VERSION 5.0

SEPTEMBER 8, 2014

PROJECT OF STREAM PROCESSING

PRESENTED BY: SILIN WANG

TUTOR: FRÉDÉRIC PENNERATH

CONTENTS

1. INTRODUCTION

This project is the realization of two well-known algorithms in the field of stream mining. The first one is a simple algorithm for finding frequent elements in streams (name first algorithm below), the second one is for estimating entropy (name second algorithm below).

In addition, to assess speed/memory performance gain and approximation error. I made another two naive but brute-force algorithms to compare with the two algorithms I mentioned above.

All the programs in this project are implemented in C++ and the code should compile on GNU G++/Linux.

2. IMPLEMENTATATION

- **FIRST ALGORITHM (Find Frequent Elements In Streams)**

Presentation:

This is an algorithm for identifying in a multiset the items with frequency more than a threshold θ. The algorithm requires two passes, linear time, and space $1/\theta$. The first pass is an on-line algorithm, generalizing a well-known algorithm for finding a majority element, for identifying a set of at most $1/\theta$ items that includes, possibly among others, all items with frequency greater than θ.

Logic:

As we can see in the figure 1 below, it's the flow chart of this algorithm. It shows the key part of this algorithm. x[1]…x[N] is the input stream, K is a set of symbols initially empty and count is an array of integers indexed by K. For i := 1,…,N, if x[i] is in K then count[x[i]] := count[x[i]] + 1, else insert x[i] in K and set count[x[i]] := 1. After that, we check if $|K| > 1/\theta$. If so, for all a in K, count[a] := count[a] -1 and if count[a] = 0 then we delete a from K. Once the set K is found in the first pass, with a second pass we can find the tallies of all symbols in K, and then delete from K the symbols with tally less than θN. After all this have been done, we can output K.

Code:

The figure 2 below shows the code of the key part of this algorithm, as we can see, the input stream is read line by line. Each symbol of the line is processed as the procedure presented above.

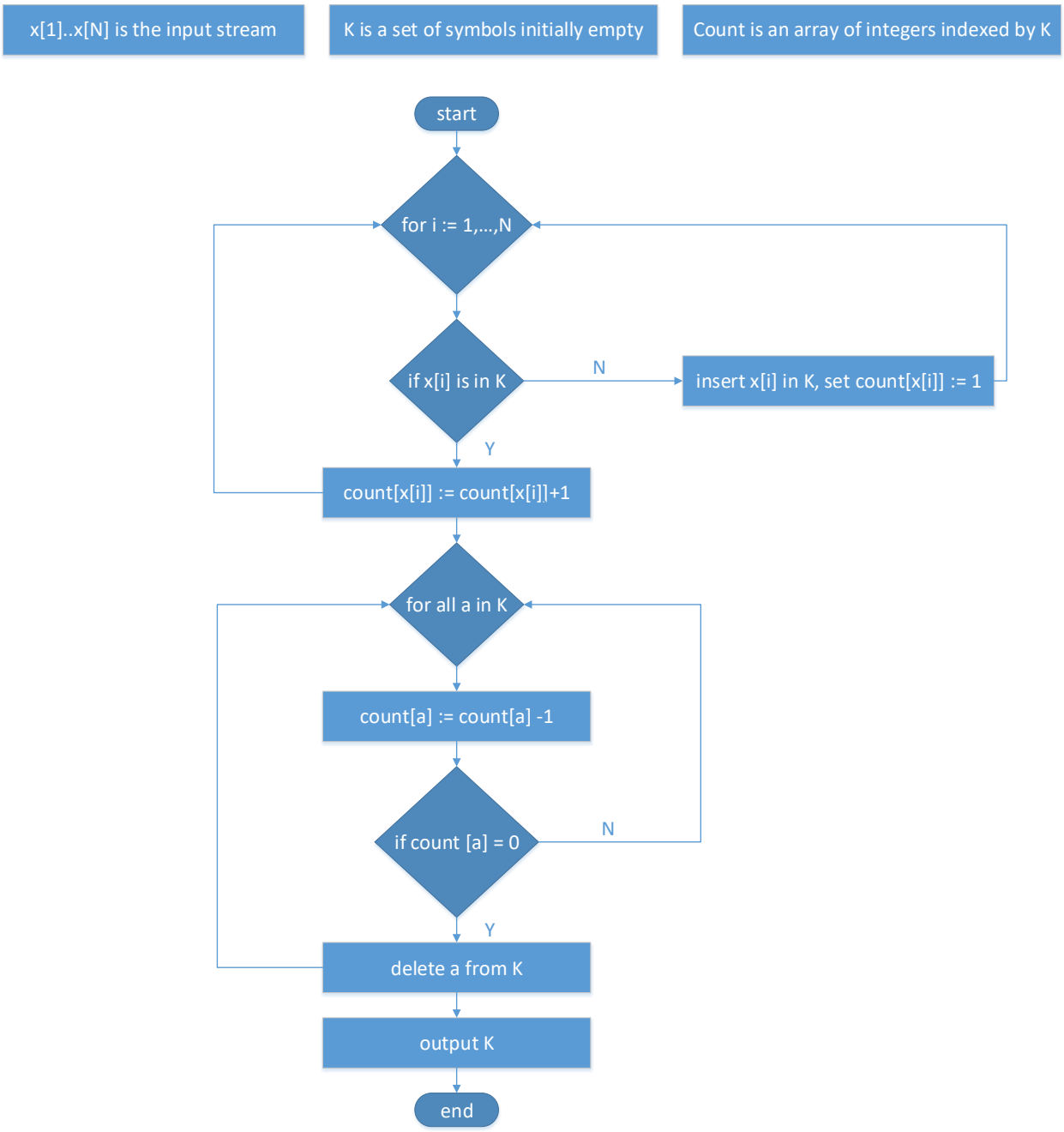

FIGURE 1 FLOW CHART OF THE FIRST ALGORITHM

```
do{
    std::cin >> line;
    // std::cout << line << " ";
    if(line != "*"){
        start = std::chrono::steady_clock::now();
        // count the length of the input stream
        ++m;
        // if x[i] is in K
        if(K.find(line) != K.end()){
            // then count[x[i]] := count[x[i]] + 1
            K[line] ++;
        }
        // else insert x[i] in K, set count[x[i]] := 1
        else{
            K[line] = 1;
        }
        // if |K| > 1/theta
        if (K.size() > 1/theta){
            // for all a in K do
            for (auto itK = K.begin(); itK != K.end();){
                // count[a] = count[a] - 1
                itK -> second --;
                // if count[a] = 0
                if(itK -> second == 0){
                    // delete a from K
                    K.erase(itK++);
                }
                else{
                    ++itK;
                }
            }
        }
        end = std::chrono::steady_clock::now();
        duration += std::chrono::duration_cast<std::chrono::duration<double>>(end - start);
    }
}while(line != "*");
// find the tallies of all symbols in K
for (auto itK = K.begin(); itK != K.end(); ++itK){
    itK -> second = 0;
}
```

FIGURE 2 CODE OF THE KEY PART OF THE FIRST ALGORITHM

- **BRUTE-FORCE ALGORITHM OF FINDING FREQUENT ELEMENTS**

Presentation:

This is a naive but brute-force algorithm for finding frequent elements in stream, it uses the most original method which count the frequency of all symbols. The result of this algorithm is exactly same as the first algorithm.

