Sequences, Maps, Genomes, and Graphs:  
Graph Compression Algorithms for Efficiently Comparing Genomes

Steve Goldstein\textsuperscript{1} \textsuperscript{2}  
Adam Briska\textsuperscript{3}  
Shiguo Zhou\textsuperscript{2}  
David C. Schwartz\textsuperscript{2}  

Technical Report #181  
Biostatistics and Medical Informatics, University of Wisconsin  
March, 2004

\textsuperscript{1}Email address: steveg@lmcg.wisc.edu.  
\textsuperscript{2}Laboratory for Molecular and Computational Genomics, University of Wisconsin—Madison, Madison, WI 53706  
\textsuperscript{3}OpGen, Inc., 510 Charmany Road, Madison, WI 53719
Abstract

We present a novel algorithm for finding repeats in a text. Any text written in a finite alphabet has a representation as a path in the de Bruijn graph. Our graph compression algorithm (GCA) constructs the adjacency list representation of a directed word graph which is equivalent to the original de Bruijn graph and has the minimal number of vertices. Certain edges of this graph correspond to the maximal repeats in the text, and thus, the adjacency list can be efficiently queried to answer questions about the repeat structure of the text.

Our primary interest in repeat finding lies in identifying matching regions in whole-genome restriction maps. Because of the imprecision in determining restriction fragment lengths, this task requires finding approximate matches. We cast the problem as one of finding regions in the graph where paths are close to each other and then use GCA to solve the problem efficiently.

The task of finding matches in restriction maps is a variation of that of finding maximal exact matches (MEMs) for seeding DNA sequence alignments. We give a version of GCA that can identify MEMs, using a data structure that requires less memory than a virtual suffix array implementation.

Finding repeats in a long text is a key component of many bioinformatics applications. GCA, a general method for performing this task, can be a useful addition to bioinformatics tool kits.
1 Introduction

Optical mapping is a robust system for producing ordered restriction maps from individual genomic DNA molecules [8]. An ensemble of single molecule maps can be assembled to yield a highly accurate consensus map of a whole genome [5]. Such a restriction map provides a description of an organism’s genome, a description not unlike the DNA sequence of the genome, albeit at a coarser resolution. Just as comparisons of genome sequences are leading to an exciting array of biological advances, comparisons of optical maps will provide a wealth of valuable information.

Optical mapping has entered the high-throughput era [9], and, thus, software tools for comparing maps are a necessary component of the system. In this paper, we present a novel class of graph-based algorithms and data structures, which we call Graph Compression Algorithm (GCA), that provide a principal component for efficient solutions to a range of computational tasks, including both comparison of whole-genome restriction maps and multiple genome alignment (of DNA sequences).

1.1 Comparing sequences and maps

1.1.1 Finding maximal matches in genome sequences

A common method for aligning genome sequences first finds suitable “anchors,” maximal unique matches (MUMs) [2] or maximal exact matches (MEMs) [4], on which to base the alignments. Typically this method is implemented using a suffix tree or one of its variants. In essence, the suffix tree is a data structure which contains an isomorphic representation of the input sequences and which can be efficiently searched to find the exact matches in the input.

The de Bruijn graph is another structure that has been used to isomorphically represent DNA sequences [6, 7]. This is a directed graph whose vertices are sequences of a fixed length \( l - 1 \) and whose edges are the sequences of length \( l \) which might overlap. The input sequences correspond to paths in this graph; MUMs and MEMs of length \( l \) or greater correspond to certain regions in the graph where the sequence paths coincide. A key advantage in this graphical representation is that MUMs and MEMs (of length \( l \) or greater), as well as many other repetitive features of the input, can be characterized in terms of the local topology of the graph.

GCA compresses the de Bruijn graph representation into an equivalent one that preserves the topology of the graph and, thus, the repeat structure of the input. While the complexity of the de Bruijn graph representation (i.e. the number of vertices) is of the order of the length of the input, the complexity of the compressed
graph is of the order of the number of repeats. GCA achieves its efficiency by only implicitly using the de Bruijn graph representation for the explicit construction of the compressed graph. For genome comparison problems, the resulting savings can be considerable.

