Universität Bielefeld

Technische Fakultät
Abteilung Informationstechnik
Forschungsberichte

A Space Efficient Representation for Sparse de Bruijn Subgraphs

José Augusto Amgarten Quitzau Jens Stoye

Report 2008-02

Impressum: Herausgeber:
Ellen Baake, Robert Giegerich, Ralf Hofestädt, Franz Kummert,
Peter Ladkin, Ralf Möller, Tim Nattkemper, Helge Ritter,
Gerhard Sagerer, Jens Stoye, Holger Theisel, Ipke Wachsmuth

Technische Fakultät der Universität Bielefeld,
Abteilung Informationstechnik, Postfach 10 01 31,
33501 Bielefeld, Germany

ISSN 0946-7831

A Space Efficient Representation for Sparse de Bruijn Subgraphs

José Augusto Amgarten Quitzau Jens Stoye

Abstract

De Bruijn graphs are structures that appear naturally in the study of strings. Therefore the rise of de Bruijn graph based sequence analysis approaches is not a surprise. The problem with de Bruijn graphs is that for most of their applications in Bioinformatics they are too large even for small genomes. A way to overcome this problem is the compression of branch-free paths to single nodes. Although this compression is a common first step in many of the de Bruijn graph based approaches, its direct construction from raw data does not seem to be documented before. Our experience shows that, though based on simple operations, implementing the construction of such graphs is a tricky and time consuming task. Therefore we shortly describe in this report our graph construction algorithm and hope that the given details are enough to help the reader skipping some pitfalls we found by doing this task.

1 Introduction

De Bruijn graphs were first defined in the end of the 19th century, though in an implicit form, and were explicitly detailed in the year of 1946, by N. G. de Bruijn [8, Chapter 3]. They are directed graphs with very nice properties. They have a clear and simple definition, a small diameter, are easy to build, regular, and both Hamiltonian and Eulerian. Maybe therefore they are used in many different fields, like network models, pseudo-random number generation, and DNA analysis algorithms [8]. The first use of such graphs in Bioinformatics is probably the Eulerian path approach to sequence assembly proposed by Idury and Waterman [7] and extended by Pevzner, Tang and Waterman [11].

Despite the success achieved by Pevzner and colleagues' Euler assembler in assembling bacterial genomes, de Bruijn graphs do not seem to be further explored in computational biology. De Bruijn graph based approaches found in the recent literature [1, 2, 3, 4, 13] focus either on improvements in error correction methods or in adapting such methods to new sequencing data. Other works present graphs that slightly remember de Bruijn graphs, but miss the main feature of them, namely, the unique representation of tuples of a given size [10, 12].

The main problem with de Bruijn graphs becomes clear as soon as one starts working with them. As Myers points out [9], de Bruijn graphs are simply space inefficient. And we believe Myers is right when he says that in the context of sequence assembly the whole process of cutting reads in small pieces to finally build the de Bruijn graph may not be necessary. But a hybrid between de Bruijn and Myer's string graphs [9] can be constructed without explicitly using sequence comparisons and still having the property of allowing the representation of a whole higher order genome in the memory of a common computer.

In many of the approaches using de Bruijn graphs, the graph construction is followed by the contractions of edges (u, v) for which $\text{indegree}(u) = \text{outdegree}(v) = 1$. The resulting graph is compact in comparison to the original one. In this report we present a way of constructing the compact form without passing through the memory expensive step of constructing the original de Bruijn graph. For doing this we define the *sequence graph*, the compact form of de Bruijn graphs, and show how to include new sequences in it using two operations on its set of nodes: `cut` and `merge`. We also show how exact repeats may be identified and marked during the graph construction, so that this procedure can be easily transformed in a method for marking repetitive regions in incompletely sequenced genomes.

The following sections are organized as follows: In Section 2 we present definitions and the notation used to describe algorithms. The sequence graph is presented in Section 3. Section 4 presents the algorithm for inserting sequences and simultaneously marking exact repeats. Section 5 is dedicated to the comparison between time and space used by de Bruijn subgraphs and sequence graphs. Finally, in Section 6 we present our conclusions and ideas about future work.

2 Notation and Definitions

In the algorithm descriptions, functions and procedures are indicated by small capital names, like MERGE() or CUT(). For references to data attributes we use a function-like notation: For instance, $\text{label}(v)$ is the label associated to node v, and statements like

$$\text{label}(v) \leftarrow \texttt{"ACGT"}$$

are not only reasonable, but often used. As we will see in Section 2.1, the vertices of de Bruijn graphs are strings, and the neighborhood of a vertex is well defined by its content. Nonetheless we separate the type "vertex" from their strings in the algorithm descriptions. Therefore we may sometimes assign a new string to a vertex without changing its neighborhood. Strictly speaking, this is impossible by the definition of de Bruijn graphs, but the algorithms guarantee that every procedure creates a valid set of vertices at the end.

Along this text, the position 0 is the first position of any array. We denote by $a[i \ldots j]$ the elements of a between the positions i and j, inclusive. The size of the array a is denoted by $|a|$. Because we treat strings as arrays of characters, all the notation used to denote arrays and their properties are extended to strings and their properties.

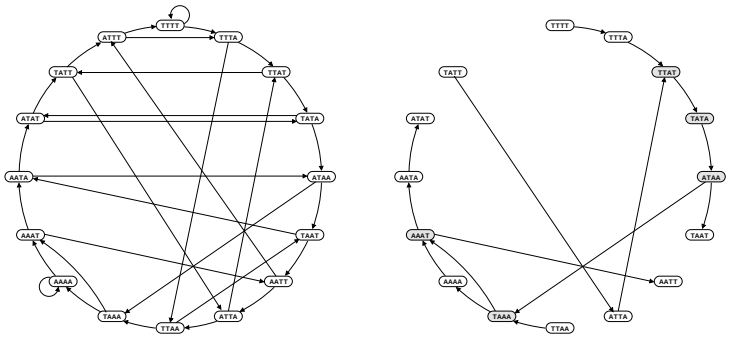

Figure 1: The 4-dimensional de Bruijn graph for the alphabet $\Sigma = \{A, T\}$ (left), and the 4-dimensional de Bruijn subgraph associated to the set $\{\texttt{TTTTATAAT}, \texttt{TATTATAAATT}, \texttt{TTAAAATAT}\}$ (right).

