M.Sc. Thesis
Master of Science in Engineering

DTU Compute
Department of Applied Mathematics and Computer Science

PECA: Proxy-based Efficient Clustering Algorithm

Enabling Bioinformatic Data Treatment in the Cloud

Tobias Bertelsen (s093267)

Kongens Lyngby 2015

DTU Compute
Department of Applied Mathematics and Computer Science
Technical University of Denmark

Matematiktorvet
Building 303B
2800 Kongens Lyngby, Denmark
Phone +45 4525 3031
compute@compute.dtu.dk
www.compute.dtu.dk

Abstract

This Master's thesis presents an algorithm for metagenomic gene clustering: *Proxy-based Efficient Clustering Algorithm* (PECA). It is designed for modern cloud computing frameworks and implemented using Apache Spark. It performs 40 times faster than UClust and 1.7 times faster than CD-Hit, running on four machines instead of one. Additional speedup can be gained by increasing the number of machines.

Further algorithmic challenges and implementation improvements is described. PECA is expected to clearly outperform CD-Hit when these are solved.

This thesis is also present the following independent scientific contributions, which lay the foundation for PECA.

- The *Dimension Independent Similarity Computation* (DISCO) algorithm [ZG13] is generalized, so it can accept real valued vectors instead of just binary vectors. Furthermore it can use a whole class of similarity measures.
- The cosine similarity and Pearson correlation between two k-mer profiles are found to be better predictors of gene identity, compared with the count of common k-mers used by UClust [Edg10].
- *Information Inspired Quality* (IIQ) is suggested as a new measure of cluster quality that is suitable for a large number of clusters.

Preface

This Master's thesis was prepared at the Department of Applied Mathematics and Computer Science at the Technical University of Denmark in fulfillment of the requirements for acquiring a Master's degree in Science and Engineering. It was tutored by associate professor Inge Li Gørtz and associate professor Philip Bille.

It was prepared in association with associate professor Manimozhiyan Arumugam's group from the Section for Metabolic Genetics at the Novo Nordisk Foundation Center for Basic Metabolic Research at the University of Copenhagen.

The project plan and auto-evaluation can be found in Appendix C as required by the study handbook of DTU.

Prerequisites

This thesis is intended for Master's students in computer science with a strong background in advance algorithms. The reader is expected to have a good understanding of basic statistics and probability theory.

It is *not* expected that the reader has any knowledge about biology, bioinformatics, or metagenomics. The required knowledge and technology will be introduced in this thesis.

Acknowledgements

The deepest and most sincere thanks go to my father Tue Bertelsen and my great friend Kasper Laursen. I was struck by a severe case of repetitive strain injury while working on this thesis. For several months I was not able to type anything on a computer. Tue and Kasper both spend long days helping me, being my hands when I needed to solve tasks that cannot be solved using dictation software. In addition to that they also helped proofreading the final product.

My dear friend Christian Kalhauge and my mother Kirsten Ølgaard also deserves my thanks for supporting me through this work and helping proofreading this thesis.

Finally I will like to thank my tutors Inge Li Gørtz, Philip Bille, and Manimozhiyan Arumugam. They have all been extremely helpful giving me scientific sparring and guidance.

Kongens Lyngby, October 11, 2015

Tobias Bertelsen (s093267)

Contents

CHAPTER 1

Introduction

Large-scale data analysis and simulations are becoming an increasingly important part of many scientific fields. A field completely dominated by data analysis and computations is bioinformatics. This field has experienced fast growth after the invention of next-generation sequencing technologies, which have dramatically reduced the cost of reading DNA. A dramatic increase in the amount of available DNA data has followed, and that has become a problem for the current methods for analyzing data.

Cloud computing provides an attractive solution to this problem, but there are still some challenges to overcome. One of these is developing bioinformatic software and algorithms that are suitable for the cloud. This thesis presents an efficient algorithm called Proxy-based Efficient Clustering Algorithm (PECA) to solve the bioinformatic problem of gene clustering.

1.1 The Case for Cloud Computing

The falling price of DNA sequencing opens up opportunities for more research with more data. In most areas the effect of Moore's law enables computers to keep up with new requirements, but that is not the case with bioinformatics. The data sizes are growing much faster than computation power and storage capacity [Ste+10]. Researchers are therefore forced to perform computations on multiple machines to keep up with the increasing data sizes. Cloud computing has been suggested as a better way to do this compared with traditional on-site computer clusters [Ste+10; SLS10].

The fast increasing data sizes are not the only reason why cloud computing is a good match for the scientific community. Scientists has extremely varying demands for computation power. Most of the time they do not need any more than their desktop, but occasionally they need to run big simulations or data analyses. This usage pattern is a perfect fit for the elasticity of cloud computing.

Universities have two options to deal with varying demand if investing in on site computation resources. They can invest in resources that can deal with demand levels close to the peak. This will mean researchers have the resources they need when they need them, but it also entails that the resources have a low utilization level. Low utilization is essentially a waste of money. The other option is to invest in resources at a level closer to the average demand. Researchers will then have to wait to get access to computational resources, but that enables the University to spread out demand and increase utilization.

Waiting a long time for computing resources is a problem for research. Data analysis is an iterative process. One determines a question, run an analysis to answer the question, and uses that knowledge to ask new and better questions. This process gets very inefficient when a researcher have to wait days for the answer instead of getting the answer after a short coffee break.

Cloud computing enables instant allocation of as many resources you need, for exactly as long as you need. There is therefore no trade-off between utilization and waiting time. Researchers can get fast answers without the university wasting money on expensive hardware that are often unused.

Implementing a efficient distributed algorithm has previously been a complicated task. Libraries like MPI provides low-level control and required significant computer science knowledge to use. The rise of cloud computing has brought along newer frameworks that allows programmers to program distributed algorithms on a much higher abstraction level. Such frameworks have already been applied successfully to some areas of bioinformatics. For example, the ADAM pipeline have reduced computation time in a sequencing pipeline from 17 hours 44 minutes to just 21 minutes [Mas+13].

1.2 Challenges Moving to Cloud Computing

The advantages presented in the previous section are naturally not the entire story. There are still some challenges and barriers to cross before moving to the cloud.

There exists an entire backlog of scientific software that were not designed for the cloud. Most of these tools cannot exploit the distributed nature of a cloud environment. Usually the algorithms and especially implementations are not parallelizable. There is thus a need for new massively parallel algorithms.

Developing these algorithms comes with a bootstrap problem. Most universities currently have expensive on-site computer clusters. Having made these investments they are reluctant to grant additional money for external computing resources. Most of the user base is therefore prevented from using cloud solutions.

Without users there is a limited motivation to develop algorithms and tools for cloud solutions, but as long as these are not developed, the universities are required to keep investing in on-site resources.

This circle seems to be breaking with recent developments. Open source projects like ADAM and commercial ones like Google Genetics [Mas+13; Pla15b] are making it more and more feasible for bioinformaticians to switch to cloud driven analytics, but there are still many problems not addressed by these tools. This thesis will contribute to the scientific community by developing an algorithm for one of these problems.

1.3 Focus of This Thesis

This thesis will solve a problem within metagenomics. While most of bioinformatics is the study of the genome of any one organism, metagenomics is the study of the genomes of multiple organisms from environmental samples.

Studies of the microbial cultures in the gut has found that certain microbes are related to obesity or increased chance of diabetes [Le +13; Qin+12]. A study of sewage water from airplanes have given us knowledge about how multi-resistant diseases spread internationally [Pet+15]. These are examples of metagenomic studies.

This thesis will work on data from the Human Microbiome Project (HMP) [Met+12]. It is a dataset containing microbial genes from 253 human fecal samples. For most genes it is unknown which organism they belong to, since we only know a fraction of the organisms in the human gut.

This thesis will solve the computational problem, called *gene clustering*, that is encountered when trying to identify which organisms that exist in the human gut. The goal of metagenomic gene clustering is to identify the same genes across different samples. It will be explained in more detail in Chapter 2.

1.4 Contributions

This thesis will help the scientific community by developing a cloud ready algorithm to solve the problem of metagenomic gene clustering. It is a step on the way to making cloud computing truly attractive for scientists.

Proxy-based Efficient Clustering Algorithm (PECA) is presented and is shown to have competitive performance when distributed on multiple machines. Several areas of improvement has been identified; some simple, some complex to solve. If all these improvements are solved and implemented, PECA is expected to also outperform existing tools on identical hardware.

Several contributions of independent value has been made as part of this work.

The algorithm for Dimension Independent Similarity Computation (DISCO) [ZG13] has been generalized from working only on binary vectors to vectors with real values.

It has been shown that the identity of two genes can be proxied by the Pearson correlation as well as cosine similarity between the k-mer profiles of those two genes.

A new measure for cluster quality, Information Inspired Quality (IIQ), has been suggested. It is efficient to calculate even on huge datasets with a large number of clusters, as opposed to existing measures.

A new algorithmic problem, proxy-based center selection, has been defined. Solving that will lead to significantly faster and better scaling gene clustering.

The practical barriers to developing and using cloud-based algorithms and ease-of-use of modern cloud computation of frameworks has been evaluated.

A parallelizability defect in Apache Spark has been identified and reported.

1.5 Thesis Overview

The main purpose of this thesis is to present PECA. It builds on top of other knowledge which have to be presented first. Some of that will be existing knowledge; other will be from new research conducted as part of this thesis. Finally the algorithm is tested and compared with existing tools, and the results are discussed along with suggestions for further improvements.

Chapter 2 will introduce computer scientists to the most important terminology within the field of bioinformatics, and explain the metagenomic gene clustering problem. It will also describe the current state-of-the-art tools to solve this problem.

Chapter 3 contains a review of modern cloud computing frameworks, which leads to the decision to implement PECA using Apache Spark. The chapter ends with the presentation of a theoretical framework for evaluating algorithms for Apache Spark.

Chapter 4 reviews the most popular measures for cluster quality, and concludes that these are not suitable for metagenomic gene clustering. The novel measure Information Inspired Quality (IIQ) is suggested as a better quality measure for problems like gene clustering.

Chapter 5 presents novel research into proxies that efficiently can indicate whether the two genes are the same. It is inspired by a proxy used by UClust [Edg10], but finds that other measures are better at predicting whether two genes are the same.

Chapter 6 describes an efficient algorithm to find all pairs of genes that the proxy from Chapter 5 indicates might be the same. The algorithm is a generalized version of the DISCO algorithm [ZG13].

Chapter 7 introduces PECA. It uses the output from the algorithm presented in Chapter 6 as input to a greedy clustering algorithm that strives to optimize the IIQ. It also presents three variations of PECA that aims at optimizing accuracy or efficiency.

Chapter 8 evaluates PECA. The tuning parameters for PECA are optimized, and the variations are compared. PECA is compared with the existing state-of-the-art software, including a comparison of their IIQ. Finally the scalability and parallelizability of PECA is evaluated.

Chapter 9 analyzes and discusses the results found in Chapter 8, along with suggestions on how the performance of PECA can be improved.

Finally Chapter 10 sums up the conclusions of this thesis.

CHAPTER 2

Introduction to Metagenomic Gene Clustering

This chapter is meant to introduce computer scientists to the bioinformatics concepts and terms. These terms are required to understand the problems and solutions presented in this thesis. A quick overview is presented in Listing 2.1. If you are already familiar with bioinformatics, you can skip ahead and go directly to the definition of metagenomic gene clustering in Definition 2.1.

The last part of this chapter will describe the current state-of-the-art tools for doing metagenomic gene clustering.

2.1 DNA, Nucleotides and k-mers

The instructions on how to make life is written in deoxyribonucleic acid, more commonly known as DNA. All cells in all living organisms contains DNA which describes

Listing 2.1: Comparisons of bioinformatic and computer science terms.

Bioinformatics	Computers Science	Notes
Nucleotide / Basepair	Character	
k-mer	n-gram	
DNA	String	Any length of text
Gene	String	A chapter of a book
Genome	String	An entire book
Metagenome	Set of Strings	A library of books
Identity	$1 - \frac{\text{edit distance}}{\text{length}}$	Computationally harder than edit distance

how the entire organism works. DNA is made out of two strands of nucleotides which are bound together.

There are four types of nucleotides: adenine, cytosine, guanine, and thymine. They are normally just denoted with their first letter `A`, `C`, `G`, and `T`. Each nucleotide in the one strand of DNA is paired with one nucleotide in the other strand. This is called a base pair. `C` always pairs with `G` and `A` always pairs with `T`, so if we know one strand we can infer the other. Because of this a piece of DNA is only represented by a string containing one of the strands (see Listing 2.2)

k-mers are just a sequence off k nucleotides, for example, `ACC` or `AGT` are 3-mers and `AATTC` is a 5-mer.

2.2 Genes, Genomes, and Metagenomes

Within an organism, DNA is organized into genes. Each gene contains the instruction for one or more of an organism's specific functions. In general there are no limits on the length of the gene that applies to all kinds of organisms. Some has as few as 30 base pairs; others as many as 2,000,000. But we know that bacterial genes, which is the focus of this thesis, typically contains less than 10,000 base pairs.

A gene has a direction and the end and beginning is marked by specific DNA sequences. Each strand of nucleotides also have a direction defined by its chemical structure. The two strands go in opposite directions. This gives us the opportunity to find a unique representation of the gene, so we do not need to switch direction or translate between the two strands when comparing two genes.

The collection of all genes for a specific organism is called the genome. That is, the genome of an organism is all and any of the DNA required to make that organism.

We can use text as a metaphor for DNA. In this metaphor a single character corresponds to a base pair and DNA is just any amount of text. A genome will then be a book containing an entire complete story, while each gene will be a chapter in the book each telling one part of the story.

In nature we often find microbial environments whose function is not defined by a single organism, but by a collection of many different microorganisms. Such environments are typically viewed as a whole, for example a microbial culture in a

Listing 2.2: A short example of the two strands of DNA and its representation.

```
Strand1:        AATGGCTG
Strand2:        TTACCGAC

Representation: AATGGCTG
```

swamp or the bacteria in the human gut. In this case the environment cannot be described by a single genome, but must be described by the set of all genomes of organisms in the environment. This collection of genomes or "genome of genomes" is called the metagenome of the environment. Typically we are interested in not only the set of genomes, but also their abundance in the environment.

In the text metaphor the metagenome will be like a library or another collection of books. Each book tell their individual story, but as a whole the library tells us about our culture.

2.3 Identity and DNA Alignment

When cells split and organisms multiply, tiny mutations in their DNA can occur. The same gene from different members of the same bacterial species will therefore be slightly different. It is therefore not possible to test if two genes are the same by testing if their DNA is 100% identical. We need a way to compare them that allows for minor mutations.

Two different types of variations can occur. A single base pair can be replaced by another base pair, or a series of base pairs can be inserted or deleted. There is generally no distinction between insertion and deletion: if a subsequence is missing from gene A, it could be either due to a deletion in gene A or an insertion in gene B. Collectively deletions and insertions are called *gaps*.

Figuring out which mutations would be the most likely to transform gene A to gene B is called alignment, because it is identical to align the parts of the genes that have not been changed. An example of a alignment is shown in Listing 2.3.

2.3.1 Computing DNA Alignment

Comparing two strings with the opportunity of characters being replaced, inserted, or deleted sounds a lot like the computer science problem of finding the minimal edit distance. The theory of the distance can however not be applied to gene alignment.

When calculating edit distance a replacement, insertion, or deletion of a single character all have the same cost. That is not the case in biology. In biology replacements are more common than insertions and deletions. The longer the insertion or

Listing 2.3: An example of two aligned genes. Matching base pairs are noted by `|`.

```
Gene1: TTACCGTACCGGGTACGGTTAAGTTACGTACCTAACGTTTACCCG---CGTTTACCCCCTT
         |   |||  ||||   |||| |    ||| || |||||||| |   ||||||||||  |
Gene2: --ATTCTACGCGGTA---TTAACT--ACTACTTACCGTTTACCGGTTACGTTTACCCC--T
```

deletion is the less likely it will be. This is typically represented by three different penalties for mutations.

The cost for single base pair replacement is called the *mismatch penalty*. Longer gaps are less likely than shorter gaps. Gaps are therefore penalized with a constant *gap open penalty*, plus a *gap extension penalty* for each base pair in the gap: $penalty_{gap} = penalty_{open} + l \cdot penalty_{extend}$, where l is the length of the gap. If $penalty_{extend}$ is equal to $penalty_{mismatch}$ and $penalty_{open}$ is zero then the problem of gene alignment will be equal to edit distance.

There is no consensus on how exactly to penalize alignments. Most agree on a match score, and penalties for mismatch, gap opening, and gap extension, but each software tool have its own weighting. As gaps are less likely to occur than substitutions, there is often this relationship between the alignments [Edg15b].

$$penalty_{extend} \leq penalty_{mismatch} \leq penalty_{open}$$

There are two general forms of alignment: global and local. The former includes all of both genes that are aligned, while the latter uses an uninterrupted subsequence from each gene. These alignments are calculated using dynamic programming algorithms called the Needleman–Wunsch algorithm and Smith–Waterman algorithm [NW70; SW81].

Performing alignment with these algorithms is an expensive operation, and they scale quadratically in the length of the genes. Bioinformatic tools, such as UClust [Edg10], CD-Hit [LG06], or Blast [Alt+90], will therefore use heuristics to reduce the number of alignments performed.

2.3.2 Gene Identity

When two genes are aligned we can decide whether we consider them to be the same or not. This is typically done using a metric called gene identity, which is the fraction of base pairs that are identical in an alignment of two genes. If that fraction is above a certain threshold we consider the two genes to be the same. We will denote that threshold $id_{\min}$. There are different definitions of how exactly to calculate the gene identity [Edg15c].

It might seem a little unusual that the term gene identity is used, when the genes are not identical in the sense of string identity in computer science. In computer science identity is often used as a boolean relationship, not as a relative relationship like gene identity. The reason for the term is that similarity has a distinct and different meaning in bioinformatics. It is primarily used when aligning protein sequences instead of DNA. For that reason it would be ambiguous to use similarity instead of gene identity.

It is important to note that the gene identity depends on the alignment. The alignment process tries to construct an alignment that makes the most biological sense, not necessarily the one that maximizes gene identity. In addition the alignment depends on the penalty parameters.

Different tools will often disagree on the gene identity, since they use different penalty parameters as well as a different definition of gene identity. Surprisingly, despite these differences it is commonly accepted that the threshold $id_{\min}$ should be 95%, when deciding whether two genes are actually the same gene from different members of the same bacterial species.

This thesis will use penalty parameters and a definition of gene identity similar to those used by UClust.

2.4 Metagenomic Gene Clustering

One of the main goals of metagenomic research is to identify the effects of microbes in the microbial environment. In the human gut, for example, you might find microbes that are linked to a better metabolism or an increased risk of diabetes [Le +13; Qin+12]. To do that one needs to know the abundance of different microbes across multiple metagenomic samples. This process has roughly three steps:

1. Read the genes present in each sample
2. Identify which genes across different samples are the same gene
3. Group the genes into organisms

In Chapter 7 an algorithm is presented called PECA for step number two and the remainder of this thesis will not deal with the two other steps.

The task of identifying the same gene across different samples is also called gene clustering, that is, we want to group all instances of a gene across multiple samples into one cluster. A formal definition of this problem is given in Definition 2.1.

Definition 2.1: Gene clustering.

Let G be a set of genes, and $id\colon G \times G \mapsto [0\%, 100\%]$ be the function describing gene identity, that is for $g, g' \in G$ has an identity percentage of $id(g, g')$

A *clustering* of G consists of a set of centers $C \subseteq G$ and an assignment $\sigma\colon G \setminus C \mapsto C$ of all non-center genes to a center. Any clustering may have one or more of these properties.

Independent centers No two centers are neighbors,
that is $\forall c, c' \in C : \quad c \neq c' \Rightarrow id(c, c') < id_{\min}$.

Dominant centers All non-center genes are neighbors to at least one center,
that is $\forall g \in G \setminus C : \exists c \in C : \quad id(g, c) \geq id_{\min}$.

Valid assignment All non-center genes are assigned to a neighboring center,
that is $\forall g \in G \setminus C : \quad id(g, \sigma(g)) \geq id_{\min}$.
A valid assignment exists if and only if the set of centers is dominant.

Optimal assignment All non-center genes are assigned to the closest center, that is $\forall g \in G \setminus C : \forall c \in C : \quad id(g, \sigma(g)) \geq id(g, c)$.

A *valid clustering* is a clustering that has independent centers and a valid assignment, but not necessarily an optimal assignment.

A *valid set of centers* is a set of centers that is independent and dominant.

2.5 Existing Tools

Currently there exist two major tools that can cluster genes: CD-Hit [LG06] and UClust [Edg10]. They have a similar approach to cluster genes which is shown in Algorithm 2.1. They both use an iterative approach handling genes in order of decreasing length while maintaining a set of center genes. For each gene they try to find an existing center that is identical enough to the gene. If such can be found the gene is assigned to the center, otherwise the gene is made a new center.

Algorithm 2.1: `GeneClustering(`G`)` as done by CD-Hit and UClust.

```
1: Sort G by length
2: C ← ∅                                          ▷ The set of centers
3: for all g ∈ G do
4:     c ← FindCenter(g, C)
5:     if c ≠ null then
6:         Assign g to c
7:     else
8:         Put g in C
9:     end if
10: end for
```

The difference between the two algorithms lies in `FindCenter(`g`, `C`)`.

CD-Hit goes through every center in C. It reduces the number of alignments by eliminating center candidates if possible using statistics on their common k-mers. It can either return the first center it finds or continue to see if it can find a center with a higher identity.

CD-Hit guarantees independent and dominant centers. It also guarantees an optimal assignment unless it returns early in `FindCenter`.

UClust first find a small set of center candidates. It does so by performing a lookup using a proxy for gene identity that will be explained in more detail in Chapter 5. This is similar to a k-nearest neighbors search. It only performs alignments on this small set of center candidates. If one is found that is returned, otherwise *null* is returned.

Independent centers and an optimal assignment is not guaranteed, but its implementation of `FindCenter` has a high probability of finding centers that fulfills those properties. UClust only guarantees dominant centers and a valid assignment.

UClust does not support any kind of parallelization. CD-Hit can run multithreaded on a single machine. It also has support for distribution to multiple machines by splitting the computation up into multiple jobs. That requires a shared file system mounted on all machines. The distribution is done either using a grid engine or by connecting directly through SSH.

2.5.1 Existing Tools and the Cloud

The case for moving to cloud computing when performing bioinformatic data analysis was justified in Chapter 1. The problem is that neither UClust nor CD-Hit are very suitable for modern cloud computing.

UClust becomes practically useless due to its lack of parallelization. CD-Hit has more potential since it can be multithreaded and distributed across multiple machines. There are however still some problems with CD-Hit.

The distribution method requires all servers to receive the data for all center genes. It depends heavily on having a single, shared filesystem available on all servers. This file system could easily become a bottleneck as more and more servers request data.

CD-Hit was not designed with the cloud in mind. It cannot deal with distributed filesystems such as Hadoop's HDFS, Amazon's S3, or Google's Cloud Storage. Likewise it cannot run in modern distribution frameworks, such as Yarn, Mesos, or Docker. It can therefore not be integrated in a modern cloud infrastructure or use any of the managed services available.

New algorithms and tools will therefore need to be developed if the bioinformatic community needs to fully reap the benefits of cloud computing.

2.6 Summary

In this chapter we got a high-level introduction to bioinformatics in order to understand the problem of metagenomic gene clustering. The problem was more formally defined and properties of a clustering were also defined.

In the end we looked at existing tools, how they work, and why they are not suitable for a modern cloud environment.

CHAPTER 3

Cloud Computation Frameworks

This chapter starts with a review of modern cloud computing frameworks, namely Apache Spark and Google Cloud Dataflow. The second part of the chapter will present a theoretical methodology for analyzing algorithms written in such cloud computing frameworks.

3.1 Review of Modern Cloud Computing Frameworks

During the last decade the MapReduce programming model [DG08] has become the dominant way to do distributed data analysis in the cloud, primarily driven by the Hadoop implementation [Fou14]. It provides a much higher level of abstraction than previous distribution libraries such as MPI, and on top of that it provides redundancy and handles fault tolerance.

The MapReduce programming model is however overly restrictive. It has become obvious, as its popularity has increased, that it is better suited for some problems than others. There is a need for more flexible programming model, that is suitable for wider range of problems while still providing the same high level of abstraction, robustness, and fault tolerance.

Two new major initiatives aims to satisfy that need: Apache Spark [Zah+10] and Google Cloud Dataflow [Pla15a].

Apache Spark is based on research from Berkeley University on resilient distributed datasets [Zah+12] and is completely open source. It can run in existing Hadoop clusters, Apache Mesos clusters, and in standalone mode. Contrary to Hadoop it aims to keep the data in RAM instead of on disk. It achieves fault tolerance by remembering how each data set is computed and allowing for redundant caching of datasets. The data stored on a machine can then be recalculated, if the machine is lost. This approach have shown to provide significantly increased performance [Xin15].

Apache Spark allows the user to manipulate the data using a wide range of higher order functions familiar to those who have used functional programming. This enables the user to write clear, succinct programs, with much less boilerplate code. See Code Example 3.1. Several libraries have been developed on top of Spark. These include libraries for machine learning, SQL, graphs and more.

Code Example 3.1: Word count algorithm in Apache Spark [Fou15c].

```
val textFile = spark.textFile("hdfs://...")
val counts = textFile.flatMap(line => line.split(" "))
                 .map(word => (word, 1))
                 .reduceByKey(_ + _)
                 .map(tuple => tuple._1 + ": " + tuple._2)
counts.saveAsTextFile("hdfs://...")
```

Code Example 3.2: Word count algorithm in Google Cloud Dataflow [Pla15c].

```
p.apply(TextIO.Read.from("gs://dataflow-samples/shakespeare/kinglear.txt"))
  .apply(ParDo.named("ExtractWords").of(new DoFn<String, String>() {
    private static final long serialVersionUID = 0;
    @Override
    public void processElement(ProcessContext c) {
      for (String word : c.element().split("[^a-zA-Z']+")) {
        if (!word.isEmpty()) {
          c.output(word);
        }
      }
    }
  }))
  .apply(Count.perElement())
  .apply(ParDo.named("FormatResults").of(new DoFn<KV<String, Long>, String>() {
    private static final long serialVersionUID = 0;
    @Override
    public void processElement(ProcessContext c) {
      c.output(c.element().getKey() + ": " + c.element().getValue());
    }
  }))
  .apply(TextIO.Write.to("gs://my-bucket/counts.txt"));
```

Google Cloud Dataflow is, as the name suggests, developed by Google who also invented the MapReduce programming model. It became generally available to the public in April 2015 [Cza15]. It is primarily a commercial product marketed as a Platform as a Service (PaaS). On top of fault tolerance it promises automatic scaling, automatic optimization and tuning, plus easy monitoring and debugging tools. They have also open sourced the SDK allowing others to implement the execution environment. Currently such exists for Apache Flink [Art15] and Apache Spark [Clo15], but these do not provide the same level of automatic scaling, optimization, and tuning.

Conceptually Apache Spark and Google Cloud Dataflow are very similar. They both build on the concept of distributed datasets which can be transformed in different ways. Google Cloud Dataflow is however based on Java 7 and does not use higher

order functions. The result is that the code contains a large amount of boilerplate code. This is clearly shown in Code Example 3.2. This difference is even more significant considering that the `reducedByKey` and the first `map` operation from the Spark example is supplied by the platform in `Count.perElement()`.

