BlastGraph: intensive approximate pattern matching in string graphs and de-Bruijn graphs

Guillaume Holley, Pierre Peterlongo

BlastGraph: intensive approximate pattern matching in string graphs and de-Bruijn graphs

Guillaume Holley[1], Pierre Peterlongo[1*]

Centre de recherche INRIA Rennes - Bretagne Atlantique, IRISA, Campus universitaire de Beaulieu, Rennes, France
guillaumeholley@gmail.com, pierre.peterlongo@inria.fr

Abstract. Many de novo assembly tools have been created these last few years to assemble short reads generated by high throughput sequencing platforms. The core of almost all these assemblers is a string graph data structure that links reads together. This motivates our work: BlastGraph, a new algorithm performing intensive approximate string matching between a set of query sequences and a string graph. Our approach is similar to blast-like algorithms and additionally presents specificity due to the matching on the graph data structure. Our results show that BlastGraph performances permit its usage on large graphs in reasonable time. We propose a Cytoscape plug-in for visualizing results as well as a command line program. These programs are available at http://alcovna.genouest.org/blastree/.

Keywords: string graph, de-Bruijn graph, string matching, high throughput sequencing, next generation sequencing, sequence assembly, Viterbi algorithm

1 Introduction

Compared to traditional Sanger technologies High Throughput Sequencing (HTS) technologies enable sequencing of biological material (DNA and RNA) at much higher throughput and a cost that is now affordable by most academic labs. They have revolutionized the field of genomics and medical research [4]. Sequencing became in a few years accessible to almost all biological labs while being able to produce sequences of full complex genomes in a few days.

HTS technologies do not output the entire sequence of a DNA or RNA molecule. Instead, they return small sequence fragments, called reads, whose length is ranging between 100 to 700 characters. HTS produce overlapping reads, thus making possible to reconstruct the original sequence by assembling them. Over the last few years, many assemblers were created, such as Euler [2, 3], Velvet [11] or Soapnovo [6] to cite a few among the most famous ones. They present different capacities and drawbacks, but all of them make use of a string graph (SG) for organizing the reads. For assembly, the most used graph is the **de-Bruijn graph** (DBG), first proposed for assembly purposes by Pevzner, Tang and Waterman [8]. In a DBG a node represents a length-k substring (called a

[*] Corresponding author

k-mer) and an edge connects nodes u and v if the two corresponding k-mers overlap over $k-1$ positions. Once the graph is created and usually after an error correction step, a traversal of the graph is performed for generating contiguous sequences called *contigs*.

In this paper, we present BlastGraph, a generic approach for aligning a (possibly large) set of query sequences on a SG or a DBG. Motivations for this work are multiple. For developers of assembly tools, it is of great interest to precisely detect query sequences in the graph, for instance while testing filter algorithms or correction algorithms. Biologically, checking the presence of approximate copies of a set of sequences in the graph, enables to detect homologies, to filter contaminants, to detect the presence of species. Avoiding the full assembly process presents two main advantages: first it avoids the time consuming contig generation phase, and second and more important, it avoids the usage of heuristics or statistical choices made while traversing the graph.

Note that the BlastGraph algorithm applies generically to any directed SG, and is also adapted to apply to a DBG. Given an oriented string graph, a set of query sequences and a maximal edit distance, BlastGraph detects paths in the graph on which query sequences align at most at the given edit distance. Our approach is a blast-like algorithm [1]. The graph is indexed using seeds, this enables to decrease the request execution time. The main originality of our work stands in the fact that both seeds and mapped query sequences may be spread over several nodes of the graph.

This work presents similarities with the famous Viterbi algorithm [10]. In a few words, Viterbi is a dynamic programming algorithm for finding the most likely path in a rooted graph while reading a query sequence. The major fundamental differences with this work stand in the fact that:

- Viterbi nodes are composed by a unique symbol while in the BlastGraph framework, nodes store a full sequence, and their reverse complement in the DBG framework;
- In the Viterbi framework, the alignment is global: the full query sequence is aligned to the whole graph, starting from the root node, while in the BlastGraph algorithm, the alignment is semi global: the whole query sequence is aligned to any un-rooted sub-graph.