The d-dimensional spectrum of a string s, spectrum(s), is the set of all its substrings of size d. As the spectrum dimension d is always well defined in this text, we call the d-dimensional spectrum simply "spectrum". A slightly different definition is denoted with d-tuples(s), the d-tuples of s, which is the array of d-tuples found in s in the order they appear in the string, therefore, d-tuples$(s)[i] = s[i \ldots i + d - 1]$, for $0 \leq i \leq |s| - d$.

Both in de Bruijn and sequence graphs, the size of a node n, size(n), is the size of the node label, shortly describing, size$(n) = |\text{label}(n)|$.

2.1 De Bruijn Graphs

A d-dimensional de Bruijn Graph $G = (V, A)$ on an alphabet Σ is the graph defined as follows:

$$\begin{aligned} V &= \Sigma^d \\ A &= \{(u, v) \mid u, v \in V \text{ and } u[i+1] = v[i], 0 \leq i < d - 1\}. \end{aligned}$$

In words, it is the directed graph that has all the possible strings of length d over the alphabet as vertices and an arc from vertex u to vertex v if, by deleting the first character of u and the last character of v, we get the same string.

Strings of length at least d over the same alphabet describe walks on the d-dimensional de Bruijn graph. Given a set of strings, we define the associated d-dimensional *de Bruijn subgraph* as the subgraph of the d-dimensional de Bruijn graph that contains all the walks described by these strings and no extra vertex or arc. A vertex in an associated de Bruijn subgraph is called a *junction* when it has in-degree greater than 1. A vertex with out-degree greater than 1 is called *bifurcation*. Figure 1 shows examples of a de Bruijn graph and a de Bruijn subgraph associated to a collection of strings. It is also possible to see that different sets may have the same associated de Bruijn subgraph. For instance,

the set {TATTATAAT, TTTTATAAAATAT, TTAAATT} also has the associated graph shown in Figure 1.

Sequence associated de Bruijn subgraphs have a nice asymptotic behavior. Their maximum number of nodes increases linearly with the size of the input, and even decreases with the dimension of the graph. Their main problem is that, although they scale well with the sequence set size, graphs corresponding to genomes as small as bacterial genomes are already huge. This is probably the reason why, despite the good results presented by the Eulerian assembler in bacterial genomes, results involving eukaryotic genomes are rare, if existent at all.

3 Representing Sparse De Bruijn Subgraphs

De Bruijn graphs are by definition sparse, since the number of edges is linearly correlated to the number of nodes [6, Chapter 7]. They have $|\Sigma|^d$ nodes and $|\Sigma|^{d+1}$ edges, which corresponds to $|\Sigma|^{1-d}$ of the maximum number of edges of a directed graph with self-loops. Note that for $|\Sigma| = 2$ and $d = 2$ this ratio is already $\frac{1}{2}$. Therefore storing de Bruijn graphs by adjacency matrices would be a considerable waste of space. For complete de Bruijn graphs, one could think about a simplification of adjacency matrices, since every node is connected to precisely $|\Sigma|$ other nodes. In our case of sparse de Bruijn graphs, we expect only a small fraction of the nodes to be connected to more than one node. In this case, even the simplified adjacency matrix is sparsely filled. We are left with incidence lists, which may also be improved.

To represent sparse de Bruijn graphs, we use an indexed structure that we call a *d-dimensional sequence graph*, or simply *sequence graph*, shown in Figure 2. Like a d-dimensional de Bruijn subgraph, every d-tuple over the given alphabet is represented by at most one vertex. As well as that, a sequence graph may contain an arc (u, v) only if the $d-1$ suffix of u is identical to the $d-1$ prefix of v. The main difference between sequence graphs and de Bruijn graphs is that vertices in a sequence graph are not limited to the size d, but may have any size between d and $|\Sigma|^d + d - 1$. This allows the representation of non-branching paths in a single node. The compression, however, depends on the way the structure is built.

There is an index mapping every d-tuple represented by the sequence graph to the node in which it is found. Remember that nodes may have size greater than d, therefore the representation of a tuple may start anywhere in the middle of a node. In order to precisely identify a tuple, not only the node, but the offset of the d-tuple is given by the index. We also extend the neighborhood concepts from nodes to tuples. Consider two tuples a and b in a sequence graph. We call b the *successor* of a if either a is the suffix of a vertex u, b is the prefix of a vertex v, and the graph contains the arc (u, v); or there is a vertex v such that $a = v[i \dots i + d - 1]$ and $b = v[i + 1 \dots i + d]$, for some non-negative integer i. We call a the *predecessor* of b in this case, and a and b may also be called *neighbors*. Note that there may exist nodes u with suffix a and v with prefix b

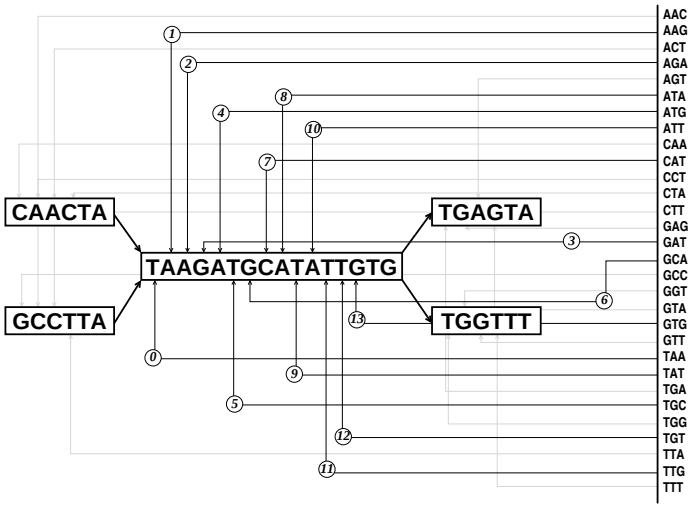

Figure 2: Sequence graph corresponding to a 3-dimensional de Bruijn subgraph on the alphabet $\Sigma = \{A, C, G, T\}$. Connectors to the node `TAAGATGCATATTGTG` are shown as black arrows with offsets, all other connectors are shown in gray.

without a and b being neighbors. See, for example, the vertices `AATT` and `ATTA` in Figure 1.

Apart from the inclusion of nodes, there are two operations that can be applied on the set of nodes: cutting and merging. They are described in the following sections.

3.1 The Cut Operation

The cut operation transforms a single node in two neighbor nodes. It is illustrated in Figure 3, and presented in pseudo-code below. During the cut, a new node is created, and the node prefix is transferred to it. In addition, every incident edge to the cut node is transferred to the new one. A cut does not change the set of sequences represented by the graph, since no new tuple of size d or greater is created, and the new edge binds two tuples that were neighbors before.

