Università di Pisa
Dipartimento di Informatica

Technical Report: TR-04-15

Fast calculation of a segment mutation distance between long sequences

C. Felicioli
c.felicioli@sns.it

July 23, 2004

Address: via F. Buonarroti 2, 56127 Pisa, Italy. Tel: +39 050 2212700 Fax: +39 050 2212726

Fast calculation of a segment mutation distance between long sequences

C. Felicioli
c.felicioli@sns.it

July 23, 2004

Abstract

BpMatch is a new algorithm whose function is to efficiently calculate, from sequences S and T, the maximum coverage of T using only subsequences and complemented reversed subsequences of S, with minimum length l, possibly overlapped, and, in such a maximum coverage, to minimize the number of subsequences used.

The problem is solved by executing a preelaboration of S (independently from the sequence on which a maximum coverage will be later looked for, and therefore executed only once for any target sequence T), generating a graph which allows a fast recognizing of S's subsequences.

Graphs G and G' must be generated from S and complemented reversed S, then, using G, G' and T, calculus of the maximum coverage can be computed in linear space and time.

1 Introduction

In the field of computational molecular biology, the *transformation distance* (TD) was first defined by J.S. Varré, J.P. Delahaye and E. Rivals [1], in response to the need of having a measure of the distance between genetic sequences with a good biological base.

The TD takes into account not only single base genetic mutations, but sequence genetic mutations as sequence copies, sequence translated copies, sequence translated complemented reversed copies and sequence deletions.

[1] presents an algorithm to calculate TD, which ignores sequence overlaps, with a time complexity of $O(n^6)$ on the worst case and a high (although not analyzed) spatial complexity which makes it generally inapplicable on sequences longer than 100kb. Another algorithm (*compact script graph*) is mentioned, showing a graph in which TD calculation seems to be quadratic in time with regard to sequences length. A precise analysis of this second version is not indicated, and no spatial complexity has been calculated. TD has then been applied in [2] for studying gene families through paralogy trees construction.

The high computational cost of the algorithms to calculate TD has prevented their widespread use. Here a new algorithm called *BpMatch* will be described and analyzed in details. BpMatch allows to calculate a measure of the distance inspired to TD: given a source character sequence S, a target character sequence T and a minimum length l, it finds a maximum coverage of T using only subsequences and complemented reversed subsequences of S, with minimum length l, possibly overlapped, and, in such a maximum coverage, it manages to minimize the number of subsequences used.

BpMatch requires a preelaboration of the source sequence S in time $O(n \log_d(n) \log(\log_d(n)))$ on the average, with $n = |S|$ and d the alphabet cardinality, in order to generate two data structures of dimension $\Theta(n)$. Then, for every target sequence T of length m, the distance can be computed in $O(m \log(\log(n)))$ time and space on the average, provided that $l \simeq 2 \log_d(n)$ or more (l should not be smaller in order to have a reliable distance measure).

For generality BpMatch is presented as alphabet independent although it was intended to be used primarily with the genetic code.

Some formal definitions and proofs are reported in the appendix for reference.

2 The BpMatch algorithm

The algorithm is divided into two phases.

During the first phase a preelaboration of the source sequence S is executed, and generates two objects which permit a time and space linear recognition of S's subsequences and complemented reversed subsequences.

In the second phase, given a target sequence T and the objects generated during the first phase (G for direct subsequences of S and G' for the complemented reversed subsequences of S), the maximum coverage and the minimized number of subsequences of S with minimum length l used for such a coverage are calculated.

If the problem is divided in this way, the preelaboration of S, computationally heavier than the second phase, is executed only once, so a fast calculus of the maximum coverage can be done for every possible target sequence T.

3 First phase: the BpGraphs generation

3.1 The graphs

A BpGraph is a data structure which represents the whole set of S's subsequences as an acyclic graph paths.

The intuitive idea of this object, for the sake of praticality called BpGraph, is based on setting forced paths of arcs on the graph (one path for every subsequence and, if possible, reusing for various subsequences common stretches of path), whose nodes are labeled with the characters of the alphabet in use.

The matching of one sequence u is established if an object placed on the nodes, called cursor, manages to cover a path through BpGraph's nodes and arcs that passes in the right order on nodes whose labels concatenation forms the string u.

The paths are said to be forced when, given a certain cursor placed on a certain node and a character to match, there is only an arc between the outgoing ones of the node where the cursor is placed that can be covered, if it exists.

In order to obtain this property without having to generate an enormous graph, cursors have an internal state, whose value limits arcs to consider valid to be covered. The state can vary every time that the cursor, covering an arc, moves from a node to another.

Given the sequence u and a cursor outside the BpGraph in the starting state, u is an S's subsequence if and only if the cursor manages to find the forced path of nodes whose labels form the string u. Obviously there is a way to insert a cursor in the BpGraph when the first character of u is processed, and, since every step of the path must be determined only by the internal state of the cursor and the character to match, a single entry point for every character of the alphabet must exist.

Thanks to the presence of the cursor's internal state, for a while the same stretch of path can be covered from different cursors with various states without losing the possibility to distinguish them and possibly to divide them on several different paths later.

The possibility to use the same stretch of path for the recognition of several subsequence is the BpGraph's property that allows it to be linear in space with regard to S's length.

The uniqueness of the path is the property that allows the cursors to cover a path of length m of the BpGraph in $O(m \log(maxlink))$ time, with $maxlink$ as the maximum amount of outgoing arcs of a single node, and, generating the BpGraph by using the following algorithm, the maximum arcs number turns out to be linear in relation of S length (n), therefore the time complexity of the path finding on the worst case is $O(m \log(n/m))$. On average the time complexity is $O(m \log(\log(n)))$ on random sequences and on genetic sequences too.

The arcs that can be covered from every cursor, independently from its internal state, are called *direct*, arcs that can be covered only from some cursors are called *lateral*.

In the following algorithm all direct arcs form a single path, acyclic monodirectional, starting from a node labeled with the first character of S, that connect in the right order the nodes that represent all and only the characters of S, and these nodes are all the nodes of which the BpGraph is formed.

3.2 Outlined BpGraph generation

Cursors are used also in the creation of the graph, but in this case they have a different behavior from those used on the completed graph, during the search of a match.

d is the cardinality of the alphabet in use

$S = c_0\ c_1\ c_2\ \dots\ c_{n-1}\ EOF$

The *cursor* object has the following fields :

- *nodeId* : the node where it is placed
- *from* : its internal state

The *node* object has the following fields :

- *id* : its identifier
- *label* : the representing character
- *outgoingArcs*[ ] : the list of its outgoing arcs

The *arc* object, which is monodirectional, has the following fields :

- *id* : its identifier
- *label* : the character of the node it points to (it is not a necessary data but makes the graph use faster)
- *from* : 0 if the arc is *directed*, otherwise, if it is *lateral*, it specifies the needed value of the field *from* that a cursor must have in order to be able to cover it
- *toNode* : the identifier of the node it points to

Array *starts*[ ], of dimension d, has as elements the identifiers of the nodes that constitute the graph entry points for the cursors, one for each different starting character

```
makeBpGraph(S) {
[INIT]  The starts[] array is initialized of dimension d and its elements set to -1;
        i = 0;
[READ]  c = the i-esim input character;
        If (c != EOF) {
[ADD]       newNodeId = nextNodeId();
            A new node with id=newNodeId and label=c is added;
            If (i != 0) {
                A new direct arc with id=nextLinkId() and label=c is added to the
                 outgoingArcs[] array of the previous iteration generated node,
                 linking to the node generated in this iteration (the one with
                 id==newNodeId);
                }
[WALK]      For each cursor {
                If the outgoingArcs[] array of the node where the current cursor is
                 placed contains an arc with label==c which is direct, or, otherwise,
                 with label==c, lateral, and with the from field equal to the from
                 field of the cursor {
                    Cursor nodeId field is updated to the found arc toNode (cursor is
                     placed the node with id==toNode);
                    If the found arc is lateral (from!=0) {
                        cursor from field is updated to the arc id field;
                        }
                    }
                Else {
                    A new lateral arc with id=nextLinkId(), label=c and from equals
                     to the cursor from field is added to the outgoingArcs[] array of
                     the node where the current cursor is placed, pointing to the
                     last created node (the one with id==newNodeId);
                    The cursor is removed;
                    }
                }
[CURSOR]    If (starts[c] == -1) then STARTS[c]=newNodeId;
            Else {
                A new cursor with nodeId=start[c] and from=c is created;
                }
            i++;
            goto [READ];
            }
        END;
        }
```

3.3 BpGraph properties

The BpGraph G (constructed from S) recognizes u if and only if u is a subsequence of S. The proofs can be found in the eppendix.

4 Second phase: the calculation of T's maximum coverage

The calculation of T's maximum coverage, using only the minimized number of subsequences and complemented reversed subsequences of S, with minimum length of l, possibly overlapped, requires the use of the BpGraphs G and G', calculated from S during the first phase, which recognize respectively S's direct subsequences and its complemented reversed subsequences.

The algorithm processes the sequence T, it starts by considering the first character as a possible beginning for a subsequence and proceeds for iterations, processing ahead by one or more character step for each iteration, until it catches up the end of T.

During each iteration two cases can take place: if the hypothesis that the character immediately previous to the one in analysis can not be covered by any subsequence is verified, then this is case 1, otherwise, if a subsequence ending exactly before the character in analysis was previously found, this is case 2.

