NemoCluster: Graph Clustering Algorithm for Structural Variant Detection

Nicola Rohde

A thesis
submitted in partial fulfillment of the
requirements for the degree of

Master of Science in Computer Science and Software Engineering

University of Washington

2020

Committee:

Wooyoung Kim

Clark Olson

Erika Parsons

Program Authorized to Offer Degree:
Computing and Software Systems

University of Washington

Abstract

NemoCluster: Graph Clustering Algorithm for Structural Variant Detection

Nicola Rohde

Chair of the Supervisory Committee:

Dr. Wooyoung Kim

Computing and Software Systems

Structural Variant detection is a problem of significant interest in the biomedical field due to the strong link between these variants and genetic and degenerative diseases. A large body of programs and approaches exist to detect these variants and they perform well on the human genome. However, benchmarks presented in this thesis show that these tools perform poorly on microbial genomes. One approach that has been shown to be effective in structural variant discovery is the use of clustering to detect anomalous regions in the genome. Well known tools such as DELLY use this approach to achieve high accuracy, however, no tools use a network-motif based clustering algorithm.

The idea of anomalous genomic regions can be likened to community detection in social networks. This can be achieved by utilizing triangle-subgraphs, or size three cliques, to calculate a triangle conductance for each edge in the network. However, using just cliques ignores a large amount of structural information within the network. This is fine in social networks where cliques represent tightly-nit groups and therefore have more significance than other structures. This however, does not extend well to other areas such as Bioinformatics, where it may be of interest to cluster networks based on network-motifs to capture more structural information contained within the graph than can be conveyed through cliques.

This thesis introduces NemoCluster, an algorithm that generalizes the triangle conductance clustering to a network-motif conductance clustering. Accompanying this program are benchmarks that show it performing better than similar tools in both social networking applications as well as biological applications, such as protein-protein interaction networks, and in synthetic networks.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

GLOSSARY

ALIGNMENT: (1) A genome reconstructed from reads using a reference genome; (2) The process of reconstructing a genome using reads and a reference genome.

BASE PAIR (BP): A single letter in a DNA sequence representing one of the four nucleobases and its complement-base.

CLIQUE: A complete graph, i.e. a graph that has the maximum number of edges possible.

CLUSTERING: The process of grouping vertices in a graph that are similar to each other and dissimilar to vertices in other groups by removing edges.

CONNECTED GRAPH: A graph in which every vertex can be reached from every other vertex by following the edges present in the graph.

COVERAGE: The average number of reads expected to be at any location within the genome.

CUT: A set of edges that, when removed, turn a connected graph into a disconnected graph.

DISCONNECTED GRAPH: A graph in which there exist at least one pair of vertices that are not connected by a path of edges.

EDGE: A relationship between two vertices in a graph. This relationship can have a numerical weight and/or direction associated with it.

GRAPH: A collection of edges and vertices.

MINIMAL CUT: The smallest possible cut that will turn a connected graph into a disconnected graph.

NETWORK: See Graph.

NETWORK MOTIF: A statistically over-represented subgraph that is found via a Monte-Carlo simulation or approximation algorithm.

NODE: See Vertex.

PARTITION: The connected components left over by applying a cut to a graph are referred to as partitions.

READ: A short strand of DNA of 100-1000 bp. Reads are produced in shotgun sequencing and are used to reconstruct the original DNA sequence.

READ SPAN: The average distance between two paired-end or mate-pair reads.

STRUCTURAL VARIANT (SV): A large scale mutation of 50 bp or more. There are 5 main types, deletions, insertions, inversions, duplications, and translocations.

SUBGRAPH: A smaller graph contained within another graph.

TECTONIC: The clustering algorithm NemoCluster is based on. It uses triangle conductance to eliminate edges in a graph to create a disconnected graph; the connected components in that graph are the clusters.

VERTEX: An entity in a graph that represents the information stored within the graph.

ACKNOWLEDGMENTS

I want to thank my committee for their time and feedback throughout my research. A special thank you to Professor Kim whose guidance and encouragement made this thesis possible.

Chapter 1
INTRODUCTION

1.1 Overview

Structural Variants (SV) are large scale mutations in an organism's genome that have been linked to genetic and degenerative diseases such as cancer [28]. In bacteria, SVs can result in negative properties such as antibiotic-resistance, which makes previously curable conditions untreatable [30]. Thus, detecting these SVs is essential in developing cures for diseases such as cancer, but also in identifying dangerous strands of antibiotic-resistant bacteria.

A variety of tools exist that allow the detection of SVs in an organism's genome with some accuracy; while these perform well on the Human genome, they do not perform as well on microbial genomes [33]. This is likely due to the tuning parameters being tuned for human samples; thus, it would require a large amount of tuning for a researcher to make the tool accurate on microbial genomes. Researchers who are unaware of this may, incorrectly, assume that the experiments presented by the authors of the tools apply to microbial genomes and result in incorrect SVs being located. This establishes a need for a SV detection tool that is specifically designed with microbial genomes in mind, ensuring the tool performs well on this type of sample out-of-the-box.

This thesis introduces NemoCluster as a the first step in the process of building an SV detection tool customized for microbial genomes. It is a network clustering algorithm based on the use of network motifs. A number of prominent SV detection tools, such as DELLY [32], use network clustering algorithms to detect regions of interest in the genome, that can then be analyzed further to determine whether or not they contain SVs; the structure of the

clusters can also suggest which type of SV might be present in that region. Due to how these networks are constructed from the genome, they are unstable and not necessarily complete. The network motifs used by NemoCluster are statistically stable patterns in the network, therefore, NemoCluster could be a good alternative to these other clustering approaches.

NemoCluster is a generalization of an existing algorithm, Tectonic [37] that uses triangle subgraphs, known as cliques, to detect communities in social networks. In such an application, cliques represent tight-nit groups of people who all know each other, therefore being a good indicator of communities. In other fields, however, cliques do not necessarily have any special meaning in all applications, thus, clustering based on them may not produce the best results everywhere. Network motifs are a more general way of looking at this problem, they are more flexible and can be molded to fit most input graphs in a variety of domains. Thus, NemoCluster extends the functionality of Tectonic to use network motifs when finding clusters instead of cliques.

Experimental results show that NemoCluster performs either equal to, or better than Tectonic suggesting that the network motif based approach is superior to that of cliques in most circumstances. In social network analysis, cliques are still likely to outperform other network motifs since cliques represent tightly-nit communities.

This thesis will first introduce the domain, problem, and go over the motivations for this project in Chapter 1. Chapter 2 provides a general background and literature review will be presented to the reader to understand the domain and problem. After this, Chapter 3 will elaborate on the aforementioned benchmarking suite for structural variant tools; this is an extended version of a paper published in the BIBM 2019 conference [33]. In Chapter 4, the NemoCluster algorithms will be discussed, along with the architecture of the NemoCluster library and a detailed overview of the software development process that went into building it. Chapter 5 will provide an overview on the experiments conducted to analyze the perfor-

mance of NemoCluster, this will be followed by the analysis of these experiments in Chapter 6. Finally, Chapter 7 will discuss the conclusions of this thesis and a brief overview of future works. In addition to this, the source code for the complete NemoCluster Library can be found on GitLab.com.

1.2 Motivation

1.2.1 Overview

The ultimate goal of the project is to build an SV detection tool specifically designed for microbial genomes that performs well out-of-the-box. Based on a thorough literature review and benchmarks of existing tools, the decision was reached to base this tool on a clustering strategy that is commonly used and to build NemoCluster to provide this clustering functionality. Existing algorithms in the domain assume that the input network has an established and consistent structure, which may not be true in genome analysis, thus making them inappropriate for this setting. NemoCluster is expected to perform better as it does not make any assumptions about an established structure, instead utilizing the structure that is present making it ideal for unstable network analysis.

1.2.2 Initial Benchmarking

The first step of this project involved building a benchmarking suite and using it to benchmark several existing tools against microbial genomes to test the performance. These experiments revealed that the existing breadth of tools either does not perform well, or does not perform well on microbial genomes. The basis of this project was the assumption that these tools do perform well, but have been tuned for the human genome which is significantly larger and differently structured from microbial genomes. As such, the tools will perform poorly on microbial genomes and would require a large amount of tuning to properly call

structural variants in microbial genomes.

1.2.3 Dedicated Tool for Microbial Genomes

While it should, in theory, be possible to tune existing tools to work with microbial genomes and result in good performance, it is not guaranteed and it could also produce suboptimal results. The aim of this research was to make the first step in producing a dedicated tool for microbial genomes to resolve this issue. As this tool will be designed to work with this type of genome, there is no need for its users to tune it significantly to achieve good results, much like the human-genome equivalents, it will perform well out-of-the-box.

The research displayed in this thesis accomplishes the first step in this process. Nemo-Cluster is a clustering algorithm that can be used as a pre-processing stage to this variant caller which can then use the clusters and the graph-based read-pair and split-read structural variant (SV) detection schemes to find the variants. Furthermore, this research extends the Tectonic algorithm [37] from using a clique-based approach to a network-motif based approach. While the use of triangles utilizes some structure from the graph to detect clusters, it ignores most structural features present in the data. The use of network motifs remedies this by allowing all structural information to be utilized. This is a new way of approaching the problem and the results show that it is possible to improve clustering performances by using this approach.

1.2.4 Goals of the Research

There were two main goals of this research. Firstly, to evaluate the performance of existing structural variant calling tools on microbial genomes to establish a baseline for a future tool. In addition to this, the second goal was to design and implement a network-motif based clustering algorithm that could be used by a structural variant calling tool and to evaluate

its performance.

The first goal of building a benchmarking suite was motivated by the lack of information regarding the performance of existing SV calling tools on microbial genomes. Almost every tool is evaluated on the human genome; this is great for people who study the human genome, but not very useful to those who don't. By providing benchmarks conducted on microbial genomes, such as E. Coli, M. Maripaludis, and D. Vulgaris, researchers now have a way of selecting a tool based on performance metrics that used data very similar to their own. In addition to this, it showed that these tools do not perform well on microbial genomes which is different from the performance on the human genome. The experimental results make this performance difference clear to those who would otherwise look at the metrics published by the tools' authors and may, incorrectly, assume that they will perform just as well on microbial genomes.

Building a new clustering algorithm had the main goal of improving clusters by utilizing network motifs in the process of finding them. Network motifs are used to solve many different problems in Bioinformatics and this clustering algorithm was an exploration to determine whether or not clustering is another place where network motifs can be used as there isn't a significant body of research in this area of Bioinformatics.

1.2.5 Cliques to Network Motifs

NemoCluster is based on triangle conductance algorithm which uses size three cliques, or triangle motifs, to integrate some structural information from the graph into the clustering. This approach has the advantage of being significantly faster than using network motifs as clique-finding algorithms are much faster than network-motif finding algorithms.

However, this performance comes at the price of the information that is encoded in the

network motifs. Cliques only contain a small amount of useful information in read-graphs as they highlight sections of the genome where many reads overlap. This information can be useful when searching for structural variants; however, network motifs allow for more fine-tuning and should produce better results. Expanding this algorithm to use any type of motif allows this additional information to be included in the analysis for increased performance.

Chapter 2

BACKGROUND

2.1 Genome Sequencing Analysis

All living organisms have genetic code in the form of Deoxyribonucleic Acid (DNA) which is a long chain of the four nucleobases, Adenine, Cytosine, Guanine, and Thymine. These bases are present in pairs in the DNA's double helix structure; thus, they are generally referred to as "base pairs" (bp) and the length of DNA is usually given in the number of base pairs it contains. Small, microbial genomes have a length of several million base pairs, while large genomes, like the human genome, are several billion base pairs in length.

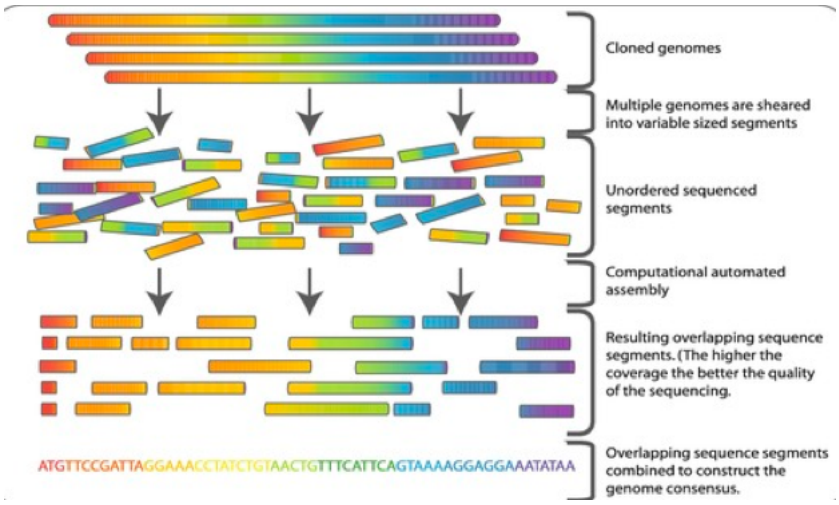

Figure 2.1: Genome sequencing process overview. The DNA is cloned, fragmented, sequenced, and reconstructed. ©2009 Commins et al. [7]

As shown in Fig. 2.1, the process of analyzing a genome involves several steps. This

particular method is referred to as shotgun sequencing which is commonly used to determine the exact base-pair sequence of an organism's genome. First, the genome is cloned a number of times, then it is fragmented into randomly sized pieces, these pieces are sequenced, and the sequenced pieces are then reconstructed into the original genome.

2.1.1 DNA Sequencing

Due to the size of an organism's genome, it is not possible to sequence it in it's entirety as modern sequencers can only read short DNA pieces between 100-1000 bp in length [31]. Shotgun sequencing [40] is a method of sequencing a large genome by first fragmenting it and then reconstructing it later.

First, the genome is cloned a number of times to produce overlap allowing higher accuracy during reconstruction. The average overlap at any point in the genome is referred to as coverage. High-coverage sequencing produces more accurate results, but also increases the cost and time investment.

After the genome has been cloned, the copies are fragmented by randomly chopping them up [31, 40], this is where shotgun sequencing gets its name. After this, small pieces of DNA are left-over that can be sequenced.

The DNA fragments are sequenced either from one side, called single-end, or both sides, called paired-end, into reads. The read is the actual DNA sequence of the portion of the fragment that was sequenced [40]. All reads have a fixed length regardless of fragment size. This means that with paired-end reads, there tends to be a gap between the two reads produced from each fragment, referred to as insert or read-span.

At the end of this process, a file contains millions of short, fixed-length reads that come from the genome [40]. The exact location where the read belongs in the organism's genome is unknown at this point, only the coverage, read-length, and average read-span are known. Using this information, the genome can be reconstructed computationally.

2.1.2 DNA Reconstruction

Shotgun sequencing produces a large number of short reads that are unordered as shown in Fig. 2.1 in the third step from the top. Reconstructing the organism's genome from this uses a process known as alignment [27].

The alignment process requires a reference genome to work [27]. This reference is a representative example of what the organism's genome should look like in the general case. As the reference represents only the average individual, each individual's genome will vary from this due to mutations [34].

The reads are aligned to the reference genome using a longest common substring (LCS) algorithm by finding the best spot to put them [27]. Some reads will not be aligned to the genome due to sequencing errors and mutations in the organism, these reads are referred to as discordant and are collected for analysis [27, 40]. In addition to this, reads can align to multiple places, these reads are referred to as split-reads and are also collected for analysis [27, 40].

After the reconstruction is finished, most reads have been assigned a location in the genome, shown in step 5 in Fig. 2.1. As there is some overlap between the reads, the DNA sequence is retrieved by taking the most common letter found at each position; this minimizes the impact of sequencing errors and misaligned-reads [27, 40]. The final result of this step is the sequenced genome of the organism.

2.1.3 Types of Genomes

Not all genomes are exactly alike, prokaryotes or single-cell organisms, such as bacteria, have a much smaller genome than eukaryotes or multi-cell organisms, such as humans [13]. While the human genome is about 1,000 times larger than the average bacterial genome, it only

contains about 10 times more genes [12, 13]. This means that the gene density of the human genome is about 100 times smaller than that of bacteria. One attributing factor of this is that humans have introns in genes [12], these are non-coding regions that are removed during transcription of the DNA. Bacteria have no introns which makes the genome more compact. The most significant difference between the genomes is that humans have 23 chromosomes in their DNA [12], while bacteria only have a single, circular plasmid [13].

2.2 Structural Variants

Sequencing an individual's DNA is done to analyze mutations present in the genome. There are two common types of mutations, single nucleotide polymorphisms (SNP) and structural variants (SV) [35].

An SNP is a single nucleotide deviation between the sequenced genome and the reference genome, this kind of mutation is often inconsequential and detecting them during alignment is trivial [31, 28].

Structural Variants are large scale mutations of more than 50 bp that can have a significant impact on the individual. Structural variants are also more important to locate than SNPs because it is possible for many SNPs to remain "silent." A silent mutation causes no actual change in the protein produced when the gene is expressed. Due to their size, SVs can damage genes, delete genes, or insert new genes which means they are of particular interest in the biomedical field [28]. A number of different SV types exist and can be seen in Fig. 2.2 such as deletions, insertions, inversions, tandem duplications, and translocations [35].