As Figure 3 shows, connectors to the cut part are out-of-date after the operation. They should be pointing now to the new node u. Updating these connectors would imply a computational cost of $\mathcal{O}(\log(|\Sigma|^d)) = \mathcal{O}(d \log |\Sigma|)$ for every tuple in the spectrum of the cut part, since all connectors must be first found in the index. To avoid the search in the index, we postpone the connector updates until the next time the connector is used. This is done by a link to one of the incoming nodes, called *followMe*. This link works like an Ariadne's thread, marking the path corresponding to what once was a prefix of the present node.

Algorithm 1 Cut Procedure

```
1:  procedure CUT(v, cutPoint)
2:      create a new node u
3:      label(u) ← label(v)[0 ... cutPoint − 1 + dimension]
4:      delete the first cutPoint characters of label(v)
5:      for each edge (u', v) do
6:          remove the edge (u', v)
7:          create the edge (u', u)
8:      end for
9:      followMe(u) ← followMe(v)
10:     starting-point(u) ← starting-point(v)
11:     starting-point(v) ← starting-point(v) + cutPoint
12:     create the edge (u, v)
13:     followMe(v) ← u
14:     for each s ∈ sequence-set(v) do
15:         put s in sequence-set(u)
16:     end for
17: end procedure
```

A connector knows that an update is needed thanks to a starting point associated to each node. Every time a node is cut, its starting point increases by the size of the prefix the node loses. As a result, the difference between the connector offset and the node starting point is only non-negative if the tuple linked by the connector is still represented by the node after the cut. In the case an update is needed, the correct node is localized by following the trace left by the followMe links. The connector to the tuple "TAA" in Figure 3 shows how it works: after the cut, the offset of the tail is set to 4, and the corresponding difference is -4. This causes the connector to follow the link in the direction to the head of the original node. At the head, the difference becomes 0, which shows that the correct node was reached.

Many of the connectors may never be used. The ones that are used can only be accessed via the index, and any operation that uses them must pay the computational cost of searching for them. When we create the followMe link, we combine two searches in one. The computational cost of finding a connector is payed by the operations that need to access nodes via the index, and must do the search anyway. Therefore avoiding connector updates reduces the cost of each update to $\mathcal{O}(1)$.

In lines 14 to 16 of Algorithm 1, we need to duplicate the sequence set of v. This can be done in $\Theta(|\text{sequence-set}(v)|)$ time. Since both the number of connectors to update and the number of characters to transfer to the new node are bounded by the size of the node, the time needed for a cut is $\mathcal{O}(|v| + |\text{sequence-set}(v)|)$. In a general case, $|\text{sequence-set}(v)|$ may be as large as the number of sequences inserted in the graph. In our application, we expect to have much smaller sequence sets, since the genome coverage is small.

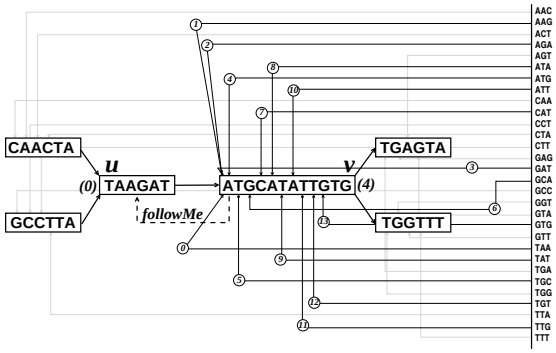

Figure 3: Example of the cut operation applied to position 4 of the central node in the graph of Figure 2. Numbers in parentheses correspond to the node starting points after the cut.

3.2 The Merge Operation

If the arc (u, v) is the only one separating the nodes, and both have identical sets of sequences, they may be merged. The merge operation is the inverse of the cut. It removes (u, v) by merging its nodes into a single node. This is done by transferring the information from v to u, and updating the edges and connectors, so that the node v may be removed from the graph afterwards. The operation is presented in pseudo-code in Algorithm 2.

Because a node is completely removed, and all connectors pointing to it must be updated, the merge operation is asymptotically more time consuming than the cut. Since each connector pointing to the second node must be found in the index, each merge operation takes $\mathcal{O}(|v|\, d \log |\Sigma|)$ time, where d is the graph dimension, Σ is the alphabet, and v is the second node.

Algorithm 2 Merge Procedure

```
1:  procedure MERGE(u, v)
2:      label(u) ← label(u) + label(v)[dimension ... size(v)]
3:      for each edge (v, v') do
4:          remove the edge (v, v')
5:          create the edge (u, v')
6:      end for
7:      for each tuple t ∈ d-tuples(v) do
8:          update connector(t)
9:      end for
10:     discard the node v
11: end procedure
```

3.3 Implementation Details

We assume that the index is implemented by a balanced tree table [5, Chapter 12]. The index maps d-tuples to simple data structures called *connectors*. A *connector* is a pair formed by a pointer to a node and an integer number. The integer node in the connector is the key to find the d-tuple representation in the node: summing up this number to the node starting point either gives the d-tuple offset in the node, or indicates that the connector is out-of-date. In the second case, the connector can be easily updated by following the $followMe$ links until the sum becomes non-negative.

In many applications, it is necessary to store in each node the sequences which are represented by it, as well as the number of occurences of the node label in each sequence. We assume that sequences can be uniquely identifyed by an integer number. And each node has a sequence multiset, where pairs <sequence identifyer,multiplicity> are stored in balanced binary search trees [5, Chapter 12]. If s is the number of sequences inserted in the graph, these trees have at most s nodes, and insertions and searches may be done in $\mathcal{O}(\log s)$ time.

Node adjacencies are stored by arrays of size $|\Sigma|$. Both the edges going from and to the vertices are stored, so that any path in the graph looks like a doubly linked list.

4 Finding Repetitive Sequences in DNA Molecules

The main challenge in using de Bruijn subgraphs as a starting point for sequence assembling is that, even for high dimensions, the subgraph associated to a collection of reads is very tangled. Fortunately, finding exact repeats is much simpler than assembling a genome, since we do not need to untangle graphs, but only to identify the tangled parts. Sequence graphs allow a compact representation of sparse de Bruijn subgraphs, but the representation efficiency depends on how cuts and merges are done. In this section, we show how to insert sequences in an initially empty sequence graph in such a way that the number of nodes is minimized, at the same time that nodes corresponding to repeats in the inserted sequences are identified.

Two properties are important when identifying nodes corresponding to repetitive regions:

marked(v)**:** A node is marked when it is part of a repeated region.

sequence-set(v)**:** The set of sequences that has label(v) as substring.

