UiO : Department of Informatics
University of Oslo

Unfolding Your DNA

Handling Chromatin 3D Data in Computer Programs

Tobias Gulbrandsen Waaler

Master's Thesis, Spring 2013

Unfolding Your DNA

Tobias Gulbrandsen Waaler

May 1, 2013

Abstract

Recent progress in experimental methods has enabled probing of the spatial organization of DNA. Several analyses have indicated that the spatial organization of DNA is non-random and is related to its function.

There is a grand potential for analyzing this data, but the currently available computational tools are lacking in their support. Chromatin 3D data represents a new paradigm in genomics and requires the development of new methods for analysis and interpretation. How this data is represented in computer programs lays the foundation for all further use. Not only does it affect the performance and efficiency of all computations, but it also sets the premises for a programming interface and the ways in which the data can be accessed.

This thesis is an account of the observations made when developing the functionality of a chromatin 3D data analysis for The Genomic HyperBrowser, a web-based tool for genomic computations. Different ways of handling chromatin 3D data are evaluated, with a particular focus on performance and usability. Suggestions and remarks are then made as to how chromatin 3D data can successfully be handled in HyperBrowser specifically, and in computer programs generally.

The effort led to significant performance improvements in a chromatin 3D data analysis performed in HyperBrowser.

In conclusion, performing analyses on the currently available chromatin 3D data is practically feasible through careful design and implementation of data structures and algorithms. However, as experimental methods improve, the increasing size of the data sets will pose new challenges to the computational methods involved.

Contents

List of Figures

List of Tables

Acknowledgments

First of all I would like to thank my supervisors *Geir Kjetil Sandve* and *Eivind Hovig* for exposing me to interesting challenges and for the guidance they have provided throughout this process. I am also grateful to *Jonas Paulsen* for answering my numerous questions, to *Nils Damm Christophersen* for reminding me of the importance of writing as opposed to programming, and to *Sveinung Gundersen* and *Kai Trengereid* for providing technical support, among other things.

Thank you to all the master students on the 10th floor of "Ole Johan Dahls hus" for filling the past year with silly humor and enjoyable breaks in between the work.

And last but not least, I am grateful to my fiancé *Maria K. Dolmen* for her love and support.

Tobias G. Waaler
University of Oslo
May, 2013

Chapter 1

Introduction

The experimental methods for capturing the spatial structure of DNA are relatively new. Unlike the data produced by regular DNA sequencing where the linear structure of DNA is captured, the data produced by *Chromosome Conformation Capture* (3C) and other similar techniques captures the spatial structure of DNA. This data, referred to as *chromatin 3D data*, has an inherently spatial structure, and can be represented as a graph.

With new data comes new challenges to the computational methods that are used for analyzing it. Chromatin 3D data represents a new paradigm within computational genomics. With its inherently spatial structure it poses new challenges to the ways we interpret, manage and process genomic data. Some popular computational tools are closely tied to the traditional concept of linear genomic data, making them incompatible with the spatially structured chromatin 3D data. Extending or adapting existing tools to support spatial data can be challenging as it breaks one of the fundamental assumptions they were built upon. Constructing new methodology and computational tools capable of handling chromatin 3D data is essential to the progress of genomics as it lays the foundation for all further research on the spatial organization of chromatin. The performance of such computational tools is of great importance, and designing them is non-trivial. The large size of the chromatin 3D data sets magnifies the cost of inefficient operations, leaving some implementations practically unusable.

Although several studies have already been performed on chromatin 3D data with great success, the computational methods involved seem to be created on an ad-hoc basis to answer the research questions at hand. When computational tools are created this way the same problems and challenges are being solved several times by different people and in different projects. An argument can be made for the necessity of generalized computational tools capable of analyzing chromatin 3D data in different contexts and for different purposes.

In general, the efficiency of the computational methods used are not overly important. Time is always an important factor in scientific projects, and the time spent on developing and optimizing the computational methods could be spent doing possibly more important things. Sometimes the programs are created with a single goal in mind and might perform reasonably well for their limited purpose. As long as the computations are finishing within a reasonable time, improving their efficiency might not pay off.

However, if the computational methods are intended to be used by numerous scientists for a great number of computations, this picture changes drastically. The efficiency of the methods become increasingly important, as the time spent on optimizing the computational methods is time saved for potentially a lot of users. So is the case for a scientific tool such as HyperBrowser, which is intended to be used by numerous users. Because of this making sure the computations it performs are efficient is worthwhile.

When working with relatively small data sets our powerful modern computers will happily crunch away at the most inefficient programs and return within seconds. How the data is structured, retrieved and manipulated is of less importance, and many aspects of the computer program might be more important than performance. How easy the source code is to read and maintain for instance, or how well its user interface communicates with the user, or its correctness and reliability. But for *some* programs performance is a requirement that can make it or break it. For this type of program its performance is what makes it a success or a complete failure. The size of the chromatin 3D data sets makes computations involving this data fall in the latter category of programs where performance is an absolute requirement. Currently the size of a typical data set can range from 100 megabytes to 10 gigabytes, but the size of the data sets are growing at a quadratic rate as the experimental methods improve. It is not unrealistic to assume that the size of the data sets can approach terrabytes, and this will dramatically change the requirements to the programs that are working with such data. When working with data of this size the ways in which the data is structured does not only determine *how long* a computation will take, but it determines if a computation is practically feasible at all. In other words, developing efficient algorithms and data structures is about more than impatience and making computations fast. Making computations efficient is the only way to get results at all.

The choice of data structure lays the foundation for every operation it will be a part of, and so if this fundamental part of the design is "wrong" everything that relies on it will suffer. By examining different data structures and algorithms with respect to how they would affect different aspects of the computer program a well informed decision can be made. The aspects to consider includes running time, space requirements and ease of use, to name a few. In this thesis some of the possible ways to handle chromatin 3D data are surveyed and the findings and experiences from this work are presented.

Some of the source code written as a part of this project can be found at http://hyperbrowser.uio.no/dev2/static/downloads/TobiasGulbrandsenWaaler_master_thesis_supplementary_material.zip.

1.1 Problem Description

How data is structured sets the premises for how it can be accessed, and so the algorithms operating on a given data structure are important when evaluating the characteristics of a data structure.

Support for chromatin 3D data was added to HyperBrowser without altering the fundamentals of the system. After experiencing performance issues with the initial implementation, the need for further research to improve the performance became obvious. The prospects of growing data sets makes planning the most fundamental part of the system necessary before developing other parts of the system that relies on the interface provided.

The main research question in this thesis can be summarized as follows: *what is the best way to structure data from various Chromosome Conformation Capture experiments*. This statement is both imprecise and ambiguous, but it provides some sense of direction. Two things must be especially clarified for this to be useful: what does "to structure data" entail, and what constitutes the "best" solution?

In this context *to structure data* refers to the design of *data structures* for representation of the data at hand. This includes data representation in main memory as well as on disk. The data to be represented is the results from different experiments called Chromosome Conformation Capture and includes 3C, 5C and HiC. The data produced in these experiments can conceptually be represented as a weighted graph.

When it comes to the definition of what constitutes the *best* solution, the criteria are not strictly measurable. In addition to being able to perform computations efficiently there are a number of concerns that must be addressed, such as how easy it is to use the interface provided by the data structure. A complete collection of "guidelines" that should be considered when evaluating solutions can be listed as follows, somewhat in order of priority:

1. Optimize for low running times.
2. Don't optimize for low memory usage unless exhaustive memory usage prohibits the implementation from working at all.
3. A slow computation is better than no computation.
4. Provide good usability through the programming interface

Although the problem is quite general and applies to any computer program that needs to handle chromatin 3D data, the focus of this thesis is directed towards an implementation in HyperBrowser. This becomes apparent throughout the thesis, for instance from the focus on Python and NumPy, both of which are essential in todays implementation of HyperBrowser.

1.2 Method

The methods used throughout this thesis to gather experience and empirical results can be divided into three main categories: experiments, calculations and literature research.

The *experiments* are usually in the form of small programs performing a given task while measuring the running time and memory usage. Some of the experiments have been performed by altering parts of the HyperBrowser source code and measure the effects, while others have been constructed as stand-alone programs with little or no dependency on the HyperBrowser system. An important part of the work that underlies the findings in this thesis was the development of one of the first analysis in HyperBrowser to handle chromatin 3D data, described in chapter 3.

Some rough *calculations*, or *estimates*, are also included. They are usually imprecise and informal, and serve to “illustrate a point” rather than providing a proof.

The main concern in this thesis is the evaluation and comparison of different solutions for handling data from Chromosome Conformation Capture experiments. As some of the criteria for assessing the different solutions are not easily quantifiable or measurable an unavoidable part of the research is from relevant *literature*. This includes scientific articles, text books and various online resources.

1.3 Scope

Bioinformatics is an interdisciplinary field. Working within bioinformatics therefore involves studying a multitude of topics, and pursuing any one of them could result in interesting discoveries. However, in order to accomplish anything within the time frame of a master thesis the scope of the thesis must be limited. Some of my research questions are closely tied to the Python programming language, the HyperBrowser system and Chromatin Interaction Data, while others are of a more general nature. As far as possible the questions and results are aimed at a wide audience, but there is always a balancing act between the general and the specific.

The use of graphs is relevant to a number of applications and the use of Python [30] and SciPy [31] is widespread within the scientific community. The experience and results related to a combination of graphs, Python and SciPy can therefore be of interest to a wider audience and throughout this thesis I try to honor that by keeping both research questions and suggested solutions as broad as possible while still being relevant to the development of HyperBrowser and the representation of Chromosome Conformation Capture data.

Naturally most topics within this thesis could be covered more extensively, especially those relating to the biological properties of Chromatin Interaction Data where interesting research is taking place at this very moment. This is one of the topics considered outside the scope of this thesis, and is only briefly discussed in section 2.1 on page 7.

Another topic that could be covered in more detail is the vast field of Graph Theory. Although the graph as a *data structure* is a central part of this thesis, the relevance of the mathematical concepts that make up Graph Theory is limited. The references to Graph Theory throughout this thesis are mostly related to terminology for describing properties of graphs. Graph Theory is therefore only briefly covered in the background material, with emphasis on the relevant parts of its terminology.

1.4 Chapter Overview

This chapter overview offers a form of narrative to the thesis, explaining *why* and *when* the different chapters came to be. The structure of this thesis reflects the fragmented way in which the experimental work it is based upon has been carried out. An important observation is that there are many minor findings rather than a few important ones. Apart from the inevitable chapters with background material and discussions, this thesis consists of several shorter chapters each summarizing the findings from distinct endeavors.

The **background** chapter covers most of the preliminary knowledge for understanding the rest of the thesis.

Chapter 3 is a case study, and the observations made here are in many ways the foundation on which this thesis is based upon. It is a detailed account of the work that was carried out as a part of developing a hypothesis test using chromatin 3D data. By working with a case for chromatin 3D data that is of practical use some problems and challenges emerged as especially relevant.

The three following chapters, **chapter 4**, **chapter 5** and **chapter 6** go more into detail about different topics from the case study that was found to be especially important: The graph implementation in HyperBrowser was heavily reliant on the NumPy library, and so **chapter 4** focuses on the usage of NumPy and the generalized concept of vector programming. Optimizing certain operations was difficult, because the data was structured in such a flexible way. In **chapter 5** suggestions are made for how this flexibility can be balanced with the need for optimizations. Monte Carlo simulations play an important role in HyperBrowser and the effect this has on the performance is the topic of discussion in **chapter 6**.

Chapter 7 takes a step back and makes a generalization based on some of the observed patterns from the earlier chapters. Specifically it discusses how memory usage and running time seems to be connected, and explores the basis for this relationship.

Chapter 8 is an assessment of how well the graph database *Neo4j* is suited for representing chromatin 3D data. Investigating Neo4j was the first part of the research that found its way into the final thesis. The reason for experimenting with Neo4j at such an early stage was the idea that if the preliminary results were promising this could determine the direction of the thesis.

Chapter 2

Background

2.1 Bioinformatics and Biology

The field of biology has experienced tremendous progress with the introduction of computers and their computational capabilities. This has given rise to the interdisciplinary field of *bioinformatics* and *computational life sciences*, where a number of different disciplines come together.

One of the major breakthroughs in biological research came with the discovery of DNA, and later the ability to *sequence* it. While an important part of biological research still revolves around “physical” experiments and laboratory work, computational tools are becoming increasingly important. Some of these experiments are producing massive amounts of data that need to be analyzed.

Recent development of experimental methods to extract the spatial organization of chromatin has led to a number of interesting discoveries. The major breakthroughs came with the development of two such experimental methods: *Hi-C* and *ChIA-PET*. *Hi-C* was described by Lieberman-Aiden *et al.* [14] in 2009, while *ChIA-PET* was described by Fullwood *et al.* [6] the same year. Hi-C builds on the principles from *Chromosome Conformation Capture* (3C) and its variants *Circularized Chromosome Conformation Capture* (4C) and *Carbon-Copy Chromosome Conformation Capture* (5C). Both Hi-C and ChIA-PET measures the number of interactions between regions in a genome. This number was found to be a reasonably precise proxy for the spatial distance between the regions. The resulting data from these experiments will be referred to collectively as chromatin 3D data in this thesis.

Several significant discoveries have been made by employing these methods and studying the resulting data. One of the findings presented by Lieberman-Aiden *et al.* [14] was that the genome had a *fractal globule* structure. They were also able to observe the presence of two distinct compartments within the chromatin. The regions in one compartment was more likely to be in spatial proximity to other regions within the same compartment, than in spatial proximity to the regions in the other compartment. Another interesting discovery (by Fudenberg *et al.* [5]) found that the architecture of chromatin in human cells was related to chromosomal translocations and structural changes in DNA.

Together these findings, among others, confirm that the spatial structure of chromatin is linked to the biological function of the genome. Understanding this structure is therefore an important part of developing modern genomics.

```
##gtrack version: 1.0
##track type: linked genome partition
##edge weights: true
##uninterrupted data lines: true
##sorted elements: true
##no overlapping elements: true
###end  id  edges
####genome=hg19; seqid=chr5; start=0; end=46405641
1000000 chr5:1*1M   chr19:1*1M=.; chr19:2*1M=.;              chr19:3*1M=.;
2000000 chr5:2*1M   chr19:1*1M=.; chr19:2*1M=54.6154626708; chr19:3*1M=44.4972605245;
3000000 chr5:3*1M   chr19:1*1M=.; chr19:2*1M=20.7421661021; chr19:3*1M=16.8994186582;
4000000 chr5:4*1M   chr19:1*1M=.; chr19:2*1M=22.9421691221; chr19:3*1M=16.293455264;
```

Listing 2.1: An excerpt from a fictitious Hi-C data set. The data is represented in the gTrack file format [12] storing “linked segments” from the hg19-genome. The file has a header of 8 lines, followed by 4 lines of data. Each row contains the start position of a region, followed by its name and a list of all its “neighbors”. Each neighbor has a value associated with it that represents the number of observed interactions between the two regions.

2.1.1 Terminology

There is quite a bit of terminology in the bioinformatics literature. Some of the concepts are derived from other similar fields such as biology, molecular biology, genetics and chemistry while others are more specific to bioinformatics.

A *genome* is the entire collection of all hereditary material for an individual organism. *DNA* encodes this hereditary information, and simplistically DNA can be perceived as a collection of “blueprints” for protein production. *Chromatin* is a combination of DNA and proteins, and one of its functions is to fold the DNA to create a more compact structure that fits within the cell. On the path from DNA to protein *RNA* is produced in a process referred to as *transcription*. The mechanisms involved in this process are complex and are undergoing research. One of the complicating factors is the impact from environmental forces.

In eukaryotic organisms the DNA is organized into a number of *chromosomes*, but their occurrence varies in number and size according to organism. The DNA has the shape of a *double helix*, but for the purpose of certain types of analyses the structure can be simplified to a long series of molecules, where each molecule is one of *adenine*, *cytosine*, *guanine* or *thymine*. These molecules are referred to as *bases*, each with their own one-letter abbreviation: A for adenine, C for cytosine, G for guanine and T for thymine. As a result of the way the double helix is constructed each base in the DNA has a complementary base on the other *strand*, and the two bases together can be referred to as a *base pair*.