Logic:

As we can see in the figure 4 below, it's the flow chart of this algorithm. The first part of this algorithm is same as the first algorithm, x[1]…x[N] is the input stream, K is a set of symbols initially empty and count is an array of integers indexed by K. For i := 1,…,N, if x[i] is in K then count[x[i]] := count[x[i]] + 1, else insert x[i] in K and set count[x[i]] := 1. In the second part, for all a in K, if count[a] < θN then we delete a from K. In the end we output K.

Code:

The figure 3 below shows the code of the key part of this algorithm.

```
do{
    std::cin >> line;
    if(line != "*"){
        start = std::chrono::steady_clock::now();
        ++m;
        if(K.find(line) != K.end()){
                // then count[x[i]] := count[x[i]] + 1
            K[line] ++;
        }
            // else insert x[i] in K, set count[x[i]] := 1
        else{
            K[line] = 1;
        }
        end = std::chrono::steady_clock::now();
        duration += std::chrono::duration_cast<std::chrono::duration<double>>(end - start);
    }
}while(line != "*");
int threshold = m*theta;
start = std::chrono::steady_clock::now();
for (auto itK = K.begin(); itK != K.end();){
    if(itK -> second > threshold){
        ++itK;
    }
    else{
        K.erase(itK++);
    }
}
```

FIGURE 3 CODE OF THE BRUTE-FORCE ALGORITHM FOR FINDING FREQUENT ELEMENTS

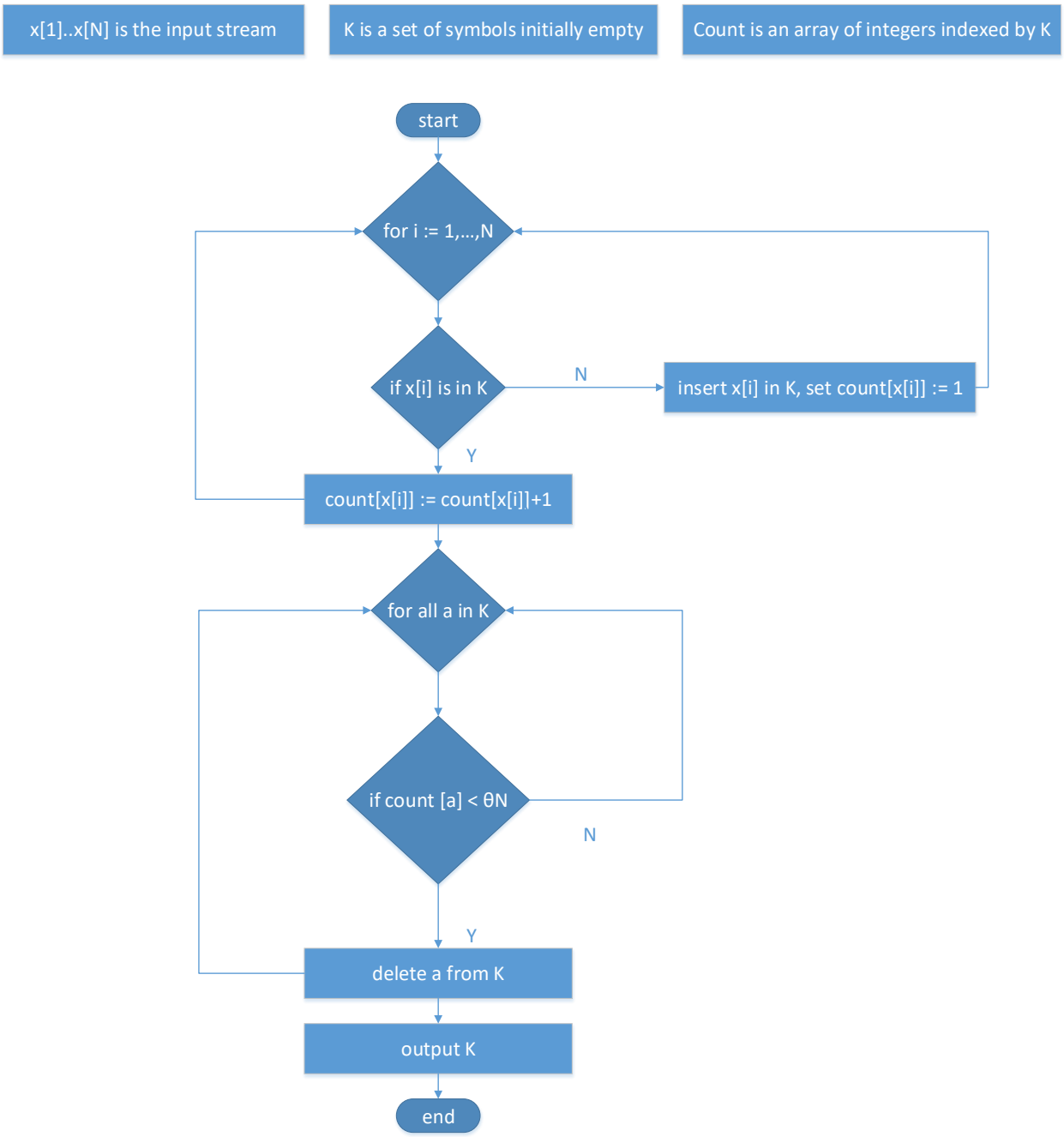

FIGURE 4 FLOW CHART OF THE BRUTE-FORCE ALGORITHM FOR FINDING FREQUENT ELEMENTS

- **SECOND ALGORITHM (Estimate Entropy)**

Presentation:

This is a two-pass algorithm for estimating entropy of a stream. It is based on a two-pass algorithm for finding a majority item such as the one by Boyer and Moore.

Logic:

As we can see in the figure 6 below, it's the flow chart of this algorithm. This algorithm needs two passes, in the first pass, we find a majority candidate x for A and compute a final estimator Z of A. Then we make a second pass over A, we count the frequency mx of x and compute a final estimator Y of A/{x}. After this done, we check if mx <= m/2, if so, we return Z as the result. Else we calculate k = m – mx and return return kY/m + (m-k)/m*lg(m/m-k) as the result.

Code:

The figure 5 below shows the code of the key part of this algorithm. It's the first pass of this algorithm, in this pass, a majority candidate will be found out and the final estimator is computed.

- **BRUTE-FORCE ALGORITHM FOR ESTIMATING ENTROPY**

Presentation:

This is a brute-force algorithm for estimating entropy. It use $H := \sum_{i=1}^{n}(mi/m)\lg(m/mi)$ to calculate the entropy of the stream.

Logic:

As we can see in the figure 8 below, it's the flow chart of this algorithm. The procedure of this algorithm is rather simple. It the realization of the formula $H := \sum_{i=1}^{n}(mi/m)\lg(m/mi)$.

