Nanopore *de novo* sequence alignment using CUDA and De Bruijn graphs

Jakub Bilski, Jakub Brojacz

June 2020

Abstract

Motivation We examine the possibilities of using a GPU unit installed on a casual device to perform a full *de novo* genome assembly on input data from Oxford Nanopore's MinION, while utilizing De Bruijn graphs in the assembly process. We try to overcome extensive memory requirements problem using solid-weak k-mer classification and correction of weak fragments.

Results Using an approach similar to the one from LORMa, we were unable to reduce the graph's size enough to fit in a GPU memory for high values of k. We created De Bruijn graph representation and examined its memory requirements. We also discussed bad prospects of parallelization of the above approach.

1 Introduction

1.1 DNA sequencing

Knowledge of DNA sequences has become essential for numerous applied fields, especially medical diagnosis and medical treatment. Comparing healthy and mutated DNA sequences can diagnose different diseases including various cancers, characterize antibody repertoire, and can be used to guide patient treatment. The process of determining the order of nucleotides in DNA is called DNA sequencing.

1.2 Genome assembly

In many cases, the whole genetic material (genome) of an organism needs to be known to utilize a specific technique. Genome assembly is the computational process of deciphering the sequence composition of the whole genetic material, using numerous short sequences called reads derived from different portions of the target DNA as input. Full genomes often consist of billions of base pairs (adenine, guanine, cytosine, and thymine), and successful genome assembly method needs to combine enormous numbers of reads in one sequence, while

retaining a reasonably low error rate. Computational challenges that arise from this task are particularly interesting from the perspective of computer science.

1.3 Oxford Nanopore

Nanopore sequencing is a sequencing approach based on transporting an unknown sample through a pore of nanometer size and measuring the electric current flowing through. The magnitude of the current density can be used to predict the composition of DNA occupying the nanopore. The big advantage of this approach stems from its non-destructive qualities and real time performance. The method does not require nucleotides to be modified in chemical labeling or PCR amplification, and indivdual reads can be as long as 10000 bases average. Our test data was produced with MinION, a portable device that uses nanopore technology, produced by Oxford Nanopore. Our test file consists of reads with 8750 average length, with a 30 times coverage. One of the drawbacks of the nanopore approach is a relatively high error rate, and it is also present in the MinION output - the average possibility of the occurrence of an error (deletion, insertion or substitution) equals 15% for every base. To aggregate the nanopore reads into a quality assembled genome regardless, a sequencing process must overcome the problem of errors. This can be achieved using different assembly techniques. One of them is based on the usage of De Bruijn graphs.

1.4 De Bruijn graphs

De Bruijn graph is a directed graph that represents overlaps between sequences of symbols. A single vertex in the graph represents a single k-mer, which is a sequence of k bases. If one of the vertices can be expressed as another vertex by shifting all its symbols by one place to the left and adding a new symbol at the end of this vertex, then the latter has a directed edge to the former vertex.

$$AATCG \rightarrow ATCGC \rightarrow TCGCT$$

One can see that, for a fixed k, any read sequence consisting of n bases can be represented as a path in a De Bruijn graph, possibly with some recurring vertices and edges, as it is obviously possible for a single k-mer to occur many times in a read. De Bruijn graphs are widely used for genome assembly and it was our goal to examine the possibilities of using them in a solution that utilizes GPU to speed up the computations.

1.5 Existing CPU solutions

With long-read high-error sequences produced by third generation technologies, including nanopore devices, a new challenge of long-read correction arose, as opposed to short-read correction performed before. Two major ways of approaching long-read correction were developed. The first one, hybrid correction, makes use of additional short reads data to perform correction. The second one,

self-correction, on the contrary, attempts to correct long reads solely based on the information contained in their sequences. There are multiple CPU solutions for DNA sequencing, but most of them belong to the first group. The only two self-correction CPU solutions based on De Bruijn graphs that we were able to find were LORMa and DACCORD. We make use of some of the LoRMA main concepts in attempts to decrease the amount of the memory required.

2 Solution overview

2.1 Parsing the input data

Data generated by MinION is stored in fastq format, a text-based format for storing both a biological sequence and its corresponding quality scores. As we did not plan to make use of the latter, we chose to ignore it and load into the data structure only the bases from the read sequences. Parsing a 21GB file into a data structure efficiently was a demanding task nonetheless.

