.598
(12)	245,760	0.228	0.101	0.016	0.227	0.102	0.018	38.372	5.648	0.661
(13)	414,720	0.436	0.187	0.033	0.450	0.205	0.035	74.056	10.972	1.062
(14)	414,720	0.491	0.214	0.038	0.504	0.229	0.036	85.701	12.234	1.266
(15)	262,144	0.277	0.116	0.025	0.276	0.126	0.024	47.112	7.173	0.697
(16)	1,745,280	2.592	1.184	0.184	2.532	1.145	0.177	439.323	62.674	5.644
(17)	5,474,148	2.691	1.342	0.377	2.488	1.260	0.288	402.185	67.144	13.515
(18)	1,048,576	1.677	0.651	0.107	1.558	0.716	0.108	268.647	38.267	3.223
(19)	995,520	0.681	0.269	0.043	0.724	0.291	0.051	104.993	15.771	1.689
(20)	387,228	0.339	0.135	0.019	0.342	0.154	0.023	59.133	8.521	0.802
(21)	393,216	0.329	0.144	0.025	0.343	0.156	0.030	56.893	8.434	0.858
(22)	262,144	0.014	0.002	0.010	0.016	0.009	0.004	2.109	0.470	0.243
(23)	154,401	0.275	0.106	0.018	0.262	0.123	0.025	43.936	6.368	0.624
(24)	393,216	0.299	0.130	0.021	0.304	0.138	0.024	51.706	7.614	0.742
(25)	245,760	0.294	0.131	0.014	0.296	0.130	0.024	49.857	7.145	0.649
(26)	262,144	0.356	0.172	0.015	0.307	0.141	0.025	52.295	7.626	0.701
(27)	2,073,600	2.513	1.313	0.188	2.528	1.152	0.219	431.911	61.197	6.356
(28)	393,216	0.350	0.150	0.021	0.360	0.162	0.028	60.802	9.059	0.827
(29)	4,271,400	2.028	0.851	0.143	1.982	0.937	0.166	333.102	49.833	6.114
(30)	17,448,000	16.950	7.056	0.904	15.700	7.061	1.050	2733.081	386.570	33.050
(31)	245,760	0.241	0.127	0.019	0.247	0.113	0.024	41.698	6.001	0.624
(32)	414,720	0.333	0.381	0.020	0.345	0.164	0.029	58.023	8.572	0.872
Sum		37.630	18.110	2.653	36.082	16.568	2.836	6141.555	892.911	91.132

As expected, the Java implementation was slower than C++ implementation (compare the results in Tables 1 and 2). Let us consider the results on the personal computer, i.e., on platforms P1 and P2. The most evident difference is at **I-2**, where platforms P1 and P2 lag behind the C++ implementation by more than 360% and 480%, respectively. Platform P2 was faster for **I-1** and **I-2**, while it was

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 19 January 2024 doi:10.20944/preprints202401.1432.v1

slightly slower for **I-3** (by less than 7%). The Java implementations were significantly slower than the C++ implementations on platform P3. Implementation **I-1**, written in Java, was slower by a hardly believable 1358%, **I-2** by more than 800%, and **I-3** by more than 460%. In the case of **I-1** coded in Java, the user would need to wait for 1 hour and 42 minutes to transform all 32 images, while, for **I-3**, this process is completed in one and a half minutes.

5. Conclusion

This paper considers three different implementations of the Move with interleaving (MwI) transform. Although MwI operates with the expected $O(n)$ time complexity, its CPU time usage differs considerably between various programming solutions. Although efficient implementations are necessary in embedded systems or computationally weaker platforms like Raspberry Pi, processing larger datasets can also be noticeably faster on modern personal computers.

Three different implementations of MwI were considered in the paper: the first utilised a standard vector to store the MwI status, the second incorporated a lookup table, while the third was based on a dynamic one-way connected list and a mechanism for reusing allocated memory chunks. All the implementations were encoded in C++ and Java. Additionally, three different platforms were used for the experiments: MS Windows 11, Linux with the Manjaro distribution, and Raspberry Pi with Linux. The first two platforms were run on the same computer to ensure a fair comparison. The experiments involved 32 continuous-tone greyscale raster images with various resolutions and contexts. Their pixels were transformed into a sequence using a scan-line approach and then processed with MwI. The results show that a careful implementation reduces the CPU time spent significantly. For example, the implementation based on the dynamic linked list was 1800% faster than the implementation using the standard vector. The lookup table accelerated the implementation, using a standard vector significantly; however, it did not reach the speed of the implementation with the dynamic linked list. Linux turned out to be slightly more efficient that Windows. As expected, the C++ implementations outperformed the Java implementations.

The Raspberry Pi platform cannot be compared directly to the two platforms running on a modern personal computer. However, the importance of careful implementation and the choice of programming language becomes even more evident. For example, the fastest implementation using a dynamic one-way connected linked list took 19 second if coded in C++, but one and a half minutes if coded in Java. The slowest implementation, using a standard vector, required 7 minutes and 32 seconds, while the same implementation coded in Java needed even 1 hour and 42 minutes.