A deletion is a section of the reference genome that is not present in the sequenced genome, this is shown in row A of Fig. 2.2. Insertions are the opposite of deletions, in this case the sequenced genome contains a sequence of DNA that is not present in the reference,

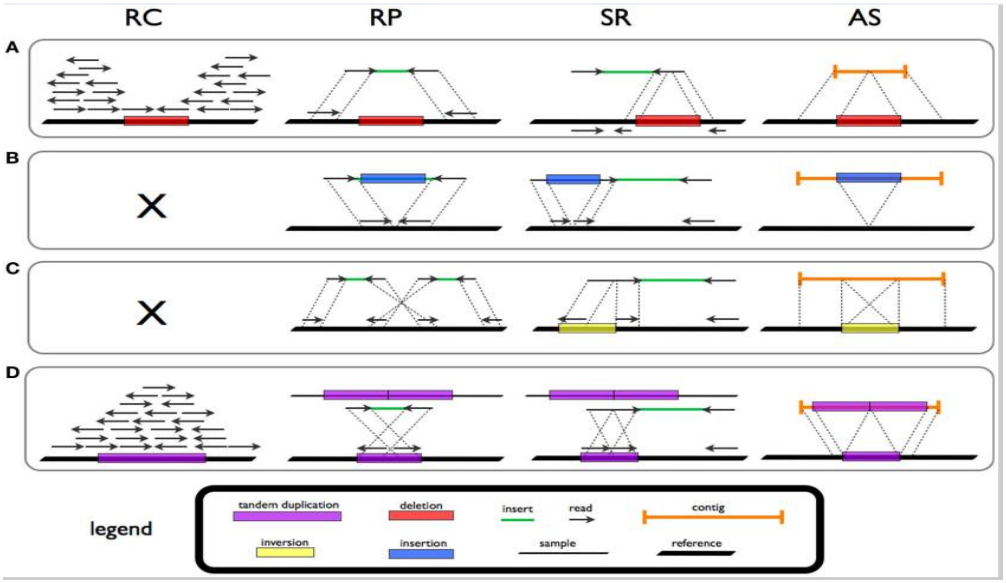

Figure 2.2: Approach of SV detection for deletions (A), insertions (B), inversions (C), and tandem duplications (D) using the read count (RC), read-pair (RP), split-read (SR), and de novo assembly (AS) methods. Adapted from [35]. Copyright © 2015 Tattini, D'Aurizio and Magi.

this is shown in row B of Fig. 2.2. Inversions are sequences of DNA that have been flipped around in the DNA; this means that the sequence occurs from left-to-right in the reference, but from right-to-left in the sequenced genome or vice-versa, this is shown in row C of Fig. 2.2. Tandem duplications are similar to insertions, as the name suggests, a sequence of DNA was copied multiple times and then inserted back-to-back, this is shown in row C of Fig. 2.2. Finally, translocations are variants that move a small piece of DNA from one chromosome to another.

Deletions, insertions, duplications, and inversions appear in microbial genomes. Translocations are unique to organisms with chromosomes and are thus not found in microbes.

The final step of genome analysis is to find where these SVs are located in the sequenced genome which is a non-trivial process. Firstly, the alignment process only allows a certain number of mismatches before flagging a read as discordant [27]. Thus, discordant and split reads are the main source of information about SVs as these are reads that could not be properly processed by the alignment tool [35].

2.3 Structural Variant Calling Methods

There are four common methods to finding SVs in a sequenced genome, namely, Read Count or Read Depth (RC), Read Pair (RP), Split Read (SR), and de novo assembly (AS). Each method has different strengths and not all methods can call all types of variants. Thus, methods are frequently combined to improve performance and cover more variant types.

2.3.1 Read Count (RC)

Fig. 2.2 shows this approach in the column labeled RC. This approach of variant calling assumes that the distribution of mapped reads over the genome follows a random distribution.

If an area diverges significantly from this expected distribution, i.e. the number of reads mapped to an area of the reference is much higher or much lower than expected, it suggests the presence of an SV [35]. As there are only two possibilities, this method can only call deletions and duplications. If the number of reads mapped to an area is significantly higher than expected, the area is called to be duplicated and based on how many more reads there are it is possible to guess as to how many times the area was duplicated. When an area has significantly fewer reads mapped to it than expected, the area is called to be deleted. It is not possible to call insertions, inversions, and translocations for this method.

2.3.2 Read Pair (RP)

Read pair analysis is based on the span and orientation of paired-end reads. This is illustrated in Fig. 2.2 in the column labeled RP [35]. Discordant reads where the span or orientation of the pair is inconsistent are collected and analyzed. If the span between the reads is further than expected, it implies a deletion has occurred resulting in them being closer together. Similarly, if the span is shorter than expected, it implies there is an insertion putting the reads further apart. Inconsistent orientation between the pairs can point to inversions and duplications. The read pair method cannot be used to call translocations.

2.3.3 Split Read (SR)

As the name suggests, this method focuses on analyzing the split reads found during the alignment process as is shown in Fig. 2.2 in the column labeled SR. Split reads are anchored to the reference genome on one side and the point of mismatch is identified as a breakpoint. These points can then be examined with different reads, allowing the identification of all types of variants with high precision [35]. Deletions mean that a read aligns with a gap. Similarly, insertions mean that the read contains a sequence that does not align to the reference, resulting in a gap on the read. As the breakpoints show the exact location where the read differs from the reference, it is possible to call the variant with single-base-pair

resolution.

2.3.4 De Novo Assembly (AS)

De novo assembly is the most precise way of finding structural variants in a genome. This process requires very long reads with high coverage to ensure that there is large overlaps between the reads to precisely reconstruct the original sequence [35]. Unlike with the other strategies, the alignment is not used, instead, reads are combined into large strands which have a high overlap in the reads called contigs. The contigs are then aligned to the genome and merged together into an assembly, as shown in Fig. 2.2 in the column labeled AS. In theory, this process can call all types of variants with single-base-pair resolution, but modern sequencing techniques cannot produce long enough reads to make this a feasible approach for most cases [31]. With this approach, a direct 1-to-1 comparison between the assembly and the reference genome would show variants, much like how SNPs are found.

2.4 Existing Structural Variant Detection Tools

2.4.1 Non-clustering Tools

Many SV detection tools use one or more of the methods discussed in Section 2.3 to analyze the reads and alignment information to find SVs. A small selection of these will be discussed here as they were used in benchmarks that will be presented in Section 3; these are, CNVNator (RC) [1], Breakdancer Max (RP) [11], and Gustaf (SR).

CNVNator (CNVN) [1] is the simplest of these tools. It uses the RC method which is a statistical method for identifying deletions and duplications in a genome. The number of reads aligned to regions of the genome are analyzed to determine if there is a statistically significant number more or less at any position in the genome. In addition to this, read

quality is considered during the analysis to account for sequencing errors that could have produced unexpected read coverage. Any regions that have an unexpected amount of read coverage after this analysis are considered to be either deletions or duplications.

Breakdancer Max (BDM) [11] is a tool that is frequently used due to how fast it provides results. As it uses the RP method, it can call most types of SVs, with the notable exception of duplications. BDM analyzes the alignment to calculate statistics about the read-mapping and then analyzes the reads to discover discordantly-mapped reads. The discordant reads are then analyzed to determine what kind of SV could produce such reads which are then outputted as the calls of the program.

Gustaf [36] uses a graph-based approach to finding SVs by creating a split-read graph from the reads and alignment. Breakpoints are calculated in the alignment to find which reads should be adjacent, or connected by an edge, in the graph. The edges are then weighted according to the edit distance of the reads within the vertices it connects. This weight is used as a penalty during SV detection. The higher the penalty, the less support there is for a variant call. The analysis is based mainly on standard graph algorithms such as shortest path, which is used to identify groups of reads that may belong to the same SV. The SV classification of these groups then depends on how the reads relate to each other and how they aligned to the reference genome.

2.4.2 Clustering Tools

A number of existing tools utilize a clustering component in addition to the methods discussed in Section 2.3. Three prominent examples are DELLY [32], SVDetect [44], and LUMPY [24] that will be discussed here.

DELLY [32] is one of the most commonly used structural variant detection tools. It uses

a mix of the SR and RP methods together with clustering. The discordant reads are used to build an undirected, weighted graph where edges between reads mean that the reads both support the presence of a particular structural variant. Clusters are produced by extracting the maximal cliques from each connected component in the graph. This means that not all vertices in the graph will be assigned to a cluster which makes the algorithm less sensitive to sequencing errors. The clusters are then used to predict the genomic region where SVs occur and they can also be used to determine what kind of SV is located there. DELLY takes these regions and analyzes the split reads to detect breakpoints to find where the SVs begin and end as well as the exact type.

Another SV detection tool that uses clustering is SVDetect (SVD) [44]. SVD uses a mix of the RC and RP methods and uses clustering, much like DELLY, to analyze discordantly mapped reads. The graph of these discordant reads is built such that each vertex represents a group of reads in the same region of the genome. Edges are used to represent overlap between the groups, i.e. if two groups contain the same read, they will be connected by an edge. Certain properties are assigned to the edges, such as number of reads in the overlap, orientation, and order of the reads. This graph is then clustered by filtering the edges using a user-defined filtering parameter. This results in clusters of reads that are then annotated with an SV type they support. The RC and RP methods are then applied in addition to this to analyze the genome further for additional SVs.

Finally, LUMPY [24] uses probability distributions to determine where breakpoints are located in the genome. Any two bases that are adjacent in the reads but not on the reference genome are marked as potential breakpoints and a probability is assigned to each breakpoint to account for sequencing and alignment errors. The overlapping breakpoints are the clustered together and the probabilities are combined. The clusters are then analyzed to determine if they provide sufficient support for a SV and those that do are returned as SV calls. In addition to this clustering approach, the SR and RP methods are also used to

further analyze the genome for higher accuracy.

2.5 Graphs

Some types of relational data can be expressed in the form of a Graph or Network. This data structure is made up of two components, the vertices (a.k.a. nodes) which are the entities stored in the collection and the edges which represent the relationships between the vertices [25]. A graph is often denoted as a tuple of these two sets, i.e. $G = (V, E)$ refers to a graph G which contains the vertices in set V and the edges in set E. The edge-set, E, contains the relationships between vertices. An edge is a tuple storing up to four components, namely, the two vertices on either end, a weight, and a direction. The latter two components are optional, but all edges are at least a pair of vertices. In addition to this, vertices have a degree (a.k.a. valency) which is the number of edges that connect to it, this is generally denoted by $deg(v)$ for some vertex v [25]. If the degree of all vertices in a graph is maximal, the graph is referred to as complete, denoted by K_n, where n is the number of vertices in the graph. Graphs have a conductance measure based on how close they are to being complete. Graphs with high conductance are referred to as dense graphs and are very close to being complete. On the other hand, graphs with low conductance, referred to as sparse graphs, are far from complete [25].

Due to the two optional components associated with the edges, weight and directedness, there are four types of graphs, namely, unweighted-undirected, unweighted-directed, weighted-undirected, and weighted-directed [25]. Examples of all types of graphs are shown in Fig. 2.3.

When working with a graph, it is possible to look at subgraphs contained within the original graph; i.e. given $G = (V, E)$ a subgraph would be $G' = (V', E')$ where $V' \subseteq V$ and $E' \subseteq E$.

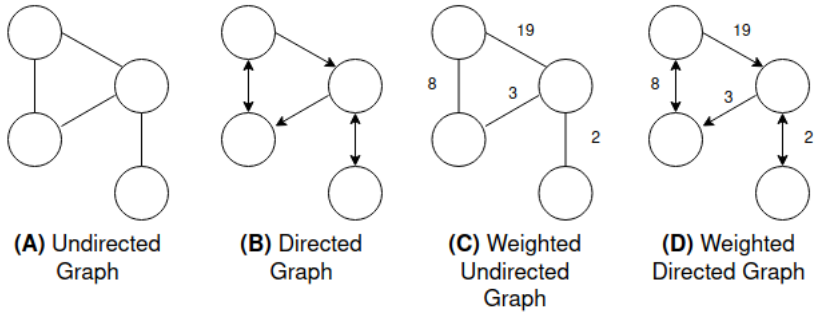

Figure 2.3: An illustration of (A) an unweighted-undirected graph, (B) an unweighted-directed graph, (C) a weighted-undirected graph, and (D) a weighted-directed graph.

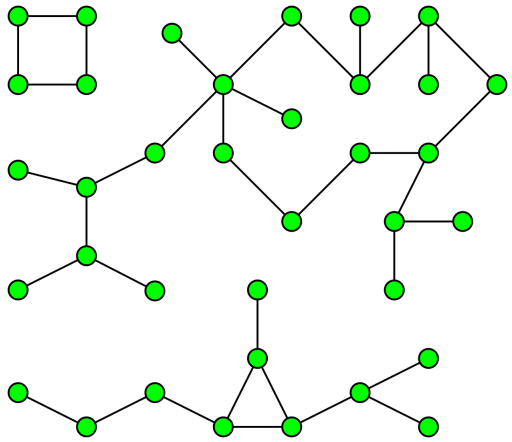

Figure 2.4: An illustration of a disconnected graph with three connected components. Adapted from David Eppstein [10].

Finally, graphs can be connected or disconnected. If a graph is connected, it means that every vertex is connected to all other vertices via a path of edges [25]. If a graph contains two vertices that are not connected by at least one path of edges, the graph is considered disconnected. An example of such a graph is shown in Fig. 2.4. A disconnected graph is made up of connected components, which are subgraphs that are connected graphs [25].

The example in Fig. 2.4 shows three such components. The connectivity of a graph is the minimum number of edges that must be removed such that the connected graph G is turned into a disconnected graph G'. The process of removing edges to turn a connected graph into a disconnected graph is called a cut, and a minimal cut is the smallest number of edges that must be removed to turn G into G' [17].

2.6 Read Graphs

In order for graph-based tools to analyze a genome and call SVs, the problem must first be translated into a graph. This process involves taking the discordantly mapped reads and turning them into a graph in one of two main ways.

The first approach, used by DELLY [32], sorts the discordant reads by the left-most position they aligned to and the chromosome that point is on. This sorted list of reads is then converted into a weighted, undirected graph. All reads are turned into nodes in this graph and edges between reads mean they support the same SV. This is achieved by insuring that any pair connected by an edge have the same orientation change and that the difference in their left-most alignment point is within the expected insert size range. The weight of the edges comes from the absolute difference between the mapping location of the two reads in question.

The resulting read graph is then clustered by identifying maximal-cliques in the graph. Any singleton clusters in the read graph are discarded during this process as anomalies. Each maximal clique can then be used to approximate start and end of the SV that it contains.

The other way of building a read-graph comes from SVDetect [44]. In this approach, the genome is divided into overlapping windows of a fixed size. These windows are then considered the vertices in the graph and they can form edges between each other if at least

one read-pair is contained in both of them. Each link is then given a number of features that describes the overlap.

Clustering this graph involves removing edges by analyzing the annotated properties of the edges and user-defined filtering parameters. During the filtering, clusters are annotated with potential SVs they support. Inversions are annotated as one window containing mostly reads aligned in one direction, with a linked window containing mostly reads aligned in the opposite direction. Deletions and insertions are called based on the insert size of the reads within a cluster. Translocations can be detected by filtering out reads that are inconsistent with the majority of the cluster.

2.7 Network Motifs

A network motif is a statistically over-represented subgraph in a graph [41]. A motif has a size, which is the number of vertices in the subgraph, and may be directed or undirected. The process of finding network motifs in a target graph involves two steps, calculating the frequency of all possible motifs and calculating the expected frequency using random graphs [41, 2]. First, all subgraphs of a particular size are enumerated to calculate the frequency of each subgraph. Then a number of random graphs is generated that share the same degree distribution as the original graph and the same process of calculating the frequency of each subgraph is repeated [41]. After that, the results from the random graphs are averaged to determine the expected frequency of each subgraph. All subgraphs that are significantly more frequent than the expected frequency are then considered network motifs [41, 2]. This process makes finding network motifs very slow in larger graphs as runtime grows exponentially with the density of the graph.

NemoLib [2] is a program that allows the detection of these network motifs in a graph; it also provides functionality to collect all subgraphs that are motifs and save them to a file [22].

2.8 Graph Clustering

2.8.1 Overview

Clustering is a form of unsupervised learning and is often used to analyze big data. The goal of a clustering algorithm is to group elements of a set that are similar into clusters. Members of the same cluster are similar to each other but dissimilar from elements in other clusters [9].

Applying a clustering algorithm to a graph is slightly different from applying it to a set because relationships already exist between vertices in the graph, namely, the edges. Clustering graphs utilizes these existing relationships by systematically removing edges to turn a connected graph into a disconnected graph [9]. After the clustering, all connected components within the disconnected graph are then considered to be the clusters produced by the algorithm.

2.8.2 Minimum Spanning Tree

One approach to clustering vertices in a graph is to first find a minimum spanning tree (MST) and then to cluster this tree instead. The MST will already have most edges removed, which means only a few edges must be removed to produce a disconnected graph. One such algorithm was proposed by Jana et al. [20] which uses a threshold value to remove edges with a weight larger than the threshold. The threshold value is found by iteratively testing different thresholds and calculating the score of the clusters and then choosing the threshold that maximizes the score.

2.8.3 Shared Nearest Neighbor

The shared nearest neighbor (SNN) clustering algorithm is based on the assumption that clusters will be densely connected within a graph. One implementation of this is the Jarvis Patrick algorithm [21], which removes edges between vertices that do not have enough neighbors in common in their neighborhoods. The number of neighbors, k, that must be shared and the neighborhood radius (i.e. how many edges away) are user-defined and can be used to tune the algorithm's performance.

2.8.4 Highly Connected Subgraphs

The highly connected subgraph (HCS) algorithm [17] divides a graph into clusters by finding a minimum cut that creates a disconnected graph. The algorithm is recursively applied to the resulting partitions of the previous stage while there are highly connected subgraphs remaining in the partitions; highly connected is defined as having a connectivity larger than $n/2$ for a graph containing n vertices in this case. At the end, all partitions created by the algorithm are considered the clusters.

2.8.5 Tectonic

Tectonic [37] is a clustering algorithm that utilizes triangle network motifs to find communities in social networks. The triangle motifs are size three cliques that can be discovered within the graph using a clique finding algorithm. Enumerating all cliques in a graph is a much simpler and faster process than finding network motifs as it only requires to enumerate a small fraction of subgraphs and no monte-carlo simulation is needed to calculate the statistical information.

The main component of Tectonic is the reweighting of the edges using the triangle-

count $t(u, v)$ of an edge connecting vertices u and v. The new weight of edge (u, v) will be $\frac{t(u,v)}{deg(u)+deg(v)}$, which produces a value in the range [0,1]. Once the graph is reweighted, the edges are filtered by removing those with a weight less than some threshold λ. After this, the graph will be disconnected and the connected components are the clusters produced by the algorithm. Tectonic has shown very good performance results in social networks and synthetic networks.