Some relevant terminology relates specifically to chromatin 3D data. A chromatin 3D data set is divided into regions where each region, or *bin*, contains a number of base pairs. Between each pair of regions there is a weight representing (at least to some extent) the spatial distance between the two regions. Each of these relationships between two regions can either be within the same chromosome, referred to as *intrachromosomal*, or between two regions from different chromosomes, referred to as *interchromosomal*.

The *resolution* of the data set denotes the number of base pairs in each bin. The data sets mentioned in this thesis have a resolution of 1 *mega base pairs* (Mb) , 500 kilo base pairs (kb), 200 kb or 100 kb.

2.2 Significance Testing

Significance testing is an important tool for scientific investigation, and is relevant to all empirical sciences. The methods involved in significance testing is one of the many contributions from applied mathematics and statistics to bioinformatics. Significance testing aims at assessing the evidence for a claim based on observations. It requires the definition of a *test statistic* appropriate to the domain from which the observations are drawn from. Based on the result of this test statistic the statistical significance can be determined.

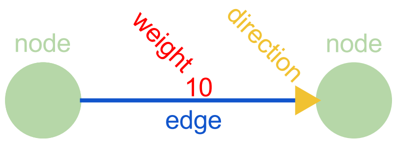
Hypothesis testing is a form of significance testing where a *null hypothesis* is defined and subsequently put to trial. Simplistically a hypothesis test can be said to answer the question of whether a hypothesis can be confirmed or discarded on the basis of data from a scientific study or experiment. To answer this question one must among other things determine the likelihood of getting this (or a more extreme) result given that the null hypothesis is true. The result of a hypothesis test is affected mainly by the number of observations and how "extreme" those observations are. For instance a result close to what would be expected by chance can be statistically significant if the number of observations is high, while a result far from the expected requires fewer observations to be considered significant.

2.3 Graphs

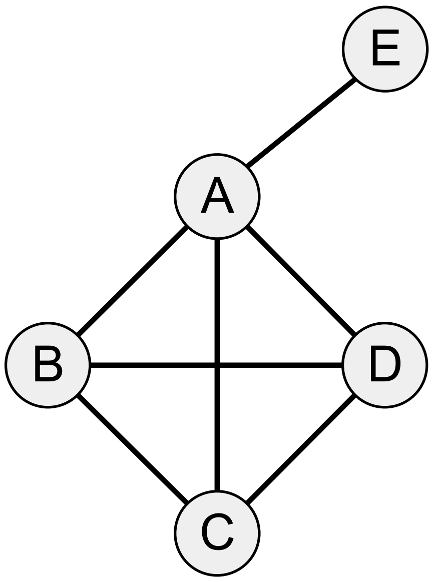
2.3.1 Graph Theory

The well known mathematician Leonhard Euler described graphs in a paper on the Bridges of Königsberg as early as 1735 [10]. Since then graph theory has been developed as a branch of mathematics, and has become relevant in the field of Computer Science.

Graph Theory can be explained informally as the study of graphs. Graphs consist of two key elements: nodes (or vertices) and edges, where nodes are connected by edges. Nodes can typically represent people, places or objects while the edges define the relationship between them. When



(a) Two nodes connected by a directed weighted edge.



(b) An undirected graph with 5 nodes and 7 edges.

presenting graphs visually it is quite common to treat the length of the edges as unrelated to their weights.

Within the graph theory literature there are some disagreements on terminology, but on some of the most fundamental concepts there seems to be a consensus. What follows is a brief overview of some of the definitions within graph theory most relevant to this thesis, as defined by Reinhard Diestel [4].

Graph A collection of nodes and edges, including all attributes that may be associated with the nodes and edges (such as weight or other values).

Node or vertex A node can typically represent a physical or abstract object, such as people or places. Nodes are connected by edges to form a graph. In this thesis vertices are referred to as nodes.

Edge An edge connects to nodes, and has exactly one initial node ("from-node") and one terminal node ("to-node"). Together with nodes, edges are the basic building blocks of graphs. A typical interpretation of an edge is as a relationship between nodes.

Weighted graph If the edges within a graph has weights associated with them, the graph is said to be weighted. Weights typically represent distance between nodes or the cost of traversing an edge.

Directed Graphs (Digraphs) If the edges in a graph has directions associated with them, the graph is said to be directed.

Loop An edge starting and ending on the same node is called a loop. A node containing a loop will thus have an edge to itself.

Path An ordered listing of distinct adjacent nodes. A path can be thought of as an instruction for how to navigate from one node to another node.

Cycle or Circuit A path starting and ending on the same node.

Subgraphs A subgraph is formed by picking nodes and edges from a graph, such that the set of nodes and edges in the subgraph is a subset of all the nodes and edges in the original graph.

Multigraph A multigraph is a graph where loops and multiple edges between nodes is allowed.

Node Degree The number of edges connected to a node.

Complete graph A graph where all nodes have edges to all the other nodes.

Adjacency Two nodes are adjacent if there is an edge between them.

Density and Sparsity Density and sparsity is related to the number of edges in a graph relative to its number of nodes. A graph with only a few edges and many nodes is *sparse* while a complete (or almost complete) graph is *dense*. There is no general consensus as to where the distinction between a sparse and a dense graph goes.

Graph theory is a versatile tool for modeling a wide range of phenomena involving relationships. There are multiple variations to what a graph can legally consist of, such as loops, directions and edge weights.

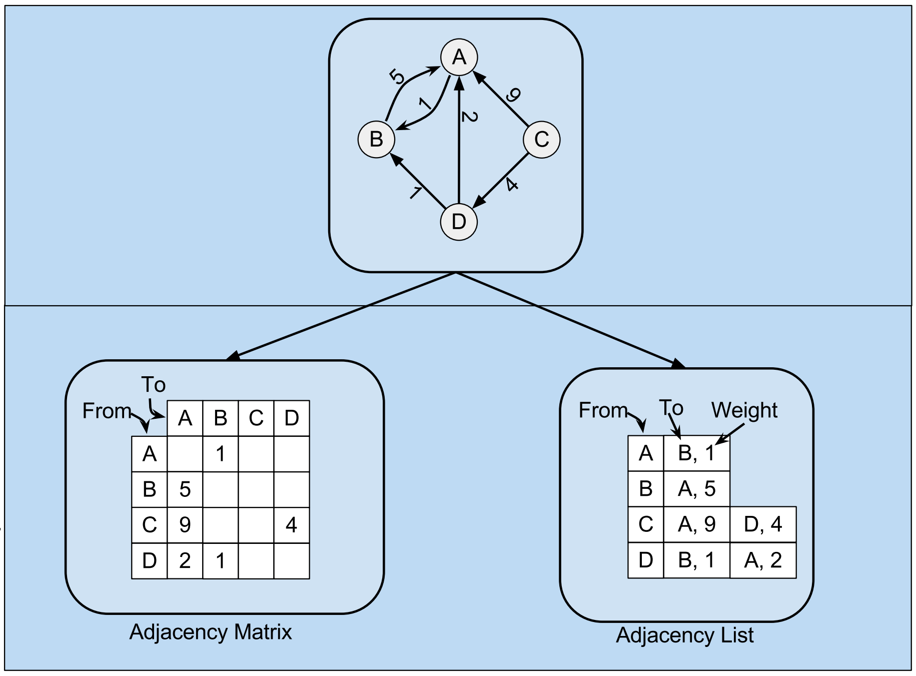
2.3.2 Graph Representation and Implementation

There are several ways to represent graphs in a computer system. Two of the more popular representations are the *adjacency list representation* and the *adjacency matrix representation*. Multiple Computer Science text books dealing with algorithms refer to these two representations [3] [1], and most of the other possible ways of representing graphs can somehow be seen as a variant of one these.

At the implementation level there are a multitude of ways to implement both adjacency lists and adjacency matrices, but their memory characteristics are mostly the same. Their memory requirement and their performance is likely similar for all practical purposes. A common trade off between different implementations is between memory consumption and running time for different operations. This is the topic of discussion in chapter 7.

Adjacency List

One of the more popular representations is the *adjacency list representation*. As implied by its name this representation uses a list of adjacent nodes to construct a graph. For a sparse graph this representation is potentially more compact than the adjacency matrix.



	A	B	C	D
A		1		
B	5			
C	9			4
D	2	1		

A	B, 1	
B	A, 5	
C	A, 9	D, 4
D	B, 1	A, 2

Figure 2.1: A comparison of the two most common weighted graph representations: the adjacency list and the adjacency matrix. For a sparse graph like this the adjacency list is more compact.

The adjacency list representation can be extended to include weighted edges in a number of ways. For instance each of the elements in the adjacency list can store two values (instead of one): a pointer to the neighboring node *and* a weight.

Adjacency Matrix

Another popular representation is the *adjacency matrix representation*. Its underlying data structure is a two-dimensional matrix of size n^2 where n is the number of nodes in the graph. For weighted graphs each cell in the matrix contains the weight of the edge between the pair of nodes from the row and column. For unweighted graphs a boolean value indicating if the edge is present or not is sufficient. The size of the matrix is always n^2, and as a result of this the adjacency matrix is not very space efficient for sparse graphs.

2.4 Performance Analysis

In the process of designing algorithms and implementing them as computer programs performance analysis is crucial. Performance analysis can be divided into theoretical analysis of algorithms and practical analysis of implementations. Both theoretical and practical analysis are useful when developing computer software.

2.4.1 Terminology

There are especially two terms that need to be defined in this context, because their meaning can be ambiguous: *Performance* is meant to describe the work performed divided by the time it takes to perform. *Efficiency* on the other hand is performance divided by resource usage. Usually, the resource in question is main memory. Thus, a computer program with high performance is a program that can achieve a lot in a short time, but might require a lot of memory. How to measure a programs achievement, and exactly what "short time" means, depends on the context. An efficient program does not only achieve a lot in a short time, but does so with reasonable resource usage. Again, what constitutes "reasonable resource usage" is context dependent.

When discussing an algorithm or an implementation *overhead* is often used in a negative sense to denote the resources spent on operations that are not directly a part of achieving the goal of the algorithm. What constitutes overhead thus depends on the context and what the goal of the algorithm is. For instance, when considering the storage of data in a data structure the overhead can refer to the space occupied by *everything else* than the actual information stored.

2.4.2 Algorithm Analysis

Algorithm analysis denotes the theoretical ways to analyze and reason about the performance of algorithms. Here the *algorithms* rather than an actual *implementation* is the subject of analysis. Algorithm analysis encompasses a number of techniques, one of which is *computational complexity class* determination. A complexity class is a description of how the resource usage of an algorithm changes as a function of its input. The resource usage can either be related to *time* (running time) or *space* (memory usage).

A typical use of algorithm analysis is to formalize the effect the size of the input has on the running time of the algorithm.

On the basis of algorithm analysis some algorithms can be discarded for practical purposes. If the size of the input for a given application of an algorithm is estimated to be incompatible with the algorithms complexity class, the algorithm can be deemed unfit as a solution to that problem. The complexity class of an algorithm does not estimate the actual running time of an implementation, for that we must turn to practical methods such as benchmarking.

2.4.3 Practical Performance Analysis: Profiling and Benchmarking

Practical methods for performance analysis includes various forms of *benchmarking* and *profiling*. These methods are tied to concrete implementations rather than abstract algorithms.

While theoretically analyzing algorithms is a necessity when designing algorithms, it is not enough to ensure that the performance of an implementation of that algorithm is satisfying. Many algorithms within bioinformatics scales poorly, but are useful nevertheless. Doing practical experiments such as benchmarking is a useful tool for determining if a given implementation of an algorithm is applicable for a problem or not. A given algorithms complexity class, and thus its ability to scale, is of lesser importance if the size of its input is estimated to be "small enough" for computations to be performed within reasonable time.

Measuring memory usage and running time is an inherently imprecise process, but the reliability can be improved by increasing the number of samples and controlling for variables such as hardware and environment.

2.5 HyperBrowser

The Genomic HyperBrowser [25], or HyperBrowser for short, is a software system providing "statistical methodology and computing power to handle a variety of biological inquires on genomic datasets" [27]. It is available through a web interface based on the Galaxy framework [26].

The Galaxy framework was developed to address some of the challenges in life science research as it is becoming increasingly reliant on computational methods [8]. It allows scientists to perform computations that are reproducible and provides easier access to the results. It also limits the need for informatics expertise to analyze the computational methods used to arrive at a given result, and instead provide information suited for scientists from other disciplines such as biologists. To achieve this, Galaxy stores extensive information about each analysis being performed. This includes the datasets used as input, a history of actions that details the computations performed on the input (also known as a workflow), and any parameters and configurations. To limit the need for informatics expertise and programming experience when using Galaxy to perform analyses, Galaxy encourages methods to be constructed as building blocks that can be chained together to create new methods. This requires each method to perform a limited task in a generalized fashion, so that it can be used in multiple contexts.

HyperBrowser was developed as a tool for biologists and bioinformaticians in need of computing power to analyze genomic data, and is based on the Galaxy framework. In addition to the capabilities HyperBrowser inherits from Galaxy, such as reproducibility of results, it provides computing power and implementations of generic methods for analyzing genomic data.

2.5.1 Architecture and Implementation Details

The design of HyperBrowser is driven by the need for high performance computing performed in a high-level programming language. To make up for the relatively slow running time of the high-level language (Python), a library (NumPy) with implementations in low-level languages (C and Fortran) is used extensively. The architecture of the system reflects the design goals by implementing mechanism to cache and reuse results from computations, as well as techniques for dividing problems into smaller parts and thus requiring less memory.

Tracks are Stored in `ndarrays` and `memmaps`

At the conceptual level data is represented as *Tracks* in HyperBrowser. The implementation of a Track relies heavily on `ndarray`s to achieve reasonable performance. For persistence, `memmap`s are used to write the `ndarray`s to disk.

New data sets are installed as tracks by providing a file in one of the supported file formats, such as *gTrack*, *wig* or *bed*. The file is then parsed by an internal parser constructing the `ndarray`s and written do disk. In addition to the contents of the track certain meta data is collected and stored by the parser.

Modular Architecture to Encourage Reuse of Components

Reuse of the methods implemented in HyperBrowser is made possible by adhering to the principle that each method should be written as generic as possible, and only perform *one* task. As the number of implemented methods in HyperBrowser grows, the amount of work required to implement new methods can potentially be reduced by combining the already implemented methods in new ways.

Caching of Intermediate Computations

A recurring pattern in software in general is that computations that are essentially equal, meaning that they have the same input and give the same result, are performed more than once. When each computation is expensive, as is typically the case for analysis of large biological datasets, reducing the number of redundant computations can increase the efficiency significantly. To avoid performing computations more than once, any calculation that requires the results from other intermediate computations must define its dependencies on other computations. Conceptually HyperBrowser creates a dependency graph for the computations, and reuses the results from the computations so that each computation with identical input is performed exactly once.