**Space-efficient implementation**  One important line of research in genome alignment has been to improve the space requirements of the algorithm. The most space-efficient method previously published is given in [4], where the authors were able to detect MEMs with a data structure — an enhanced suffix — that requires 5 bytes of storage per input character. We prove that GCA can perform the same task using a data structure with space complexity proportional to the number of maximal repeats and whose construction requires about 4 bytes per input character.

1.1.2 Comparing maps

The computational task of comparing whole-genome restriction maps can be posed in a manner analogous to that of comparing DNA sequences. A map is determined by the sequence of lengths (in base pairs) between consecutive enzyme recognition sites. In other words, a map can be thought of as a text written in the alphabet consisting of lengths of its restriction fragments, and the task, in both cases, amounts to finding matching regions in a text. However, the two types of text differ in scale and precision; these differences necessitate changes in the solution.

The number of restriction sites recognized by a given enzyme in a genome is typically several orders of magnitude smaller than the number of nucleic acids. This difference in scale has two implications. First, the information obtained from comparing genomes will be of a coarser resolution. For example, a single base pair difference is detectable only if it occurs at a restriction site. On the other hand, often the desired output of a genome comparison algorithm—identification of large homologous regions and their relative rearrangements—is on a scale congruous with the scale of restriction maps. A second implication is that the implementation need not be optimized for space efficiency, although the space requirements cannot be ignored: The algorithm will be used for comparing maps of mammalian-size genomes and for comparing large numbers of closely related microbial genomes as the maps of these genomes are obtained.

In the current generation of the optical mapping system, the lengths of the restriction fragments are determined by fluorescence microscopy, yielding a precision typically in the range of several hundred base pairs for the consensus map.

To handle this imprecision in the input, we replace the maximal exact matching component of the sequence comparison algorithm with a method that can detect approximate matches described as follows. First, we define a maximal accurate
match (MAM). Informally, an accurate match of a collection of maps corresponds to a region in the de Bruijn graph where the paths representing those maps come “close” to each other. We prove that the accurate matches that are maximal can be characterized in terms of the local topology of the graph. Thus, we can apply a variation of GCA to obtain a feasible algorithm for detecting all of the MAMs in a collection of maps.

2 Notation, Definitions and Some Basics of the Graph

2.1 Definitions and notation

Let $S = \{g_1, g_2, \ldots, g_k\}$ be a set of strings written in the finite alphabet $\Sigma = \{a_1, a_2, \ldots, a_\sigma\}$. Of course, we think of $g_i$ as the sequence of the $i^{th}$ genome when $\Sigma = \{A,C,G,T\}$ and as its restriction map when $\Sigma$ is a bounded subset of the positive integers.

Assume that $|g_i| \geq l$ for every $1 \leq i \leq k$.

**Definition (De Bruijn graph).** Define the de Bruijn graph $G(S^l)$ to be the directed graph with vertex set $S^l$ (the set of all $(l-1)$-tuples from $S$) and edge set $S^l$ such that the directed edge $e \in S^l$ joins $v \in S^l$ with $w \in S^l$ if $v$ is a prefix of $e$ and $w$ is a suffix of $e$.

Note that $G(S^l)$ is a subgraph of $G_{\sigma,l}$, the well-known de Bruijn graph of order $\sigma$ and dimension $l$.

**Definition (Paths in the graph).** Let $s$ be a substring of $g_i$, $|s| \geq l$. By sliding a window of length $l-1$ along $s$, we see that $s$ defines a path $p(s) = v_0, \ldots, v_n$ in $G(S^l)$ that visits $n_s = |s| - l + 1$ edges and $n_s + 1$ vertices. Call $v_0$ and $v_n$ the starting and stopping vertices of $s$. Let $\pi(S)$ be the set of paths $\{p(g_1), \ldots, p(g_k)\}$.