The next Section introduces preliminaries and definitions. In Section 3 we expose the BlastGraph algorithm when applied on a SG, while in Section 4 we show how BlastGraph is modified to apply on a DBG. We present some practical results in section 5.

2 Preliminaries

A *sequence* is composed by zero or more symbols from an alphabet Σ. A sequence s of length n on Σ is denoted also by $s[0]s[1]\ldots s[n-1]$, where $s[i] \in \Sigma$ for $0 \leq i < n$. The edit distance between two sequences is the minimal number of insertions, deletions and substitutions to transform one into the other. The length of s is denoted by $|s|$. We denote by $s[i,j]$ the *substring* $s[i]s[i+1]\dots s[j]$ of s. In this case, we say that the substring $s[i,j]$ *occurs* at position i in s. We call *k-mer* a sequence of length k. If $s = u.v$ for u and $v \in \Sigma^*$, we say that v is a *suffix* of s and that u is a *prefix* of s, the symbol "." designating the concatenation between two sequences. Let $s[i..]$ denote the suffix of s starting at position i (*i.e.* $s[i..] = s[i, |s|-1]$).

The symbol "$\overset{\circ}{_k}$" designates the concatenation of two sequences, removing the first k symbols of the second. Formally, $u\overset{\circ}{_k}v = u.v[k..]$. In the DNA context, $\Sigma = \{A, C, G, T\}$, and, given $s \in \Sigma^*$, $\overline{s}$ designates the reverse complement of s, that is s, read from right to left, switching characters A and T, and C and G.

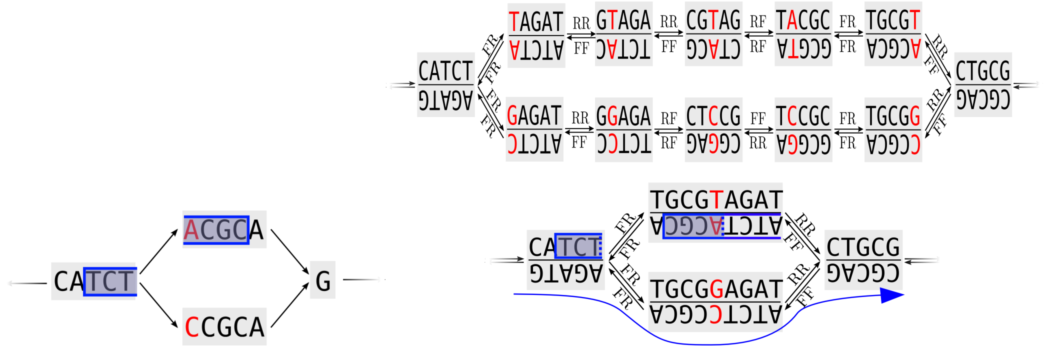
2.1 Directed string graph (SG)

In a directed string graph $\mathcal{G}$, each node N stores a sequence s, denoted by $S(N)$. A node N_1 linked to a node N_2 denotes the fact that the sequence $S(N_1).S(N_2)$ is stored in $\mathcal{G}$. Example of a directed string graph is given Figure 1a.

2.2 De-Bruijn Graphs (DBG)

DBGs were first used in the context of genome assembly in 2001 by Pevzner *et al.* [8]. In 2007, Medvedev *et al.* [7] modified the definition to better model DNA as a double stranded molecule. In this context, given a fixed k value, a DBG is a bi-directed multigraph, each node N storing a k-mer s and its reverse complement $\overline{s}$. The sequence s, denoted by $F(N)$, is the forward sequence of N, while $\overline{s}$, denoted by $R(N)$, is the reverse complement sequence of N. An arc exists from node N_1 to node N_2 if the suffix of length $k-1$ of $F(N_1)$ or $R(N_1)$ overlaps perfectly with the prefix of $F(N_2)$ or $R(N_2)$. Each arc is labelled with a string in $\{FF, RR, FR, RF\}$. The first letter of the arc label indicates which of $F(N_1)$ or $R(N_1)$ overlaps $F(N_2)$ or $R(N_2)$, this latter choice being indicated by the second letter. Because of reverse complements, there is an even number of arcs in the DBG: if there is an arc from N_1 to N_2 then, necessarily, there is an arc from N_2 to N_1 (*e.g.* if the first arc has label FF then the second has label RR).