2.1.1 Loading the file

We load the file into memory piece by piece using std::fstream.read() in the binary mode, which proved to be a faster way than stream access or block access in the text mode. We do not use file mapping, as Windows does not support it. The size of a single file part residing at once in the program memory is controlled with a variable to make it possible to run the program on devices with a different RAM capacity.

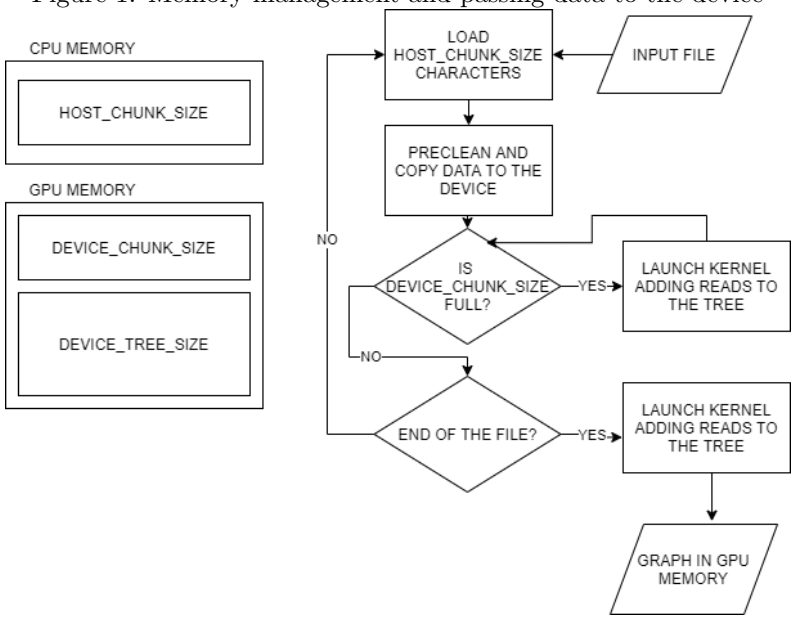
2.1.2 Jumping through the fastq

Fastq format ensures that the quality scores are of the same length as the base sequences. That gives us the opportunity to skip the quality info, since after reading the sequence we already know its length. This is also the main reason the block access is superior to the stream access, as skipping a number of characters in the former is much faster.

2.1.3 Precleaning data

We want to perform De Bruijn graph creation inside the GPU device to speed up the process. However, since we can skip parts of the file during reading, loading the whole file into the GPU memory would be unnecessary. To avoid this, we copy only the base sequences gathered by CPU, separating them with a newline, and launch the kernel only after there is no more free memory left on the device. This ensures that the kernel launches are performed as rarely as possible, while intertwining them with the CPU computations. Additionally, to simplify the graph creation, we copy to the device only fully gathered reads - if some part of a read do not fit into the memory, we pass it on to the next kernel launch.

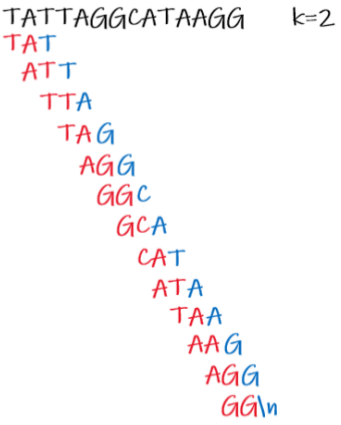
Figure 1: Memory management and passing data to the device



2.2 Construction of De Bruijn graph

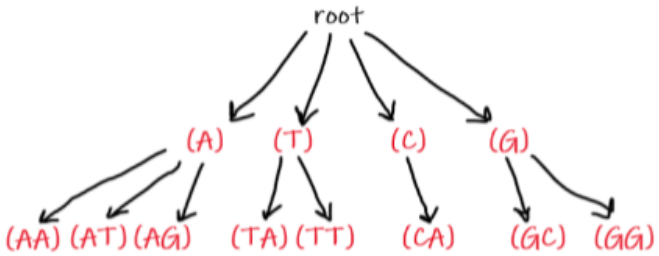
2.2.1 Data structure

In all of the figures in this subsection we assume a De Bruijn graph is created from the following sequence