2.8.6 MAPPR

An algorithm similar to NemoCluster is the Motif-based Approximate Personalized PageRank (MAPPR) algorithm introduced in [43]. MAPPR, much like NemoCluster, uses a version of conductance to reweigh a graph and then produce clusters using the reweighed edges. This algorithm is motif-based. However, it uses an approximation of the motifs rather than the actual motifs. In addition to this, it attempts to find clusters given a seed-vertex in the graph, thus it does not allow arbitrary clustering without any initial information about clusters like NemoCluster does.

2.8.7 Tensor Spectral Clustering

The Tensor Spectral Clustering (TSC) algorithm, first proposed in [4], clusters graphs based on a specific network motif, much like NemoCluster. The main difference between NemoCluster and TSC is that TSC attempts to preserve the selected motif in the graph while removing edges that are not part of these motifs. This results in similar behavior to that of NemoCluster. However, NemoCluster does not actively attempt to preserve motifs and it also allows the clustering to use multiple motifs, something TSC does not. As a result, TSC will produce large clusters by removing few edges. In the applications that NemoCluster is intended for, this is unlikely to be a desired result as read-graph will, in theory, contain a

large number of smaller clusters representing anomalous genomic regions.

Chapter 3

BENCHMARKING STRUCTURAL VARIANT DETECTION TOOLS

3.1 Overview

Presented in this chapter is an extended version of a paper published in the IEEE BIBM 2019 conference [33]. This chapter contains benchmarking results from testing different Structural Variant (SV) calling tools representing most of the methods available, with the notable exception of de novo assembly (AS). The AS method was not benchmarked here because there were no tools available that worked for the whole-genome sequencing (WGS) paired-end reads used for these benchmarks. The tools that were selected are CNVNator, which uses the read count (RC) method [1]; Breakdancer-Max, which uses the read pair (RP) method [11]; Gustaf, which uses the split read (SR) method [36]; SVDetect, which uses the RC and RP methods [44]; and LUMPY, which uses the RP and SR methods [24].

These tools were selected based on the work of Tattini et al. [35] and were picked to fit the available data while also being representative of the different methods. Tools that are used relatively infrequently were purposely selected to provide alternatives over the popular tools such as Delly [32] and Pindel [42]. It was not necessary to provide comparisons between these common tools the tools selected for this study because the authors of these tools already provide these benchmarks in their own evaluation.

The benchmarks here do not contain a tool that uses the AS method, or the combination of the RC and SR methods, or any tool that uses more than two methods combined. There are tools available that use these approaches. However, they either did not support WGS

paired-end reads or it was not possible to locate a functional version of the tool for the Linux environment where these benchmarks were conducted.

3.2 Experiment

3.2.1 Tool Selection Criteria

SV detection tools were selected based on three main categories: 1) they must support next-generation whole-genome sequencing (WGS) data for microorganisms with paired-end or mate-pair reads; 2) they should not be commonly used in most studies; and 3) they should cover most of the structural variant calling strategies without being redundant.

3.2.2 Data simulation

The data used for experiments was simulated using RSVSim [3] for SVs generation and ART [18] to produce simulated reads from the selected genomes. RSVSim generates a list of all SVs that were inserted into the genome along with the exact location in the altered genome. Using this list, it is possible to evaluate the accuracy of each tool without the ambiguity of using real structural variants where the exact type, location, and sequence may not be known. In addition to this, ART can simulate real reads by mimicking a sequencing machine and randomly inserting sequencing errors based on how frequently the selected machine makes these errors. These additional errors create some noise making the benchmarks more realistic. During the evaluation process, these small errors, which are actually single nucleotide polymorphisms were not considered as the criteria for evaluation was solely the tool's ability to call SVs.

A total of three microorganisms were used for the benchmarks which represent a very small, medium, and large sized microbial genome. The three microbes used are *Methanococ-*

cus Maripaludis (M. Maripaludis), *Desulfovibrio Vulgaris (D. Vulgaris)*, and *Escherichia Coli (E. Coli)*. The reference genomes for these microbes were retrieved from [8] and NC_015847 (M. Maripaludis X1 strain), NC_008751 (D. Vulgaris strain DP4), NC_12967 (E. Coli B strain) were used for the organisms, respectively. It should be noted that only the main genome was used during the benchmarking and additional plasmids that the microbes may have were not considered for simplicity.

The simulated SVs are insertions, inversions, deletions, and duplications; inter-chromosomal translocations were not included as the chosen microorganisms only have a single chromosome; intra-chromosomal translocations are supported as mobile-elements insertions by RSVSim and were included in the insertions category. Simulated SVs were within one of two size ranges, small (100-1,000 bp in length) and large (10,000 - 15,000 bp in length). Each genome contains a 10:1 ratio of small to large variants for all simulated data-sets. In total, 15 altered genomes were created (5 data-sets for each genome) and reads were simulated for each with 20x, 30x, 40x, and 50x coverage for a total of 60 different data-sets. Table 3.1 summarizes the five types of data-sets for each genome. The simulated reads also contained random sequencing errors produced by ART.

3.2.3 Tools

Tattini et al. [35] provided a list of SV detection tools with four different categories, RP, RC, SR and AS. The selected tools are Breakdancer-Max, CNVNator, Gustaf, LUMPY, and SVDetect which represent a variety of strategies. Breakdancer-Max uses the RP method and can detect all variant types except duplications. It is also frequently cited in other benchmarks and appears to be a standard tool to compare against. Therefore, in this study, Breakdancer-Max was used as a baseline to compare with other infrequently cited tools. CNVNator uses the RC method which allows it to call deletions and duplications. CNVNator is infrequently cited in literature. Gustaf, as one of a limited number of SR tools, was chosen

Table 3.1: Overview of Simulated Data Sets and the SVs they contain © 2019 IEEE.

Set	**Number of Simulated Structural Variants**			
	Deletion	**Duplication**	**Insertion**	**Inversion**
1	22	22	22	22
2	22	0	0	0
3	0	22	0	0
4	0	0	22	0
5	0	0	0	22

as the other SR-only tools could not be located; it can call all variants except insertions. LUMPY combines the SR and RP methods and was selected as an alternative to DELLY [32], which is a commonly cited program; LUMPY cannot be used to detect insertions. Finally, SVDetect represents another combination of methods, namely RC and RP. Other tools, that were considered but not used due to difficulties with the testing environment, are SoftSearch (SR and RP methods) [16], CREST [39] (SR and AS methods), Genome STRiP (RP, SR, and RC methods) [15], and Magnolya (AS method) [29].

Table 3.2 provides an overview of how the different tools compare in terms of which variants they can call. If a tool was selected solely based on breadth, SVDetect is clearly the tool that has the most coverage as it can call all four types of variants investigated here. However, tools that call fewer types of variants may be tuned to a particular type and perform much better with that type; whether or not this is true will be discussed in the results in Section 3.3. Overall, the tools have a good spread over which types of variants they can call. However, only two of them call inversions which significantly restricts choice if this is a variant being searched for.

Table 3.2: Comparison of Selected Tools based on Structural Variants they call.

Tool Name	**Can Call SVs of Type**			
	Deletion	**Duplication**	**Insertion**	**Inversion**
Breakdancer Max	Yes	No	Yes	Yes
CNVNator	Yes	Yes	No	No
Gustaf	Yes	Yes	No	Yes
LUMPY	Yes	Yes	No	Yes
SVDetect	Yes	Yes	Yes	Yes

3.2.4 Method

The selected tools were tested on each data-set described in Table 3.1 with four levels of coverage, namely, 20x, 30x, 40x, and 50x. Each program run was timed and this timing includes the pre- and post-processing of the inputs and outputs. Note that the alignment and read simulation were not considered in the pre-processing time as they are the same for all tools. The tools were run in sequence to avoid resource contention and multi-threading options were set to the number of physical cores available on the system. All settings, apart from the multi-threading, were set to the default value or the recommended value from the author; if one of the pre-processing tools calculated any settings, these were used instead of defaults or recommended values.

The benchmarking procedure was as follows:

1. Generate 15 Altered Genomes with RSVSim

2. Generate Paired-end and mate-pair reads for all 15 genomes and of all 4 coverages

3. Generate alignments for all 60 data-sets using paired-end reads only

4. Run each tool on all data-sets (this step was timed)

5. Convert all results to the standard VCF format for comparison

6. Evaluate results

One point to note is that not only paired-end but also mate-pair reads were generated for each of the 60 data-sets. Mate-pair reads are very similar to paired-end reads but tend to be slightly longer and have different insert sizes. To ensure equal comparison, the mate-pair and paired-end reads were all 150 bp in length and only the insert size was different. The tools that only worked with mate-pair reads were provided with these instead of the paired-end reads. However, all alignments were produced with the paired-end reads for consistency as it has no effect on the tools.

3.2.5 Evaluation Method

The results of each tool were compared with the true-positive information generated by RSVSim during the SV simulation process. Due to the fact that it is very unlikely that a tool will call the exact location where an SV is located, a small deviation from the real location was allowed during evaluation by applying a threshold. The values for this threshold were $t(n) = 10 \times 2^n$, where $n \in \mathbb{Z} : n \in [1, 10]$. This threshold was applied to the starting index i of the detected SV to match any true-positive in the range $[i - t(n), i + t(n)]$.

Using this threshold, each result was evaluated ten times to determine how the precision of the tool changed as the leniency was increased. A tool that provides very reliable positions

for an SV would be uneffected by the threshold. Tools that provide less reliable starting positions would gain precision as the threshold grows and calls are allowed to be further off to be considered correct. The general idea behind this threshold is to reevaluate the calls that were previously considered incorrect and to ask if they are now "close enough", as defined by the threshold. This way, slight deviations that might be caused by outside forces (sequencing errors or alignment errors) can be mitigated.

With this process, a detected SV was considered correct if the type of SV matched one of the true-positives falling in the range of each threshold. True-positives were marked after being matched to avoid matching multiple calls to the same true-positive with larger thresholds.

Three additional properties were considered to evaluate the correctness of the calls, namely, end-point of the SV, length of the SV, and base-pair sequence of the SV. The end-point of the SV was not considered because it does not significantly differ from the starting point approach and using both would introduce a very large margin of error that could be larger than the smaller SVs being simulated. The length is a good measure to evaluate the correctness. However, it is difficult to determine how the length would weigh in on the final correctness score as it would also require a threshold to allow slight deviation. Using length and starting/ending position together would also increase the overall error. Finally, the base-pair sequence was not considered because simulated sequencing-errors as well as alignment errors would impact this score significantly. Furthermore, it would be difficult to evaluate this because reads can contain additional symbols other than the 4 nucleotides in DNA, which could lead to decreased accuracy. Thus, only the starting point of the variant and the type of variant were used during the evaluation stage.

Precision, is a conventional measurement to determine the success of predictions when the classes are imbalanced. This metric was used since the tools can define positive data (SVs), but not negative data (non-SVs). Precision is defined as the ratio of the number of detected

true-positives divided by the number of detections, measuring the relevancy of the results. A high precision means that a called SV is very likely to be a true SV. The precision of each tool was plotted as the threshold is increased in Fig. 3.1 - Fig. 3.5. In these figures, an ideal tool would have a horizontal line at $y = 1$, meaning ideal performance at the lowest threshold. In general, a tool that has a large slope makes less reliable calls than a tool that does not.

3.3 Results

3.3.1 Overview

The performance of all tools was evaluated using the five data-sets shown in Table 3.1 by running each tool against all three versions of each data-set. The comparison of each tool using precision was done by the starting location of the variant and the type of variant; length and end-point were not considered during these benchmarks. A threshold was introduced allowing the starting position to differ by a set difference from the correct position to allow the tools to be off by a margin of error. Using the threshold, it is possible to see how far off each tool tends to be from the correct starting point.

The full numerical data for the experiments discussed here is available in Appendix A.

3.3.2 Precision curves

The tools were run on three genomes, D. Vulgaris, E. Coli, and M. Maripaludis and the precision curves of them are shown below. In the curves, the data points show how precision is effected by increasing the acceptance threshold of a particular call. In some instances, increasing the threshold causes no change. In that case, the curve is a horizontal line. This is the ideal result, as this implies that the tool performs at a particular level regardless of the acceptance threshold; though, higher precision is still better than making reliable calls. Some of the plots have fewer than five curves, this is because not all tools call all types of

variants as shown in Table 3.2.

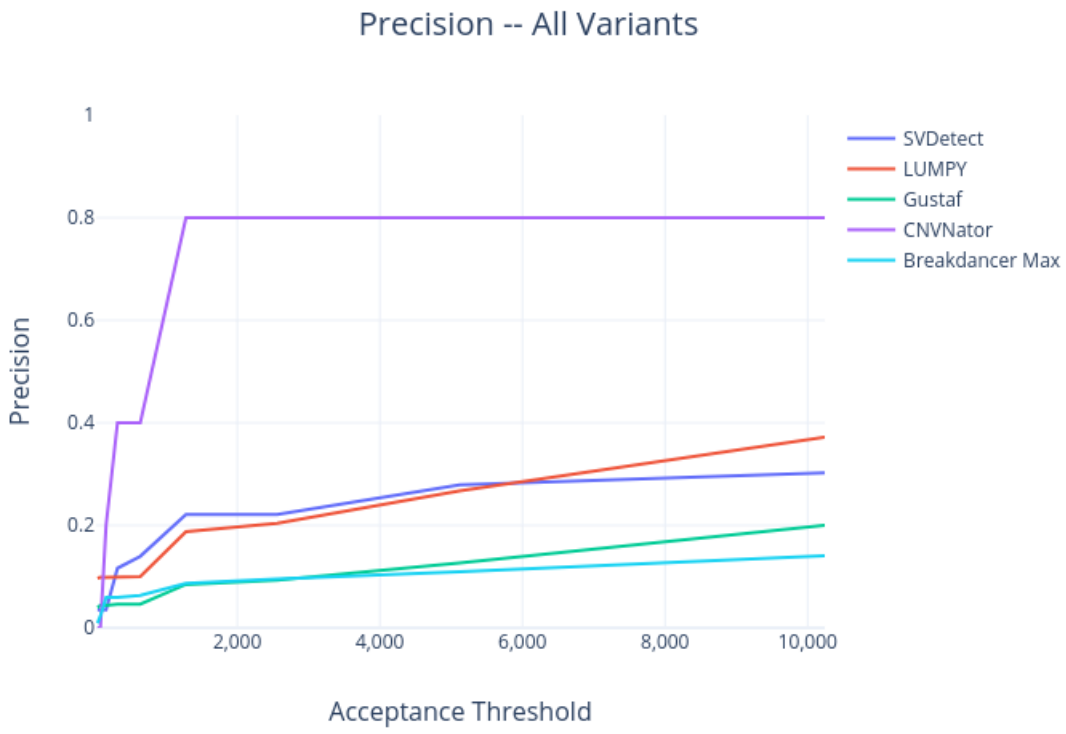

Figure 3.1: Precision Curve for all tools as threshold grows on data-set 1 (all variants).

Fig. 3.1 shows the precision curves for the M. Maripaludis genome on the first data-set including 22 of all four kinds of SVs from Table 3.1. Overall, there were 88 true-positives in the data-set and most tools called upwards of 100 variants. In this case, CNVNator achieved the highest precision of 80% regardless of coverage with a very low threshold. Most of the tools hovered between 20% and 40% precision as the threshold increased.

The precision curves for the deletions-only data-set are shown in Fig. 3.2 for the M. Maripaludis genome. The highest precision is SVDetect with about 52%, just ahead of CN-

Figure 3.2: Precision Curve for all tools as threshold grows on data-set 2 (only deletions).

VNator and LUMPY with 50%. LUMPY scored about 50% precision on this set with the largest threshold. This data-set only contained 22 true-positives and most tools called approximately that number of variants, except for CNVNator which called far fewer SVs with just 5 or 6.

Figure 3.3 displays the precision curves of the duplications-only data-set for the M. Maripaludis genome. All tools had very low precision, with CNVNator making no correct calls at all. CNVNator made 2 calls for all coverage and thresholds but both of these were incorrect. LUMPY managed to get the highest precision with about 20% for all coverages and thresholds. Gustaf made quite a few incorrect calls for this data-set, with over 6 times the number of calls than true-positives, making at least 111 calls but usually between 132 and

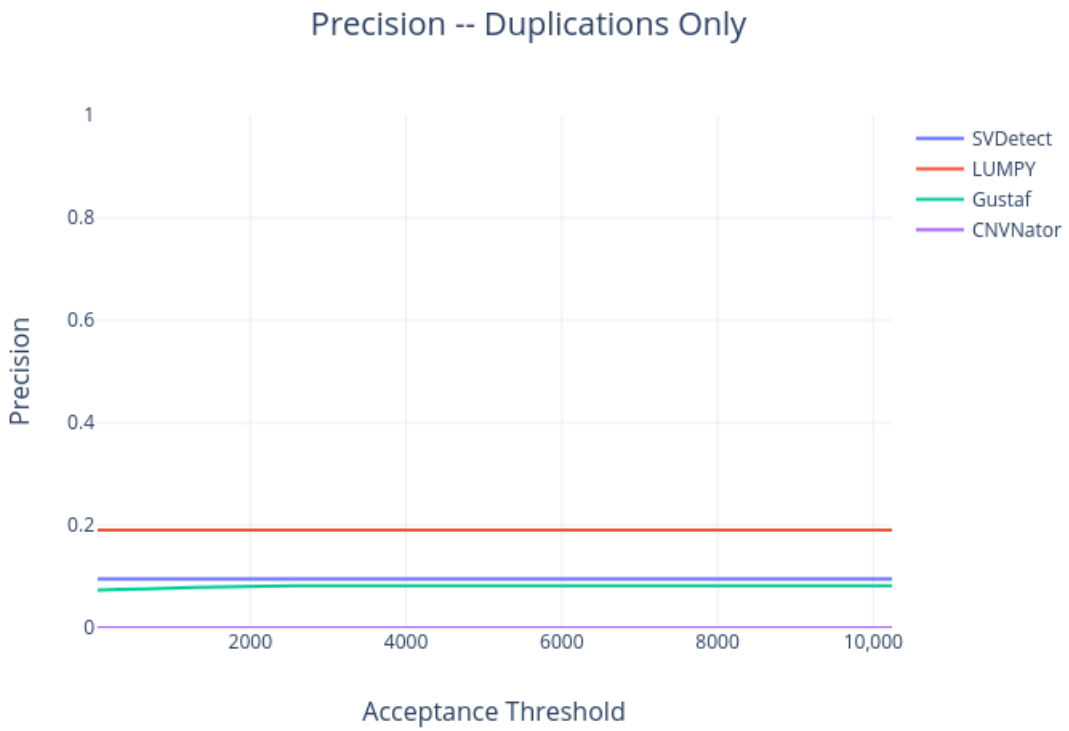

Figure 3.3: Precision Curve for all tools as threshold grows on data-set 3 (only duplications).