**Example 1.** Let $g_1$ be the sequence TAATCGA and $g_2$ be the sequence CATCGG. The path $p(g_1)$ in $G(S^3)$ is

$$\text{TA} \rightarrow \text{AA} \rightarrow \text{AT} \rightarrow \text{TC} \rightarrow \text{CG} \rightarrow \text{GA}$$

and $p(g_2)$ is

$$\text{CA} \rightarrow \text{AT} \rightarrow \text{TC} \rightarrow \text{CG} \rightarrow \text{GG}$$

See figure 1.
Definition (Graph topology). A vertex \( v \) is called a branching vertex if \( \text{indegree}(v) \cdot \text{outdegree}(v) > 1 \). A vertex is called topologically non-trivial (or more simply, non-trivial) if it is either a branching vertex or a starting or stopping vertex of \( g_i \) for some \( i \). A trivial vertex is one that is not non-trivial.

Definition (Repeated strings). A string \( r \), \( |r| \geq l \) is a repeat if it is a substring that occurs in two different locations in \( S \): either in two different locations in a single genome or in two different genomes. A repeat is specified by its length and the two locations where it starts.

A repeat is left-maximal if the character immediately preceding \( r \) differs at those two locations or if one of the locations is at the beginning of a genome. It is right-maximal if the immediately following character differs or if one of the locations is at the end of the genome. A repeat is maximal if it is both left- and right-maximal [3, page 143].

Definition (2-path). A 2-path is a directed subpath consisting of three consecutive vertices \( u, v, w \) (visiting two consecutive edges). Call \( v \) the midpoint of the 2-path. A 2-path is uniquely specified by its midpoint and its first and last characters, \( (v, x, y) \in S^{l-1} \times \Sigma \times \Sigma \); the \( l \)-tuples \( xv \) and \( vy \) are its edges.

Example 2. Let \( g_1 \) and \( g_2 \) be the sequences defined in example 1. The branching vertices are AT and CG. The trivial vertices are AA and TC.

2.2 Key observations

We can now state a few obvious facts relating the repeats in the genome to the topology of the graph. These simple observations are the main ideas underlying our algorithms.
1. The relationship between the subpaths of $\pi(S)$ and substrings (of length at least $l$) of $g_i$, $1 \leq i \leq k$, is an isomorphism.

2. The subpath $p(r)$ corresponding to a repeat $r$ is traversed at least twice in $\pi(S)$.

3. The starting (stopping) vertex of a left- (right-) maximal repeat is branching.

4. Every branching vertex is the starting or stopping vertex of a left- or right-maximal repeat.

5. The number of branching vertices is of the same order of magnitude as the number of left- or right-maximal repeats.

6. A vertex $v$ is branching if and only if it is the midpoint of two distinct$^1$ 2-paths of $\pi(S)$.

Example 3. (Continuing example 1.) The 2-paths (AT, A,C) and (AT, C,C) demonstrate that AT is branching.

From observation 6, we can derive what is arguably the central idea behind GCA. Sorting all the 2-paths — the $(l+1)$-tuples — on their midpoints reveals much about the local topology of the de Bruijn graph, which, in turn, reveals much about the repeat structure of the input.

3 Algorithms — Overview

The Graph Compression Algorithm takes $S = \{g_1, \ldots, g_k\}$ as input (of total length $n$, say) and produces a compressed graph, the directed word graph $\hat{G}(S)$, as output. $\hat{G}(S)$ satisfies the following three properties: (1) The vertex set of $\hat{G}(S)$ is precisely the set of non-trivial vertices of $G(S^l)$. (2) There exists a directed edge joining a vertex $v$ with a vertex $w$ if and only if there is a path in $G(S^l_i)$ $v = v_0, v_1, \ldots, v_t = w$ with $v_i$ trivial for $1 \leq i < t$. (3) The edge joining $v$ with $w$ is labeled with the substring corresponding to the path $v_0, v_1, \ldots, v_t$.

In simple words, $\hat{G}(S)$ is a graph with the same topology as the de Bruijn graph of $S$ and whose edges are labeled either with the maximal unique substrings of $S$ or with left- or right-maximal repeats.