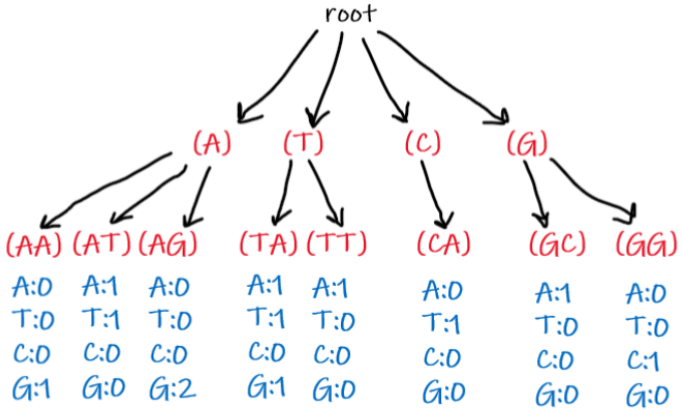


We use our own representation of De Bruijn graph. In order to minimize the memory required, we decided to represent the graph in a form of a 4-ary tree,

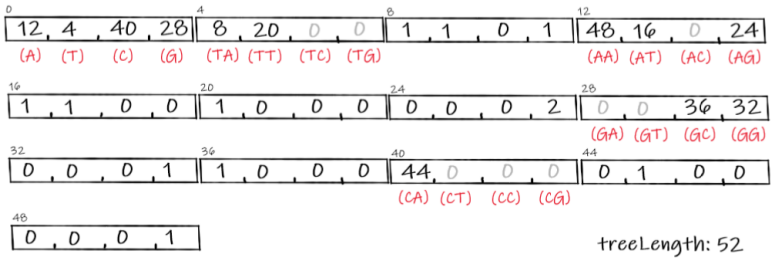
with each leaf representing one k-mer.



In each leaf, there are 4 values, representing the weights of all four possible edges, leading to the k-mers that are connected to the given k-mer in the graph.



In order to use this representation in CUDA kernels, it is necessary to save it in a one-dimensional array. In our representation, the array was divided into fragments consisting four cells each. Number in each cell indicate the array index of the successor node, in ATCG order. If the number in cell is equal to 0, the given node doesn't exist. The whole structure is created in parallel while reading data, and no memory is wasted on non-existing nodes in a tree.



In the leaf nodes, the numbers in cells represent weights of the outgoing edges. This property of the representation changes at a certain point, when high enough edge weights get replaced by pointers to the successor leaves to speed up graph traversing, and others are set to -1. This step is discussed in part 2.3.

2.2.2 Estimated memory requirements

Memory complexity of the representation is $O(4^k)$, for a sequence of n reads analyzed as k-mers. The approximate amount of required memory can be expressed as

$$neededInts \approx 4 \cdot (4^k + 4^{k-1} + ... + 1) \cdot coverage,$$

where coverage is the number of k-mers present in the data divided by the number of all possible k-mers. One of the advantages of using De Bruijn graphs for DNA sequencing is that for k-mers of higher order coverage drastically declines. However, in case of our data, to witness lower coverage values, we need to make sure we even reach high numbers of k and not run out of memory in the process. To visualize this problem, figure 2 shows the amount of memory needed by the De Bruijn graph created with our data.

Figure 2: Tree memory needed, naive approach

k	tree size [MB]	coverage
8	1	100
9	5	100
10	21	100
11	85	100
12	329	99,99
13	1354	99,23

The amount of memory grows very fast with increasing k, and already for $k = 15$ it is impossible to contain in a typical GPU memory. To face this problem, we need a more efficient data structure or some way of ruling out some of the k-mers. In LORMa, both approaches are implemented, as it uses

a Bloom filter to store k-mers, and performs a solid-weak classification to rule out k-mers with a high probability of being products of errors. We chose to implement the latter, as we thought it would be more beneficial.

2.3 Solid-weak classification

In our solution, we try to use LoRMA methods of intermediate correction of reads in order to contain De Bruijn graphs of higher order in a device memory. The first step of the algorithm is unifying the file format. Then, we perform the construction of a De Bruijn graph from the input reads and use this graph to correct the reads (strange as it may seem, correcting the reads using the very same reads is the fundamental idea in *de novo* genome assembly). Then, the corrected reads become an input for another graph creation with a higher k and the process continues for three iterations, with k increasing in each iteration. After each correction some k-mers of higher order should disappear.