Apache Spark has been chosen as a platform for this thesis because of several reasons. First of all Google Cloud Dataflow was not publicly available when the work on this thesis began. Secondly Spark has a much cleaner API and libraries for graph computation, which will be useful when we need to do clustering. Lastly Spark is open source, free, and can be executed on already owned hardware without the need to pay for usage. This makes it more accessible for scientific use.

The remainder of this chapter will introduce Apache Spark in more detail, and it will suggest a theoretical framework for evaluation of algorithms designed for Apache Spark.

3.2 Introduction to Apache Spark

The website for Apache Spark contains excellent introductory material [Fou15e]. While understanding Apache Spark is important for understanding this thesis, a description of it has no scientific value.

If you have no experience with Apache Spark please read Appendix A before continuing with this chapter. It contains excerpts of the official documentation that are relevant for this thesis. If you already know about Apache Spark or you do not want to read the documentation, please feel free to continue to the next section.

3.3 Evaluation of Spark Algorithms

There is no established way to theoretically evaluate Spark algorithms, since the framework is very new. This section will therefore propose a simple analytic methodology that can be used to analyze Spark algorithms, including the one presented in this thesis. It will be based on the methodology used to analyze algorithms for MapReduce [GM12; ZG13].

Three main complexity measures need to be analyzed for each step in a Spark algorithm. These are:

Definition 3.1: Most important complexity measures for analyzing algorithms for Apache Spark.

Total computation time: The total time spent to transform data across all threads on all machines.

Strangler computation time: The time spent to transform the slowest element.

Shuffle size: The amount of data that gets sent across the network, which is often a bottleneck.

The total computation time is naturally interesting as it describes how many resources that will be needed to complete the computation. In an optimal world this can be spread evenly among any numbers of machines, so if we double number of machines we will get the results twice as fast. But reality is not the optimal world, which leads us to the strangler.

The slowest element is often known as a strangler, because if a few elements will take much longer than average to compute then most of the system might idle while it waits for those to complete. This strangles performance. The strangler computation time thus indicates how well the problem parallelizes.

Sometimes the strangler refers to the slowest partition. This makes sense from practical point of view, since a partition is the smallest data chunk operated on. The effects of a strangler partition can however often be mitigated by reducing the partition size. The extreme case of this is having only a single element per partition. From a theoretical standpoint we are therefore interested in the strangler element, whose effects cannot be mitigated.

Shuffles are expensive operations. They require worker nodes to be in sync with one another and sends large amount of data over the network. Even though network speeds have improved greatly during the last decade, they are still much slower than for example RAM access speeds. Consequently shuffles are still often the bottleneck of distributed algorithms.

In addition shuffle times can sometimes not be mitigated by adding more servers. Having 100 servers with 10 Gb/s network connections does not do you much good if your network infrastructure can only handle 100 Gb/s total.

3.3.1 Example Analysis

We will now go through an example, to help understand the analysis process better. The example will be the word count algorithm presented in Code Example 3.1. We will ignore time spent to load and save the data.

The first step is the `flatmap` transformation. Each call to `split` takes $O(|s|)$ time, where $|s|$ is the size of the string. The total computation time is therefore $O(M)$ where M is the total number of characters in the input data. The strangler will be the longest line L, that is, $O(|L|)$. Depending on the data this can be good or bad. In most cases the input data will have broken long lines in which case there will be no problems with stranglers. But let us imagine that the input file contains all of Shakespeare's Hamlet in a single line, then strangling will be a significant problem.

The next transformation is a `map`. This just creates a new object so all invocations will take constant time $O(1)$. Neither `map` nor `flatmap` will do any shuffling.

The last transformation, `reduceByKey`, is a little more interesting. Each worker node will perform as many reduces as possible, before shuffling the results so the global results can be collected. In this way there will only be shuffled one element per unique word per worker, instead of shuffling the entire data. The shuffle size is therefore $O(UW)$, where U is the number of unique words and W is the number of worker nodes.

The reduce operation is a simple integer addition, so it takes constant time and does not increase the size of the elements. The strangler time is therefore constant and the total time will be $O(N)$, where N is the total number of non-unique words, since there will be $N - U$ invocations of the function.

3.4 Summary

In this chapter we have looked at two new frameworks for implementing distributed algorithms: Google Cloud Dataflow and Apache Spark. Their programming model is very similar to each other, but their API and syntax differs a lot. Apache Spark, in addition to having the best syntax, is also open source and has libraries e.g. for graph computation. Apache Spark has thus been chosen as the framework for this thesis.

In order to work with algorithms for Apache Spark in a scientific context, an evaluation methodology has been proposed. It prescribes that three main measures must be analyzed: total computation time, strangler computation time, and shuffle size.

CHAPTER 4

Information Inspired Quality

This chapter will introduce a new measure that can be used to assess the quality of gene clustering. A new measure is needed because the existing commonly used ones neither scale well nor provides useful information for gene clustering. They are therefore infeasible to use when evaluating datasets with millions of genes.

The first part of this chapter will describe existing quality measures and explain why they do not scale well. The last part will introduce the *Information Inspired Quality* measure or IIQ.

4.1 Existing Cluster Quality Measures

In this section we will focus on three different quality measures for clusters: the Davies–Bouldin index, the Dunn index, and silhouettes, see Definitions 4.1 to 4.3. These definitions use the following notation:

Let E be the set of all elements. Let $d(e, e')$ be the distance between two elements $e, e' \in E$. Let a clustering be defined by its set of centers C, and for each center c its assigned members M_c. Let $A : E \mapsto C$ be the assignment function that takes any element and returns the center it is assigned to such that $A(m) = c \Leftrightarrow m \in M_c$.

Let S_c be the size of the cluster centered around c. This size can be defined in different ways depending on the application. In the case of gene clustering it will make most sense to use the average distance to the center.

$$S_c = \frac{1}{|M_c|} \sum_{m \in M_c} d(m, c)$$

Definition 4.1: Davies-Bouldin index [DB79; Wik15b].

The motivation behind the Davies-Bouldin index is that clusters should be as dissimilar as possible. It defines a similarity score between two clusters c, c' as $\frac{S_c + S_{c'}}{d(c,c')}$ and only considered the similarity with the nearest other cluster.

$$\mathrm{NC}(c) = \max_{c' \neq c} \frac{S_c + S_{c'}}{d(c, c')}$$

We then take the average across all clusters weighted by their number of members.

$$\mathrm{DBI} = \frac{1}{|E|} \sum_{e \in E} \mathrm{NC}(A(e))$$

Definition 4.2: Dunn index [PBM04; Wik15c].

The Dunn index is the smallest distance between clusters over the size of the largest cluster:

$$\mathrm{DI} = \frac{\min\limits_{c \in C} \min\limits_{c' \neq c} d(c, c')}{\max\limits_{c \in C} S_c}$$

Definition 4.3: Silhouettes [Rou87; Wik15e].

Let D_e be the distance from e to the closest cluster other than $A(e)$:

$$D_e = \min_{c' \neq A(e)} d(e, c')$$

We then define the silhouette to be[†]:

$$\mathrm{Sil} = \frac{1}{|E|} \sum_{e \in E} \frac{D_e - d(e, A(e))}{\max\left(D_e, d(e, A(e))\right)}$$

[†]This definition is slightly modified to be suitable for center-based gene clustering. The original version uses distance between an element and all cluster members instead of the distance between an element and the cluster center.

All three quality measures suffers from two major problems. They do not scale well to a large number of clusters, and they are not very suitable for problems with requirements on the similarity between the center of a cluster and its members, such as gene clustering. We will deal with the latter first.

4.1.1 Issues With Distance and Identity Requirement

Clustering of genes is different from other common clustering methods such as k-means. Normally a natural distance measure is defined between all elements. An element should be assigned to the cluster that is closest to it, but there are no maximum distance between an element and its cluster center. A distance measure is

therefore a good choice when comparing an element with any cluster, both the one it is assigned to and other clusters.

Clustering genes is a little different A gene can not be assigned to any cluster, since some cluster assignments are invalid according to Definition 2.1. It would be natural to use gene identity as basis for the distance between a gene and a valid center for that gene. We would like the gene to be assigned to the center that it has the highest identity with. But how do we define the distance to centers that will be an invalid assignment?

It is practically impossible to define a real valued distance measure that correctly captures the boolean concept of a valid assignment. Assume we define a linear distance measure, e.g., $d(e, e') = 1 - \frac{id(e,e')}{100\%}$.

Now assume we have two cluster centers c^-, c^+ and an element e such that $d(e, c^-) = id_{\min} - \epsilon$ and $d(e, c^+) = id_{\min} + \epsilon$. Using the linear distance measure says that $d(e, c^-)$ is almost as god as $d(e, c^+)$ and thus indicating that e can be assigned to either cluster c^- or cluster c^+. This is not true since assigning e to c^- is not a valid assignment, since $d(e, c^-) < id_{\min}$.

Now have c^{--} such that $d(e, c^{--}) = id_{\min} - 50\%$. The linear distance measure suggests that the c^{--} will be a much worse fit to e than c^-, since it gets a higher distance. In reality c^{--} and c^- are "equally bad", since they both are invalid assignments.

If the distance measure indicates that some invalid centers are "better" than others by giving them as shorter distance, then the quality measures might give a misleading indication of the quality of the clustering.

We could define the distance to be a large constant d_{invalid}, whenever the identity is below the threshold $id_{\min}$. In this case the distance between centers will be constant $d(c_i, c_j) = d_{\text{invalid}}$, assuming that the centers are independent. The Dunn index and the Davies-Bouldin index depend a lot on the smallest distance between clusters. Hence they are not well suited for distance measure that define all inter-cluster distances to be a constant.

The Dunn index will just be the ratio between d_{invalid} and the cluster with the largest size. This is not very informative. The Davies–Bouldin index will be reduced to the sum of the size of the largest cluster and the average cluster size. That will reward singleton clusters which is a sign of a bad clustering.

Silhouettes are not inhibited as much by setting a constant distance for genes with an identity below $id_{\min}$. They will still capture some information of how close cluster members are to the center. On the negative side they will give singleton clusters a good score, which is contrary to what we want.

In conclusion the three quality measures in Definitions 4.1 to 4.3 are mostly suited for clustering tasks in Euclidean space with a fixed number of clusters. They all reward singleton clusters and cannot handle clusters defined by similarity limits.

4.1.2 Scalability Problems

Apart from not being very informative the three quality measures also presents a computational challenge when the number of clusters becomes large. This is made worse by the fact that comparing two genes by aligning them is a very expensive operation compared to other distance measures.

The Davies–Bouldin index requires us to calculate $\frac{S_i+S_j}{d(c_i,c_j)}$ for every pair of clusters. This will take $O(Q^2)$ time, where $Q = |C|$ is the number of clusters. There is no obvious way to optimize this calculation.

The Dunn index and silhouettes are also slow to calculate naïvely, taking $O(Q^2)$ and $O(NQ)$ time respectively. It might however be possible to optimize these calculations. Chapter 7 explains how we will construct the graph where all pairs of genes are connected if they have a identity above $id_{\min}$. Computing the Dunn index or silhouettes would be much more efficient if you have such a graph and a constant distance measure between genes with an identity below $id_{\min}$. The constant distance measure is however a bad idea as explained above. Secondly we would only be able to compute the quality measure if we have such a graph.

It would require an advanced algorithm just to efficiently calculate the quality measures for any clustering given to us. That is not desirable, especially not since the three quality measures would probably get very little information on the quality of the gene clustering.

Hence there is a need for new quality measure.

4.2 Information Inspired Quality

To solve the problems presented earlier in this chapter we need the new measure to be scalable and have the following properties.

- Singleton clusters should reduce the overall quality.
- Fewer and larger clusters should increase the overall quality.
- Higher similarity between members and cluster centers should increase the overall quality.

This section introduces the Information Inspired Quality measure (IIQ) to address these properties. As the name suggests it uses the idea taken from information theory that different events carry a certain amount of information. Information theory is based on the probability of events, but we do not know what the probability of any two elements being highly identical is. We can therefore not use information theory directly, but only the idea of information carrying events.

Imagine that each member m of a cluster contributes a certain amount of information $I^M(m)$ to that cluster. The information depends on how identical the member

is to the cluster center and ranges from 0 to 1. We could describe this as a linear function of the identity between the member and the center: $I^M(m) = \frac{id(m,c)-id_{\min}}{100\%-id_{\min}}$.

A linear relationship might not always be the best fit for real world applications like bioinformatics. We will therefore add a bending parameter α, which allows the relationship to be changed to be supported by domain knowledge. The information contribution from a member is therefore defined as:

$$I^M(m) = \left(\frac{id(m,c) - id_{\min}}{100\% - id_{\min}}\right)^{\alpha}$$

The next step is to define the information content of a cluster c based on the information from its members M_c. Using the average information contribution per member will be a bad choice. That will likely penalize large clusters which is not what we desire. Adding a member with low information to a cluster, whose members provide high information, do not deteriorate the information contribution of its existing members. On the contrary it adds more information. The aggregate should reflect that.

Below are three suggestions for aggregation functions for defining the information content of a cluster. They all increase if the sum of information contributions to the cluster increases, but they present different trade-offs between the distribution of members between clusters.

$$S(c) = \sum_{m \in M_c} I^M(m)$$

$$I_i^C(c) = \log_2(1 + S(c)) \tag{4.1}$$

$$I_{ii}^C(c) = S(c) \tag{4.2}$$

$$I_{iii}^C(c) = S(c)\log_2(1 + S(c)) \tag{4.3}$$

Let us look at two examples to understand how these three aggregation functions are different.

Example 1 Two clusters each with 10 members.

Example 2 One cluster with two members and one cluster with 18 members.

Let the information contribution be the same for all members.

There is no obvious prioritization of which of the two examples that represents the most desirable clustering. That might be domain specific. We can however analyze how the three different aggregation functions prioritizes the examples. The important part is whether the total information content of the two clusters is larger in Example 1 or Example 2.

- $I_i^C(c)$ will give a higher total information content in Example 1,
- $I_{ii}^C(c)$ produces the same total information content for both examples, and

- $I^C_{iii}(c)$ yields the highest total information content in Example 2.

You could argue that $I^C_i(c)$ is the most correct if you strictly view each member as contributing information. Then the information from a member could overlap with information from other members. The value of each new member therefore decreases as the information in the cluster increases.

Another way to argue is that the most important thing is that members gets assigned to the cluster center they are most identical with. This supports the optimal assignment property from Definition 2.1. In this case $I^C_{ii}(c)$ will be most suitable.

Finally one can argue for $I^C_{iii}(c)$ in a probabilistic way. If we view the assignment of members to centers as a random process, and even distribution will be more likely than most elements concentrated in a few clusters. Events, that are less likely, carry more information, so we should reward concentrated assignments and $I^C_{iii}(c)$ will be the best.

It will require a deeper discussion to settle the argument between these three approaches. This thesis is primarily about efficient, distributed algorithms, so such a discussion is outside its scope. We will therefore just go with $I^C_i(c)$.

We take the average of the information in all clusters to calculate the Information Inspired Quality for an entire clustering. This will ensure that the IIQ gets penalized by having a lot of singleton clusters. The total definition of IIQ is presented in Definition 4.4.

Definition 4.4: Information Inspired Quality.

Let a clustering be defined by its set of centers C, and for each center c its assigned members M_c. Let $id(m, c)$ be the function of identity between a member and a center, and let $id_{\min}$ be the minimum identity for assigning a member to a cluster.

We then define the information contribution from a member to its cluster to be

$$I^M(m) = \left(\frac{id(m,c) - id_{\min}}{100\% - id_{\min}} \right)^{\alpha}$$

where α is a tuning parameter to control the bend of the function between identity and information.

We define the information for a cluster to be

$$S(c) = \sum_{m \in M_c} I^M(m)$$

$$I^C(c) = \log_2 (1 + S(c))$$

and the overall Information Inspired Quality to be

$$IIQ = \frac{1}{|C|} \sum_{c \in C} I^C(c)$$

4.3 Summary

In this chapter we saw how existing cluster quality measures are not well suited to evaluate gene clustering. The information they provide are barely usable, and they are infeasible to calculate for large datasets.

To solve this issue the Information Inspired Quality (IIQ) was introduced, which will be the quality measure used for the remaining of this thesis.

CHAPTER 5

Proxies for Gene Identity

In this chapter we will introduce several proxies for gene identity. These proxies are derived from k-mer profiles, which will also be explained.

We need these proxies since there is no known efficient way of calculating the identity for all pairs off genes. The goal is to use a proxy to efficiently find all pairs of genes that very likely are highly identical.

In the last sections of this chapter we will look at two characteristics of k-mer profiles: the frequency of the most frequent k-mer and the average correlation between k-mer profiles. These characteristics will be useful later on when we are analyzing the runtime and scaleability.

This thesis uses the following notation for vectors. A vector is written as $\vec{x}$ and the i'th element is denoted $\vec{x}_i$. When a vector has a label or an identifier it is written as $l_{\vec{x}}$.

5.1 k-mer Profiles

A major problem when dealing with gene identities is that we do not know which nucleotides end up being aligned to each other. Hence it is not possible to do things like location sensitive hashing. This problem can be solved if we use k-mer profiles.

For the sake of explanation we say that the k-mer's are alphabetically ordered, so for example the first 3-mer is AAA and the third 2-mer is AG. For a gene and a k we can construct a k-mer profile. It is a vector of length 4^k. The i'th position is the number of occurrences of the i'th k-mer in the gene. Figure 5.1 shows an toy example with a few short genes and their corresponding 2-mers.

5.2 k-mer Profiles as Proxies

It turns out that the k-mer profiles of two genes can be used as an indicator of their identity [Edg04]. [Edg10] defines a similarity measure between two k-mer profiles and shows that it correlates highly with the gene identity. It will be called common count in this thesis (Definition 5.1). [Edg10] does not analyze any other correlation metrics for two k-mers.

(a) Genes (b) First six positions of the 2-mer profiles

Figure 5.1: Example of 2-mer profiles for a few short genes.

We will also look at two popular similarity measures for vectors: cosine similarity (Definition 5.2) and Pearson correlation (Definition 5.3). Theses definitions will use the following notation.

Let $\vec{p}$ and $\vec{q}$ be two k-mer profiles and n be their length. Let $\|\ldots\|_2$ denote the normal Euclidean norm and let $\|\ldots\|_1$ denote the Taxicab norm, that is, the sum of absolute values.

Definition 5.1: Common Count [Edg10].

The common count is the number of (non-unique) k-mers the two genes have in common. It is normalized by the number of k-mers in the shortest gene, so it ranges from 0 to 1.

$$\frac{1}{\min\left(\|\vec{p}\|_1, \|\vec{q}\|_1\right)} \sum_{i=1}^{n} \min\left(\vec{p}_i, \vec{q}_i\right)$$

This only applies to nonnegative values.

Definition 5.2: Cosine Similarity [Wik15a].

We can view the two k-mer profiles as vectors in a high dimensional space. The cosine similarity is then the cosine of the angle between the two vectors. It

ranges from 0 to 1 when the vectors are nonnegative.

$$\frac{1}{\|\vec{p}\|_2 \|\vec{q}\|_2} \sum_{i=1}^{n} \vec{p}_i \vec{q}_i$$

Definition 5.3: Pearson Correlation [Wik15d].

The Pearson correlation is popular in statistics. Its formula is very similar to the cosine similarity, but the mean of each vector is subtracted from its values.

$$\frac{\sum_{i=1}^{n}(\vec{p}_i - \mu_{\vec{p}})(\vec{q}_i - \mu_{\vec{q}})}{\sqrt{\sum_{i=1}^{n}(\vec{p}_i - \mu_{\vec{p}})^2}\sqrt{\sum_{i=1}^{n}(\vec{q}_i - \mu_{\vec{q}})^2}}$$

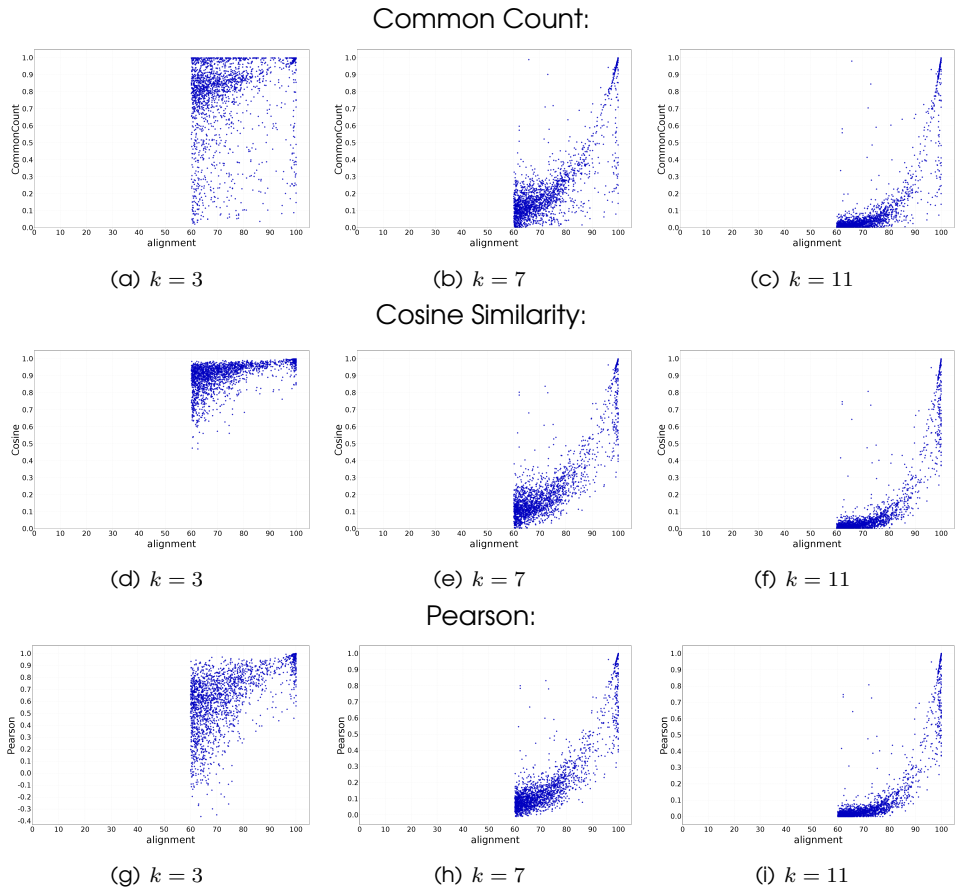
where $\mu_{\vec{p}}$ is the mean of the values in $\vec{p}$.

It would be beneficial to know if other metrics have similar or maybe better prediction of gene identity. In this way a measure can be chosen whose computation fits better with new algorithms. This chapter will therefore analyze how well cosine similarity, Pearson correlation, and common count works as a proxy for gene identity and how well they can identify gene pairs with an identity above the threshold $id_{\min}$.

Figure 5.2 shows scatters of the three metrics over the gene identity. They are based on the subsample of 16,000 genes from the Human Microbiome Project (HMP) dataset. All pairs with an identity above 60% have been included. The plots clearly show that all three metrics quickly drop as k increases for pairs with a relatively low identity. For pairs with a high identity that drops happen much slower, and most of these pairs maintain high values across all three metrics.

All three metrics should therefore be usable as a proxy for gene identity. One important difference is that a few pairs with a high identity has a very low common count, to a much higher extend than Pearson correlation and cosine similarity. It will therefore be more difficult to use common count as a predictor of identities above $id_{\min}$, since a threshold will need to be much lower and thus include more false positives.

For the remainder of this thesis we will focus on cosine similarity. There are two primary reasons for this: we would expect it to be a better predictor than common count as explained above, and we can scale the k-mer profiles to get rid of the denominator in the function for cosine similarity. This will be useful later on in Chapter 7. Pearson correlation could also have been chosen, since it has the same properties, but to simplify its function we would need to both scale and shift the k-mer profiles.

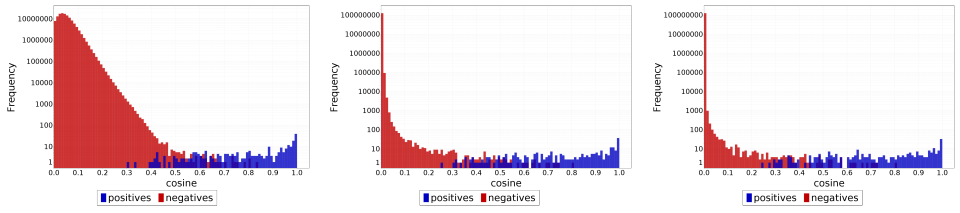


(a) $k = 3$ (b) $k = 7$ (c) $k = 11$

(d) $k = 3$ (e) $k = 7$ (f) $k = 11$

(g) $k = 3$ (h) $k = 7$ (i) $k = 11$

Figure 5.2: Scatterplots of the three similarity measures for different k. The x-axis is the gene identity in percent. More figures are available in Appendix B.



(a) $k = 7$ (b) $k = 11$ (c) $k = 15$

Figure 5.3: Histograms over the cosine similarity between k-mer profiles for different k. More figures are available in Appendix B.

5.2.1 Finding a Good Cutoff Value

We need to use the proxy as a predictor of whether or not a given pair of genes has an identity above $id_{\min} = 95\%$. To do this we will need to determine a threshold. Figure 5.3 shows histograms of cosine similarity between pairs of genes for different k. Blue indicates pairs with an identity above $id_{\min} = 95\%$. It is obvious that we cannot choose a perfect threshold without either false positives and false negatives. It is however possible to choose a threshold without any false negatives that still cut off the vast majority of all the negative pairs.

Having a false negative is a significant problem, since it might mean we create a suboptimal clustering. On the contrary, some false positives are not a big issue, as long as they a few enough so we can feasibly calculate their actual identity and filter on that. A threshold with no false negatives and limited false positives will therefore be preferred. Such thresholds are possible for higher k.

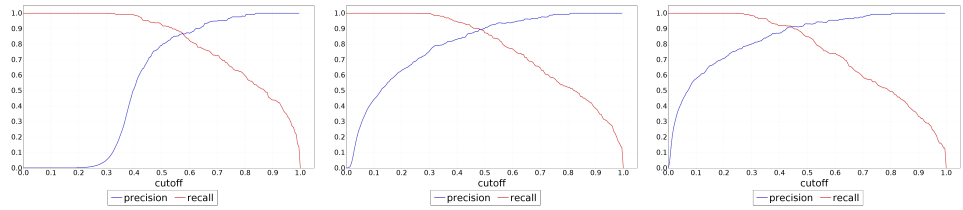
Figure 5.4 visualizes the precision and recall depending on the threshold. These are based on a subsample of 16,000 genes, so we will preferably choose a threshold which gives no false negatives with a margin to spare. E.g. for $k = 15$ a cutoff value of 0.1 would be a safe choice, while 0.2 might also work if one were willing to accept a negligible number of false negatives.

5.3 Distribution of k-mers

Not all k-mers are equally common and sometimes we are interested in the frequency of the most common k-mer, since that might be the computational bottleneck.

A gene of length l has $l - k + 1$ non-unique k-mers. The total number of k-mers found in N genes is therefore $|bp| - kN + N$, where $|bp|$ is the total number of base pairs. The average frequency of k-mers is therefore $\frac{|bp|-kN+N}{4^k}$.