I define c_0 as the character in actual examination and c_i as the character distant ahead i positions from c_0, in the T sequence. The character $\overline{c}$ is c complemented.

4.1 Case 1

The hypothesis that the character c_{-1}, immediately before the one in analysis, can not be covered by any sequence is verified, therefore c_0 can be covered only if it is the starting character of a covering sequence.

Let us start by searching a direct sequence of length at least l, starting from c_0, covering T as far as at least c_{l-1}. Such a sequence exists only if a cursor of the G' BpGraph, in the starting state, while processing in order the complemented characters $\overline{c_{l-1}}\ \overline{c_{l-2}}$ ... $\overline{c_0}$, manages to find a path, and $c_{l-1}\ c_{l-2}$ ... c_0 is one of S's reversed subsequences.

If the operation of recognition through G' is successful, then c_0 is the first character of a direct sequence of length at least l, so let us start by searching such a direct sequence, processing in order $c_0\ c_1$ ... to a G's cursor C in the starting state until, processing c_i ($i \geq l$), it fails to find a path, therefore the direct S's subsequence $c_0\ c_1$ ... c_{i-1} is found.

Otherwise, if the research of the reversed subsequence starting from c_{l-1} fails during the process of a certain c_i, this means that c_0 can not be covered by any direct sequence of the minimum length l, because such a sequence must contain the subsequence $c_i\ c_{i+1}$ ... c_{l-1} which, since it is not an S's direct subsequence (since it is not recognized by G' its complemented reversion), can not be contained in any S's direct subsequence and so, from the case 1 hypothesis, c_0 can not be covered by any direct subsequence, but this means that c_1 falls back in case 1 hypothesis (at least for the direct subsequences) too, and, for the same reasons, it can not be covered also by direct sequences. Therefore all the characters up to c_i included can not be covered by direct subsequences.

In order to find a complemented reversed subsequence of S, starting from c_0, the same procedure has to be used, but using G in place of G' and vice versa.

After that, three distinct situations can occur :

1 - If only a subsequence is found, direct or complemented reverse, starting from c_0, then it will be added to the coverage and the character immediately after such a subsequence has to be processed by following the procedure described on case 2.

2 - If both the direct and the complemented reverse subsequences are found, then the longer one of them will be added to the coverage and the character immediately after such a subsequence has to be processed by following the procedure described on case 2.

3 - If neither the direct nor the complemented reverse subsequences are found, then I exclude all the characters that can not be covered by neither direct nor complemented reverse subsequences and the character immediately after these has to be processed by following the procedure described on case 1.

4.2 Case 2

The hypothesis that the character c_{-1}, immediately before the one in analysis, is the last character of a covering sequence is verified, therefore all the previous characters up to c_{-l} included are covered and c_0 can be not only the first character of a new sequence, but it can occur in a central position of a sequence, of length at least l, partially overlapped to the sequence ending at character c_{-1}.

In order to find the best direct sequence covering c_0, if it exists, c_0 c_1 ... c_i are processed by a G's cursor C in the starting state, until, during the c_i processing, it fails to find a path.

If $i \geq l$ then it is found. Such a sequence can also be just the final part of an overlapping sequence, but it is obvious that does not exists any overlapping sequence that includes the characters from c_0 to c_i because c_0 c_1 ... c_i is not recognized as a direct S's subsequence, so every sequence having c_0 c_1 ... c_i as one of its substrings can not be recognized too.

If instead $i < l$, then we have to make a search for a partial overlapping direct sequence, covering the maximum possible amount of not yet covered characters and known to be unable to cover c_i (not being c_0 c_1 ... c_i a direct subsequence).

Characters $\overline{c_0}$ $\overline{c_{-1}}$ ... $\overline{c_{-k}}$ have to be processed by a G''s cursor while $k < l$ or until a fail occurs , so c_{-1} c_{-2} ... c_{-k} is a reverse S's subsequence and it is the longest totally overlapped prefix (of length k) of the direct partially overlapping sequence I am searching for.

The longest suffix (c_0 c_1 ... c_{i-1}) is of length i, then if $k + i < l$ I have failed and the direct sequence does not exists, else c_{i-l} c_{i-l+1} ... c_{i-1} has to be processed by a G's cursor.

If the cursor successfully finds a path to c_{i-1}, then a direct subsequence covering all the maximum suffix is found, else, if it fail at c_j, it means that c_{i-l} c_{i-l+1} ... c_j is not a direct S's subsequence and therefore, since it is not able to be included in any direct subsequence of minimum length l, as c_{i-l+1} can not be the starting character of a direct sequence arriving to c_{i-1} (it would be at most of length $l-1$), the length of the longest suffix need to be reduced from i to j.

If $k+j<l$ then I have failed and the direct sequence does not exists, else the same procedure has to be repeated replacing i with j.

On the worst and very improbable case, it is necessary to repeat this procedure for a maximum $l-1$ times in order to find, if it exists, the searched direct and partially overlapping sequence.

In order to find a complemented reversed subsequence of S, starting from c_0, the same procedure has to be used, but using G in place of G' and vice versa.

After that, three distinct situations can occur :

1 - If only a subsequence is found, direct or complemented reverse, starting from c_0, then it will be added to the coverage and the character immediately after such a subsequence has to be processed by following the procedure described on case 2.

2 - If both the direct and the complemented reverse subsequences are found, then the longer one of them will be added to the coverage and the character immediately after such a subsequence has to be processed by following the procedure described on case 2.

3 - If neither the direct nor the complemented reverse subsequences are found, then c_0 can not be covered by neither direct nor complemented reverse subsequences and c_1 has to be processed by following the procedure described on case 1.

5 Minimum length l : meaning and quantification

As said before, in order to get a statistically significative S-T comparison, i.e. in order to find covering sequences that are not randomly taken but statistically significative, an $l > \log_d(n)$ $(n = |S|)$ needs to be used, with d the alphabet cardinality.

The greater is l, the more reliable is the coverage.

On the other hand, the more is the coverage reliability, the less is the sensibility, this because we do not consider all the sequences of length less than l and, among these, also those who could have the importance that we are seeking for.

The l value needs to be chosen while keeping in mind the reliability/sensibility ratio we want to obtain.

It is observed that, even if the case 2 subcycle can cause a great increase of the computational complexity of the algorithm, such an increase is present only when the value of l is near to $\log_d(n)$.

Some long test conducted in parallel, using either pseudo-random generated genetic sequences or real genetic sequences as S and T, varying l over many different values, experimentally confirm that, to get a good reliability of the coverage, the value of l must been bigger than the computationally critic zone $\log_d(n)$.

Images 1 and 2 (right graph are rescaled) show two examples of these tests: the green area indicates the *case1* iterations, the blue area the *case2* and the red area indicates the *case2* subcycle iterations, as previously described in the BpMatch algorithm in section 4.

Using $l = 20 \simeq 2\log_4(1000000) = 2\log_d(n)$ the coverage is not empty only comparing pieces of real genetic sequences.

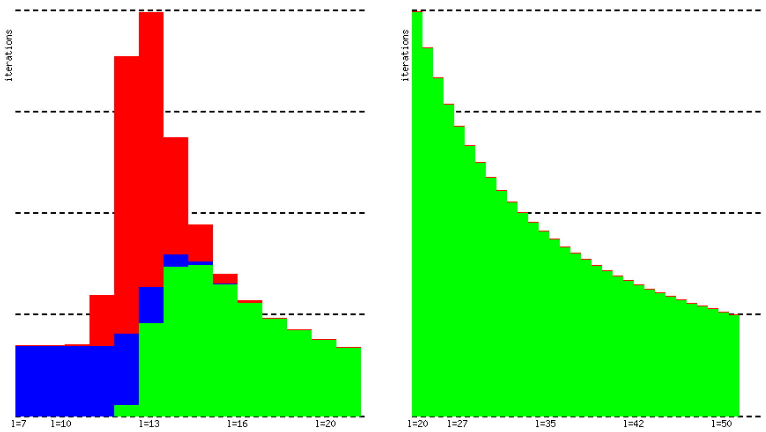

Figure 1: l variation, S is a pseudo randomized genetic sequence and T is a piece of the first human chromosome.

6 Space and time complexity

6.1 BpGraphs

Generating a BpGraph from a sequence S is the heaviest step of the algorithm but, on the other hand, there is the advantage to make it only once and then use it to cover each possible sequence T.

During the execution of the algorithm of BpGraph construction, described on page 4, the main loop [READ][ADD][WALK][CURSOR] is repeated exactly n times ($n = |S|$) and the [WALK] subcycle is iterated once for each cursor on the graph.

6.1.1 Graph dimension and space complexity of graph generation

A single node is added to the graph only in [ADD] so that BpGraph has exactly n node.

A single direct arc can be added to the graph only in [ADD] so that BpGraph has exactly $n - 1$ direct arcs.