153 calls for the different coverages and thresholds. This had quite an impact on the precision getting less than 10% for the most part. Breakdancer Max does not detect duplications and is therefore not shown.

Figure 3.4, shows that both tools have a low detection rate for the insertions-only data-set for M. Maripaludis. SVDetect did slightly better than Breakdancer Max with about 10% precision. Breakdancer-Max achieved about 8% precision. CNVNator, Gustaf, and Lumpy do not detect insertions and are not shown here.

Figure 3.5 shows the performance on the inversions-only data-set for M. Maripaludis. All tools provided good results on this data-set; CNVNator is excluded as it does not detect

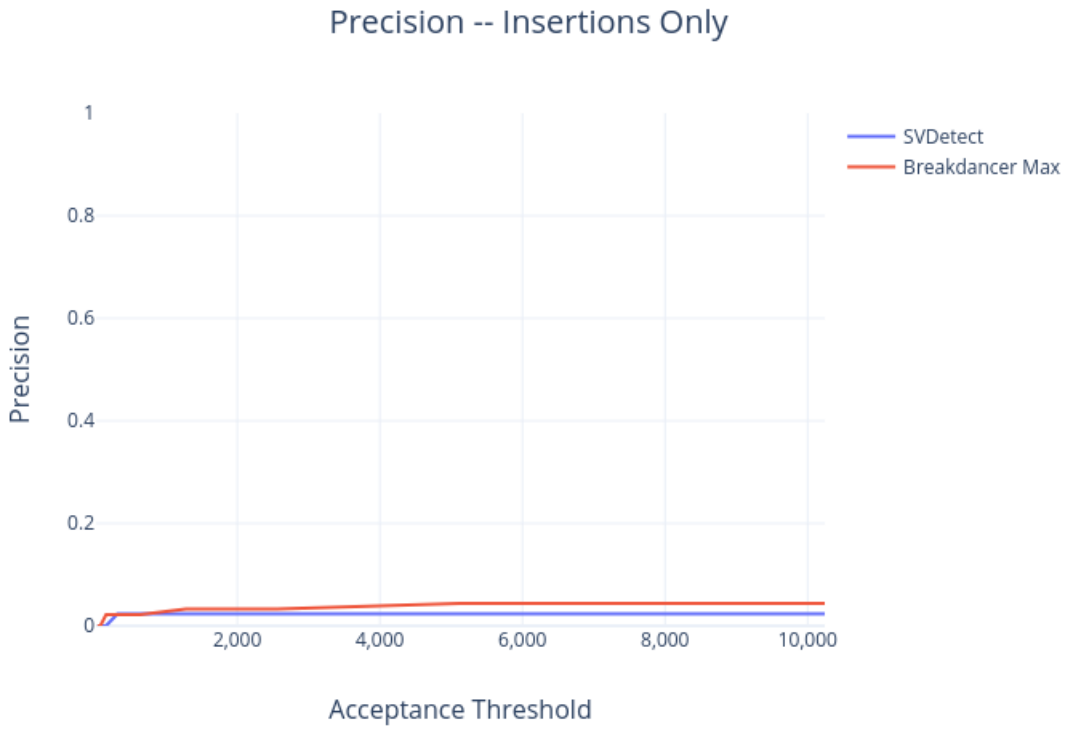

Figure 3.4: Precision Curve for all tools as threshold grows on data-set 4 (only insertions).

inversions. The highest precision in this data-set was achieved by Gustaf with a 95% precision at the highest threshold. LUMPY and SVDetect came quite close to this, with 90% and 88% precision, respectively. Breakdancer-Max did not manage to get a very high precision, with the best at 39%. In this case, LUMPY and SVDetect significantly outperformed all other tools at the lower threshold values.

3.3.3 Time Efficiency

The execution time of each tool was recorded on all data-sets to measure its time efficiency. Any tool-specific pre- or post-processing was also included with the execution time of the program to gain an understanding of the cost associated with using that program. All data-

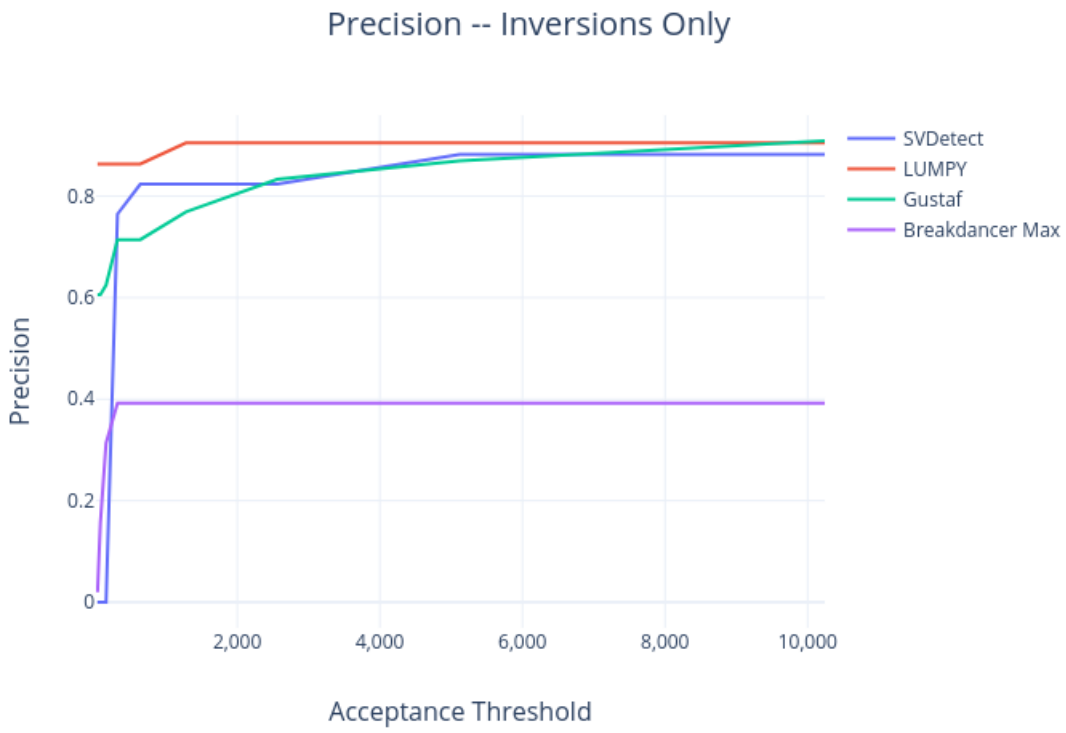

Figure 3.5: Precision Curve for all tools as threshold grows on data-set 5 (only inversions).

points were averaged and are shown in Figure 3.6; the error bars in this plot show the standard deviation. It is easy to see that Gustaf takes significantly longer than the other tools, taking almost 5,000 times longer than Breakdancer Max and around 100 times longer than the next slowest tool, SVDetect. Breakdancer Max is easily the fastest tool, taking around 1 second for all runs; interestingly, coverage appears to have no effect on Breakdancer's execution time. The other three tools, CNVNator, LUMPY, and SVDetect all perform roughly equivalent in terms of execution time finishing in between 5 and 10 seconds for the most part.

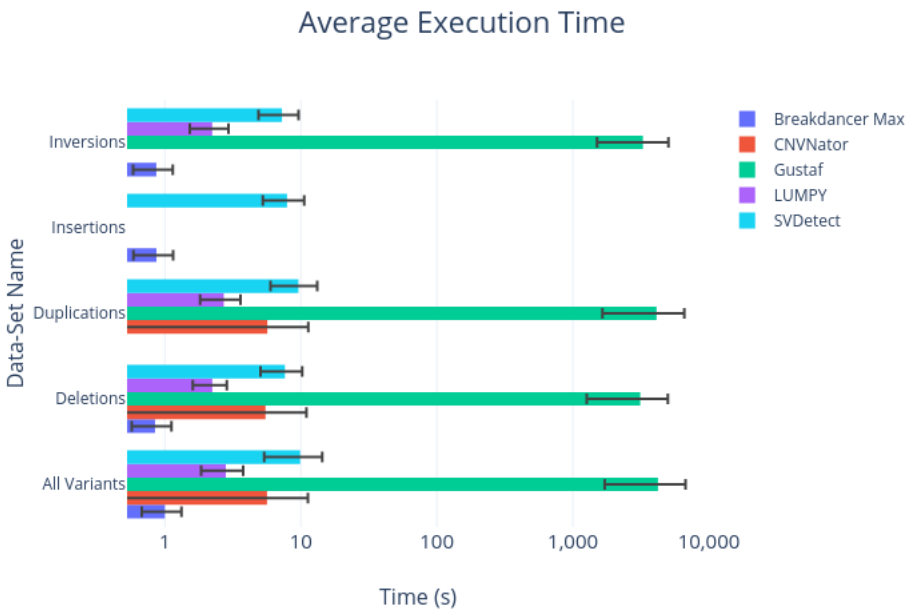

Figure 3.6: Execution times of the different tools on the 5 data-sets.

3.4 Analysis

3.4.1 Precision

CNVNator, which uses the RC method, achieved the highest precision in data-set 1 which contained all types of variants. This indicates that CNVNator is quite reliable when it calls a variant, but it cannot generally be trusted to call all variants; usually, it makes far fewer calls than variants in the data-set. Also, it does not detect insertions or inversions. This makes it less useful in many situations where various types of SVs are expected.

LUMPY, which uses the RP and SR methods, managed to get a very good precision on the smallest threshold value with 86% precision on detecting inversions. It generally performs in the middle of the pack for other types of SVs. When the data includes all types of SVs, LUMPY managed to achieve just under 60% precision with the highest threshold.

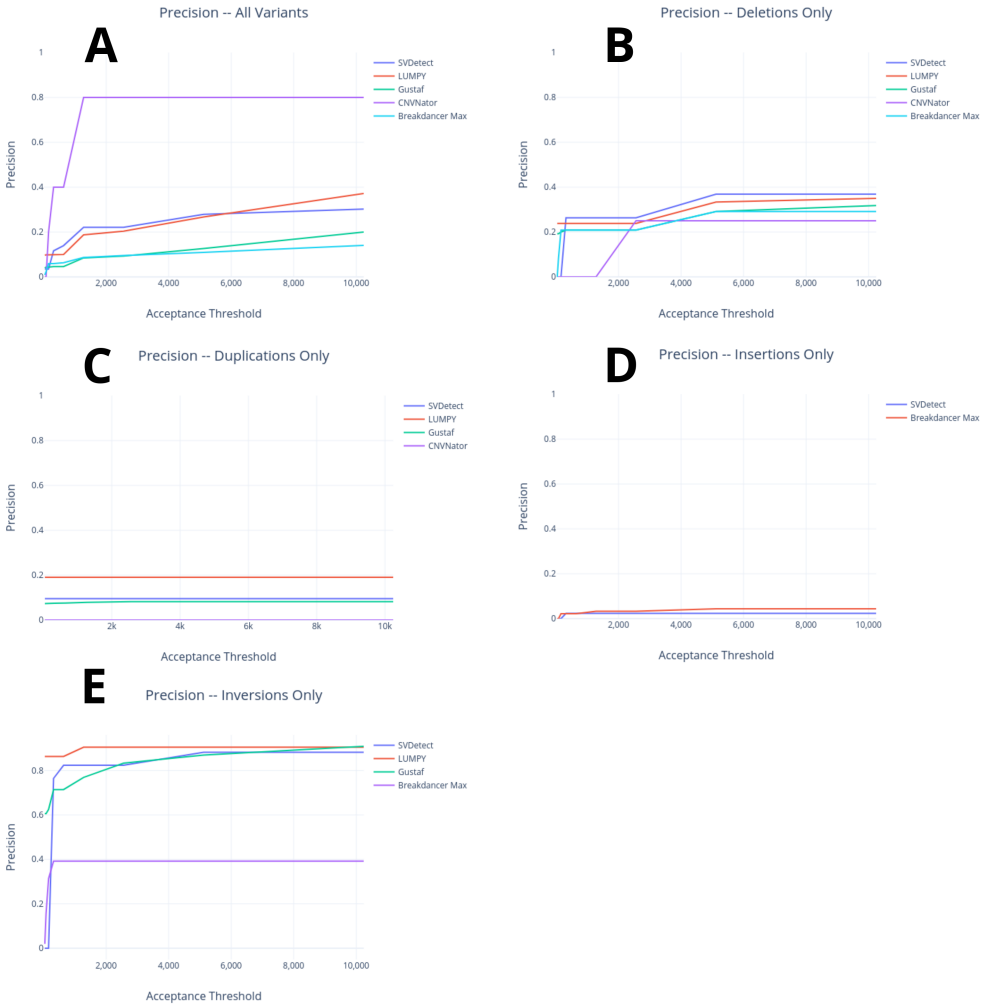

Figure 3.7: Precision curves for data-set 1 (A), data-set 2 (B), data-set 3 (C), data-set 4 (D), and data-set 5 (E). This is a combined view of figures 3.1 through 3.5.

In most cases, LUMPY has a very steep slope on the precision curve, showing that the calls are less reliable in terms of starting position.

Gustaf, which uses the SR method, also does not detect insertions. Much like the other tools, it performs very well on detecting inversions, getting up to 95% precision on data-set 5. Gustaf, however, did poorly in detecting other types of SVs. It usually was among the lower one or two tools in terms of precision; never achieving more than 48% precision on detecting deletions and staying below 10% on duplications.

SVDetect that combines RC and RP methods also performed well on detecting inversions. For the most part, SVDetect was generally in the top two or three when it comes to precision. On data-set 2, it managed to get 52% precision at the second largest threshold. With the inversions, SVDetect was able to achieve a 76% precision with a reasonable threshold of 160 bps and this increased to 88% precision with a threshold of 2560 bps. Overall, the calls of SVDetect tended to be quite reliable at low thresholds and it did not gain too much performance at the very large thresholds.

Interestingly, the baseline tool, Breakdancer-Max, which uses the RP method, performed no better than the rest of the tools. The highest precision was for inversions-only data with 41%. In most cases, it was either the worst, or second worst tool together with CNVNator. However, Breakdancer had a tendency to reach the maximum performance at a very low threshold, indicating that the calls are very reliable in terms of SV location.

3.4.2 Read Coverage

Reads of the data-sets were simulated in 20x, 30x, 40x, and 50x coverage to see how this would effect the precision of the tools. Unexpectedly, some tools performed identically between 20x and 50x coverage, while others actually performed better on 20x coverage than on 50x cover-

age, and only two tools actually improved in precision when the read coverage was increased.

Unsurprisingly, the tools that improved in performance with higher read coverage were the RC tools, namely, SVDetect and CNVNator. CNVNator was less affected by the coverage than SVDetect. For example, on data-set 1, CNVNator performed well on 20x and 50x coverage, but performed worse on 30x and 40x coverage. The maximum performance was always achieved on the high and low coverage, which suggests that it may not have a significant impact on CNVNator. SVDetect, on the other hand, gained a small performance increase across the board from the additional read coverage, varying as much as 4% in precision from 20x to 50x coverage on data-set 1 and 3% on data-set 3. While this is a small increase, it does show that RC tools' precision is increased with additional read coverage.

SVDetect, Breakdancer Max, and CNVNator were the only tools where read-coverage made any difference between the precision of the different tools; LUMPY and Gustaf had identical precision values for all four read-coverages. Interestingly, LUMPY and Gustaf are both based on the SR method, which could mean that this strategy is fairly independent of coverage. In the case of LUMPY, varying coverage also had no effect on the precision of the calls, it was solely impacted by the threshold value and, even then, did not change dramatically in most data-sets. However, the increased threshold did cause the precision to go from 9% to 59% between the smallest and largest threshold on the data-set 1.

3.4.3 Acceptance Threshold

The acceptance threshold was used to determine whether a variant call was considered to be correct or not. Ten different thresholds, varying from small (20 bps) to very large (10240 bps), were used to determine how accurately the tools can identify the location of the structural variants. Due to the size of the variants used (100 - 1000 bps for the most part) and the size of the genome, accepting a call that is within $\pm$320 bps of the correct starting point seems like a reasonable range. A tool that performs well should have a precision of at least

50%, in other words, it should submit more correct calls than incorrect ones.

At this range, the only data-set where any tool was able to get a precision of over 50% was data-set 5 which contained only inversions. With this data-set, LUMPY and SVDetect were able to achieve over 80% precision. Breakdancer-Max performed the poorest, only achieving a maximum of 39% precision. Gustaf was situated between the two, resulting in as much as 71% precision. LUMPY and SVDetect both use different combinations of the RP strategy, combining it with SR and RC, respectively. On top of this, LUMPY and SVDetect also outperformed Breakdancer-Max and Gustaf on all other data-sets in this range and CNVNator only outperformed them on data-set 1. It appears that the combination of the RP method makes the tools especially accurate in this range, while tools such as Gustaf, which only uses SR, and CNVNator, which only uses RC, do not perform as well in general. This observation suggests that combination of compatible methods would improve the performance.