The most common k-mer will have to occur more often than $\frac{|bp|-kN+N}{4^k}$, but for most applications it will be sufficient if it can be bounded by $O\left(\frac{NL}{r^k}\right)$ for some $r > 1$, where L is the maximum length of a gene.



(a) $k = 7$ (b) $k = 11$ (c) $k = 15$

Figure 5.4: Plots of the precision and recall depending on the cutoff value for different k. More figures are available in Appendix B.

We will analyze the bound in two different ways: a theoretical and empirical. Theoretical approaches usually gives a basic guarantee of generalization, while empirical analyses depends on the dataset. In this case the theoretical analysis depends heavily on an assumption that might not be true in real life. The empirical analysis is therefore required.

5.3.1 Theoretical Analysis of Most Frequent k-mer

The theoretical analysis will take shape of an induction proof. Before we can get to the proof, we need to introduce some notation.

Let K be any specific k-mer ($k \geq 0$), and $\Pr[K]$ be the probability of finding K in any sequence of $|K|$ base pairs. Extending K with an additional base pair will be denoted as $K\mathsf{A}, K\mathsf{C}, K\mathsf{G}, K\mathsf{T}$ for each of the four possible base pairs respectively. Let the possibilities of each of these instances be denoted as:

$$a_K = \frac{\Pr[K\mathsf{A}]}{\Pr[K]} \qquad c_K = \frac{\Pr[K\mathsf{C}]}{\Pr[K]} \qquad g_K = \frac{\Pr[K\mathsf{G}]}{\Pr[K]} \qquad t_K = \frac{\Pr[K\mathsf{T}]}{\Pr[K]}$$

The sum $a_K + c_K + g_K + t_K$ can be less than 1, since K might be in the end of some genes so an additional base pair can not be added.

Now that the notation is settled we can define the assumption. Assume there exists a constant $r > 1$ such that $\frac{1}{r} = \max_{\forall K} \max(a_K, c_K, g_K, t_K)$, in other words, there exist no k-mer that is followed by a specific base pair with probability higher than $\frac{1}{r}$.

Given that assumption we can define the induction hypothesis as:

The probability of finding any specific k-mer K in a sequence of $|K|$ is bounded by:

$$\Pr[K] \leq \frac{1}{r^{|K|}}$$

The base case is the 0-mer, which is the empty string ϵ. The chance of finding the 0-mer in a string of length 0 is of course $\Pr[\epsilon] = 1$. The base case supports the hypothesis since $\frac{1}{r^0} = 1 = \Pr[\epsilon]$.

The induction step is to show that the hypothesis holds for $K\mathsf{A}, K\mathsf{C}, K\mathsf{G}, K\mathsf{T}$, assuming that it holds for K. We only do the step for $K\mathsf{A}$. The rest follows due to symmetry.

$$\begin{aligned}
\Pr[K\mathsf{A}] &= a_K \Pr[K] \\
&\leq \frac{1}{r} \Pr[K] && \text{Using the assumption} \\
&\leq \frac{1}{r} \frac{1}{r^{|K|}} && \text{Using the induction hypothesis} \\
&= \frac{1}{r^{|K|+1}}
\end{aligned}$$

The hypothesis is thus proved to hold.

5.3.2 Weakness of the Assumption

The assumption in Section 5.3.1 seems reasonable, but there might be k-mers that also specific that the next one of several base pairs are given. This will be easier to exemplify if you think of DNA as text and k-mers as small excerpts. Imagine the excerpt "*the President of the United St*" it is practically given that the next characters will be "*ates*". Or imagine the probability of the next character if the previous 100,000 characters was exactly the first 100,000 characters of Shakespeare's Hamlet.

We can also break the assumption from a more mathematical standpoint. As indicated above the longer any string of characters or DNA is, the more certain we will be about the next character or base pair. We do not have to be absolutely certain, that is, forcing $r = 1$. We can break the assumption if we let r be dependent on $|K|$, which we will denote as $r(k)$. We still have that $r(k) < 1$, but we will now show that that is not a strong enough assumption.

Now let us say that long sequences of A's are very likely. The sequences tend to persist, so the probability of the next base pair being A increases the more A that comes before it. Let K' be such k-mer, that is, $K' = \mathsf{AAA}\ldots\mathsf{A}$ is a k-mer of any size $|K'| \geq 1$ containing only A's.

Let the probability of K' be given by

$$\Pr[K'] = \frac{1}{2} + \frac{1}{4^{|K'|}}$$

This probability will clearly break the induction hypothesis. If we can find a definition of $a_{K'}$ that yields that probability but still satisfy $r(k) < 1$, then the induction hypothesis does not hold.

Such $a_{K'}$ can easily be defined:

$$\begin{aligned} a_{K'} \Pr[K'] &= \Pr[K'\mathsf{A}] \\ a_{K'} \left(\frac{1}{2} + \frac{1}{4^{|K'|}} \right) &= \left(\frac{1}{2} + \frac{1}{4^{|K'|+1}} \right) \\ a_{K'} &= \frac{\frac{1}{2} + \frac{1}{4^{|K'|+1}}}{\frac{1}{2} + \frac{1}{4^{|K'|}}} \end{aligned}$$

Both intuition and a mathematical arguments indicate that the assumption might be weak, so an empirical analysis is in place. The theoretical analysis is however not without value. The theoretical proof gives us a higher confidence that the empirical findings will generalize.

5.3.3 Empirical Analysis of Most Frequent k-mer

Though the assumption that a constant r exists seems reasonable, we must test it on real data. Figure 5.5 shows the frequencies of the most common k-mers in the dataset analyzed in this thesis. The figures is based on a subsample of 16,000 genes containing 12 million base pairs. The implementation limits k at 15.

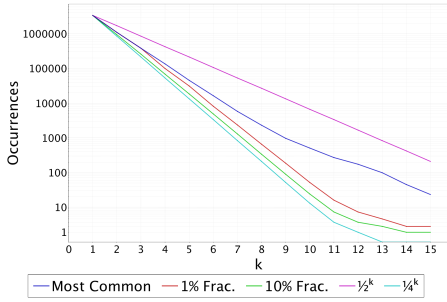


Figure 5.5: The frequency of the most frequent k-mer, the 1% and 10% quantiles, and reference lines for $r = 2$ and $r = 4$. The references are rounded to nearest integer.

Within this range it is easy to see that $r > 2$ is perfectly valid, since the average correlation stays beneath the reference curve for $O\left(\frac{1}{2^k}\right)$. There is a slight tendency that the curve for the most frequent k-mer might be flattening towards higher k. This could be due to the limited data size. For $k = 15$ we have $4^{15} \approx 10^9$ possible k-mers, but we have only $\approx 10^7$ basepairs in the test dataset. It could also be because the assumption does not hold for large k, but that lies outside the range of this application.

In conclusion it is safe to assume for this application that the frequency of the most common k-mer is bounded by $O(\frac{NL}{2^k})$.

5.4 Average Correlation

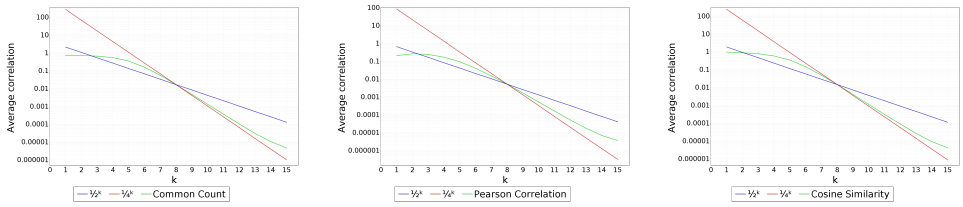
Previously in this chapter we have seen how the correlation between pairs of genes drops as k increases. It is however difficult to tell the exact magnitude of the drop.

Figure 5.6 shows how the average correlation decreases exponentially for all three metrics when k increases, at least for $5 \leq k \leq 15$. There seems to be a slight flattening of the decline at $k = 15$, but we will not go above that point in this thesis.

5.5 Summary

In this chapter we have shown the correlation between the k-mer profiles of two genes can be used as a good proxy for their identity. We found that Pearson correlation and cosine similarity works even better than common count which is used by UClust.

In the range $1 \leq k \leq 15$ we found that a higher k will make the correlation the better predictor of whether two genes have a high identity. Additionally we showed



(a) Common Count (b) Pearson (c) Cosine Similarity

Figure 5.6: The average correlation between k-mer profiles of two genes (green) and exponentially decreasing reference lines.

that the average correlation between two pairs and the frequency of the most frequent k-mer both decrease exponentially in k.

CHAPTER 6

Distributed Correlation Computation

In Chapter 5 it was shown that the correlation between k-mer profiles can be used as a good proxy for gene identity. This chapter presents an efficient, distributed algorithm to find highly correlated pairs of vectors. Chapter 7 will combine the two and show, how efficiently finding pairs of highly correlated k-mer profiles can be used to make an efficient gene clustering algorithm.

The algorithm presented in this chapter is a generalization of the DISCO algorithm developed at Twitter to find correlate words and tweets [ZG13]. To begin with, this chapter will explain the difference between the problem they solved at Twitter and the problem needed to be solved in this thesis. The DISCO algorithm will then be introduced along with an explanation of the changes needed to generalize it. Finally we will go over proofs of scalability and error bounds for the generalized algorithm. This section will not include the proof made for the original DISCO algorithm.

6.1 Difference in Problem Statements

Let us settle some terminology before we start describing the problems.

We will view a set of vectors as rows in a matrix. Each column will thus be the frequencies of one specific k-mer across all genes. We would call these cross columns to emphasize that they go across the k-mer profiles.

The data is sparse so let us recap the definition of sparsity. A vector or cross column is x-sparse if there is at most x nonzero elements in the vector or cross column.

And finally, let us recap that an "addend" is one of the parts added together in a sum, that is, a_i is an addend in $\sum_{i=1}^{N} a_i$.

Now we are ready to understand the problems. The problems do not define which measure to use as a correlation measure. It is left to the algorithms to define which correlation measures they support.

6.1.1 The Original Problem

[ZG13] presents the Dimension Independent Similarity Computation (DISCO) algorithm to find highly correlated words in tweets on Twitter using the MapReduce

framework. The algorithm is able to find highly correlated tweets with shuffle size that is independent of the number of tweets.

They formulate the problem as such.

> **Problem 6.1**: Approximate word occurrence correlation.
>
> For each of N words you have an M-dimensional binary occurrence vector indicating whether the word occurs in each of M tweets. Each cross column is S sparse, in other words, each tweet has at most S words. $S \ll N \ll M$
>
> Approximate the correlation of the occurrence vectors for all highly correlated word pairs.

They assume that the data is stored per tweet, that is, the values are stored in cross columns. In addition all servers know the frequency of all N words.

6.1.2 The Generalized Problem

There are several issues that prevents the DISCO algorithm to be applied directly to calculating the correlation of k-mer profiles.

1. k-mer profiles are not binary vectors.
2. We have no guarantee on the sparsity of cross columns, only a guarantee of the sparsity of each vector.
3. The k-mers are initially stored vector-wise.

The last can be solved by doing an shuffle of all the data, but the other two need a more generalized version of the DISCO algorithm. Formally we need a solution to Problem 6.2.

> **Problem 6.2**: Approximate real vector correlation.
>
> You have N real vectors of dimension M. Each vector is L sparse, and each cross column is S sparse. There are defined no bounds on L or S.
>
> Approximate the correlation between all pairs of highly correlated pairs of vectors.

6.2 Algorithmic Changes

The problem differences require the DISCO algorithm to be changed. It is not possible just to change the input type. To understand why we first need to know how the original DISCO algorithm works.

Both algorithms will use a tuning parameter λ. [ZG13] calls this the oversampling parameter and uses a different notation.

6.2.1 The Original DISCO Algorithm

The DISCO algorithm in [ZG13] was designed for the MapReduce framework [DG08]. It is a cloud distribution framework like Apache Spark, but has been around for longer. There are two primitive steps: a map step and a reduced step. In a map step each input element is transformed into a series of key-value pairs. These pairs are "emitted" into the framework. The framework then performs a "shuffle" that groups all emitted pairs with the same key. The reduced function is passed a list of all pairs with the same key and reduces them to a single value.

[ZG13] defines a general approach for every correlation measure (Algorithm 6.1). The map function is a template that iterates over every unique pair which is passed to an emission function that is customized for each similarity measure (`CustomEmitFunction`). The reduce step is general for all correlation measures and just sums the emitted values for each pair and divides it by λ.

A series of custom emitters are presented for different correlation measures. The emitter for cosine similarity is shown in Algorithm 6.2, as an example. The fraction in line 1 is the addend from the equation for cosine similarity. The enumerator is 1

Algorithm 6.1: `ApproximateCorrelation`(*tweets*)
tweets is a parallel collection.
Each element is a set $\{l_w, \ldots\}$ of labels of all words in a tweet.

1: ▷ *Map step: Process each tweet in parallel:*
2: **for all** $tweet \in tweets$ **do in parallel**
3: ▷ *For each unique pair of words:*
4: **for all** $l_w, l_{w'} \in tweet \times tweet$ **where** $l_w < l_{w'}$ **do**
5: `CustomEmitFunction`(l_w, $l_{w'}$)
6: **end for**
7: **end for**
8: ▷ *Reduce step:*
9: $X \leftarrow$ sum all emissions by pair label
10: $Corr \leftarrow X \cdot \frac{1}{\lambda}$
11: **return** $Corr$

Algorithm 6.2: `CosineEmitFunction`(l_w, $l_{w'}$).

1: $a \leftarrow \lambda \frac{1}{\|\vec{w}\|_2 \left\|\vec{w'}\right\|_2}$
2: **if** $a >$ `random`() **then**
3: `emit`$(l_w, l_{w'}) \mapsto 1$
4: **end if**

since all nonzero elements in the vector are 1. $\|\vec{w}\|_2 = \sqrt{freq(w)}$ is the Euclidean norm of the occurrence vector for the word w. Please refer to the original article for other emitters.

6.2.2 Generalized DISCO Algorithm

It is obvious that the DISCO algorithm needs to be updated to deal with real valued vectors. For example is the actual value of a vector never considered; all nonzero values are assumed to be one. It might be tempting just to update line 1 in Algorithm 6.2 to something like this:

$$a \leftarrow \lambda \frac{\vec{x}_i \vec{y}_i}{\|\vec{x}\|_2 \|\vec{y}\|_2}$$

but that leads to some new problems. What do we do when $a < 0$, or how do we ensure correctness if sometimes $a > 1$.

The solution to these problems is not one that is specific for cosine similarity. The DISCO algorithm will therefore not only be generalized not only to handle real valued vectors for cosine similarity, but for a whole class of correlation measures.

Random Addends

We assume that the correlation $c^{\vec{x},\vec{y}}$ of two vectors $\vec{x}, \vec{y}$ can be calculated as the sum of addends:

$$c^{\vec{x},\vec{y}} = \sum_{i=1}^{M} f^{\vec{x},\vec{y}}(\vec{x}_i, \vec{y}_i) \tag{6.1}$$

It is required that $f^{\vec{x},\vec{y}}(0, z) = f^{\vec{x},\vec{y}}(z, 0) = 0$ for any value z.

Note that the addend function $f^{\vec{x},\vec{y}}$ is annotated with x, y to indicate that it might include auxiliary information. For the cosine similarity the addend function is $f^{\vec{x},\vec{y}}(\vec{x}_i, \vec{y}_i) = \frac{\vec{x}_i \vec{y}_i}{\|\vec{x}\|\|\vec{y}\|}$ and it needs to know the norm of the vectors as auxiliary information.

Collecting all addends will not be efficient, so similar to [ZG13] we use randomness to emit fewer values. We will then calculate the approximate correlation $c^{\vec{x},\vec{y}}_{\text{app}}$ instead of the exact correlation $c^{\vec{x},\vec{y}}$.

We define a series of random variables $X_i^{\vec{x},\vec{y}}$, one for each addend. We want these random addends to be zero often, so we define them as

$$a = \lambda f^{\vec{x},\vec{y}}(\vec{x}_i, \vec{y}_i) \tag{6.2}$$

$$X_i^{\vec{x},\vec{y}} = \begin{cases} a & \text{if } |a| \geq 1 \\ B(1, |a|) \cdot \text{sign}(a) & \text{if } |a| < 1 \end{cases} \tag{6.3}$$

where $B(1, |a|)$ is the Bernoulli (i.e. binary) random variable which is 1 with probability $|a|$ and 0 otherwise.

This definition gives us the same expected value for the random addends as [ZG13] achieved, but with the ability to handle real values:

$$E[X_i^{\vec{x},\vec{y}}] = \lambda f^{\vec{x},\vec{y}}(\vec{x}_i, \vec{y}_i) \tag{6.4}$$

Summing the random addends gives us

$$X^{\vec{x},\vec{y}} = \sum_{i=1}^{M} X_i^{\vec{x},\vec{y}} \tag{6.5}$$

$$c^{\vec{x},\vec{y}} = \frac{1}{\lambda} E[X^{\vec{x},\vec{y}}] \tag{6.6}$$

$$c_{\text{app}}^{\vec{x},\vec{y}} = \frac{1}{\lambda} X^{\vec{x},\vec{y}} \tag{6.7}$$

Modifying the DISCO Algorithm

The random addends enables us to create one algorithm for every correlation measure, without the need for customized emission functions. The generalized DISCO algorithm is almost the same as Algorithm 6.1, but the emission part has been updated to match with the definition of the random addends.

Algorithm 6.3 shows the process: first we emit all non-zero random variables $X_i^{\vec{x},\vec{y}}$; then we find $X^{\vec{x},\vec{y}}$ by summing the emissions grouped by their pair key; and lastly we find the correlation by dividing by λ.

Algorithm 6.3: ApproximateCorrelation(crossColumns)
crossColumns is a parallel collection
Each element is a list $[(l_{\vec{x}}, \vec{x}_i), \ldots]$ of all nonzero values from a cross column.

▷ *Process each cross column in parallel:*
for all *column* ∈ *crossColumns* **do in parallel**
  ▷ *For each unique pair of values:*
  for all $(l_{\vec{x}}, \vec{x}_i), (l_{\vec{y}}, \vec{y}_i) \in column \times column$ **where** $l_{\vec{x}} < l_{\vec{y}}$ **do**
    ▷ *Emit the random variable:*
    $a \leftarrow \lambda f^{\vec{x},\vec{y}}(\vec{x}_i, \vec{y}_i)$
    if $|a| >$ `random()` **then**
      emit $(l_{\vec{x}}, l_{\vec{y}}) \mapsto \max(|a|, 1)\,\text{sign}(a)$
    end if
  end for
end for
$X \leftarrow$ sum all emissions by vector pair label ▷ $X^{\vec{x},\vec{y}}$ for all pairs
$Corr \leftarrow X \cdot \frac{1}{\lambda}$ ▷ $c_{\text{app}}^{\vec{x},\vec{y}}$ for all pairs
return *Corr*

Code Example 6.1: Approximate correlation with Spark.

```
1  /**
2   * Calculates approximate correlation
3   * @param elements an RDD of (columnId, (vectorId, value))
4   * @return RDD of (vectorIdPair,correlation)
5   */
6  def approxCorrelation(elements : RDD[(Int, (Long, Double))]) = {
7    // Group values by column index
8    val crossColumns = elements.groupByKey()
9    // "Emit" (flapmap) all addends
10   val addends = crossColumns flatMap {case (_, idValPairs) =>
11     for ((id1,val1) <- idValPairs;
12          (id2,val2) <- idValPairs;
13          if id1 < id2;
14          a = lambda * f(val1, val2);
15          if a > math.random
16     ) yield {
17       ((id1, id2), math.max(1.0, a))
18     }
19   }
20   // Sum all addends and calculate the correlation
21   addends.reduceByKey{_ + _}.mapValues{_ / lambda}
22 }
```

Spark handles the distribution for us as explained in Chapter 3, so the implementation is straight forward as shown in Code Example 6.1. The implementation assumes that `f(val1, val2)` is never negative (line 14, Code Example 6.1). This assumption is valid when using cosine similarity on nonnegative vectors such as k-mer profiles.

6.3 Error Bounds

The analysis done in [ZG13] only applies to binary vectors. The proofs will therefore have to be done once more for real valued vectors.

The original error bounds are still included for reference, but most of the section will be the proofs for the generalized DISCO algorithm.

6.3.1 The Original Error Bounds for Cosine Similarity

[ZG13] analyzes the error bounds separately for each correlation measure. For simplicity only the error bounds for cosine similarity will be described here. Please refer to the original article for other correlation measures and for proofs.

We are only interested in correlations for highly correlated pairs. [ZG13] defines highly correlated as having a correlation value above ϵ.

The approximation of the cosine similarity for any pair (x, y) is denoted by $\cos_{approx}(x, y)$. If the cosine similarity is large enough $\cos(x, y) \geq \epsilon$ the error is bounded by

$$\Pr\left[\cos_{approx}(x,y) > (1+\delta)\cos(x,y)\right] \leq \left(\frac{e^{\delta}}{(1+\delta)^{(1+\delta)}}\right)^{\epsilon\lambda} \tag{6.8}$$

$$\Pr\left[\cos_{approx}(x,y) < (1+\delta)\cos(x,y)\right] < \exp\left(-\frac{\epsilon\lambda\delta^2}{2}\right) \tag{6.9}$$

Note that the approximation error decreases exponentially in λ.

6.3.2 Generalized Error Bounds for Nonnegative Correlation Measures

When determining the error bounds for the generalized DISCO algorithm, there will not be made any assumptions on the minimum correlation, as opposed to the bounds found in [ZG13]. The error bounds will depend on the correlation instead.

We assume that $a \geq 0$ during these proofs. That will be sufficient for the problem of metagenomic gene clustering, since we will use cosine similarity on nonnegative vectors. The generalized DISCO algorithm is still correct even when a is sometimes negative, but the error bounds might be different.

We will be analyzing the correlation between two vectors $\vec{x}$ and $\vec{y}$. To ease reading the superscripts of $X^{\vec{x},\vec{y}}$, $f^{\vec{x},\vec{y}}$, etc. will be omitted.

The upper bounds are presented in Theorem 6.1. Note that especially the upper bound differs from that found in [ZG13].

Theorem 6.1: Error bounds on Distributed Correlation Computation.

For any two vectors $\vec{x}$ and $\vec{y}$ with correlation c, the error of running the Distributed Correlation Computation with parameter λ is bounded by.

$$\Pr\left[\frac{1}{\lambda}X \leq (1-\delta)c\right] \leq \exp\left(-\lambda\frac{c\delta^2}{2}\right) \tag{6.10}$$

$$\Pr\left[\frac{1}{\lambda}X \geq (1+\delta)c\right] \leq \exp\left(-\lambda\frac{c\delta^2}{2+\delta}\right) \tag{6.11}$$

In order to ease the analysis of the error bounds, we introduce two new random variables. X_f is the sum of the fixed addends, which are addends that are emitted with probability 1. Note that $E[X_f] = X_f$ always holds. X_r is the sum of all actually random addends, that have a non-zero probability of being 0.

$$X_f = \sum(X_i) \qquad \text{where } E[X_i] \geq 1 \tag{6.12}$$
$$X_r = \sum(X_i) \qquad \text{where } E[X_i] < 1 \tag{6.13}$$
$$X = X_f + X_r \tag{6.14}$$

Further let μ denote the expected value such that

$$E[X] = \mu \tag{6.15}$$
$$E[X_f] = \mu_f \tag{6.16}$$
$$E[X_r] = \mu_r \tag{6.17}$$

The first step in analyzing the bounds, is to get rid of the fixed addends (Lemma 6.2). Afterwards we will prove Theorem 6.1 using the Chernoff Bound (Definition 6.4). This requires the use of Lemma 6.3.

Lemma 6.2:
Let $\delta_r = \delta\frac{\mu}{\mu_r}$ where $\delta \geq 0$, then

$$\Pr[X \geq (1+\delta)\mu] = \Pr[X_r \geq (1+\delta_r)\mu_r] \tag{6.18}$$
$$\Pr[X \leq (1-\delta)\mu] = \Pr[X_r \leq (1-\delta_r)\mu_r] \tag{6.19}$$

Proof of Lemma 6.2. For the lower bound

$$\begin{aligned} X &\leq (1-\delta)\mu \\ X &\leq \mu - \delta\mu \\ X_f + X_r &\leq \mu_f + \mu_r - \delta\mu \\ X_r &\leq \mu_r - \delta\mu \\ X_r &\leq \mu_r - \delta_r\mu_r \end{aligned}$$

and symmetrically for the upper bound

$$\begin{aligned} X &\geq (1+\delta)\mu \\ X_f + X_r &\geq \mu_f + \mu_r + \delta\mu \\ X_r &\geq \mu_r + \delta_r\mu_r \end{aligned}$$

□

Lemma 6.3:

$$\exp\left(-\frac{\mu\delta^2}{2\frac{\mu_r}{\mu}+\delta}\right) \leq \exp\left(-\frac{\mu\delta^2}{2+\delta}\right) \tag{6.20}$$

Proof of Lemma 6.3.

$$\begin{aligned}
\exp\left(-\frac{\mu\delta^2}{2\frac{\mu_r}{\mu}+\delta}\right) &\leq \exp\left(-\frac{\mu\delta^2}{2+\delta}\right) \\
-\frac{\mu\delta^2}{2\frac{\mu_r}{\mu}+\delta} &\leq -\frac{\mu\delta^2}{2+\delta} \\
\frac{1}{2\frac{\mu_r}{\mu}+\delta} &\geq \frac{1}{2+\delta} \\
2\frac{\mu_r}{\mu}+\delta &\leq 2+\delta \\
\frac{\mu_r}{\mu} &\leq 1
\end{aligned}$$

□

Definition 6.4: Chernoff Bound.

