Compressibility - LZ77 & BWT

I. The program

1. Lempel-Ziv 1977 / LZ77

a. The idea

To implement the LZ77, we used the 5 hash tables method we have seen in class. We use 5 unordered_map with a string as the key and an unsigned integer array as the second element. The strings are all the combinations of letters we have seen so far and the integers in the array are the indexes where you can find them. We use 5 different containers because we only store combinations of 1, 2, 4, 8 and 16 characters. Each map contains only words of 1 length.

b. lz77()

The algorithm is built on a loop that works until we haven't seen all the characters of the string we are working on. First, the algorithm verifies if there is at least 16 characters left in the string. If yes, we enter the condition and we call *searchInMap()* which will verify if the next 16 characters have already been seen. If they haven't been seen, we then enter an other condition which will also call *searchInMap()* but with the hash table of length 8 and only 8 characters. If the 8 characters have never appeared before, we look the next 4 character using the same method. Same with 2 characters. If we haven't seen anything in those 4 hash tables, we look the next character in the map of length 1. If we find it in the map, we add the next 2 characters in the map of length 2 and if we don't find it, we add it in the map of length 1. We then add 1 to the variable *results* which count the number of runs and we add the length of the word we found to *i* (which is the index position in our string). Finally, we print the results and how much time the algorithm took.

c. searchInMap()

Almost all our implementation of LZ77 is based on *searchInMap()* function. We give a pointer to a string, a mapNumber (which is the length of the words in the map), a map and 3 values that we use to give the result back. First, the function will search for the next mapNumber characters of the string in the map. If we don't find it, we leave the function. But if we find it, we need to find the next substring that we can add. For this, we go through all the next characters 1 by 1, until we find a substring that have never been seen before. To compare the new substring with what has already been seen, we use the indexes array which is the second element of our unordered_map. When we find a substring that has never been seen before, we change the Boolean *found* to true (so we don't enter in each if condition) and we have a *i* and a *j* that give us all we need to find the string: i being the index position and j the length. If this function find something, *insertSuffix()* is called to make sure all the new suffixes are stored and their indexes updated.

d. insertSuffix()

This method takes all the maps as parameter, as well as the index of the new word and its length. It then inserts all the suffixes of length 1, 2, 4, 8 and 16 in the good maps.

2. Burrows-Wheeler Transform / BWT

a. The idea

At first, we tried to implement the BWT with a simple method. We would put every suffix of the string in an array and we would sort it and then compute the BWT string. Sadly, this method only works with small strings because it takes a lot of memory space and crashes for big string. We then add the idea to use an unsigned integer vector, the integers being the starting index of the suffix. We would only need to store the original string and an array of unsigned integer going from 0 to the number of characters in the string – 1. We would then only need to compute the BWT by taking it in account.

b. BWT()

The main function is simple, it is just a bridge between all the other methods. First, we fill an array with numbers from 0 to the size of the string - 1. We then sort everything thank to a radix sorting method we implemented. Finally, we compute the BWT string, we print it, and we compute and print the number of runs.

c. Sorting method

Sorting the array with a fast and efficient method was the hardest thing we had to do for this BWT. After having the idea of the indexes array, we had to create a special function to give to the std::sort function of C++. For this, we tried to use the data.substr(i) method for the strings which gives the substring of data from the character at index i to the end. Sadly, it was too long when used on bibleline.txt. We then tried to use the MSD Radix Sorting we coded in a previous assignment, but we faced a stack overflow because of its recursive calls. To counteract this problem, we have written a new version of the MSD Radix Sort as an iterative function. For the small array, we used our previous std::sort method using .substr(), but even with all of this it was too long. When searching for what took a lot of time, we found that it was our std::sort that was really slow because of the .substr() function. We then changed our comparison function to a loop which would only compare the number of characters needed to sort the array. With this technique, we manage to have something which is only less than 10 seconds long for all the example files.

d. BWTstring()

When the array is sorted, we need to compute the BWT string. For this, we just need to go through the array we just sorted and add to a string the letter which is before the index number (if we have 3 in our array box, we put the letter at index 2 in the string). We have a special case when finding the 0 : we put the '\0' character in the string.

e. numberOfRuns()

Our last function is the one that compute the final result we need to show to the user. For each letter in the BWT string, we check if it is different from the last one (using a temporary variable storing the last character). If it is different, we change the temporary variable to the new character. We then check if the character is different than '\0', if it is, we add 1 to the result.

II. The results

1. Lempel-Ziv 1977 / LZ77

```
What do you want to do ?
1) Change file  -  Current file: files/aaa.txt
2) LZ77: Lempel-Ziv 1977
3) BWT: Burrows-Wheeler Transform
Else) Quit
>> 2

Result: 16


      LZ77 was : 0.524598 s long
```

LZ77 results for aaa.txt

```
What do you want to do ?
1) Change file  -  Current file: files/aabb.txt
2) LZ77: Lempel-Ziv 1977
3) BWT: Burrows-Wheeler Transform
Else) Quit
>> 2

Result: 29


      LZ77 was : 0.307169 s long
```

LZ77 results for aabb.txt

```
What do you want to do ?
1) Change file  -  Current file: files/bibleline.txt
2) LZ77: Lempel-Ziv 1977
3) BWT: Burrows-Wheeler Transform
Else) Quit
>> 2

Result: 61393


      LZ77 was : 4.02521 s long
```

LZ77 results for bibleline.txt

As we can see, the LZ77 algorithm is pretty fast for all the example files, even though it is a little longer for bibleline.txt.

2. Burrows-Wheeler Transform / BWT

All the BWT runs below don't have the BWT string printed for a better understanding.

```
What do you want to do ?
1) Change file  -  Current file: files/aaa.txt
2) LZ77: Lempel-Ziv 1977
3) BWT: Burrows-Wheeler Transform
Else) Quit
>> 3

Sorting...
Number of runs: 1


      BWT was : 8.89821 s long
```

BWT results for aaa.txt

```
What do you want to do ?
1) Change file  -  Current file: files/aabb.txt
2) LZ77: Lempel-Ziv 1977
3) BWT: Burrows-Wheeler Transform
Else) Quit
>> 3

Sorting...
Number of runs: 4


      BWT was : 4.48201 s long
```

BWT results for aabb.txt

```
What do you want to do ?
1) Change file  -  Current file: files/bibleline.txt
2) LZ77: Lempel-Ziv 1977
3) BWT: Burrows-Wheeler Transform
Else) Quit
>> 3

Sorting...
Number of runs: 226944


      BWT was : 0.142592 s long
```

BWT results for bibleline.txt

We can see that our BWT implementation is significantly longer for strings with a lot of repetitions because it needs to sort everything. Even though aaa.txt is only made of repetitions and is the worst-case scenario for our program, it doesn't go above the 10 seconds mark. We can also see that it is very fast for bibleline.txt, but we have a little error of 1 in the result. We think it is because we don't have the '\n' character in our string because the getline() function we use to create the string stops at every jump to the line.