With these stipulations applied to the results it is clear that all tools do not perform well on all but the inversions-only data-set. LUMPY and SVDetect were the tools that came the closest to the 50% precision line at 320 bp threshold or below. This suggests that, while none of the tools fit the definition of good in this comparison, LUMPY and SVDetect are the most promising candidates and some tuning of the parameters could push their performance above the 50% line. In these tests, almost all tuning parameters were left at the default or suggested values. These parameters may be tuned for the human genome, which could lead to the poorer performance on microbial genomes. As LUMPY and SVDetect managed to achieve precision in the high 30s and low 40s in some of the data-sets with a reasonable threshold, it is likely that these tools can be tuned to achieve a good performance in most cases. The same cannot be said for Breakdancer Max, Gustaf, and CNVNator, which rarely even got close to the 50% mark. This could mean that these three tools are optimized for the human genome and not ideal for use with microgenomes.

3.4.4 Usability

Another important question this benchmark is supposed to answer is: how easy is this tool to use? While this may not impact the performance, it impacts the user experience and the tools vary significantly in this area.

The easiest one to use, by far, is LUMPY, which comes built into the alignment tool SpeedSeq [5]. Although CNVNator also comes with SpeedSeq, it is not part of the program itself, like LUMPY. LUMPY is also quite easy to use as it comes with an express version which presets many settings to common settings and the remaining settings are simple to understand and work with. It also generates output in VCF format making it easy for post-processing.

CNVNator is quite simple to use. Although it involves many steps, the documentation is adequately detailed to make following them simple. Unfortunately, it does not come with many presets and may require a lot of parameter-tuning. While the output is in a custom format, the program comes with a script to convert it to VCF format for easy post-processing.

SVDetect is rather challenging to use as it uses a configuration file for most settings, this makes automating the process more difficult as a configuration file must be created for each run instead of passing command-line parameters. Furthermore, SVDetect consists of multiple scripts that must be run in sequence and the output from one, which is sent to standard output, is required by the next; this also makes automation challenging. The documentation is not straightforward, and the output format is non-vcf format.

Similarly, Breakdancer-Max had some additional steps involved with pre-processing steps, but the files can simply be passed on to the next step, allowing for easy automation. However, documentation lacks detailed instructions, making it challenging to understand the

exact process. Breakdancer-Max also provides a custom output format which means an extra step is required to either parse this format or convert to VCF.

The most challenging tool to use was Gustaf. Gustaf requires multiple pre-processing steps and these programs do not support the full FASTA alphabet of the reference genome or reads. It required an additional pre-processing step to normalize the reference genome and reads to use only DNA5 characters (A, C, G, N, T). Furthermore, Gustaf has its own alignment program that is not compatible with other alignment tools. This means that alignments used for other tools can not be reused. The output of Gustaf is in the standard VCF format, though.

3.4.5 Execution Time

The times displayed in Table 3.6 were collected from each run of the program during the benchmarking process and may not, for a total of 4 samples for each tool. The times show a strong trend in that Breakdancer-Max is always the fastest tool and Gustaf is always the slowest tool. The results can be used to make generalizations about whether or not the time invested in the tool is worthy based on the other factors that have been discussed.

In terms of execution time and precision, LUMPY brings the best performance per second invested into running the program. It performs well in most cases and is only slightly outperformed by SVDetect in some cases. CNVNator provides decent results in a limited set of circumstances and is slower than LUMPY. While Breakdancer-Max is the fastest, it does not outperform the other tools in these experiments.

SVDetect, while it takes slightly longer than LUMPY, also has good performance in many cases. In addition, LUMPY and SVDetect also scale well as read coverage increases. In these experiments, the programs' execution time scaled linearly.

Chapter 4

METHOD

4.1 Overview

The benchmarks presented in Chapter 3 show that clustering approaches, such as those used by SVDetect [44] and LUMPY [24], provide better results than tools that do not use a clustering approach. DELLY [32] uses a clique-based clustering approach to detect SVs with high accuracy. This raises the question if the performance of an SV detection tool can be improved by using network motifs instead of cliques. While cliques allow the extraction of structural information from a graph, the structure of a read-graph is effected by many factors such as coverage, sequencing errors, and alignment errors. Thus, it may be beneficial to use a network-motif approach instead that is not bound to a particular graph-structure, but instead uses structures that are found to be present in the graph. This approach could, in theory, improve the performance of a clique-based approach. This chapter will outline how the clique based algorithm was turned into a network motif based algorithm, Nemo-Cluster; in addition to this, the software development life-cycle, implementation details, and challenges associated with this process will be discussed.

4.2 Algorithm

4.2.1 Read Graph

Constructing the read-graph can be done in a number of ways as described in the discussion of other clustering tools in Section 2.4. The main reason for such a graph is to leverage the similarity between different reads to determine the correct position to align them to. Discordantly mapped reads will be considered the vertices in the read-graph and edges represent

an overlap between the reads and it will be weighted proportional to the overlap.

Clustering this graph using the suggested approach below should result in reads that belong to the same genomic region to end up in the same cluster as they will have a large number of overlaps and therefore form network motifs. This approach is similar to that of DELLY [32] which uses maximal cliques to find clusters in the read-graph, though DELLY constructs the graph differently. Each cluster of reads can then be assigned a correct location within the genome based on where the majority of the reads contained within were aligned to. Once each cluster is assigned to a genomic region, the reads can then be analyzed by another process to determine exactly what kind of SV occurred at that particular location. It may also be possible to determine the kind of SV from the cluster itself, but the main goal is to find the correct place in the genome for the reads.

4.2.2 Tectonic

Tectonic is an algorithm designed to find clusters in social networks and has been shown to perform very well on both benchmarking and real-world data [37]. The algorithm is based on the use of cliques to determine the significance of existing edges in the input graph and reweigh them based on that. In specific, Tectonic is based on the use of K_3 cliques, i.e. a triangle, however, they claim that it can be extended to cliques of any size. This approach is referred to as triangle conductance which extends the idea of conductance from just edges to triangle cliques. The general idea behind this approach is that if a subgraph has high triangle conductance, this subgraph is a community while other parts of the graph will have lower triangle conductance. This will result in edges that connect communities being removed and edges within communities being kept.

The central part of this algorithm is the reweighting step which considers the triangle conductance of each edge as $\frac{t(u,v)}{deg(u)+deg(v)}$ where u and v are vertices, (u, v) is an edge con-

necting them, and $t(u, v)$ is the triangle count of the edge (u, v). The main function here, $t(u, v)$, calculates how many triangles contain the edge (u, v) and returns that number. The denominator of the fraction is used to smoothen the function and normalize the result into the range [0,1]. This conductance value is then used as the new weight of the edge. In theory, densely connected components of a graph, which represent communities, will have edges with a high weight after this process, while edges between these dense components will have a low weight. Once this process concludes, it is then possible to remove all edges below a certain threshold value λ from the graph which results in a disconnected graph. The connected components in this new graph are the clusters.

4.2.3 NemoCluster

NemoCluster uses the same inherent approach as Tectonic since it was modeled after it. Instead of using cliques to determine how significant an edge is, it uses network motifs. Network motifs reveal more complex structures within a graph and should therefore produce better clusters. The approach used by Tectonic works well in the domain of social networking where communities are going to be complete or almost complete subgraphs, something a clique-based approach will excel at finding. However, in a more general application, clusters may not contain many cliques but may contain other types of structures that would qualify as network motifs for the graph. This means, using network motifs allows this algorithm to be extended to new areas such as Bioinformatics that do not generally work with graphs where communities are of interest.

Much like with Tectonic, NemoCluster's central component is also the reweighing process of the graph. However, instead of using the triangle conductance $t(u, v)$, it uses the motif conductance $m(u, v)$ while keeping the reweighing function identical $\frac{m(u,v)}{deg(u)+deg(v)}$. This still has the same exact benefits as Tectonic has, in fact, if only triangle motifs are used, NemoCluster will behave identical to Tectonic. In addition to keeping the benefits, though,

this approach allows the user to fine tune the algorithm to work with any graph using the network motifs present in that graph. Most importantly, the clique-based approach used by Tectonic is limited to a single type of subgraph at a time; NemoCluster removes this restriction allowing the clustering process to consider motifs of different sizes and shapes simultaneously to best utilize the structural information encoded in the graph.

The main downside of NemoCluster is the fact that finding network motifs is significantly more costly than finding cliques. Network motif discovery requires a thorough analysis of the graph as well as a Monte-Carlo simulation to determine whether or not the frequencies of the candidate structures are statistically significant. Overall, this process is NP-Complete and can therefore take a very long time on larger graphs. In contrast, clique-finding algorithms can disregard most subgraphs in a graph as they only care about complete subgraphs. It is also not required to do any statistical analysis to find cliques, therefore the Monte-Carlo simulation, which represents the most time in network motif finding algorithms, is eliminated with cliques. While the use of cliques is faster, it also disregards most of the structural information available in the graph, thus, in applications such as Structural Variant discovery, it might be of benefit to invest the additional time as cliques have little to no meaning in read graphs. A clique in a read-graph would simply represent reads that come from a similar part of the genome and the alignment can already find these, thus there is no need for a clustering algorithm to worry about them. However, reads that connect with certain network motifs could suggest the presence of something unusual occurring at that location in the genome involving those reads.

4.3 Architecture

4.3.1 Overview

The NemoCluster library is written in C++ with a mix of template metaprogramming (TMP) and object-oriented programming (OOP). The reason why C++ was chosen for this project was for the performance; alternatives that were considered are Python and Rust. Python, being an interpreted language, does not provide the performance for this kind of algorithm and so it was not chosen for the core application, however, it was used for pre and post processing scripts. Rust is a low-level language like C++ which can achieve high-performance, however, due to unfamiliarity with Rust, C++ was chosen instead. Furthermore, the standard template library (STL) of C++ provides similar memory guarantees that Rust makes using the Resource Acquisition Is Initialization (RAII) technique used by all STL containers. Therefore, a C++ program can achieve the same memory guarantees that the Rust compiler makes by utilizing these containers and it only comes at the cost of always catching exceptions. The use of RAII will be discussed in Section 4.4.3

4.3.2 Preprocessing

The NemoCluster algorithm requires a small amount of preprocessing of the data as shown in Fig. 4.1, namely, NemoLibrary [2, 22] must be run on the input graph, and the input files must be normalized. Firstly, the NemoCollection feature of NemoLibrary is used to generate a file containing the network motifs to use during the clustering. These motifs can then be filtered based on the user's preferences before sending them to NemoCluster, however, it is not required. The second preprocessing step is to take the NemoCollection file and the Graph and relabel it to have consecutive integer vertices, this is done to keep the internals of NemoCluster efficient and avoid working with strings which are considerably slower than integers.

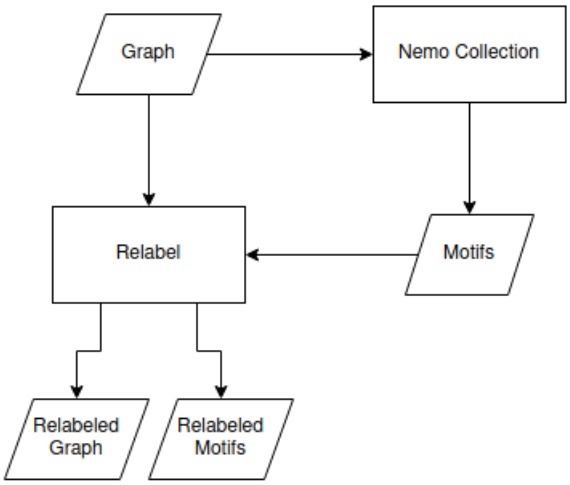

Figure 4.1: An overview of the different architectural components of the NemoCluster Library.

4.3.3 Nemo Cluster

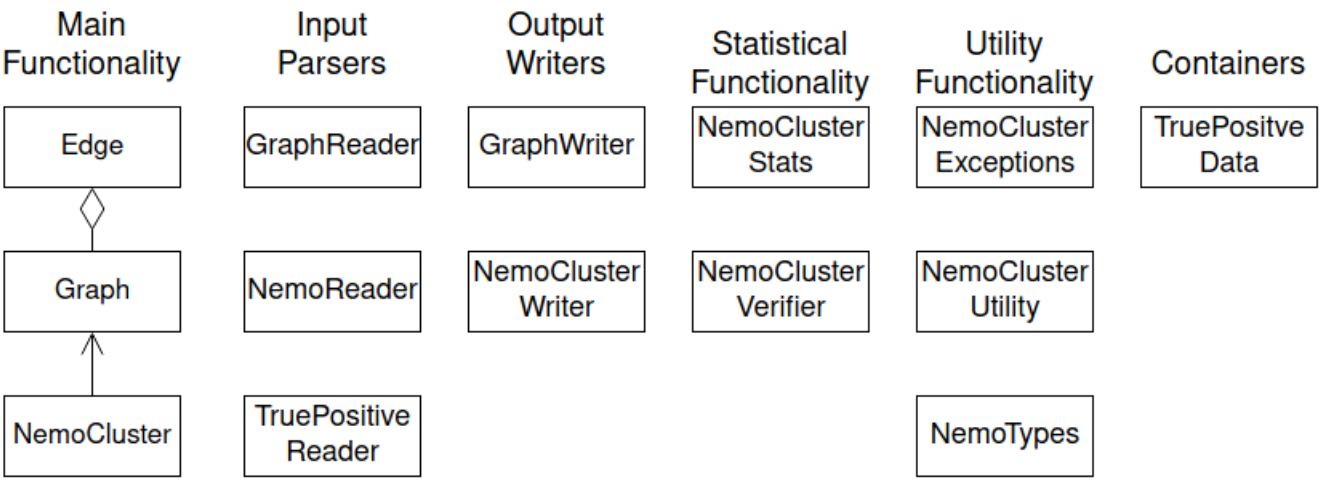

Figure 4.2: An overview of the different architectural components of the NemoCluster Library.

The main functionality of the program lives in the NemoCluster class which provides the clustering functions. As shown in Fig. 4.2, this class inherits from a simple Graph class containing edges. The reasoning behind this design is that the clustering of a Graph is in itself a Graph and thus the NemoCluster class is a Graph. Another way this could have been designed would be to take the Graph as input to NemoCluster, however, this would have meant that NemoCluster would need to either alter the Graph object or copy it; with this design, changes can be held in a separate data-structure internally hiding the fact that the Graph is never actually altered, improving performance significantly.

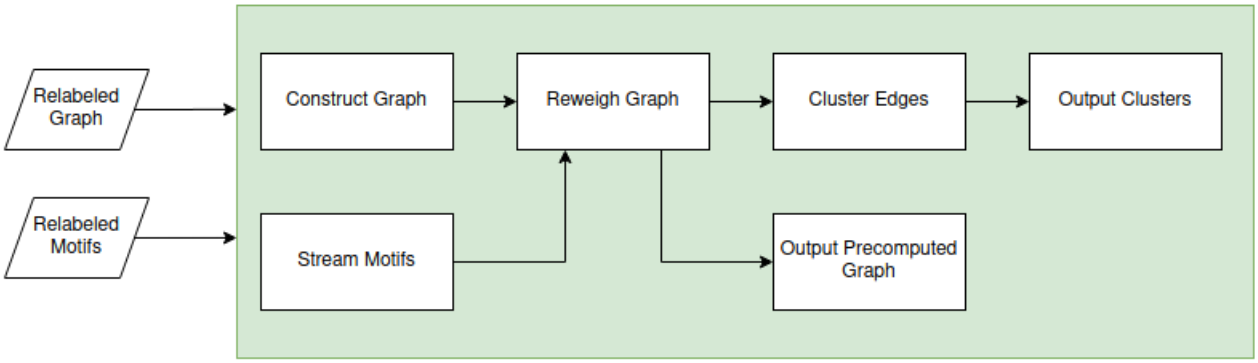

Figure 4.3: Internal view of the NemoCluster class, the green-shaded area denotes NemoCluster class members.

Fig. 4.3 shows the internals of the NemoCluster class. The NemoCluster object is constructed around a Graph object which is produced by the GraphReader class. Once the Graph is constructed, the reweigh graph function can be invoked which prepares the graph for clustering by calculating the motif count for all edges and then applying the new weights to them.

During this process, the NemoReader class streams the network motifs from an input file for the NemoCluster class to use as shown in Fig. 4.4. The NemoReader class starts

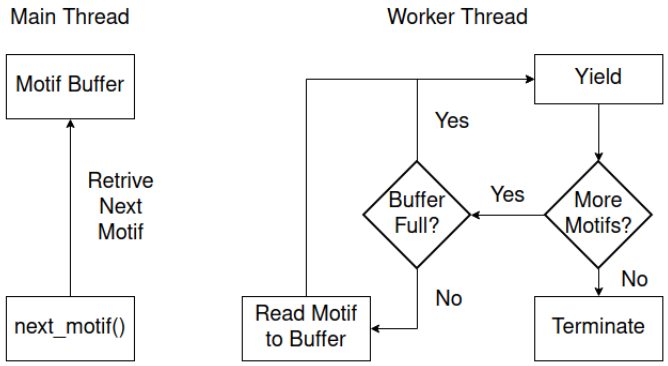

Figure 4.4: Internal view of the NemoReader class.

a worker thread which reads lines from the motif file into a buffer that is always ready to answer requests for the next motif by NemoCluster in the main thread. Whenever the buffer is filled up, the worker thread yields to other threads until the buffer has more room to read. Once the worker thread is done reading the file, it terminates and signals that reading is done and no more motifs will be added to the buffer, once the buffer is emptied after this, the user will be signaled that the eof has been reached. As the reading of motifs from input files is the most expensive part of the program, using some multi-threading here improves performance slightly without increasing the memory footprint unnecessarily by reading the whole file into memory.

After the Graph has been reweighed, it is possible to store the weighted graph in a file to retrieve in the future, this is discussed in more detail in section 4.4.1, reading this weighted graph is significantly faster than reconstructing it from the motifs. With the weighted Graph, the edges can then be clustered with a user-defined clustering constant λ, all edges having a weight less than this will be *flagged* as removed internally.