Lateral arcs can be added to the graph only in [WALK] subcycle but, given that after a lateral arc creation a cursor is removed, and that a single cursor is

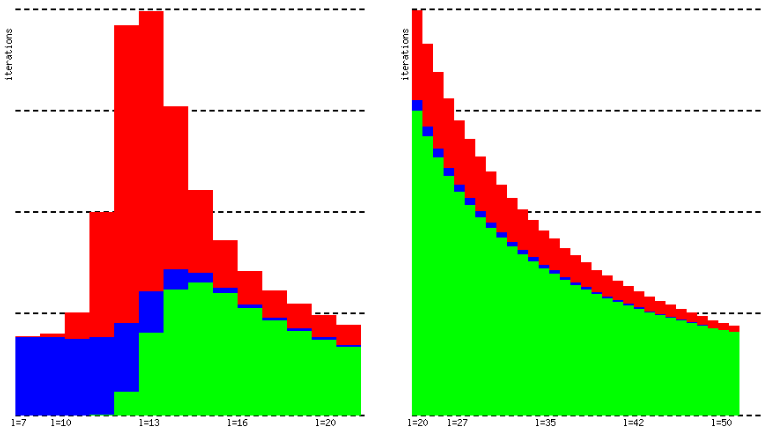

Figure 2: l variation, both S and T are pieces of the first human chromosome.

added to the graph only in [CURSOR], BpGraph has at most $n - d$ direct arcs.

In addition to nodes, direct and lateral arcs, a BpGraph has an array $starts[\,]$ of dimension d (the entry points).

The dimension of a BpGraph and the spatial complexity of its generation is $\mathrm{O}(n \log(n) + d \log(n))$ and, with the assumption that a graph's arc can be represented in a fixed costant space and that the alphabet cardinality d is smaller than S's length, it can be considered $\mathrm{O}(n)$.

6.1.2 Time complexity of graph generation

On the worst case, on the i-esim step, there are $i - 1$ cursors (one cursor can be added on each [CURSOR]), but, on the average of a random character sequence S, there are only $\log_d(i)$ because it is the estimate value of the maximum length of an S's substring ending on S's i-esim character that can be already found in S before its i-esim character.

The average number of cursors on the i-esim step result to be $\log_4(i)$ for real genetic sequence too, but there are short (though strong) fluctuations when the algorithm is analyzing a repetitive genetic sequence (as mini and microsatellites, SINES, LINES, tandem gene family arrays, ecc.).

On each of the n main loop iterations, the [WALK] subcycle has to find an arc for each corsor. The time required to find such an arc is logaritmic to the number of outgoing arc of the node on wich the cursor is placed (as the lateral arcs can be ordered by their field $from$).

There is only one direct arc for each node. The number of a node's outgoing lateral arcs on the i-esim step is on the worst case less than i, but, on the average of a random character sequence S, it is observed to be, for the m-esim node, near to $\log_d(i/m)$.

The average number of outgoing lateral arcs on the i-esim step on the m-esim node results to be $\log_4(i/m)$ also for real genetic sequence, but there are fluctuations for the same situation as described previously for the case of average number of cursors.

On the worst case, the graph generation requires n main iterations and, on the i-esim of these, i iterations of the [WALK] subcycle, each requiring $\mathrm{O}(\log(i))$ time. Therefore, the time complexity on the worst case is $\mathrm{O}(n^2 \log(n))$.

On the average case, the graph generation requires n main iterations and, on the i-esim of these, $\log_d(i)$ iterations of the [WALK] subcycle, each requiring $\mathrm{O}(\log(\log_d(i)))$ time. Therefore, the time complexity on the average case is $\mathrm{O}(n \log_d(n) \log(\log_d(n)))$.

6.1.3 Time complexity of the use of BpGraph

The length of a path on a BpGraph a cursor needs to cover, in order to recognize the match or the mismatch of a sequence, is the length u of the sequence.

The first walk step requires to find the entry point in the array $starts[\,]$, of dimension d, and then each other walk step requires to find an arc among the outgoing ones of the node on wich the cursor is placed (the quantity of outgoing arcs has been analysed in 6.1.1).

On the worst case, the time required to recognize a string of length u is $\mathrm{O}(u \log(n) + log(d))$. On the average it is $\mathrm{O}(u \log(\log_d(n)) + log(d))$ and, if the alphabet dimension is a constant, $\mathrm{O}(u \log(\log(n)))$.

6.2 Calculation of T's maximum coverage

The calculation of T's maximum coverage requires to process u ($u = |T|$) character, switching from case 1 to case 2 and vice versa. Both the time required to examine a case and the number of character I can move forward depend on the case number.

On case 1:
- on failure: the time to match a sequence of length m ($m <= l$), moving forward $l - m + 1$ characters.
- on success: the time to match a sequence of length l and then one other of length m ($m > l$), moving forward $m - 1$ characters.

On case 2:
- on failure: moving forward 1 character, on the worst case $l + 1$ sequences of length l need to be matched.
-on success, finding a non overlapped sequence: the time to match a sequence of length m ($m > l$), moving forward $m - 1$ characters.
-on success, finding an overlapped sequence: the situation on the worst case can be as in the failure case.

On the worst case (if the alphabet dimension is a constant) the time required to calculate the coverage is $O(l^2 u \log(n))$. On the average (if the alphabet dimension is a constant), using $l = 2\log_d(n)$, the time required to calculate the coverage is $O(u \log(\log(n)))$.

7 Examples

The two examples following show graphically the computations.

The alphabet used, for the sake of simpleness, is composed of three character only (A, B and C) and a character complement is the same character itself.

The use of some real genetic sequences is finalized only to test if the algorithm varies its average time complexity, estimated on randomized sequences.

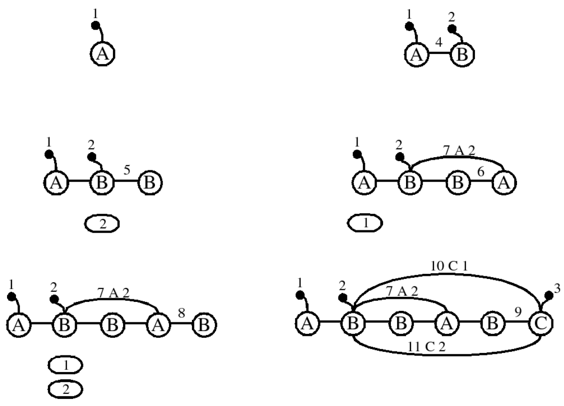

Figure 3: BpGraph generation example.

7.1 BpGraph generation example

Images 1, 2 and 3 graphically show an execution of the algorithm of BpGraph construction described on page 4 with ABBABCACBBAC as the input string.

The circles are BpGraph's nodes, labeled with their represented character, the black dots linked to the nodes, labeled 1, 2 and 3, are the three BpGraph's entry points. Each arc has an identifyer, but for the sake of clearness on the *direct* arcs it has been represented only in its creation step. The labels of the *lateral* arcs are: identifyer, label of the node it is pointing to and arc's $from$ field. The ovals placed under the nodes of the graph are the cursors, they are labeled with their $from$ field value.

Remember that a cursor can cover a lateral arc only if it has a $from$ field equals to the arc's $from$ field.

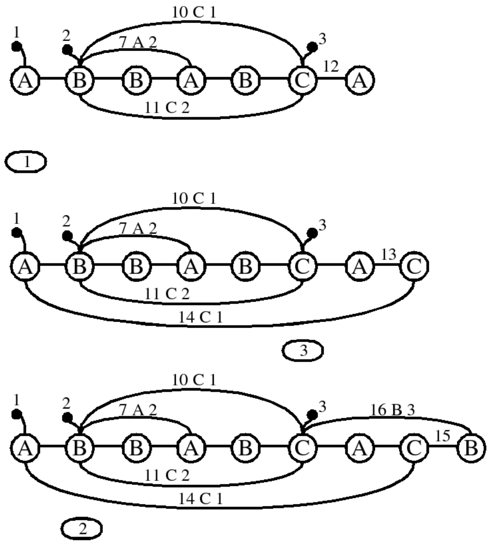

Figure 4: BpGraph generation example.

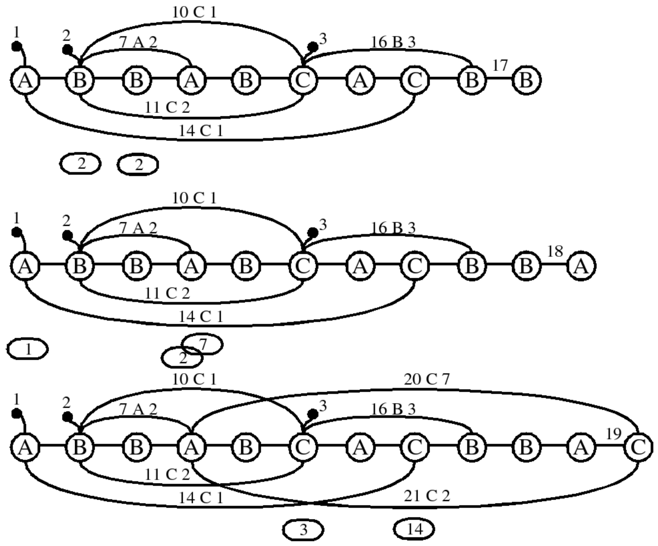

Figure 5: BpGraph generation example.