Let $X_1, \ldots, X_n$ be independent random variables, such that $0 \leq X_i \leq 1$ for all i. Let $X = \sum_{i=1}^{n} X_i$ and $E[X] = \mu$. Then for any $\delta \geq 0$ we have

$$\Pr[X \geq (1+\delta)\mu] \leq \exp\left(-\frac{\delta^2\mu}{2+\delta}\right) \tag{6.21}$$

$$\Pr[X \leq (1-\delta)\mu] \leq \exp\left(-\frac{\delta^2\mu}{2}\right) \tag{6.22}$$

Proof of Theorem 6.1. By applying the Chernoff Bound (Definition 6.4) to Lemma 6.2. First for the lower bound:

$$\begin{aligned}
\Pr[X_r \leq (1-\delta_r)\mu_r] &\leq \exp\left(-\frac{\mu_r\delta_r^2}{2}\right) \\
\Pr[X_r \leq (1-\delta_r)\mu_r] &\leq \exp\left(-\frac{\mu_r\delta_r^2}{2}\right) && \text{Using (6.19)} \\
&= \exp\left(-\frac{\mu_r\left(\delta\frac{\mu}{\mu_r}\right)^2}{2}\right)
\end{aligned}$$

$$
\begin{aligned}
&= \exp\left(-\frac{\mu\delta^2}{2} \cdot \frac{\mu}{\mu_r}\right) \\
&\leq \exp\left(-\frac{\mu\delta^2}{2}\right) \\
\Pr\left[\frac{1}{\lambda}X \leq (1-\delta)c\right] &\leq \exp\left(-\lambda\frac{c\delta^2}{2}\right)
\end{aligned}
$$

And then the upper bound:

$$
\begin{aligned}
\Pr[X_r \geq (1+\delta_r)\mu_r] &< \exp\left(-\frac{\mu_r\delta_r^2}{2+\delta_r}\right) \\
\Pr[X \geq (1+\delta)\mu] &< \exp\left(-\frac{\mu_r\delta_r^2}{2+\delta_r}\right) && \text{Using (6.18)} \\
&= \exp\left(-\frac{\mu_r\left(\delta\frac{\mu}{\mu_r}\right)^2}{2+\left(\delta\frac{\mu}{\mu_r}\right)}\right) \\
&= \exp\left(-\frac{\mu\delta^2}{2+\left(\delta\frac{\mu}{\mu_r}\right)} \cdot \frac{\mu}{\mu_r}\right) \\
&= \exp\left(-\frac{\mu\delta^2}{2\frac{\mu_r}{\mu}+\delta}\right) \\
&\leq \exp\left(-\frac{\mu\delta^2}{2+\delta}\right) && \text{Using Lemma 6.3} \\
\Pr\left[\frac{1}{\lambda}X \geq (1+\delta)c\right] &\leq \exp\left(-\lambda\frac{c\delta^2}{2+\delta}\right) && \square
\end{aligned}
$$

6.3.3 Scalability

We need to analyze the total time, slowest element, and shuffle size for each step, as described in Chapter 3. Recall that N is the number of vectors, that is, the number of tweets or genes. M is the dimension of the vectors. Each vector is L sparse, and each cross column is S sparse.

6.3.4 Scalability of the Original DISCO Algorithm

[ZG13] did not compute the total time, but they analyzed the slowest element and shuffle size. They did that separately for each correlation measure.

For cosine similarity to found that the expected shuffle size is $O(SN\lambda)$ and the slowest element in the reduced operation to take $O(\lambda)$ time. They argue that S can be issued to be constant, since in their case it is a fixed low number, namely the number of words in a tweet.

They did not analyze the total time nor the slowest map operation, but it is easy to see that every map operation takes $O(S^2)$ time, and the total time follows to be $O(S^2M)$.

The key takeaways from this analysis are

- the shuffle size does not depend on the dimension of the vectors, i.e., the number of tweets, and
- the running time increases linearly in λ, while the error decreased exponentially in λ.

The next subsection will show that the same takeaways are true for the generalized DISCO algorithm.

6.3.5 Scalability of the Generalized DISCO Algorithm

Listing 6.1 shows an overview of the runtime for each step. The time and network I/O it takes to distribute auxiliary information to the addend function should be added on top.

The step collect cross columns is only required if the data is not stored in cross columns. It requires a complete shuffle of all the data $O(NL)$. The slowest element is the cross column with the most values. That is defined to be S, so the runtime will be $O(S)$.

Each non-zero value will be compared with at most S other value when creating the addends, which gives a total time of $O(NLS)$. The slowest element is the cross column with the most non-zero values, which takes $O(S^2)$. The data is not shuffled at this step.

The expected number of addends for each $X^{\vec{x},\vec{y}}$ is at most $\lambda c^{\vec{x},\vec{y}}$, since each addend is at least 1. Thus the expected total number of addends will be $O(\lambda C)$ where C is the sum of all correlations $C = \sum_{x=1}^{N} \sum_{y=x+1}^{N} c^{\vec{x},\vec{y}}$. Summing them up will thus take $O(\lambda C)$ time. There is no bottle neck, since `reduceByKey` calculates the sum continuously. This will also mean, that the shuffle size will likely be lower than the number of addends. How much lower depends on the number of servers versus λ and

Listing 6.1: Runtime for different steps of distributed correlation computation. C is the sum of all correlations $C = \sum_{x=1}^{N} \sum_{y=x+1}^{N} c^{\vec{x},\vec{y}}$.

Step	Spark Operation	Total Time	Slowest	Shuffle Size
Collect Cross Columns	`groupByKey`	$O(NL)$	$O(S)$	$O(NL)$
Emit Addends	`flatMap`	$O(NLS)$	$O(S^2)$	–
Sum Addends	`reduceByKey`	$O(\lambda C)$	$O(1)$	$O(\lambda C)$
Calculate Correlation	`map`	$O(\lambda C)$	$O(1)$	–

the distributions of addends on these servers. It is also possible that no two addends belong to the same pair, e.g., if $\lambda = 1$. We can therefore not reduce the asymptotic shuffle size.

The worst case for calculating the actual correlation is when all addends belongs to different pairs. In this case we have $O(\lambda C)$ non-zero correlations.

6.4 Application to Gene Clustering

When clustering genes we will like to calculate the cosine similarity between all pairs of k-mer profiles. This will mean that the addend function should be:

$$f^{\vec{x},\vec{y}}(\vec{x}_i, \vec{y}_i) = \frac{\vec{x}_i \cdot \vec{y}_i}{\|\vec{x}\| \, \|\vec{y}\|}$$

This function require the norm of the vectors as auxiliary information, but we can get rid of that by normalizing the vectors, since the cosine similarity is stable towards scaling. We then get this addend function, that require no auxiliary knowledge:

$$f(\vec{x}_i, \vec{y}_i) = \vec{x}_i \cdot \vec{y}_i$$

In the previous section we showed how the shuffle size and runtime depends on the sparsity L and S plus the sum of all correlations C. L is equal to the length of the genes, but S and C depends highly on k. We saw in Chapter 5 how the correlation between k-mer profiles drops exponentially as k increases, so we can say that $C = O\left(\frac{N^2}{2^k}\right)$. We also saw how the frequency of the most common k-mer decreases exponentially, so we have $S = O\left(\frac{NL}{2^k}\right)$. Substituting this knowledge into Listing 6.1, we get the runtimes shown in Listing 6.2.

The key take away from this, is that we can reduce the runtime by increasing k, even if it means that we need to increase λ to find all the pairs we want. We have also shown that the expected bottle neck will either be emitting addends or one of the

Listing 6.2: Runtime for different steps of distributed correlation computation of k-mer profiles.

Step	Spark operation	Total Time	Slowest	Shuffle Size
Collect Cross Columns	`groupByKey`	$O(NL)$	$O\left(\frac{NL}{2^k}\right)$	$O(NL)$
Emit Addends	`flatMap`	$O\left(\frac{N^2L^2}{2^k}\right)$	$O\left(\frac{N^2L^2}{2^{2k}}\right)$	–
Sum Addends	`reduceByKey`	$O\left(\frac{\lambda N^2}{2^k}\right)$	$O(1)$	$O\left(\frac{\lambda N^2}{2^k}\right)$
Calculate Correlation	`map`	$O\left(\frac{\lambda N^2}{2^k}\right)$	$O(1)$	–

two data shuffles. Chapter 8 contains experimental tests of the runtime and accuracy dependent on the tuning parameters.

6.5 Summary

In this chapter it was shown that the DISCO algorithm in [ZG13] can be generalized to real vectors. The runtime benefits from sparse and weakly correlated data. It was proven that the runtime increase linearly in the tuning parameter λ, while the error bounds decreased exponentially in λ.

This algorithm can be used to efficiently find pairs of genes that are likely to be highly identical, by exploiting the k-mer-based proxy presented in Chapter 5. It was shown that large parts if the runtime will decrease exponentially as k increases.

These findings will be used to develop an efficient algorithm for gene clustering in Chapter 7

CHAPTER 7

Proxy-based Efficient Clustering Algorithm

In this chapter we will combine the knowledge from Chapters 2 to 6 into one coherent algorithm for gene clustering. It will be called *Proxy-based Efficient Clustering Algorithm* or PECA for short. The algorithm consists of several steps as shown in Figure 7.1 and explained below.

Create k-mer profiles First we load the data, construct the k-mer profiles and distribute them so they are stored crosswise.

Approximate correlation Then we use the algorithm presented in Chapter 6 to find the pairs whose k-mer profiles are highly correlated. This creates a proxy graph, where nodes are genes and edges represents those pairs.

Align and filter For all these pairs we calculate the actual identity and filter for pairs with an identity above $id_{\min}$.
This will yield an identity graph, where nodes are genes and the edges connects genes that are more than $id_{\min}$ identical.

Clustering Based on the graph we will greedily select cluster centers, until all genes are either a center or a neighbor to one. The neighbors will then be assigned to the center it is most identical to.

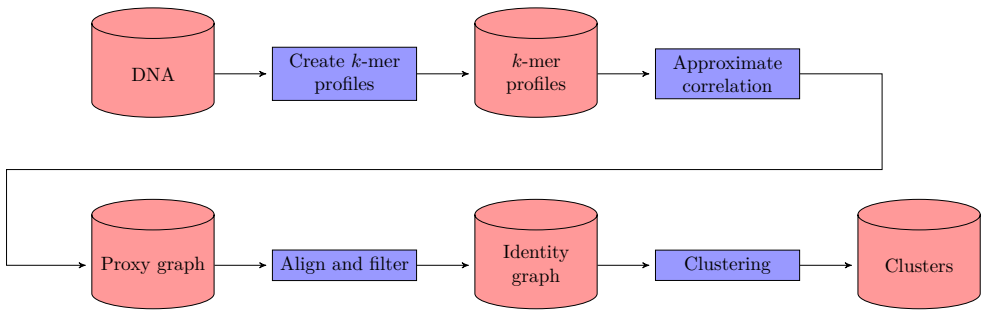


Figure 7.1: Overview of Proxy-based Efficient Clustering Algorithm.

7.1 Find Correlated k-mer Profiles

Before we begin we must load the data into memory. Each base pair in a sequence can be represented as a 2-bit value. We are therefore able to store up to 15-mers in a 32-bit signed, positive integer, which most libraries uses as vector index. We can save space by only storing the non-zero values and their indexes, since the k-mer profile is sparse.

The non-zero values are distributed as tuples on the form (k-mer, (id_{gene}, value)) so Spark can group them by k-mer.

Once the data is distributed, we run the algorithm presented in Chapter 6 to find approximate correlations for pairs of k-mer profiles. These pairs are then filtered for those with a correlation above a tunable threshold: $\min_{\text{app-corr}}$.

7.2 Calculate Identity

The raw gene data is required to calculate the identity, but when we find the highly correlated pairs the raw data is lost and we only have gene identifiers. To get the raw data to each pair we need to perform a pair lookup (Code Example 7.1), which will do two joins, i.e., shuffle the data twice.

Once we have the data we can use the Needleman-Wunsch algorithm to align the sequences and calculate their identity, and filter for pairs whose identity is above $id_{\min}$.

This leaves us with the data needed to create a graph, where the nodes are genes and the edges connects all nodes with an identity above $id_{\min}$ and has the identity as attribute. We will call this the identity graph.

Code Example 7.1: Looking up a pair of keys using Spark.

```
/**
 * Finds the values for each key in a pair of keys.
 * @param pairs The pairs of keys to lookup.
 * @param data The index from key to value.
 * @tparam V The value type
 * @return An RDD of `((id1, id2),(val2,val2))` where `(id1,id2)` is in
 *         `pairs` and `(id1,val1)` and `(id2,val2)` are in `data`
 */
def pairLookup[V](pairs : RDD[(Long, Long)], data : RDD[(Long, V)])
      : RDD[((Long,Long),(V,V))] = {
  val join1 = pairs.map(p => (p._1, p)).join(data)
  val join2 = join1.map{case (_, (p,v1)) => (p._2, (p,v1))}.join(data)
  join2.map{case(_, ((p, v1), v2)) => (p, (v1, v2))}
}
```

7.3 Find Clusters

The final step is to create a clustering according to Definition 2.1. Given a set of centers it is trivial to find the optimal assignment of non-centers, since we can just check all neighbors of a non-center in the identity graph.

The difficult part is thus to select the centers. In the ideal world we would like to find the valid set of centers, that maximizes the average IIQ for the clusters. This would be an instance of the *maximum-weight independent set problem*, if we knew the exact IIQ for each node in case it should be selected as center.

Finding the maximum-weight independent set is NP-hard. In addition, the IIQ for a given center depends on which of it neighbors are assigned to it, which in turn depends on which other nodes are selected as centers. Given that optimizing centers for IIQ has an extra layer of complexity on top of the maximum-weight independent set problem, it is very likely NP-hard as well. We will therefore settle for a greedy algorithm.

Algorithm 7.1 first calculates 2^{IIQ} for each node under the assumption, that it will get all its neighbors assigned. It then selects as centers all nodes that has no neighbors with a higher IIQ. These centers and all their neighbors are then removed from the graph. The algorithm repeats until all nodes have been removed this way. Ties of IIQ are assumed to be broken deterministically, e.g. by an unique gene ID.

7.3.1 Validity

Algorithm 7.1 finds a valid set of centers, which can be proved using these two invariants: At the beginning and end of each iteration, the set C is independent in the input graph, and it is dominant in the removed part of the graph.

Algorithm 7.1: SelectCenters(G).

$C \leftarrow \emptyset$ ▷ Set of centers
while G is not empty **do**
    ▷ *Find nodes with IIQ-potential higher than all neighbors:*
    $w[v] \leftarrow \sum_{v'} \left(\frac{id(v,v') - id_{\min}}{100\% - id_{\min}} \right)^{\alpha}$ where v and v' are neighbors, or 0 if v is isolated.
    $m[v] \leftarrow \max(w[v'])$ where v and v' are neighbors, or -1 if v is isolated.
    $C' \leftarrow \{v \mid w[v] > m[v]\}$
    ▷ *Make them centers:*
    $C \leftarrow C \cup C'$
    Remove C' and all neighbors to C' from G
end while
return C

We will prove the independence of C by contradiction. Let c, c' be two neighboring nodes in C. They can either have be added to C in the same or in different iterations. The latter is impossible, since all neighbors of C has been removed from G at the end of each iteration. For the former to be true we must have, $w[c] > m[c]$ and $w[c'] > m[c']$. This is impossible since c and c' are neighbors so $w[c] \leq m[c']$, which leads to the contradiction $w[c] > m[c] \geq w[c]$

It is trivial that C will always dominate the removed part of the graph, as we only remove nodes that are in C or neighbors to C.

As all of G is removed in the end of the algorithm we have found a set of centers C that is both independent and dominant in G.

7.3.2 Termination

The clustering algorithm will terminate if G is decreased in all iterations, which we will prove by contradiction. Let G' be an non-empty graph that will not be reduced during an iteration. G' cannot contain any isolated nodes as they all will be part of C' since $0 = w[v] > m[v] = -1$ if v is isolated. Now let $v^{\max} = \arg\max_v w[v]$. $v^{\max}$ will be unique and $w[v^{\max}] > m[v^{\max}]$, since ties are broken deterministically. $v^{\max}$ will thus be in C' which contradicts the assumption that G' will not be reduced.

7.3.3 Implementation and Runtime

Calculating w in line 5 can be done using `aggregateMessages` in Spark, where the messages are the information contribution from each edge $\left(\frac{id(v,v')-id_{\min}}{100\%-id_{\min}}\right)^{\alpha}$, which are aggregated by summation. Similarly the m (line 5) can be found using the value of the neighbor as message and aggregating by keeping the maximum. Selecting C' (line 6) is then just a simple filter operation.

Finding the neighbors of C' in line 9 can be done by aggregating boolean messages from any edge connected to a center. The subsequent remove operation is then a simple filter.

Each iteration will therefore use a constant number of `aggregateMessages` and filters. The `aggregateMessages` take $O(|E|)$ each, since the send and aggregate functions take constant time. The total time for each iteration is therefore $O(|E|)$

In theory the number of iterations could be very high. In the worst case we have $O(|V|)$ iterations:

Let the graph be a path of n nodes $v_1, \ldots, v_n$, where there is an edge between each pair of consecutive nodes $(v_i, v_{i+1}), 1 < i < n-1$. If we have $\left(\frac{id(v_i,v_{i+1})-id_{\min}}{100\%-id_{\min}}\right)^{\alpha} = \frac{1}{2^i}, 1 < i < n-1$ then we have $w[v_i] > w[v_{i+i}], 2 < i < n-1$ in the first iteration. We will have $C' = \{v_2\}$ and remove $\{v_1, v_2, v_3\}$. In the t-th iteration we will likewise have $C' = \{v_{3t-1}\}$ and remove $\{v_{3t-2}, v_{3t-1}, v_{3t}\}$, until there is less than three nodes in the graph.

That theoretical worst case is however extremely unlikely. In most cases we will see very few iterations, where the first runs will remove the majority of the graph, making subsequent runs that much faster.

7.4 Variations

This section will propose a few variations to PECA. Each represent an idea or a trade of that influences runtime or accuracy. Figure 7.2 shows how they fit in with the steps of the normal version of PECA. The proposals are evaluated in Chapter 8.

7.4.1 Proxy Centers

Aligning genes to compute their identity is expensive. In PECA, we compute the identity for all pairs whose correlation is above the cutoff, but it is possible to reduce that. If we have a set of centers, we can calculate only the identity for pairs, which contain a center.

To do this we must be able to find a good set of centers, without knowing the identity between genes. We do however know the correlation between them, which we can use as an approximation for the alignment, if we assume there is a monomial mapping $m(x) = x^\beta, 0 < \beta$ from the range of valid correlations $[corr^{min}, 1]$ to the range of valid identities $[id^{min}, 100\%]$:

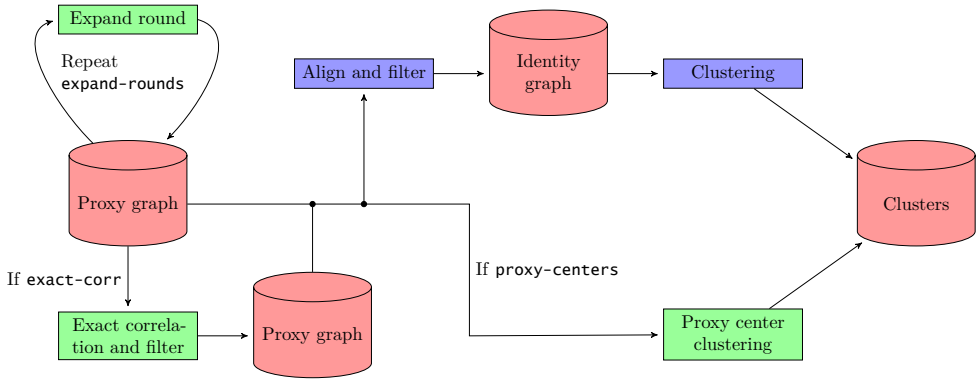


Figure 7.2: Overview of the variations to PECA. Variations are marked with green, and steps leading to the proxy graph are omitted.

$$id(g, g') \approxeq id^{min} + (100\% - id^{min}) \left(\frac{corr(g, g') - corr^{min}}{1 - corr^{min}} \right)^{\beta} \tag{7.1}$$

$$\frac{id(g, g') - id^{min}}{100\% - id^{min}} \approxeq \left(\frac{corr(g, g') - corr^{min}}{1 - corr^{min}} \right)^{\beta} \tag{7.2}$$

$$\left(\frac{id(g, g') - id^{min}}{100\% - id^{min}} \right)^{\alpha} \approxeq \left(\frac{corr(g, g') - corr^{min}}{1 - corr^{min}} \right)^{\alpha\beta} \tag{7.3}$$

Applying this assumption we can estimate the IIQ of nodes and select centers before we calculate identities. The estimation is naturally coarse, but this is probably OK, since Algorithm 7.1 is already selecting centers greedily based on an upper bound for the IIQ, and not the actual, final IIQ.

A few changes needs to be made to Algorithm 7.1, if it takes a proxy graph as input. Apart from slightly changing the calculation of w, we must change the way we remove nodes (line 9). We cannot just remove all neighbors to centers, based on their correlation. We must only remove those, whose identity is above the minimum identity, to ensure that the set of centers are dominating all removed genes with respect to the identity graph.

7.4.2 Exact Proxies

One disadvantage of the approximate calculation of proxy correlations is that we need to lower the threshold for which pairs we compute identity. Otherwise we might miss out on pairs that have their correlation underestimated. If we did not have that uncertainty we could raise the threshold and reduce the number of alignments we must make.

To get rid of the uncertainty we can compute the exact correlation for each pair, before we compute the alignments. We will then end up with three thresholds, a threshold for approximate correlation $corr^{min}_{approx}$, a threshold for exact correlation $corr^{min}_{exact} > corr^{min}_{approx}$ and one for the actual identity id^{min}, and the process will be like this.

1. Calculate approximate correlation for all pairs, and filter for $corr \geq corr^{min}_{approx}$.
2. Calculate exact correlation for those pairs, and filter for $corr \geq corr^{min}_{exact}$.
3. Calculate identity, and filter for $id \geq id^{min}$.

It is far from guaranteed that the extra step will improve runtime. It is determined by how many pairs of genes we can avoid calculating the identity for, which depends a lot on the difference between $corr^{min}_{approx}$ and $corr^{min}_{exact}$. This is tested in Chapter 8.

7.4.3 Expansion Rounds

In general we want to with a very high probability find all pairs of genes with a high identity. This forces us to use a low cutoff value for the proxy correlations, which must be even lower to deal with the uncertainty as mentioned above.

Many of the pairs do not exist in isolation. On the contrary many genes will be part of a cluster, where they are not only very identical to the center, but also many of the other cluster members.

Imagine a cluster with a center and 10 additional members. The identity between all members are above the threshold $id_{\min}$. In other words the cluster is a complete clique in the identity graph. Lets assume that we have a 10% chance of not finding any given link, due to the randomness when calculating correlations. We will therefore expect to miss one of the cluster members, which is bad.

The center is however not only directly connected to each member. It is also connected via paths through each of the other members. Hence there is 9 more ways of length 2 for each member to reach the center. The chances that we find non of these ways decrease exponentially. In this example it is $\frac{1}{10} \cdot (1 - \frac{9^2}{10^2})^9 = 3.23 \cdot 10^{-8}$.

This is a really significant improvement, and it makes us able to deal with a relatively large uncertainty when finding proxy correlations. It does however require that the genes exists in clusters that are somewhat interconnected.

7.5 Summary

In this chapter we have gone through Proxy-based Efficient Clustering Algorithm (PECA). It consists of three steps:

1. Find pairs of genes with correlated proxies.
2. Calculate the identity of those pairs.
3. Select cluster centers and assign genes to them.

We were also introduced to three variations to PECA: proxy centers, exact proxies, and expansion rounds.

The algorithm was designed using knowledge from Chapters 2 to 6. The next chapters will perform tests and evaluations of PECA (Chapter 8) analyze the results and discuss further improvements (Chapter 9).

CHAPTER 8

Tests

In this chapter we will go over the series of tests that have been conducted to evaluate PECA and its variations. The first two sections will analyze tuning parameters and algorithmic variations, so we get the best configuration when conducting benchmarks. Following that, PECA will be compared to the two existing solutions: CD-Hit and UClust. Then we will analyze how well PECA scales with input data and how well it parallelizes on multiple servers.

The results of the tests will be further analyzed and discussed in Chapter 9.

All these tests being executed on Amazon Web Services. The data was stored in the Amazon Simple Storage Service (S3) and computation was done on the Amazon Elastic Compute Cloud (EC2) instances of the type `r3.8xlarge` which has 32 CPUs, 244 GB RAM, and 640 GB on SSD disks. This instance was selected to give Spark the best opportunity to keep cached data in RAM, and to have high-performance disks in the case that Spark would need to spill data onto the disk.

The input data for the tests was a sample of the gut microbiome from 253 persons from the Human Microbiome Project [Met+12]. The full data set contains a little more than 32,000,000 genes. This has been randomly subsampled to a series of smaller datasets from 2000 genes in the smallest and doubling in size up to 8,192,000 genes.

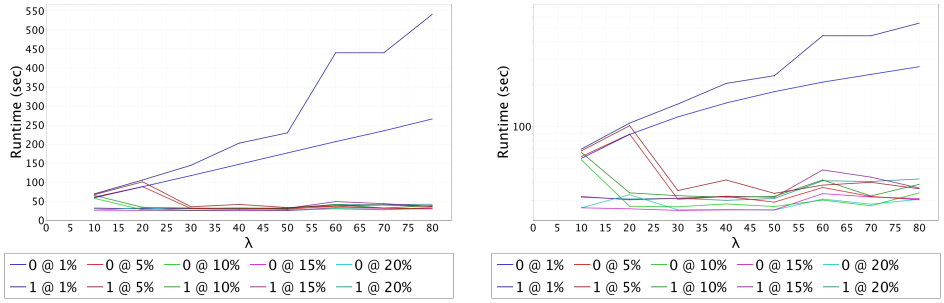
8.1 Tuning the Computation of Approximate Correlations

The first test that was conducted had the goal of finding the best tuning parameters for the approximate correlation computation. From the research conducted in Chapter 5 we know that it will be best to use $k = 15$, but we do not know the best λ, cutoff values, or whether or not to run expansion rounds.

To test this we ran the approximate correlation computation three times for different configurations, for which we varied λ from 10 to 80, the cutoff value from 1% to 20%, and tried with and without an expansion round. The execution included removal of false positives by calculating the actual identity for all pairs of genes with a correlation above the cutoff value.

The tests were executed in a cluster with four worker nodes, on a data set with 128,000 genes.

Figure 8.1 shows how the runtime developed across the different configurations. The most apparent thing is that the lines for 1% cutoff value skyrockets while the



(a) Linear (b) Log

Figure 8.1: The runtime in seconds for different configurations.The legend should be interpreted as `[NumberOfExpandRounds]@[CutoffValue]`.

other stays mostly flat. The lines for the 5% cutoff follows the same path initially, but drops down once $\lambda > 20$. The conclusion is that the cutoff value should always be above $\frac{1}{\lambda}$. If not we will get a very large number of false positives, that is genes for which the proxy is above the cut of value but the identity is not above $id_{\min}$. That will require us to use a huge amount of time to calculate the gene identities when removing false positives.

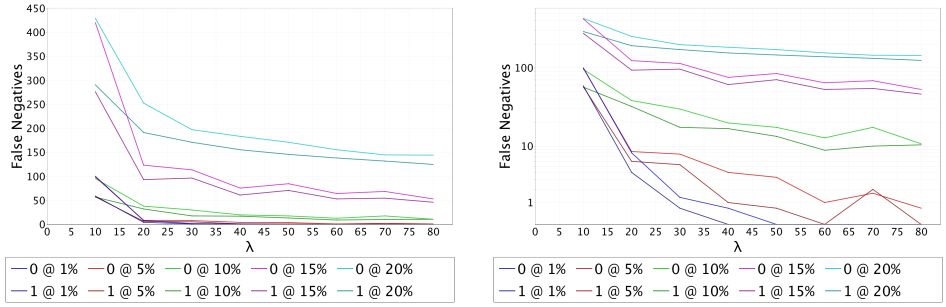
These false positives comes from the fact that a single pair of genes only needs to emit a single value to be above the cutoff. Even though the probability of this happening is pretty low, the number of pairs are so huge, that the expected number of times this will happen becomes large. When the cutoff is above $\frac{1}{\lambda}$ any parent will need at least two emissions to be above the cutoff. This becomes so unlikely that it does not present a problem, even with a huge number of pairs.

It is peculiar that the execution time seems to be constant even though we increase λ. This could indicate that the used data size is so small that the majority of the time you spend doing overhead work. This will be looked into in Section 8.4, but for now we will ignore that and base the conclusions on the data at hand.