A DBG can be compressed without loss of information by merging *simple* nodes. A simple node denotes a node linked to at most two other nodes. Two adjacent simple nodes are merged into one by removing the redundant information. A valid path (see Definition 2) composed by $i > 1$ simple nodes is compressed into one node storing a sequence of length $k + (i-1)$ as each node adds one new character to the first node. Figure 1b represents a DBG (upper) and the corresponding compressed DBG (lower). In the remainder of the paper, we denote by cDBG a compressed DBG.



(a) SG

(b) DBG

Figure 1: **(a)** Directed string graph. **(b)** Uncompressed (upper) and compressed (lower) de-Bruijn Graphs with $k = 5$. For each node, lower text is the reverse complement of the upper text, it should be read from right to left. **Boxes** both on (a) and (b): example of a seed of length 7 (TCTACGC) spread over 2 nodes. In the de-Bruijn Graph, the $k-1$ first characters of the second node are pruned due to overlap, and the reverse part of the second node is considered as the edge between the first (left) and the second (central) node is F**R**.

Definition 1 (Active strand of a node in a DBG) *The active strand of a node N in a DBG denotes which strand of the node, forward or reverse, is considered while traversing N.*

Definition 2 (Valid path) *The traversal of a node N is said to be valid if the rightmost label (F or R) of the arc used for entering the node is equal to the leftmost label of the arc used for leaving the node.*

A path in the graph is valid if for each node involved in the path, its traversal is valid, that is, each pair of adjacent arcs in the path are labelled, respectively, XY and YZ with $X, Y, Z \in \{R, F\}$.

Definition 3 (Sequence stored in a cDBG) *A valid path in a cDBG composed by ordered nodes $N_0, N_1, \ldots, N_l$, stores two sequences as following:*

1. $s = F/R(N_0) \circ_k F/R(N_1) \circ_k \cdots \circ_k F/R(N_l)$*, the choice between R or F for node N_0 is equal the first label of the edge going from N_0 to N_1, while for $i \in [1, l]$, the choice between R or F for node N_i is equal the second label of the edge going from N_{i-1} to N_i.*
2. $\overline{s}$.

For instance, the arrowed path on the cDBG presented Figure 1b, stores the sequences

$$\begin{aligned} s &= CATCT_{\overset{\circ}{k}}ATCTCCGCA_{\overset{\circ}{k}}CGCAG \\ &= CATCT.CCGCA.G \\ &= CATCTCCGCAG \\ \text{and} & \\ \overline{s} &= CTGCGGAGATG \end{aligned}$$

2.3 Approximate pattern matching in a graph (SG or cDBG)

Definition 4 (Approximate pattern matching in a graph) *Given a query Q, a graph $\mathcal{G}$ (SG or cDBG), and a parameter d, approximate pattern matching consists in finding all occurrences of Q in sequences stored in $\mathcal{G}$ within an edit distance of at most d.*

3 The BlastGraph algorithm

Blast like seed-based heuristics rest on the idea that if two sequences share some similarities, then there exists (at least) a common word (a seed) between these two sequences. Such algorithms consist in, first, anchoring the detection of similarities by exact matching of short sub-sequences, the seeds, and then, performing the similarity distance computation once sequences are anchored. The algorithm we propose applies these ideas between a graph (the bank) and a string (the query). It is divided into four main stages:

1. Index all seeds present in the graph $\mathcal{G}$.
2. Anchor query sequences to nodes of $\mathcal{G}$ using seeds. In the case of genomic data, reverse complement of query sequences may also be used as queries.
3. Align anchored query sequences on the left and right of the matched seeds.
4. Merge left and right alignments.

