ETH Zürich D-ITET

P&S Genome Sequencing on Mobile Devices HS2022

SneakySnake

Lorenzo Lazzaroni, Joël Lindegger

Index

Abstract

Genome analysis needs to be more intelligent nowadays. Our genomes need to be read, analyzed and interpreted not only quickly and efficiently, but also accurately. The genome analysis needs to be faster, in order to achieve an improved genetic disease prevention, and more intelligent for various reasons, such as privacy preserving, rapid surveillance of disease outbreaks, decreased cost of hospitalization.

In this presentation I will briefly introduce what genome sequencing means, what problems this implies and the existing techniques to accelerate it. After that I will show in detail the work done with the algorithm *SneakySnake*.

1- Introduction

The DNA which is contained in all of our cells is a polymer composed of two polynucleotide chains; there are four different kinds of polynucleotide, and their order carries the information for the production of proteins. Sequencing DNA means identifying the exact sequence of polynucleotides. Unfortunately there are no machines which allow to read the entire DNA sequence; what they can do instead, is read smaller subsequences, called reads. However, there is no information about the location of those reads, so they need to be sticked together through comparison with a so-called reference genome in order to get the complete sequence.

Of course comparing each read to the whole reference genome would be very slow, so mapping a read consists of three steps: indexing the reference genome (mostly through hashing), querying the index using subsequences of the reads, and verifying whether the mapping was right by calculating how much two strings differ from each other.

This last step is what makes read mapping slow, and needs to be accelerated. This is achieved through pre-alignment filtering techniques, which come before the alignment step, and read alignment acceleration.

2- Pre-alignment filtering techniques and SneakySnake

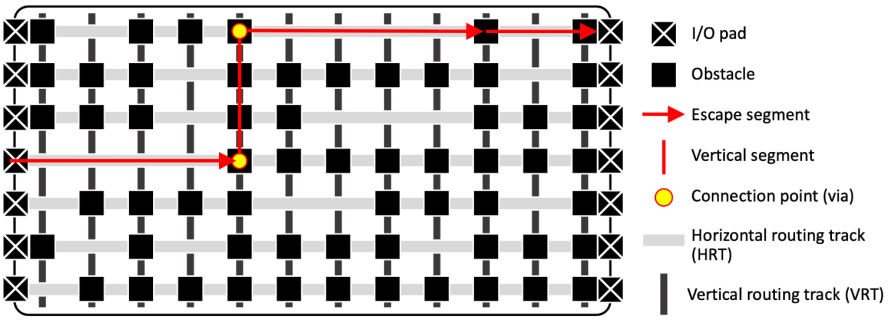
The goal of the pre-alignment filtering techniques is filtering out most of the incorrect mapping with cheap computation, in order to reduce the workload of dynamic programming, which will be used in the read alignment step. Of course the correct mappings need to be preserved, so underestimating the edit distance is acceptable, but not overestimating it.

SneakySnake is one of these pre-alignment filtering algorithms and remarkably reduces the need for computationally costly sequence alignment. The key idea is to reduce the *approximate string matching* (ASM) problem to the *single net routing* (SNR) problem. While the ASM problem means calculating the edit distance, the type of each edit and the location of each edit, the SNR problem aims to find the shortest path by passing through the minimal number of obstacles. Therefore, there is no

need for keeping track of the subpath and the independence of the subpaths can be exploited to solve many SNR problems in parallel.

Practically the *SneakySnake* algorithm consists of taking a string from the reference genome and one read, then shifting the read and calculating the length of the subsequence of characters that match. This operation is reiterated over all the possible shifts (from the shifting of *-E* characters to the shifting of *E* characters), and the longest subsequence is returned. After that, the edit distance between the two strings increases by 1 (which means that there is one obstacle), and the operation of finding the longest subsequence is repeated until the sum of all sizes of all the subsequences is equal to the size of the string from the reference genome.

The output of the *SneakySnake* algorithm is this kind of edit distance between the two strings. This computation is very cheap compared to the ASM problem. Furthermore not every position of the grid needs to be computed, as once an obstacle is found, everything which comes afterwards up to the size of the longest subsequence can be ignored. One of the inputs is the desired maximal edit distance, and as soon as the number of obstacles reaches this number, the function can stop and output -1.



Example of a grid coming from the SneakySnake *algorithm.*
Each row corresponds to the shifting of a certain number of characters

3- SneakySnake: my implementations

My first implementation of the *SneakySnake* algorithm is its standard form, which takes two strings, compares them character per character with the various shifting offsets and outputs the edit distance between the two strings. The inputs are the two strings and an unsigned integer, which is the maximal shifting offset accepted. We decided no to consider the maximal accepted edit distance, as the goal was to compare the runtimes of the algorithm in its different forms, so the function never has -1 as output.

```cpp
int SneakySnake(const std::string& query , const std::string& reference , const int& E) {

  int checkpoint = 0; //as soon as checkpoint reaches query.size() the function outputs its value
  int PropagationDelay = 0; //this is going to be the output of the function

  unsigned long m = query.size();

  while (checkpoint < m) {


    int count = MainHRT(query , reference , checkpoint); //this function compares the two strings
                                                         //without any shift
    if (count == m) return PropagationDelay;


    int longest_es = std::max(UpperHRT(query , reference , checkpoint , E) ,
                              LowerHRT(query , reference , checkpoint , E));

    //UpperHRT compares the strings with a positive shift, LowerHRT compare the string with a
    //negative shift


    if (count > longest_es) longest_es = count;

    checkpoint += longest_es;

    if (checkpoint < m) {
      PropagationDelay++;
      checkpoint++;
    }

  }

return PropagationDelay;
}
```