Code:

The figure 7 below shows the key part of the code of this algorithm. It's the realization of the procedure I presented above.

```
do{
    std::cin >> line;
    if(line != "*"){
        start = std::chrono::steady_clock::now();
        // find a majority candidate x as in Boyer and Moore
        // if count == 0
        if(count == 0){
            candidate = line;
            count = 1;
        }else{
            if(line == candidate){
                ++count;
            }
            else{
                --count;
            }
        }
            // calculate r
        ++m;
        for (int j = 0; j < s1s2; ++j)
        {
            int p = rand()%m;
            if(p == (m - 1)){
                al[j] = line;
                r[j] = 1;
            }else{
                if(al[j] == line){
                    ++r[j];
                }
            }
        }
        end = std::chrono::steady_clock::now();
        duration += std::chrono::duration_cast<std::chrono::duration<double>>(end - start);
    }
}while(line != "*");
// compute final estimator Z
double Z = getZ(r, s1, s2, m);
```

FIGURE 5 CODE OF THE SECOND ALGORITHM

A (a[1],…,a[m]) is the input stream

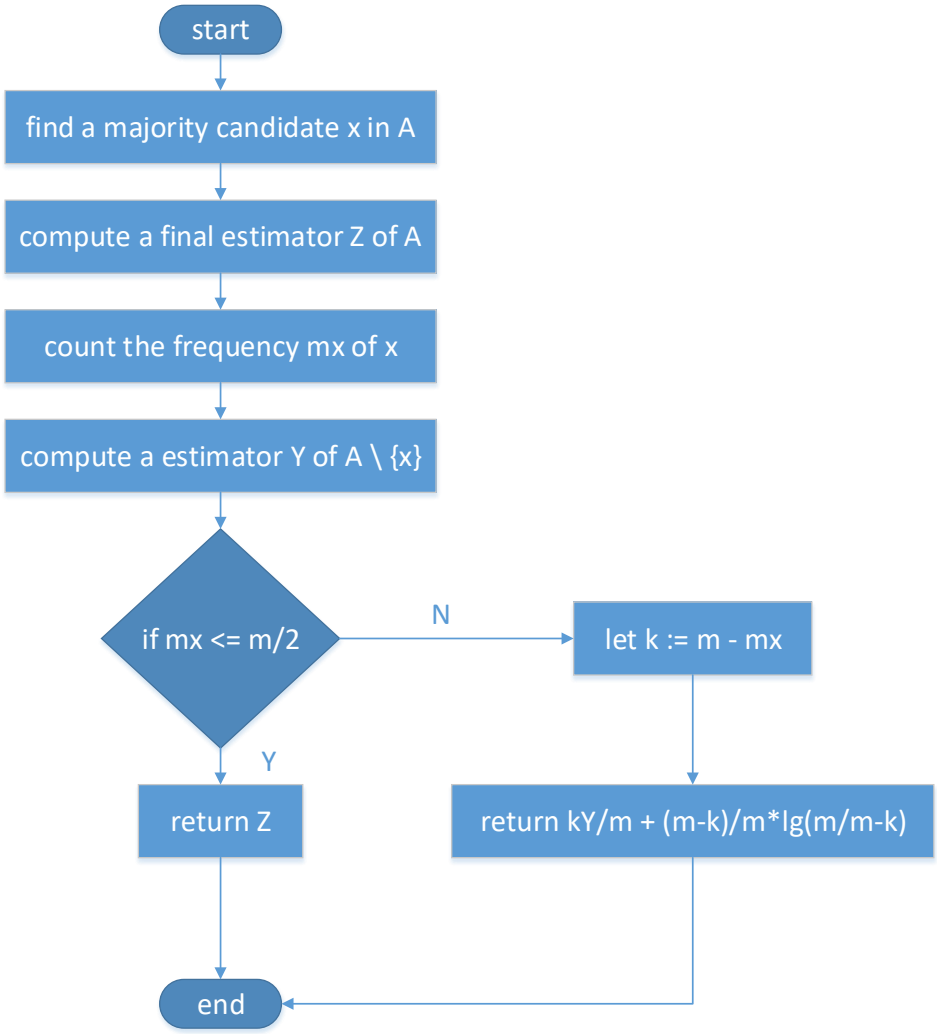

FIGURE 6 FLOW CHART OF THE SECOND ALGORITHM

```
do{
    std::cin >> line;
    if(line != "*"){
        ++m;
        if(K.find(line) != K.end()){
                // then count[A[i]] := count[A[i]] + 1
            K[line] ++;
        }
            // else insert A[i] in K, set count[A[i]] := 1
        else{
            K[line] = 1;
        }
    }
}while(line != "*");
double H = 0.0;
for (auto itm = K.begin(); itm != K.end(); ++itm){
    H += ((double)(itm -> second)/(double)m*log2((double)m/(double)(itm -> second)));
}
```

Figure 7 code of the brute-force algorithm for estimating entropy

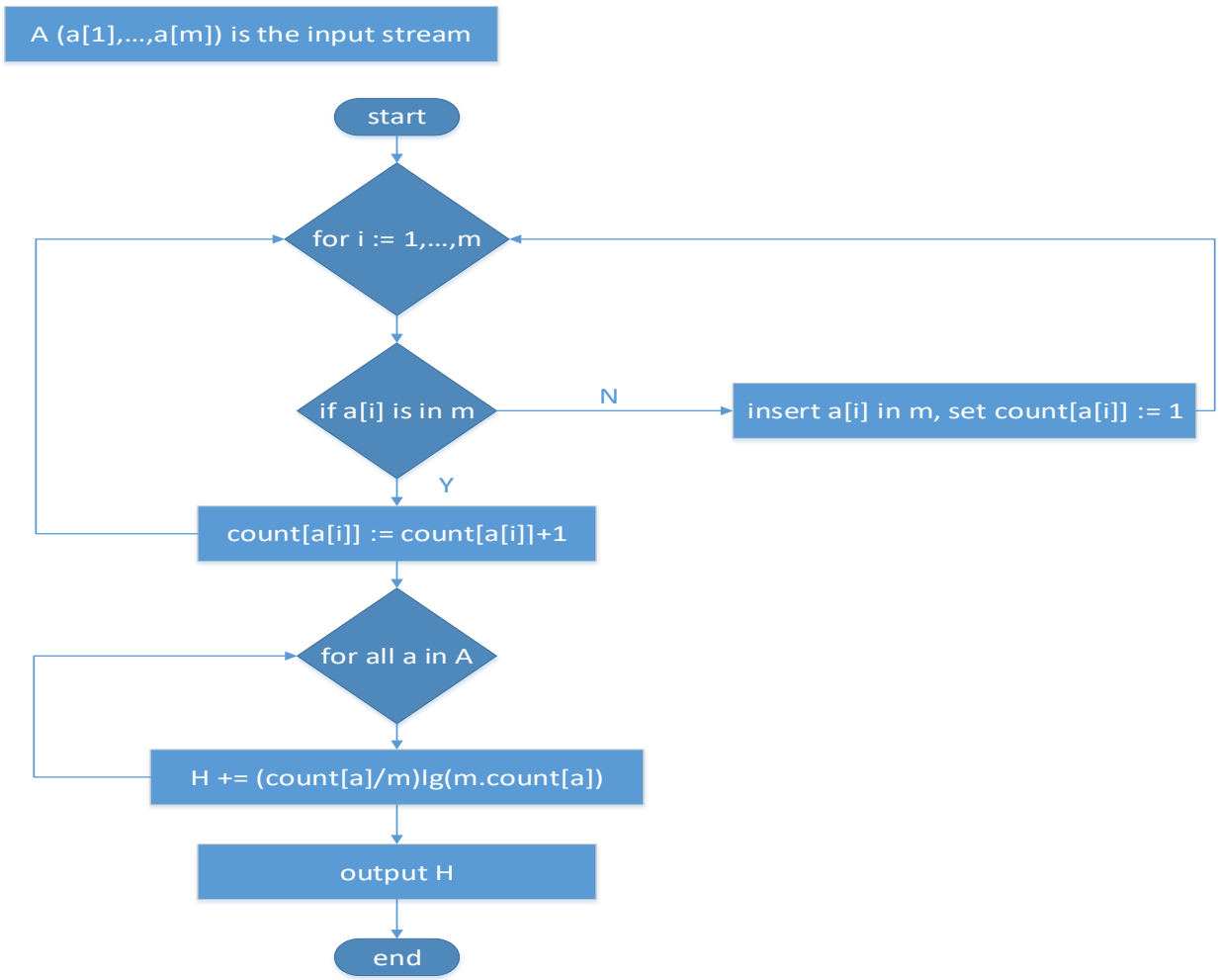

FIGURE 8 FLOW CHART OF THE BRUTE-FORCE ALGORITHM FOR ESTIMATING ENTROPY

3. TESTING

- **GENERATE STREAM**

Presentation:

For testing these algorithms, I made a small program for printing a pseudo random stream. Symbols in the stream are random generated numbers.

It has four parameters which are <number of lines> <number of symbols> <type of distribution> <times> <seed> (<r>). <number of lines> defines the size of stream, <number of symbols> is the number of all the different symbols in a stream, <type of distribution> can let us choose the one of the 2 different types of the distribution which are uniform, geometric distribution. <times> is the times for printing a stream, i.e 1 means to print the stream once and 2 means 2 times. I made this parameter because the first and second algorithm both need two passes. <seed> is the seed for the random number generator, different seed can yield different number. <r> is a factor of the geometric law.

Code:

The figure 9 below shows the code of the key part of this program.

```
// random number generator
for (int i = 0; i < numPasses; ++i){
    std::default_random_engine generator(seed);

    std::discrete_distribution<int> distribution(weights.begin(), weights.end());
    // generate stream
    for (int i = 0; i < numLines; ++i){
        // define the symbol
        int symbol = distribution(generator);
        // output line
        std::cout << symbol << std::endl;
    }
    std::cout << "*" << std::endl;
}
```

FIGURE 9 CODE OF THE GENERATE STREAM PROGRAM

4. RESULTS

- **FIND FREQUENT ELEMENTS**

Presentation:

This is the comparison between the two algorithms for finding the frequent items in stream, one is the algorithm of Karp et al. ("maj"), and the other one is the naive algorithm that uses the most original method ("majB").

Pre-Analysis:

The time and space complexity of "maj" and "majB" is shown in the table below (**m**: size of stream, **n**: number of symbols, **θ**: threshold, **K**: the size of **std::map K** in this two algorithms):

Name of algorithm	Time complexity	Space complexity
maj	$O(m\,log(1/\theta))$	$O(1/\theta)$
majB	$O(m\,log(K))$	$O(K)$

TABLE 1 COMPLEXITY OF "MAJ" AND "MAJB"

Comparison of the processing time between "maj" and "majB" with different size of stream:

- Performance

The number of symbols of the input stream is 10^6, and the threshold for finding frequent elements is 0.1. We can see in the figure 10 below, the curve in blue is the processing time of "maj" and the one in orange represents "majB". The processing time of both algorithms increase when the size of the input stream becomes larger between 10^5 and 10^7. When the size of the input stream is relatively small (smaller than 10^5 symbol), the performance of "maj" and "majB" are almost same. But when the size is larger than 10^5, the processing time of "majB" increases much faster than "maj".

The figure 11 below shows the processing memory of "maj" and "majB". With the increasing of the size of stream from 10^3 to 10^7, the memory of the "majB" is augmenting. But the processing memory of "maj" is totally different, it maintains at a fixed number.