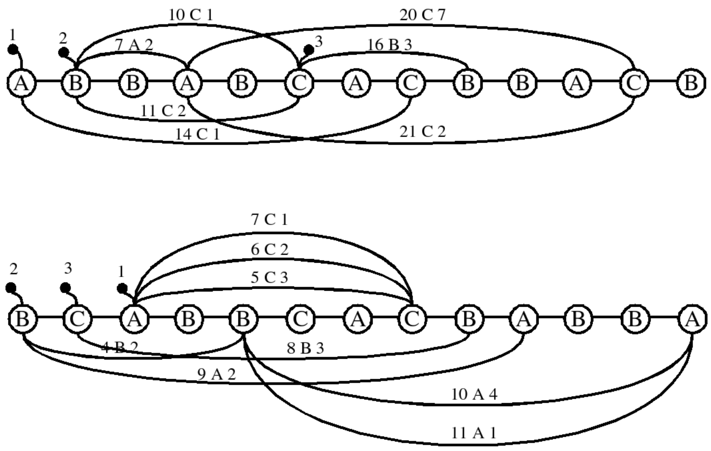

Figure 6: BpGraphs G and G', S=ABBABCACBBACB.

7.2 Maximum coverage calculation example

Image 7.1 graphically shows an execution of the maximum coverage calculation algorithm BpMatch, previously described in section 4.

The source sequence is S=ABBABCACBBACB and the target sequence to cover is T=BAABCAAABBCBACBBACBAABCBCBCACBABCAC.

Image 7.1 shows the two BpGraphs: the first, G, recognizes the direct S's subsequences whereas the second, G', recognizes the complemented reversed ones. In this simplified example the complement operation is the identity, so G' recognizes the reversed S's subsequences too.

The minimum length for a covering subsequence is $l = 5$.

Labels *dir* and *inv* means direct subsequence and complemented reversed subsequence search, the arrows over the sequence represent a successful subsequence recognition thanks to the specified BpGraph.

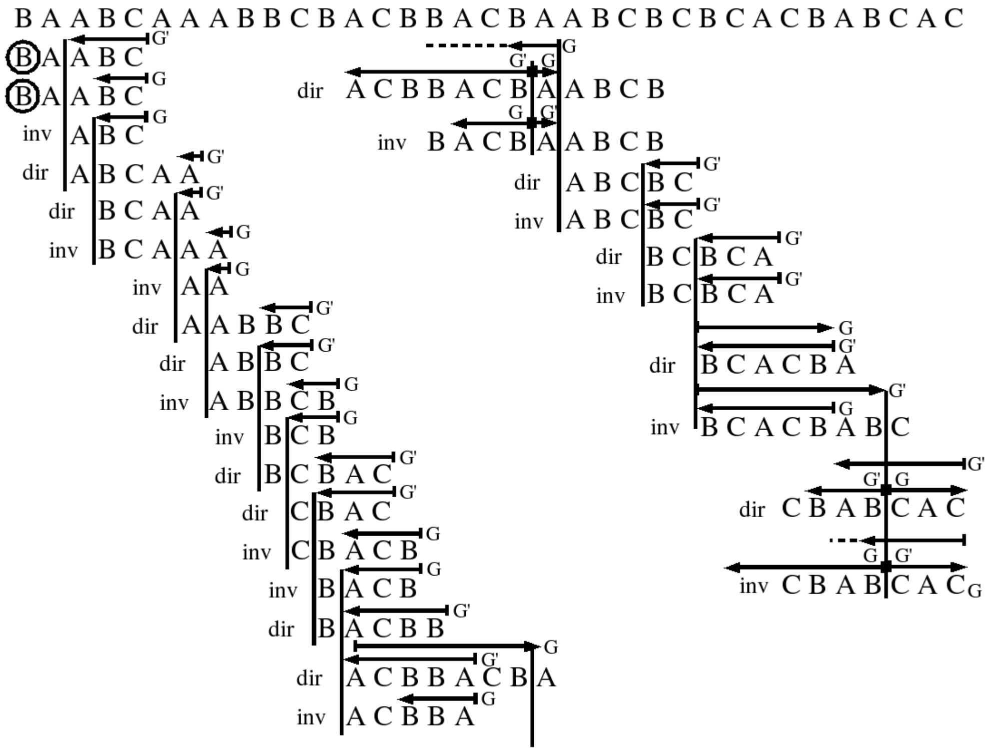

Figure 7: maximum coverage calculation simplified example.

8 Number of cursors during BpGraph construction

The algorithm of BpGraph construction described on page 4 uses cursors for the lateral arcs generation, on each computation step a new cursor can be added and some can be removed.

The number of cursors is the fundamental parameter in the time complexity analysis of the construction algorithm, as described in section 6.

Images from 8 to 17 show the number of cursors during the execution of the algorithm using, as source genetic sequences, pseudo randomized genetic sequence. The red line is the aspected number of cursor, $y = \log_4 x$.

Images from 18 to 26 show the number of cursors during the execution of the algorithm using, as source genetic sequences, pieces of the first human chromosome, of length 2^{20} base pairs. The red line is the expected number of cursor, $y = \log_4 x$.

Observation about these graphs can be found in section 6.

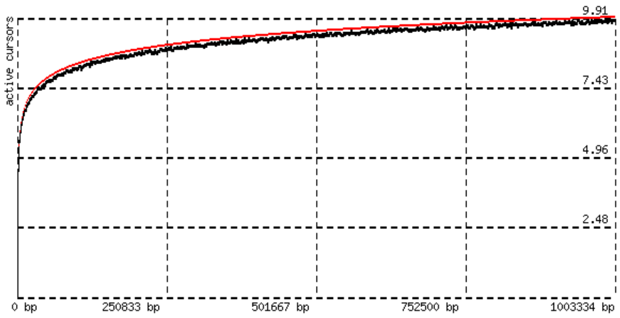

Figure 8: number of cursors during BpGraph construction, using a pseudo-randomized genetic sequence.

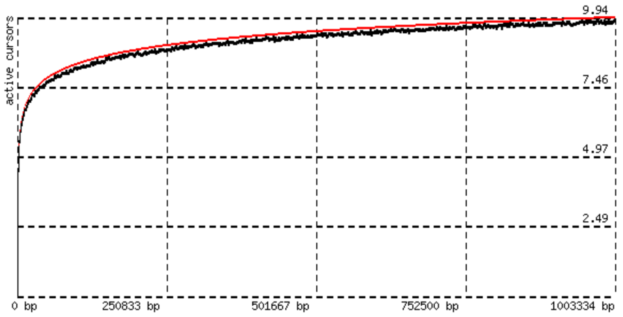

Figure 9: number of cursors during BpGraph construction, using a pseudo-randomized genetic sequence.

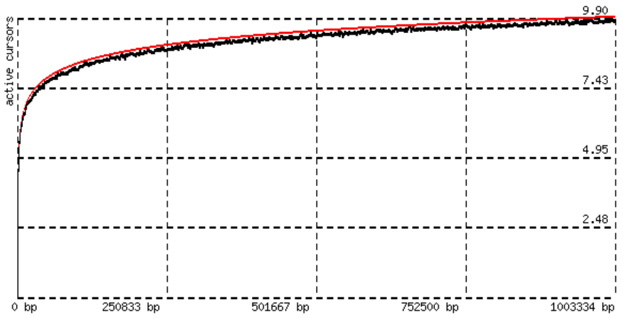

Figure 10: number of cursors during BpGraph construction, using a pseudo-randomized genetic sequence.

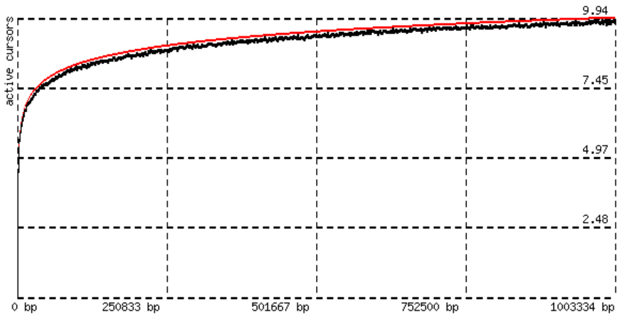

Figure 11: number of cursors during BpGraph construction, using a pseudo-randomized genetic sequence.

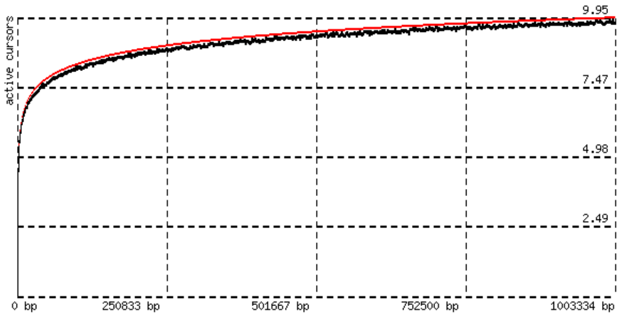

Figure 12: number of cursors during BpGraph construction, using a pseudo-randomized genetic sequence.

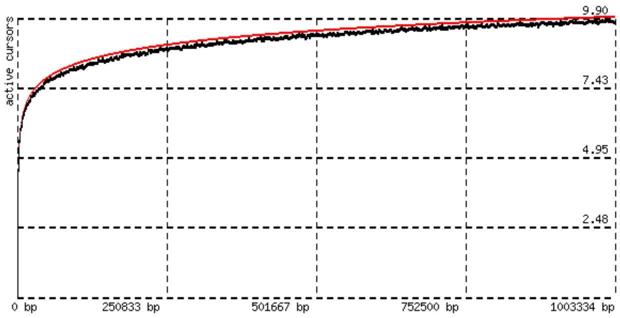

Figure 13: number of cursors during BpGraph construction, using a pseudo-randomized genetic sequence.