First SneakySnake *implementation*

My second implementation of *SneakySnake* is called *SneakySnakeMat*. This function takes as input not the two strings anymore, but a precomputed matrix. This kind of algorithm is slower than the first one. On the first hand the computation of the matrix takes longer, because every comparison is done, while before some of them were not necessary. However, we considered this as a precomputation, so we assumed that this is done before, and it does not have to be included in the runtime of the algorithm. On the other hand, memory access with a matrix takes longer than with two strings, so the runtime is still going to be larger.

```cpp
int SneakySnakeMat(const matrix& table) {

  int checkpoint = 0;
  int PropagationDelay = 0;

  unsigned long m = table.at(0).size();

  while (checkpoint < m) {

      checkpoint += longest_path(table , checkpoint);
      //longest_path is a function which outputs the longest matching subsequence
      //starting from the checkpoint.

      if (checkpoint < m) {
          checkpoint++;
          PropagationDelay++;
      }
  }

return PropagationDelay;
}
```

Second SneakySnake *implementation*

```cpp
matrix string_table(std::string a , std::string b , unsigned int e) {

    matrix table(2 * e + 1 , std::vector<bool>(a.size()));

    int r = e;

    for (unsigned int i = 0 ; i < table.size() ; i++) {
        for (unsigned int k = 0 ; k < table.at(0).size() ; k++) {
            if (k - r >= 0 && k - r < table.at(0).size()) {
                if (a.at(k - r) == b.at(k)) {
                    table.at(i).at(k) = true;
                }
            }
        }
        r--;
    }

    return table;
}
```

Precomputation of the matrix

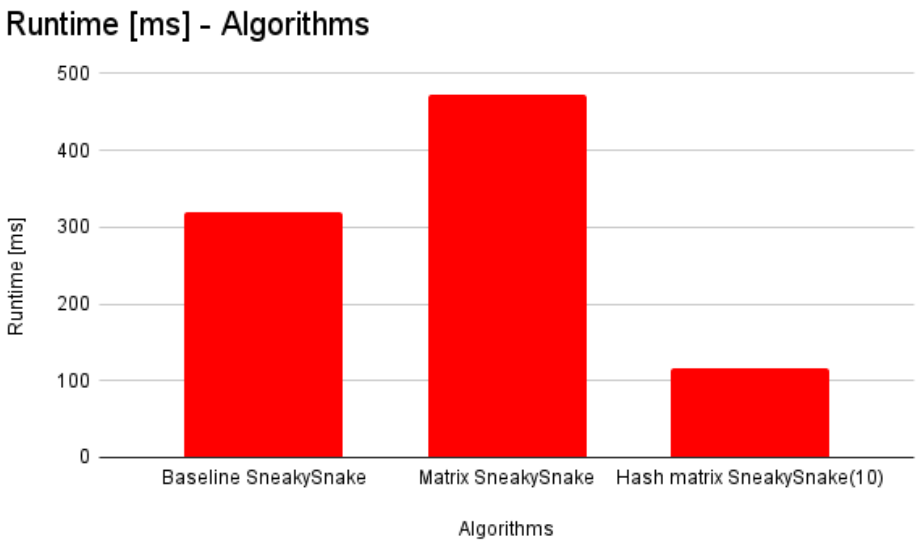
My last implementation of the *SneakySnake* is the same as the second implementation. What changes is the way the matrix is precomputed. This implementation includes a hash function, which takes a string of n characters and hashes them together to an unsigned integer. The string from the reference genome is divided into substrings, and the hash function is applied to each one of these substrings. The read is divided into substrings too, which are hashed together into unsigned integers, but the starting and the ending points are again shifted. This leads to a greater efficiency, because the matrix is going to be much smaller.

```
matrix string_table_murmur(std::string a , std::string b , unsigned int e , unsigned int c ,
                           unsigned int murmur2) {

    matrix table(2 * e + 1 , std::vector<bool>(a.size() / c));

    unsigned int r = e;

    for (unsigned int i = 0 ; i < a.size() ; i += c) {
        for (unsigned int j = 0 ; j < table.size() ; j++) {

            if (i + c <= a.size() && i + c + r <= a.size() && i + r >= 0 && i + r < a.size()) {
                std::string a_little;
                std::string b_little;

                for (unsigned int k = 0 ; k < c ; k++) {
                    b_little.push_back(b.at(i + k));
                    a_little.push_back(a.at(i + k + r));
                }

                uint32_t hash_1 = murmur3_32((uint8_t *)a_little.c_str() , c , murmur2);
                uint32_t hash_2 = murmur3_32((uint8_t *)b_little.c_str() , c , murmur2);

                if (hash_1 == hash_2) {
                    table.at(j).at(i / c) = true;
                }
            }
            r--;
        }
        r = e;
    }

    return table;
}
```