After each sequence insertion we want the following invariants to hold:

- The value of marked(v) is `true` if and only if label(v) is a repetitive sequence.
- If (u, v) is an edge, then either the out-degree of u or the in-degree of v is greater than 1, or sequence-set(u) $\neq$ sequence-set(v).

4.1 Special Operations

Other minor functions and procedures are used to transform sequence graphs or access their nodes. They are:

cutLeft(c) This procedure cuts the node linked to the connector c exactly at the beginning of the tuple the connector points to. As a result, after the cut, the node begins with the tuple pointed by the given connector.

cutRight(c) Similar to cutLeft, this procedure cuts the node to which the connector c points, creating a node where the tuple is in one of its extremities. However, in this case the corresponding tuple is in the last characters of the node pointed by followMe(node(c)) after the cut.

getConnectors($s, path, i, j$) Let s be a string, $path$ be an array of connectors of length |spectrum(s)|, and $0 \leq i \leq j <$ |spectrum(s)|. This procedure acts on the content of $path[i \dots j]$ in the following way:

a. Let $R =$ spectrum(s)$[i' \dots j']$, $i \leq i' \leq j' \leq j$, be a maximal region such that there is a node n in the sequence graph representing all the d-tuples in R, and in which for every pair of tuples a, b such that b is successor of a in R, b is a successor of a in n. After the execution of getConnectors, $path[i'] =$ connector(spectrum(s)$[i']$), $path[j'] =$ connector(spectrum(s)$[j']$), and $path[k] =$ null, for $i' < k < j'$.

b. For every index i such that the d-tuple spectrum(s)$[i]$ is not represented in the sequence graph, create connector(spectrum(s)$[i]$), and let $path[i] =$ connector(spectrum(s)$[i]$).

The changes on $path$ after calling getConnectors is schematically represented by Figure 4.

getNodes($path$) This operation assumes that the operation getConnectors updated the whole array $path$, so that the entire array looks like the scheme in Figure 4. The function returns the list of nodes containing the regions in $path$, one copy by contiguous region.

contiguous($path, i, j$) Let $path$ be an array of connectors. This boolean function returns `true` if the connectors in $path[i \dots j]$ correspond to a substring of a node.

Both cut operations clearly have the same running time as the cut operation described in Algorithm 1. getConnectors only uses the index to localize the desired connectors, therefore it runs in $\mathcal{O}((j-i)d \log |\Sigma|)$ time. The function getNodes simply accesses the nodes of a collection of connectors, updating the connectors. However, the computational cost for updating a connector was already "payed" by the cut operation that caused this cost. Therefore we consider

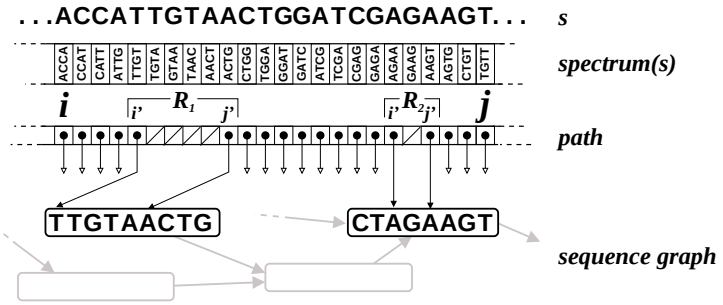

Figure 4: Schematic representation of the array *path* after the procedure call GETCONNECTORS$(s, path, i, j)$. New connectors are represented by white headed arrows. The old connectors are divided in two regions, R_1 and R_2.

that the connector update is made in constant time, and the function GETNODES runs in $\mathcal{O}(|path|)$ time. CONTIGUOUS can clearly be calculated in $\mathcal{O}(j-i)$ time.

4.2 Sequence Insertion

The insertion of a string, shown in pseudo-code in Algorithm 3, is done in three phases, which are described below.

4.2.1 Phase One: Changes in the Set of Nodes

In the first phase, the graph is prepared for the sequence insertion. In this phase, existing nodes are cut in order to represent the common substrings that the new sequence shares with already inserted sequences. At the same time, new nodes are inserted in the graph, so that the unique new parts can be represented as well. This phase is preceded by a short initialization step, where the connectors to the sequence tuples are either found or created and inserted in the index. At the end of the initialization step, we have an array of connectors which looks like the one shown in Figure 4. This structure guides the creation of new nodes, as well as the adaptation of old nodes to the new sequence. Nodes are created or adapted while the sequence spectrum is analyzed from *left* to *right*, that is, from the first to the last d-tuple.

New nodes Lines 7 to 17 of Algorithm 3 detect and fill places where new nodes are needed. The necessity of a new node is identified by the presence of *empty connectors*, which are connectors that are not associated to any node. Empty connectors are represented by white headed arrows in Figure 4. When the leftmost in a series of empty connectors is found, a node is created (line 9) and labeled with the first $d-1$ symbols of the corresponding d-tuple (line 10). Neighboring empty connectors are iteratively found, and the necessary updates

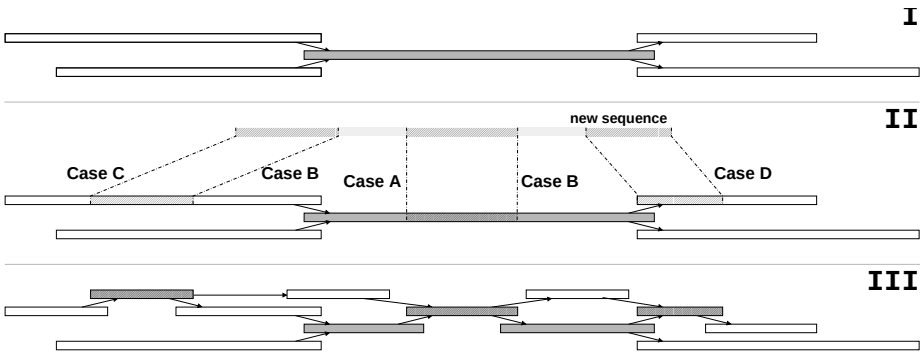

Figure 5: Snapshots of a de Bruijn subgraph in three different moments of a sequence insertion. In **I**, we see the graph before the insertion. In **II**, the contiguous regions already in the graph are identified. Finally, **III** shows the graph after the cuts, with the new repeated regions marked.

both in the new node and in the connectors are done (lines 11 to 16). This step continues until the first non-empty connector or the end of the string is reached.