In the four following sections, we provide some more details for each of these four stages simply considering the graph as a SG. Then, in Section 4, we describe the modifications needed for applying the algorithm on a cDBG.

3.1 Stage 1: Indexing the seeds

Let n denote the length of the seeds. Each word of length n of the sequence of each node of $\mathcal{G}$ as well as those spread over several linked nodes are indexed using a hash table. The index contains for each seed a set of its occurrence positions.

Occurrence position in a graph: An occurrence position in the graph is defined as a couple (node identifier N, position on $S(N)$) indicating the starting position of the occurrence.

Seeds spread over more than one node: Any seed starting at less than n positions to the end of the sequence of a node is spread over more than one node. For instance, the seed $TCTACGC$ starting at position 2 on the leftmost node of Figure 1a, is spread over two distinct nodes. Seeds spread over more than one node are detected thanks to a depth first algorithm recursive approach.

In order to make a light index, the BlastGraph algorithm only stores the starting position of a seed (node identifier N, position on $S(N)$) and not all possible nodes over which the seed is spread.

3.2 Stage 2: Anchoring query sequences to sequences of the graph

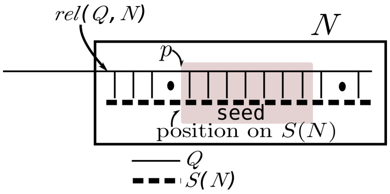


Figure 2: Value $rel(Q, N)$ while anchoring a query sequence Q on a node N with a seed.

For each query sequence Q, all overlapping words of length n (seeds) are read. Let s be such a seed occurring at position p on Q, also having at least one occurrence in the graph. Then the index provides a set of couples (node N, position on $S(N)$). For each such couple, the query Q is anchored on the sequence $S(N)$, giving a relative position $rel(Q, N)$ of Q on $S(N)$. More precisely, $rel(Q, N) = p-$position on $S(N)$ (see Figure 2) is the position where Q aligns to $S(N)$. Note that $rel(Q, N)$ could be < 0 if a prefix of $S(N)$ is not aligned to Q. This is the case of $S(Q, L_1)$ in the example presented Figure 3.

Computing an alignment only once: If a node N and a sequence Q share more than one seed for the same alignment, each of them generate the same value $\{rel(Q, N)\}$. As this is a very usual case, in order to avoid computing several times the same alignments, while aligning sequence Q, the values $\{rel(Q, N)\}$ is stored in memory. Thus, the same alignment anchored at position $\{rel(Q, N)\}$ is computed only once.

3.3 Stage 3: Alignment between query sequence and string graph nodes

Given a query sequence Q anchored at position $\{rel(Q, N)\}$ to a node N of the graph, this stage computes all possible alignments (based on edit distance) between Q and all paths readable from node N (see Figure 3 for an example).

Computing the edit distance between two strings is a dynamic programming procedure that involves the usage of a matrix of size the product of the string