$^1$A maximal repeat of length $l - 1$ is a branching vertex. However, with respect to the task “find all repeats of length at least $l,” we may wish to classify it as topologically trivial and call it a false branching vertex. An alternate definition of branching reflecting this can be given by: $v$ is branching if it is the midpoint of either two distinct 2-paths $(v, x, y)$ and $(v, x, y')$ or two distinct 2-paths $(v, x, y)$ and $(v, x', y)$. In this exposition, we will sacrifice correctness for clarity and ignore the distinction. Implementation details can be adjusted to yield correct results.
Example 4. (Continuing example 1.) The path \( p(g_1) \) in \( \hat{G}(S) \) is
\[
\text{TA} \xrightarrow{TAAT} \text{AT} \xrightarrow{ATCG} \text{CG} \xrightarrow{CGA} \text{GA}
\]
and \( p(g_2) \) is
\[
\text{CA} \xrightarrow{CAT} \text{AT} \xrightarrow{ATCG} \text{CG} \xrightarrow{CGG} \text{GG}
\]
The edge labels are indicated.

The following theorem ensures that \( \hat{G}(S) \) is well-defined.

**Theorem (Edge uniqueness).** Let \( a \) and \( b \) be branching vertices in \( G(S') \). Let \( p \) and \( q \) be two paths in \( G(S') \) from \( a \) to \( b \). \( p = a, v_1, \ldots, v_t, b \) and \( q = a, w_1, \ldots, w_s, b \). Suppose that \( v_i \) is trivial for all \( 1 \leq i \leq t \). If \( v_1 = w_1 \), then \( p = q \).

**Proof.** If \( p \neq q \), let \( v_k \neq w_k \) be the first vertex where the paths differ. Then the 2-paths \( v_{k-2}, v_{k-1}, v_k \) and \( w_{k-2}, w_{k-1}, w_k = v_{k-2}, v_{k-1}, w_k \) are distinct (defining \( v_0 = w_0 = a \) if necessary). Thus, \( v_{k-1} \) is branching, contrary to assumption. \( \square \)

GCA is can be implemented in two steps, each involving a linear scan of the input. In the first scan, the non-trivial vertices of \( G(S') \) are identified. In the second scan, the edge labels are computed. Both steps are simple and can be implemented in a straightforward manner.

**Sort the 2-paths** We achieve the first step by taking a sliding window of length \( l + 1 \), enumerating and sorting the \((l + 1)\)-tuples in \( S \). However, we implicitly appeal to the de Bruijn graph \( G(S') \), regarding the tuples as 2-paths, and sort on the midpoint. By property 6 of section 2.2, a vertex can be classified as branching or non-branching in constant time as a final step in sorting the 2-paths. The list of non-trivial vertices is completed by adding the 2 \( k \) starting and stopping vertices of the genomes. The list of non-trivial vertices can be output in a hash table requiring space proportional to the number of such vertices, which is of the order of magnitude of the number of left- or right-maximal repeats.

If the vertex sort is implemented using a radix sort, it can be done in time \( O((l + 1)n) \) and space \( 8n \) bytes. Observe, however, that the objective of the sort is to identify branching vertices, not to output a sorted list. The modified radix sort presented in the next section exploits this observation and uses half as much space.

Sometimes it is advantageous to implement the 2-path sort via hashing. For a 2-path \((v, x, y)\), take the midpoint \( v \) as the hash key and the pair of characters \((x, y)\) as the hash value. A branching vertex can be identified on the fly at the first “witness” to its branching, i.e., the first time \( v \) occurs as the midpoint of a second distinct 2-path. With a good hash function, this may perform better than a radix-sort implementation.
**Build the graph**  The second step of GCA takes as input \( S \) and the hash table of non-trivial vertices, \( V_{NT} \) and outputs the graph \( \hat{G}(S) \) in the form of an adjacency list. Since \( V_{NT} \) exactly contains the vertices of \( \hat{G}(S) \), we use it to store the information accumulated in this step, transforming it into the adjacency list.

We again scan through \( S \) with a sliding window implicitly traversing the de Bruijn graph. This implicit traversal implies a traversal of \( \hat{G}(S) \) so we can construct the graph by the formula: (1) skip over trivial vertices of \( G(S') \); (2) every time we visit a non-trivial vertex, add a directed edge in \( \hat{G}(S) \) joining the previous non-trivial vertex with this one and label the edge with the substring corresponding to the path traversed in \( G(S') \).