Old nodes Any node connected to a non-empty connector is an *old node*. Even newly created nodes can be considered old if they contain tuples which are duplicated in the new sequence. The importance of distinguishing between new and old nodes is that only old nodes may be repetitive. This fact is used in Phase Three (Section 4.2.3) to mark repetitive nodes.

If only a part of an old node matches the new sequence, the node must be cut. Any pair of neighboring d-tuples in the new sequence may force a node cut. It is only necessary that at least one of the two tuples is already represented, and that the tuples are not neighbors in the graph. A cut is necessary every time at least one of the cases described below is observed. (Figure 5 illustrates these cases.)

Case A: Let a and b be two tuples in the string to be inserted, such that a is the predecessor of b. Suppose b is already represented in the graph by the node n. If a is not the predecessor of b in n and b is not the leftmost tuple of n, then n must be cut in two nodes, creating the arc (n_1, n_2), in such a way that b is the leftmost tuple of n_2.

Case B: Let a and b be two tuples in the string to be inserted, such that a is the predecessor of b. Suppose a is already represented in the graph by the node n. If b is not the successor of a in n and a is not the rightmost tuple of n, then n must be cut in two nodes, creating the arc (n_1, n_2), in such a way that a is the rightmost tuple of n_1.

Case C: Let a be the leftmost tuple in the string to be inserted. Suppose a is already represented in the graph in the node n. If a is not the leftmost

tuple of n, then n must be cut in two nodes, creating the arc (n_1, n_2), in such a way that a is the leftmost tuple of n_2.

Case D: Let a be the rightmost tuple in the string to be inserted. Suppose a is already represented in the graph in the node n. If a is not the rightmost tuple of n, then n must be cut in two nodes, creating the arc (n_1, n_2), in such a way that a is the rightmost tuple of n_1.

In Algorithm 3, we use the boolean function CONTIGUOUS to identify the longest substring that matches an existing node (line 20). Once this substring is found, we are able to analyze the extremities of the corresponding node region, so that the necessary cuts may be done (lines 21 and 22).

4.2.2 Phase Two: Walk Connection

The set of nodes may have been modified by cuts in phase one. Therefore phase two also starts with two initialization steps: the update of the connectors in $cPath$ (line 26), and the conversion of $cPath$ into the corresponding vector of nodes $nPath$ (line 27). Phase two begins with the connection of the nodes in $nPath$ (line 28) and the insertion of the sequence identifier in each of the nodes' sequence sets (line 29).

If nodes may be merged, this is done in lines 30 to 32. It is only possible to merge two nodes if they have the same set of sequences and are neighbors. Nodes with the same set of sequences are either created in the same insertion or are two parts of a cut node. If they are created during the same sequence insertion, they are not neighbors by construction. If they are parts of a cut node, they can only remain with the same set of sequences if the rightmost portion of the node corresponds to the beginning of the sequence, and the leftmost portion corresponds to the end of the sequence. Any other disposition would create either a difference between the sequence sets or a prohibitive degree greater than one. Therefore the only nodes that may be merged after a sequence insertion are the first node in the sequence walk and its previous node.

4.2.3 Phase Three: Repeat Identification

After phase two, there is a walk representing the new sequence in the graph, and this walk is stored by the array $nPath$. This last phase identifies nodes in this walk which correspond to repetitive sequences, we call these *repetitive nodes*. Each repetitive node was found in the graph during phase one and marked as *old* in line 23. Algorithm 3 differentiates repetitive nodes from common old nodes by comparing each sequence set to the set of sequences that either enter the new sequence's walk through a junction or leaves it through a bifurcation.

In the third phase, every maximal contiguous region formed only by old nodes is separately analyzed (lines 33 to 45). For each node v in such a region, we find the set of sequences which access v through an alternative walk (line 37). These are all the sequences belonging at the same time to the sequence set of v and the sequence set of the neighbors of v which are not in the newly created

Algorithm 3 Sequence Insertion

```
1: procedure InsertSequence(s)
2:     index ← 0
3:     last ← |s| − d + 1
4:     cPath is an empty array of connectors of size last
5:     getConnectors(s, cPath, 0, last)                                  ▷ Phase One
6:     while index < last do
7:         if node(cPath[index]) = null then                             ▷ new node
8:             start ← index
9:             create a new node n
10:            label(n) ← d-tuples(s)[start][0 . . . d − 1]
11:            repeat
12:                node(cPath[index]) ← n
13:                offset(cPath[index]) ← index − start
14:                append the last symbol of d-tuples(s)[index] to label(n)
15:                index ← index + 1
16:            until index = last or node(cPath[index]) ≠ null
17:        end if
18:        if index < last then                                          ▷ old node
19:            start ← index
20:            index ← min(x : x ≥ start ∧ ¬contiguous(cPath, start, x))
21:            if Case A or Case C then cutLeft(cPath[start])
22:            if Case B or Case D then cutRight(cPath[index − 1])
23:            mark node(cPath[start]) as old
24:        end if
25:    end while
26:    getConnectors(s, cPath, 0, last)                                  ▷ Phase Two
27:    nPath ← getNodes(cPath)
28:    for each pair of neighbors u, v in nPath, create the edge (u, v)
29:    for each node n ∈ nPath, insert s in sequence-set(n)
30:    if sequence-set(nPath[0]) = sequence-set(followMe(nPath[0])) then
31:        merge(followMe(nPath[0]), nPath[0])
32:    end if
33:    for each maximal contiguous region old nodes nPath[i . . . j] do  ▷ Phase Three
34:        R ← ∅
35:        for each v in nPath[i . . . j],
36:                    and its left and right neighbors, u and w, both possibly null do
37:            U ← ⋃_{u′∈in(v)\{u}} sequence-set(u′) ∪ ⋃_{w′∈in(v)\{w}} sequence-set(w′)
38:            R ← R ∪ (sequence-set(v) ∩ U)
39:        end for
40:        for each unmarked node n ∈ nPath[i . . . j] do
41:            if sequence-set(n) ∩ R ≠ ∅ then
42:                marked(n) ←TRUE
43:            end if
44:        end for
45:    end for
46: end procedure
```

walk. In Algorithm 3, these sequences are cumulated in a set R (line 38). Finally, each node containing one of the sequences in R is for sure a repeat, and may be marked (lines 40 to 44).

Not that every time the walk of a new sequence passes through an old node, the node may be a repetitive node. However, two walks sharing nodes are not always evidence of repeat. When the end of a walk corresponds exactly to the beginning of another one, the common part between them corresponds to a portion of the genome which was sequenced twice. The same is true if one walk is completely contained in another one.