2.3.1 Weak leaves deletion

In order to recognize parts of the reads to correct, we introduce terminology of strong k-mer and weak k-mer. Weak k-mers are all k-mers that do not exist in De Bruijn graph or those whose number is lower than the threshold. Strong k-mers are all k-mers which are not weak.

2.3.2 Threshold calculation

Threshold in LoRMA is a number of occurrences that should separate k-mers that were created solely as products of errors from the ones that were actually present in the reads. The expected value of a number of how many times a fixed k-mer of the genome is expected to occur in the reads is computed in a following way

$$E(C_{l\geq k}) = \frac{N}{G}\sum_{i=k}^{\infty} q^2(1-q)^i(i-k+1)$$

where N is the length of a concatenation of all the reads, and $q = (pN + n)/(N + n) \approx p$, where p is the error rate. Because of the characteristics of this distribution discussed in the LoRMA paper, the standard deviation can be expressed as

$$\sigma(C_{l\geq k}) = \sqrt{E(C_{l\geq k})}$$

Threshold values $h \in [E(C_{l\geq k}) - 2\sigma(C_{l\geq k}), E(C_{l\geq k}) - \sigma(C_{l\geq k})]$ should guarantee that the rejected k-mers have a high chance of being products of errors.

2.3.3 Rejecting k-mers

After construction of the full graph, every leaf is examined. If sum of the weights of four outgoing edges is lower than the threshold, all four weights are set to

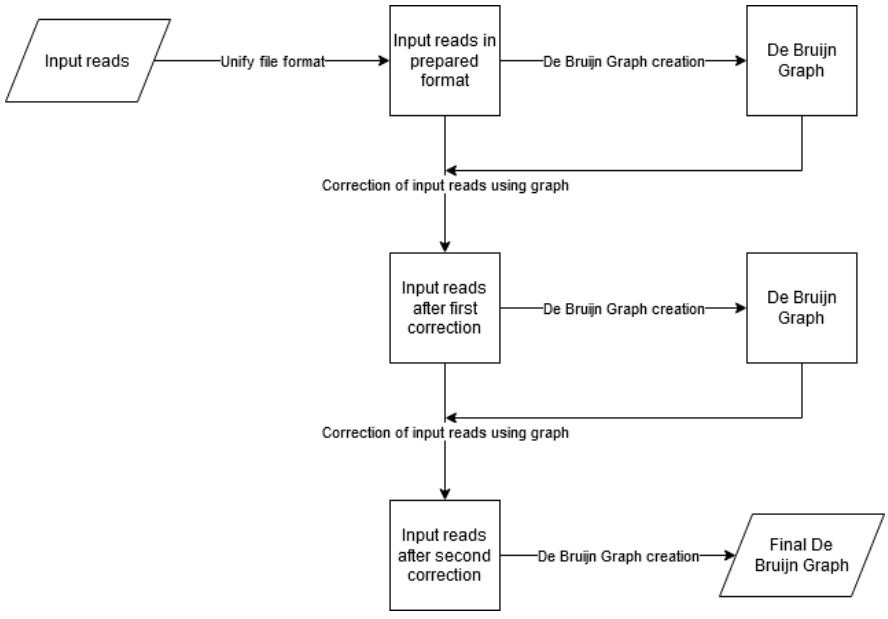
-1. Otherwise, every edge weight is replaced with an index of the node that the edge was leading to, to allow for a fast graph traversing during the error correction. The whole process is highly parallelized and performed solely on the GPU.

2.4 Error correction method

During the error correction phase, for each read we detect weak fragments surrounded by strong k-mers, named head and tail of the weak fragment. Then, we try to find the closest, based on the edit distance, path in the graph between head and tail. We use DFS to traverse the graph and use similar stop conditions as the original LoRMA solution. For calculating edit distance we use a simple dynamic programming approach with a two-row matrix. In the future, there is a possibility to use Levenshtein Automata for a faster calculation of the edit distance, although it would not affect the memory complexity.