Dividing Computations Using MapReduce

MapReduce is a technique for tackling computations on large data sets. It requires computations to be expressed as two functions: one to perform the computation on a partition of the data, and one to combine the results in an appropriate way. MapReduce has a number of benefits: Computations can be performed in parallel to reduce the running time; Memory usage can be

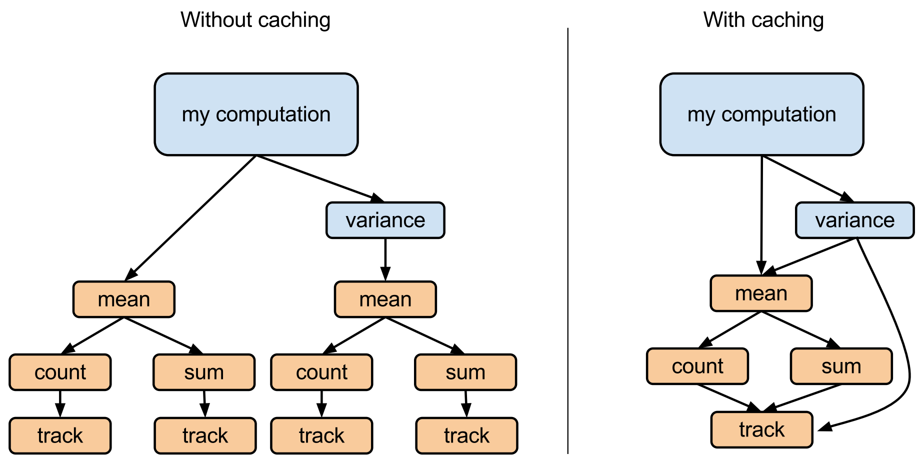


Figure 2.2: Showing the difference in number of computations performed with and without caching of results. There are a number of redundant computations performed when caching is disabled, such as calculating *mean* and its children twice.

lowered because only a partition of the data set needs to be kept in memory at one time; Reduce running time by limiting swapping of data between memory and disk as a result of using less memory;

HyperBrowser implements a variant of MapReduce, foremost to limit the memory usage. In order to use MapReduce when developing a statistic for HyperBrowser a *splittable* method must be defined, as well as a reduce method for combining the results of each computation. Some problems are not dividable and can be defined as such in HyperBrowser by implementing an *unsplittable* method.

2.6 NumPy

NumPy is a Python library for efficient scientific computing. At the core of NumPy is its `ndarray`: a multidimensional array of a specified data type. NumPy includes a lot more than the functionality surrounding multidimensional arrays, and is an essential tool for working with numerical data in Python.

As described in the Online NumPy Documentation [32] NumPy achieves its high performance by vectorizing operations, limiting the amount of data to be copied in memory and reduces the number of operations by implementing certain functions in C rather than Python.

The Standard Python Library provides an array class much like NumPy, but this only supports one dimensional data. In addition there are a lot of methods for working with arrays implemented in NumPy that is not a part of the standard array class.

Chapter 3

Case Study: Assessing Spatial Co-localization of Regions

One of the first statistics that was implemented in HyperBrowser to analyze chromatin 3D data was a hypothesis test for assessing the spatial co-localization of regions. Most of the results and observations presented in this thesis are somehow related to the development of this statistic. The search for efficient data structures and algorithms for working with chromatin 3D data began after experiencing severe performance issues with the initial implementation in HyperBrowser. This chapter is an account of how the development of this hypothesis test progressed and the experiences gained along the way.

The focus was on implementing the statistic in an efficient manner given the way HyperBrowser currently stores graphs. Changing the way HyperBrowser stores graphs is outside of the current scope, but several weaknesses and limitations were found when this hypothesis test was developed. Chapter 5 addresses some of these aspects and provides suggestions for improvements.

3.1 Background

Before discussing the details of the implementation of this hypothesis test, some background material is necessary.

3.1.1 The Statistic

The statistic that was developed provides an answer to questions of whether certain regions, referred to as *query regions*, within the genome are closer to each other than what would be expected. A typical use case would be to pick a number of genes known to be somehow functionally related as the query region and determine their spatial proximity.

The calculation is performed by computing the average of the weights within the query region and comparing that to what would be expected by chance. To determine what would be expected by chance a number of Monte Carlo simulations are performed with randomized query regions as input. On the basis of the results from the Monte Carlo simulations a distribution is created and the results from the "real computation" (from using the actual query region) is compared to this distribution to assess its statistical significance. The way in which the query regions are randomized throughout the Monte Carlo simulations are carefully designed and is one of the topics in an article by Paulsen *et al.* [21].

There are several ways to implement this calculation in HyperBrowser. An important characteristic of this calculation is that it requires a large number of read operations from random positions in the data structures responsible for storing the chromatin 3D data. Further, the number of times that an average is calculated can become large as it depends on the number of Monte Carlo simulations performed. The implementation of this statistic went through several revisions, where each change was driven by the need for lower running times or lower memory use.

3.1.2 How Graphs are Stored in HyperBrowser

Graphs are stored much in the same way other sorts of genomic data is stored in HyperBrowser. At the lowest level graphs are being stored in NumPy `ndarrays`, and moved between disk and main memory using *memmaps* [19] provided by the NumPy library. Using memmaps is a convenient way of serializing `ndarrays` for persistence, and supports reading segments into memory rather than the entire array. There are three separate `ndarrays` used to represent a graph:

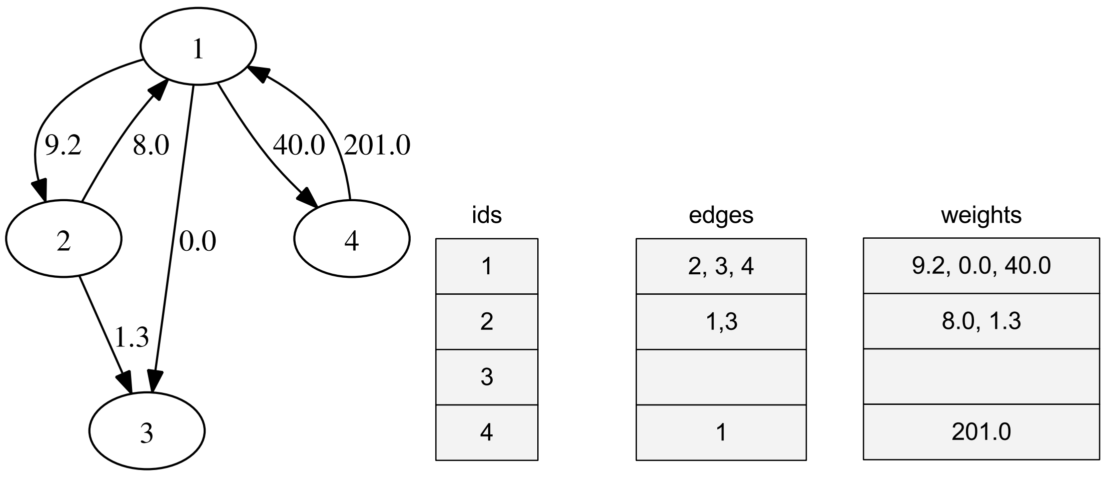
ids A one-dimensional array containing the names of all the nodes in the graph as strings.

edges A two-dimensional array containing lists of neighbors for each node.

weights A two-dimensional array containing a list of weights for each edge.

There are some important aspects of this representation that affects the performance, particularly its great flexibility. There is no guarantee about the ordering of the nodes in the *id* and *edges* array. Also, the graph might include loops (an edge from a node to itself) and it might be complete (all nodes have edges to all the other nodes), but there are no guarantees for this either. This uncertainty about the ordering and structure of the data makes the data structures flexible, but has a major impact on the performance of the operations that relies on it. A further discussion of this and suggestions for how to improve the situation is provided in chapter 5 on page 39.

A genome consists of multiple chromosomes, and HyperBrowser stores each chromosome separately. Internally a chromosome is referred to as a *Genome Region*. This is a more general concept than a chromosome, because it can refer to any region within a genome. However, for chromatin 3D data a genome region is synonymous with a chromosome.



ids	edges	weights
1	2, 3, 4	9.2, 0.0, 40.0
2	1,3	8.0, 1.3
3		
4	1	201.0

Figure 3.1: A graph of 4 nodes, and how it would be represented as `ndarrays` in HyperBrowser.

Accessing the Graphs

An important part of any data structure is the interface it provides for programmers to utilize in their programs.

An apparent way of accessing the graphs is to access the `ndarrays` directly. This is efficient and requires no additional layer of abstraction. However, the complex organization of the data makes it challenging to write robust code for operating on graphs, and a lot of the logic for common operations will typically be duplicated.

To provide a friendlier interface to the graphs a layer of abstraction that provided the concepts of *edges* and *nodes* was implemented. Edges and nodes could be accessed by calling iterators, and the appropriate objects would be returned. This required little effort to implement, and followed a typical object-oriented pattern. The logic in the computations can be greatly simplified by using this interface instead of accessing the `ndarrays` directly, but the performance is poor.

In developing the hypothesis test the limitations of these two interfaces became obvious. Out of necessity a third option for accessing the graph was developed: an adjacency matrix with weights represented as a NumPy `ndarray`. Throughout this thesis this adjacency matrix is referred to as a *graph matrix*. The rationale for creating this additional data structure is discussed in section 3.2.2.

3.2 Implementations

What follows are descriptions of four different implementations of the statistic, presented in chronological order. The implementations are evaluated and discussed further in section 3.3.

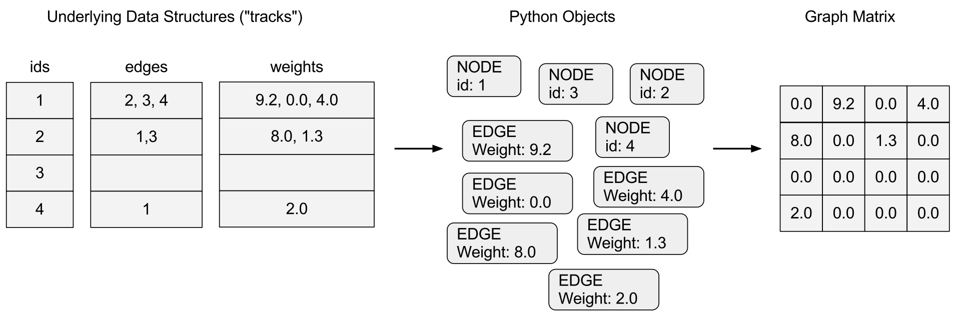


Figure 3.2: Implementation #2 creates a matrix representation of the entire graph (a graph matrix) by creating Python objects for every node and edge from the underlying data structure.

3.2.1 Implementation #1: Creating Subgraphs

The aim of the statistic was to create a subgraph consisting of a given set of nodes and compute the average of all the edge weights within this subgraph. One way of doing this is to create a copy of the graph containing all the nodes and filter out the ones that are outside the desired subgraph. This operation can be quite efficient as a result of some implementation details in the way HyperBrowser stores graphs. The graph does not have to be *physically* copied, new references to the same physical data can be made. Before calculating the average, a two-dimensional NumPy `ndarray` representing the subgraph is created and the edge weights are inserted.

The running time of creating a subgraph this way is $O(n^2)$ where n is the number of nodes in the subgraph. This is a result of the way HyperBrowser represents graphs: as seen in figure 3.1 each node has an array of its neighboring nodes, and for each of those neighbors a decision has to be made to include this node in the subgraph or not.

3.2.2 Implementation #2: Creating the Graph Matrix Using Iterators

To perform the calculations involved in assessing the co-localization of regions, it seemed sensible to create a *graph matrix* as an intermediate representation of the graph. As the elements of the matrix would be accessed often and in a random order, a matrix representation implemented using NumPy `ndarray` had the desired performance characteristics: the small memory footprint meant that the entire matrix could be kept in memory, and the efficient implementation of a *mean-function* included in the library.

This graph matrix had to be constructed from the underlying data structures in HyperBrowser. The data structures responsible for handling chromatin 3D data in HyperBrowser is not suited for direct access from developers (for a brief explanation of how chromatin 3D data is stored in HyperBrowser see chapter 3.1.2 on page 18. To create an easy to use interface for graph data a layer of abstraction was added on top of the underlying data structures. This layer relied heavily on *iterators* and *objects*: Nodes and edges were represented as objects and each had an iterator for accessing its related entities.

The initial implementation of the method for creating the graph matrix used the iterators from the previously described "convenience layer" with objects and iterators. Thus a lot of objects were being created and lots of functions were being called in the process of creating the graph matrix.

Creating a Graph Matrix

There are several good reasons why creating a Graph Matrix is a good idea. One of the reasons for creating a graph matrix is the possibility of performing normalizations or other operations that require data from the entire graph. Also, this graph matrix representation can act as an interface for programmers that need access to graph data. This interface combines two important features. It is easy to use because a NumPy `ndarray` is familiar to many programmers experienced with scientific computing. It is also efficient, because the NumPy operations are fast and the matrix is stored in a memory efficient manner. A graph matrix can therefore make it easier for future developers to create their own statistics without worrying about the underlying data structure that represents graphs in HyperBrowser. This makes the efficiency of creating the graph matrix important for the efficiency of all other computations involving chromatin 3D data.

To encapsulate all the data associated with the graph matrix, a `GraphMatrix` class was defined. It contains the `ndarray` with edge weights between every pair of node, as well as a mapping from node id to row number and column number. The row number and the column number for a given node can be different, because this allows for an optimization to be made when creating the graph matrix. The optimization involves keeping the ordering of nodes that is present in the `ndarray`, instead of rearranging the data when creating a `GraphMatrix`.

Reusing Results Across Monte Carlo Simulations

The time it takes to perform a hypothesis test can easily be dominated by the number of Monte Carlo simulations performed. A naive implementation might perform the exact same computations for each Monte Carlo simulation. The efficiency of such computations can be improved by defining independent Stat-classes with limited responsibilities for performing parts of the computation. This way the HyperBrowser system is able to identify the parts of the computation that needs to be recomputed for each Monte Carlo simulation, and which parts that can be reused. A more thorough discussion of how using Monte Carlo simulations affects the performance is the topic of chapter 6.

For the statistic described in this chapter a Stat-class was defined with the responsibility of creating the graph matrix. As a result the graph matrix was reused for all Monte Carlo simulations. The time it takes to create a graph matrix is substantial, and thus reducing the number of times this computations is being performed significantly reduces the running time.

3.2.3 Implementation #3: Creating the Graph Matrix More Efficiently

As the data source was already an `ndarray` and the target was an `ndarray`, although with a different organization of its elements, the costly operation of temporarily creating objects to represent every single element seemed unnecessary. A significant increase in running time was achieved by creating the target `ndarray` directly from the source without creating objects in between. This was implemented by accessing the underlying data structures directly and copying the contents of it into a newly created `ndarray`. The order and position of the elements did not correspond between the underlying data structures and the graph matrix. This required reordering of the elements and the lack of assumptions that could be made about the order of the elements in the underlying data structure limited the performance.

Optimizing by Making Assumptions

By assuming that the graphs in the underlying data structures were complete, meaning that each node has edges to every other node, How the data in the underlying data structures are laid out is decided by the component that is responsible for installing new tracks in HyperBrowser. There is no guarantee for the assumptions that enable this optimization to always be true, but currently the HyperBrowser system complies with it. Several other optimizations that could be made by following this approach are discussed in chapter 5 on page 39, as well as suggestions for how this technique could be made more robust and not rely on assumptions that might not be true.

3.2.4 Implementation #4: Splitting The Computation

As a result of the way HyperBrowser is designed a computation can be performed for each chromosome separately, called *local analysis*, as well as for all chromosomes, called *global analysis*. Implementation #2 and #3 creates a graph matrix for each chromosome during the local analysis. Then, in the global analysis, a bigger graph matrix is created consisting of all the chromosomes. As a result the graph matrices constructed in the local analysis is also created as a part of the global analysis. This leaves room for improvement, as the graphs matrices from the local analysis could be reused.

Implementation #4 takes advantage of the MapReduce pattern implemented in HyperBrowser. The graph matrices constructed in the local analyses are being reused by defining a method capable of combining the matrices into the graph matrix needed for the global analysis.

3.3 Results

To summarize the results the running times of the different implementations are presented in figure 3.3.

3.3.1 Implementation #1: Scales Poorly

Implementation #1 creates subgraphs and calculates the mean weight of all the edges within that subgraph. As can be observed from figure 3.3 the running time of this implementation is acceptable for "smaller computations" where the number of Monte Carlo simulations and the size of the query region is small. However, the running time grows at a high rate as the number of Monte Carlo simulations and/or the size of the query region increases.