4.3 Running Time

In this section, l is the length of the inserted sequence, s denotes the number of sequences inserted so far, and L the maximum length among the s sequences.

Phase one takes $\mathcal{O}(lL + ld)$ time. In the worst case, the $\mathcal{O}(l)$ tuples in the sequence are individually analyzed, and all operations, but the cut, can be executed in constant time. Notice that every l-tuple may cause at most 2 cuts, both bounded by the maximum vertex size, which is bounded by the length of the longest sequence already inserted. At the same time, each tuple may force a search in the index at most two times (in lines 5 and 26). Each search in the index takes $\mathcal{O}(\log |V|) = \mathcal{O}(\log |\Sigma|^d) = \mathcal{O}(d \log |\Sigma|)$ time. Since the alphabet size is constant for us, each search requires $\mathcal{O}(d)$ time.

In phase two, the running time is dominated by the insertions of sequences in the nodes' sequence sets (iterative statement in line 29), and by the merge in line 31. The update is done in $\mathcal{O}(l \log s)$ time, since insertions in balanced search trees with size at most s require $\mathcal{O}(\log s)$ time, and at most l insertions are done. Since the only nodes that may be merged are the first and the last nodes of the new sequence walk, merges are done in $\mathcal{O}(l \log l)$ time. Summing up, phase two is executed in $\mathcal{O}(l \log s + l \log l)$ time. Notice that observing the factor $l \log l$ is unlikely, given that situations where the merge is allowed are rare.

The third phase is executed in $\mathcal{O}(ls \log s)$ time. The union in line 37, together with the intersection in line 38 (in parentheses), requires traversing at most $2(|\Sigma| - 1) + 1$ trees, with at most s nodes each. This cannot result in a set with more than s distinct elements. The union with R is done by inserting the elements of the resulting set in R. The whole computation is done in $\mathcal{O}(s \log s)$ time, and at most once for each tuple, giving the proposed running time bound. The overall running time of a sequence insertion is therefore bounded by

$$\mathcal{O}(lL + ld + ls \log s).$$

Remarks on the Running Time Analysis. The reader might have noticed that the time for duplicating the sequence set in the cut operations was not added to the running time of phase one. Since the cuts are responsible for the lL factor in the overall insertion running time, one could argue that the upper bound should be larger. In fact, the time for duplicating the set is covered by

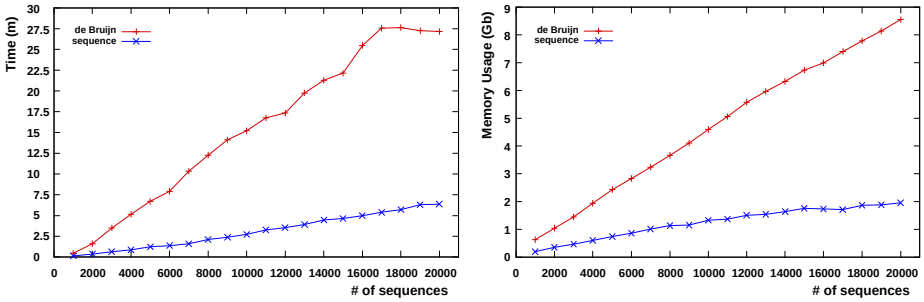

Figure 6: Time and space required by the two representations of de Bruijn subgraphs.

the upper bound for the sequence sets update in line 29. To understand why, consider cutting a node v with σ sequences in its sequence set. Since v can be cut, it must represent $\tau > 1$ tuples. And the σ sequences in its sequence set show that for σ times we computed $\tau - 1$ times an unused cost of $\mathcal{O}(\log s)$. We have therefore a "credit" of $\mathcal{O}(\tau\sigma \log s)$, whereas at the time of the cut, the sequence set duplication can be done in $\mathcal{O}(\sigma) = \mathcal{O}(\tau\sigma \log s)$. Therefore we may ignore the duplication in the overall running time.

5 Empirical Results

We compared the space usage and time required by both the usual and the proposed representation of de Bruijn subgraphs. This was done by constructing subgraphs based on sets of DNA sequences. The data sets were created based on the first chromosome of the plant *Arabidopsis thaliana*. We artificially created 20 sets with sizes varying between 2,000 and 40,000 reads. The reads had length 750bp and were normally distributed along the chromosome. For each set we built both the de Bruijn subgraph and the sequence graph using dimension 19, and measured time and space requirements. This whole procedure is called a *round*. We ran a total of 20 rounds. The average measures are presented in Figure 6.

Although the worst case analysis for the running time shows that the sequence graph construction may take much longer than the corresponding de Bruijn graph, we may expect to construct such a representation faster than the normal de Bruijn graph, as we see in the graphics. This happens because the worst case not only requires a very peculiar data set, but also requires the sequence insertions to be done in a very specific order, which is very seldom the case. Usually, the reduced number of nodes in the sequence graph diminishes the time needed to localize a node or transverse a path in the graph. This explains the reduction of time proportional to the reduction of required memory.

6 Conclusion and Future Work

We presented a representation of de Bruijn subgraphs with long non-branching paths that is able to represent exactly the same content of a de Bruijn subgraph using less than 25% of the space required by the traditional representation. We also presented empirical results showing that the reduction of the required space also implies a proportional reduction in the running time for the sequence graph.

More work can still be done to both reduce and understand the structures of the proposed graph. The discrepancy between the theoretical and the practical running time analysis shows that this is a case where an average case analysis could be very helpful. A more precise analysis may not be done while the length of induced paths, as well as the number of bifurcations and junctions, cannot be estimated.

Concerning the use of sequence graphs for representing DNA sequences, we may also explore the complementarity of these sequences to reduce even more the graph size, since in graphs representing DNA sequences the information is stored twice: as the sequence itself and as its complement.

References

[1] S. H. Bokhari and J. R. Sauer. A parallel graph decomposition algorithm for DNA sequencing with nanopores. *Bioinformatics*, 21(7):889–896, 2005.

[2] J. Butler, I. MacCallum, M. Kleber, I. A. Shlyakhter, M. K. Belmonte, E. S. Lander, C. Nusbaum, and D. B. Jaffe. ALLPATHS: De novo assembly of whole-genome shotgun microreads. *Genome Res.*, 18:810–820, March 2008.

[3] M. Chaisson, P. Pevzner, and H. Haixu Tang. Fragment assembly with short reads. *Bioinformatics*, 20(13):2067–2074, 2004.