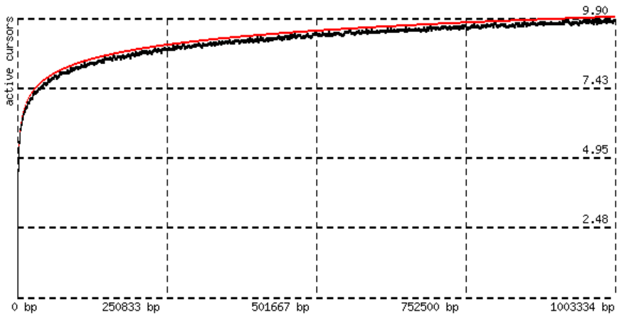

Figure 14: number of cursors during BpGraph construction, using a pseudo-randomized genetic sequence.

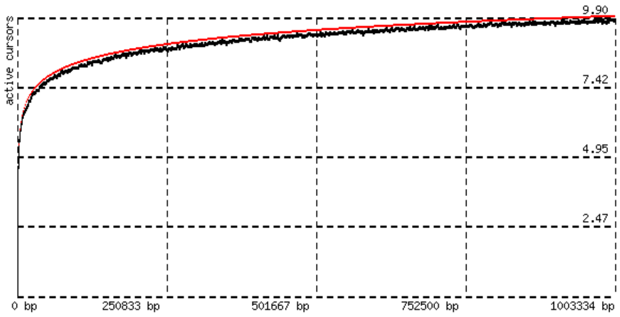

Figure 15: number of cursors during BpGraph construction, using a pseudo-randomized genetic sequence.

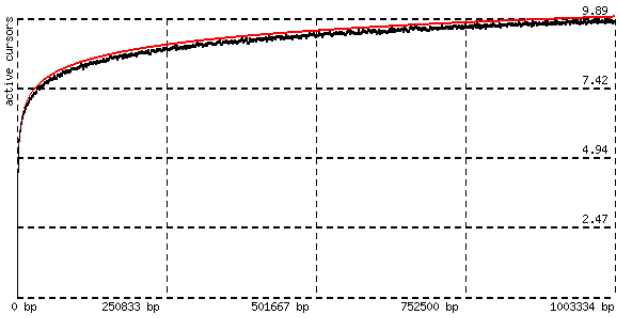

Figure 16: number of cursors during BpGraph construction, using a pseudo-randomized genetic sequence.

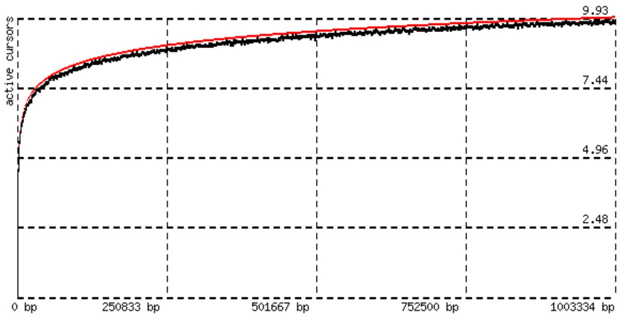

Figure 17: number of cursors during BpGraph construction, using a pseudo-randomized genetic sequence.

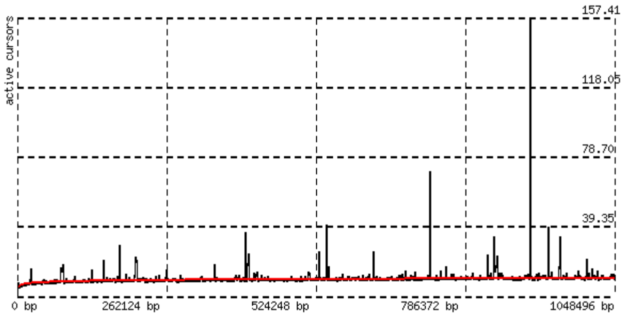

Figure 18: number of cursors during BpGraph construction, using a piece of the first human chromosome.

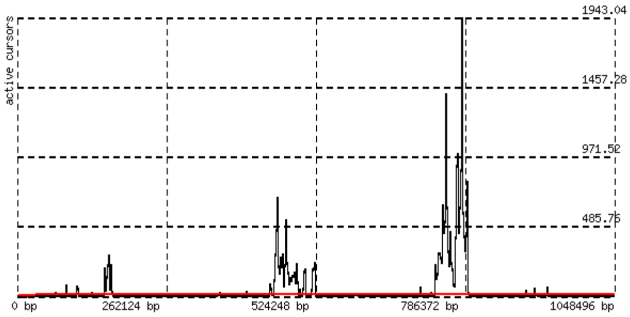

Figure 19: number of cursors during BpGraph construction, using a piece of the first human chromosome.

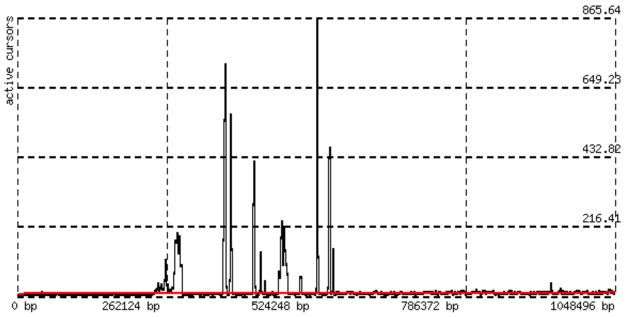

Figure 20: number of cursors during BpGraph construction, using a piece of the first human chromosome.

Figure 21: number of cursors during BpGraph construction, using a piece of the first human chromosome.

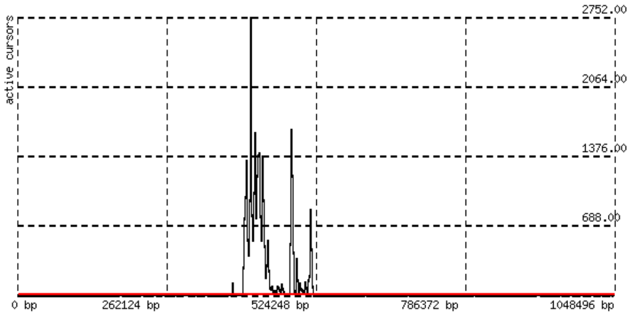

Figure 22: number of cursors during BpGraph construction, using a piece of the first human chromosome.

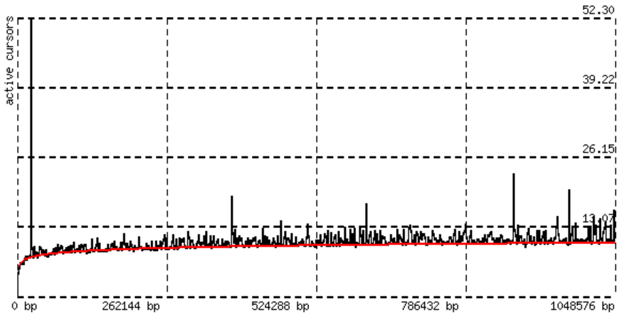

Figure 23: number of cursors during BpGraph construction, using a piece of the first human chromosome.

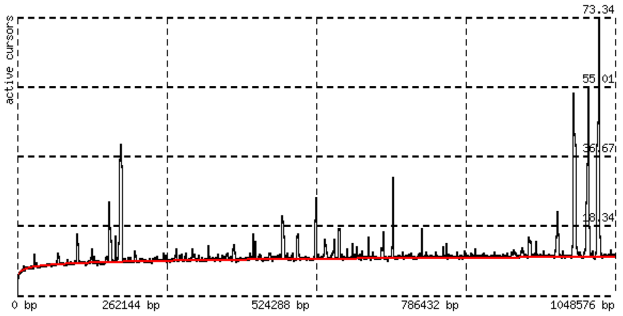

Figure 24: number of cursors during BpGraph construction, using a piece of the first human chromosome.

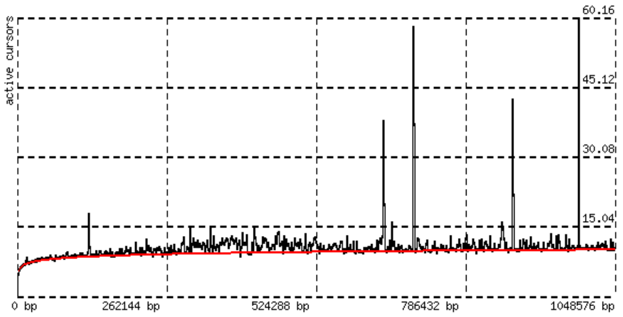

Figure 25: number of cursors during BpGraph construction, using a piece of the first human chromosome.

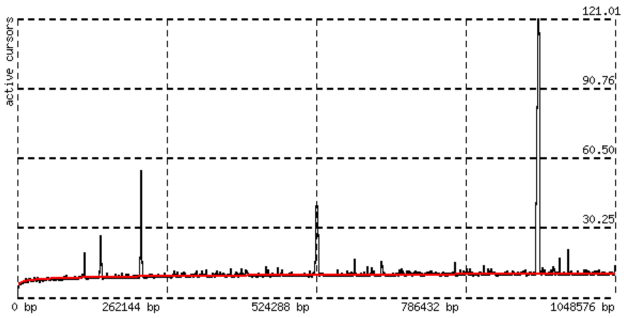

Figure 26: number of cursors during BpGraph construction, using a piece of the first human chromosome.