The last thing we need to analyze is the number of false negatives. We will like that to be as close to zero as possible for two reasons: first, there is no simple way to correct false negatives, and second, any false negatives might break the guarantees on cluster properties that we presented in Chapter 7.

Figure 8.2 shows the number of false negatives. For scale the number of true positives is 16,010. First of all it is clear that our results gets better as we increase λ. It is also clear that having one expand round gives us a better result. We can however also achieve the same improvement by increasing λ. If we cross reference with Figure 8.1, we see that it is more efficient to increase the accuracy by increasing λ rather than adding an expand round.

In Chapter 5 we found the cutoff value of 10% or 20% would be suitable, but that



(a) Linear (b) Log

Figure 8.2: Plots of the number of false negatives for different configurations. The legend should be interpreted as `[NumberOfExpandRounds]@[CutoffValue]`.

did not take the inaccuracy of the approximate correlation computation into account. These inaccuracies causes the two cutoff values have a significant number of false negatives, as seen in Figure 8.2. We are only able to achieve a negligible number of false negatives, when the cutoff value is either 1% or 5%. We get slightly more false negatives at 5%, but not enough to justify the increased runtime that comes with the cutoff value of 1%. The false-negative rate is still lower than $\frac{1}{16{,}010}$ when $\lambda = 80$.

In conclusion these experiments shows that the optimal configuration will be $\lambda = 80$, CutoffValue = 5%, and not to do any expand rounds. This configuration will be used for the remaining tests.

8.2 Comparing Algorithmic Alternatives

Having found good tuning parameters we need to figure out which of the algorithmic alternatives will perform the best. This both include runtime as well as the IIQ of the produced cluster assignment.

This was tested by running a data set of 1,024,000 genes through PECA in three different configurations.

nomods without any of the variations introduced in Chapter 7,

exact-corr where we calculates the exact correlation and increases the cutoff value to 10%, and

proxy-centers where we find cluster centers based on the proxy and delay identity computation.

If we look at the IIQ in Figure 8.3 we will see that they perform almost identical. The proxy-centers finds a cluster assignment that is slightly worse than the two other,

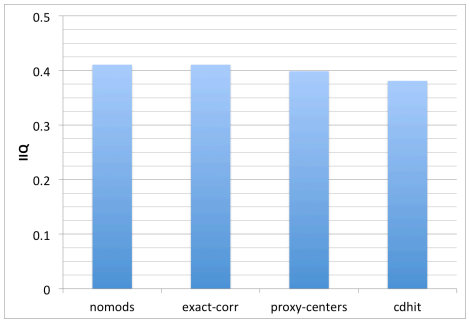


Figure 8.3: IIQ for the different algorithmic variations. The tests were performed on a data set of 1,024,000 genes. CD-Hit has been included for reference.

but it is still better than that found by CD-Hit. It is worth noting that CD-Hit has a drawback, since it uses slightly different alignment parameters, but we would get back to that later.

Having practically identical quality the most important difference becomes runtime as shown in Figure 8.4. It is obvious that finding cluster centers based on the proxy kills performance. We would expect clustering to take a little longer, since we have delayed the identity computation to the clustering iterations. But the large difference we see is unexpected.

When we look closer at what goes on in the clustering (Figure 8.5) we see that each round of clustering takes a little longer as we would expect, but that the real difference is that we have many more rounds of clustering, when we do it based on proxies. The current clustering algorithm does not deal well with the condition that

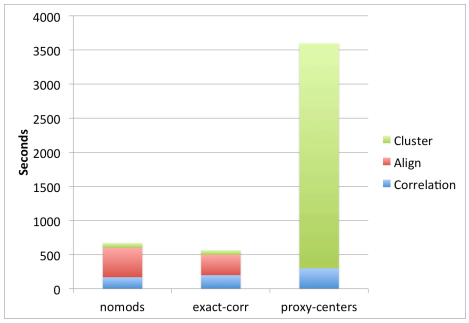
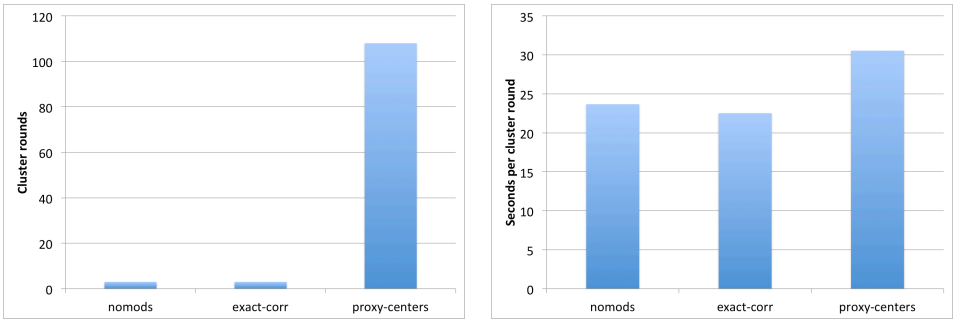


Figure 8.4: The runtime for the three different variations, divided into the time spend on calculating correlations, removing false positives by analyzing gene pairs, and, dividing genes into clusters. The tests were performed on a data set of 1,024,000 genes.



(a) Number of rounds (b) Time per round

Figure 8.5: The time spent on clustering broken up into a number of rounds and time around.

some genes are connected in the graph when they do not have high enough identity.

Having ruled out clustering based on the proxy, the choice stands between the two remaining variations. The central question is: *is it worth calculating the exact correlation, to be able to apply a higher cutoff value?* Figure 8.4 clearly shows that the reduction in alignment time is far greater than the extra time spend on calculating exact correlations.

For the remainder of the performed tests we will have to use the "exact-corr" version of PECA.

8.3 Scalability and Comparison with CD-Hit and UClust

There are two main properties to analyze when analyzing distributed algorithms: scalability and parallelizability. Scalability is how much slower the algorithm will run when the data size increases. Parallelizability is how much faster it will run when we executed on more cores or servers.

We will handle parallelizability in the next section. Scalability will be handled here along with a comparison with CD-Hit and UClust.

If we look at the IIQ (Figure 8.6) we can see that PECA and UClust performs equally well. CD-Hit has a slightly lower IIQ, but that is likely because it uses slightly different alignment parameters. The IIQ was calculated based on the alignment parameters used by UClust, so they yields biologically more accurate alignments [Edg15a]. It is not possible to configure the alignment parameters for CD-Hit, but it would be reasonable to expect the IIQ to be at least as good as UClust, if that was possible.

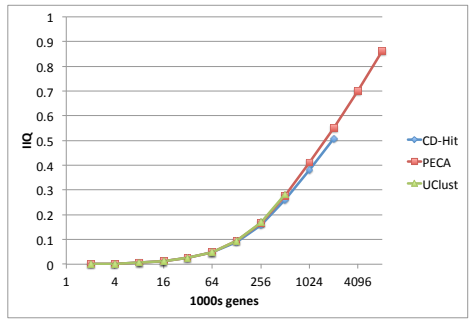
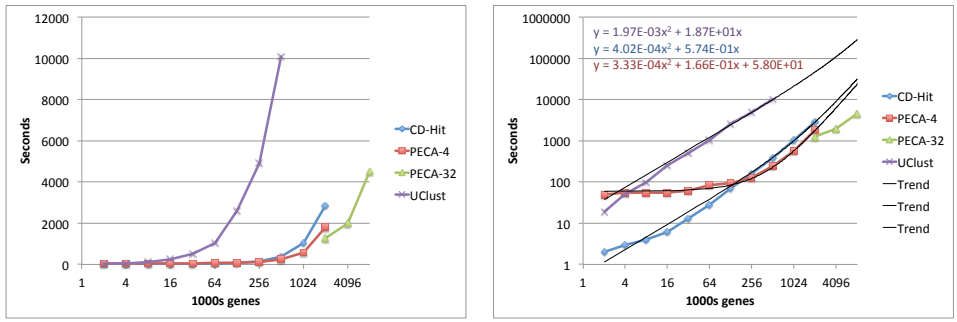


Figure 8.6: IIQ comparison between UClust, CD-Hit, and the Spark algorithm, for different data sizes.



(a) Linear

(b) Log with trend lines

Figure 8.7: Runtime for PECA, UClust, and CD-Hit, for different data sizes.

The runtime benchmark in Figure 8.7 is based on executions on different hardware. The reason for this is that CD-Hit and UClust supports limited or no parallelizability respectively, and we want to show the advantages of parallelization. PECA has been run on 4 and 32 servers with 32 cores each. UClust does not support any kind of parallelism so it was executed using a single core. CD-Hit was executed on a single machine with 32 cores. It is supposed to be able to distribute the computation among several servers, but that functionality is broken in the current version.

The most apparent thing is that Apache Spark comes with a high level of overhead. For small datasets it is vastly outperformed by CD-Hit. As data sizes grow the overhead becomes a negligible part of the total runtime, and PECA stops performing slightly better than CD-Hit. A closer look into the data from 512 to 2048 genes reveals that it performs 1.68 times better on average, using four servers instead of one.

UClust suffers heavily from the lack of parallelization and runs 40 times slower

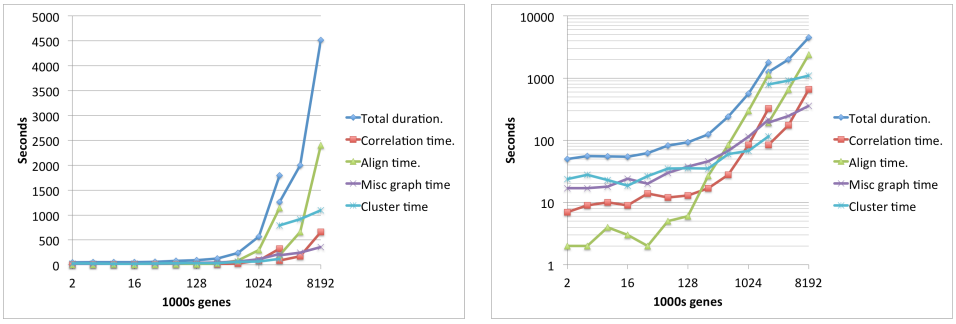
than PECA. In the other end of the spectrum we see that the runtime of PECA is significantly improved when using 32 servers instead of 4. We will take a closer look at that in the next section.

We can analyze the scalability of the three programs in more detail by looking at the equations for the trend lines. The most important thing is that all three programs has a significant square part. Even though the coefficient is quite small it will dominate the runtime for larger datasets. CD-Hit and PECA have very similar coefficients with a slight edge to PECA. Both are significantly better than UClust. This means that the ranking of the programs will remain the same as the data size increases even further – given this hardware setup.

8.4 Details of Scalability

In order to understand why PECA scales as it does, the different steps in the algorithm has been plotted in Figure 8.8. It is very clear that the biggest scalability problem is the time spent on aligning genes. Even though the alignment time is cut dramatically when using 32 servers, it quickly becomes the dominant part when the data size keeps increasing.

Correlation time also scales pretty badly, but takes approximately a quarter of the time spent on alignment for larger data sizes. For data sizes below 256,000 the correlation time seems dominated by a constant part. Recall that Section 8.1 found that λ did not have much influence on the runtime. This data shows that it was likely because the runtime was dominated by constant factors. At larger data sizes λ will probably have the expected linear influence on the runtime.



(a) Linear (b) Log

Figure 8.8: Execution time divided into the main parts of PECA as the data size increase. From 2000 to 2,048,000 genes PECA was executed on four servers. From 2,048,000 to 8,192,000 genes PECA was executed on 32 servers. This is the reason for the jump in the chart lines.

It is difficult to do much about the alignment time as long as the clustering algorithm handles proxy centers as poorly as it do now. It might be possible to improve it by using a native implemented alignment algorithm, but that will only give us a constant speed up which will not help a lot on the scalability issue.

If we on the other hand could improve on the clustering algorithm, so it would efficiently select centers based on proxy information only, then we would be able to cut away gene alignment completely. This will remove the worst scaling part, cut away up to 80% of the execution time. The correlation computation will then be the most time-consuming part, but as opposed to alignment it is possible to trade-off correlation speed for correlation accuracy using the λ parameter.

8.5 Parallelizability

Now that we have gotten a good idea of how PECA scales with larger datasets, the next step is to understand how it parallelizes to multiple servers. To do this the data size was fixed on 2,040,000 genes and the number of servers increased from 2 to 32. The overall results are shown in Figure 8.9.

With a low number of servers the parallelization is almost perfect: it takes half the time with twice the amount of servers. But that effect quickly flattens and the runtime becomes even slower when reaching 32 servers. To understand that we need to look at the runtime for each part of PECA. This is done in Figure 8.10.

Let us start by looking at alignment time and correlation time, since those are the the parts that scales worse when the data size increases. These two parts parallelizes perfectly at least up to 16 servers. At this point the correlation time seems to hit a floor. If we take a look back on Figure 8.8 we see the correlation time only doubles from 2,048,000 to 4,096,000 genes as opposed to quadrupling as we would expect. Knowing this, we can conclude that the reason we do not see better parallelization is that 2,048,000 genes is not enough to benefit from 32 servers.

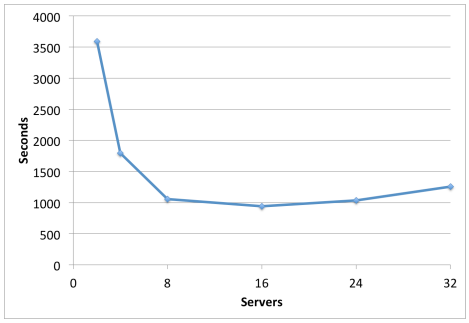
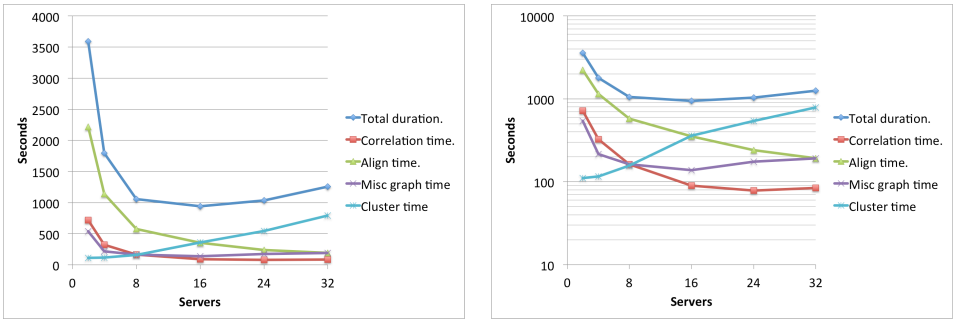


Figure 8.9: Overall runtime on different number of servers. The test was executed on 2,048,000 genes.



(a) Linear (b) Log

Figure 8.10: The runtime divided into different parts of PECA, as the number of servers increases. The test was executed on 2,048,000 genes.

Alignment time continues to drop all the way up to 32 servers. The rate is almost perfect, but we do begin to see a slight flattening caused by some overhead. All in all we can conclude that correlation time and alignment time can be efficiently reduced by increasing the amount of machine power.

That can however not be said about the clustering time. It increases linearly in the number of servers, which indicates that the runtime is dominated by overhead time. This will be analyzed in greater detail in Section 8.6.

8.6 Details on Clustering

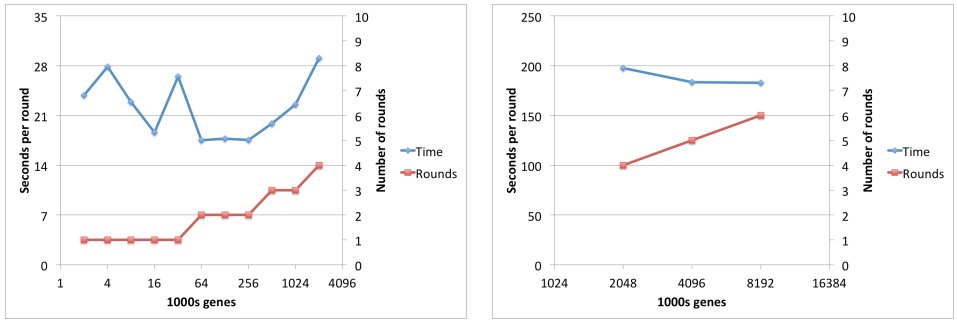
This section will look into why the clustering parallelizes so badly.

We would expect the time spent in each clustering round to be constant, if the hypothesis is correct that the clustering time is dominated by parallelization overhead. Figure 8.11 sheds some light on that. The data is too volatile, as long as there is only one cluster round, but as soon as we have more than two rounds it stabilizes. The round time is constant, for the smaller data sizes on 4 servers and on all executions on 32 servers. For larger data sizes it scales linearly.

This shows that the runtime is indeed dominated by parallelization overhead. The effect of the data size is so small that it only shows at ratios between data size and number of servers, where alignment and correlation dominates the overall runtime.

The clustering part is unique in the way that it is the only part of PECA that is iterative. Dynamic iteration (where the number of iterations is unknown) is implemented in Apache Spark by launching a job for each iteration. Thus there will be synchronization between the driver program and the worker nodes between each iteration.

To analyze the effect of that synchronization Figure 8.12 shows how much of the clustering time that is spend waiting on the synchronization in between jobs. Even



(a) 4 servers (b) 32 servers

Figure 8.11: The number of clustering rounds and average time per round. Note that they have separate y-axes.

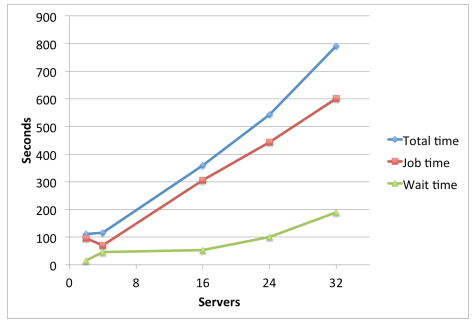


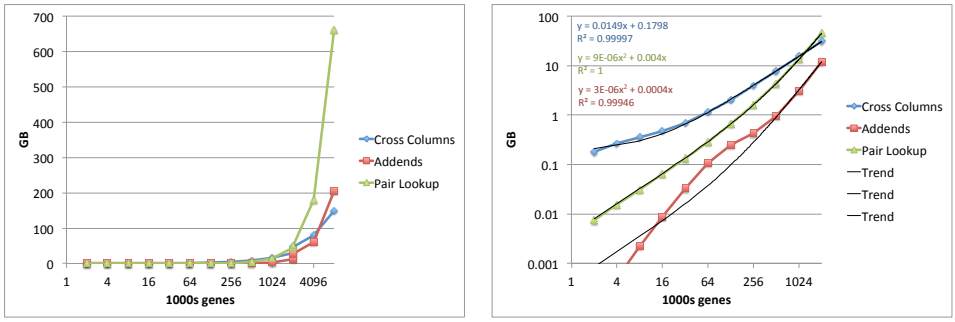
Figure 8.12: Time spent on clustering broken down into time spent working on jobs and time spent waiting on the Spark framework between jobs. The test was done on 2,048,000 genes.

though the cost of synchronization increases with more servers, the majority of the time is still spent by the worker nodes performing jobs. The poor parallelizability is therefore not caused by iterating.

Another interesting fact presented by Figure 8.11 is that the number of cluster rounds scales logarithmically in the number of genes. That is much better than the theoretical worst-case of linear scalability.

8.7 Shuffle Size

The three steps with the largest shuffling size are collecting cross columns and shuffling addends, which both are part of the approximate correlation computation, and third the pair lookup performed when calculating exact proxies.



(a) Linear (b) Log

Figure 8.13: Shuffle size for three stages of PECA: Collect cross columns, shuffling addends, and the pair lookup of k-mers.

We would expect the collection of cross columns to scale linearly in the data size, and Figure 8.13 clearly shows that this is the case.

The shuffling of addends will increase linearly in the sum of the correlation between all pairs of genes. If we assume that the average correlation between a pair of two genes does not depend on the data size, then the sum of correlations will increase quadratically in the number of genes. Figure 8.13 confirms that assumption, and we see how the quadratically scaling size of the addends changes from being insignificant to be larger than the size of the cross columns. The number of addends can of course be tweaked by changing λ.

This shuffle size for the addends is however not as large as the shuffle size when performing the pair look up k-mer profiles when calculating the exact proxies. This will naturally scale linearly in the number of pairs found in the approximate correlation computation. If there is a constant probability that the k-mers of any two genes have a correlation above the cutoff value, then the number of pairs we find will scale quadratically. This seems to be the case, as shown in Figure 8.13. The number of pairs found can naturally be reduced by increasing the cutoff value. Shuffle size might also be decreased by shuffling raw gene sequences instead of the k-mer profiles, but more on this in Chapter 9.

The steps with the largest shuffle size are the approximate correlation computation and the distribution of k-mers for the exact correlation computation. The development of the shuffle size as the number of genes increases are shown in Figure 8.13.

The tests revealed a peculiar behavior of the shuffling performed when aligning genes. The shuffle size seem to scale linearly with the number of servers, see Figure 8.14. An increase that large cannot be explained by the reduced data locality, i.e., that less of the required data is available on the local worker machine. This operation was implemented using the `aggregateMessages` transformation from the GraphX library. This unexpected increase in shuffle size might therefore be related

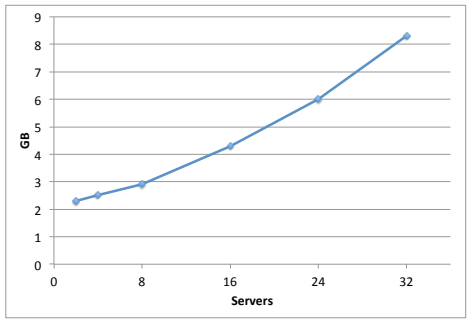


Figure 8.14: Shuffle size when aligning genes. This is performed as an `aggregateMessages` transformation. The test was done on 2,048,000 genes.

to the poor parallelizability of the clustering rounds. This will be discussed further in Chapter 9.

8.8 Summary

This chapter presented a series of tests that evaluates PECA. It was compared with existing tools and the parallelizability and scalability of each part was analyzed.

The most important findings are:

- PECA performs 1.68 times better than CD-Hit and 40 times better than UClust, using four servers instead of one.
- The algorithm for finding cluster centers performs very poorly when selecting centers based on proxies.
- Aligning genes takes up the majority of the runtime.
- The clustering steps parallelizes very poorly; the other major steps have great parallelization.

The consequences of this and especially what can be done about the problems are discussed in Chapter 9.

CHAPTER 9

Discussion

This chapter will further analyze and discuss the results found in Chapter 8.

It will start with a comparison between PECA and CD-Hit and UClust. It will then discuss the series of suggestions for improving the performance of PECA. Some of these improvements will be simple to implement, others will require efficient algorithms to solve a subproblem. This will be followed by an analysis of a major scalability problem preventing the tests from dealing with even larger datasets. The findings of the sections will be combined as the future prospects of PECA will be assessed.

Finally the experiences made with Apache Spark during the work of this thesis will be discussed, and the framework will be assessed as an easy to use and powerful alternative to traditional computing.

9.1 Comparison with Existing Tools

Our tests showed that there is practically no difference in the quality of the clustering produced by PECA and the existing tools: UClust and CD-Hit. This is somewhat surprising since PECA strives to optimize IIQ when choosing center genes. The system tools both use length and are not aware of the IIQ measure.

There could be a few reasons for this. Length could be a practically perfect proxy for the quality of a center. But it is also possible that this data has a practically unambiguous way of defining clusters. It will be interesting to repeat the tests on different datasets for which we know that the clustering is ambiguous.

In terms of runtime we clearly outperformed UClust, which is impeded by being single threaded. The runtime is also significantly better than CD-Hit, but that have to be viewed in the light of CD-Hit only used one instead of four machines. That being said we do not know how well CD-Hit performs when distributed to multiple machines.

When taking into account that CD-Hit is a mature and thoroughly optimized tool, the current performance is pretty good for an algorithm developed and implemented during a master thesis. There is likely still plenty of room for improvements that will increase performance. We will take a look at some opportunities for this in the next section.

9.2 Performance Improvements

The performance of PECA can be improved in several ways. Some are simple implementation changes while others require more work. The two parts of PECA that consumed the most resources were alignment and the clustering. In a moment I will explain that time spent on alignment is best reduced by improving the clustering. Then we will look into ways of improving the clustering. But first we will go through some of the simpler improvements.

9.2.1 Simple Implementation Improvements

The implementation of PECA for this thesis has not been aimed at production use. Some implementation details, that might increase performance, have been omitted due to time constraints. These improvements have low scientific interest, but are still likely to increase performance.

Apache recommends users of Spark to switch to the Kryo serializer [Sof15] instead of the Java serializer which is the default [Fou15h]. Kryo runs 10 times as fast and produces serialized objects that are a third the size of those produced by the built in Java serializer [Smi15]. One way Kryo improves efficiency is by not writing the entire class name for every object, as the Java serializer does. That will be very efficient in the case of PECA, since especially the distributed correlation computation uses a huge number of tiny tuple objects.

We can also reduce the size of the tuples we shuffle during the distributed correlation computation. Each gene currently has a `long` as identifier. We can reassign them a unique ID and use an `int` instead. If we did not normalize the k-mer profiles we could store the values as `int` instead of `double`. The addend function will then need to know the length of each gene. Storing the length and the value in each tuple will take up exactly as much space as the normalized value. A smarter way would be to broadcast the lengths to all servers. Apache Spark can perform this operation efficiently. Overall it would be more efficient, since the number of k-mers per gene vastly outnumber the number of servers.

Calculating exact correlations before performing gene alignment quadruples the shuffle size for those tasks. This is because the k-mer profiles for each pair are shuffled instead of the raw gene sequence. Each nucleotide only use two bits while each nonzero position in the k-mer profile use up 96 bits. It might therefore be more efficient to shuffle the raw gene data and recalculate the k-mer profile. Benchmarking will have to confirm that this actually leads to improved performance.

All these enhancements are not particularly complicated to implement, and they will likely help to improve the performance of PECA. But they do not tackle the two major resource consuming tasks, which are the clustering steps and gene alignment. We will now therefore investigate further how to improve these steps, starting with alignment time.

9.2.2 Reducing Alignment Time

The Needleman-Wunsch algorithm is over 40 years old and is still considered the most efficient way of aligning genes. It is therefore unlikely that we will be able to improve that. We therefore need to reduce the number of pairs we align, if we want to reduce time spent aligning genes.

Some reduction could be achieved if the cutoff threshold was raised. This would however also mean an increased risk of false negatives. At the current cutoff value there is only 40% false positives. It is therefore impossible to get a speedup of just a factor of two, even with a perfect cutoff value.

The reduction would be much greater if we could find centers based on the proxy and only align genes with centers.

9.2.3 Efficient Algorithm for Proxy-based Center Selection

The current algorithm for selecting centers based on the proxy is performing horribly, so we need something better. Stated formally we need an efficient solution to Problem 9.1. The problem definition assumes that we are able to define a function, which given the proxy value for a pair of genes returns the probability of them being more than $id_{\min}$ identical. That should be straightforward to do based on a statistical analysis of the proxy related to that done in Chapter 5.

Problem 9.1: Proxy-based center selection.

Let V be a set of vertices, E a set of edges. E is unknown, but we know a set $E_p \supseteq E$ of proxy edges.