While that formula is correct, it redundantly computes edge labels. To avoid that extra work, observe that if we have already constructed an edge \( e \) in \( \hat{G}(S) \), we do not have to traverse \( G(S') \) (i.e., slide the window) on subsequent visits to \( e \). We can just follow \( e \) in \( \hat{G}(S) \) to the next vertex. By edge uniqueness theorem, we can detect this event by looking ahead one character.

**Analysis**  The graph-building step does a constant amount of work for each \((l-1)\)-tuple (or none at all), and so it is \( O(n) \) in time. The space required is that which is needed for \( V_{NT} \). For each vertex \( v \), we need to store each edge that leaves \( v \). Each edge has the following data: length, terminal vertex, and the location of each of its instances. The space required is proportional to the number of edges in \( \hat{G}(S) \) plus the number of occurrences in \( S \) of the non-trivial vertices. In genome comparison problems, we expect this to be a few orders of magnitude smaller than \( n \). At worst, it is \( O(n) \).

## 4 Multiple Genome Alignment of DNA Sequences

In this section we show how to identify maximal exact matches in a space-efficient manner using the graph reduction algorithm. The bound is close to or beats the bound in the MGA algorithm.

### 4.1 Finding Multiple Maximal Exact Matches

A multiple exact match is a repeated substring that occurs in every genome. A multiple maximal exact match (multiMEM) [4] is a multiple exact match that cannot be simultaneously extended in every genome. Formally, we define multiMEM in terms of our pairwise definition of a repeat.

**Definition (multiMEM).** A **multiMEM** is a length \( t \) and a set of locations \( L_1, \ldots, L_k \) such that (1) each \( L_i \) specifies a location in \( g_i \); (2) for each pair \( i, j (i \neq j) \), \( (t, L_i, L_j) \)
is a repeat; (3) for at least one pair, it is left-maximal; and (4) for at least one pair, it is right-maximal.

A multiMEM is bounded by the pair of non-trivial vertices implied by (3) and (4). In a single traversal of $\hat{G}(S)$, we can identify all such pairs of vertices and match them. We omit the details.

4.2 Modified Radix Sort

A radix tree [1, page 269] can be used to sort a set of strings of a fixed length $t$. This is a tree of depth $t + 1$ and in which each node has $\sigma = |\Sigma|$ children. Each branch is labeled with a character from $\Sigma$ and the branches are in lexicographical order. Thus, the labels of the nodes at level $i$ ($0 < i \leq t$) give the strings of length $i$ in order.

For simplicity, assume we have one genome so that $S = g_1$ is a string of length $n$. We sort the $(l - 1)$-tuples in $S$ by populating the radix tree with them in a depth-first construction. At any stage in the construction, the tuples populating the subtree rooted at a node $N$ have $N$’s label as a prefix and are maintained in a linked list. The construction proceeds by the processing the nodes with procedure `refineNode()`. `refineNode(N)` iterates through $N$’s list of tuples, splitting it into $\sigma$ separate lists, populating $N$’s children and then recursively calling `refineNode()` on the lexicographically smallest child. When we reach a leaf node, we can examine the 2-paths having that node’s label as a midpoint and determine whether or not the vertex is branching. If it is branching, enter it into $V_{NT}$.

Since each tuple is in exactly one of these linked lists at any given stage in the construction, we can store all the lists in one array of length $n - l + 1$. To maintain the lists, we need to store indexes into this array that point to the head of each of the lists. However, once we have processed a leaf node, we no longer need to use its list. At any stage in the construction, there are at most $\sigma$ nodes at each of the $l - 1$ levels that have not been processed. Assuming each of the indexes and list elements require 4 bytes, the storage requirement is $4(n + \sigma(l - 1)) + O(#\text{non-trivial vertices})$. The time requirement is $O((l + 1)n)$, because we look at each position of every $(l + 1)$-tuple (including the 2-path examination), doing a constant amount of work each time.

5 Multiple Genome Alignment of Restriction Maps

In optical mapping, DNA molecules are elongated, immobilized on a surface, and digested with a restriction enzyme. They are then stained with a fluorescent dye and the images are analyzed. In this way, ordered restriction maps of single DNA
molecules are produced. An ensemble is assembled to produce a consensus map of the genome, in a manner analogous to sequence assembly.