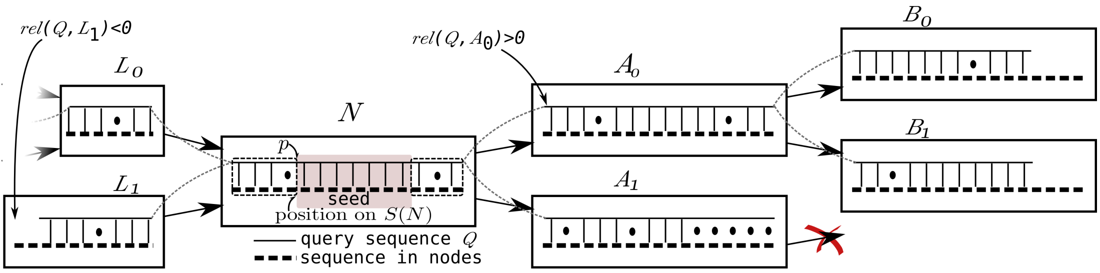


Figure 3: Overview of the alignment process. **Anchoring**: Using a seed, a query sequence Q is anchored to the node N. **Right alignment**: edit distance is computed between $Q[rel(Q,N)+i+k..]$ and $S(N)[i+k..]$ (right dotted square in node N), then between $Q[rel(Q,A_0)..]$ and $S(A_0)$, between $Q[rel(Q,B_0)..]$ and $S(B_0)$, between $Q[rel(Q,B_1)..]$ and $S(B_1)$, and so on. In this example, path using node A_1 presents an edit distance higher than the threshold; its children are not explored. **Left Alignment**: the same procedure is applied on the sequence on the left of the seed (left dotted square in node N), then on parents L_0, L_1 of node N, and so on ...

lengths. However, in the particular case of this work, the user restricts the maximal edit distance for having a match. Consequently, the matrix computation is limited to a diagonal (see Figure 4 for an example) of width $\left\lfloor \frac{maximal\ edit\ distance}{cost\ indel} \right\rfloor \times 2$. Outside the diagonal, number of insertions or deletions becomes bigger than maximal number of insertions or deletions accepted equal to $\left\lfloor \frac{maximal\ edit\ distance}{cost\ indel} \right\rfloor$. Thus during this stage, the time and memory complexity for aligning query Q to one path of the graph is in $O(|Q|)$ considering *maximal edit distance* and *cost indel* as fixed parameters.

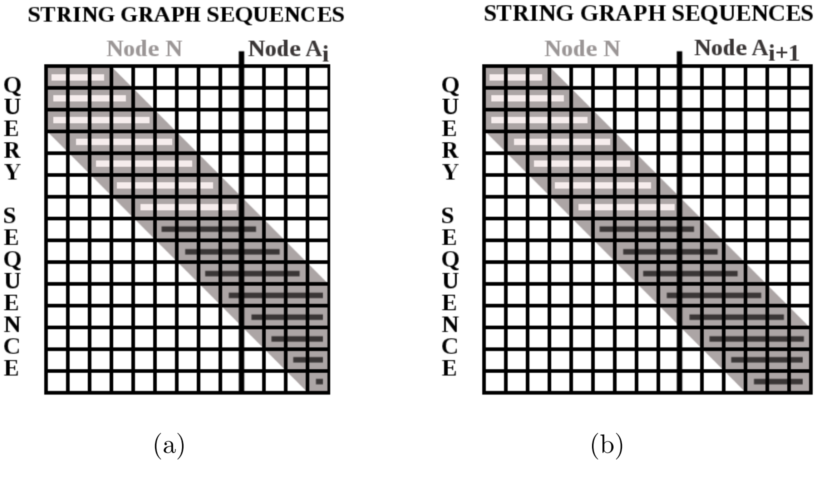


Figure 4: Dynamic programming matrix. Only the shadowed diagonal is computed. **(a)** distance computed between the query sequence S and $S(N.A_i)$. **(b)** distance computed between S and $S(N.A_{i+1})$. Lighter lines are not recomputed for computing matrix (b) if matrix (a) was already computed.

Right alignments The alignment is done between Q and $S(N)$ on the right of the matched seed. Additionally, as shown Figure 3, right extremity of the query sequence may finish after $S(N)$. In such a case the alignment has to be done on children $A_0, A_1, \ldots, A_n$ of node N. On each child A_i, the right extremity of the query sequence may finish after the $S(A_i)$, in this case, alignment continues on its children $B_0, B_1, \ldots, B_{n'}$, and so on. Thus, right part of sequence Q (starting after the anchored seed), may be aligned to $S(N.A_i.B_j....)$. This is done via a recursive depth-first traversal of the graph, starting from N as long as the full right part of S is not aligned. An alignment between the sequence of a node and Q is never computed twice. For instance (Figure 3), if the alignment between Q and $S(N.A_0.B_0)$ was computed, the computation between Q and $S(N.A_0.B_1)$ starts from the last full line of the alignment of Q with $S(N.A_0)$. Thus the alignment between Q and $S(N)$ and $S(A_0)$ is never recomputed.