Let the two functions $f\colon E_p \to [0,1]$ and $g\colon E_p \to \{\texttt{true}, \texttt{false}\}$ be defined as follows. $f(e) = \Pr[e \in E]$ returns the probability that e is in the set E, and $g(e) = \texttt{true} \Leftrightarrow e \in E$ returns `true` if and only if e is in E.

Calls to g are expensive and must be minimized.

Unweighted problem Given V, E_p, f and g find the minimum subset $C \subset V$ that is dominant in the graph (V, E).

Weighted problem Given V, E_p, f, g and a weight function[†] $w\colon V \to \mathbb{R}^+$ find the subset $C \subset V$ that is independent and dominant in the graph (V, E) and maximizes the average weight $\frac{1}{|C|}\sum_{c \in C} w(c)$.

[†] This is a simplification compared to the effort to maximize IIQ. The IIQ for each center is unknown, while the weights in this problem are known.

Literature dealing with distributed algorithms for selecting weighted maximal independent sets or minimal dominant sets could help with inspiration, but these algorithms cannot be applied directly. They are not designed for the situation where only E_p is known and calls to g must be minimized.

The tests showed that the poor performance was due to a high number of clustering rounds. That must mean that only a few genes are removed from the graph in each round. The number of removed genes depends on the number of centers and the number of members assigned to each center.

In general when the density of the graph increases, i.e., it gets more edges, those factors are influenced in two ways. The number of centers will drop as the larger number of neighbors will make it less likely for a gene to have the highest priority. On the other hand, the number of members assigned to each center will increase as well. These two effects will likely cancel each other out.

In the case of the proxy-based clustering a large number of edges are added to the graph, for which the proxy is high enough, but not the actual identity. These edges will only lead to a decrease in the number of centers we can select, but not increase the number of members per center.

A way to fix this could be to initially select centers without guaranteeing that they are independent. We then need to test the identity of two centers, if they are neighbors in the proxy graph. One will then become the member and the other the center if and only if the identity is above the threshold. Otherwise they will both remain centers.

The possibility of breaking neighboring centers will give us more flexibility in deciding an algorithm to select centers. One way could be to just ignore edges with a low proxy value when finding the maximal priority among neighbors. A probabilistic variation of that would be to include each edge e with probability $f(e)$ in a temporary subgraph for each round. Centers will then be selected based on that temporary subgraph.

An third way could be to let each vertex choose only the edge with the highest proxy value, and then compare only along those edges. Choosing one edge per vertex would create a forest of trees in which exactly one edge has been chosen twice. That fact could also be used to select centers, for example by choosing one of the vertices connected to that edge that has been chosen twice.

In all three suggestions it will be required to check the centers selected in each round and and break up any neighboring centers by revoking one of them.

These are just a few ideas for efficient solutions to Problem 9.1, but developing a complete solution will have to be done in future works.

9.2.4 Improving the Implementation of the Clustering Algorithm

The overall performance can also be improved by fixing the parallelization problems of the clustering steps. A quick and dirty solution could be to reduce parallelization of the clustering. Apache Spark does not have a good support for running some part of an application on fewer servers than another part. To work around this we could run PECA as two separate Spark applications, and store the intermediate results on disk or in a remote storage system.

That solution is not very elegant and it does not really addresses the core part of the problem. We will not be able to distribute the clustering when the data size increases. It will be preferable to remove the penalty of parallelization.

Unfortunately detailed data about the cluster steps was not logged during the tests. The alignment part of PECA was however also implemented using the GraphX library, and we got some detailed data on that step. As shown in Figure 8.14 the amount of data shuffled increases much more than what can be explained by each machine having a smaller part of the data. This was a clue that the GraphX library might create additional data.

A review of the source code of GraphX revealed that `aggregateMessages` did some auxiliary work behind the scenes. Part of that work was building a RDD of references between any pair of an edge partition and a vertex partition. That RDD contained an element for every pair even if there were no references between them, i.e., P^2 elements where P is the number of partitions. P is normally the number of cores multiplied by a small constant. The work involved in creating and shuffling that RDD could very likely be the cause of the poor distribution.

I have filed a bug with GraphX describing the problem [Ber15]. It should definitely be possible to fix the issue so hopefully future versions of GraphX will be able to be distributed without severe overhead penalties. Until that happens it will also be possible to reimplement the clustering algorithm using normal RDDs.

9.3 Scalability Problems

The observant reader might have noticed that none of the tests were executed on the full data set of 32,000,000 genes. Such executions were attempted, but they all failed before completion. The execution time for each step increased much more than expected, and at some point during the distributor correlation computation the connection to the worker nodes began to time out. Increasing the timeout to 10 minutes did not solve the problem.

Reviewing the logs and monitoring data revealed that the problem was likely caused by garbage collection. The test were executed using JDK 7, which uses concurrent mark and sweep as its default garbage collection algorithm [Ora15]. That algorithm includes execution pauses, so called stop-the-world pauses, in which all user threads are stopped. The duration of these pauses increases linearly in the number of live (non-garbage) objects.

In extreme cases these pauses become longer than the network timeout limits. Connections will therefore timeout since the code responsible for responding is paused because of a long garbage collection. This is probably the cause of the timeout problems when dealing with the full dataset.

9.3.1 The Java Garbage Collector

The Java memory is divided into a young and old generation to optimize performance. New objects are created in the young generation. When the young generation is full, a minor garbage collection executes only on the young generation. Typicality most objects have a very short lifespan, so the amount of live objects in the young generation is expected to be small. Garbage collection will therefore be fast while at the same time cleaning up a large number of garbage objects.

Objects that survived several garbage collections in the young generation are moved to the old generation. When that is full a full garbage collection occurs. It will scan all live objects and is therefore a slow process.

If an application quickly creates a huge amount of objects it might fill up the young generation within the lifespan of those objects. The minor garbage collection will then be inefficient and a larger fraction of the objects would be moved to the old generation. This means that a full garbage collection will be executed much more frequently.

This is probably what happens during the approximate correlation computation. Every nonzero value is wrapped in a tuple object. Even though their lifetime is limited they might be able to fill up the young generation faster due to their vast number. A full garbage collection will therefore have to occur frequently.

An other possibility is that the concurrent mark of live objects is not completed when the heap is full. In that case execution will be paused until object marking is completed. This has happened to other users of Apache Spark [KY15].

To make things worse these objects are large in numbers but small in size. This increases garbage collection time which scales linearly in the number of objects not the size of those objects. A large number of small objects is therefore the worst-case for the garbage collector.

There are two major things that can be done to reduce garbage collection time. The garbage collector can be tuned and the number of objects can be reduced.

9.3.2 Tuning the Java Garbage Collector

The concurrent mark and sweep garbage collector was designed for applications with at most a few GB of RAM and not server applications with several 100's GB of RAM. The default parameters controlling the garbage collector is therefore not optimal for large-scale applications.

Plenty of documentation on this subject exist [Ora15; WH15; KY15; Lee15]. The process is a lengthy one involving experiments and a good amount of learning by trial and error.

For the problem at hand the process would be very time-consuming and financially expensive, since the test were executed on 32 rented, large servers and the problem did not reveal itself the first 30 to 90 minutes. Tuning a garbage collector is of practically no scientific value for this thesis. The tests on smaller datasets gave a sufficient results

to do an analysis of PECA. It was decided not to experiment with garbage collection tuning due to the high cost and low gains.

An alternative to tuning parameters for the concurrent mark and sweep algorithm is to choose an other algorithm. Oracle has introduced a new algorithm called "Garbage First" or G1. It was available in JDK 7 and made the default in JDK 8. G1 is designed to perform better on very large heaps and would most likely reduce the garbage collection problems.

The G1 algorithm might however still need additional tuning [WH15; KY15], so it was assessed that the expected cost of trying out G1 did not justify the expected gains.

The test were therefore stopped after the data set of 8,192,000 genes.

9.3.3 Reducing the Number of Objects

As opposed to tuning how the garbage collector works, we can also reduce the number of objects it has to keep track of. Doing this exploits the fact that garbage collection time depends on the number of updates not the size of those objects. Having fewer larger objects instead of many small will therefore improve performance.

This is especially powerful when dealing with primitive types such as integers and doubles. Instead of having a list with a lot of integer-double-pairs we can store two lists. One list of all the integers and one of all the doubles. We will then end up with only two objects (the lists) instead of one object per element (the pairs).

That approach is regrettably not possible when using a framework like Spark. We need to use an API which accepts and returns small objects.

A similar approach can be used when dealing with complex objects. They can be stored in serialized form in a byte array, instead of storing them as normal Java objects. The garbage collection will then only have to deal with a single object, just as before. A drawback of this approach is that it will require additional overhead to convert the objects between serialized form and plain old Java objects. This approach is thus best for objects that are kept around for a long time and accessed infrequently.

A good candidate for this optimization is cached RDD's. Apache Spark allows the user to decide to cached RDD's in serialized form. This was already done before these garbage collection problems showed, so there was no more performance gain to be achieved.

A third and more advanced approach is to partly take over memory management from the JVM. This can be done using `sun.misc.Unsafe` [Koz13]. Manual memory management is cumbersome and error-prone, especially in languages like Java that in principle do not support it. It should therefore be avoided as much as possible, and left to libraries and frameworks.

The good news is that users of Apache Spark might soon get the benefits without having to deal with memory management. The developers behind Apache Spark are working on efficient ways of exploiting manual memory management [XR15]. The improvements are planned to be included in version 1.5 of Apache Spark.

A last and more indirect way of reducing objects is to use more and smaller JVM's. For example, instead of running one Spark worker instance on each of a number of 32-core servers, one can start four Spark worker JVM's per server using eight cores each. Alternatively one can start four times as many servers with eight cores each, but that will naturally increase network I/O.

This is a very simple way to reduce the number of objects per JVM, which will lead to shorter garbage collection pauses. Hopefully that reduction will be more significant than the extra overhead of more JVM's. All garbage collectors across JVM's will still have to manage the same number of objects, but hopefully there will be fewer problems with full young generations and incomplete concurrent marks.

9.4 Prospects of PECA

There is a lot of opportunities for improving the performance of PECA as shown above. Some of these are improvements in the Apache Spark framework, but that is not a major problem. The issue with garbage collection is expected to be solved later this year, and the problem with GraphX can be worked around by implementing the clustering algorithm using normal RDD's.

The suggested implementation improvements will primarily improve time spent on shuffling data. This is expected to have a significant impact on performance, since most calculations are very simple so the overhead of handling and moving the data is expected to be proportionately large.

One exception of this is the alignment step. This step is also the most resource consuming part of PECA. If we ignore the time spent on the alignment, PECA is running more than four times faster than CD-Hit using only four machines. The tests showed that the time per cluster round is not increased a lot when selecting centers based on proxies.

An efficient solution to Problem 9.1 enable us to avoid the alignment step. This improvement alone are likely to reduce the runtime to a level where it will be similar to that of CD-Hit, even on the same hardware. Improving the implementation as suggested will increase the performance even further, and PECA therefore has the prospect of clearly outperforming CD-Hit.

9.5 Evaluation of Apache Spark

One of the goals with this thesis was to show that it is possible to move bioinformatic computations to the cloud and that it is relatively simple to implement new, massively distributed algorithms using modern tools and frameworks. The thesis will therefore not be complete without an evaluation of the opportunities, barriers, and challenges when doing this.

"Ease of use" is one of the four main selling points of Apache Spark [Fou15d]. That is definitely met when it comes to writing programs. The code required to manipulate

distributed data is concise, short, and easy to read, as we saw in Chapters 3, 5 and 7. This lets the programmer focus on the logic of the algorithm, and enables more advanced algorithms to be implemented with less effort.

There are however still significant problems. Users of Apache Spark needs to manually specify the number of partitions for the RDD's. They have to decide which number of partitions would yield the best trade-off between overhead and parallelizability. If the partitions are too small there would be too much overhead; if they are too large a few of them might become stranglers, and performance will decline.

One of the advantages of Apache Spark is that the backend automatically rearranges operations to improve performance. Unfortunately the monitoring framework report statistics based on this rearranged execution graph. It therefore become a bit of a detective work to figure out which lines of one's code that belong to each task listed in the monitoring interface. Version 1.4.0 included some updates improving this part of the monitoring interface [Fou15g]. Future versions will likely improve this even more.

We have seen how Apache Spark needed tuning of the garbage collector to handle PECA, even though the input data was less than 25 GB. Tuning a garbage collector is an advanced topic that would be a prohibitive factor for many programmers, who may have a background in bioinformatics and not advanced software engineering. It will be interesting to see if this problem gets solved in version 1.5.

The test showed that the GraphX library has significant parallelizability problems. The library is officially still in alpha, so hopefully these problems will be fixed before it gets general availability status.

Apache Spark has limited support for multiple users and dynamic scaling of resource allocations for jobs [Fou15b]. Similarly there is no functionality for autoscaling the number of provisioned servers in a cloud environment. Apache Spark currently has tooling for running it on Amazon Web Services, but that require the user to set a specific number of servers. In addition the startup time is very slow. Starting a 32 machine cluster takes almost one hour. That is too much time to be a viable option for one time computations.

Automatically provisioning the servers needed depending on demand is one of the main advantages of cloud computing. A few commercial services promises to solve the provisioning problem. Amazon Web Services has a product, called Elastic MapReduce, that promises to take the pain of provisioning away [Ser15a]. It makes it easier to start up or resize a cluster, but it neither perform autoscaling nor autotuning [Ser15b; Ser15c]. And other online service, called Qubole, provides autoscaling managed clusters [Qub15b], but does provides any autotuning or garbage collection tuning [Qub15a].

In summary Apache Spark is a very promising programming framework. The version used for this project (1.3.0) still has a few, but significant problems that could be prohibitive for widespread use. It is not plug and play framework for massively distributed computations, yet. Releases planned for this autumn addresses some of the issues experienced during the work on this thesis. The most important of these being in the work to prevent problems with the garbage collector.

9.5.1 A Quick Revisit to Google Cloud Dataflow

Many of the problems addressed in this section had already been solved by Google Cloud Dataflow, which became generally available on August 12, 2015 [Sch15]. Their main goal is exactly to provide a managed service that completely takes care of autoscaling, provisioning, autotuning, etc. At least that is what is promised.

Those advantages might more than make up for the significantly more cluttered API and the fact that you are bound to use and pay for Google's servers. These problems can in theory be mitigated but not in a practical feasible way. There are projects working on wrapping Cloud Dataflow's API and exposing a nicer API similar to that used by Apache Spark [Ju15; Hou15], but none of these seems to be mature or in active development. One can naturally avoid being tied to Google servers by using the Cloud Dataflow open source backend that is implemented on top of Apache Spark. This will however give you the worst of both worlds: the cluttered API from Cloud Dataflow and the tuning problems from Apache Spark.

It would be interesting with a thorough comparison between Google Cloud Dataflow and Apache Spark in terms of performance, ease of use, flexibility, etc., that could give the scientific community sufficient knowledge to choose between the two.

9.6 Summary

PECA outperformed UClust by a wide margin and was barely twice as fast as CD-Hit, but using four machines instead of one. Overall that is pretty good performance, since CD-Hit is a mature and optimized tool and is expected to have a distribution bottleneck by requiring one shared file server.

Several relatively simple implementation improvements can improve the performance even further, but the highest performance gain would be achieved by efficiently solving the *proxy-based center selection* problem which has been formally defined. Solving that problem would significantly reduce alignment time, which is the majority of the runtime.

The scalability of PECA was hampered by some problems in Apache Spark, primarily problems with garbage collection and poor parallelizability of the GraphX library. Version 1.5 of Apache Spark contains an update and that will hopefully solve the garbage collection problems without the need of tuning the garbage collector. The problems with GraphX has been reported and will hopefully also be resolved in future versions.

Further work on PECA might very well make it outperform CD-Hit on an equal amount of servers, when the problems in the Spark framework are resolved, and especially if a good solution is found to proxy-based center selection.

In conclusion, this thesis has shown that it is feasible to develop and implement new algorithms that are designed for the cloud to replace the existing bioinformatic applications. There are still algorithmic challenges that needs to be overcome, and

practical issues that needs to be fixed in Apache Spark – but possibly Google Cloud Dataflow has already overcome those practical issues. We are not completely there yet where cloud solutions clearly outperforms the old-fashioned tools, but the work in this thesis has taking us much closer.

Cloud computing is a lot more accessible now that it used to. New computation frameworks like Apache Spark and Google Cloud Dataflow makes it easier and simpler to write advanced distributed algorithms. There are still a few quirks; Apache Spark has some tuning problems, and the API for Google Cloud Dataflow still requires a lot of boilerplate code. Cloud computing is therefore on the brink between the early adopter stage and being available for widespread audience.

CHAPTER 10

Conclusions

Proxy-based Efficient Clustering Algorithm (PECA) has been presented. It is a distributable algorithm that efficiently clusters genes. Using four servers it runs 40 times faster than UClust and 1.68 times faster than CD-Hit on a single server. The runtime can be further reduced by increasing the number of servers.

This performance was achieved in part by generalizing the DISCO algorithm for correlation between binary vectors. It was shown that with minor changes the DISCO algorithm will work with real valued vectors and any correlation measure that is a sum of real values. The generalized DISCO algorithm maintains the property that the error bound can be reduced exponentially with a linear increase runtime, assuming the correlation measure is a sum of positive values.

Another important step to the efficiency of PECA was new research conducted into the link between k-mer profiles and gene identity. It was found that Pearson correlation and cosine similarity between two k-mer profiles works well as a proxy for gene identity. Pearson correlation and cosine similarity was found to be better to identify pairs of genes with an identity above 95%, than the similarity measure used in UClust [Edg10].

A series of implementation improvements has been identified. They are expected to provide a significant increase in performance.

Even larger performance improvements can be achieved by algorithmic changes. The subproblem of *proxy-based center selection* has been formally described, and finding an efficient algorithm to solve that will improve performance much more than any implementation improvements are able to.

Overall PECA is found to be a promising way of clustering genes. The implementation for this thesis is mostly a prototype, but it is still able to outperform existing tools using more hardware. There are still some challenges to overcome, but once those have been solved PECA is expected to clearly outperform existing tools, even on the same hardware. PECA is a good example of the opportunities the scientific community have in the cloud.

Existing cluster quality measures was found to be unsuitable for gene clustering. They provides a poor indication of the quality and are infeasible to calculate on a large number of clusters. A new scalable measure for cluster quality, Information Inspired Quality or IIQ, was suggested and it showed that PECA produces clusters of quality identical to existing tools.

Apache Spark was found to be a succinct, flexible, and powerful cloud framework for writing massively parallel algorithms. Its program model is very similar to that

of Google Dataflow, but its API is much cleaner and easier to both read and write.

Apache Spark does have some production problems. It has very little support for autoscaling jobs, but that might be mitigated by using commercial third-party services. There is also no support for autotuning. The user has to manually define the number of partitions, but even worse is the need for complex tuning of the garbage collector. Lastly the monitoring interface has a weak resemblance of the written code. The problems with the monitoring interface and garbage collection are however expected to be fixed later this year. All in all Apache Spark is on the edge between being a framework primarily for early adopters and being a framework that makes cloud computing accessible to a wider user base.

APPENDIX A

Spark Documentation

This appendix contains excerpts from the official documentation for Apache Spark [Fou15f; Fou15a]. It is meant to give the reader all the knowledge needed about Apache Spark to understand this thesis.

At a high level, every Spark application consists of a driver program that runs the user's main function and executes various parallel operations on a cluster. The main abstraction Spark provides is a resilient distributed dataset (RDD), which is a collection of elements partitioned across the nodes of the cluster that can be operated on in parallel. RDDs are created by starting with a file in the Hadoop file system (or any other Hadoop-supported file system), or an existing Scala collection in the driver program, and transforming it. Users may also ask Spark to persist an RDD in memory, allowing it to be reused efficiently across parallel operations. Finally, RDDs automatically recover from node failures.

A.1 RDDs

RDDs support two types of operations: transformations, which create a new dataset from an existing one, and actions, which return a value to the driver program after running a computation on the dataset. For example, map is a transformation that passes each dataset element through a function and returns a new RDD representing the results. On the other hand, reduce is an action that aggregates all the elements of the RDD using some function and returns the final result to the driver program (although there is also a parallel reduceByKey that returns a distributed dataset).

All transformations in Spark are lazy, in that they do not compute their results right away. Instead, they just remember the transformations applied to some base dataset (e.g. a file). The transformations are only computed when an action requires a result to be returned to the driver program. This design enables Spark to run more efficiently – for example, we can realize that a dataset created through map will be used in a reduce and return only the result of the reduce to the driver, rather than the larger mapped dataset.

By default, each transformed RDD may be recomputed each time you run an action on it. However, you may also persist an RDD in memory using the persist (or cache) method, in which case Spark will keep the elements around on the cluster for much faster access the next time you query it. There is also support for persisting RDDs on disk, or replicated across multiple nodes.

A.2 Basics

To illustrate RDD basics, consider the simple program below:

```
val lines = sc.textFile("data.txt")
val lineLengths = lines.map(s => s.length)
val totalLength = lineLengths.reduce((a, b) => a + b)
```

The first line defines a base RDD from an external file. This dataset is not loaded in memory or otherwise acted on: lines is merely a pointer to the file. The second line defines lineLengths as the result of a map transformation. Again, lineLengths is not immediately computed, due to laziness. Finally, we run reduce, which is an action. At this point Spark breaks the computation into tasks to run on separate machines, and each machine runs both its part of the map and a local reduction, returning only its answer to the driver program.

If we also wanted to use lineLengths again later, we could add:

```
lineLengths.persist()
```

before the reduce, which would cause lineLengths to be saved in memory after the first time it is computed.

A.3 Working with Key-Value Pairs

While most Spark operations work on RDDs containing any type of objects, a few special operations are only available on RDDs of key-value pairs. The most common ones are distributed "shuffle" operations, such as grouping or aggregating the elements by a key.

In Scala, these operations are automatically available on RDDs containing Tuple2 objects (the built-in tuples in the language, created by simply writing (a, b)). The key-value pair operations are available in the `PairRDDFunctions` class, which automatically wraps around an RDD of tuples.

For example, the following code uses the reduceByKey operation on key-value pairs to count how many times each line of text occurs in a file:

```
val lines = sc.textFile("data.txt")
val pairs = lines.map(s => (s, 1))
val counts = pairs.reduceByKey((a, b) => a + b)
```

We could also use `counts.sortByKey()`, for example, to sort the pairs alphabetically, and finally `counts.collect()` to bring them back to the driver program as an array of objects.

A.4 Transformations

The following description lists some of the common transformations supported by Spark.

`map(func)` Return a new distributed dataset formed by passing each element of the source through a function func.

`filter(func)` Return a new dataset formed by selecting those elements of the source on which funcreturns true.

`flatMap(func)` Similar to map, but each input item can be mapped to 0 or more output items (so funcshould return a Seq rather than a single item).

`sample(withReplacement, fraction, seed)` Sample a fraction fraction of the data, with or without replacement, using a given random number generator seed.

`distinct([numTasks]))` Return a new dataset that contains the distinct elements of the source dataset.

`groupByKey([numTasks])` When called on a dataset of `(K, V)` pairs, returns a dataset of `(K, Iterable<V>)` pairs.
Note: If you are grouping in order to perform an aggregation (such as a sum or average) over each key, using reduceByKey or aggregateByKey will yield much better performance.
Note: By default, the level of parallelism in the output depends on the number of partitions of the parent RDD. You can pass an optional numTasks argument to set a different number of tasks.

`reduceByKey(func, [numTasks])` When called on a dataset of `(K, V)` pairs, returns a dataset of `(K, V)` pairs where the values for each key are aggregated using the given reduce function func, which must be of type `(V,V) => V`. Like in groupByKey, the number of reduce tasks is configurable through an optional second argument.

`join(otherDataset, [numTasks])` When called on datasets of type `(K, V)` and `(K, W)`, returns a dataset of `(K, (V, W))` pairs with all pairs of elements for each key. Outer joins are supported through `leftOuterJoin`, `rightOuterJoin`, and `fullOuterJoin`.

`repartition(numPartitions)` Reshuffle the data in the RDD randomly to create either more or fewer partitions and balance it across them. This always shuffles all data over the network.

A.5 Actions

The following description lists some of the common actions supported by Spark.

`reduce(func)` Aggregate the elements of the dataset using a function `func` (which takes two arguments and returns one). The function should be commutative and associative so that it can be computed correctly in parallel.

`collect()` Return all the elements of the dataset as an array at the driver program. This is usually useful after a filter or other operation that returns a sufficiently small subset of the data.

`count()` Return the number of elements in the dataset.

`isEmpty()` Returns `true` if there are no elements in the RDD.

`take(n)` Return an array with the first `n` elements of the dataset.

`takeSample(withReplacement,num, [seed])` Return an array with a random sample of num elements of the dataset, with or without replacement, optionally pre-specifying a random number generator seed.

`saveAsTextFile(path)` Write the elements of the dataset as a text file (or set of text files) in a given directory in the local filesystem, HDFS or any other Hadoop-supported file system. Spark will call toString on each element to convert it to a line of text in the file.

`saveAsSequenceFile(path)` Write the elements of the dataset as a Hadoop SequenceFile in a given path in the local filesystem, HDFS or any other Hadoop-supported file system. This is available on RDDs of key-value pairs that implement Hadoop's Writable interface. In Scala, it is also available on types that are implicitly convertible to Writable (Spark includes conversions for basic types like `Int`, `Double`, `String`, etc).

A.6 Graph Analysis with GraphX

GraphX is a new component in Spark for graphs and graph-parallel computation. At a high level, GraphX extends the Spark RDD by introducing a new Graph abstraction: a directed multigraph with properties attached to each vertex and edge. To support graph computation, GraphX exposes a set of fundamental operators (e.g., subgraph, joinVertices, and aggregateMessages) as well as an optimized variant of the Pregel API.

In addition, GraphX includes a growing collection of graph algorithms and builders to simplify graph analytics tasks.

The property graph is a directed multigraph with user defined objects attached to each vertex and edge. A directed multigraph is a directed graph with potentially multiple parallel edges sharing the same source and destination vertex. The ability to support parallel edges simplifies modeling scenarios where there can be multiple relationships (e.g., co-worker and friend) between the same vertices. Each vertex is keyed by a unique 64-bit long identifier (`VertexID`). GraphX does not impose any ordering constraints on the vertex identifiers. Similarly, edges have corresponding source and destination vertex identifiers.

The property graph is parameterized over the vertex (`VD`) and edge (`ED`) types. These are the types of the objects associated with each vertex and edge respectively.

GraphX optimizes the representation of vertex and edge types when they are primitive data types (e.g., int, double, etc…) reducing the in memory footprint by storing them in specialized arrays. In some cases it may be desirable to have vertices with different property types in the same graph. This can be accomplished through inheritance. For example to model users and products as a bipartite graph we might do the following:

```
class VertexProperty()
case class UserProperty(val name: String) extends VertexProperty
case class ProductProperty(val name: String, val price: Double) extends
     VertexProperty
// The graph might then have the type:
var graph: Graph[VertexProperty, String] = null
```

Like RDDs, property graphs are immutable, distributed, and fault-tolerant. Changes to the values or structure of the graph are accomplished by producing a new graph with the desired changes. Note that substantial parts of the original graph (i.e., unaffected structure, attributes, and indicies) are reused in the new graph reducing the cost of this inherently functional data structure. The graph is partitioned across the executors using a range of vertex partitioning heuristics. As with RDDs, each partition of the graph can be recreated on a different machine in the event of a failure.