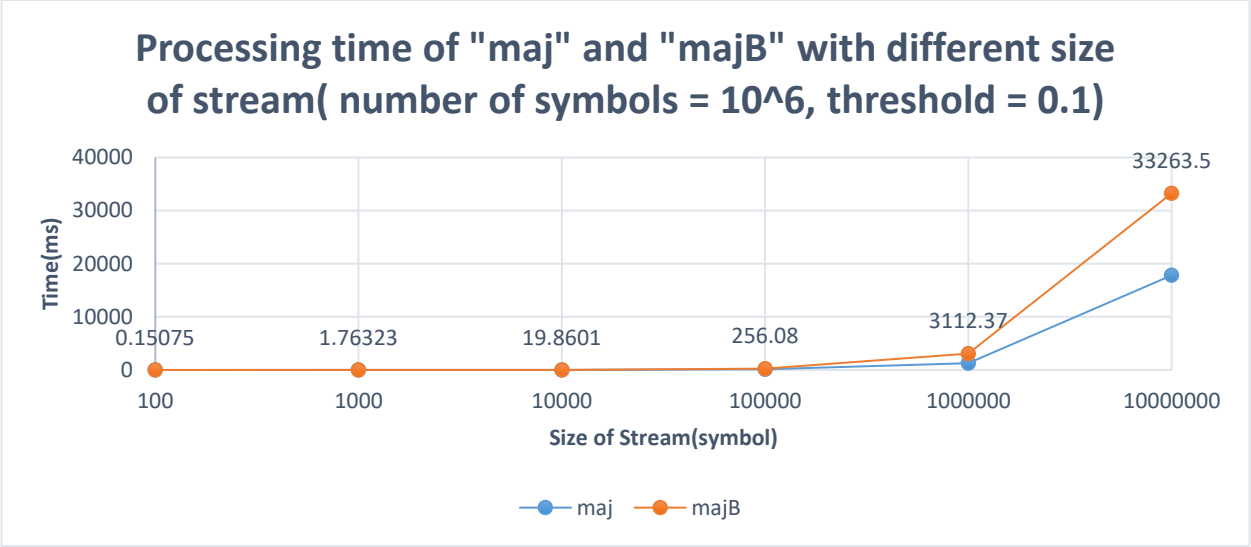

FIGURE 10 PROCESSING TIME OF "MAJ" AND "MAJB" WITH DIFFERENT SIZE OF STREAM

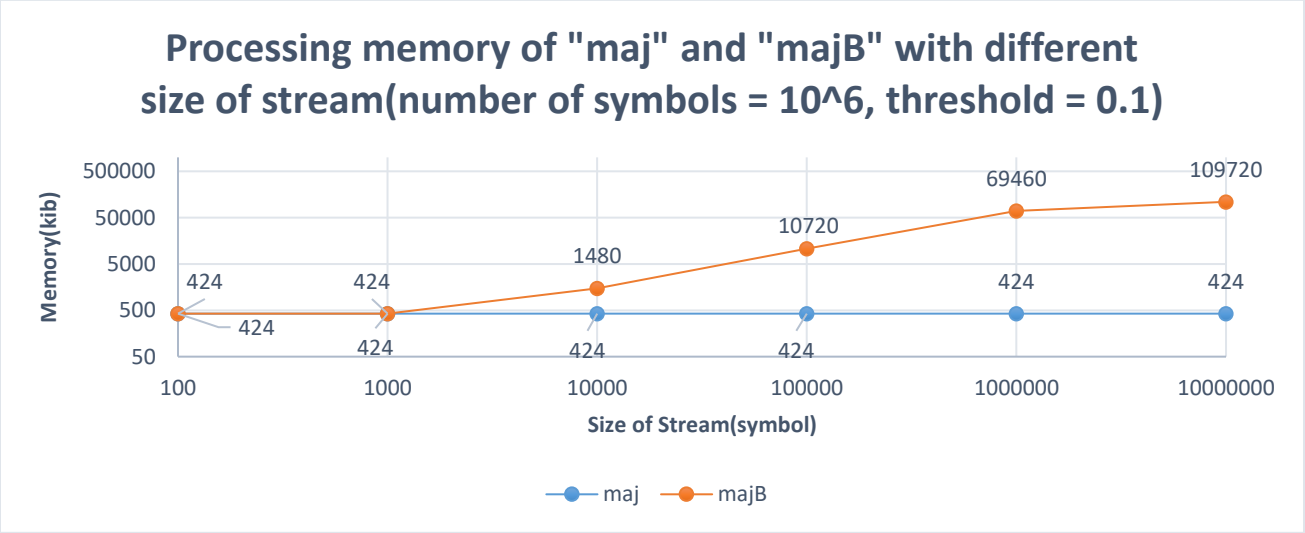

FIGURE 11 PROCESSING MEMORY OF "MAJ" AND "MAJB" WITH DIFFERENT SIZE OF STREAM

- Analysis

Processing time:

As we can see in the table 1, the time complexity of "maj" is $O(m\log(1/\theta))$. In this test, the threshold θ is always 0.1, so the time complexity of "maj" will augment only because of the increasing of the size of stream m.

Differently, the time complexity of "majB" is $O(m\,log(K))$, K is the size of std::map K in "maj" (equals to the number of all the different symbols in the input stream). When the size of the input stream increases, there will be more different symbols presenting in the input stream.

That means K will become larger and larger with the augment of the size of stream m. In this situation, when the size of stream increases, the time complexity of "majB" will increase because of both the augments of m and K.

Thus, when the size of the input stream becomes larger, the processing time of "majB" is increasing much faster than "maj". That means "maj" has the better performance compared to "majB" when it processes large size stream.

Processing memory:

Since the space complexity of "maj" is $O(1/\theta)$, as mentioned above, the threshold θ remains unchanged in this test, so the space complexity of "maj" will not change with the increasing of the size of stream m.

The space complexity of "majB" is $O(K)$, as explained above, K will become larger when the size of stream m increases. Thus, the space complexity of "majB" will keep increasing with the augment of the size of stream m.

Comparison of the processing time between "maj" and "majB" with different number of symbols:

- Performance

The size of the input stream is 10^6, and the threshold for finding frequent elements is 0.1. We can see in the figure 12 below, the curve in blue is the processing time of "maj" and the one in orange represents "majB". The processing time of "maj" almost remains unchanged when the number of symbols increases. Differently, the processing time of "majB" augments with the increasing of the number of symbols from 10 to 10^8, after 10^6, the speed of the augment seems to decrease.

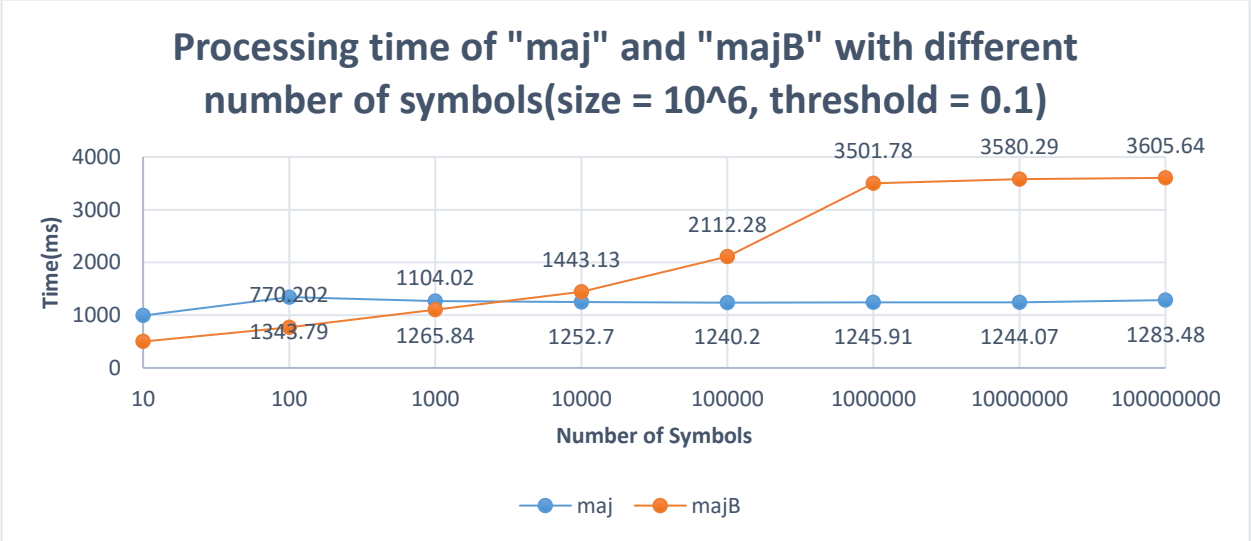

FIGURE 12 PROCESSING TIME OF "MAJ" AND "MAJB" WITH DIFFERENT NUMBER OF SYMBOLS

The figure 13 below shows the processing memory of “maj” and “majB”. With the increasing of the size of stream from 10 to 10^8, the memory of “majB” is augmenting between 10^3 and 10^8, after 10^6, the increasing becomes slower. Differently, the processing memory of “maj” always remains the same.

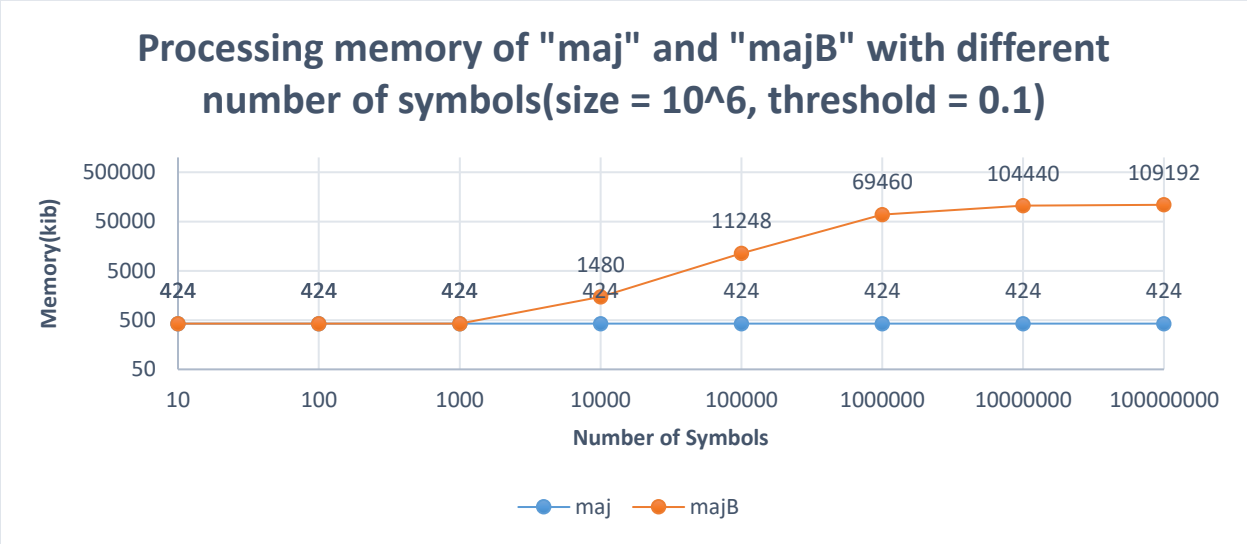

FIGURE 12 PROCESSING MEMORY OF "MAJ" AND "MAJB" WITH DIFFERENT NUMBER OF SYMBOLS

- Analysis

Processing time:

In this test, the size of stream m and the threshold θ keep unchanged. In this situation, the time complexity $O(m\log(1/\theta))$ of “maj” will not change in this test theorically.