9 Distribuition of lateral arcs in a BpGraph

From the algorithm of BpGraph construction described on page 4 immediately derives that the number of lateral arcs is upper limited to the source sequence length (n). This happens because a lateral arc can be generated only by a cursor, such a cursor is removed immediately after the arc generation, and during the graph construction there is only a cursor that can be be generated during each of the n iterations.

The exact number of lateral arcs is equals to n minus the number of character in the alphabet (4 in the genetic case), minus the number of cursor of the last iteration of the construction algorithm.

Images from 27 to 36 show the lateral arcs distribution on the nodes of a BpGraph constructed using as source sequence a pseudo randomized genetic sequence. Nodes are ordered from the first created to the last one. The colors visualize *when* the lateral arcs are generated: arcs created in the same time have the same color.

Images from 37 to 46 show the lateral arcs distribution on the nodes of a BpGraph constructed using as source sequence pieces of the first human chromosome, of length 2^{20} base pairs. Nodes are ordered from the first created to the last. The colors visualize *when* the lateral arcs are generated: arcs created in the same time have the same color.

Observation about these graphs can be found in section 6.

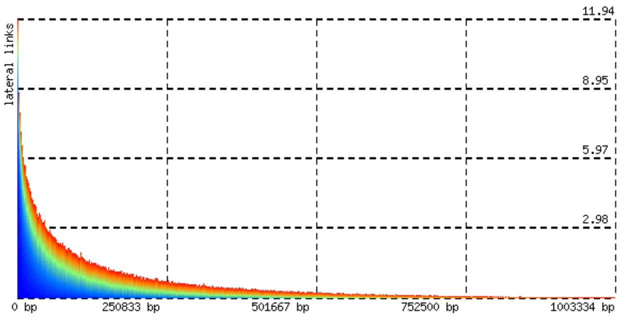

Figure 27: BpGraph's lateral arcs distribution, using a pseudo-randomized genetic sequence.

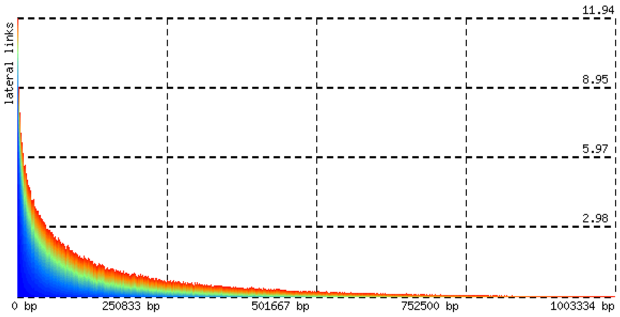

Figure 28: BpGraph's lateral arcs distribution, using a pseudo-randomized genetic sequence.

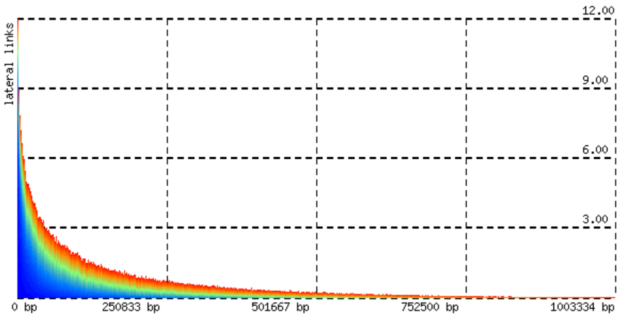

Figure 29: BpGraph's lateral arcs distribution, using a pseudo-randomized genetic sequence.

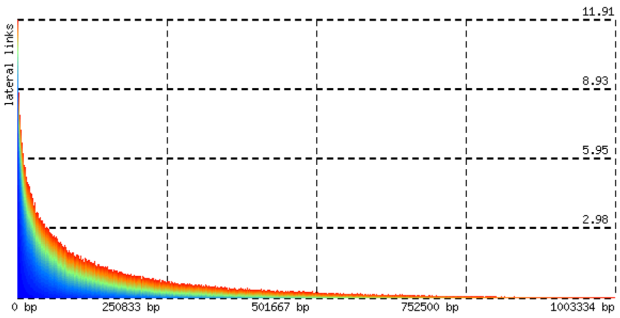

Figure 30: BpGraph's lateral arcs distribution, using a pseudo-randomized genetic sequence.

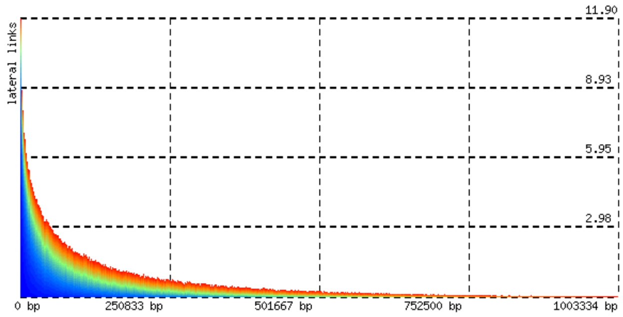

Figure 31: BpGraph's lateral arcs distribution, using a pseudo-randomized genetic sequence.

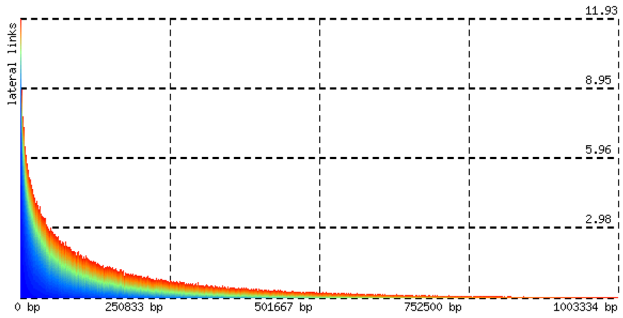

Figure 32: BpGraph's lateral arcs distribution, using a pseudo-randomized genetic sequence.

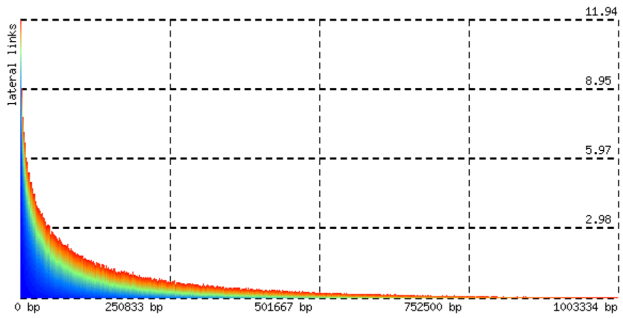

Figure 33: BpGraph's lateral arcs distribution, using a pseudo-randomized genetic sequence.

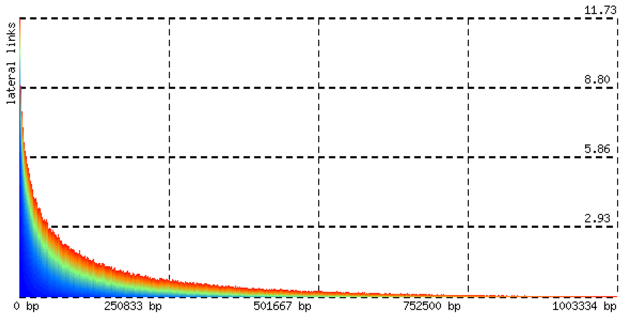

Figure 34: BpGraph's lateral arcs distribution, using a pseudo-randomized genetic sequence.

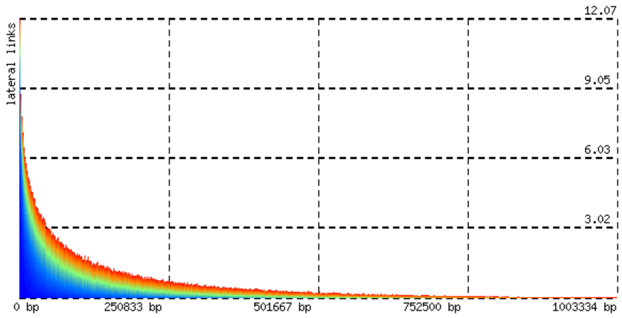

Figure 35: BpGraph's lateral arcs distribution, using a pseudo-randomized genetic sequence.

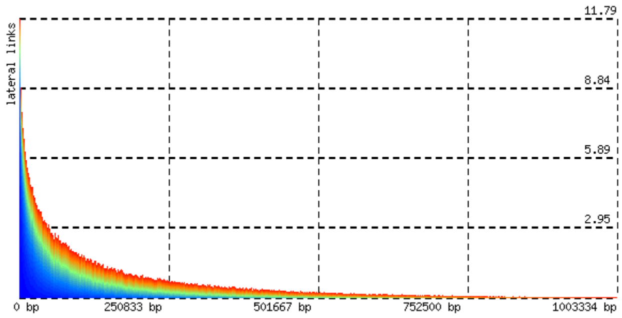

Figure 36: BpGraph's lateral arcs distribution, using a pseudo-randomized genetic sequence.