[4] M. J. Chaisson and P. A. Pevzner. Short read fragment assembly of bacterial genomes. *Genome Res.*, 18:324–330, March 2008.

[5] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, 2nd edition, 2001.

[6] R. Diestel. *Graph Theory*. Number 173 in Graduate Texts in Mathematics. Springer-Verlag, third edition, 2005.

[7] R. M. Idury and M. S. Waterman. A new algorithm for DNA sequence assembly. *J. Comput. Biol.*, 2(2):291–306, 1995.

[8] E. Moreno. *De Bruijn graphs and sequences in languages with restrictions*. PhD thesis, Université de Marne-La-Vallée, May 2005.

[9] E. W. Myers. The fragment assembly string graphs. *Bioinformatics*, 21:ii79–ii85, 2005.

[10] P. A. Pevzner, H. Tang, and G. Tesler. *De novo* repeat classification and fragment assembly. In *Proceedings of RECOMB'04*, pages 213–222, March 2004.

[11] P. A. Pevzner, H. Tang, and M. S. Waterman. An Eulerian path approach to DNA fragment assembly. *Proc. Natl. Acad. Sci. USA*, 98(17):9748–9753, August 2001.

[12] B. Raphael, D. Zhi, H. Tang, and P. Pevzner. A novel method for multiple alignment of sequences with repeated and shuffled elements. *Genome Res.*, 14:2336–2346, 2004.

[13] D. R. Zerbino and E. Birney. Velvet: Algorithms for de novo short read assembly using de Bruijn graphs. *Genome Res.*, 18:821–829, March 2008.

Bisher erschienene Reports an der Technischen Fakultät
Stand: 2008-06-23

94-01 Modular Properties of Composable Term Rewriting Systems
(Enno Ohlebusch)

94-02 Analysis and Applications of the Direct Cascade Architecture
(Enno Littmann, Helge Ritter)

94-03 From Ukkonen to McCreight and Weiner: A Unifying View of Linear-Time Suffix Tree Construction
(Robert Giegerich, Stefan Kurtz)

94-04 Die Verwendung unscharfer Maße zur Korrespondenzanalyse in Stereo Farbbildern
(André Wolfram, Alois Knoll)

94-05 Searching Correspondences in Colour Stereo Images – Recent Results Using the Fuzzy Integral
(André Wolfram, Alois Knoll)

94-06 A Basic Semantics for Computer Arithmetic
(Markus Freericks, A. Fauth, Alois Knoll)

94-07 Reverse Restructuring: Another Method of Solving Algebraic Equations
(Bernd Bütow, Stephan Thesing)

95-01 PaNaMa User Manual V1.3
(Bernd Bütow, Stephan Thesing)

95-02 Computer Based Training-Software: ein interaktiver Sequenzierkurs
(Frank Meier, Garrit Skrock, Robert Giegerich)

95-03 Fundamental Algorithms for a Declarative Pattern Matching System
(Stefan Kurtz)

95-04 On the Equivalence of E-Pattern Languages
(Enno Ohlebusch, Esko Ukkonen)

96-01 Static and Dynamic Filtering Methods for Approximate String Matching
(Robert Giegerich, Frank Hischke, Stefan Kurtz, Enno Ohlebusch)

96-02 Instructing Cooperating Assembly Robots through Situated Dialogues in Natural Language
(Alois Knoll, Bernd Hildebrand, Jianwei Zhang)

96-03 Correctness in System Engineering
(Peter Ladkin)

96-04 An Algebraic Approach to General Boolean Constraint Problems
(Hans-Werner Güsgen, Peter Ladkin)

96-05 Future University Computing Resources
(Peter Ladkin)

96-06 Lazy Cache Implements Complete Cache
(Peter Ladkin)

96-07 Formal but Lively Buffers in TLA+
(Peter Ladkin)

96-08 The X-31 and A320 Warsaw Crashes: Whodunnit?
(Peter Ladkin)

96-09 Reasons and Causes
(Peter Ladkin)

96-10 Comments on Confusing Conversation at Cali
(Dafydd Gibbon, Peter Ladkin)

96-11 On Needing Models
(Peter Ladkin)

96-12 Formalism Helps in Describing Accidents
(Peter Ladkin)

96-13 Explaining Failure with Tense Logic
(Peter Ladkin)

96-14 Some Dubious Theses in the Tense Logic of Accidents
(Peter Ladkin)

96-15 A Note on a Note on a Lemma of Ladkin
(Peter Ladkin)

96-16 News and Comment on the AeroPeru B757 Accident
(Peter Ladkin)

97-01 Analysing the Cali Accident With a WB-Graph
(Peter Ladkin)

97-02 Divide-and-Conquer Multiple Sequence Alignment
(Jens Stoye)

97-03 A System for the Content-Based Retrieval of Textual and Non-Textual Documents Based on Natural Language Queries
(Alois Knoll, Ingo Glöckner, Hermann Helbig, Sven Hartrumpf)

97-04 Rose: Generating Sequence Families
(Jens Stoye, Dirk Evers, Folker Meyer)

97-05 Fuzzy Quantifiers for Processing Natural Language Queries in Content-Based Multimedia Retrieval Systems
(Ingo Glöckner, Alois Knoll)

97-06 DFS – An Axiomatic Approach to Fuzzy Quantification
(Ingo Glöckner)

98-01 Kognitive Aspekte bei der Realisierung eines robusten Robotersystems für Konstruktionsaufgaben
(Alois Knoll, Bernd Hildebrandt)

98-02 A Declarative Approach to the Development of Dynamic Programming Algorithms, applied to RNA Folding
(Robert Giegerich)

98-03 Reducing the Space Requirement of Suffix Trees
(Stefan Kurtz)

99-01 Entscheidungskalküle
(Axel Saalbach, Christian Lange, Sascha Wendt, Mathias Katzer, Guillaume Dubois, Michael Höhl, Oliver Kuhn, Sven Wachsmuth, Gerhard Sagerer)

99-02 Transforming Conditional Rewrite Systems with Extra Variables into Unconditional Systems
(Enno Ohlebusch)

99-03 A Framework for Evaluating Approaches to Fuzzy Quantification
(Ingo Glöckner)

99-04 Towards Evaluation of Docking Hypotheses using elastic Matching
(Steffen Neumann, Stefan Posch, Gerhard Sagerer)

99-05 A Systematic Approach to Dynamic Programming in Bioinformatics. Part 1 and 2: Sequence Comparison and RNA Folding
(Robert Giegerich)

99-06 Autonomie für situierte Robotersysteme – Stand und Entwicklungslinien
(Alois Knoll)