Compared to “maj”, the time complexity $O(m\,log(K))$ of “majB” will keep augmenting because K is becoming larger with the increasing of the number of symbols n.

In addition, because of size of the input stream equals to 10^6, so after the point whose number of symbols is 10^6, the increasing of the number of the different symbols in the stream will slow down. That means the augment of K will slow down, that’s why the speed of the augment of the processing time of “majB” decreases after the number of symbols equals to 10^6.

Processing memory:

Since the space complexity of “maj” is $O(1/\theta)$, so it will stay the same with the increasing of the number of symbols of the input stream n.

Differently, the space complexity $O(K)$ of “majB” will keep increasing because of the augment of K when the number of symbols of the input stream becomes larger.

Same as the processing time, after the number of symbols equaling to 10^6, the speed of the increasing of the different symbols in stream slows down, so the augment of the processing memory of “majB” slows down.

Comparison of the processing time between “maj” and “majB” with different threshold

- Performance

The size of the input stream and the number of symbols are both 10^6. We can see in the figure 14 below, the curve in blue is the processing time of “maj” and the one in orange represents “majB”. The processing time of “majB” almost remains unchanged when the threshold for finding frequent elements increases. Differently, the processing time of “maj” decreases with the increasing of the threshold.

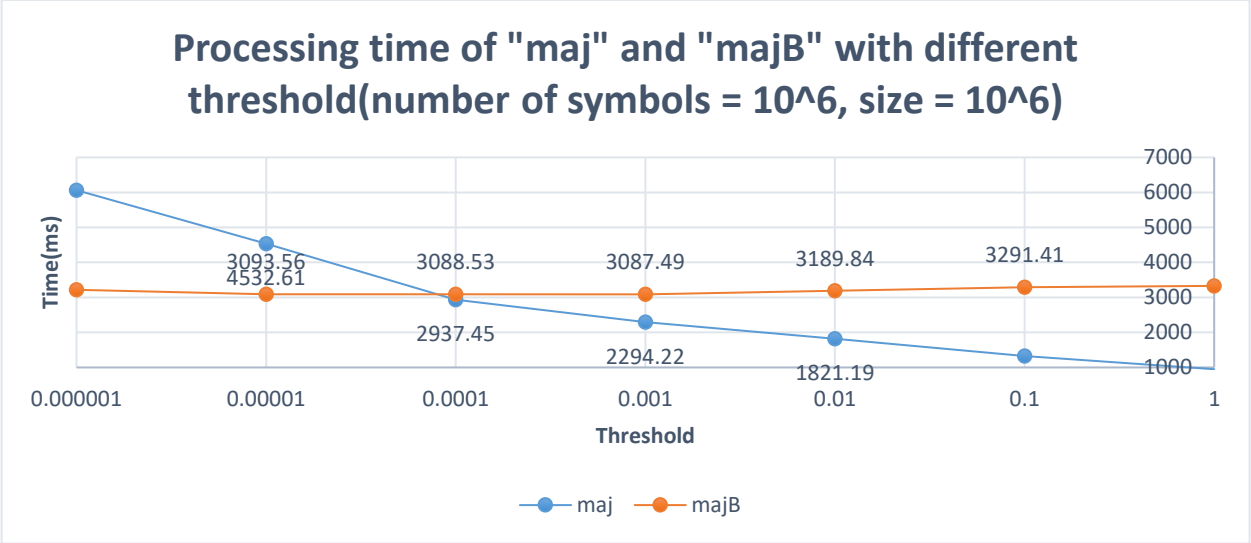

FIGURE 14 PROCESSING TIME OF "MAJ" AND "MAJB" WITH DIFFERENT THRESHOLD

The figure 15 below shows the processing memory of “maj” and “majB”. With the increasing of the threshold from 10^-6 to 1, the memory of “majB” remains the same. Differently, the processing memory of “maj” decreases between the threshold equals to 10^-6 and 10^-3. After 10^-3, the size of the processing memory of “maj” maintains at 424 kib.

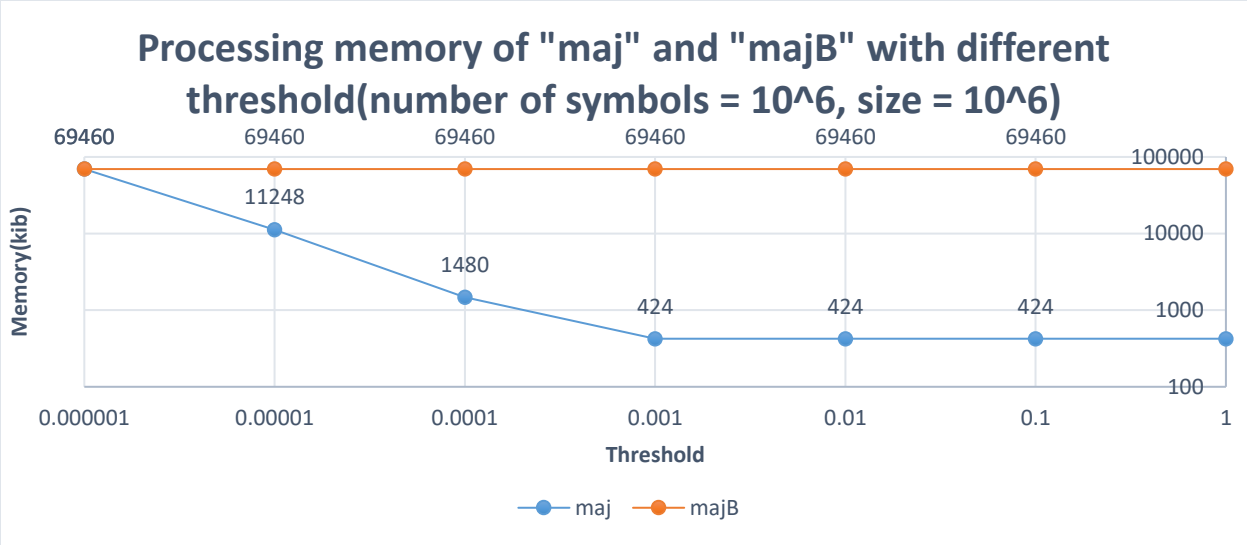

FIGURE 13 PROCESSING MEMORY OF "MAJ" AND "MAJB" WITH DIFFERENT THRESHOLD

- Analysis

Processing time:

The time complexity $O(m \log(1/\theta))$ of "maj" will decrease with the increasing of the threshold θ.