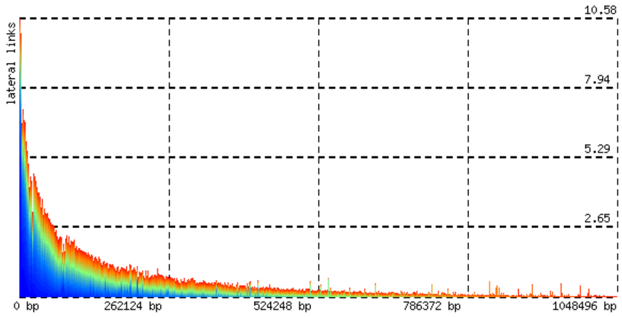

Figure 37: BpGraph's lateral arcs distribution, using a piece of the first human chromosome.

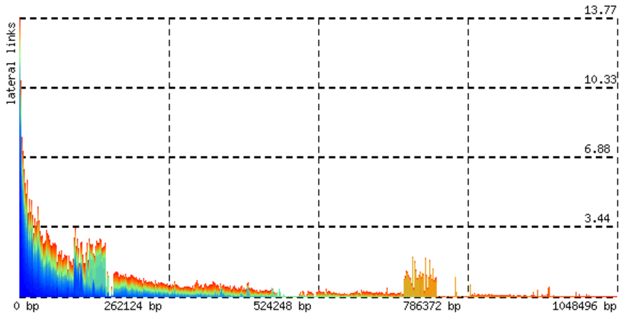

Figure 38: BpGraph's lateral arcs distribution, using a piece of the first human chromosome.

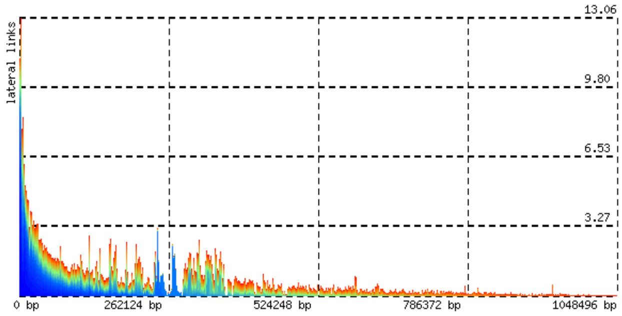

Figure 39: BpGraph's lateral arcs distribution, using a piece of the first human chromosome.

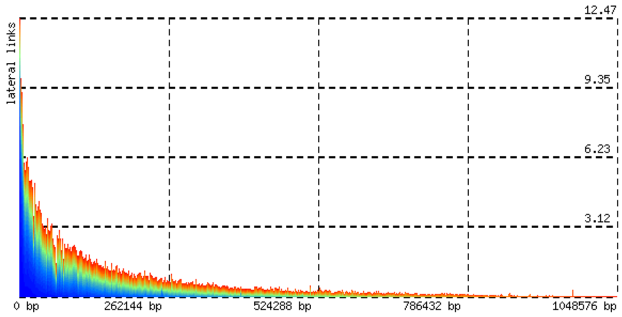

Figure 40: BpGraph's lateral arcs distribution, using a piece of the first human chromosome.

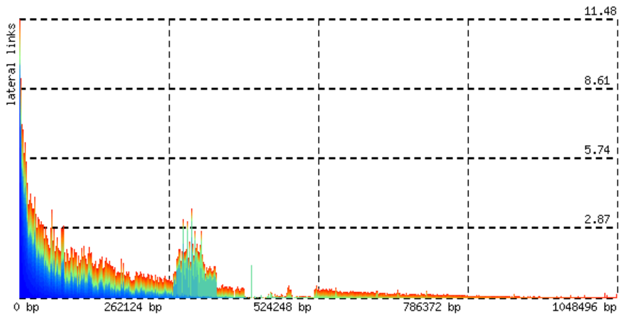

Figure 41: BpGraph's lateral arcs distribution, using a piece of the first human chromosome.

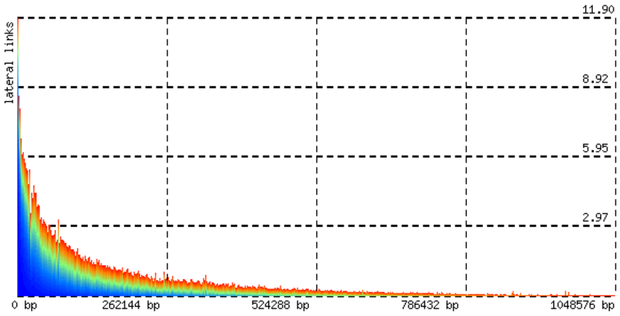

Figure 42: BpGraph's lateral arcs distribution, using a piece of the first human chromosome.

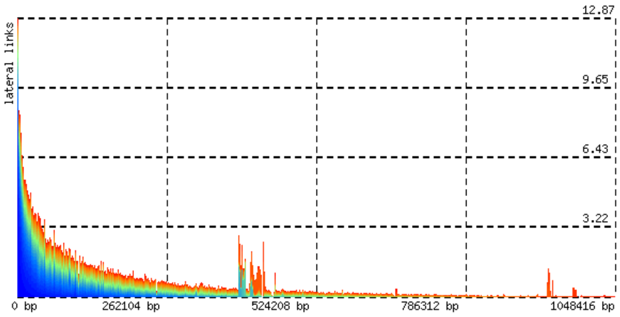

Figure 43: BpGraph's lateral arcs distribution, using a piece of the first human chromosome.

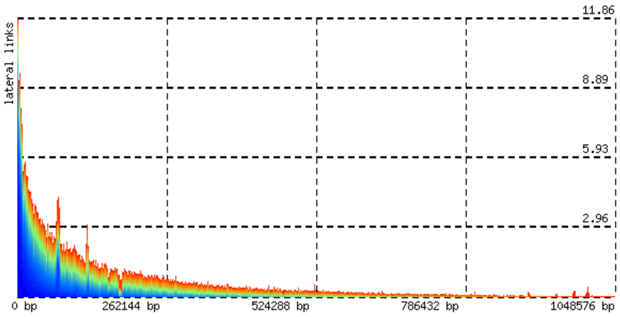

Figure 44: BpGraph's lateral arcs distribution, using a piece of the first human chromosome.

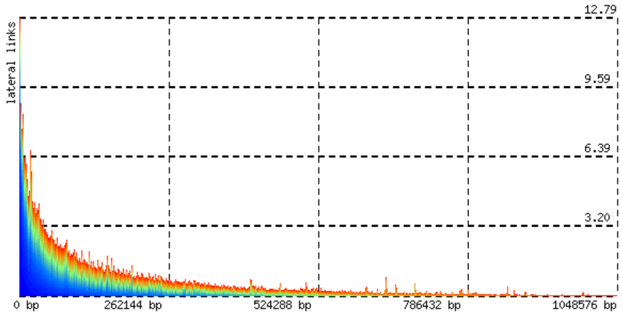

Figure 45: BpGraph's lateral arcs distribution, using a piece of the first human chromosome.

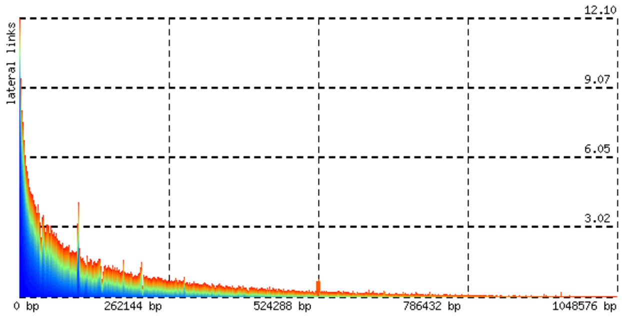

Figure 46: BpGraph's lateral arcs distribution, using a piece of the first human chromosome.

10 Further works

A version of the BpMatch algorithm using modified prefix trees instead of Bp-Graphs is currently in analysis.

References

[1] J.S. Varré, J.P. Delahaye and E. Rivals. *Transformation distances: A family of dissimilarity measures based on movements of segments.* Bioinformatics 15, 1999, Pp. 194-202.

[2] N. Pisanti, R. Marangoni, P. Ferragina, A. Frangioni, A. Savona, C. Pisanelli and F. Luccio. *PaTre: A Method for Parology Trees Construction.* Journal of Computational Biology Volume 10, Number 5, 2003, Pp. 791-802.

Appendix

Proofs of BpGraph properties

Here follows the demonstration that if and only if u is a subsequence of S then the BpGraph G (constructed from S) recognizes it, but first some definitions:

Nodes are entities of the graph with the identifier *id*, a *label* indicating a character that they represent and an array of outgoing arcs.

Arcs are entities of the graph that connect two nodes, they have the identifier *id*, a *label* indicating the character represented by the node to which they point to, they are assigned to the node where they start and have a field $from$ that limits their uses by cursors. Arcs with field $from == 0$ are called *direct* and can always be covered by any cursor, the others are called $lateral$.

Starts[ ] is an array of indices of size equal to the alphabet's cardinality. If c is a character of the alphabet, $starts[c]$ contains the *id* of the entry point node identifier for a cursor that begins its path with the character c.