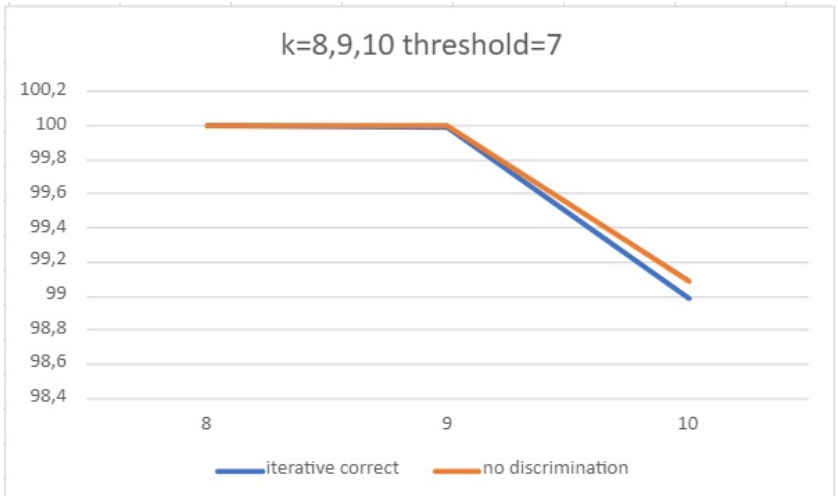
The whole phase of error correction is performed on CPU, as the goal of our work was to examine possibilities of facing the memory requirements, and not implement the full DNA sequencing on the GPU. We refer to the possibilities of parallelizing this part of an algorithm in the Conclusions section.

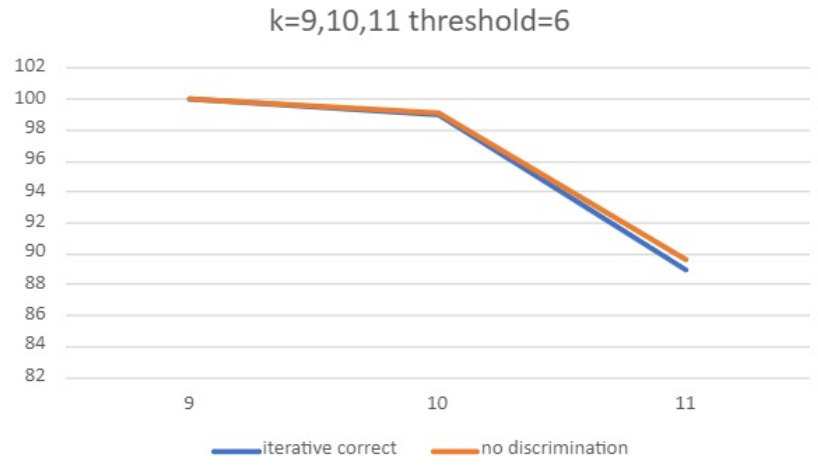
2.5 Workflow of the final approach

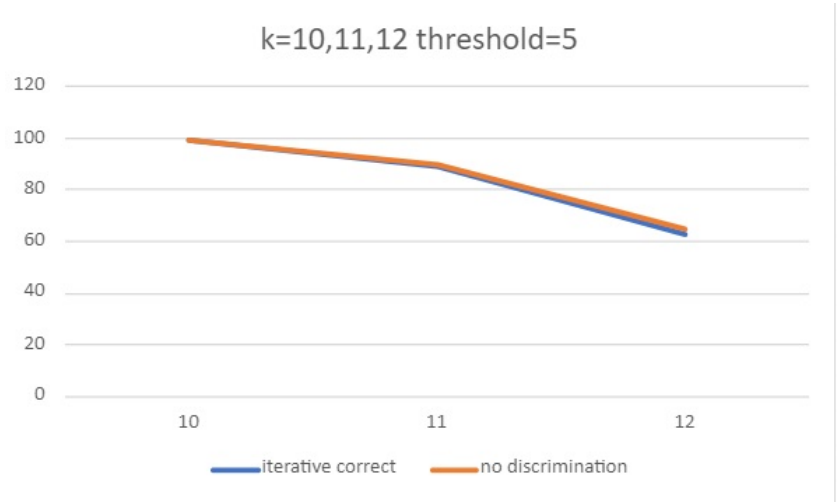


3 Experimental results

Regrettably, due to the fact that the most demanding task in our solution is performed on the CPU, it has not proven possible to perform a full process for high k values on a casual computer. Moreover, all of the experiments were performed on the first 100MB of the file. The highest we managed to get was $k \in \{10, 11, 12\}$, and this lasted for more than two days. For lower values of k, the benefits of the iterative correction process in terms of memory requirements are negligible.