Compared to "maj", the time complexity $O(m\,log(K))$ of "majB" is irrelevant to the threshold. So the change of the threshold has no effect in the processing time of "majB".

Processing memory:

The space complexity $O(1/\theta)$ of "maj" will reduce with the increasing of the threshold θ. Since there is no relation between the threshold and the space complexity $O(K)$ of "majB", then the processing memory of "majB" remains the same when the threshold increases.

- **ESTIMATE ENTROPY**

Presentation:

This is the comparison between the two algorithms for estimating the entropy of the input stream, one is the algorithm of Amit et al. ("ent"), and the other one is the realization of the formula $H := \sum_{i=1}^{n}(mi/m)\lg(m/mi)$ ("entB").

Pre-Analysis:

The time and space complexity of “ent” and “entB” is shown in the table below (**m**: size of stream, **n**: number of symbols, **R**: times of the iteration for computing r, **K**: the size of **std::map K** in “entB”):

Name of algorithm	Time complexity	Space complexity
ent	$O(mR)$	$O(R)$
entB	$O(m\,log(K))$	$O(K)$

TABLE 2 COMPLEXITY OF “MAJ” AND “MAJB”

Comparison of the processing time between “ent” and “entB” with different size of stream

- Performance:

The figure 16 below shows the processing time of “ent” and “entB” with different size of stream. The blue curve is the performance of “ent”, the orange one is “ent B”. As we can see in the figure, the processing time of both “ent” and “ent B” increase when the size of the input stream becomes larger between 10^5 and 10^7. When the size is relatively small (under 10^5 symbol). There is few difference between the performance of “ent” and “entB”. When the size is larger than 10^5, the processing time of “ent” augments much slower than “entB”.

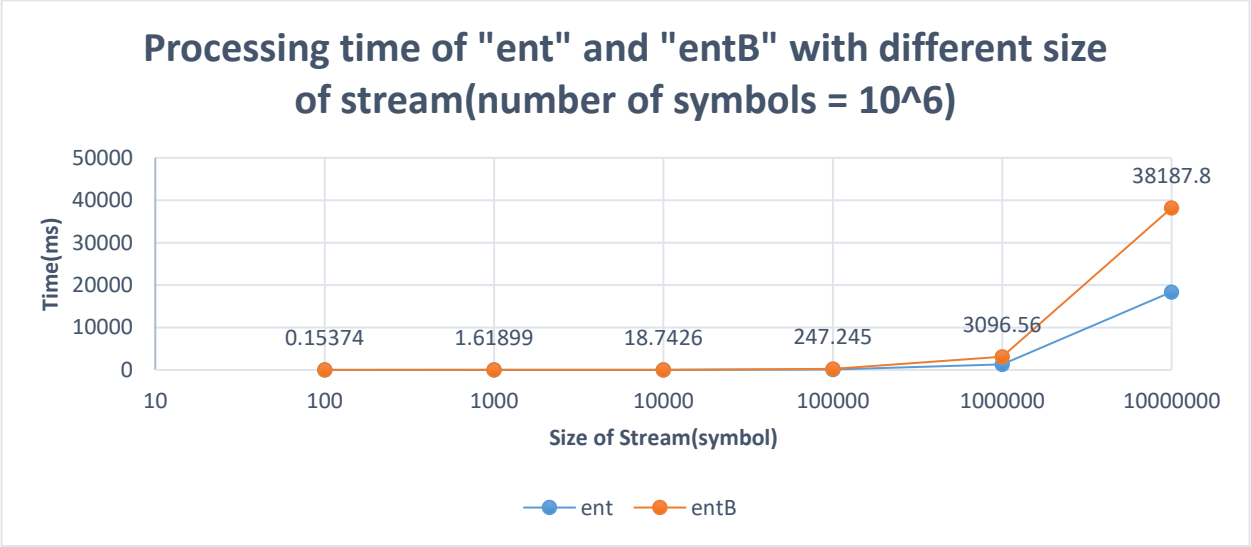

FIGURE 14 PROCESSING TIME OF "ENT" AND "ENTB" WITH DIFFERENT SIZE OF STREAM

The performance in figure 17 shows the performance of the processing memory of “ent” and “entB” with different size of the input stream. The blue curve is the performance of “ent”, the orange one is “entB”. We can see that the processing memory of “ent” maintain at 424 kib

when the size of stream increases. Unlike “ent”, the memory of “entB” increases between the size equal 10^3 and 10^7.

- Analysis

Processing time:

The times of iteration for computing r R will keep unchanged in this test. Thus, the time complexity $O(mR)$ of “ent” will become larger only because of the increasing of the size of stream m.

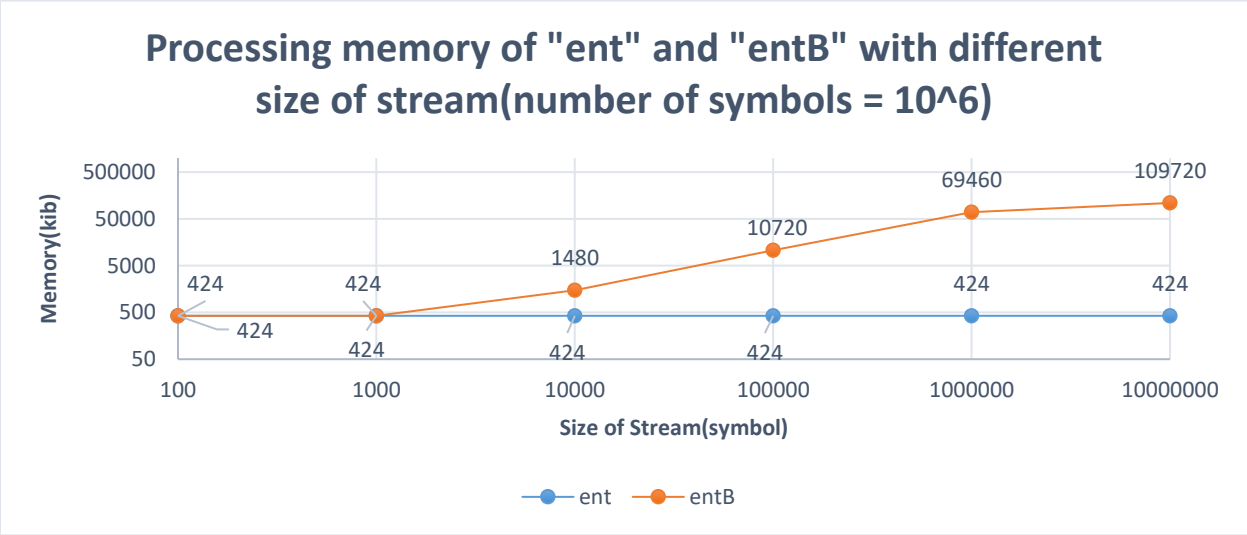

FIGURE 15 PROCESSING MEMORY OF "ENT" AND "ENTB" WITH DIFFERENT SIZE OF STREAM

The time complexity of “entB” is $O(m\,log(K))$, it will increase because of both the increasing of size of stream m and K like “majB”. That’s why the processing time of “entB” augment faster than “ent” when the size of stream becomes larger.

Processing memory:

The space complexity of “ent” is $O(R)$, as R will not change in this test, so the space complexity of “ent” will remain unchanged.