Logically the property graph corresponds to a pair of typed collections (RDDs) encoding the properties for each vertex and edge. As a consequence, the graph class contains members to access the vertices and edges of the graph:

```
class Graph[VD, ED] {
  val vertices: VertexRDD[VD]
  val edges: EdgeRDD[ED]
}
```

The classes `VertexRDD[VD]` and `EdgeRDD[ED]` extend and are optimized versions of `RDD[(VertexID, VD)]` and `RDD[Edge[ED]]` respectively. Both `VertexRDD[VD]` and `EdgeRDD[ED]` provide additional functionality built around graph computation and leverage internal optimizations. We discuss the `VertexRDD` and `EdgeRDD` API in greater detail in the section on vertex and edge RDDs but for now they can be thought of as simply RDDs of the form: `RDD[(VertexID, VD)]` and `RDD[Edge[ED]]`.

A.6.1 Operations on Graphs

Just as RDDs have basic operations like map, filter, and reduceByKey, property graphs also have a collection of basic operators that take user defined functions and produce new graphs with transformed properties and structure. The core operators that have optimized implementations are defined in `Graph`.

A key step in many graph analytics tasks is aggregating information about the neighborhood of each vertex. For example, we might want to know the number of followers each user has or the average age of the the followers of each user. Many iterative graph algorithms (e.g., PageRank, Shortest Path, and connected components) repeatedly aggregate properties of neighboring vertices (e.g., current PageRank Value, shortest path to the source, and smallest reachable vertex id).

The core aggregation operation in GraphX is `aggregateMessages`. This operator applies a user defined `sendMsg` function to each edge triplet in the graph and then uses the mergeMsg function to aggregate those messages at their destination vertex.

class Graph[VD, ED] def aggregateMessages[Msg: ClassTag](sendMsg: EdgeContext[VD, ED, Msg] => Unit, mergeMsg: (Msg, Msg) => Msg, tripletFields: TripletFields = TripletFields.All) : VertexRDD[Msg]

The user defined `sendMsg` function takes an `EdgeContext`, which exposes the source and destination attributes along with the edge attribute and functions (`sendToSrc`, and `sendToDst`) to send messages to the source and destination attributes. Think of `sendMsg` as the map function in map-reduce. The user defined `mergeMsg` function takes two messages destined to the same vertex and yields a single message. Think of `mergeMsg` as the reduce function in map-reduce. The `aggregateMessages` operator returns a `VertexRDD[Msg]` containing the aggregate message (of type `Msg`) destined to each vertex. Vertices that did not receive a message are not included in the returned `VertexRDD`.

In addition, `aggregateMessages` takes an optional `tripletsFields` which indicates what data is accessed in the `EdgeContext` (i.e., the source vertex attribute but not the destination vertex attribute). The possible options for the `tripletsFields` are defined in `TripletFields` and the default value is `TripletFields`.All which indicates that the user defined `sendMsg` function may access any of the fields in the `EdgeContext`. The `tripletFields` argument can be used to notify GraphX that only part of the `EdgeContext` will be needed allowing GraphX to select an optimized join strategy. For example if we are computing the average age of the followers of each user we would only require the source field and so we would use `TripletFields.All` to indicate that we only require the source field

In the following example we use the `aggregateMessages` operator to compute the average age of the more senior followers of each user.

```
// Import random graph generation library
import org.apache.spark.graphx.util.GraphGenerators
// Create a graph with "age" as the vertex property.  Here we use a random graph for
      simplicity.
val graph: Graph[Double, Int] =
  GraphGenerators.logNormalGraph(sc, numVertices = 100).mapVertices( (id, _) => id.
       toDouble )
// Compute the number of older followers and their total age
val olderFollowers: VertexRDD[(Int, Double)] = graph.aggregateMessages[(Int, Double)
     ](
  triplet => { // Map Function
    if (triplet.srcAttr > triplet.dstAttr) {
      // Send message to destination vertex containing counter and age
      triplet.sendToDst(1, triplet.srcAttr)
    }
  },
  // Add counter and age
  (a, b) => (a._1 + b._1, a._2 + b._2) // Reduce Function
)
// Divide total age by number of older followers to get average age of older
     followers
val avgAgeOfOlderFollowers: VertexRDD[Double] =
  olderFollowers.mapValues( (id, value) => value match { case (count, totalAge) =>
       totalAge / count } )
// Display the results
avgAgeOfOlderFollowers.collect.foreach(println(_))
```

The aggregateMessages operation performs optimally when the messages (and the sums of messages) are constant sized (e.g., floats and addition instead of lists and concatenation).

APPENDIX B

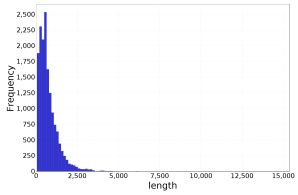
Plots for Analysis of Proxies for Gene Identity

This appendix contains the full collection of plots created to evaluate the three correlation measures analyzed in Chapter 5.

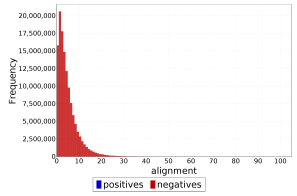
The plots were done on a subsample of 16,000 genes from the Human Microbiome Project (HMP) dataset [Met+12].

B.1 Alignment and Length

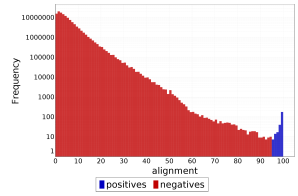
These plots show a simple overview of the distribution of gene lengths in the sample, and the distribution of gene identity between every pair in the subsample.



Gene length



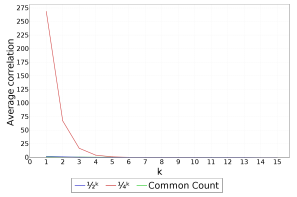
Identity distribution



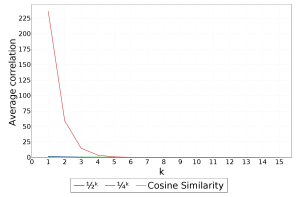
Identity distribution (log)

B.2 Average Correlation

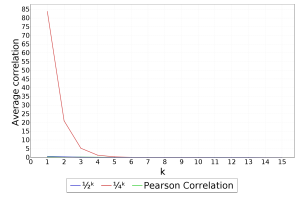
These plots show, for all three correlation measures, the decline of the average correlation between a pair of two genes as k increases.



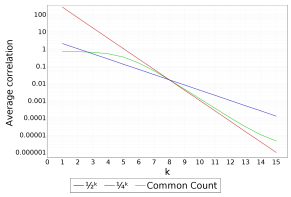
Common count



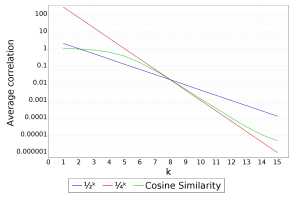
Cosine



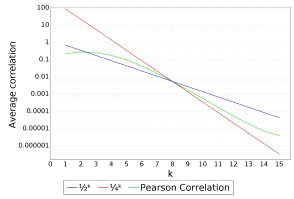
Pearson



Common count (log)



Cosine (log)



Pearson (log)

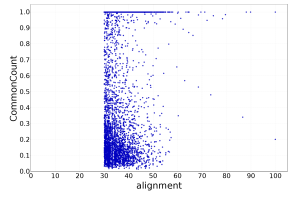
B.3 Scatterplots

This section shall scatter plots for all three correlation measures. Each point is a pair of genes, placed with their identity on the x-axis and that correlation value on the y-axis.

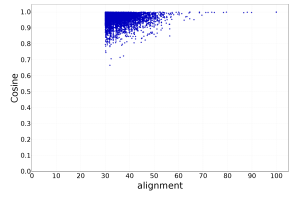
There are two sets of plots. The first is based on subsample of 2000 genes, and contains all pairs with an identity above 30%. The second is based on the subsample of 16,000 genes, and contains all pairs with an identity above 60%.

Each line of plots show a different value of k.

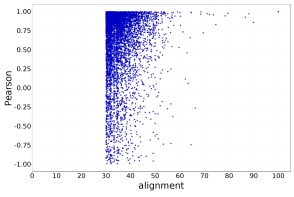
B.3.1 2,000 Genes



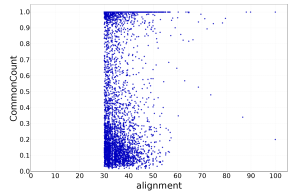
Common count, $k = 1$



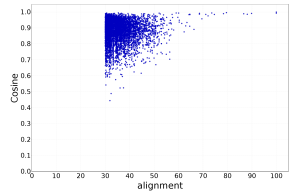
Cosine, $k = 1$



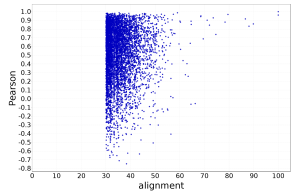
Pearson, $k = 1$



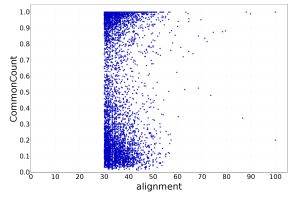
Common count, $k = 2$



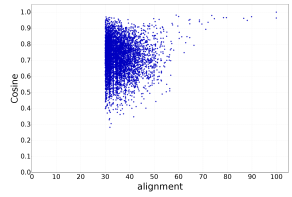
Cosine, $k = 2$



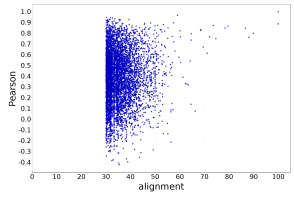
Pearson, $k = 2$



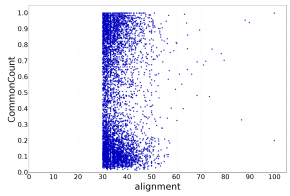
Common count, $k = 3$



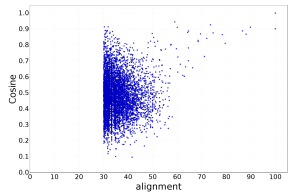
Cosine, $k = 3$



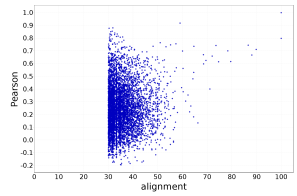
Pearson, $k = 3$



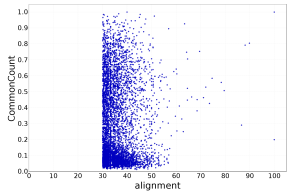
Common count, $k = 4$



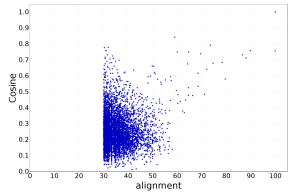
Cosine, $k = 4$



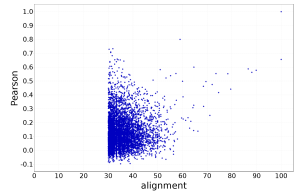
Pearson, $k = 4$



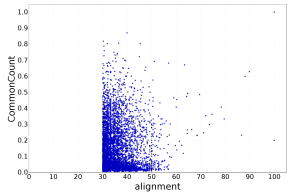
Common count, $k = 5$



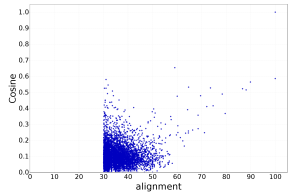
Cosine, $k = 5$



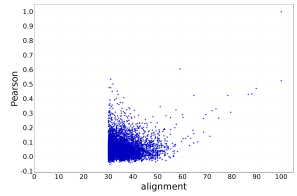
Pearson, $k = 5$



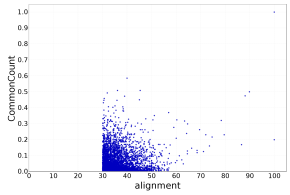
Common count, $k = 6$



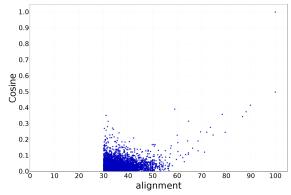
Cosine, $k = 6$



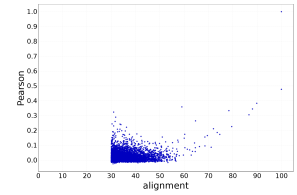
Pearson, $k = 6$



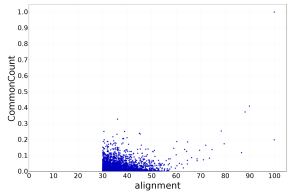
Common count, $k = 7$



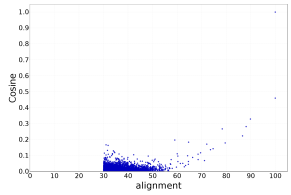
Cosine, $k = 7$



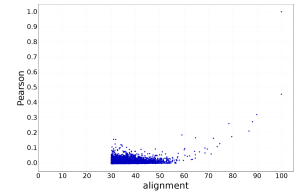
Pearson, $k = 7$



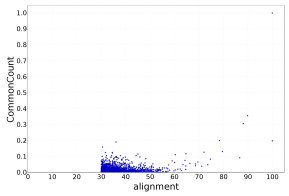
Common count, $k = 8$



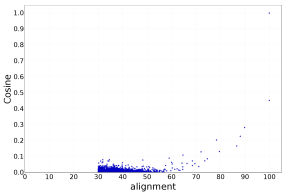
Cosine, $k = 8$



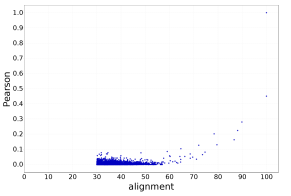
Pearson, $k = 8$



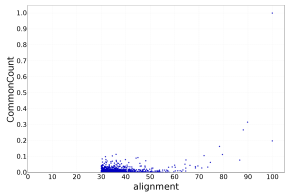
Common count, $k = 9$



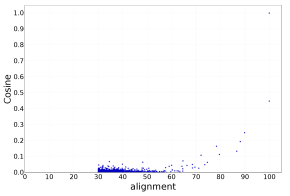
Cosine, $k = 9$



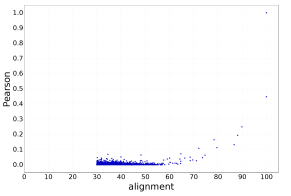
Pearson, $k = 9$



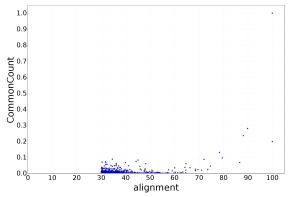
Common count, $k = 10$



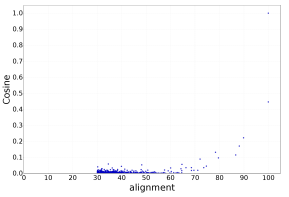
Cosine, $k = 10$



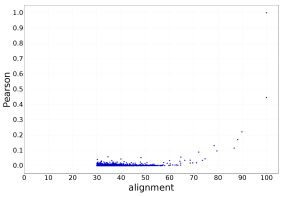
Pearson, $k = 10$



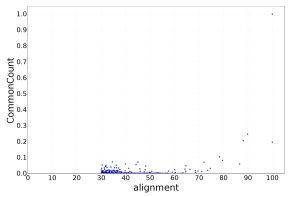
Common count, $k = 11$



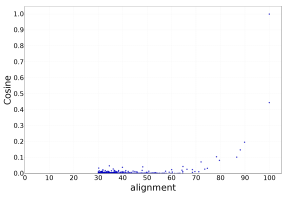
Cosine, $k = 11$



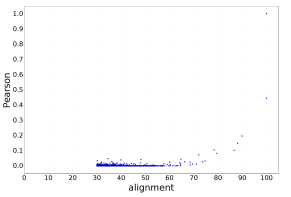
Pearson, $k = 11$



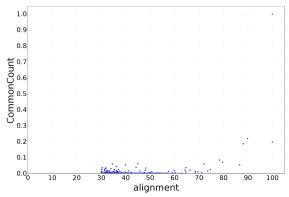
Common count, $k = 12$



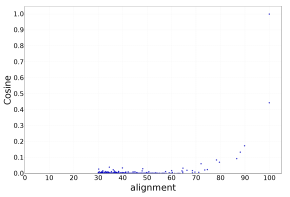
Cosine, $k = 12$



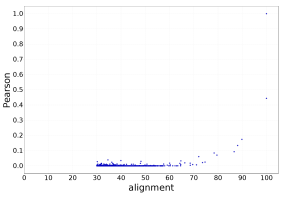
Pearson, $k = 12$



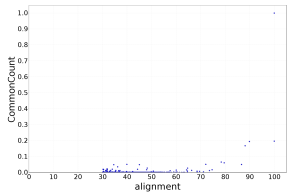
Common count, $k = 13$



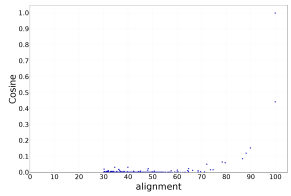
Cosine, $k = 13$



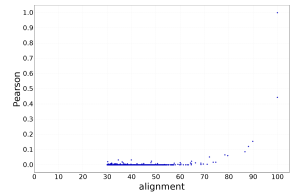
Pearson, $k = 13$



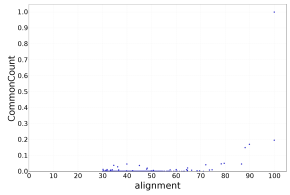
Common count, $k = 14$



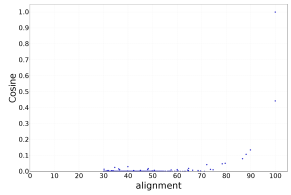
Cosine, $k = 14$



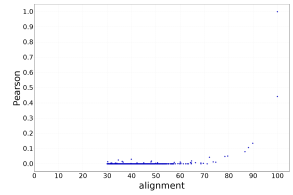
Pearson, $k = 14$



Common count, $k = 15$

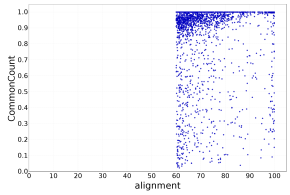


Cosine, $k = 15$

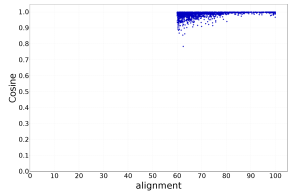


Pearson, $k = 15$

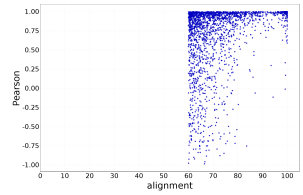
B.3.2 16,000 Genes



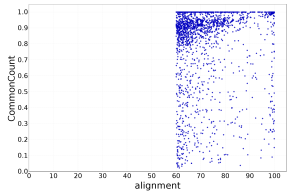
Common count, $k = 1$



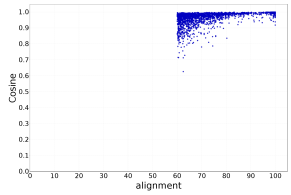
Cosine, $k = 1$



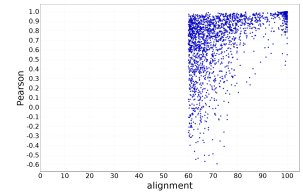
Pearson, $k = 1$



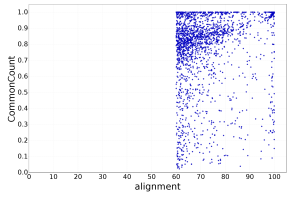
Common count, $k = 2$



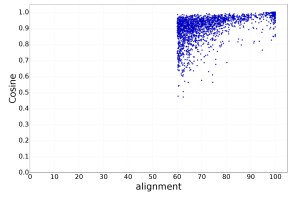
Cosine, $k = 2$



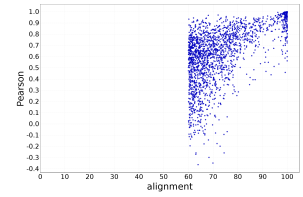
Pearson, $k = 2$



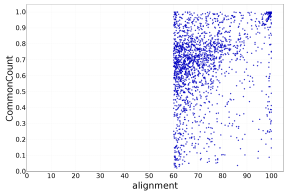
Common count, $k = 3$



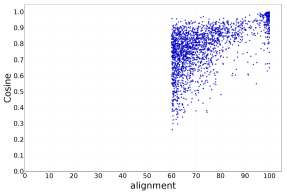
Cosine, $k = 3$



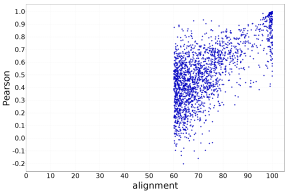
Pearson, $k = 3$



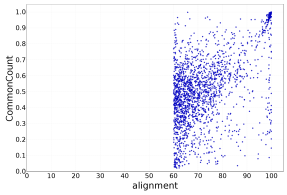
Common count, $k = 4$



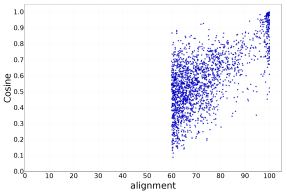
Cosine, $k = 4$



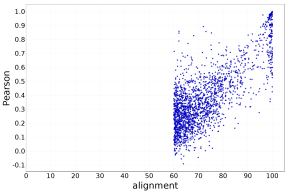
Pearson, $k = 4$



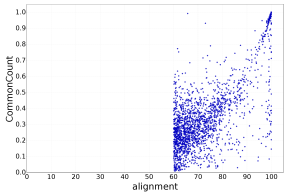
Common count, $k = 5$



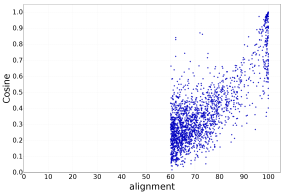
Cosine, $k = 5$



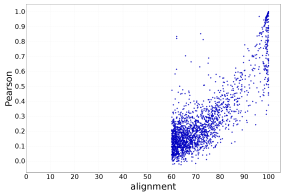
Pearson, $k = 5$



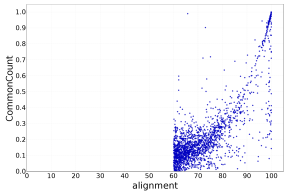
Common count, $k = 6$



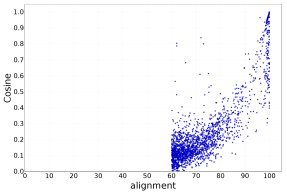
Cosine, $k = 6$



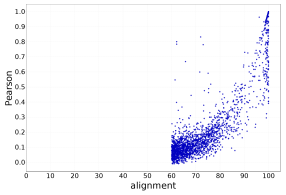
Pearson, $k = 6$



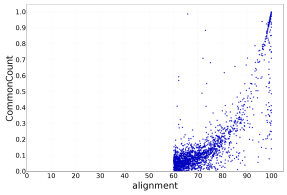
Common count, $k = 7$



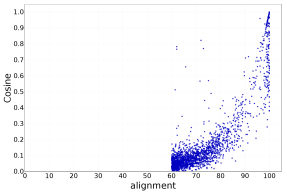
Cosine, $k = 7$



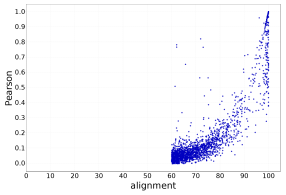
Pearson, $k = 7$



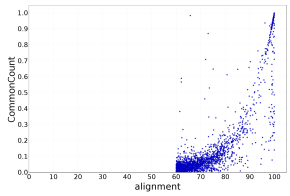
Common count, $k = 8$



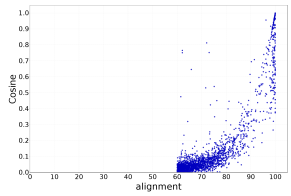
Cosine, $k = 8$



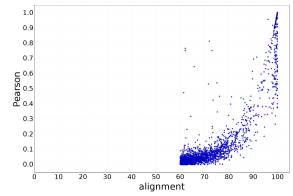
Pearson, $k = 8$



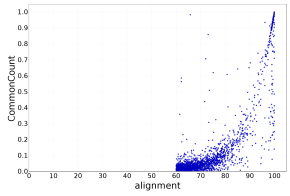
Common count, $k = 9$



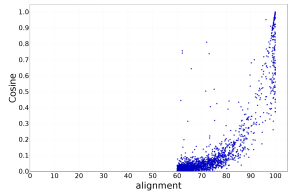
Cosine, $k = 9$



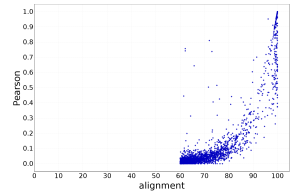
Pearson, $k = 9$



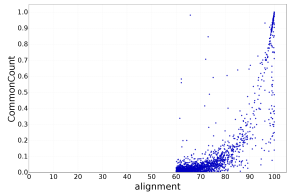
Common count, $k = 10$



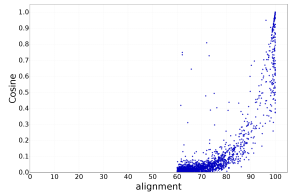
Cosine, $k = 10$



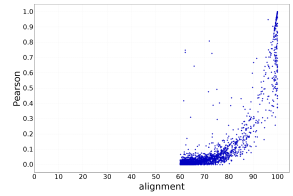
Pearson, $k = 10$



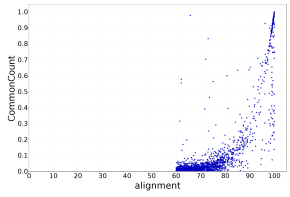
Common count, $k = 11$



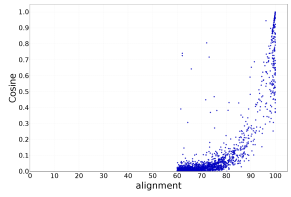
Cosine, $k = 11$



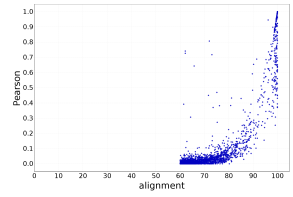
Pearson, $k = 11$



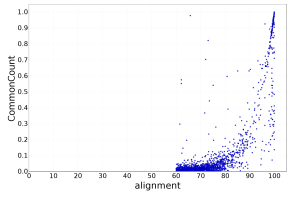
Common count, $k = 12$



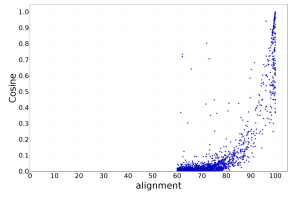
Cosine, $k = 12$



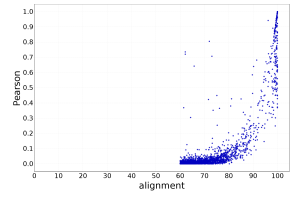
Pearson, $k = 12$



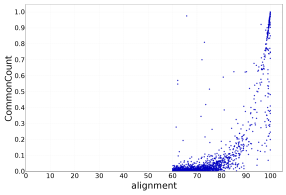
Common count, $k = 13$



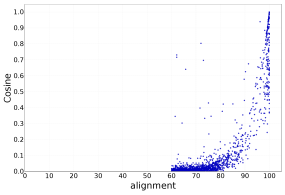
Cosine, $k = 13$



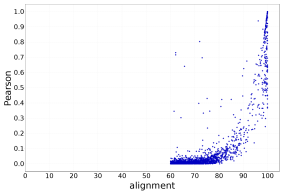
Pearson, $k = 13$



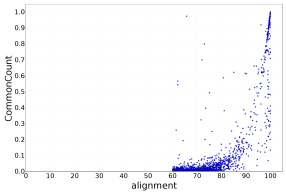
Common count, $k = 14$



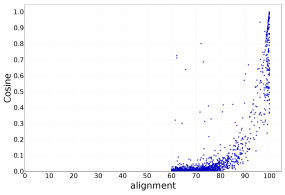
Cosine, $k = 14$



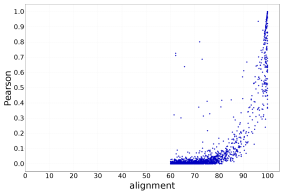
Pearson, $k = 14$



Common count, $k = 15$



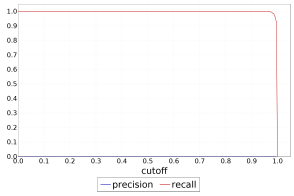
Cosine, $k = 15$



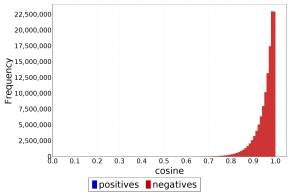
Pearson, $k = 15$

B.4 Precision and Recall

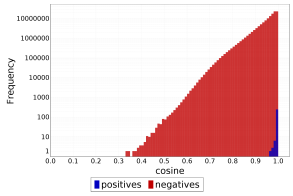
This list of plots shows how the precision and recall develops as k increases in the first column. The second and third column shows the distribution of cosine similarity between all pairs of k-mer profiles as k increases.