2000-01 Advances in DFS Theory
(Ingo Glöckner)

2000-02 A Broad Class of DFS Models
(Ingo Glöckner)

2000-03 An Axiomatic Theory of Fuzzy Quantifiers in Natural Languages
(Ingo Glöckner)

2000-04 Affix Trees
(Jens Stoye)

2000-05 Computergestützte Auswertung von Spektren organischer Verbindungen
(Annika Büscher, Michaela Hohenner, Sascha Wendt, Markus Wiesecke, Frank Zöllner, Arne Wegener, Frank Bettenworth, Thorsten Twellmann, Jan Kleinlützum, Mathias Katzer, Sven Wachsmuth, Gerhard Sagerer)

2000-06 The Syntax and Semantics of a Language for Describing Complex Patterns in Biological Sequences
(Dirk Strothmann, Stefan Kurtz, Stefan Gräf, Gerhard Steger)

2000-07 Systematic Dynamic Programming in Bioinformatics (ISMB 2000 Tutorial Notes)
(Dirk J. Evers, Robert Giegerich)

2000-08 Difficulties when Aligning Structure Based RNAs with the Standard Edit Distance Method
(Christian Büschking)

2001-01 Standard Models of Fuzzy Quantification
(Ingo Glöckner)

2001-02 Causal System Analysis
(Peter B. Ladkin)

2001-03 A Rotamer Library for Protein-Protein Docking Using Energy Calculations and Statistics
(Kerstin Koch, Frank Zöllner, Gerhard Sagerer)

2001-04 Eine asynchrone Implementierung eines Microprozessors auf einem FPGA
(Marco Balke, Thomas Dettbarn, Robert Homann, Sebastian Jaenicke, Tim Köhler, Henning Mersch, Holger Weiss)

2001-05 Hierarchical Termination Revisited
(Enno Ohlebusch)

2002-01 Persistent Objects with O2DBI
(Jörn Clausen)

2002-02 Simulation von Phasenübergängen in Proteinmonoschichten
(Johanna Alichniewicz, Gabriele Holzschneider, Morris Michael, Ulf Schiller, Jan Stallkamp)

2002-03 Lecture Notes on Algebraic Dynamic Programming 2002
(Robert Giegerich)

2002-04 Side chain flexibility for 1:n protein-protein docking
(Kerstin Koch, Steffen Neumann, Frank Zöllner, Gerhard Sagerer)

2002-05 ElMaR: A Protein Docking System using Flexibility Information
(Frank Zöllner, Steffen Neumann, Kerstin Koch, Franz Kummert, Gerhard Sagerer)

2002-06 Calculating Residue Flexibility Information from Statistics and Energy based Prediction
(Frank Zöllner, Steffen Neumann, Kerstin Koch, Franz Kummert, Gerhard Sagerer)

2002-07 Fundamentals of Fuzzy Quantification: Plausible Models, Constructive Principles, and Efficient Implementation
(Ingo Glöckner)

2002-08 Branching of Fuzzy Quantifiers and Multiple Variable Binding: An Extension of DFS Theory
(Ingo Glöckner)

2003-01 On the Similarity of Sets of Permutations and its Applications to Genome Comparison
(Anne Bergeron, Jens Stoye)

2003-02 SNP and mutation discovery using base-specific cleavage and MALDI-TOF mass spectrometry
(Sebastian Böcker)

2003-03 From RNA Folding to Thermodynamic Matching, including Pseudoknots
(Robert Giegerich, Jens Reeder)

2003-04 Sequencing from compomers: Using mass spectrometry for DNA de-novo sequencing of 200+ nt
(Sebastian Böcker)

2003-05 Systematic Investigation of Jumping Alignments
(Constantin Bannert)

2003-06 Suffix Tree Construction and Storage with Limited Main Memory
(Klaus-Bernd Schürmann, Jens Stoye)

2003-07 Sequencing from compomers in thepresence of false negative peaks
(Sebastian Böcker)

2003-08 Genalyzer: An Interactive Visualisation Tool for Large-Scale Sequence Matching – Biological Applications and User Manual
(Jomuna V. Choudhuri, Chris Schleiermacher)

2004-01 Sequencing From Compomers is NP-hard
(Sebastian Böcker)

2004-02 The Money Changing Problem revisited: Computing the Frobenius number in time

$O(k\,a_1)$

(Sebastian Böcker, Zsuzsanna Lipták)

2004-03 Accelerating the Evaluation of Profile HMMs by Pruning Techniques
(Thomas Plötz, Gernot A. Fink)

2004-04 Optimal Group Testing Strategies with Interval Queries and Their Application to Splice Site Detection
(Ferdinando Cicalese, Peter Damaschke, Ugo Vaccaro)

2004-05 Compressed Representation of Sequences and Full-Text Indexes
(Paolo Ferragina, Giovanni Manzini, Veli Mäkinen, Gonzalo Navarro)

2005-01 Overlaps Help: Improved Bounds for Group Testing with Interval Queries
(Ferdinando Cicalese, Peter Damaschke, Libertad Tansini, Sören Werth)

2005-02 Two batch Fault-tolerant search with error cost constraints: An application to learning
(Ferdinando Cicalese)

2005-03 Searching for the Shortest Common Supersequence
(Sergio A. de Carvalho Jr., Sven Rahmann)

2005-04 Counting Suffix Arrays and Strings
(Klaus-Bernd Schürmann, Jens Stoye)

2005-05 Alignment of Tandem Repeats with Excision, Duplication, Substitution and Indels (EDSI)
(Michael Sammeth, Jens Stoye)

2005-06 Statistics of Cleavage Fragments in Random Weighted Strings
(Hans-Michael Kaltenbach, Henner Sudek, Sebastian Böcker, Sven Rahmann)

2006-01 Decomposing metabolomic isotope patterns
(Sebastian Böcker, Zsuzsanna Lipták, Anton Pervukhin)

2006-02 On Common Intervals with Errors
(Cedric Chauve, Yoan Diekmann, Steffen Heber, Julia Mixtacki, Sven Rahmann, Jens Stoye)

2007-01 Identifying metabolites with integer decomposition techniques, using only their mass spectrometric isotope patterns
(Sebastian Böcker, Matthias C. Letzel, Zsuzsanna Lipták, Anton Pervukhin)

2007-02 2-Stage Fault Tolerant Interval Group Testing
(Ferdinando Cicalese, José Augusto Amgarten Quitzau)

2008-01 Online Abelian Pattern Matching
(Tahir Ejaz, Sven Rahmann, Jens Stoye)