Precomputation of the matrix using the hashing function

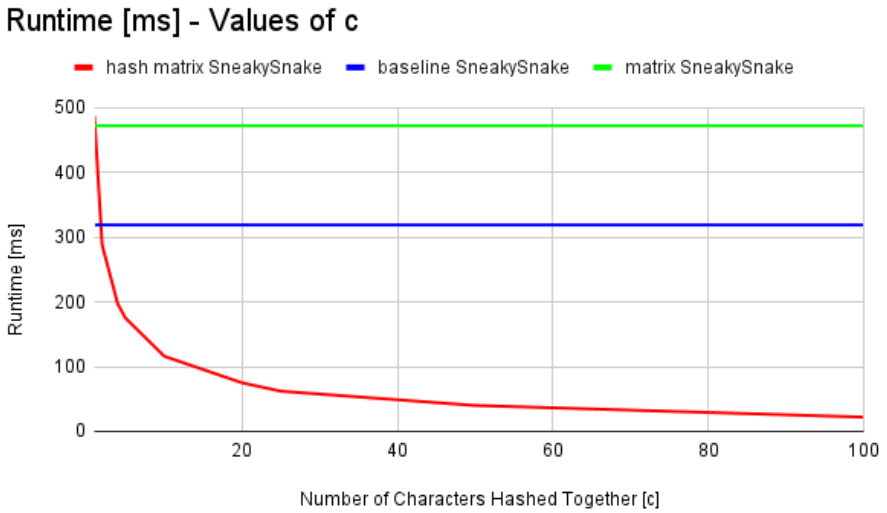
-Runtime comparisons

Taking a file composed of 30 000 pairs of strings of 100 characters and measuring the runtimes, as expected the slowest algorithm is the *SneakySnakeMat*. The standard *SneakySnake* comes right after, while the *SneakySnakeMat* hashing 10 characters together is the fastest.



This graph shows the comparison of the algorithm runtimes

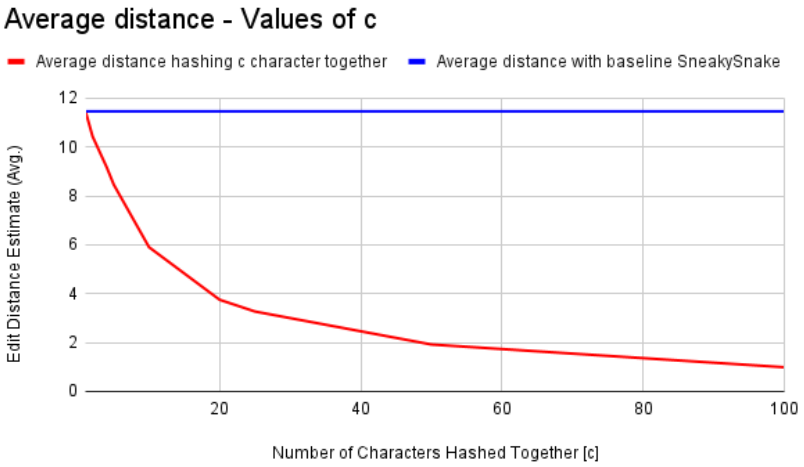
It is also interesting to see the runtimes of the last *SneakySnake* implementation compared, hashing together different numbers of characters. As the strings contain 100 characters, we chose to hash together *n* characters, such that *n* is a divider of 100 (i. e. 1, 2, 4, 5, 10, 20, 25, 50 and 100). As expected, with 1 character hashed together, the runtime is the same as the *SneakySnakeMat*, and the runtime decreases when more characters are hashed together.



This graph shows how the runtime decreases when more characters are hashed together

-Edit distance comparisons

Last but not least, the correctness of the algorithms must be controlled. Of course hashing more characters together, but still increasing the edit distance only by 1, when an obstacle is found, will lead to underestimating the edit distance. However, this is not a problem according to the preconditions of the pre-alignment filtering. The final result would be filtering out less reads (while the correct one is still preserved), but the computation would be much faster. To quantify the correctness of the algorithm I computed the average edit distance resulting from the various computations (summing the results of the 30 000 computations and dividing by 30 000). Assuming the *SneakySnakeMat* hashing one character together is correct, as expected the average edit decreases when more characters are hashed together.



This graph shows the average edit distance compared to the number of characters hashed together

4- Conclusion

During this P&S I have had the possibility of seeing how what we learn in our informatics course can be used for real problems, and I have experienced what research in this area could look like. I have written three different versions of the same algorithm, comparing the advantages and the disadvantages of each of them, still keeping in mind the big picture of the complete genome analysis. Last but not least, I have come to handle larger amounts of data than usual, measuring how much time is needed to analyze them.