To compare maps of closely related organisms (digested with the same enzyme), we adapt the sequence comparison algorithm to be tolerant of the imprecision in the maps. Since a map is given by a string of integers, many of the concepts and some of the definitions from section 3 apply. In particular, the de Bruijn graph of a set of maps is well-defined. However, the definition of repeat needs to be replaced.

**Definition.** An $\varepsilon$-repeat is a pair of distinct substrings $r_1, r_2, \ldots, r_t$ and $s = s_1s_2\ldots s_t$, $t \geq l$ satisfying $|r_i - s_i| \leq \varepsilon$ for all $1 \leq i \leq t$. A repeat is specified by its length and the two locations where it starts.

We build on this definition in modifying others.

**Definition.** A pair of vertices is $\varepsilon$-branching if it is an $\varepsilon$-repeat and the vertices are midpoints of a pair of 2-paths that is not an $\varepsilon$-repeat.

**Definition.** A vertex is $\varepsilon$-trivial if it is not a member of any $\varepsilon$-branching pair and is not a starting or stopping vertex.

**Definition.** An $\varepsilon$-multiple accurate match (multiMAM) is a length $t$ and a set of locations $L_1,\ldots,L_k$ such that (1) each $L_i$ specifies a location in $g_i$; (2) for each pair $i, j$ $(i \neq j)$, $(t,L_i,L_j)$ is an $\varepsilon$-repeat; (3) for at least one pair, it is left-maximal; and (4) for at least one pair, it is right-maximal.

The analog of the edge-uniqueness theorem implies that we will be able to compute multiMAMs just by considering $\varepsilon$-nontrivial vertices.

**Theorem.** Let $(a^1, a^2)$ and $(b^1, b^2)$ be $\varepsilon$-branching. Let $p^j$ be a path from $a^j$ to $b^j$, $p^j = a^j, v_1^j, \ldots, v_t^j, b^j$ and let $e^j$ be the edge joining $a^j$ to $v_1^j$, for $j = 1, 2$. Suppose that $v_i^j$ is $\varepsilon$-trivial for all $1 \leq i \leq t$. If $(e^1, e^2)$ is an $\varepsilon$-repeat, so is $(p^1, p^2)$.

**Proof.** This can be proven in a manner similar to the proof of the edge uniqueness theorem.

To enable GCA in this case, we show how to efficiently identify the $\varepsilon$-branching pairs. To picture this, think of a vertex as a point in $\mathbb{R}^{l-1}$.

First, bin the lengths — the characters in $\Sigma$ — into bins of size $\varepsilon$. This binning partitions $\mathbb{R}^{l-1}$ into $(l-1)$-dimensional $\varepsilon$-boxes. If we replace each vertex with the center of the box it occupies, we can sort them. Now if $(v, w)$ is an $\varepsilon$-repeat, we

---

$^2$Note the dual nature of a “character” from the alphabet $\Sigma$, the lengths of the restriction fragments. We treat it as both a single letter in the alphabet and an integer.
must be in v’s box or in one of the $3^{l-1} - 1$ adjacent boxes. Thus we can compute all of the $\varepsilon$-branching pairs by considering the 2-paths of all pairs of vertices that fall into neighboring boxes.

However, a naive implementation of this scales exponentially in $l$, limiting its practicality. On the other hand, the inherent complexity of the problem is not the number of boxes neighboring a given box, but the number of occupied boxes. Adapting the radix tree approach from section 4, we overcome this potential pitfall in our implementation by searching only the non-empty subtrees of neighboring nodes.

### 6 Results and Future Work

This work is in its early stages. We have two examples implementing some of the work discussed here. We have software for identifying and a GUI for visualizing MAMs in a pair of whole-genome maps. We have also an implementation of CGA for correcting isolated base-calling errors from a shotgun sequencing project [7]. An immediate goal is to produce a fully-functional comparison tool for optical maps to replace the laborious methods now being used.

### Acknowledgments

Steve Goldstein is supported by NLM training grant, number T15 LM07359, Computation and Informatics in Biology and Medicine. We would like to thank Clotho Advanced Media for limited support for portions of this research.

### References