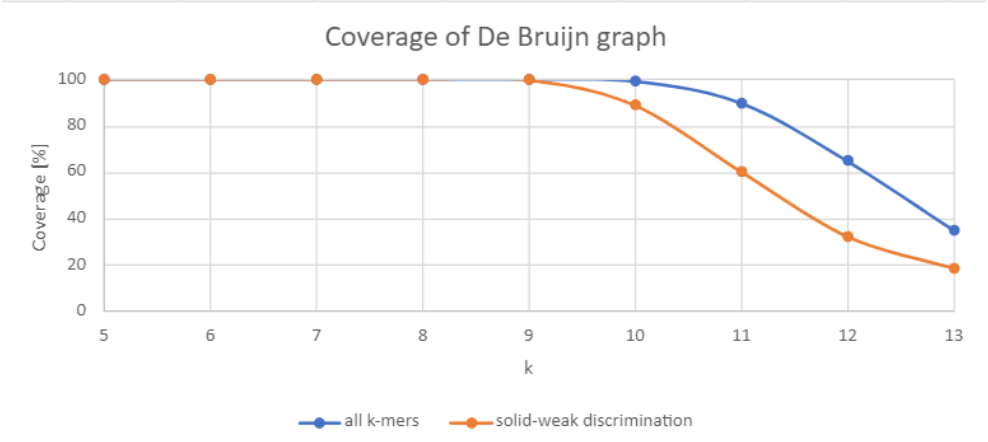




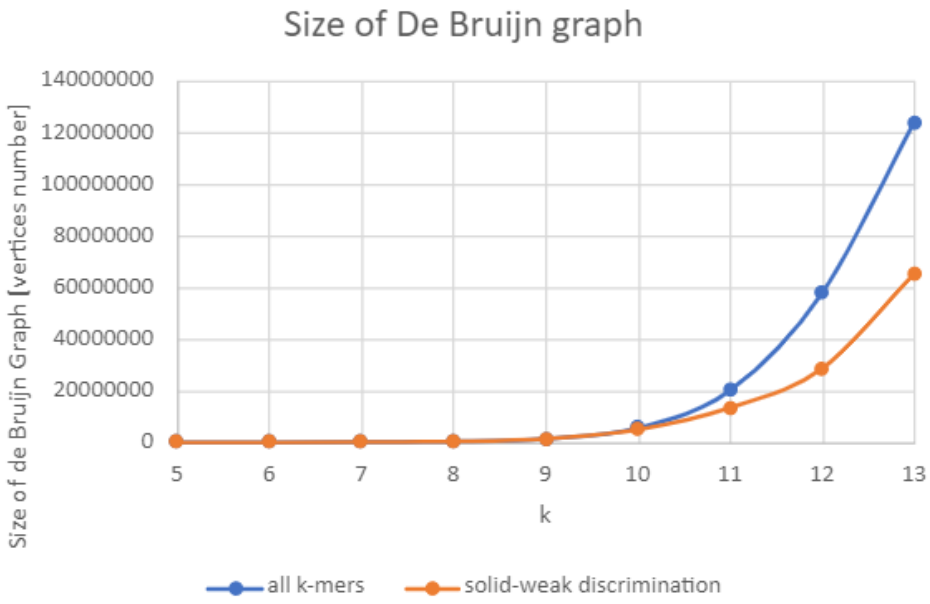
It should be noted, however, that after weak classification the weak leaves are not discarded from the graph, but marked, to avoid rebuilding the graph or costly deletions. Because of that, the influence of the deleted leaves can only be seen in the next iterations, and not in the one that the leaves belonged to. We found this approach sensible, as this does not affect the maximal memory requirements, since there's no need to shrink the size of the graph if it already fits into memory.