Different to “ent”, the space complexity of “entB” is $O(K)$. Like “majB”, the size of std::map

K in “entB” will keep increasing when the size of stream m becomes larger. For this reason, the space complexity of “entB” keeps increasing with the augment of size of stream m.

Comparison of the processing time between “ent” and “entB” with different number of symbols

- Performance:

The figure 18 below shows the processing time of “ent” and “entB” with different number of symbols. The blue curve is the performance of “ent”, the orange one is “entB”. As we can see in the figure, the processing time of “ent” doesn’t change so much when the number of symbols increases. But the processing time of “entB” keeps augmenting with the increasing of the number of symbols.

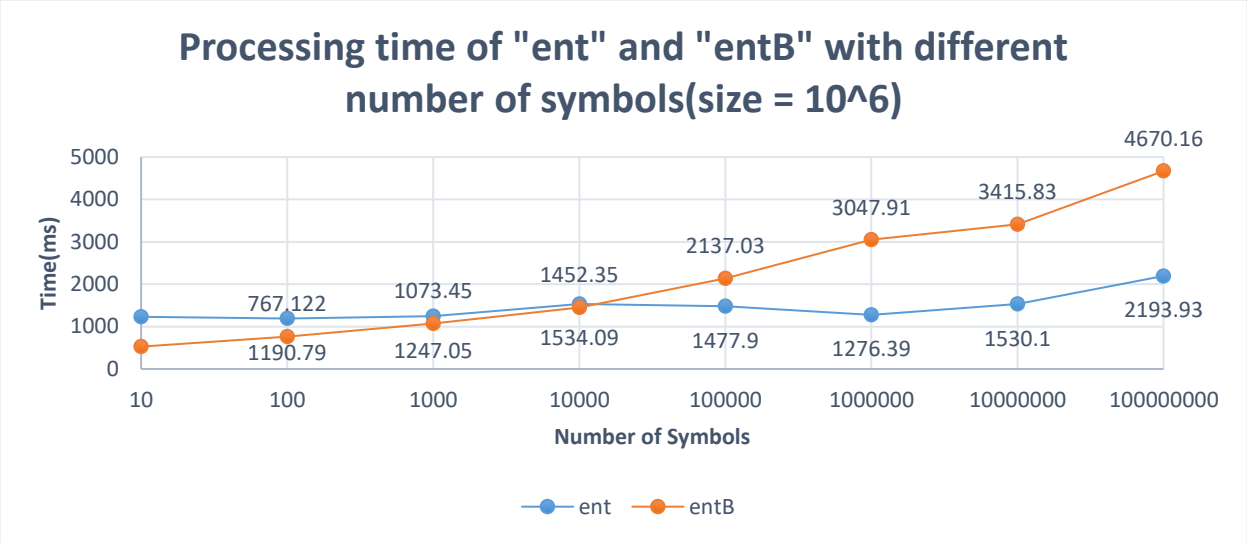

FIGURE 16 PROCESSING TIME OF "ENT" AND "ENTB" WITH DIFFERENT NUMBER OF SYMBOLS

The performance in figure 19 shows the performance of the processing memory of “ent” and “entB” with different number of symbols of the input stream. The blue curve is the performance of “ent”, the orange one is “entB”. We can see that the processing memory of “ent” maintain at 424 kib when the number of symbols increases. Unlike “ent”, the memory of “entB” keeps increasing between 10^3 and 10^8.

- Analysis

Processing time:

The time complexity $O(mR)$ of “ent” is irrelevant to the number of symbols, so it doesn’t change in this test theorically.

Differently, the time complexity $O(m\,log(K))$ of “entB” will augment with the increasing of number of symbols as “majB”.

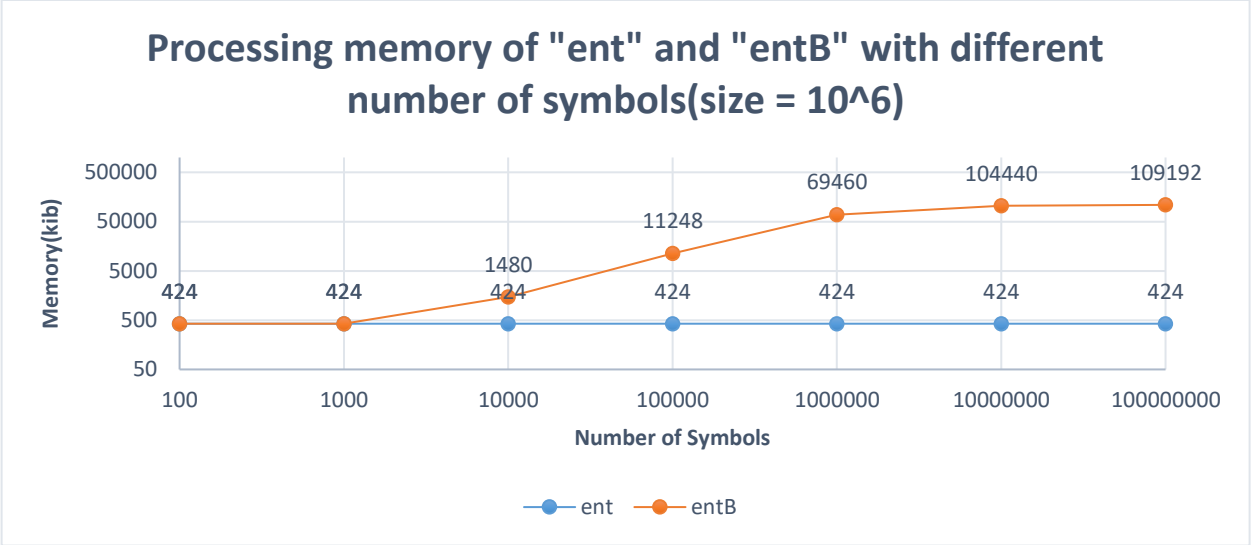

FIGURE 17 PROCESSING MEMORY OF "ENT" AND "ENTB" WITH DIFFERENT NUMBER OF SYMBOLS

Processing memory:

The space complexity of “ent” is $O(R)$, as R will not change in this test, so the space complexity of “ent” will remain unchanged.

The space complexity of “entB” is $O(K)$. K will become larger when the number of symbols increases. So the space complexity of “entB” keeps increasing with the augment of number of symbols.

5. CONCLUSION

After testing these algorithms, and comparing the performance between “maj” and “majB”, “ent” and “entB”. I think the first algorithm (“maj”) and second algorithm (“ent”) both have obviously better performance in running time and memory usage when processing large size data stream with a large alphabet compared to the naive algorithms. And these advantages will become more and more evident with the increasing of the size of stream or the number of symbols. Especially in the situation that the stream is too long to be stored in memory (sometime we can’t use the naive algorithms), these two algorithms (“maj” and “ent”) will be excellent choice for processing the stream.

As presented above, “maj” is an algorithm for finding frequent elements and “ent” is for estimating the entropy of stream. Those two algorithm are very practical for processing the huge data streams coming from the network. For example, “maj” can help us find the elements whose frequent is more than a defined threshold in a stream, in this way, maybe we can know the main content of this stream. As we know, entropy is a significant information for a data stream, it represents the amount of information of this stream. So through “ent”, we

can know the entropy of a data stream from the network, and use this information to some corresponding treatments.

6. ACKNOWLEDGEMENTS

Thanks Mr. Frédéric Pennerath for proposing the project and tutoring.