The running time of this implementation relates linearly to the number of Monte Carlo simulations: an increase in Monte Carlo simulations by a factor of ten results in a corresponding increase in running time. Increasing the resolution of the graph from 1Mb to 200kb represents an increase in the number of edges by a factor of 25, while the difference in running time increases by a factor much lower than this. The biggest problem with this implementation in terms of scalability is the relationship between running time and query region size. When interpreting the results in figure 3.3 it is important to keep in mind that the query region size is given as the number of *nodes* in the subgraph, while the size of the computation is related to the number of *edges* in that subgraph. The number of edges is approximately given as the square of the number of nodes, as the subgraphs are complete graphs. Increasing the size of the subgraph from 10 nodes to 100 nodes should thus be considered an increase from 100 edges to 10,000 edges. By analyzing the algorithm used in this implementation the running time can be estimated to grow at a quadratic rate related to the number of edges. Although the data is limited, there is support for this in the data.

The combination of a high growth rate and expensive operations makes this implementation practically unusable. But the idea of creating the subgraphs directly might not be a bad idea if the execution is better. While this implementations could have been optimized a great deal, going further with it was not interesting because the arguments for trying other approaches were so strong.

3.3.2 Implementation #2: Slow Graph Matrix Creation

Implementation #2 was developed with two goals in mind: to improve the scalability of the implementation and to allow for efficient global operations (such as normalizing the weights of the graph). This was achieved by creating a graph matrix representing the entire graph as a matrix. Creating the graph matrix can be seen as an initial process taking place before any other operation is performed. The running time of this operation is determined by the size of the graph.

As shown in figure 3.3 the running time of this implementation is high compared to implementation #1, but more stable. Implementation #2

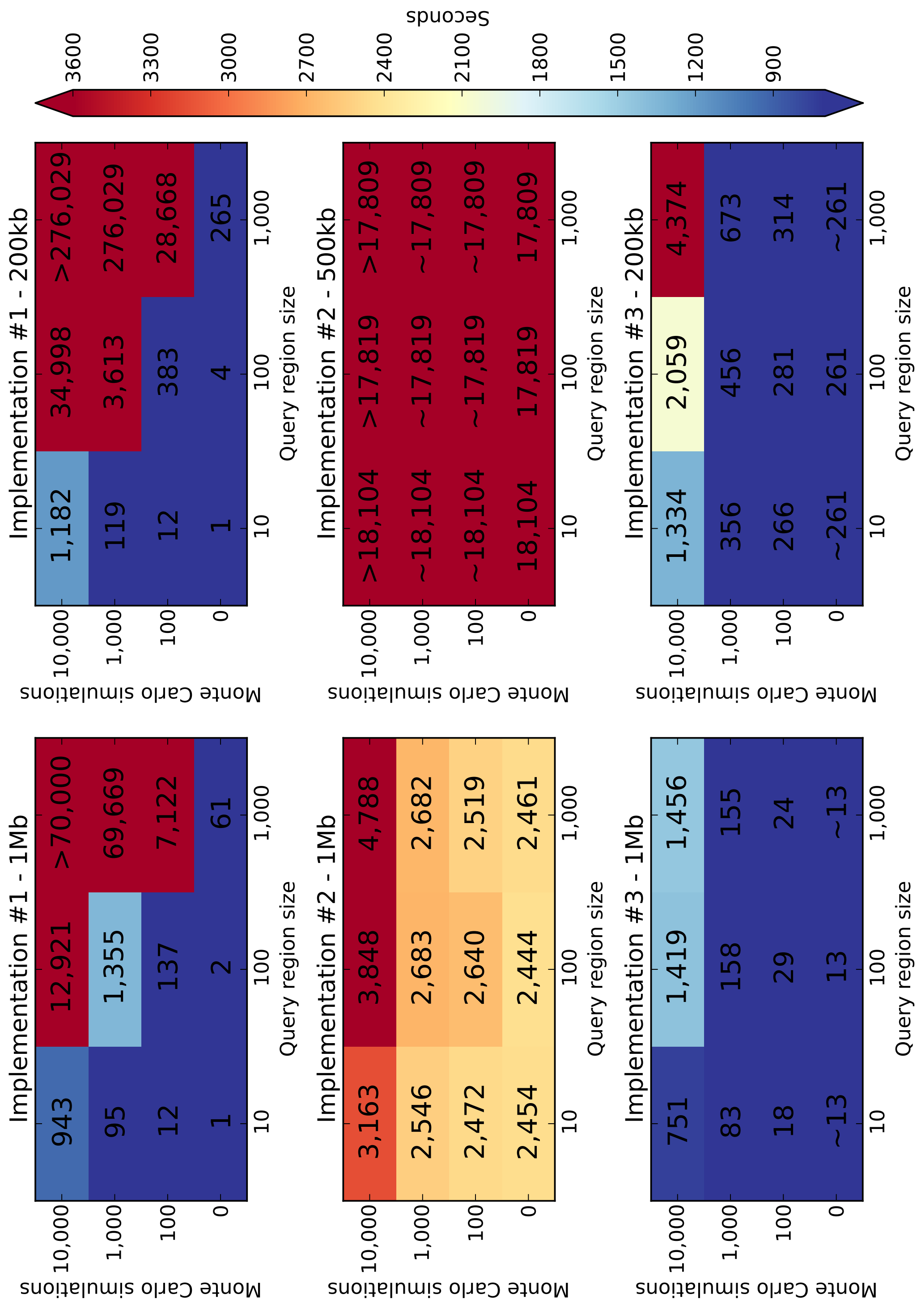


Figure 3.3: Color coded matrices showing the running time (in seconds) for each of the three implementations as the number of Monte Carlo simulations and the size of the query region varies.

outperforms implementation #1 when the resolution of the graph is low and the number of Monte Carlo simulations and the size of the query region is high. This can be explained by considering the high one-time-cost associated with creating a graph matrix. When this graph matrix is created the rest of the computation can be quite fast.

By profiling the computation in particular three operations emerged as potential bottlenecks:

- Creating a two-dimensional array from the underlying graph storage mechanism in HyperBrowser.
- Creating a sub matrix from a super matrix and a set of indices to be included.
- Finding the index in a matrix for a given position within a corresponding genome.

Figure 3.4 illustrates how different parameters affect the running time of implementation #2 and #3. From this figure it becomes apparent that one operation in particular (creating the graph matrix) is responsible for the high running time of implementation #2 in most cases.

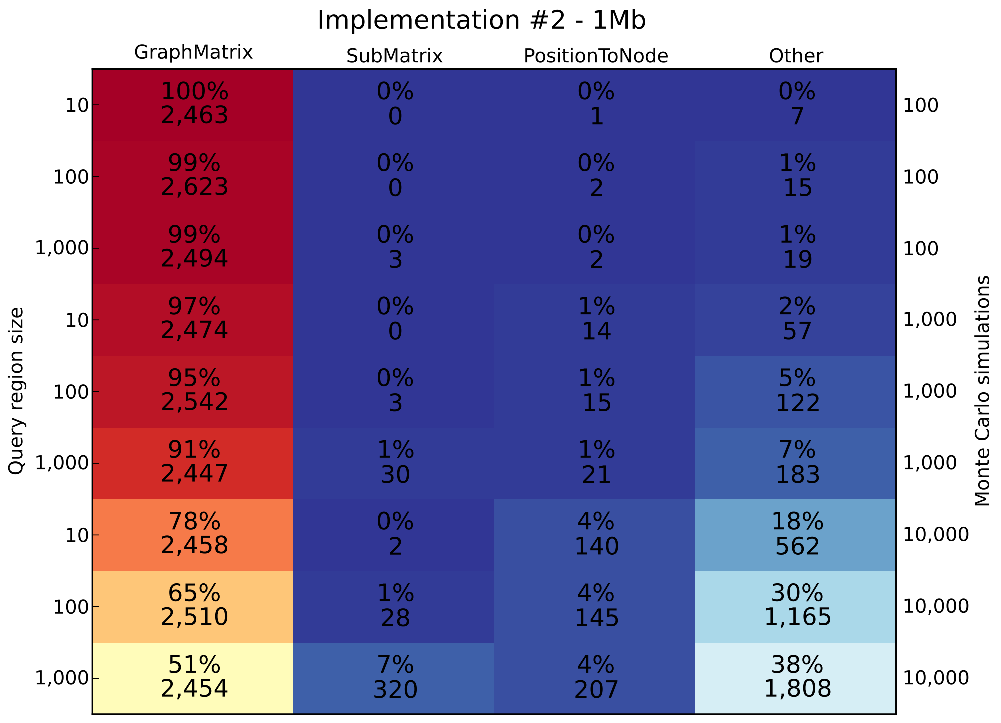
The way the graph matrix is created explains this high running time. From a bird's eye perspective, the process of creating the graph matrix involved going from the underlying NumPy `ndarray`s via Python objects to another `ndarray`. This implementation creates one object for each edge and node in the graph, and it is well known that creating and managing Python objects is expensive. As a result this implementation is practically unusable.

3.3.3 Implementation #3: Faster Graph Matrix Creation

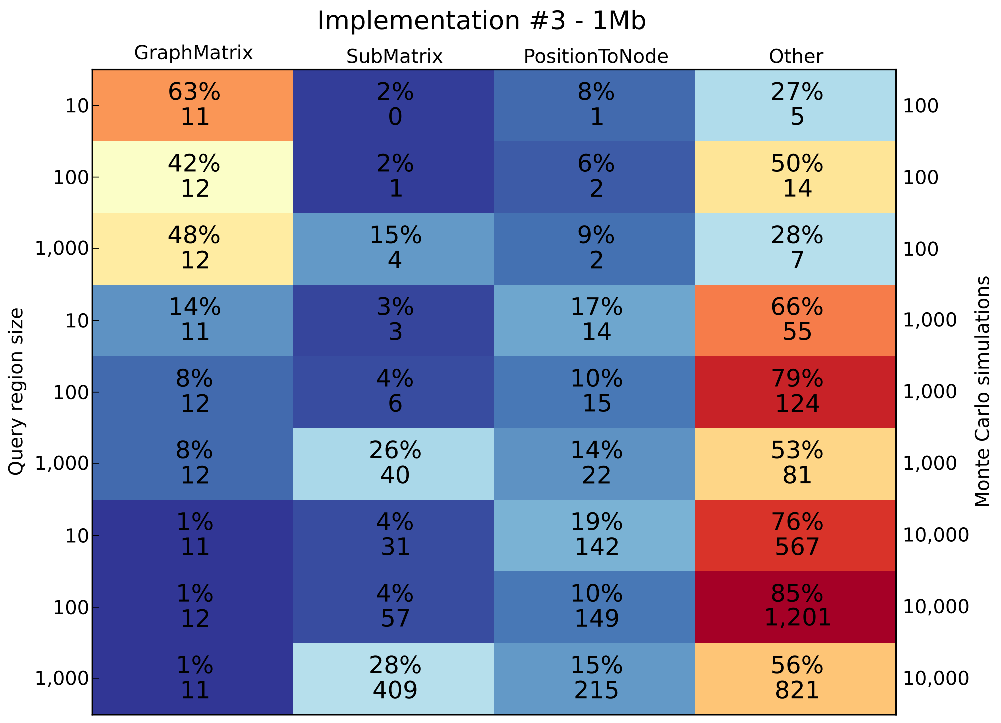
Implementation #3 is similar to implementation #2, but with one major improvement. The graph matrix creation is implemented entirely by using NumPy operations, making it significantly faster. Except for this improvement the characteristics and scalability of this implementation are identical to implementation #2: the time it takes to create a graph matrix is still linearly related to the number of edges in the graph and the graph matrix is reused for all Monte Carlo simulations. The size of the query region has an insignificant effect on the running time, as the it is achieved through highly efficient NumPy operations.

As can be seen in figure 3.4 the time usage is distributed more evenly among the operations in implementation #3. This is a result of lowering the running time for the graph matrix creation. By comparing the absolute running time (in seconds) between implementation #2 and #3 the difference for all the other operations can be determined as relatively small.

By making the graph matrix creation faster, the running time of the other operations in the computation has become more significant. In particular, the "overhead" (referred to as "other" in figure 3.4) is making up a major part of the running time as the number of Monte Carlo simulations



Query region size	GraphMatrix	SubMatrix	PositionToNode	Other	Monte Carlo simulations
10	100% 2,463	0% 0	0% 1	0% 7	100
100	99% 2,623	0% 0	0% 2	1% 15	100
1,000	99% 2,494	0% 3	0% 2	1% 19	100
10	97% 2,474	0% 0	1% 14	2% 57	1,000
100	95% 2,542	0% 3	1% 15	5% 122	1,000
1,000	91% 2,447	1% 30	1% 21	7% 183	1,000
10	78% 2,458	0% 2	4% 140	18% 562	10,000
100	65% 2,510	1% 28	4% 145	30% 1,165	10,000
1,000	51% 2,454	7% 320	4% 207	38% 1,808	10,000



Query region size	GraphMatrix	SubMatrix	PositionToNode	Other	Monte Carlo simulations
10	63% 11	2% 0	8% 1	27% 5	100
100	42% 12	2% 1	6% 2	50% 14	100
1,000	48% 12	15% 4	9% 2	28% 7	100
10	14% 11	3% 3	17% 14	66% 55	1,000
100	8% 12	4% 6	10% 15	79% 124	1,000
1,000	8% 12	26% 40	14% 22	53% 81	1,000
10	1% 11	4% 31	19% 142	76% 567	10,000
100	1% 12	4% 57	10% 149	85% 1,201	10,000
1,000	1% 11	28% 409	15% 215	56% 821	10,000

Figure 3.4: The relative and absolute running time (in seconds) of each operation involved in computing the statistic on a 1 Mb data set. The number of Monte Carlo simulations and the size of the query region varies, and the total cost of the computation increases towards the bottom of each table.

are increasing. This overhead includes the randomization of data that occurs for each Monte Carlo simulation, and this can explain a part of the overhead and why it becomes so significant when the number of Monte Carlo simulations increases.

Comparing implementation #3 to the two other implementations from figure 3.3 shows how this implementation outperforms the two others in all situations. The experience from using this implementation to compute the statistic described in section 3.1.1 proved it useful for practical purposes, with an acceptable running time.

3.3.4 Implementation #4: To be Continued

Faster is always better when it comes to running times, and implementation #3 left some room for further improvement: when the statistic was run on more than one chromosome, there was some redundancy in the creation of the graph matrix.

Surprisingly this implementation performed so poorly that it was practically unusable. Both the running time and the memory usage was exceptionally high, and further investigations pointed towards an underlying issue in HyperBrowser that can induce a memory leak. As a result no final conclusion on the efficiency of this implementation can be drawn. However, this implementation should, at least in theory, be capable of achieving lower running times and possibly lower memory usage compared to the other implementations.

Chapter 4

Array programming can result in high performance, but constrains the data and usage patterns

NumPy plays an essential part in many scientific applications, including HyperBrowser. Its importance in improving the performance of the hypothesis test described in chapter 3 makes it relevant for further discussion.

The difference in running time and memory usage can vary greatly between a pure Python implementation, limited to using the Standard Library, and an implementation that uses NumPy. This can be true even for seemingly equivalent implementations of the same algorithm. This is an example of how the complexity class of an algorithm is only partly relevant to the performance of an implementation.

In order to use NumPy efficiently there are some constraints. In this chapter NumPy is being compared with Python and its Standard Library. The focus is mainly on storage and computation on chromatin 3D data, but some observations are of a more general nature.

The findings presented in this chapter are a combination of general observations regarding NumPy and Python, and experiences from developing one of the first statistics working on chromatin 3D data in HyperBrowser as described in chapter 3.

4.1 Data Structures for Storing Graphs

4.1.1 Implementing Graphs with Linked Python Objects Suffers from Poor Performance

Besides the matrix representation dominating this thesis, graphs can also be represented using various forms of object hierarchies. In many Computer Science textbooks graphs are introduced using some sort of object hierarchy, with objects representing nodes and edges. The reason for using a different approach to graph data within the HyperBrowser is partly an historical one: From an early point in the development of HyperBrowser NumPy `ndarray`s were being used as a data structure for most of the data sets. Representing graphs was not a part of the original design, but was added later when this kind of data became available through various chromatine conformation capture experiments. As all the other data sets were represented using `ndarray`s already, the reason of choosing this solution for representing graph data could be suspected of being driven by the desire for easy integration rather than an analysis of performance. Investigating whether a more efficient implementation to support graph data could be made using other data structures altogether therefore seems worthwhile.