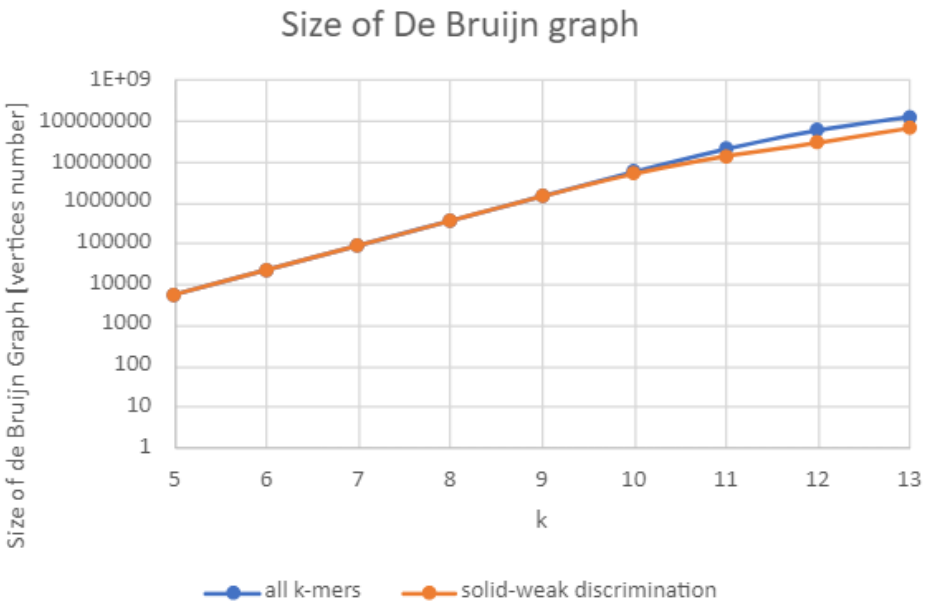
To try and measure the possible influence on the graph size regardless of the time restrictions, we performed a single k solid-weak classification, then checked the graph coverage that would result from deleting the weak leaves, and did not perform the weak fragments correction phase at all, as it is normally done only to affect the higher k steps. The threshold values were 5 for $k \geq 9$, and 9 for $k < 9$.

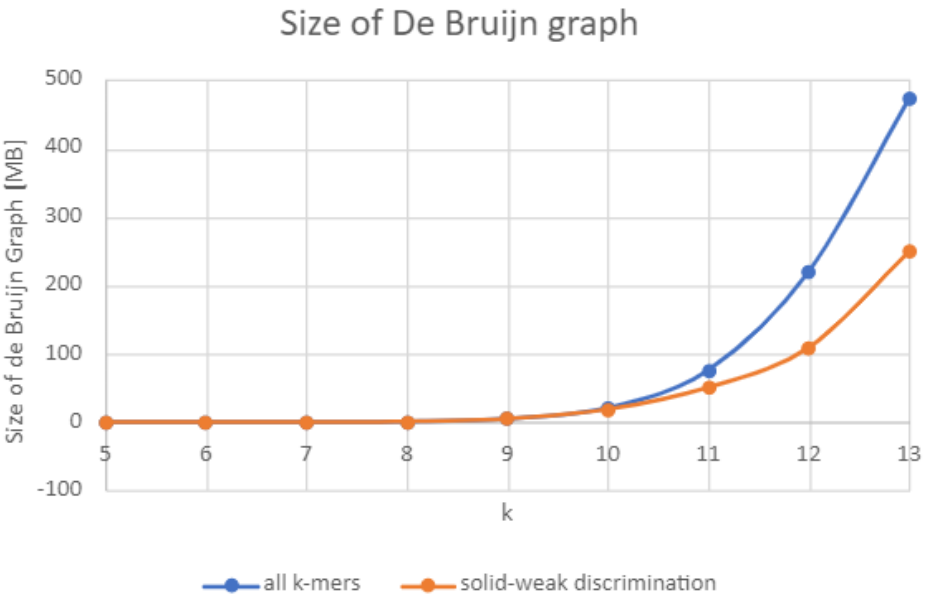
The coverage of De Bruijn graph decreased for larger ks



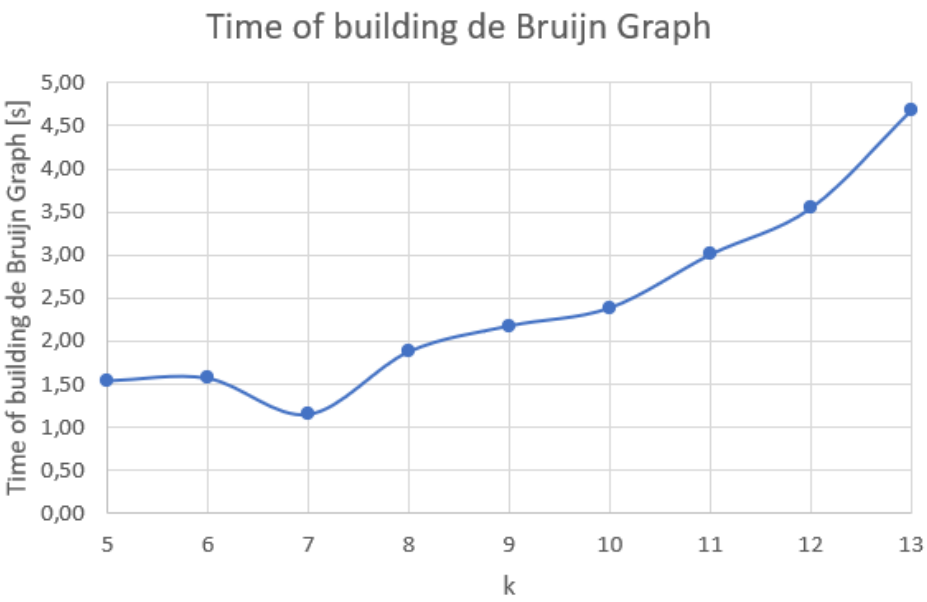
However, the required memory was still growing exponentially. The figures below show the relationship between k and the number of vertices in the constructed De Bruijn graph.