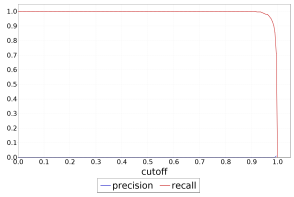
Precision and recall, $k = 1$



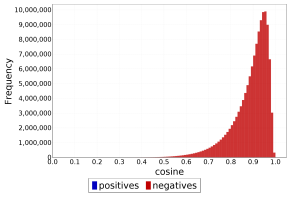
Cosine similarity, $k = 1$



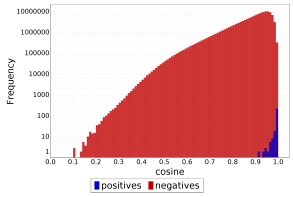
Cosine similarity (log), $k = 1$



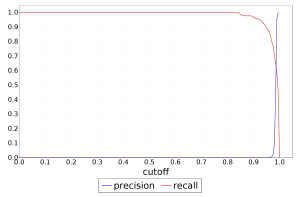
Precision and recall, $k = 2$



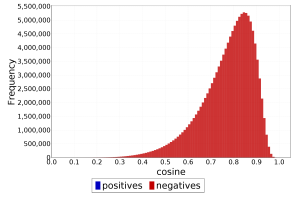
Cosine similarity, $k = 2$



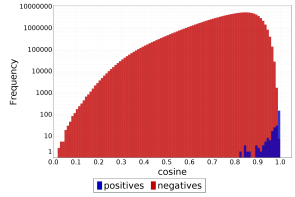
Cosine similarity (log), $k = 2$



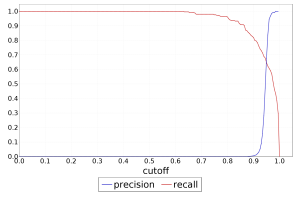
Precision and recall, $k = 3$



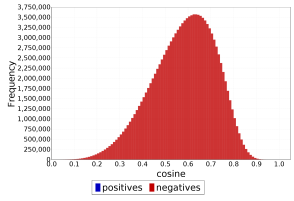
Cosine similarity, $k = 3$



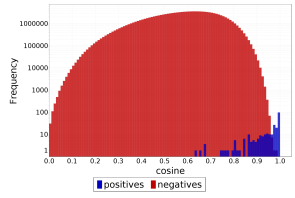
Cosine similarity (log), $k = 3$



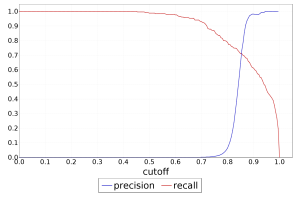
Precision and recall, $k = 4$



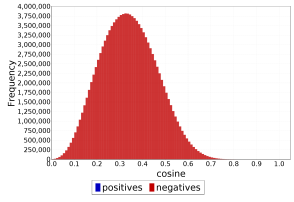
Cosine similarity, $k = 4$



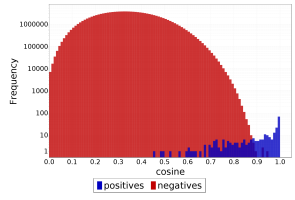
Cosine similarity (log), $k = 4$



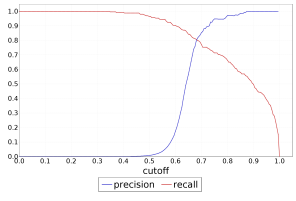
Precision and recall, $k = 5$



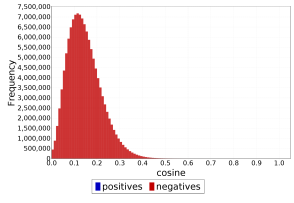
Cosine similarity, $k = 5$



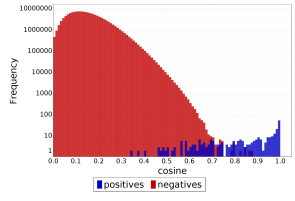
Cosine similarity (log), $k = 5$



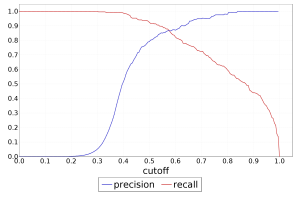
Precision and recall, $k = 6$



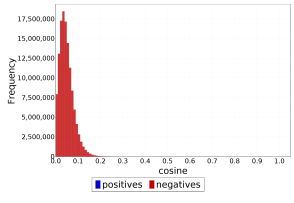
Cosine similarity, $k = 6$



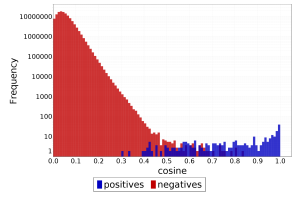
Cosine similarity (log), $k = 6$



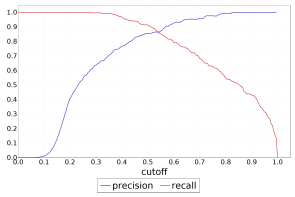
Precision and recall, $k = 7$



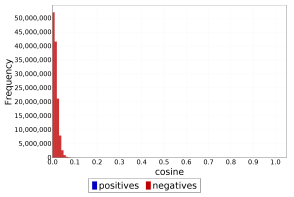
Cosine similarity, $k = 7$



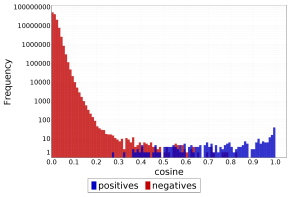
Cosine similarity (log), $k = 7$



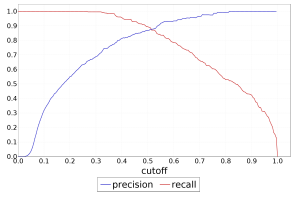
Precision and recall, $k = 8$



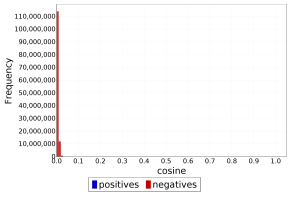
Cosine similarity, $k = 8$



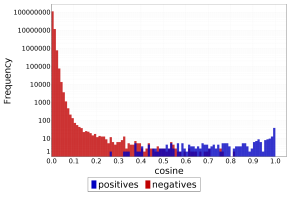
Cosine similarity (log), $k = 8$



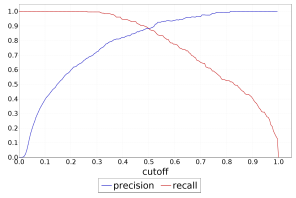
Precision and recall, $k = 9$



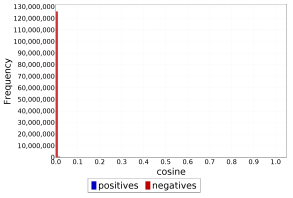
Cosine similarity, $k = 9$



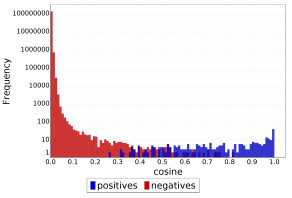
Cosine similarity (log), $k = 9$



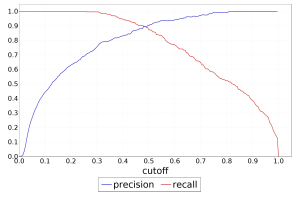
Precision and recall, $k = 10$



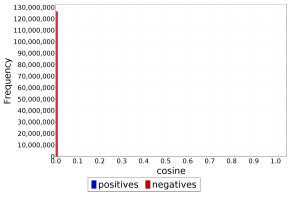
Cosine similarity, $k = 10$



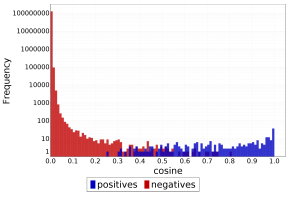
Cosine similarity (log), $k = 10$



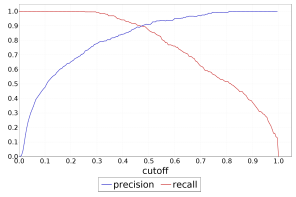
Precision and recall, $k = 11$



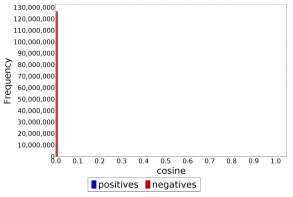
Cosine similarity, $k = 11$



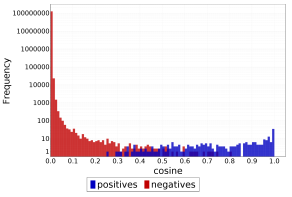
Cosine similarity (log), $k = 11$



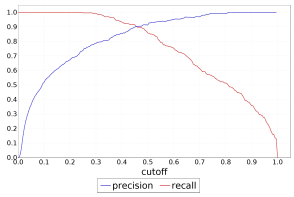
Precision and recall, $k = 12$



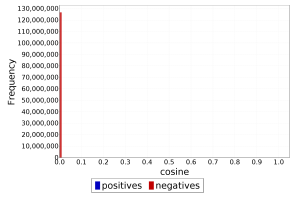
Cosine similarity, $k = 12$



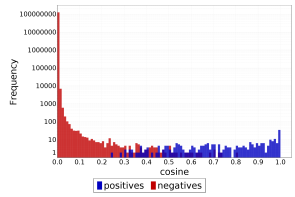
Cosine similarity (log), $k = 12$



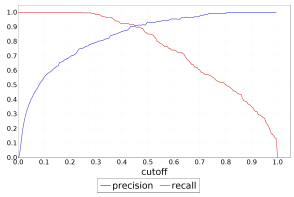
Precision and recall, $k = 13$



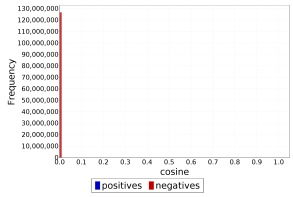
Cosine similarity, $k = 13$



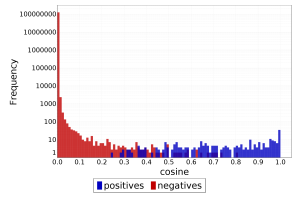
Cosine similarity (log), $k = 13$



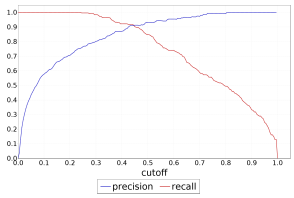
Precision and recall, $k = 14$



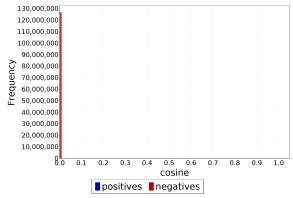
Cosine similarity, $k = 14$



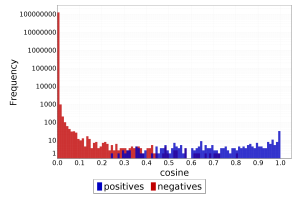
Cosine similarity (log), $k = 14$



Precision and recall, $k = 15$



Cosine similarity, $k = 15$



Cosine similarity (log), $k = 15$

APPENDIX C

Project Plan and Auto-evaluation

Project Plan

Formalities

Title	Algorithms in Metagenomic Sequencing
Type	MSc thesis
Student	Tobias Bertelsen
DTU Advisors	Inge Li Gørtz and Philip Bille
External Advisor	Manimozhiyan Arumugam
External Institution	The Novo Nordisk Foundation Center for Basic Metabolic Research University of Copenhagen
ETCS	35 ECTS points
Evaluation	7 step scale
Period	01.12.2014 – 01.06.2015

Project Description

The project will investigate the area of metagenomic sequencing, which is the study of DNA sequences in complex microbial environments. The goal is to identify problems in this area, where the current state-of-the-art algorithms can be improved. Focus will be on scalability, due to the massive data sizes within bioinformatics.

Teaching Goals

The student will show he is able to:

1. Identify and describe important computer science problems within metagenomic sequencing.
2. Analyze the current state-of-the-art solutions to these problems.
3. Design new algorithms based on experiences with existing state-of-the-art algorithms.

4. Implement a prototype of the new algorithm(s).
5. Analyze and evaluate the efficiency of the solutions from a theoretical and practical perspective.
6. Document the work in a written thesis.

Revisions of the Project Plan

The period for the project was extended twice due to a severe repetitive strain injury in my fingers, which prevented me from typing on a computer. The final due date ended up being August 24, 2015.

Auto-evaluation

The teaching goats has been fully fulfilled:

1. The problem *metagenomic gene clustering* has been identified and described (Definition 2.1). It is an important computer science problem, since the nature of gene alignment makes it impossible to use common indexing strategies. An efficient solution will ever contribute to the computer scientific knowledge.
   In addition the problem *proxy-based center selection* has been identified (Problem 9.1), though solving it is outside the scope of this thesis.
2. CD-Hit and UClust has been analyzed as current state-of-the-art solutions.
3. Proxy-based Efficient Clustering Algorithm (PECA) has been designed
   - The proxies for gene identity is based on experience with UClust.
   - The generalized DISCO algorithm is based on experience with the original DISCO algorithm.
4. A fully functional prototype of the algorithm has been implemented.
5. PECA and the generalized DISCO algorithm has been analyzed and evaluated from a theoretical perspective in Chapters 6 and 7 and from a practical perspective in Chapters 8 and 9.
6. The work is documented in this thesis.

APPENDIX D

Source Code

The source code will be published on GitHub at
`https://github.com/tbertelsen/thesis-public`.

Bibliography

[Alt+90] Stephen F Altschul et al. "Basic local alignment search tool". In: *Journal of molecular biology* 215.3 (1990), pages 403–410.

[Art15] Data Artisans. *GitHub page for flink-dataflow*. 2015. URL: https://github.com/cloudera/flink-dataflow (visited on July 26, 2015).

[Ber15] Tobias Bertelsen. *GraphX Performance: Partition overhead scales quadratically*. 2015. URL: https://issues.apache.org/jira/browse/SPARK-9937 (visited on August 14, 2015).

[Clo15] Cloudera. *GitHub page for spark-dataflow*. 2015. URL: https://github.com/cloudera/spark-dataflow (visited on July 26, 2015).

[Cza15] Grzegorz Czajkowski. *Big data is easier than ever with Google Cloud Dataflow*. 2015. URL: http://googlecloudplatform.blogspot.dk/2015/04/big-data-is-easier-than-ever-with-Google-Cloud-Dataflow.html (visited on July 26, 2015).

[DB79] David L Davies and Donald W Bouldin. "A cluster separation measure". In: *Pattern Analysis and Machine Intelligence, IEEE Transactions on* 2 (1979), pages 224–227.

[DG08] Jeffrey Dean and Sanjay Ghemawat. "MapReduce: simplified data processing on large clusters". In: *Communications of the ACM* 51.1 (2008), pages 107–113.

[Edg04] Robert C Edgar. "Local homology recognition and distance measures in linear time using compressed amino acid alphabets". In: *Nucleic acids research* 32.1 (2004), pages 380–385.

[Edg10] Robert C Edgar. "Search and clustering orders of magnitude faster than BLAST". In: *Bioinformatics* 26.19 (2010), pages 2460–2461.

[Edg15a] Robert C Edgar. *CD-HIT misalignment due to banding*. 2015. URL: http://www.drive5.com/usearch/cdhit_band.html (visited on July 26, 2015).

[Edg15b] Robert C Edgar. *CD-HIT-EST alignment parameters*. 2015. URL: http://drive5.com/usearch/cdhit_params.html (visited on July 26, 2015).

[Edg15c] Robert C Edgar. *Definitions of pair-wise sequence identity*. 2015. URL: http://www.drive5.com/usearch/id_defs.html (visited on August 9, 2015).

[Fou14] The Apache Software Foundation. *Welcome to Apache Hadoop*. 2014. URL: https://hadoop.apache.org/ (visited on July 25, 2015).

[Fou15a] The Apache Software Foundation. *GraphX Programming Guide*. 2015. URL: https://spark.apache.org/docs/latest/graphx-programming-guide.html (visited on July 25, 2015).

[Fou15b] The Apache Software Foundation. *Job Scheduling*. 2015. URL: http://spark.apache.org/docs/latest/job-scheduling.html (visited on August 15, 2015).

[Fou15c] The Apache Software Foundation. *Spark Examples*. 2015. URL: http://spark.apache.org/examples.html (visited on July 25, 2015).

[Fou15d] The Apache Software Foundation. *Spark - Lightning-fast cluster computing*. 2015. URL: http://spark.apache.org/ (visited on July 25, 2015).

[Fou15e] The Apache Software Foundation. *Spark Overview*. 2015. URL: https://spark.apache.org/docs/latest/index.html (visited on July 25, 2015).

[Fou15f] The Apache Software Foundation. *Spark Programming Guide*. 2015. URL: https://spark.apache.org/docs/latest/programming-guide.html (visited on July 25, 2015).

[Fou15g] The Apache Software Foundation. *Spark Release 1.4.0*. 2015. URL: https://spark.apache.org/releases/spark-release-1-4-0.html (visited on August 15, 2015).

[Fou15h] The Apache Software Foundation. *Tuning Spark*. 2015. URL: http://spark.apache.org/docs/latest/tuning.html (visited on August 15, 2015).

[GM12] Ashish Goel and Kamesh Munagala. "Complexity measures for map-reduce, and comparison to parallel computing". In: *arXiv preprint arXiv:1211.6526* (2012).

[Hou15] Juliet Hougland. *scala-dataflow-dsl*. 2015. URL: https://github.com/jhlch/scala-dataflow-dsl (visited on August 15, 2015).

[Ju15] Han Ju. *Scalaflow*. 2015. URL: https://github.com/darkjh/scalaflow (visited on August 15, 2015).

[Koz13] Mykhailo Kozik. *Java Magic. Part 4: sun.misc.Unsafe*. 2013. URL: http://mishadoff.com/blog/java-magic-part-4-sun-dot-misc-dot-unsafe/ (visited on August 15, 2015).

[KY15] Eric Kaczmarek and Liqi Yi. *Taming GC Pauses for Humongous Java Heaps in Spark Graph Computing*. 2015. URL: http://www.slideshare.net/SparkSummit/kaczmarek-yi (visited on August 15, 2015).

[Le +13] Emmanuelle Le Chatelier et al. "Richness of human gut microbiome correlates with metabolic markers". In: *Nature* 500.7464 (2013), pages 541–546.

[Lee15] Sangmin Lee. *How to Tune Java Garbage Collection*. 2015. URL: http://www.cubrid.org/blog/dev-platform/how-to-tune-java-garbage-collection/ (visited on August 15, 2015).

[LG06] Weizhong Li and Adam Godzik. "Cd-hit: a fast program for clustering and comparing large sets of protein or nucleotide sequences". In: *Bioinformatics* 22.13 (2006), pages 1658–1659.

[Mas+13] Matt Massie et al. "Adam: Genomics formats and processing patterns for cloud scale computing". In: *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2013-207* (2013).

[Met+12] Barbara a. Methé et al. "A framework for human microbiome research". In: *Nature* 486.7402 (June 2012), pages 215–221. ISSN: 0028-0836. DOI: 10.1038/nature11209. URL: http://www.nature.com/doifinder/10.1038/nature11209.

[NW70] Saul B Needleman and Christian D Wunsch. "A general method applicable to the search for similarities in the amino acid sequence of two proteins". In: *Journal of molecular biology* 48.3 (1970), pages 443–453.

[Ora15] Oracle. *Java Platform, Standard Edition HotSpot Virtual Machine Garbage Collection Tuning Guide*. 2015. URL: https://docs.oracle.com/javase/8/docs/technotes/guides/vm/gctuning/toc.html (visited on August 14, 2015).

[PBM04] Malay K Pakhira, Sanghamitra Bandyopadhyay, and Ujjwal Maulik. "Validity index for crisp and fuzzy clusters". In: *Pattern recognition* 37.3 (2004), pages 487–501.

[Pet+15] Thomas Nordahl Petersen et al. "Meta-genomic analysis of toilet waste from long distance flights; a step towards global surveillance of infectious diseases and antimicrobial resistance". In: *Scientific reports* 5 (2015).

[Pla15a] Google Cloud Platform. *Google Cloud Dataflow*. 2015. URL: https://cloud.google.com/dataflow/ (visited on July 25, 2015).

[Pla15b] Google Cloud Platform. *Google Genetics*. 2015. URL: https://cloud.google.com/genomics/ (visited on August 16, 2015).

[Pla15c] Google Cloud Platform. *WordCount Example Pipeline*. 2015. URL: https://cloud.google.com/dataflow/examples/wordcount-example (visited on July 25, 2015).

[Qin+12] Junjie Qin et al. "A metagenome-wide association study of gut microbiota in type 2 diabetes". In: *Nature* 490.7418 (2012), pages 55–60.

[Qub15a] Qubole. *Debugging Spark Jobs*. 2015. URL: http://docs.qubole.com/en/latest/user-guide/spark/debugging-spark.html (visited on August 20, 2015).

[Qub15b] Qubole. *Qubole – Click to Query Your Big Data*. 2015. URL: http://www.qubole.com/ (visited on August 20, 2015).

[Rou87] Peter J Rousseeuw. “Silhouettes: a graphical aid to the interpretation and validation of cluster analysis”. In: *Journal of computational and applied mathematics* 20 (1987), pages 53–65.

[Sch15] Eric Schmidt. *Announcing General Availability of Google Cloud Dataflow and Cloud Pub Sub*. 2015. URL: http://googlecloudplatform.blogspot.dk/2015/08/Announcing-General-Availability-of-Google-Cloud-Dataflow-and-Cloud-Pub-Sub.html (visited on August 15, 2015).

[Ser15a] Amazon Web Services. *Apache Spark on Amazon EMR*. 2015. URL: https://aws.amazon.com/elasticmapreduce/details/spark/ (visited on August 20, 2015).

[Ser15b] Amazon Web Services. *Configure Spark*. 2015. URL: https://docs.aws.amazon.com/ElasticMapReduce/latest/ReleaseGuide/emr-spark-configure.html (visited on August 20, 2015).

[Ser15c] Amazon Web Services. *Resize a Running Cluster*. 2015. URL: http://docs.aws.amazon.com/ElasticMapReduce/latest/ManagementGuide/emr-manage-resize.html (visited on August 20, 2015).

[SLS10] Michael C Schatz, Ben Langmead, and Steven L Salzberg. “Cloud computing and the DNA data race”. In: *Nature biotechnology* 28.7 (2010), page 691.

[Smi15] Eishay Smith. *JVM Serializers*. 2015. URL: https://github.com/eishay/jvm-serializers/wiki (visited on August 15, 2015).

[Sof15] Esoteric Software. *Kryo*. 2015. URL: https://github.com/EsotericSoftware/kryo (visited on August 15, 2015).

[Ste+10] Lincoln D Stein et al. “The case for cloud computing in genome informatics”. In: *Genome Biol* 11.5 (2010), page 207.

[SW81] Temple F Smith and Michael S Waterman. “Identification of common molecular subsequences”. In: *Journal of molecular biology* 147.1 (1981), pages 195–197.

[WH15] Daoyuan Wang and Jie Huang. *Tuning Java Garbage Collection for Spark Applications*. 2015. URL: https://databricks.com/blog/2015/05/28/tuning-java-garbage-collection-for-spark-applications.html (visited on August 15, 2015).

[Wik15a] Wikipedia. *Cosine similarity — Wikipedia, The Free Encyclopedia*. 2015. URL: https://en.wikipedia.org/w/index.php?title=Cosine_similarity&oldid=672752823 (visited on August 16, 2015).

[Wik15b] Wikipedia. *Davies–Bouldin index — Wikipedia, The Free Encyclopedia*. 2015. URL: https://en.wikipedia.org/w/index.php?title=Davies%C3%A2%C2%80%C2%93Bouldin_index&oldid=653340461 (visited on August 9, 2015).

[Wik15c] Wikipedia. *Dunn index — Wikipedia, The Free Encyclopedia*. 2015. URL: https://en.wikipedia.org/w/index.php?title=Dunn_index&oldid=668791825 (visited on August 9, 2015).

[Wik15d] Wikipedia. *Pearson product-moment correlation coefficient — Wikipedia, The Free Encyclopedia*. 2015. URL: https://en.wikipedia.org/w/index.php?title=Pearson_product-moment_correlation_coefficient&oldid=674309491 (visited on August 16, 2015).

[Wik15e] Wikipedia. *Silhouette (clustering) — Wikipedia, The Free Encyclopedia*. 2015. URL: https://en.wikipedia.org/w/index.php?title=Silhouette_(clustering)&oldid=674814193 (visited on August 9, 2015).

[Xin15] Reynold Xin. *Spark officially sets a new record in large-scale sorting*. 2015. URL: https://databricks.com/blog/2014/11/05/spark-officially-sets-a-new-record-in-large-scale-sorting.html (visited on July 26, 2015).

[XR15] Reynold Xin and Josh Rosen. *Project Tungsten: Bringing Spark Closer to Bare Metal*. 2015. URL: https://databricks.com/blog/2015/04/28/project-tungsten-bringing-spark-closer-to-bare-metal.html (visited on August 15, 2015).

[Zah+10] Matei Zaharia et al. "Spark: cluster computing with working sets". In: *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*. Volume 10. 2010, page 10.

[Zah+12] Matei Zaharia et al. "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing". In: *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. USENIX Association. 2012, pages 2–2.

[ZG13] Reza Bosagh Zadeh and Ashish Goel. "Dimension independent similarity computation". In: *The Journal of Machine Learning Research* 14.1 (2013), pages 1605–1626.

{2.7182818284}ϑφερτυθιοπσδφγηξκλ