In Python, where "everything is an object", the distinction between this type of implementation and one using nested Python Lists to form matrices is not clear. The vague definition used here is meant to include all graph implementations where objects and their relation, rather than a matrix, constitutes the graph.

There are several ways to implement undirected weighted graphs with objects as the basic building blocks.

Representing chromatin 3D data requires a weighted graph.

When the data sets are small and traversal is important, working with this sort of representation can be both efficient and intuitive. Implementing graphs using object hierarchies falls short when the data sets grow. At least in Python there is a significant overhead associated with creating and managing objects, and this takes a toll on both running times and memory usage.

From a software developers perspective working with graphs represented in an object-oriented way differs greatly from graphs represented by matrices. While matrix representation allows for efficient operations on the values stored in the matrix, the object representation favors graph traversal. Which of the two representations is more intuitive or "natural" is dependent on the sort of operations most likely to be performed on the graphs.

The size of graphs for dealing with chromatin 3D data renders any object-oriented implementation in Python practically unusable. To illustrate the problem of an object-oriented implementation consider this simple experiment:

The goal of this experiment is to estimate the minimum size of an object-oriented graph implementation in Python. There are many ways to implement graphs in Python and analyzing all of them would be infeasible. Estimating the minimal memory usage for *any* of the possible implementations is a bit easier, because only one implementation must be analyzed.

The challenge lies in constructing this "minimal graph implementation" in a convincing way, so that no other implementation could possibly be smaller. One approach is to create an implementation so small that it is not even a working graph implementation. It consists of an object representing edges, and this object has only one attribute namely its weight. Many such

Edge-objects were created to gain a more precise measurement for the size of each object. In an attempt to subtract the size of the overhead from the list in which the Edge-objects are kept, another list of the same size with only float-objects was created and used as a “baseline”.

```python
from random import random

class Edge:
    def __init__(self):
        self.weight = random()

@profile
def edges():
    edges = [Edge() for i in xrange(10**6)]

if __name__ == '__main__':
    edges()
```

Listing 4.1: The source code where the Edge-class is defined and a million instances of it is created.

```python
from random import random

@profile
def baseline():
    baseline = [random() for i in xrange(10**6)]

if __name__ == '__main__':
    baseline()
```

Listing 4.2: The source code for the script used to create a baseline.

As shown in listing 4.3 Memory Profiler measures the baseline list consisting of one float-object to be about 62 MB. A list consisting of the same number of Edge-objects are measured to be roughly 408 MB. A rough estimate for the size of a million Edge-objects, without the overhead stemming from the list they are contained within can thus be found by subtracting the baseline from the size of the `edge` list: $407.72 - 62.18 = 345.54$ This gives a size per Edge-object of $345.54/10^6 = 0.00034554$ MB or $3.62325e8/10^6 \approx 362$ bytes.

```
>> python -m memory_profiler baseline.py
Increment   Line Contents
=========================
            @profile
  0.00 MB   def baseline():
 62.18 MB       baseline = [random() for i in xrange(10**6)]


>> python -m memory_profiler edges.py
Increment   Line Contents
=========================
            @profile
  0.00 MB   def edges():
407.72 MB       edges = [Edge() for i in xrange(10**6)]
```

Listing 4.3: An excerpt from running Memory Profiler on the scripts listed in figure 4.2 and 4.1 respectively

The results from this simple experiment can be used to reason about the memory requirements involved when working with object-oriented graph implementations in Python. Underestimating the size like this allows the results to be used in argumentation for why object-oriented graph representations in Python are unsuited for data sets as large as the ones used to represent Chromatin Interaction Data.

From this experiment the minimal memory size per edge of an object-oriented implementation in Python was estimated to be more than 300 bytes. For chromatin 3D data representing all chromosomes at a resolution of 100kb, translating to 900 million edges, this would require $300\text{ bytes} \cdot (900 \cdot 10^6)\text{ edges} \approx 250\text{ gigabytes}$ of memory. Even with the capacity of todays hardware this is an uncomfortable memory requirement.

4.1.2 NumPy Arrays can Store Chromatin 3D Data Efficiently

Although the focus of this thesis is on running software on high-end servers with high memory capacity, limiting the memory usage is still important. First of all, reading from- and writing to memory is time consuming, so the amount of transmitted data affects the running time. Another reason for limiting the memory usage is the prospect for larger data sets, possibly outgrowing the memory capacity even for high-end clusters. NumPy arrays can be used to address the problem of creating a two-dimensional array from the underlying graph representation in HyperBrowser, where memory usage directly impacts running time and the ability to keep large data sets in memory.

A series of experiments (presented in table 4.1) shows that the memory size grows linearly with the number of elements in the matrix, both for Python lists and NumPy `ndarrays`. Although they both grow at linear rates, their total memory usage is quite different. While NumPy arrays occupy 8 bytes per element Python lists are using 4 times as much memory with about 32 bytes per element when storing float values of equivalent precision.

Size of matrix (number of rows/columns)	Python List (bytes per edge)	NumPy ndarray (bytes per edge)
5,000	37.58	15.94
7,500	35.03	11.52
10,000	34.73	9.98
15,000	33.71	8.88
30,000		8.22
60,000		8.05

Table 4.1: A comparison of the memory usage for Python Lists and NumPy `ndarray`.

Estimating the memory usage for a NumPy array is simple: upon creation a chunk of memory is allocated, and the size of this chunk is determined by the size of each element times the number of elements. All the elements in the array are of the same size and the array can thus be called homogeneous [18]. There is little overhead associated with a NumPy array: the header contains a "Data Type Object" describing among other things the type and size of the elements, but the size of this header is constant and does not depend on the size of the array.

Measuring the memory usage for a Python list however, is more difficult due to the way some Python implementations reuse immutable objects. If the elements in the list are of an immutable type they may refer to the same object, and the element will be a pointer to this object. The size of that element will depend on the size of pointers for that Python implementation, but generally they are either 32- or 64-bits depending on the processor. For instance: a large Python list with a lot of "None"-values will have pointers to the same "None"-instance, and so it will occupy less memory than a list of equal length filled with unique float-objects. As a result the density of the graph affects the memory usage of the list, making a sparse matrix more memory efficient than a dense matrix. This could be relevant for storage of chromatin 3D data, where the number of empty cells can be large.

In the case of chromatin 3D data each element is a float. NumPy provides several float data types of different sizes, while Python has one built-in float data type. The data type supporting the highest precision in the NumPy library is the "float64". As the name indicates a scalar of this type is stored using 64 bits, and is capable of storing decimal numbers with 11 bits exponent and 52 bits mantissa in addition to a sign bit. A float object offers about the same precision as the float64 data type and comparing a list of float objects to a list of float64 objects shows that the list of float objects occupies less memory. This indicates that the big advantage of using NumPy is not the float64 object itself, but the multidimensional arrays in which they can be stored. The explanation for the difference between in memory usage for two data types lies in the inevitable overhead associated with objects, as well as the overhead required to manage the list data structure. When an element from a NumPy array is extracted an object is created, "wrapping" the float64 scalar.

Even though nested Python Lists occupy far more memory than NumPy `ndarrays`, they can not be ruled out completely for use with Chromatine Interaction Data. Memory usage is one of several concerns, but in general the concern for running times is more important, assuming that the memory usage is not too exhaustive for the implementation to run at all, even on a high-end server.

4.2 Efficiency of Operations

Another important aspect to consider when comparing NumPy to pure Python is the efficiency of their operations. This section is comparing their efficiency by focusing on the operations that are likely to be used frequently for chromatin 3D data.

4.2.1 Vectorized Operations on `ndarrays` are Less Time-Consuming than Operations on Python Lists

Applying a function to all edges of a graph is a common operation when working with chromatin 3D data. The performance of such operations is therefore highly relevant when evaluating different data structures. Vectorization is a technique where the same operation operates on multiple scalars at the same time. NumPy includes many array-wide operations for computation on its multi-dimensional arrays, and they achieve great performance by exploiting the CPUs talent for vector processing.

When comparing the running time for some of the operations essential to analysis of chromatin 3D data, functions from the NumPy library outperform their equivalents from the Python Standard Library. Whether data is copied or transformed in-place makes a big difference in running times and memory usage.

In order to compare the performance of NumPy operations with operations from the Python Standard Library there are some challenges. While NumPy operates natively on multi-dimensional arrays, the Python Standard Library does not include arrays of higher dimensions. A simple Python implementation could be made as a sequence of sequences, where each sequence could either be a list or a tuple. Defining operations on a data structure like this soon becomes complex and difficult to comprehend. For the sake of clarity this can be simplified to a one-dimensional sequence, containing the same number of elements as its NumPy counterpart. This simplification probably leads to lower estimates than what would be the realistic case for a two-dimensional structure, but should give a decent indication for the purpose of comparison.

4.2.2 Native Python Lists are Superior to NumPy for Random Access

Here the term *Random Access* is meant to describe a usage pattern where elements from arbitrary positions within the array are accessed for reading

	Python List (nanoseconds per operation)	NumPy ndarray (nanoseconds per operation)
read	120	260
write	140	220

Table 4.2: Average time per operation for NumPy `ndarray` and Python List.

or writing. In working with chromatin 3D data this can be a useful way of accessing the data, for instance when performing significance tests involving random sampling.

The results were produced on the same server that runs HyperBrowser and shows an average time per operation for graphs of multiple sizes.

One possible explanation for the poor performance of NumPy `ndarray` stems from the time it takes to create an object in Python. While Python Lists store objects, NumPy `ndarray` does not and therefore an object must be created for each element being read, *when* it is being read. One might argue that the objects in a Python List must be created as well, but this happens upon initialization of the List and therefore the total time spent creating objects is not different, but the time is spent at an earlier stage when using Python Lists. There is no built-in caching mechanism storing the objects that have been read from a NumPy `ndarray`, so a computation where the same element is read many times will spend less time creating objects if the underlying data structure is a Python List compared to a NumPy `ndarray`.

4.2.3 Slicing can Improve Iteration Speed for NumPy `ndarrays`

Although many operations seem natural to implement using iteration, this can sometimes be avoided by extensive usage of the operations provided by the NumPy library. Still, in some cases, iterating through the elements of an array is necessary and doing this in an efficient manner can be time saving.

When iteration through elements in an `ndarray` is necessary, temporarily copying a part of the array and storing it as a Python List can be a cheap technique to improve the performance. This comes at the cost of increased memory usage from storing the copied slice of the array. Also, the size of the slices must be determined in such a way that the overhead associated with creating a slice does not diminish the benefit of increased iteration speed. However, even relatively small slices can improve the iteration speed. This technique can not be used directly to improve random access performance, but can be a serious improvement when iterating through a NumPy array in a predetermined order.

max([axis, out])	Return the maximum along a given axis.
mean([axis, dtype, out])	Returns the average of the array elements along given axis.
min([axis, out])	Return the minimum along a given axis.
sum([axis, dtype, out])	Return the sum of the array elements over the given axis.
sort([axis, kind, order])	Sort an array, in-place.
var([axis, dtype, out, ddof])	Returns the variance of the array elements, along given axis.

Table 4.3: An excerpt of the 53 methods that operate on `ndarray`s from the online NumPy reference [20].

4.3 Other Remarks

4.3.1 Giving Developers Access to the `ndarrays` Exposes the Full Power of NumPy

Besides running times and memory usage, an important part of a successful graph implementation is its ability to allow developers to utilize it in an efficient way without requiring too much knowledge about its inner workings. When programming with NumPy using the included operations is a safe way to ensure the most efficient implementation are being used. The same is desirable for a graph implementation. It is also desirable to encapsulate the underlying graph representation and bundle it with other associated data such as node identities and other meta data. One way of accomplishing this Python is by defining a class that acts as a thin layer of abstraction over the graph representation.

By implementing a thin layer of abstraction on top of a NumPy `ndarray` developers of statistics for HyperBrowser can access the `ndarray` directly and utilize all the functionality NumPy has to offer.

Although exposing NumPy data structures to developers requires knowledge of NumPy, the library has a simple interface with extensive documentation making it relatively easy to learn.

4.3.2 Readability of Python Code using NumPy

An important motivation for the development of Python has always been to guide developers towards writing readable (some will even say beautiful) code. This is an important part of the *Zen of Python* [22], acting as a style guide for Python programmers.

Following the best practices for writing readable Python becomes difficult when using NumPy. For instance, using indexes to lookup values in arrays can be considered less readable than using iterators, but is unavoidable when using NumPy.

On the other hand, NumPy code can be quite readable because the NumPy library provides a lot of the needed functionality. The developer should therefore focus on finding and applying the right NumPy functions rather than implement their own functions. This can possibly lead to more readable code considering the maturity and the quality of the documentation of the NumPy library.

The implications of how easy source code is to read and understand are many, and can often be overlooked in favor of other factors that are easier to quantify. Measuring readability is difficult and many arguments used in discussions concerning programming language design and readability are not exactly scientific. This argument is no exception, but might still be worth taking into account when determining which situations using NumPy is appropriate. When developing the statistic described in chapter 3 NumPy was essential for achieving acceptable performance, and bad readability would not be a valid argument for not using it.

4.3.3 Modularized Code and Overhead from Function Calls

There is a trade off between writing modularized code with plenty of functions and its performance. While both readability and possibly the correctness of the code might improve when using function calls and objects, it comes at the cost of performance. In contrast to compiled languages that can utilize *inlining* as a part of the optimization step during compilation, Python will go through the same expensive function call mechanisms every time a function is called. Inlining involves copying the body of a function to the places where the function is called from. This leads to duplication of code and thereby makes the size of the file to be executed larger, but has the advantage of removing the overhead from the function call procedure.

Chapter 5

Storing Meta Data With Graphs Can Improve Best Case Performance

The data structure representing a graph in HyperBrowser is flexible and allows data to be stored in several different ways. In some cases this flexibility is an advantage, but when developing algorithms where performance is critical, the cost of this flexibility can result in poor performance. To improve performance without sacrificing this flexibility altogether the implementation could use meta data to improve best-case performance.

Graceful degradation describes various types of algorithms that perform at their best effort, but are fault tolerant and work even for non-optimal conditions. In this context graceful degradation refers to the ability of the algorithms that perform operations on graphs to do as many optimizations as the data set allows for, while still functioning when the underlying data set is represented in a way that is less ideal. When the data set is represented in a way that is non-optimal for a given computation a gracefully degrading algorithm will still perform the computation, but at the cost of higher running times or higher memory usage.

There are a few pieces of information about the way a given graph is stored that can allow for highly optimized algorithms to process it. Instead of reducing the flexibility of the graph-holding data structure by limiting the ways data can be stored, meta data can be stored together with the data structure and hold information about which optimizations can be applied to a given graph. The meta data, or flags, can be created either when parsing the source file, or later by inspecting certain properties of the resulting data structure. This strategy does not change the worst-case performance of an algorithm, but improves its best-case performance.

5.1 Matrix Creation Can Be Optimized if Appropriate Flags are Set

An important feature of HyperBrowser is its "extendability", meaning that developers can extend its functionality by writing scripts to run as a part of HyperBrowser. For this to be successful the interface provided by HyperBrowser to the developers must be friendly and ideally should not require too much in depth knowledge about the internal workings of the system. Working directly on the data structures used to represent graphs at the lowest level is difficult and error prone, thus creating a simpler and more intuitive data structure for graphs is desirable. For chromatin 3D data a well suited data structure combining efficiency and a developer friendly interface is a two-dimensional NumPy `ndarray` consisting of weights between all pairs of nodes in the graph. By creating a single `ndarray` developers are exposed to a much simpler data structure than the one offered by HyperBrowser natively, and provides access to all of the operations provided by the NumPy library for working with this data structure. This simplifies the process of writing algorithms to be applied to graphs, and improves efficiency.