Left alignments Aligning the part of the sequence Q on the left of the seed to the graph is done using almost the same approach as the one previously described for right alignments. However, there are two main differences: 1) Sequences both from Q and from the nodes are reversed (read from right to left); 2) when the reversed query sequence is longer than the reversed sequence of a node N, the parents $L_0, L_1, ...$ of N are explored in depth first search approach (see Figure 3 for an example).

3.4 Joining left and right alignments

For a given aligned query sequence, each left alignment is compared to each right alignment. For each such couple whose sum of the cost of the alignments is below or equal the user defined maximal edit distance, the full alignment is reported.

4 BlastGraph on compressed de-Bruijn graphs

The three main differences between the SG and the cDBG are:

1. In the cDBG, the sequences of two connected nodes overlaps over $k-1$ characters. Thus, whatever the stage, the concatenation of the sequences of two nodes of the cDBG, has to be done removing the $k-1$ overlapping characters using the "$\underset{k}{\circ}$" concatenation instead of the classical "." one.
2. In the cDBG, each node N stores a sequence ($F(N)$) and its reverse complement ($R(N)$).
3. Label of edges have to be considered while traversing the graph. Thus, in the cDBG, the general rule is the following: a node N is always traversed either as forward ($F(N)$) or as reverse complement ($R(N)$), with F or R being its "active strand" (see Definition 1).
   In the first case (resp. second case), accessing the children of the node is done following edges starting with the letter F (resp. R).

While following an edge, the active strand of the targeted node F (resp. R) is the second letter of the label of the edge.

Seeding in a DBG: The seeding approach is the same than the one applied on the SG. By convention, all seeds start on forward sequence of each node. This is done without loss of information as each query is considered both in its forward and its reverse complement directions.

Right extension in a DBG: The right extension in a DBG is the same as the one described for a SG. However, the algorithm takes into account some DBG specificities:
- query sequence is mapped on $F(N)$ (seeds are indexed only on forward strands);
- children of a node N are reached using only outgoing edges whose label first character corresponds to the active strand of N, and, once a child is reached, its active strand is the one corresponding to the second character of this label;
- concatenation of sequences of two linked nodes is done pruning the overlapping $k-1$ characters.

Left extension in a DBG: Left extensions in the DBG are done by right extending the reverse complement $\overline{Q}$ of the sequence Q to the DBG, starting from the reverse strand of the node N: $R(N)$.

5 Results

Two prototype versions of this algorithm are implemented. Under the CeCILL License, they can be downloaded here: http://alcovna.genouest.org/blastree. A Java version is implemented in a Cytoscape plug-in. Cytoscape [9] is an open-source platform for visualization and interaction with complex graph, especially in bioinformatics. The second version is implemented in C and can be run under Unix platforms. In the two prototypes, while working on nucleotides, characters are coded on two bits.

The next section proposes a use case of the Java version, while section 5.2 proposes some results over the C prototype.

5.1 Use case

We present in this section a use case, on a toy example. We created a string graph containing five nodes (Figure 5). We searched for the sequence

$$ggcgTtcagac/cTatacgcatacgcagcagact/agCctacg,$$

spreads over 3 nodes of this graph and containing two mismatches and one insertion. To help the reader, we indicated here substitutions and indel with an upper case letter and we indicated separations between nodes with a '/' character. Of

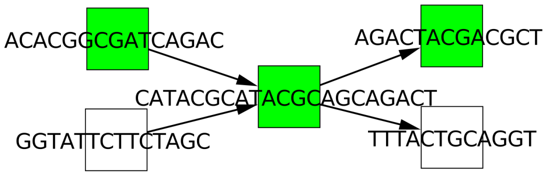


Figure 5: Cytoscape view of the selected nodes (green) in the string graph after the research of query sequence.

course the practical query sequence is a raw un-annotated sequence. We fixed the cost of a mismatch to 1, the cost of an indel to 2 and the maximal edit distance to 4. BlastGraph (Cytoscape plug-in version)) found the correct path, as presented Figure 5 where selected nodes are those in which alignment in found between the query and the graph.