Cursors are the objects that can move from one graph's node to another, through the arcs. They are necessary for both the construction of the graph and for its use. They have the *nodeId* field (the *id* of the node on which it is placed) and the $from$ field, to keep memory of the last covered *lateral* arc *id* or, if it has not yet covered a *lateral* arc, contains the character of the alphabet through which it has been inserted in the graph.

Definition 1 *Nodes N_a and N_b are connected (by the arc L) in relation to the cursor C placed on N_a, if and only if there is an outgoing arc L connecting N_a to N_b and L is direct or it is lateral but with $L.from = C.from$.*

Definition 2 *Nodes N_0 N_1 ... N_n are a path for the cursor C_0 placed on the node N_0 if and only if every pair of successive nodes N_i N_{i+1} is connected (by an arc L_i) in relation to a cursor C_i placed on N_i defined as (for each $i > 0$) $C_i.nodeId = N_i.id$ and $C_i.from = C_{i-1}.from$ if L_{i-1} is direct, $C_i.from = L_{i-1}.id$ if L_{i-1} is lateral.*

Definition 3 *The G graph recognizes the string u (u=u_0 u_1 ... u_n) if and only if a path N_0 N_1 ... N_n with $N_0.id = G.starts[u_0]$ exists for a cursor placed on N_0 with $from = s_0$ and such that for every index i, $0 \leq i \leq n$, $N_i.label = u_i$.*

Lemma 1 *If nodes N_a and N_b are connected (by the arc L_0) in relation to a cursor C placed on N_a, then a third node N_c such that $N_c.label = N_b.label$ and N_a and N_c are connected (by the arc L_1) in relation to the same cursor C does not exist.*

dim :
From the algorithm of construction of the BpGraph described on page 4 it turns out that an arc L is added to N_a only in two cases :

1 - inserting the *direct* arc between N_a and N_{a+1}, in this case N_a does not have yet any other outgoing arc assigned to it ([DIRECT]).

2 - when there is not yet an arc connecting N_a to a node with a particular label in relation to a cursor with a certain internal state ([LATERAL]).

The thesis derives in a banal way. □

Lemma 2 *If G graph recognizes u (u=u_0 u_1 ... u_n) then a path N_0 N_1 ... N_n as previously defined in $Definition$ 3 exists unique.*

dim :
The existence comes from $Definition$ 3, the unicity derives in a banal way from *Lemma* 1. □

Observation 1 *If a cursor C, entered in the graph from one of the $starts[]$ entry points, is placed on a certain node N_i and $C.from = f$, then the last arc that it has covered (if N_i is not his entry point) is the $lateral$ arc with $id = f$ if that arc points to N_i, otherwise C comes from the outgoing direct arc from the previous node N_{i+1}.*

Observation 2 *A cursor with a determined $from$ field, inserted in the graph from one of the $starts[]$ entry points, placed on a determined node, has been able to cover only one possible path.*

Observation 3 *Given a string u and a graph G, if the G' graph has been constructed from the very same string used for the G construction, but read reversed, then G recognizes u if and only if u reversed is recognized by G'.*

Theorem 1 *If u (u=u_0 u_1 ... u_n) is a substring of S and G is the graph constructed from S by the algorithm of construction of the BpGraph described on page 4, then G recognizes u.*

dim :
During the construction of G, there will be $n+1$ cycles [READ] [ADD] [WALK] [CURSOR] in which [READ] processes, in sequence, u_0, u_1 ... u_n; during the first of these cycles, on the [CURSOR] step, either a new cursor C is added, or the newly created node is marked as the starts[u_0] (in the second case the thesis is banally derived: u will be recognized through the simple path of *direct* arcs starting from the u_0 entry point).

If during the step 0 a new cursor is added (placed on a node called N_0), during every successive step i (if the cursor has not been eliminated) for that cursor the IF condition contained in the [WALK] cycle has to be verified.

If the IF condition is true then N_{i-1} and N_i are connected (by an arc L_i) in relation to the cursor C placed on N_{i-1} and if $L_i.from! = 0$ then $C.from$ has to be updates to $L_i.from$.

Otherwise, if the condition is false, a new *lateral* arc is added, making N_{i-1} and N_i connected (by this new arc) in relation to the cursor C placed on N_{i-1}, then C is removed.

If, iterating in this way, always with the IF condition true, the last character of u is reached, then the path covered by C is the one that makes G recognize the string u, otherwise, if a new *lateral* arc is added (removing C), until that a stretch of path exists and, from that point, having created an arc that connects to the heading node during the construction of the graph, a path of *direct* arcs, coverable by every cursor, leads to the remaining character; such a path is the one that makes G recognize the string u. $\square$

Theorem 2 *If u (u=u_0 u_1 ... u_n) is **not** a substring of S and G is the graph constructed from S by the algorithm of construction of the BpGraph described on page 4, then G does **not** recognizes u.*

dim :
If u is not an S's substring then either u_0 does not appear in S (in this case G banally does not recognizes u, failing to insert a cursor) or there is a j such that :

u_0 u_1 ... u_j is an S's substring

but u_0 u_1 ... u_j u_{j+1} is not an S's substring

From *Theorem* 1 derives that u_0 u_1 ... u_j is recognized by G.

From *Lemma* 2 derives that exists unique the path N_0 N_1 ... N_j that recognizes u_0 u_1 ... u_j.

All cursors, starting from N_0 ($= starts[u_0]$) having $from = u_0$, which cover $j-1$ nodes with labels u_1 u_2 ... u_j cover the same arcs, therefore when they reach N_j they have the same field $from = f$.

N_0 N_1 ... N_j N_{j+1}, with $N_{j+1}.label = u_{j+1}$, are a path (being N_0 N_1 ... N_j a path) only if N_j and N_{j+1} are connected by an arc L.

Such an arc L can be either a *lateral* or a *direct* arc, we begin to consider the hypothesis that it is *lateral*.

In order to laterally connect N_j and N_{j+1} in relation to a cursor having the $from$ field equal to f, the *lateral* arc L must have $L.from = f$, but such an arc exists only if a cursor with $from = f$ has been placed on N_j and the next read character has been u_{j+1}.

But from the *Observation* 2 a cursor C on N_j characterizes univocally a single path, in our case ($C.from = f$) formed of nodes with label u_0 u_1 ... u_j.

Given that $u_0\ u_1\ ...\ u_j\ u_{j+1}$ is not an S's substring, this character sequence never appears during the G construction and therefore a *lateral* arc L that connects N_j to N_{j+1} with $N_{j+1}.label = s_{j+1}$ in relation to a cursor with $from$ field equals to f does not exists.

Therefore L can not be *lateral*, if it exists it must be *direct*.

Naming L_i the arc that connects, for each i, N_i and N_{i+1} in the path $N_0\ N_1\ ...\ N_j$ (this path exists because from $Theorem$ 1 $u_0\ u_1\ ...\ u_j$ is recognized by G), we consider the arc L_k, $0 \le k < j$, such that L_k is *lateral* and does not exists another *lateral* arc $L_{k'}$, $k' > k$, $k' < j$ ($< j$ because L_j is L).

L_k must exists otherwise it means there is a path $N_0\ N_1\ ...\ N_j\ N_{j+1}$ formed only by *direct* arc, but it can not be because, from the algorithm of construction of the BpGraph, a path of only *direct* arcs exists only if the $u_0\ u_1\ ...\ u_j\ u_{j+1}$ string appears in S, and it is false by hypothesis.

Therefore the path $N_{k+1}\ N_{k+2}\ ...\ N_j\ N_{j+1}$ is formed by *direct* arcs only, so it means, as a deduction from the BpGraph construction algorithm, that the string $u_{k+1}\ u_{k+2}\ ...\ u_j\ u_{j+1}$ appears in S.

A cursor that reaches N_{k+1} after covering the path $N_0\ N_1\ ...\ N_k$ starting from N_0 with $from$ field equals to u_0, must have only a possible $from$ field, equals to f' (starting with $from = s_0$ and covering, forced, the arc sequence $L_0\ L_1\ ...\ L_{k-1}$).

But the *lateral* arc L_k, coverable by cursors with $from$ field equals to f', placed on the node N_k, linking to node N_{k+1}, can exist only if during the graph construction a cursor with $from = f'$ was placed on N_k and the u_{k+1} character has been processed.

From $Observation$ 2 a cursor placed on N_k univocally determinate a path, in ours case composed by the nodes labeled $u_0\ u_1\ ...\ u_k$.

Then, to generate the *lateral* arc L_k, character u_{k+1} was processed, therefore the string $u_0\ u_1\ ...\ u_k\ u_{k+1}$ has to be a S's substring.

The existence of a path $N_{k+1}\ N_{k+2}\ ...\ N_j\ N_{j+1}$ formed by *direct* arcs only means, however, that the substring $u_0\ u_1\ ...\ u_k$ in S was followed by not only the character u_{k+1}, but by $u_{k+2}\ ...\ u_j\ u_{j+1}$ and so $u_0\ u_1\ ...\ u_j\ u_{j+1}$ needs to be an S's substring too, but this is false by hypothesis so L can not be either *direct*, so it does not exists.

Therefore a path $N_0\ N_1\ ...\ N_j\ N_{j+1}$, $N_i.label = u_i\ \ \forall i, 0 \le i \le j+1$, $N_0 = starts[u_0]$, does not exists for a cursor C placed on N_0 with $C.from = u_0$.

Therefore, from $Definition$ 3, G does not recognizes the string u. $\square$