As described in section 3.1.2 graphs in HyperBrowser are stored in a data structure involving several `ndarrays`, so creating a matrix representation of this graph requires merging data from all of these in the correct way. The data structure for graph storage implemented in HyperBrowser is very flexible and a generalized algorithm for extracting a matrix representation from it is quite slow. One important reason for this is the lack of assumptions that can be made about the number of edges in the edges-array, and the ordering of the edges in it.

To make the resulting matrix as intuitive and predictable as possible for developers, it should follow a common practice when representing graphs as matrices where the order of rows and columns should be identical. This has the advantage of having its diagonal consisting of loops only.

The following chapters will show how storing certain properties of a graph as meta data can allow for optimizations, and the affect this will have on running times and memory usage. In comparing the following algorithms employing complexity analysis alone falls short to capture the significant differences in performance between them. The performance of the various algorithms is deeply dependent on which NumPy operations can be applied, so showing anything other than an actual implementation using Python and NumPy seems meaningless.

The most generic procedure for creating a matrix of weights from the graph representation implemented in HyperBrowser involves the following steps:

```
$m \leftarrow$ empty matrix
**for** each $id$ in $ids$ **do**
    **for** each $edge$ in $id.edges$ **do**
        $m_{id,edge} \leftarrow edge.weight$
    **end for**
```

```
end for
```

This algorithm contains two nested for-loops, resulting in a worst-case run-time of $O(n^2)$. When looking at the benchmarks of an implementation of this algorithm it becomes apparent that in addition to scaling poorly the inner-most statement is a costly operation. It involves writing a value to a single cell in a NumPy `ndarray`, which is known to be an inefficient operation.

5.1.1 If All Edge-Lists are Equal

By assuming that the edges appear at the same index in each edge-list the inner for-loop can be omitted, and thus reducing the worst-case run-time of the algorithm to $O(n)$. To accomplish this a flag can be held together with the graph indicating whether or not that assumption holds for the given graph.

```
$m \leftarrow$ empty matrix
for each $id$ in $ids$ do
    $m_{id} \leftarrow id.weights$
end for
```

Now the entire weight-list can be copied as-is, circumventing the expensive object creation operations needed in the previous algorithm. The resulting matrix may not have ids in the same order on the rows and columns, but the matrix will be complete and correct.

To determine if the flag that allows for this optimization can be set for a given graph, all the edge-lists must be compared to an arbitrary edge-list for equality and all the comparisons must yield true.

The optimization can be taken even further if the order of the ids in the new matrix *m* does not matter *or* if the order of the edges in edge-list is equal to the order of the nodes in *ids*. In that case the *weights*-matrix is already on the form we want, and can be directly copied to *m*.

5.1.2 If Nodes are Sorted by Position and Chromosome

Working with sorted data has a few advantages when developing algorithms. In order to sort a composite data structure a prerequisite is having an unambiguous definition of what constitutes a correct ordering. For chromatin 3D data an appropriate sorting of nodes can be based on *chromosome* and *position*. In this context position is defined as the start point of the bin along the linear representation of the chromosome. Chromosome indicates which chromosome the bin is contained within, but could be defined more generally to also include nonchromosomal elements such as *plasmids*.

The cost of having sorted data is the sorting process that has to be performed at some point. If several algorithms benefits from the data being sorted, storing the data in a sorted order might be a good idea. The cost of performing a sort operation once can easily be justified if multiple algorithms can take advantage of the data being sorted.

5.2 Mapping Linear Position to Nodes

Finding which node contains a given position is a common operation in working with chromatin 3D data. If we consider the nodes to be represented as intervals with a start position and an end position, the problem can be defined as determining which of a collection of intervals a given position is contained within. There are several ways to find which interval a given position belongs to depending on how the data is stored and whether the size of the intervals, referred to as bin size, is fixed.

The slowest algorithm involves doing a linear search. This means that every interval can potentially be looked at and the running time is $O(n)$ where n is the number of intervals. The linear search will work regardless of whether the intervals are sorted or the interval size is fixed. This is a good example of a situation where a seemingly sufficiently efficient algorithm, such as a linear search, can become a bottleneck because it is being used so frequently.

If the nodes are sorted by their start position within the chromosome this operation can be reduced to a $O(\log(n))$ operation, by performing a binary search.

Both the linear search and the binary search will work for graphs with fixed bin sizes and for graphs with varying bin sizes. However, in many cases the size of a bin is constant within a graph. If, in addition, the nodes are ordered by start position, the node containing a given position can be found by dividing the position by the (fixed) bin size. Rather than looping through a collection of intervals, a single computation is performed, giving a constant running time for this operation.

To determine which of the algorithms that can be used for a given graph, meta data can be stored together with the graph indicating whether or not the bin size is fixed and the nodes are sorted. By selecting the best algorithm for the job the best-case performance can be increased.

And as a side note: as a result of the overhead associated with function calls in Python, a simple way to reduce the running time is to let the function that does the actual work get a list of all positions instead of calling the function once for each position.

Chapter 6

Expensive Initial Processing Can Reduce The Total Running Time Of Monte Carlo Methods

Significance testing is an important tool in scientific research, and is briefly covered in the background material in section 2.2 on page 9. A practical approach to significance testing is through the use of Monte Carlo methods. Monte Carlo methods denotes computations involving randomization. Approximating statistical significance is one application of Monte Carlo methods. By randomizing the input to a given algorithm the same method used to calculate the observed result can be used to generate samples for a probability distribution. Using Monte Carlo methods for hypothesis testing has several advantages: By using the same code for computing the result of a statistic and for performing hypothesis testing less code can be written. Another benefit of code reuse is that the computations involved in hypothesis testing will be as similar as possible to the computations involved in computing the original statistic.

An important feature of HyperBrowser is its ability to perform hypothesis testing. HyperBrowser is built around the assumption that calculating the statistical significance of the results it produces is a common operation. To simplify development of hypothesis tests within HyperBrowser developers are provided with a "framework" for calculating the statistical significance in the form of P-values. In order to create a hypothesis test in HyperBrowser all that is needed is a definition of the test statistic to be employed.

Based on this test statistic a series of Monte Carlo simulations will be performed with randomized data. By comparing the results from computations on randomized data with the result of the same computation on "real" data a P-value is determined. A method for randomizing the data has to be defined for each hypothesis test, to tailor its domain specific needs. Some of the more common methods for randomization are predefined in HyperBrowser, but additional methods can be implemented as well.

6.1 Performance of Monte Carlo Methods

Employing Monte Carlo methods for hypothesis testing can have important implications for the overall performance of the computation. It is not uncommon to perform thousands, or even tens of thousands, of simulations in a single hypothesis test. A significant portion of the total running time for the computation can be related to these simulations. The reliability of the P-value increases with the number of Monte Carlo simulations, so the more simulations one can afford to perform the better. The cost (running time) of each Monte Carlo simulation should therefore be as small as possible. To achieve this all the operations that do not rely on randomized data should be done outside the simulations and its results shared between simulations. If, for instance, one of the data structures needed for a computation can be identical for all simulations it should be computed once, stored and reused across the simulations, rather than recreated for each simulation.

Within HyperBrowser adhering to the principle of reusing intermediate results and randomizing as few parameters as possible is manageable. By following the convention of encapsulating all independent computations within "Stat"-objects their results will automatically be reused and shared between all Monte Carlo simulations. HyperBrowser silently takes care of re-processing the computations that are based on randomized data, while keeping and reusing the results from the computations that are based on non-randomized data. Refer to section 2.5 on page 14 for a more elaborate explanation of the HyperBrowser architecture and design.

By using this technique the way the running time is reduced is essentially by keeping more data in memory throughout the whole hypothesis testing computation. This requires increased memory usage and can be quite memory intensive. When considering the possibility of larger data sets the extensive memory usage can prohibit the use of this technique.

Chapter 7

The Relationship Between Memory Usage and Running Time

As stated in the problem description (see section 1.1 on page 3) the memory usage is being treated as less important than the running time in this thesis. This is a pragmatic choice based on the assumption that the CPU speed will be the limiting factor for analyses of chromatin 3D data. Keeping up with the increasing demand for fast storage can be tackled simply by adding more main memory. For computational speed however this is not as simple: the clock speed of the CPUs have stagnated and so increasing the capacity of a computer system means adding more processors or more cores. In order to achieve lower running times when more processors are being added the computations must be constructed in a way that supports at least some degree of parallelization.

So while the computer systems dealing with chromatin 3D data will most likely run on hardware where the size of the main memory increases at a sufficient rate, the speed of the individual CPUs will remain the same. Memory usage is therefore not a case of dealing with limited resources, but the impact it can have on the running time can be an issue.

This chapter examines the relationship between memory usage and running time. It is written from a more generalized perspective than previous chapters, and addresses some of the fundamental challenges for any performance critical application dealing with big data. But the topics of this chapter are important for the future of chromatin 3D data analyses, as the data sets might outgrow the current capacity of the computational methods.

7.1 CPU caches, RAM and HDD: Speed and Size

In software development it is common practice to abstract away the actual hardware. The location and storage medium of the data being used in a program can easily be abstracted away in modern programming languages.

Device	Cycles	Time
CPU register	0 cycles	a few nanoseconds
CPU cache	1 to 30 cycles	a few nanoseconds
main memory	50 to 200 cycles	10-100 nanoseconds
disk	tens of millions of cycles	3-12 milliseconds

Table 7.1: The approximate time and number of CPU cycles spent when accessing different storage devices. (Sources: "Computer Systems: A Programmer's Perspective" [2, chapter 6] and "Database Systems: The Complete Book" [7, chapter 13])

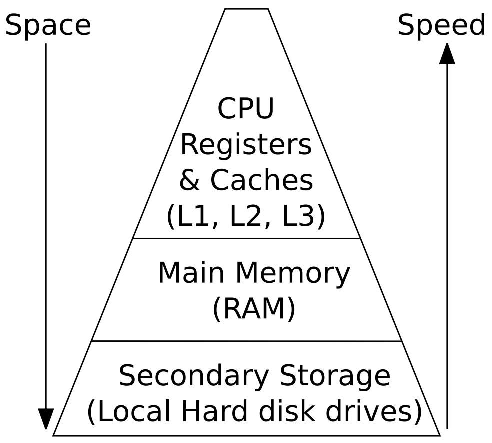

Space
Speed
CPU Registers & Caches (L1, L2, L3)
Main Memory (RAM)
Secondary Storage (Local Hard disk drives)

Figure 7.1: The relationship between the storage devices in the memory hierarchy. The access speed increases towards the top of the pyramid, while the space they provide increases towards the bottom of the pyramid.

There are good reason for this, but abstractions comes at a cost. In high performance computing this cost can become unbearable, and knowing and exploiting the different properties of the hardware is important.

Much in the same way the laws of physics applies even for the abstract plans of an architect, the performance of a computer program is constrained by the hardware it runs on. Analyzing the characteristics of the hardware gives valuable information about the constraints and challenges that must be addressed. The performance of storage devices are highly relevant to data intensive application.

7.1.1 The Memory Hierarchy

Modern computer systems have a *memory hierarchy* consisting of different components capable of storing data. Besides the CPU caches there are usually two types of storage devices available in a computer system. The main memory, often referred to as *primary storage*, and the hard drive, often referred to as *secondary storage*. In both categories there are vendors and models with different performance characteristics, but the difference in time for reading and writing to the CPU caches, main memory (RAM) and Hard disk drive (HDD) is significant for all.

Even with the recent progress of *solid state drives* (SSD) the performance gap between secondary storage and primary storage is a big concern in high performance computing. Like many problems in scientific computing (and in many other areas for that matter), the problems concerning chromatin 3D data are closely related to the size of the data sets involved in the computations. This makes the time it takes to access the data highly relevant, because each access operation has to be performed frequently to process the entire data set. Where the data resides, whether it is in the CPU cache, RAM or on a hard disk drive, can be what determines the execution speed of a program.

An approximate presentation of the access times for the different devices is provided in table 7.1. Since accessing the CPU caches is clearly the fastest of the three, using them actively seems like a good idea. The limited storage space provided by the CPU caches makes it impossible to store a large data structure in it. Besides, while effective usage of CPU caches can improve the overall performance of a program, this is not something developers of higher level languages like Python deal with directly. Programs can be designed to take advantage of the CPU cache, but in general CPU caching is performed “behind the scenes”.

The next best thing performance wise is the main memory. The difference in access times between main memory and disk is substantial, so whenever a program needs to read from or write to disk the chances are it will introduce a significant latency. As a rule of thumb the more you can fit in memory, the faster the computations. This is true up to a certain threshold where memory access is no longer the bottleneck.

The most important characteristics to consider when evaluating the performance of a storage medium is its *latency* and *throughput*. While latency is the time it takes to receive a reply from a request, the throughput denotes how much data can be transfered per time unit. When accessing small segments of data in a random order the latency will be the limiting factor, while reading a larger portion of data sequentially will be limited by the throughput.

In the memory hierarchy pyramid in figure 7.1 the latency increases and the throughput decreases towards the bottom.

The operations involved when reading from main memory is quite different from the operations required to read from a disk. While accessing CPU caches and main memory is measured in clock cycles, accessing data on disk is measured in time. In other words the time it takes to read from main memory is relative to the frequency the CPU operates at. For disk access this is not the case. Accessing the disk is an asynchronous operation carried out by a disk controller. This means that a CPU running at a higher frequency can not automatically perform disk access operations any faster.

7.2 Virtual Memory and Swapping

An example of how abstractions can come at the cost of performance is the *virtual memory* abstraction and *swapping* as implemented by most modern operating systems. Virtual memory has many benefits, The advantages of using swapping to increase the capacity of the virtual memory in general purpose computer systems are many, but for high performance programming swapping is “deadly”. A *page fault* occurs whenever the processor wants to access data that no longer resides in the physical memory because it has been swapped out.

7.3 Storing Additional Data Structures to Improve Performance

By adding additional data structures such as indexes the efficiency of certain operations can be improved. This is different from caching in that the data being stored in these data structures are not exact copies of the data that resides on disk. Instead the additional data structures can hold the results of precomputed calculations or other information, augmenting the original data.

The profitability of adding an additional data structure is highly dependent on the access patterns for the data: If the data is read frequently and rarely modified maintaining additional data structures can be cheap. On the other hand, if the data is frequently modified the cost of keeping additional data structures updated can diminish their positive effect on performance. This stems from the fact that each additional data structure must be updated to be consistent with the contents of the primary data structure. Additional data structures require additional space, and ideally they should reside in memory. For certain applications keeping only an additional data structure in memory and the primary data structure on disk can be profitable.

In the case of chromatin 3D data the primary data structure is the graph used to represent the interactions captured by Hi-C and other methods, while an additional data structure could be an index of all edge-weights. The chromatin 3D data is written once and read frequently, meaning that it is cheap to keep additional data structures updated.

Various forms of indexes can be found in a lot of software (such as database systems and search engines). An index can be seen as any redundant data structure aimed at improving performance. The data in an additional data structure is redundant in the sense that if it was deleted the information it held would still be present in the main data structure, although the organization of the data would be different. Indexes and other data structures aimed at improving the running time for frequent operations highlights the trade-off between running time and memory usage. Simply put adding an index will in most cases increase performance drastically, but at the cost of increased space. For optimal performance the index should be stored in memory. In some cases, when a relatively small subset of the data set is being used, it is more important to keep the index in memory than keeping the primary data structure in memory. Keeping the index in memory allows for fast lookup while avoiding to read the entire data set into memory.