As the edges are not actually removed, but only flagged, it is possible to reuse a NemoCluster object with different clustering thresholds without needing to reconstruct the data.

This saves a significant amount of time when trying different thresholds on the same Graph.

In addition to this, the clusters are only realized once the user explicitly asks for them, before that, the clusters are only present in the edges that were flagged as not removed, but no costly copies have been made of them. This ensures that copying is reduced to a minimum and copies are only made once the user explicitly requests them. Once this flagging process is done, the clusters can be written to an output stream with the NemoClusterWriter class which formats them according to one of two output formats.

4.3.4 Postprocessing

The final step of the NemoCluster algorithm is to undo the normalization of the input data. A python script is provided that is run on the output files to replace the sequential integers assigned to each vertex of the graph with the original string used for the vertex.

4.4 Implementation Details

4.4.1 Weighted Graph Format

The weighted graph produced by NemoCluster can be stored in a custom binary format to be more memory efficient and to create a recovery point for the rest of the program.

The issue with the file generated by NemoLibrary is that it is a plain-text view of the data, this is inherently inefficient as it uses several bytes to store what could be stored in a single byte. The binary format used by the GraphReader and GraphWriter classes, provided by the NemoCluster library. The GraphWriter automatically detects the minimum number of bytes that must be used to store all vertices in the graph and then uses this to optimize storage size.

The optimization flag, storing the number of bytes per vertex, is prepended to the data to ensure the GraphReader class can correctly interpret the file later on. The optimization flag is 1 byte in size, thus creating no meaningful overhead in terms of file size. The weighted edge-set can then be stored in at least 10 bytes (1 byte per vertex and 8 bytes for the weight) and at most 24 bytes per edge (8 bytes per vertex and 8 bytes for the weight), significantly reducing the maximum size of each edge. In plain text, the 24 byte case would require up to 20 bytes per vertex, and at least 17 bytes for the weight, thus reducing size from 57+ bytes to just 24 bytes.

The 24 bytes-per-edge is for extreme cases only, as this is for graphs with more than 2^{32} vertices. In the general case, graphs will require between 12 and 16 bytes per edge; this is still significantly smaller than if it was stored in plain text as there each vertex would need up to 10 bytes, plus 17+ bytes for the weight, so a total of 37+ bytes to achieve the same result.

In addition to this, the GraphReader and GraphWriter classes compress the binary data using the lz4 compression algorithm [6] to further reduce file size. This results in excellent compression results, for example, the Amazon network [26] is 12 MiB in size and the size 4 NemoCollection file is 1.8 GiB. With this compression scheme, these two files can be combined into a weighted graph using only 8.5 MiB, effectively reducing required disk space by over 200 times for this graph. The gain of this compression strategy increase with file size as discussed by [6].

4.4.2 Parallelism

As discussed in Section 4.3.3, the NemoReader class uses parallelism and buffering to overlap the computation and data reading steps which can be handled by separate CPU cores making the program more efficient, this efficiency increases as the file being read increases.

Unfortunately, most tasks completed by NemoCluster are inherently sequential making it useless to implement parallelization there; furthermore, for large graphs, 99% of the execution time comes from reading the massive NemoCollection file, while everything else goes very quickly in comparison. Thus, there is little to gain and the overhead of adding parallelization to the rest of the program may actually cause a slow down.

4.4.3 Resource Acquisition is Initialization

NemoCluster is written in C++ which, together with C, is well known for memory leaks and pointer-related problems. Modern C++ uses a paradigm called "Resource Acquisition is Initialization" (RAII) to mitigate the problems associated with dynamic (heap) memory. RAII states that acquiring a resource automatically initializes it and freeing the resource automatically deallocates any internal memory used by it. Using RAII compliant classes, such as those provided by the Standard Template Library (STL) guarantees that memory will be freed as long as all exceptions are caught; in the case that an exception is not caught, the operating system will automatically free all memory of the process, though files, sockets, and similar resources will not be closed properly in this case.

All classes in NemoCluster use RAII compliant containers to store all data, with the exception of the GraphReader and GraphWriter classes. Furthermore, the storage containers implemented in the NemoCluster library are RAII compliant, too, making them easy to use for library users. The GraphReader and GraphWriter classes use "raw" memory arrays as the LZ4 library requires them. However, they are used only in a single place and it can easily be verified that all code paths free the memory. The gains of using the LZ4 library, even at the cost of violating RAII, is well worth it as discussed in Section 4.4.1.

4.5 Software Development Lifecycle

4.5.1 Source Control

The entire development of NemoCluster was done through a git repository on GitLab.com. Git is one of the most common source control softwares available today and it is used for many projects, such as the Linux kernel (OS used for this project), GNU G++ (compiler used for this project), and many more.

In addition to using git for source control, a changelog was kept and updated with almost every commit to the repository to ensure that the evolution of NemoCluster is well documented. The changelog details what bugs have been fixed, what files/features were added, removed, or deprecated, and provides an overview of what each release contains.

Lastly, the standard semantic versioning scheme was used whenever a release of NemoCluster was created. This ensures that anyone who wants to use the library is able to understand what a new version entails simply by seeing what part of the version was incremented. In the current release, NemoCluster is at version v1.4.0; this version of the library is compatible with the previous three versions (1.0.z - 1.3.z) and will be compatible with any future release 1.y.z, while it will be incompatible with the next major release 2.0.0. This system is used by many companies and open-source projects to make understanding what changes a new release makes and how it will impact any existing system integrating that new release.

4.5.2 CI/CD

NemoCluster was developed with the GitLab CI feature which provides a build and testing pipeline that can be run on-demand or on each commit to a particular branch. In the NemoCluster project, it is setup to be run on-demand whenever a commit message includes the

phrase "run cicd" for simplicity. The pipeline builds the program in a debian linux environment with GCC 9.3 and then runs automated tests if the build succeeds. GitLab stores the pipeline logs and history for up to a year, making it easy to review bugs that were found previously and put into a bug-queue for review and resolution.

4.5.3 Build System

As NemoCluster is not a small software program that can be compiled by hand, a build system is used to automate this process. The most common C++ build system is CMake, however, in this project Meson was used together with Ninja.

CMake is an older system that is designed to extend GNU make to all platforms and does a good job at this. However, CMake, just like GNU make, is not very intuitive and difficult to learn.

In 2013, Meson build was released and it instantly became one of the most popular C++ build systems because of it's simplicity and package-manager. Meson uses a python-like DSL that uses intuitive names such as "declare_dependency" or "get_compiler" and allows the build file to look more like a python script, which many people know to interpret. Meson itself does not actually build anything, it only provides a user-friendly front-end to a range of build-systems. Ninja build is known for its speed and is used by companies like Google to build large projects; however, its syntax is even more difficult than CMake. Meson allows the use of Ninja to efficiently build large projects without the need to learn the complicated syntax. Another big feature of Meson is the "wrap" system, this is a package manager for C++ projects. Wrap files are created that give the location of a Meson compatible project, which Meson then automatically downloads and builds for the user to easily integrate 3rd party libraries into a project. Meson also maintains a database of many commonly used libraries that may not support Meson out-of-the-box and supplies a Meson.build file for them.

4.5.4 Documentation

NemoCluster provides two types of documentation, a references manual and in-code documentation. The reference manual supplies detailed information extracted from the in-code documentation as well as visual representation of inheritance relationships and the likes. This file is generated using the Doxygen software which is one of the most commonly used programs for documenting C++ code. It produces both a pdf manual, as well as a basic website that can be used to host the documentation online. The in-code documentation uses the javadocs format to specify function and class documentation in the headers. In addition to the fact that Doxygen can use this to build a reference manual, many IDEs can extract this information and provide it to a programmer as they code.

In addition to the code-documentation, the project also comes with a very detailed README file. The README lists exactly what requirements there are for the software, it provides current links of where to get the needed softwares, detailed discussion of building and using the software, explanation of the example code, and detailed discussion of the data format expected and produced by NemoCluster and the associated scripts.

4.5.5 Development Process

For this project, the waterfall and agile development practices were used. Most of the project was well defined from the start, therefore it was acceptable to use the waterfall model and design a well thought-out library that could then be implemented with minimal changes. Once the library was complete, this allowed for plenty of testing and benchmarking, though it created the risk of running into problems that are cemented into the design. Overall, this worked very well for the first part of the project.

During the second part, it was not possible to come up with detailed designs and review these as the Waterfall model intends, instead a more Agile approach was used to update functionality and maintain the software. As there was only a single developer working on the project, simple TODO notes were used for the most part, but GitLab Issues were created for backlog items that may not be implemented by the end of the project so that future maintainers can look into them.

4.6 Challenges

One of the major challenges faced during this research was working with the massive nemo collection files produced by [22] for the large social networks. This produces three issues, namely, it takes a very long time to generate them, they take a massive amount of storage, and they can't be read into memory in their entirety.

The first issue is just the cost of working with network motifs. This is why other algorithms, such as MAPPR [43], only use approximations and work with a single motif, as this can be done efficiently. Working with network motif made each test run extremely valuable as it could take days to see results, mistakes and bugs resulted in huge setbacks.

File size actually caused two problems, firstly, they take a lot of time to read and they cannot easily be transferred from a research server to a local computer. This meant that debugging the software became much harder, reading the input file is not a trivial task that can be repeated many times over to reproduce errors and analyze them in a debugger; furthermore, a large file cannot be downloaded from a server to a local machine that actually has a proper debugger. Hence, finding errors in the software was mostly a conceptual problem, working out with very small input files what could be going wrong with the large one. This makes issues like stack-overflows extremely difficult to notice as they don't occur with test data in the debugger where it is actually possible to analyze stack contents and find the

reason for them.

Finally, reading a 10 MiB file into memory is no issue on modern computers, it has no effect on the system in the general case. However, with a 12 GiB file this is very different. Initially, NemoCluster read in the entire file and kept it in memory during the computation of the graph; this, of course, was not a scalable design. This is how the NemoReader class came to be designed the way it is. It was necessary to read large files, ideally fast, but not keep more than a reasonable amount of data in memory. The solution, discussed in Section 4.4, was to use a buffering strategy and overlap computation and data-transfer while the buffered data is processed. This increased the performance of NemoCluster by a factor of 10 on small files, but even higher on the larger files. The main reason for this was that the system was no longer memory-starved while doing the reweighing of the graph. This was a good example of when an algorithm must be redesigned from scratch because the original idea is unfixable, keeping gigabytes of data in memory that are used once and never again is inherently flawed.

Chapter 5
EXPERIMENTS

5.1 Overview

NemoCluster's performance was evaluated by comparing it to the original Tectonic algorithm. This chapter outlines the data that was used, how the experiments were setup, and how the results were evaluated. The results will be shown in Chapter 6 along with commentary and analysis.

5.2 Data

The data used in these experiments came from two main sources, namely, social networks retrieved from the SNAP library [26], and a protein-protein-interaction (PPI) network acquired from the MTGO tool [38]. In addition to these, 30 synthetic networks were created using the NetworkX library [14].

From the SNAP library, three graphs were used during the performance evaluation, namely, the email-eu-core graph containing just over 1005 vertices and over 25,500 edges; the com-amazon graph containing about 335,000 vertices and just over 900,000 edges; and the com-dblp network containing about 317,000 vertices and just under 1,050,000 edges. All of these networks represented some kind of social or organizational network and had ground-truth communities associated with them for evaluation.

The PPI network is a very small network containing just under 500 nodes and about 2000 edges. This is a biological protein-interaction network which contains pathways associ-

ated with the medical condition myocardial infarction (heart attack). Associated with this network was an incomplete gene-ontology containing labels for most of the vertices in the graph; those that were unlabeled were left in the graph but not considered during evaluation.

Finally, 30 synthetic benchmark networks of 3 different sizes, 250, 500, and 1000 vertices were created with 10 versions of each size. The algorithm used for this was the commonly used Lancichinetti-Fortunato-Radicchi (LFR) algorithm [23]. The LFR algorithm is designed to produce realistic clusters in the benchmark graphs. The implementation used provided ground-truth communities for each vertex in the graph.

In total, 34 networks were used to benchmark the performance of NemoCluster. This may sound like a small number, however, the network motif finding process requires a large amount of time, making it extremely time consuming to benchmark graphs of the sizes provided by the SNAP library.

5.3 Network Motifs

The network motifs for all graphs were produced using the NemoCollection feature of NemoLib [2, 22]. Network motifs of size three and four were found for all graphs, however, with the smaller graphs size five and six also were found. As the NemoCollection algorithm is NP-Complete, it is impossible to find size five and six network motifs on the large social network graphs such as the amazon or dblp graphs; however, for the small PPI network and benchmarking graphs it was possible to find size 5 for all and size six for the PPI network only. As the hypothesis is that NemoCluster will perform better the more network motifs are available to chose from, these additional motifs allowed for greater flexibility during testing.

5.4 Experimental Setup

The clustering threshold, λ, determines which edges are removed during the clustering process. The authors of Tectonic suggest it should be set to 0.06 [37] for ideal results in social networks. However, to ensure a fair comparison between NemoCluster and Tectonic, a varying threshold value $\lambda \in \{0.005k | k \in \{0, 1, 2, ..., 199, 200\}\}$ was used instead. This produced 200 data-points for each algorithm and included the recommended value of 0.06 allowing for a more thorough analysis.

The NemoCluster algorithm decays to the Tectonic algorithm when only triangle motifs are given as input. Thus, all benchmarks were conducted using only the NemoCluster program and for the Tectonic tests triangle motifs were supplied as input, while NemoCluster tests used a variety of different size motifs. While it would be possible to provide a combination of different motifs to NemoCluster to achieve higher accuracy results, due to time-constraints this was not done in the tests shown here.

NemoCluster and Tectonic are both deterministic algorithms, therefore all tests were only run a single time as the output will never change.

5.5 Evaluation

The evaluation of the clusters was modeled after the ARI scoring function [19] which is frequently used to evaluate machine learning algorithms. Unfortunately, the ARI metric requires clusters to be labeled, which is not the case in the graphs used here, therefore it could not be used directly. Thus, a function that behaves similarly was used to evaluate the clusters.

Consider the clustering results in Table 5.1, this graph contains five vertices and in both

Table 5.1: Example Clustering Results

Vertex	Ground Truth Cluster	Predicted Cluster
1	A	X
2	B	Y
3	C	Y
4	B	Z
5	A	X

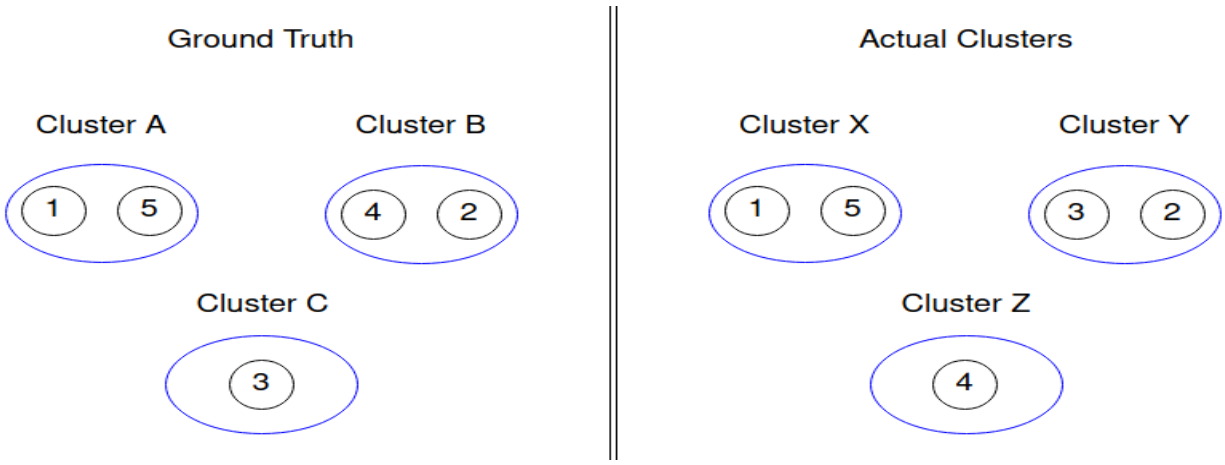

Figure 5.1: Example Clustering Results

the ground truth and predicted clusters they are divided over three clusters; however, the predicted clusters do not always match. This is visually represented in Fig. 5.1, while one cluster is a perfect match, another is completely wrong, and one is partially correct. This creates an issue when comparing them with the ground truth, especially considering that, in reality, most clusters will only be partial matches. The biggest problem created by this is, which clusters should be compared, in this case, it is possible to compare clusters B and Z as 50% match and C and Y as 50% match, but it is also possible to compare B and Y as a 50% match and C and Z as a 0% match; which is the correct way? Clearly, comparing the clusters themselves is ambiguous and results in biased results. **Note:** It is important to keep in mind that the labels of the clusters (i.e. "A" or "X") have no meaning, they are arbitrary, these exist solely to differentiate the clusters. Thus, it is not possible to compare the labels assigned to each vertex, if this was possible, the ARI metric could be used.

Table 5.2 shows the scoring of the previous clusters. The scoring function enumerates all pairs of vertices and then determines whether or not the algorithm correctly predicted that they should or should not be in the same cluster. This produces four possible outcomes. Firstly, the vertex pair $1, 2$ represents a True Negative (TN), the algorithm correctly predicted that these two vertices belong to different clusters (A and B, respectively). Secondly, the vertex pair $2, 4$ represents a False Negative (FN) as the algorithm predicted that they should not be in the same cluster, however, they should have been (both belong to cluster B). Similarly, the pair $2, 3$ represents a False Positive, in this case, the algorithm put them in the same cluster, but in reality they belong to different clusters (B and C, respectively). Finally, the pair $1, 5$ represents the only True Positive in this example, in this case the algorithm correctly predicted that both vertices belong to the same cluster.