Author Contributions: Conceptualisation, B.Ž.; methodology, B.Ž. and B.R.; software, B.Ž. and B.R.; validation I.K. N.L. and A.J.; formal analysis, B.Ž. N.L. and I.K.; investigation, B.Ž., B.R., and I.K; resources, I.K.; data curation, B. Ž, B. R. N.L., and A.J; writing—original draft preparation, B.Ž.; writing—review and editing, B.Ž., N.L., I.K., B.R., A.J.; visualisation, A.J.; supervision, I.K. and B.Ž.; project administration, I.K. and B.Ž.; funding acquisition, I.K.and B.Ž. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by the Slovenian Research and Innovation Agency under Research Project J2-4458, Research Programme P2-0041, and the Czech Science Foundation under Research Project 23-04622L.

Institutional Review Board Statement: Not applicable

Informed Consent Statement: Not applicable

Data Availability Statement: The data presented in this study are available on request from the corresponding author.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Clark, B.G. An efficient implementation of the algorithm "CLEAN". *Atron Astrophy.* **1980** *89(3)*, 377–378.
2. Hadjiconstantinou, E.; Christofides, N. An efficient implementation of an algorithm for finding K shortest simple paths. *Int J Networks.* **1999** *34(2)*, 88–101.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 19 January 2024 doi:10.20944/preprints202401.1432.v1

3. Goldberg, A. V. An Efficient Implementation of a Scaling Minimum-Cost Flow Algorithm. *J Algorithm.* **1997** *22(1)*, 1–29.
4. Lippert, R.A.; Mobarry, C.M.; Walenz, B.P. A Space-Efficient Construction of the Burrows–Wheeler Transform for Genomic Data. *J Comput Biol.* **2005** *12(7)*, 943–951.
5. Devkota, S.; Aschwanden, P.; Kunen, A.; Legendre, M.; Isaacs, K. E. CcNav: Understanding Compiler Optimizations in Binary Code. *IEEE T Vis Comput Gr.* **2021** *27(2)*, 667–677.
6. Chattopadhyay, S. *Compiler design*, 2nd ed.; PHI Learning: New Delhi, India, 2022.
7. Basford, P.J.; Johnston, S.J.; Perkins, C.S.; Garnock-Jones, T.; Tso, F.P.; Pezaros, D.; Mullins, R.D., Yoneki, E.; Singer, J.; Cox, S.J. Performance analysis of single board computer clusters. *Future Gener Comp Sy.* **2020** *102*, 278–291.
8. Khan, N.; Yaqoob, I.; Hashem, I.A.T.; Inayat, Z.; Ali, W.K.M.; Alam, M.; Shiraz, M.; Gani, A. Big Data: Survey, Technologies, Opportunities, and Challenges. *Sci World J.* **2014** *2014*, 712826.
9. Stunkel, C.B.; Graham, R.L.; Shainer, G.; Kagan, M.;Sharkawi, S.S.; Rosenburg, B.; Chochoia, G.A. The high-speed networks of the Summit and Sierra supercomputers. *IBM J Res Dev.* **2020** *64(3/4)*, 3:1–3:10.
10. Iserte, S.; Prades, J.; Reaño, C.; Silla, F. Improving the management efficiency of GPU workloads in data centres through GPU virtualization. *Concurr Comp-Pract E.* **2019** *22(2)*, e5275.
11. Khriji, S.; Benbelgacem, Y.; Chéour R.; El Houssaini, D.; Kanoun, O. Design and implementation of a cloud-based event-driven architecture for real-time data processing in wireless sensor networks. *J Supercomput.* **2022** *78*, 3374–3401.
12. Žlaus, D.; Mongus, D. Efficient method for parallel computation of geodesic transformation on CPU. *IEEE T Parall Distr.* **2020** *31(4)*, 935–947.
13. Qiu, M.; Ming, Z.; Li, J.; Gai, K.; Zong, Z. Phase-Change Memory Optimization for Green Cloud with Genetic Algorithm. *IEEE T Comput.* **2015** *64(12)*, 3528–3540.
14. Albers, S. Energy-efficient algorithms. *Communications of the ACM* **2010** *53(6)*, 86-96.
15. Žalik, B.; Strnad, D.; Podgorelec, D.; Kolingerová; Lukač, L.; Lukač, N.; Kolmanič, S.; Rizman Žalik, K.; Kohek, Š. A New Transformation Technique for Reducing Information Entropy: A Case Study on Greyscale Raster Images. Entropy **2023**, *25(12)*, 1591.
16. Shannon, C.E. A Mathematical Theory of Communication. *AT&T Tech J.* **1948**, *27(3)*, 379–423.
17. Cover, T.M.; Thomas, J.A. *Elements of Information Theory*, 2nd ed.; Wiley: Hoboken, USA, 2006.
18. Burrows, M.; Wheeler, D.J. A block-sorting lossless data compression algorithm. Technical report No. 124, Digital Systems Research Center, 1994.
19. Adjeroh, D.; Bell, T.; Mukherjee, A. *The Burrows-Wheeler Transform: Data Compression, Suffix Arrays, and Pattern Matching*, 2nd ed.; Springer Science + Business Media: New York, USA, 2008.
20. Abel. J. Post BWT stages of the Burrows-Wheeler compression Algorithm. *Software Pract Exper.* **2010**, *40(9)*, 751–777.
21. Ryabko, B. Y. Data compression by means of a 'book stack'. *Probl Inf Transm.* **1980**, *16(4)*, 265–269.
22. Salomon, D.; Motta, G. *Handbook of Data Compression*, 5th ed.; Springer: London, U.K., 2010.
23. Arnavut, Z.; Magliveras, S. S. Block sorting and compression. In *Proceedings of the IEEE Data Compression Conference, DCC'97, Snowbird, Utah, USA, 25-27 Mar 1997*; Storer, J. A.; Cohn, M., Eds.; IEEE Computer Society Press, USA, 1997; pp. 181-–190.
24. Sayood, K. *Introduction to Data Compression*, 4th ed.; Morgan Kaufman, Elsevier: Waltham, USA, 2012.
25. Rivest, R. On self-organizing sequential search heuristics. *Comm of ACM.* **1976** *19(2)*, 63–67.
26. Bentley, J. L.; Sleator, D. D.; Tarjan, R. E.; Wei, V. K. A Locally Adaptive Data Compression Scheme. *Commun ACM.* **1986**, *29(4)*, 320-–330.
27. Dorrigiv, R.; López-Ortiz, A.; Munro, J. I. An Application of Self-organizing Data Structures to Compression. In *Experimental Algorithms, Proceedings of the* 8th *International Symposium on Experimental Algorithms, SEA 2009, Dortmund, Germany, 3-6 Jun 2009*; Vahrenhold, J., Ed.; Lecture Notes in Computer Science 5526; Springer: Berlin, Germany, 2009; pp. 137–148.
28. Jelenković, P.J.; Radovanović, A. Least-recently-used caching with dependent request. *Theor Comput Sci.* **2004** *326*, 293-327.
29. Žalik, B.; Lukač, N. Chain code lossless compression using Move-To-Front transform and adaptive Run-Length Encoding. *Signal Process Image Commun.* **2014**, *29(1)*, 96-–106.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 19 January 2024 doi:10.20944/preprints202401.1432.v1