7.4 Memory Mapping Reduces the Memory Requirement at the Cost of Speed

A *memory mapping* is a virtual memory space where each byte has a corresponding position on disk. Unix based operating systems provide a function for creating memory mappings called `mmap`. `NumPy` provides a convenient interface to `mmap` though `numpy.memmap` to facilitate memory mapping of `ndarrays`. There are mainly two reasons for using memory mapping: providing faster random access to a file or creating a "private" virtual memory space. When a file has been memory mapped it works like virtual memory by swapping pages in and out on demand. This makes it possible to write programs that require more memory than what is physically available on the hardware it runs on. The operating system already employs virtual memory to (among other reasons) increase the total memory capacity of the system, but creating a "private" memory mapping for a process and thereby giving the process its own virtual memory can still be useful. For instance in situations where even the virtual memory provided by the operating system is insufficient, or to provide greater control over what to swap and what too keep in memory. The cost of using memory mapping is running time: pages will be swapped in and out and each swap is expensive as it involves reading and writing to secondary storage. Reading and writing to a memory mapped file does not happen until the corresponding memory area is touched, and for this reason memory mapping can reduce running times when only parts of a memory mapped file is being accessed. Memory mapping makes it easier to work with data sets that are too big to fit in memory, by swapping pages transparently. Memory mapping can be a tool for speeding up computations, but its greatest benefit is that it makes it possible to perform computations that require more memory than what is physically available.
[2, Chapter 9.8.4]

7.5 Efficient Employment of MapReduce

MapReduce can be a powerful technique for increasing throughput within limited memory constraints. MapReduce works by dividing a problem into subproblems, or *tasks*, where each task can be performed separately and its results combined to produce the total solution. The performance of an algorithm using MapReduce highly depends on the number of tasks the problem is divided into.

7.5.1 MapReduce for Reducing Running Time

If the tasks are performed in parallel the optimal number of tasks is dependent on the *overhead* associated with creating and maintaining each parallel process. There are many ways to achieve parallelism, from *distributed computing* to *threads* or *GPU programming*. Common to all parallel computing technologies is that each parallel process has some overhead associated with it. This means that splitting a serial computation into two tasks does not reduce the running time by half, because the overhead of running a process is now doubled. The actual overhead varies in both *time* and *space* for the different parallel computing technologies. As a result the optimal number of subproblems depends on the technology used to achieve parallel computing.

7.5.2 MapReduce for Reducing Memory Usage

Even if the tasks in a MapReduce computation does not run in parallel there are some benefits compared to a serial computation. The amount of memory required to perform a computation can be reduced by dividing the data that needs to reside in memory. Each task will only have access to the data it needs in order to perform its computation. For serial computations the optimal number of tasks to divide the computation into depends on the amount of available memory. Currently this is the case for HyperBrowser. It does not use parallelism, but limits the memory usage by performing computations on smaller parts of the data sets.

There is also a special case where the running time is reduced as a result of reduced memory usage: if the computation requires more memory than what is available, and thereby forces swapping to occur, the running time will suffer drastically. In this case dividing the computation into smaller tasks and thereby reducing the memory requirement will reduce the running time, even if the tasks are not performed in parallel.

7.6 Using Compression to Reduce IO

Various data compression schemes can be applied to reduce the size the data structures occupies on disk, and thereby reduce the amount of data that needs to be transfered between disk and main memory. This is beneficial if the time it takes to read the compressed data into memory and decompress it is lower than the time it would take to read the uncompressed data into memory. Even if there is no difference in running times one could argue for using compression anyway because lowering the size the data occupies on disk is a benefit in itself. This is one of the techniques used by the PyTables package [23] to improve performance. The basis for this optimization is that CPUs have “outperformed” memory access speeds.

Chapter 8

Neo4j is not a Quick Fix

With the recent developments in graph database technology, mainly driven by Neo4j [17], the interest for graph databases has increased. This increased interest can perhaps be seen in light of the new possibilities opening up for analyzing social networks. Also, the interest for *big data* and alternatives to relational databases (often referred to as *NoSQL*) has increased recently. Storing biological data poses a serious challenge with its large data sets, but technologies such as Neo4j can not be ruled out without proper investigation. The licensing policy for Neo4j [16] permits its use in Open Source software free of charge, but requires a commercial license if it is included in a closed source product. Together, this makes Neo4j a possible candidate for use in HyperBrowser, at least in theory.

In this chapter different aspects of Neo4j will be discussed, but the most important issues are performance and scalability both in terms of running times, memory usage and disk usage.

8.1 Graphs in Neo4j

In Neo4j terminology a graph consists of *nodes* and *relationships*, where a relationship is the equivalent of an edge. Both nodes and relationships can have *properties* associated with them. Properties are stored in key-value pairs, and their values can be one of many data types such as integers, floating point numbers and strings. All meta-data must be stored as properties, such as the name of the node and the weight of the edge/relationship. The use of properties makes Neo4j flexible in terms of what the graphs can represent. It also makes achieving high performance easier, because the internal representation of nodes and relationships can be limited to integers rather than various data types.

8.2 Client-Server Communication with Neo4j

Neo4j can be run as a stand-alone program, or embedded as a part of a program running on the *Java Virtual Machine* (JVM). Either way Neo4j can be perceived as a server, and the programs that are using its services can be called clients, and in the rest of this chapter they will be referred to as such.

To interact with the Neo4j server from a Python program there are currently two alternatives: communicating with a REST API or using Neo4js Java interface through JPype [13]. There are multiple third-party Python libraries built on the REST API, and one library [24] (developed by the creators of Neo4j) that calls Java methods directly through JPype.

Regardless of which interface is being used for communicating with Neo4j, a decision has to be made between using Neo4j as a “thin” back-end for storing nodes and edges, or as a “thicker” back-end performing computations as well as storing the contents of the graph. Determining which operations should be performed by the database system and which should be performed by the client is a familiar and ongoing discussion among software developers. Many relational databases have procedural languages embedded within them, allowing the development of functions and complex logic as a part of the database system. Neo4j has a similar capability through its query languages, and so the question of where to implement different parts of the logic is relevant when using Neo4j as well. A discussion of the two most distinct ways of using Neo4j takes place in section 8.3 and section 8.4.

```
for relationship in db.relationships:
    a_function(relationship['property_stored_in_relationship'])

for node in db.node:
    a_function(node['property_stored_in_node'])
```

Listing 8.1: A typical pattern when using Neo4j as a thin server for storing and retrieving data. In the first part of this example all relationships in a database are fetched and `a_function()` is applied to a property stored on each of the relationships. In the second part the same technique is used on nodes. The interface is provided by Neo4js Java API through JPype.

8.3 Neo4j for Storage and Retrieval

One way of using a database system is as a simple *persistence layer* providing storage for more or less structured data without dealing with the file system directly. The benefits of using a database system instead of operating on a file system are many, even if the use of the database is limited to storage and retrieval. Neo4j can be used in such a way, hereafter referred to as a *thin* server (based on the notion of a *thin client*).

If Neo4j is used as a thin server all computations would be performed by the client after fetching the requested data from Neo4j. In other words Neo4j is used as a simple facility for storage and retrieval of nodes and relationships with their associated properties. For the case of HyperBrowser the client would be a Python program, and so all the processing of the data from Neo4j would be performed in Python. There are especially two performance related issues with using Neo4j as a thin server communicating with a Python client:

First, every “entity” fetched from Neo4j such as nodes, relationships and properties would have to be encapsulated in Python objects in order to be accessible for the Python program. For instance, applying a function to all the edge weights of a graph would require all the edges to be fetched from Neo4j and represented as Python objects before the function could be applied. Creating objects is expensive (both in terms of running time and memory usage) in Python, and based on the typical size of the graphs involved in chromatin 3D data analyses this could result in a serious performance issue.

Secondly, depending on the the communication protocol used, overhead from client-server communication could be significant. If communication with Neo4j is performed over its REST API this would undoubtedly suffer from poor performance as a result of the overhead involved in communicating over HTTP.

An informal experiment (shown in listing 8.2) performed by fetching all 62500 relationships from a graph through a REST API and computing the sum of the “weight”-property stored on each relationship showed that the running time could be estimated to about 2 milliseconds per edge. Under the optimistic assumption that the running time of this operation will scale linearly with the number of edges this would still result in a running time of 5 hours for fetching all edges in a graph storing chromatin 3D data at a 1mb resolution. This obviously flawed experiment is not of much use, but does give an indication of how inefficient communicating through a REST API can be.

```python
from py2neo import neo4j, cypher

db = neo4j.GraphDatabaseService("http://localhost:7474/db/data/")

query = "START n = node(*) MATCH n-[r]->m return r"
relationships, metadata = cypher.execute(db, query)

sum_of_weights = sum(relationship[0]['weight'] for relationship in
    relationships)
```

Listing 8.2: Running a query using the py2neo REST interface to return all relationships for further processing in Python, where the sum of all the edge weights is computed.

8.4 Neo4j as a Computational Engine

Another way of using Neo4j is to utilize its capabilities as a computational engine. This is in contrast to the thin back-end approach described previously, where the database is only used for its storage and retrieval capabilities. To perform computations in Neo4j the computational procedure must be expressed in one of the supported languages for graph querying, or by using the provided Java methods in the Core API and the “Traversal Framework”. Currently the supported graph querying languages are Cypher [29]

Nodes	Edges	Running time (milliseconds)	Time per edge (milliseconds)
250	62,500	157	0.00251
500	250,000	672	0.00269
750	562,500	2,040	0.00363
1,000	1,000,000	4,294	0.00429

Table 8.1: The time it took (best of 3) to run the Cypher query in listing 8.3 on graphs of different sizes.

and Gremlin [9].

8.4.1 Cypher

Cypher is one of the “graph query languages” supported by Neo4j. It has been developed as a part of the Neo4j package, and is a core part of Neo4js functionality. Cypher bears resemblance with other query languages such as SQL [34] and SPARQL [33] both in that it is a declarative language and in its choice of syntax and keywords. Cypher can be used both for creating, updating and deleting data as well as reading and performing computations. In the case of chromatin 3D data the most important requirements are fast read operations and efficient computations. Rather complex computations can be expressed in Cypher and performed in its entirety by Neo4j. This requires the computation to be expressed in terms of the operators and functions provided by Cypher, and it is hard to determine how far this can be stretched. However, by using a combination of Cypher queries for fetching data and Python programming for further processing, any computation should be theoretically possible to perform.

The consequences of Cypher being a declarative language is the source of both its strengths and weaknesses. Not having to deal with *how* a query is performed, but simply defining *what* it should return can makes queries easier to write and read. However, estimating the performance of a query can be difficult due to the declarative nature of the language: Neo4j will do its best to transform the declarative query to an efficient procedure, but this process is not apparent to the author of the query and is not always guaranteed to produce the most efficient procedure.

```
START n=node(*)          //select all nodes
MATCH n-[r]->m           //for each node n, select all
                         // outgoing relationships
RETURN sum(r.weight)     //sum the value of the property
                         // ‘‘weight’’ on each remaining
                         // relationship
```

Listing 8.3: Calculating the sum of all weights in a graph using Cypher.

From the running times listed in table 8.1 from running the Cypher query in listing 8.3 it is apparent that the time it takes to run the query does not relate linearly to the number of edges in the graph.

Another observation (although not included in the table) is that the first time a query is performed and the caches are cold, the query takes a long time to complete, but as soon as they become "warm" the running time decreases significantly.

These two observations can be related, and the fact that the running time does not increase linearly in accordance with the number of edges can be due to insufficient amounts of available memory. If the graph can not be kept in memory it will increase disk access and thereby slow down the computation. No definite conclusion can be drawn from these results, but it can be used as an indication that Neo4j does not scale greatly "out of the box" and needs configuration. A further discussion on the memory usage of Neo4j is discussed in section 8.6.

```
START n=node(*)          //select all nodes
MATCH n-[r]->m           //for each node n,
                         // select all
                         // outgoing relationships
WHERE r.weight > 0.9     //filter out all
                         // relationships with
                         // weight less than 0.9
RETURN avg(r.weight)     //calculate the average value of
                         // the property ''weight'' on each
                         // remaining relationship
```

Listing 8.4: Additonal example of a Cypher query: getting the average of all the edge weights above 0.9 in a graph.

8.4.2 Gremlin

Gremlin is a graph traversal language based on the Groovy [11] programming language. Neo4j supports the use of Gremlin through a plugin, and is capable of running arbitrary Groovy scripts. This means that Gremlin code can be seamlessly blended with Groovy code and computations can be performed efficiently without overhead from communication protocols.

For the purpose of HyperBrowser some of the same challenges as for Cypher also applies to Gremlin: writing computations for chromatin 3D data will require the use of a third-party language.

The only communication between the client and the server will be a request from the client containing the Groovy/Gremlin code, followed by the results of the computation from the server. This way the time spent communicating between Neo4j and the Python program that initiates the computation is very limited, and the costly operation of creating objects in Python for each entity fetched from Neo4j can be avoided entirely. This means that the performance bottleneck will be in Neo4js computational engine rather than in the Python code. Whether communication with the database is performed through the REST API or by calling Java methods through JPype becomes far less important in this case because the number of messages between the Python program and the database is minimal. This means that deciding between running Neo4j as an embedded component of the program or as a standalone server becomes a question of which is more efficient.

8.4.3 Summary

By expressing computations in one of the query languages supported by Neo4j the entire computation can be performed by the Neo4j server. This reduces the communication between the client and the server to a minimum and the entire computation can be performed fairly efficiently.

The major drawbacks of expressing computations in a query language is that it can be challenging for developers unfamiliar with it and requires snippets of Cypher or Gremlin code to be a part of the HyperBrowser code base.

Cypher is constructed as a language for describing graph traversals: it defines starting points (nodes or relationship) and rules for how traversal from those starting points can be performed. This way of thinking about computations on chromatin 3D data is quite different from the matrix operations currently in use. Another drawback is that introducing Neo4j as a dependency for the HyperBrowser system increases the need for system administration.

8.5 Graph Serialization and Disk Usage

Neo4j, like most database systems, stores the contents of a database in a number of files, as can be seen in listing 8.5. In Neo4j most of the files that make up a database has a revealing name that indicates what it contains.

```
>> du -a -hm
1       ./active_tx_log
1       ./index/lucene-store.db
1       ./index/lucene.log.active
1       ./index
1       ./messages.log
1       ./neostore
1       ./neostore.id
1       ./neostore.nodestore.db
1       ./neostore.nodestore.db.id
40      ./neostore.propertystore.db
1       ./neostore.propertystore.db.arrays
1       ./neostore.propertystore.db.arrays.id
1       ./neostore.propertystore.db.id
1       ./neostore.propertystore.db.index
1       ./neostore.propertystore.db.index.id
1       ./neostore.propertystore.db.index.keys
1       ./neostore.propertystore.db.index.keys.id
1       ./neostore.propertystore.db.strings
1       ./neostore.propertystore.db.strings.id
32      ./neostore.relationshipstore.db
1       ./neostore.relationshipstore.db.id
1       ./neostore.relationshiptypestore.db
1       ./neostore.relationshiptypestore.db.id
1       ./neostore.relationshiptypestore.db.names
1       ./neostore.relationshiptypestore.db.names.id
1       ./nioneo_logical.log.active
1       ./tm_tx_log.1
1       ./tm_tx_log.2
71      .
```

Listing 8.5: The ouput from du showing the size of each file in the database in megabytes. The database in this example has one thousand nodes and one million edges.

Table 8.2 shows the disk usage per edge for graphs of different sizes, and the size of the two most significant files storing relationships and properties. Drawing from the observed sizes, the total disk usage of a database can be estimated to be about 74 bytes per edge with one property

Number of edges	Size of database per edge	Size of relationshipstore per edge	Size of propertystore per edge
10,000	109 kB	<1 bytes	95 bytes
62,500	77 kB	33 bytes	42 bytes
250,000	75 kB	33 bytes	41 bytes
562,500	75 kB	33 bytes	41 bytes
1,000,000	74 kB	33 bytes	41 bytes

Table 8.2: The measured disk usage per edge for graphs of different sizes using Neo4j.

stored on each edge, and one property stored on each node. The size of the properties will most likely vary greatly according to data type and value, but this is a realistic implementation for storing chromatin 3D data where each edge has a weight and each node has a name. According to these results storing chromatin 3D data for the entire humane genome at a resolution of 100kb would require approximately $9 \cdot 10^8$ edges $\cdot$ 74 bytes $\approx$ 62 gigabyte

This can be considered large, as the property stored on each edge only carries 8 bytes of information in the form of a 64-bit float representing the edge weight. But this alone might not be large enough to rule out Neo4j entirely as a storage facility for chromatin 3D data. Other aspects are more important such as memory usage and running time. Also, this might not be the most efficient usage of Neo4j and it can not be ruled out that the serialized graph could be smaller with a different implementation.