This scoring function turns the clustering algorithm into a prediction algorithm that determines whether or not a pair should belong to the same cluster or not; thus making it possible to score the results without cluster labels. It does come with a drawback however,

Table 5.2: Example Clustering Scores

Vertex Pair	**Common Cluster Expected?**	**Common Cluster Predicted?**	**Classified**
1, 2	No	No	TN
1, 3	No	No	TN
1, 4	No	No	TN
1, 5	Yes	Yes	TP
2, 3	No	Yes	FP
2, 4	Yes	No	FN
2, 5	No	No	TN
3, 4	No	No	TN
3, 5	No	No	TN
4, 5	No	No	TN

Table 5.3: Example Clustering Statistics and Metrics

True Positives (TP)	**False Positive (FP)**	**False Negative (FN)**	**True Negative (TN)**
1	1	1	7

Metric	**Formula**	**Value**
Precision	$\frac{TP}{TP+FP}$	$\frac{1}{2} = 0.50$
Recall	$\frac{TP}{TP+FN}$	$\frac{1}{2} = 0.50$
F_β	$\frac{(1+\beta^2)\cdot TP}{(1+\beta^2)\cdot TP+\beta^2\cdot FN+FP}$	$\Big\vert_{\beta=1} \frac{2}{2+2+1} = 0.20$

namely, the vast majority of these pairs will be true negatives, thus shifting the focus to how well an algorithm predicts that two vertices should not belong to the same cluster.

Using these statistics, three metrics can be calculated (among others), namely, Precision, Recall, and the F-score, these are shown for the example in Table 5.3. The F-score is generally used with three values for β, namely, 0.5, 1 and 2; with β set to 1, both precision and recall are weighted equally, with β set to 0.5, recall is weighted as half as important as precision, and with β set to 2 recall is weighted twice as important as precision.

The drawback of this scoring function is that it is time-consuming to calculate as there are $\frac{(n-1)n}{2}$ pairs of vertices, but also that the precision value will reach 1 when all vertices are in their own cluster as there are no false-positives left. This means the precision value is a poor measure for performance and recall suffers a similar problem, but less significantly. The F-score is the most reliable of these metrics and will be used for comparing the two algorithms instead of precision and recall.

Chapter 6

ANALYSIS

This section discusses the results of benchmarks conducted to evaluate the performance difference between Tectonic and NemoCluster. All figures presented in this chapter will feature Tectonic on the left and NemoCluster on the right unless otherwise stated. In addition to this, all figures will use a logarithmic scale on the x-axis to highlight the lower clustering thresholds where change occurs; the y-axis on all graphs will be linear and show the range [0,1]. Finally, while the precision values will be shown in the figures below, these do not accurately represent performance (see Section 5.5 for details); the f-scores should be considered instead, as they are much better performance indicators.

In the social networks Tectonic and NemoCluster performed similarly, but NemoCluster was able to performs slightly better on the Amazon network as can be seen in Fig. 6.1, showing a PR curve, and Fig. 6.2, showing the f-scores.

On this network, Tectonic performed best at the clustering threshold of 0.07, where it achieved an $F_{0.5}$ score of 0.48, F_1 score of 0.46, and F_2 score of 0.44 and it assigned the vertices to a total of 98,270 clusters, slightly overshooting the expected 75,149; after this point Tectonic assigns almost all vertices to their own cluster, suggesting that the ideal clustering threshold is 0.07 for this graph, very close to the value of 0.06 suggested by [37]. NemoCluster shows similar behavior, in this case a size 4 motif was used and it resulted in an $F_{0.5}$ score of 0.58, F_1 score of 0.50, and F_2 score of 0.44 and it assigned the vertices to a total of 136,199 clusters, almost double the expected. However, the $F_{0.5}$ and F_1 scores are significantly larger than those of Tectonic, suggesting that the clusters were of higher

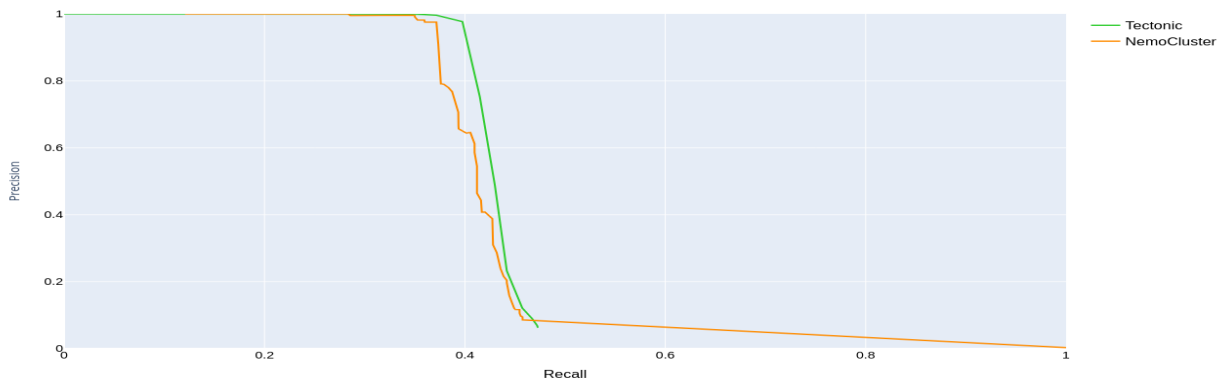

Figure 6.1: Precision vs Recall of Tectonic and NemoCluster on the Amazon network from [26].

quality overall, even though more incorrect clusters were created. In addition to this, the area-under-the-curve (AuC) for the ROC curve of this data is 0.72 for NemoCluster, while it is only 0.01 for Tectonic. This is due to the fact that NemoCluster is able to produce better sized clusters. As the threshold is increased, Tectonic quickly puts all vertices in their own clusters, resulting in a very fast drop of the true-positive rate (TPR) to 0. However, in terms of average precision, NemoCluster and Tectonic both achieve about the same, with 0.41 and 0.42, respectively.

This shows that, even in social networks where cliques have special meaning, network motifs can improve the results of clustering over just using cliques. The main reason for this performance on this graph is likely the fact that it is not a social network in the sense of personal connection, but rather products that are similar. If the network was taken from a service such as Facebook, cliques would likely outperform network motifs significantly; however, as this graph represents items that are frequently purchased together, network motifs have more meaning and convey more useful information. One thing that should be noted about the curves is that, Tectonic produces much smoother curves than NemoCluster. This

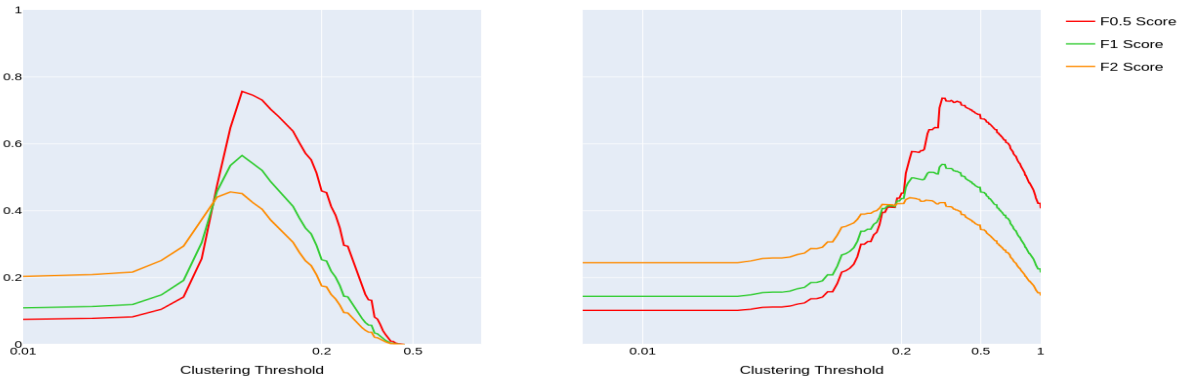

Figure 6.2: F-scores of Tectonic and NemoCluster on the Amazon network from [26].

means that NemoCluster plateaus at certain points, suggesting that the use of a more complicated motif, or just more motifs might produce better results by smoothing the curve out and potentially increasing its peak before performance decays as vertices are clustered by themselves. Due to the size of this network, it was not possible to investigate this hypothesis in the limited time available; however, this is a trend that persists through all test cases.

In stark contrast to the previous findings are those shown in Fig. 6.3 and Fig. 6.4. Much like before, both tools perform roughly equal, but the performance is awful in this case.

The email-core network showed very different results in terms of performance than that of the Amazon network. Previously, both tools managed to achieve around 0.5 on the f-scores, in this case, neither tool even achieves even 0.25 for any of them. In the case of the $F_{0.5}$ score, Tectonic achieves a maximum of 0.22 and NemoCluster achieves a static score of 0.06; with the F_1 score, Tectonic maxed out at 0.18 and NemoCluster at 0.1; the F_2 score is no different, Tectonic reaches a maximum at 0.19, but at the peak performance only achieve 0.16, NemoCluster achieves a score of 0.18. Average precision and AuC for the PR curve show the same thing, both tools achieve very low scores. These results show two interesting

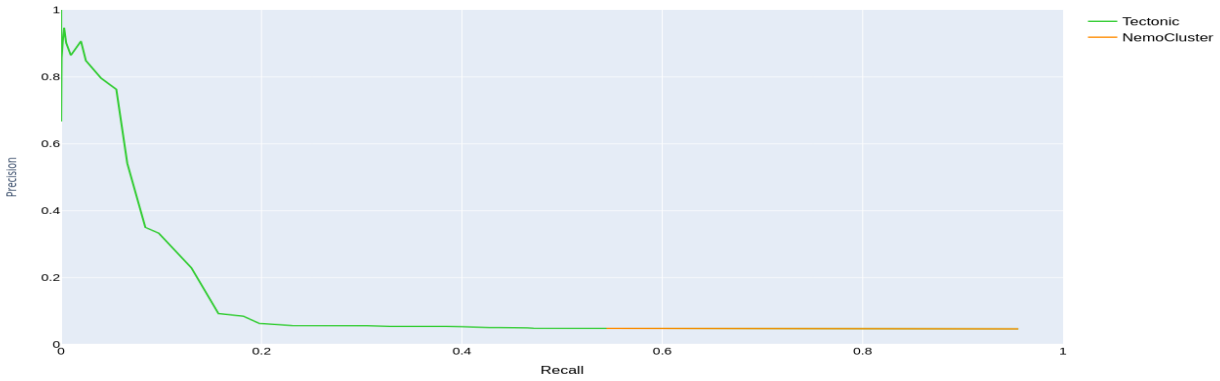

Figure 6.3: Precision vs Recall of Tectonic and NemoCluster on the Email-Core network from [26].

things; firstly, NemoCluster appears to be uneffected by the varying threshold, the scores are horizontal lines for the most part; secondly, both tools perform horribly on this network.

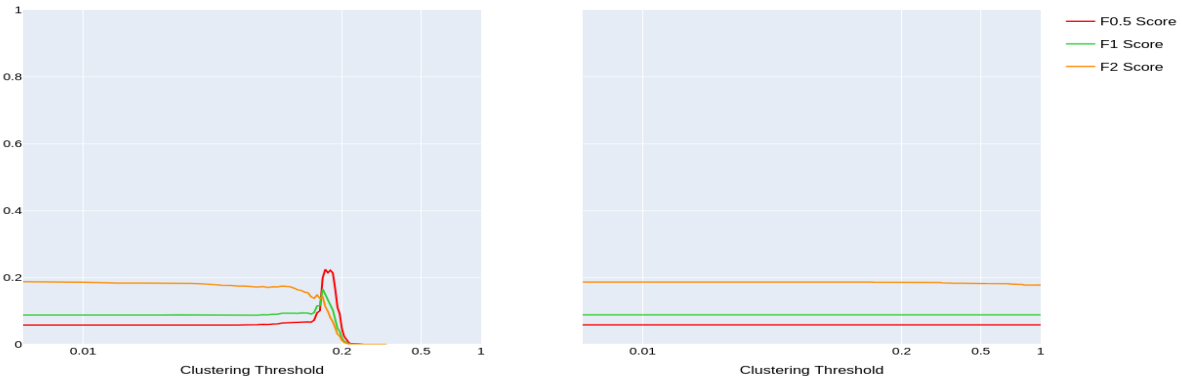

Figure 6.4: F-scores of Tectonic and NemoCluster on the Email-Core network from [26].

Beginning with the first point, the question arises why does the varying threshold not

impact NemoCluster in this situation? For thresholds greater than 0, NemoCluster maps the vertices into clusters in the range [131, 275] which is significantly larger than the expected number of 42 clusters. With a threshold of 0, it puts them into 20 clusters, but this threshold means "do nothing", so NemoCluster apparently performs better when it does nothing in this case. Tectonic behaves similarly to NemoCluster, however, the number of clusters very quickly explode to the number of vertices which is 1004 for this network. The difference in this performance comes from the fact that NemoCluster uses a network motif, while Tectonic uses cliques and cliques are far more susceptible to a problem in this network than network motifs.

Table 6.1: Statistics for NemoCluster (NemoC.) and Tectonic for all data-sets.

Data Set	**Avg. Precision**		**AuC PR**		**AuC ROC**	
	Tectonic	**NemoC.**	**Tectonic**	**NemoC.**	**Tectonic**	**NemoC.**
Amazon	0.42	0.41	0.43	0.32	0.01	0.72
Email	0.11	0.05	0.12	0.02	0.49	0.32
PPI	0.41	0.42	0.43	0.35	0.56	0.57
Syn. 250	0.42	0.39	0.68	0.55	0.67	0.66
Syn. 500	0.30	0.47	0.64	0.62	0.63	0.73
Syn. 1000	0.29	0.41	0.63	0.54	0.63	0.69

Coming to the second interesting part, why do they perform so poorly here when they did quite well before? Taking a closer look at the network's properties sheds some light on this, there are 1004 vertices and 25,571 edges; while this graph is far from being complete, it is significantly closer than the amazon graph in the first example which had 334,863 vertices and 925,872 edges. The number of triangles in the email network is 105,461 meaning each

vertex is part of approximately 105 triangles; in the amazon network there are 667,129, so each vertex is part of approximately 3 triangles [26]. This is the reason why the performance differs so strongly between these two graphs and why Tectonic is more impacted by this than NemoCluster. The email network is very dense and contains a large fraction of triangles compared to vertices, for this reason Tectonic does not perform well on it as it depends on triangles being well spread out to detect communities. NemoCluster has a similar issue, network motifs are only rare in sparse graphs, in connected graphs such as this they are very frequent, meaning there is not much to filter on, this is why the curve is so flat, most edges are never removed due to the high frequency of the motifs. This is also the reason why the performance of Tectonic spikes and drops immediately, most edges in the graph have a triangle conductance of around 0.15, meaning once that threshold is reached, most edges are removed from the graph in very quick succession, resulting in a short spike of the number of True Positives.

Similar performance to that in the email network can be seen in the protein-protein-interaction (PPI) network shown in Fig. 6.5 and Fig. 6.6.

While it is not as pronounced here as in the email network, NemoCluster has the same flat performance and the graphs show sharp drops in performance over very small changes to the threshold. In this case, the graph has just under 500 vertices and about 1800 edges with 6804 triangles, so the triangle frequency is about 13 triangles per vertex. This supports the hypothesis that as the number of triangles per vertex increases, the performance of the algorithms will drop significantly.

These two examples clearly show that this approach of clustering is not fit for applications where densely connected graphs are in use. Somewhere between 13 and 100 triangles per vertex the network becomes too dense. However, at this lower bound both Tectonic and NemoCluster still perform quite well. In dense graphs, the number of motifs or cliques is to

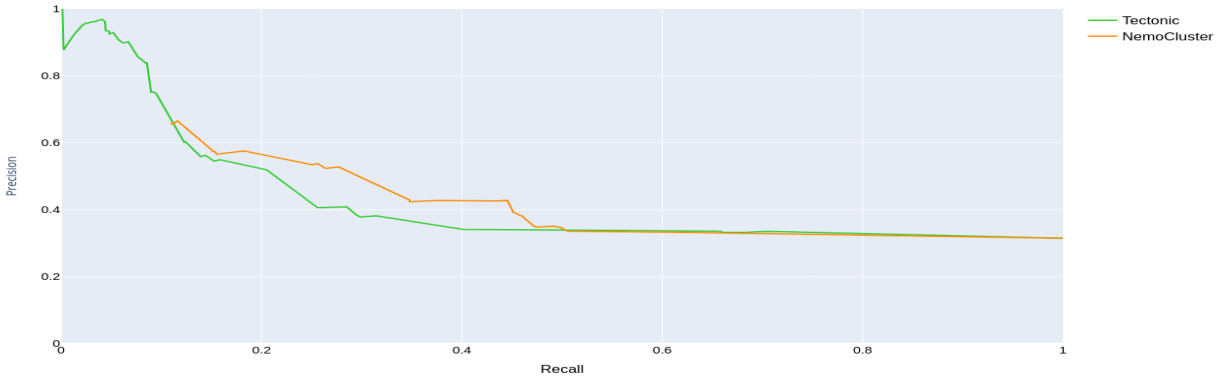

Figure 6.5: Precision vs Recall of Tectonic and NemoCluster on the Protein network from [38].