30. Arnavut, Z. Move-To-Front and Inversion Coding. In *Proceedings of the IEEE Data Compression Conference, DCC'2000, Snowbird, Utah, USA, 28-30 Mar 2000*; Cohn, M.; Storer, J. A., Eds.; IEEE Computer Society Press, USA, 2000; pp. 193-–202.
31. Žalik, B.; Žalik, M.; Lipuš, B. On Move-To-Front Implementation. Proceedings on 27th International Conference on Circuits, Systems, Communications and Computers (CSCC), Rhodes Island, Greece, 19-22 July, 2023. IEEE Computer Society, Los Alamitos (CA), 2023; 43–47.
32. Manber, U.; Myers, G. Suffix arrays: a new method for on-line string search. Proceedings of the first annual ACM-SIAM symposium on Discrete algorithms (SODA '90), San Francisco (CA), USA, 22-24 Jan, 1990, Society for Industrial and Applied Mathematics, Philadelphia (PA); 319–327.
33. Manber, U.; Myers, G. Suffix arrays: a new method for on-line string search. *SIAM J Comput.* **1993** *22(5)*, 935–948.
34. Nong, G.; Zhang, S.; Chan, W. H. Two efficient algorithms for linear time suffix array construction. *IEEE T. Comput.* **2011**, *60(10)*, 1471—1484.
35. Kärkkäinen, J.; Sanders, P.; Burkhardt, S. Linear work suffix array construction. *J. ACM.* **2017** *53(6)*, 918—936.
36. Li, Z; Li, J.; Huo, H. Optimal In-Place Suffix Sorting. In *String Processing and Information Retrieval, 25th International Symposium, SPIRE 2018, Lima, Peru, 9-11 Oct 2018*; Gagie, T.; Moffat, A.; Navaro, G.; Cuadros-Vargas, E. Eds.; Lecture Notes in Computer Science 11147; Springer: Cham, Germany, 2018; pp. 268—284.
37. Stroustrup, B. *The C++ Programming Language*, 4th ed.; Addison-Wesley: Upper Saddle River, NJ, USA, 2013.
38. What every programmer should know about memory. Available online: https://people.freebsd.org/~lstewart/articles/cpumemory.pdf (accessed on 30. 12. 2023).
39. Cormen, T.H.; Leiserson, C.E.; Rivest, R.L.; Stein, C. *Introduction to algorithms*, 3th ed.; Massachusetts Institute of Technology :Cambridge, MA, USA, 2009.
40. Repnik, B.; Žalik, B. Six implementations of MwI transform. Available online: https://gemma.feri.um.si/projects/slovene-national-research-projects/j2-4458-data-compression-paradigm-based-on-omitting-self-evident-information/software/ (accessed on 15. 01. 2024).

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.