Another observation from the same experiment is that the disk usage in a Neo4j database seems to scale with the number of nodes, edges and properties. When the size of the graph gets above a threshold of approximately 60 000 edges the size of the database is determined with great accuracy by the number of nodes, relationships and properties. This is a positive sign for the scalability of Neo4j as a storage facility, even though the absolute size can be seen as quite large.

8.6 The Performance of Neo4J is Constrained by Memory

Although the way Neo4j stores graphs may be sufficiently compact it does not escape the fact that main memory is a limited resource and disk access is relatively slow. To perform an operation with acceptable performance a big part of the data that is required for the operation (for instance a graph) must reside in memory. In large part the performance of Neo4j is constrained by the amount of memory available. This is supported by the official Neo4j documentation, stating that “Neo4j tries to memory-map as much of the underlying store files as possible. If the available RAM is not sufficient to keep all data in RAM, Neo4j will use buffers in some cases, reallocating the memory-mapped high-performance I/O windows to the regions with the most I/O activity dynamically. Thus, ACID speed degrades gracefully as RAM becomes the limiting factor.” [28].

Neo4j supports some of the techniques mentioned in chapter 7 to improve performance, such as memory mapping and indexing. Memory mapping makes operating with limited memory resources possible, at the cost of speed. Indexing on the other hand can improve the speed of lookup operations (finding a node or edge given an attribute it possess), but at the cost of memory. Generally, when discussing indexing, the question of which properties should be indexed depends on the queries that will be performed. The size of the resulting index is difficult to estimate, but it will at least have some impact on the memory usage.

A common approach among modern database systems to tackle the memory requirements posed by large data sets is to facilitate interconnection of multiple computer systems into *computer clusters*. Neo4j supports clustering of instances, coined as *High Availability*, but is only available in the *Enterprise Edition* of Neo4j [15].

8.7 Performing Read Operations Outside of Transactions can Increase Performance

Neo4j is aimed at storing, modifying and querying graphs while having some of the properties expected from a modern database system. An important feature for many database systems, including Neo4j, is being *ACID compliant*. ACID is a set of requirements regarding the integrity of the data being stored in the database. The mechanisms implemented to enforce ACID compliance are mostly related to the operations that modifies the database, and are therefore not overly important in the context of analyzing chromatin 3D data which only involves read operations. However, there is one ACID related mechanism that *can* effect read performance: *transactions*.

A transaction is an atomic unit of work that can contain many operations, but will be performed as one operation without interference from other operations. The concept of a transaction is vital to the ACID principles, and handling transactions makes up a substantial part of a database systems workload. Disabling the transaction handling mechanisms, or performing operations “outside” of a transaction, can thus increase the overall performance of most database systems, including Neo4j. However, the biggest increase in performance is related to write operations, and this is likely a rare operation for chromatin 3D data. Chromatin 3D data is typically written once and read often, and the performance increase on read operations from disabling transactions is limited. Neo4j allows read operations to be performed outside transactions by default.

The use cases Neo4j and most other database systems are developed for differs quite a bit from the use case presented in this thesis. The emphasis on data integrity, data security and concurrent write operations common to database systems is irrelevant for chromatin 3D data in HyperBrowser.

It is likely that enforcing these unnecessary constraints comes at the price of performance, and therefore configuring the Neo4j server instance to employ only a minimum of them can potentially improve its performance.

8.8 Architectural Challenges when Storing Multiple graphs

Neo4j works with the concept of databases as a collection of graphs. This leaves two alternatives for storing chromatin 3D data in Neo4j: storing all graphs in one database, or storing graphs in multiple databases with possibly as little as one graph per database.

Each genome is represented as a graph, and often there are multiple versions of the same genome at different resolutions. If all the genome graphs are stored in the same database this database will quickly become very large, and possibly outgrow the current limit of approximately 34 billion relationships (meaning less than 40 genomes at a 100kb resolution). It is possible to increase the number of relationships that can be stored, but to increase this limit the size of the data type used for relationship identifiers must be increased. This would result in an overall increase in the size of the database both on disk and in memory.

The genome graphs could alternatively be stored in different databases. This could make it problematic to run queries involving multiple graphs, spanning multiple databases. If Neo4j is used as a computational engine (as described in section 8.4) querying multiple databases might not even be possible.

8.9 Implications for HyperBrowser as a Software Project

If Neo4j was to be included as a part of the HyperBrowser project, there would also be some implications unrelated to performance and strictly technical aspects.

It is impossible to know what the future might hold, and tomorrows requirements to the storage facility of chromatin 3D data might be different than todays requirements. This means that being able to extend or modify the implementation is crucial. Although Neo4j is open source and can thus be modified, doing so would possibly require a lot of work. Unlike open source software projects created in a truly collaborative way, Neo4j has been created by a commercial vendor by a relatively small team of developers. This means that the importance of making the source code easy to understand and thereby contribute to is possibly lower for a project like Neo4j, and thus making it difficult for third-parties to extend or modify.

Chapter 9

Conclusions

To add support for analysis of chromatin 3D data to the HyperBrowser framework, a number of implementations were developed and assessed. The assessments were based on a practical use case from developing the computational methods needed to support a hypothesis test for assessing the spatial co-localization of regions within a genome. Although drawing conclusions from a single use case can be misleading, several shortcomings of the implementations were successfully identified. After several iterations an implementation with satisfying performance was reached.

Throughout the process of improving the implementations several observations were made and examined further:

The performance of Python programs can be improved with NumPy.
Python is a high-level interpreted programming language with dynamic typing. It offers great expressibility, at the cost of performance. In particular, there is significant performance cost associated with object creation and function calls. Developing efficient Python programs can thus become incompatible with developing well structured and modularized code. And even if the concern for readability is completely ignored, objects are such an integral part of Python that limiting their usage sufficiently is almost impossible. Several tools for improving the performance of Python programs are available, among them NumPy. Parts of the NumPy library proved to be well suited for managing and processing chromatin 3D data. Compared to pure Python operations, employing NumPy significantly improved the performance of the computational methods.

Meta data can increase best case performance while sustaining flexibility.
The data structures responsible for storing chromatin 3D data in Hyper-Browser are flexible. Some properties were found to potentially improve the performance of certain operations. In particular, information about whether or not nodes and edges are sorted, if the graph contains loops and if the graph is complete. The cost of determining (or even enforcing) these properties are one-off costs that occur when a new data set is being con-

structed. Associating meta data with each installed data set would be a pragmatic way of improving the best-case performance of some important operations. This meta data would contain enough information for each operation to decide whether optimizations can be made or not.

Monte Carlo simulations can easily become a bottleneck.
When a large number of Monte Carlo simulations are performed, limiting the operations that are necessary for each simulation is crucial. This can be achieved by reusing results and caching data that are common for the simulations.

The Neo4j graph database is not a solution as-is.
The Neo4j graph database is a promising tool for storing and querying graph data. It is capable of storing graphs in the most flexible way and supports complex queries to be expressed with specialized graph query languages. Several challenges were discovered from experiments with storing and retrieving chromatin 3D data using Neo4j. To achieve acceptable performance the graph operations must be expressed in a graph query language. Although this language is flexible and capable of performing a number of different operations, expressing computations involving chromatin 3D data does not seem like a good fit. Even if computations were expressed in the graph querying language, the performance of the queries were not convincing and the high memory and disk usage were unsettling. Neo4j should not be dismissed entirely in the context of chromatin 3D data, but there is great uncertainty surrounding both its performance potential and the appropriateness of its interface.

From this master's thesis project a tangible result has been produced: analyses of chromatin 3D data can now be developed in HyperBrowser. But the findings presented in this thesis are relevant beyond this. Several other computational tools will face the same challenges that motivated this work. There is already a desire for computational tools to integrate and analyze chromatin 3D data, and this desire will likely increase with the growing interest in this type of data. The size of the data sets requires the implementations to be carefully designed. Storing and representing the data in an efficient way that encourages high performance is an absolute requirement, as it lays the foundation for all further use.

9.1 Future Work

The challenges that come with supporting chromatin 3D data analyses are not solved for good. Although HyperBrowser is capable of performing analyses on the currently available data sets, higher resolution data sets may pose new challenges to the current representation of chromatin 3D data.

Improve Implementation # 4
A concrete improvement that could be made in the near future is the implementation referred to as implementation # 4 (described in section 3.2.4 on page 22). This implementation is the first step towards parallelism for chromatin 3D data analyses in HyperBrowser, and could potentially improve the efficiency of certain operations significantly.

Support for Graph Operations
To provide a complete programming interface for chromatin 3D data the necessary components to support graph specific operations such as traversal or path finding should be developed. Implementing operations like these in an efficient manner may require the use of Cython or other similar tools for integrating high-performing low-level programming languages (such as C, FORTRAN or GO) with a Python code base.

Using Compression to Increase Performance
An interesting idea is the use of compression to improve running time. This was briefly mentioned in section 7.6 on page 50. A thorough investigation of how this technique could benefit the performance of chromatin 3D data analyses (or HyperBrowser in general for that matter) could reveal interesting results.

Bibliography

Journal papers

[5] G. Fudenberg, G. Getz, M. Meyerson, and L. A. Mirny. “High order chromatin architecture shapes the landscape of chromosomal alterations in cancer.” In: *Nature biotechnology* 29.12 (2011), pp. 1109–1113 (cit. on p. 7).

[6] M. J. Fullwood, M. H. Liu, Y. F. Pan, J. Liu, H. Xu, Y. B. Mohamed, Y. L. Orlov, S. Velkov, A. Ho, P. H. Mei, et al. “An oestrogen-receptor-bound human chromatin interactome.” In: *Nature* 462.7269 (2009), pp. 58–64 (cit. on p. 7).

[8] J. Goecks, A. Nekrutenko, J. Taylor, and T. G. Team. “Galaxy: a comprehensive approach for supporting accessible, reproducible, and transparent computational research in the life sciences.” In: *Genome Biology* 11.8 (2010), R86 (cit. on p. 14).

[10] I. Gribkovskaia, O. Halskau, and G. Laporte. “The bridge of Konigsberg - A historical perspective.” In: *Networks* 49.3 (2007), pp. 199–203 (cit. on p. 9).

[14] E. Lieberman-Aiden, N. L. van Berkum, L. Williams, M. Imakaev, T. Ragoczy, A. Telling, I. Amit, B. R. Lajoie, P. J. Sabo, M. O. Dorschner, R. Sandstrom, B. Bernstein, M. A. Bender, M. Groudine, A. Gnirke, J. Stamatoyannopoulos, L. A. Mirny, E. S. Lander, and J. Dekker. “Comprehensive Mapping of Long-Range Interactions Reveals Folding Principles of the Human Genome.” In: *Science* 326.5950 (2009), pp. 289–293 (cit. on p. 7).

[21] J. Paulsen, T. G. Lien, G. K. Sandve, L. Holden, Ø. Borgan, I. K. Glad, and E. Hovig. “Handling realistic assumptions in hypothesis testing of 3D co-localization of genomic elements.” In: *Nucleic Acids Research* (2013) (cit. on p. 18).

[25] G. Sandve, S. Gundersen, H. Rydbeck, I. Glad, L. Holden, M. Holden, K. Liestol, T. Clancy, E. Ferkingstad, M. Johansen, V. Nygaard, E. Tostesen, A. Frigessi, and E. Hovig. “The Genomic HyperBrowser: inferential genomics at the sequence level.” In: *Genome Biology* 11.12 (2010), R121 (cit. on p. 14).

[32] S. van der Walt, S. Colbert, and G. Varoquaux. “The NumPy Array: A Structure for Efficient Numerical Computation.” In: *Computing in Science Engineering* 13.2 (Mar. 2011), pp. 22–30 (cit. on p. 16).

Other written references

[1] K. A. Berman and J. L. Paul. *Algorithms: Sequential, Parallel and Distributed*. Thomson Course Technology, 2005. Chap. 11. ISBN: 0-534-42057-5 (cit. on p. 11).

[2] R. Bryant, R. Bryant, and D. O'Hallaron. *Computer Systems: A Programmer's Perspective*. Pearson, 2010. ISBN: 0-13-713336-7 (cit. on pp. 46, 49).

[3] T. H. Cormen, C. E. Leiserson, and C. S. Ronald L. Rivest. *Introduction to Algorithms*. third. The MIT Press, 2009. ISBN: 978-0-262-03384-8 (cit. on p. 11).

[4] R. Diestel. *Graph Theory*. Springer, 1997. ISBN: 0-387-98210-8 (cit. on p. 10).

[7] H. Garcia-Molina, J. D. Ullman, and J. Widom. *Database Systems: the complete book*. second. Pearson Education International, 2009. ISBN: 0-13-135428-0 (cit. on p. 46).

Online references

[9] *Gremlin: a Graph Traversal Language*. Apr. 2013. URL: https://github.com/tinkerpop/gremlin/wiki (cit. on p. 54).

[11] *Groovy: a Dynamic Language for the Java Platform*. Apr. 2013. URL: http://groovy.codehaus.org/ (cit. on p. 55).

[12] *GTrack - a practical unification of tabular file formats*. Apr. 2013. URL: http://www.gtrack.no (cit. on p. 8).

[13] *JPype: Bridging the Worlds of Java and Python*. Apr. 2013. URL: http://jpype.sourceforge.net/ (cit. on p. 52).

[15] *Neo4j: High Availability*. Apr. 2013. URL: http://docs.neo4j.org/chunked/stable/ha.html (cit. on p. 58).

[16] *Neo4j: Pragmatic Licensing Guide*. Apr. 2013. URL: http://www.neo4j.org/learn/licensing (cit. on p. 51).

[17] *Neo4j: The World's Leading Graph Database*. Apr. 2013. URL: http://www.neo4j.org/ (cit. on p. 51).

[18] *NumPy Online Reference: Arrays*. Apr. 2013. URL: http://docs.scipy.org/doc/numpy/reference/arrays.html (cit. on p. 33).

[19] *NumPy Online Reference: numpy.memmap*. Apr. 2013. URL: http://docs.scipy.org/doc/numpy/reference/generated/numpy.memmap.html (cit. on p. 18).

[20] *NumPy Online Reference: numpy.ndarray*. Apr. 2013. URL: http://docs.scipy.org/doc/numpy-1.7.0/reference/generated/numpy.ndarray.html (cit. on p. 36).

[22] *PEP20: The Zen of Python*. Apr. 2013. URL: http://www.python.org/dev/peps/pep-0020/ (cit. on p. 36).

[23] *PyTables: Getting the Most out of Your Data*. Apr. 2013. URL: http://www.pytables.org/ (cit. on p. 50).

[24] *Python bindings for the embedded version of the Neo4j graph database*. Apr. 2013. URL: https://pypi.python.org/pypi/neo4j-embedded (cit. on p. 52).

[26] *The Galaxy Project*. Apr. 2013. URL: http://galaxyproject.org/ (cit. on p. 14).

[27] *The Genomic HyperBrowser*. Apr. 2013. URL: http://hyperbrowser.uio.no/hb/ (cit. on p. 14).

[28] *The Neo4j Manual: Chapter 11.5, Capacity*. Apr. 2013. URL: http://docs.neo4j.org/chunked/stable/capabilities-capacity.html (cit. on p. 58).

[29] *The Neo4j Manual: Part III, Cypher Query Language*. Apr. 2013. URL: http://docs.neo4j.org/chunked/milestone/cypher-query-lang.html (cit. on p. 53).

[30] *The Python Programming Language*. Apr. 2013. URL: http://www.python.org/ (cit. on p. 4).

[31] *The SciPy Library*. Apr. 2013. URL: http://www.scipy.org/ (cit. on p. 4).

[33] *Wikipedia: SPARQL*. Apr. 2013. URL: http://en.wikipedia.org/wiki/SPARQL (cit. on p. 54).

[34] *Wikipedia: SQL*. Apr. 2013. URL: http://en.wikipedia.org/wiki/SQL (cit. on p. 54).