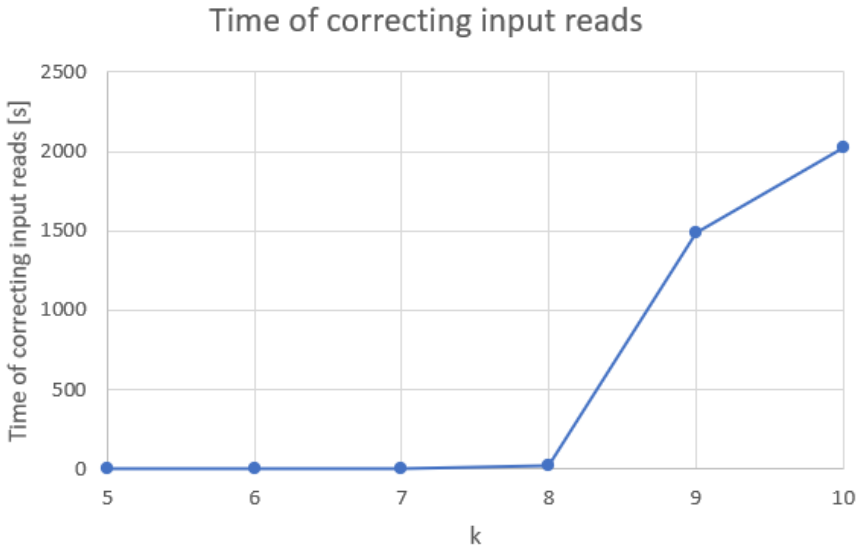




Due to the usage of GPU algorithms, the amount of time needed to construct the graph rose much slower than the required memory.



However, time of correcting the input reads using De Bruijn graph for k at least 9 was more than 25 minutes. This is not surprising, as it is the most computationally demanding part of the algorithm, and it was computed exclusively on the CPU.



4 Conclusions

4.1 Graph size

Despite the usage of the solid-weak classification and ruling out a sizeable part of the input k-mers, the sizes of the final De Bruijn graphs for $k > 15$ are still out of reach for a typical GPU memory.

4.2 Possibilities of parallelization

During the processing of the graph, the most demanding task in terms of computation was performed on the CPU. The method used to substitute the weak fragments with the solid ones uses dynamic programming, giving no possibilities of parallelization, as it has extensive memory requirements, and, what is worse, the amount of memory needed to correct a single weak fragment is impossible to predict before the run.

4.3 General

Ruling out k-mers via solid-weak classification and performing weak fragments correction is not a good way to meet the memory requirements of GPUs installed on casual devices. While achieving a speedup using GPU in combination with De Bruijn graphs in *de novo* DNA sequencing could still be possible, we are sceptical about it, as the two crucial requirements, predictable memory management and good prospects of parallelization, seem not to be met.

References

[1] Pierre Morisse, Thierry Lecroq, and Arnaud Lefebvre. Long-read error correction: a survey and qualitative comparison. *bioRxiv*, 2020.

[2] Leena Salmela, Riku Walve, Eric Rivals, and Esko Ukkonen. Accurate self-correction of errors in long reads using de Bruijn graphs. *Bioinformatics*, 33(6):799–806, 06 2016.

[3] German Tischler and Eugene W. Myers. Non hybrid long read consensus using local de bruijn graph assembly. *bioRxiv*, 2017.