5.2 Performances on DBG

We present results obtained in a typical use case while applying BlastGraph on a DBG graph. Results were obtained on a 64 bit 2x2.5 GHz dual-core computer with 3 MB cache and 4 GB RAM memory. From the NCBI Sequence Read Archive (SRA, http://www.ncbi.nlm.nih.gov/Traces/sra), we downloaded the DRR000096 Illumina run containing approximately 4 million reads and approximately 150 millions of nucleotides.

Increasing graph sizes Subsets of different sizes were generated by randomly sampling DRR000096 reads. For each subset, we constructed the de-Bruijn graph using k=31. Table 1 reports the total number of nodes and nucleotides stored in some of these graphs.

Nb Reads	Nb Nodes	Nb nucleotides
10K	59K	1833K
100K	573K	17774K
150K	849K	26306K

Table 1: Total number of nodes and nucleotides stored in the graph with respect to the number of reads.

On each graph, we applied the C version of BlastGraph, aligning a set of 10000 query sequences derived from the initial read set. We used seeds of length 19, a mismatch cost equal to 1 and an indel cost equal to 2 and a maximal edit distance equal to 5. We report Figure 6 time and memory needed both for constructing the index and for performing the 10000 queries.

We can observe that memory footprint and both indexation and query execution times increase linearly with the quantity of information contained in the

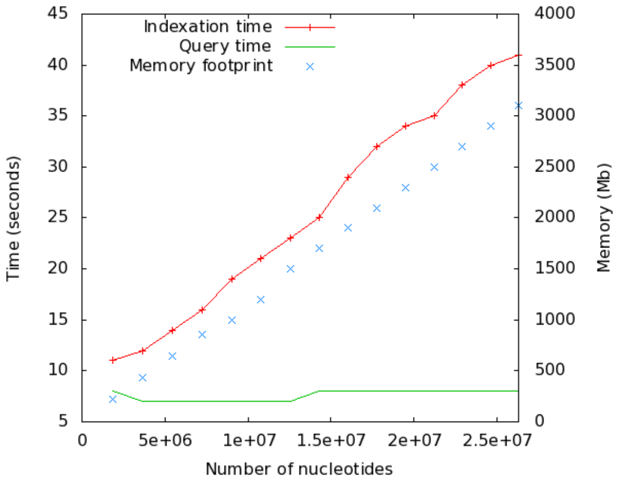


Figure 6: Time and memory consumption with respect to the number of nucleotides stored in the graph

graph. While memory usage is the main bottleneck of this approach, the indexation and query time are acceptable. Even on the biggest tested graph (containing more than 26 million characters stored in approximately 849000 distinct nodes), indexation is done in 43 seconds and the 10000 queries performed are in less than nine seconds.

Increasing number of queries In order to measure the impact of the number of queries on the execution time, we used the graph composed of 100000 reads from the DRR000096 data set using $k = 31$. We ran BlastGraph using queries dataset composed of 500, 1000, . . . 10000 reads taken from the 100000 reads used for creating the graph. We report the query time (not including the indexation time) Figure 7. We note that, as expected, the query time increases slowly and linearly with the number of queries.

6 Conclusion

We presented BlastGraph, a new algorithm for performing intensive approximate string matching between a set of query sequences and a string graph including the application to de-Bruijn graphs. This blast-like algorithm presents novelties with respect to classical blast-like approaches as seeds and alignments may be spread over several nodes and as the algorithm takes into account double stranded de-Bruijn graph features. Results showed that BlastGraph performances permit its usage on quite large graphs in reasonable time.

The main bottleneck of the approach comes from the memory footprint. Storing in memory graphs containing hundreds of millions of nucleotides together with seed index is challenging. Future work will include a non indexed version of

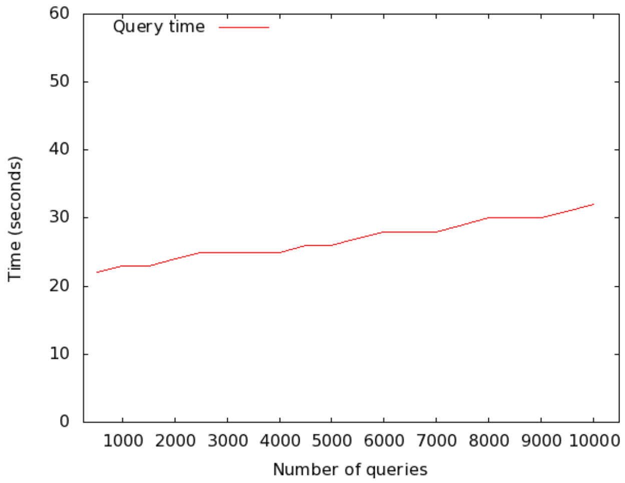


Figure 7: Query time with respect to the number of queries. Note that reported time do not include the indexation time equal to 22 seconds independently of the number of queries.

the algorithm, for instance based on KMP [5] algorithm. This will increase the query time, while decreasing the memory usage. Such a version would fit a unique or a limited number of query sequences. On the other hand, future work will also consist in modelling and implementing new data structures storing the graph together with their associated index. Possible applications will exceed the frontiers of the current work as this problem is central in many algorithms associated to high throughput sequencing problems.

7 Acknowledgements

Authors warmly thank Vincent Lacroix and François Coste for their participation to discussions. This work was born from and supported by the Inria "action de recherche collaborative" ARC Alcovna http://alcovna.genouest.org/.

References

1. S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman: *Basic local alignment search tool.* Journal of Molecular Biology, 215(3) 1990, pp. 403–410.
2. M. J. Chaisson, D. Brinza, and P. A. Pevzner: *De novo fragment assembly with short mate-paired reads: Does the read length matter?* Genome Research, 19(2) 2009, pp. 336–346.
3. M. J. Chaisson and P. A. Pevzner: *Short read fragment assembly of bacterial genomes.* Genome Research, 18(2) 2008, pp. 324–330.
4. S. Feature: *Next-generation sequencing transforms todays biology.* Most, 5(1) 2008, pp. 16–18.
5. D. E. Knuth, J. James H. Morris, and V. R. Pratt: *Fast pattern matching in strings.* SIAM Journal on Computing, 6(2) 1977, pp. 323–350.
6. R. Li, H. Zhu, J. Ruan, W. Qian, X. Fang, Z. Shi, Y. Li, S. Li, G. Shan, K. Kristiansen, S. Li, H. Yang, J. Wang, and J. Wang: *De novo assembly of human genomes with massively parallel short read sequencing.* Genome Research, 20(2) 2010, pp. 265–272.

7. P. Medvedev, K. Georgiou, G. Myers, and M. Brudno: *Computability of models for sequence assembly*, in WABI, 2007, pp. 289–301.
8. P. A. Pevzner, H. Tang, and M. S. Waterman: *An Eulerian path approach to DNA fragment assembly.* Proceedings of the National Academy of Sciences of the United States of America, 98(17) 2001, pp. 9748–53.
9. M. E. Smoot, K. Ono, J. Ruscheinski, P.-L. Wang, and T. Ideker: *Cytoscape 2.8: new features for data integration and network visualization.* Bioinformatics, 27(3) 2011, pp. 431–432.
10. A. Viterbi: *Error bounds for convolutional codes and an asymptotically optimum decoding algorithm.* Information Theory, IEEE Transactions on, 13(2) april 1967, pp. 260 –269.
11. D. R. Zerbino and E. Birney: *Velvet: Algorithms for de novo short read assembly using de bruijn graphs.* Genome Research, 18(5) 2008, pp. 821–829.