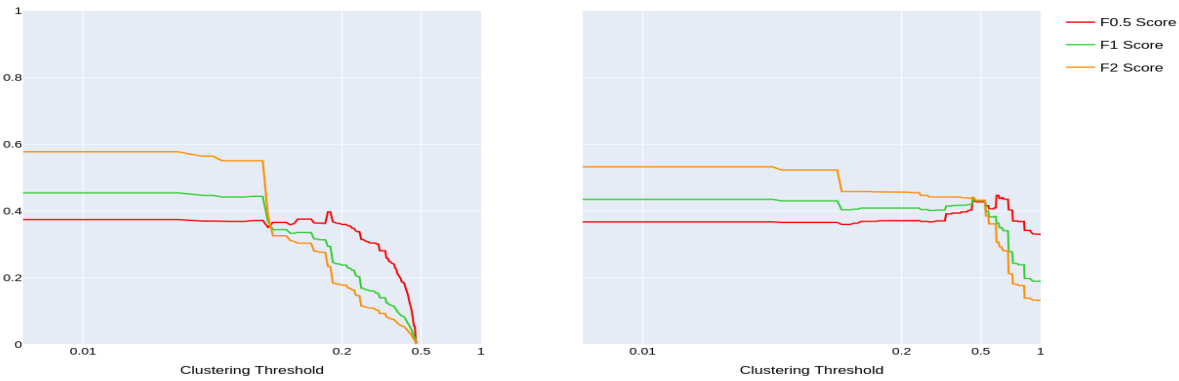

Figure 6.6: F-scores of Tectonic and NemoCluster on the Protein network from [38].

high to allow the algorithms to distinguish between edges that are important, i.e. part of a cluster, and edges that need to be removed.

The real performance improvements of NemoCluster over Tectonic are best shown by the benchmark graphs shown in Fig. 6.7 - 6.12. As these graphs are much smaller than those previously discussed, it was possible to use larger motifs, size five, instead of size four used previously.

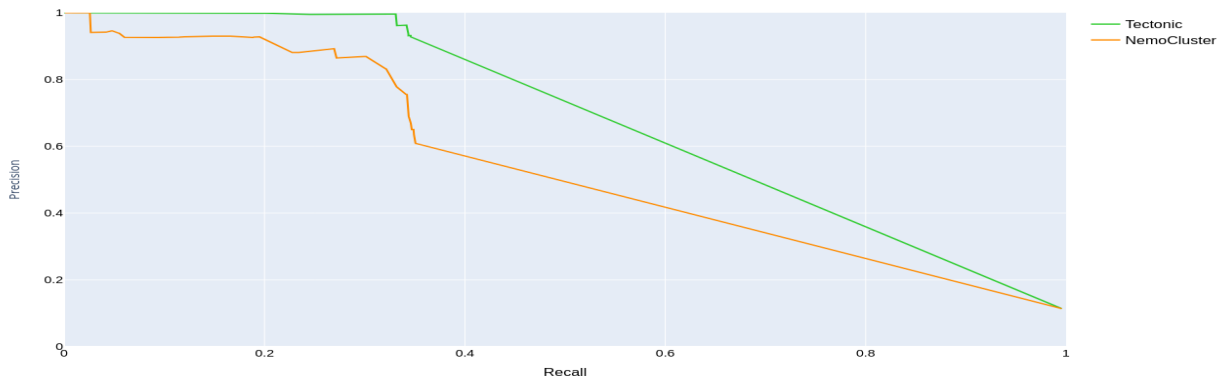

Figure 6.7: Precision vs Recall of Tectonic and NemoCluster on the 250 Vertex synthetic networks (average over 10 networks).

On the 250 vertex benchmarking graph, NemoCluster outperformed Tectonic by a significant amount. The results shown in Fig. 6.7 show the performance of Tectonic and NemoCluster averaged over ten different networks of this size, all slightly differing in make-up. On average, the triangle frequency was about one triangle per vertex, with all graphs having 250 vertices and approximately 500 edges; thus, these graphs were significantly more sparse than the social networks. On these graphs, Tectonic and NemoCluster both achieve an $F_{0.5}$ score of about 0.7; with the F_1 score, Tectonic maxed out at 0.5 and NemoCluster

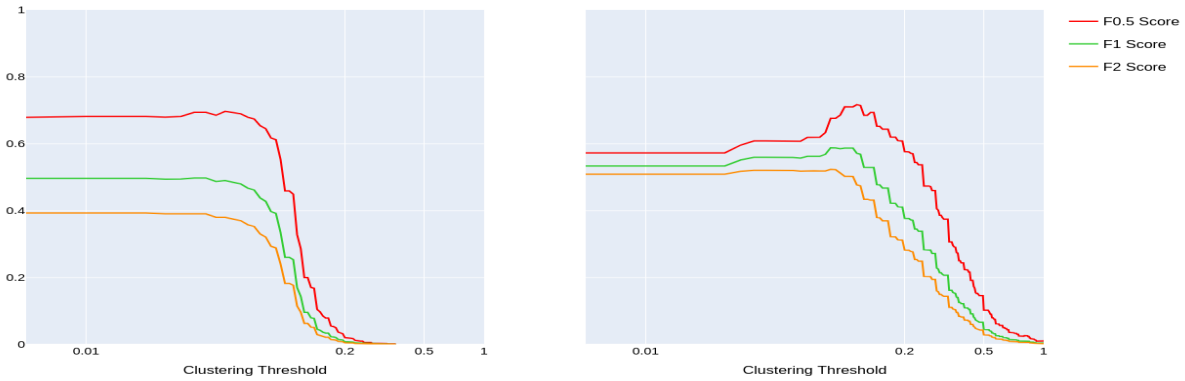

Figure 6.8: F-scores of Tectonic and NemoCluster on the 250 Vertex synthetic networks (average over 10 networks).

at 0.59; finally, the F_2 score is no different, Tectonic reaches a maximum at 0.39 and NemoCluster manages about 0.49. Similarly, the average precision and AuC values for both are also very close, only the AuC for the PR curve varies significantly, with Tectonic getting 0.68 and NemoCluster 0.55. This comes mostly due to the fact that NemoCluster has a much higher recall than Tectonic, while Tectonic scores higher in terms of precision; however, this is because Tectonic immediately puts the vertices into about 190 clusters, while NemoCluster puts them into about 80 clusters, the true number of clusters in these examples is between 8 and 15. These results suggest that for very small graphs, in terms of edges, the algorithms do not perform so well as they are based on removing edges to find clusters. In these graphs, the number of edges was just about twice the number of vertices, far less than in any of the previous examples.

Fig. 6.9 displays the results for the 500 vertex benchmarking graph. This plot shows similar performance as with the 250 vertex graphs, though the divide between Tectonic and NemoCluster widens. The $F_{0.5}$ score of NemoCluster peaks at about 0.7, with Tectonic reaching about 0.62; for the F_1 score, Tectonic got approximately 0.41 and NemoCluster got 0.58;

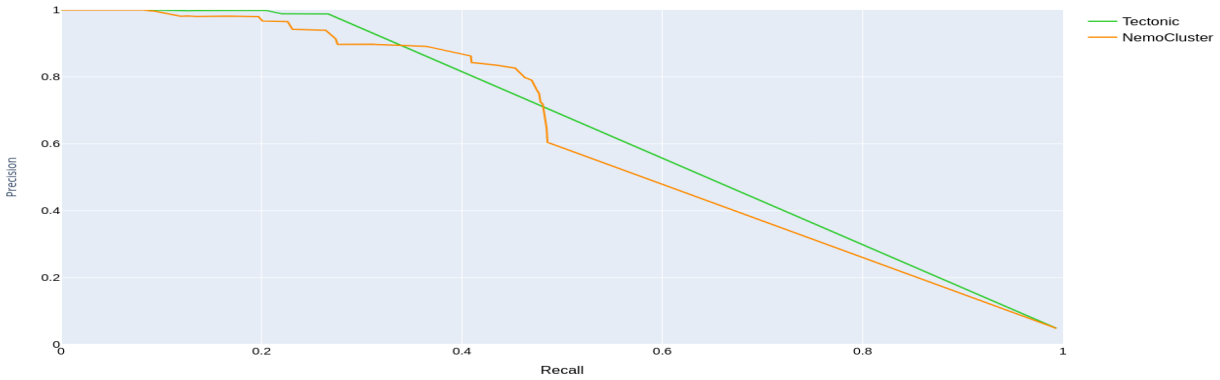

Figure 6.9: Precision vs Recall of Tectonic and NemoCluster on the 500 Vertex synthetic networks (average over 10 networks).

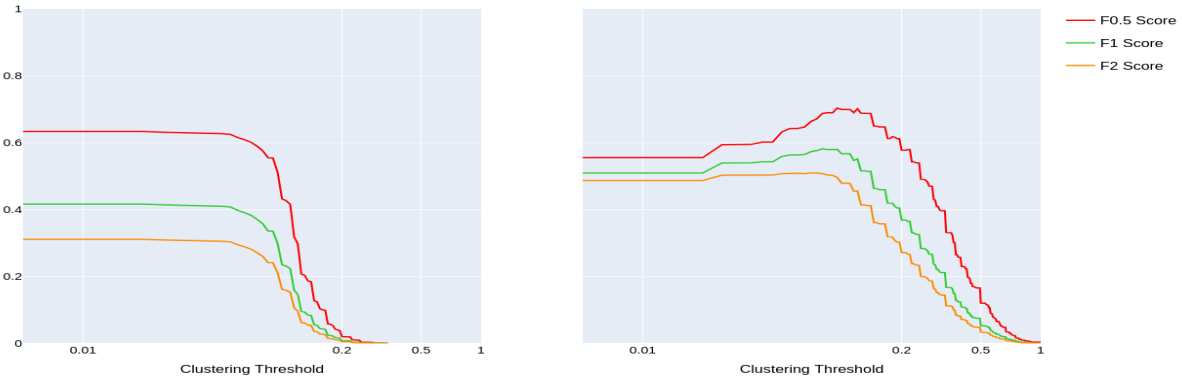

Figure 6.10: F-scores of Tectonic and NemoCluster on the 500 Vertex synthetic networks (average over 10 networks).

the F_2 score further widens the gap, Tectonic reaches a maximum at 0.31 and NemoCluster manages about 0.49. Much like the f-scores are diverging, so are the average precision and AuC values. Tectonic still manages to slightly outperform NemoCluster in terms of AuC for the PR curve, with 0.64 compared to 0.62. However, NemoCluster achieves much higher average precision with 0.47 compared to 0.3 and also much higher AuC for the ROC curve with 0.73 compared to 0.63. Once again, Tectonic is struggling with this graph because the number of edges is only double the number of vertices, however, NemoCluster is actually performing slightly better on this larger graph. This shows one of the main advantages of the use of Network Motifs, in these sparse graphs, there aren't going to be many cliques, thus, Tectonic has little to work with. NemoCluster, on the other hand, can be adjusted to use a different network motif that better utilizes the available information, thus being much more flexible.

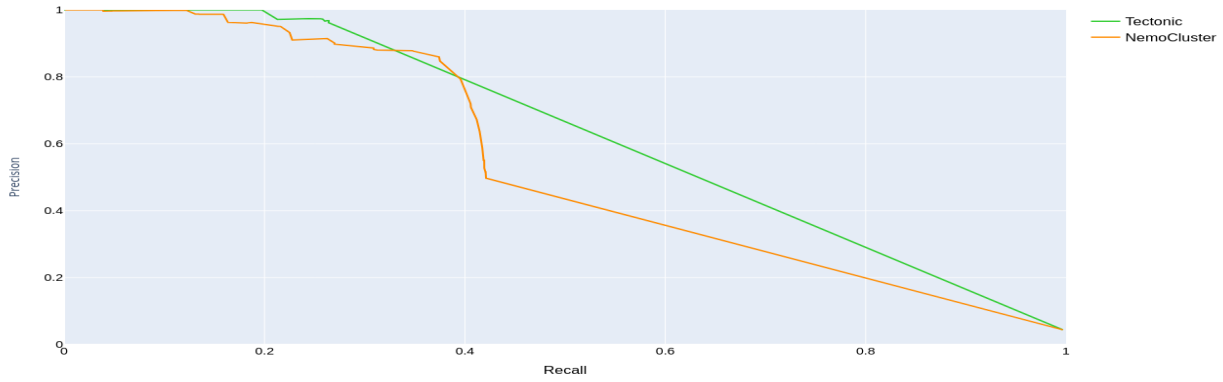

Figure 6.11: Precision vs Recall of Tectonic and NemoCluster on the 1000 Vertex synthetic networks (average over 10 networks).

Lastly, the final group of benchmark graphs had 1000 vertices and the performance plots are shown in Fig. 6.11. The results for this graph are very similar to the previous two. Once

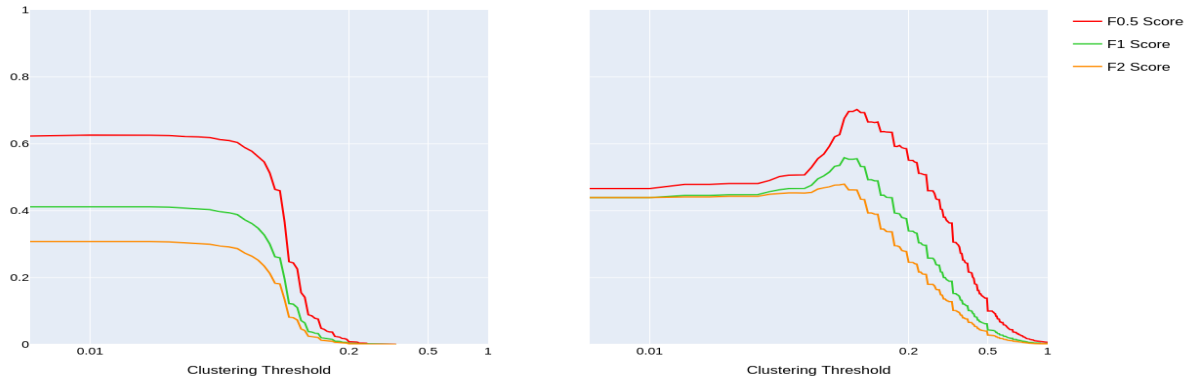

Figure 6.12: F-scores of Tectonic and NemoCluster on the 1000 Vertex synthetic networks (average over 10 networks).

again, NemoCluster peaks it's performance at a threshold of 0.1 with an $F_{0.5}$ score of about 0.7, with Tectonic reaching about 0.62; for the F_1 score, Tectonic got approximately 0.41 and NemoCluster got 0.57; the F_2 score of Tectonic is round about 0.32 and for NemoCluster it is about 0.48. In terms of average precision, the gap got a little bit smaller here, but not by much, NemoCluster achieved 0.41 and Tectonic only 0.29. The AuC for the ROC curve now has NemoCluster at 0.69 and Tectonic at 0.63. Once again, it is clear that Tectonic is having issues with this type of graph, the average cluster size in all of these three examples at the threshold of 0.06 (as recommended by [37]) was less than 1.5; meaning the vast majority of nodes were in their own clusters, resulting in the high precision values that can be seen in Fig. 6.7 through 6.11. As mentioned in Section 5.5, the reason for the increasing precision is due to how it is calculated, as the average cluster size decreases, the number of true-positives and false-positives both decrease, but false-positives decrease faster, thus the score approaches one as the average cluster size approaches one. That is why NemoCluster's performance is better, even though it has much lower precision, the clustering is of higher quality, as can be seen by the larger recall value, which depends on false negatives and those are inversely proportional to cluster size.

Chapter 7

CONCLUSION

7.1 Overview

This section will provide the overall conclusions for the two main topics presented in this thesis. Firstly, the conclusions of the benchmarks of existing tools, first presented in Chapter 3, are discussed in Section 7.2. After this, the conclusions regarding the analysis of Nemo-Cluster is discussed in Section 7.3.

7.2 Benchmarking

This thesis presented benchmarking results to compare several SV detection tools: CNVNator, Breakdancer-Max, Gustaf, SVDetect, and LUMPY. The selected tools represent RC, RP, SR, and combinations of the methods. Various data-sets were simulated that include a selection of SVs, such as deletions, insertions, duplications, and inversions. The tools were compared with PR curves, their usability, and execution time. In addition to this, a practical guideline was proposed on what tools could be useful in which types of experiments. The results show that there is no single tool that can detect all SVs with high accuracy in microbial genomes.

Through the benchmarks, a list of suggestions was compiled for each tool. CNVNator is fast and fairly simple to use, with the cost of low detection rates, however, the calls that it does make are mostly correct. Breakdancer-Max, although the fastest, does not provide reliable results, its detection rate is low and the precision is even lower. Gustaf is slow and challenging to use, in exchange for this it performs in the middle of the pack. The two tools

that outperform the others, in this experiment, are SVDetect and LUMPY. While SVDetect lacks usability, especially for automation, it can call all variants with high precision and recall. The read coverage, also, has little effect on the precision and recall of SVDetect, meaning that good results can be obtained from low coverage reads. LUMPY cannot detect insertions, making it less useful than SVDetect; however, it provided good detection rates with reasonable execution time. Furthermore, LUMPY was the easiest to use, it comes bundled with the speedseq alignment tool, and it offers good scalability as read coverage increases in terms of execution time.

Overall, this experiment shows that LUMPY and SVDetect would be preferable for use with microorganisms. These tools both use the RP method mixed with the SR and RC methods, respectively, suggesting that the RP method is a good candidate for SV detection in microgenomes. However, the breadth of this experiment is not large enough to make any conclusions about the effectiveness of the methods themselves, only the implementations thereof.

7.3 NemoCluster

NemoCluster, first introduced in this thesis, is a program that generalizes the triangle-conductance clustering approach [37] to any network motif of any size. The goal of this is to allow the algorithms users to have greater control about what structural information to use during the clustering. In social networking applications, cliques have special meaning, however, in other fields, such as Bioinformatics, a variety of motifs have meaning, while cliques may not.

Accompanying NemoCluster are benchmarks that show it performs better in a variety of scenarios; namely, benchmarks included the original social networks, a protein-protein-interaction (PPI) network, and different sizes of benchmarking graphs. These benchmarks led to a number of conclusions about NemoCluster: Firstly, NemoCluster performs better

when larger motifs are used. Secondly, NemoCluster does not perform well on very dense graphs due to the high motif density in these graphs, making the conductance of all edges roughly the same. Thirdly, NemoCluster is less sensitive to the exact type of networks used compared to similar algorithms. This adaptability shows that network motifs are a superior clustering method outside of social networking applications, where the two perform roughly equally. Through these benchmarks, it was possible to determine that the graphs where the algorithms perform well have a triangle-to-vertex ratio in the range [13, 100], where performance decays as the ratio reaches 100 and as it drops below 13. Finally, in sparse graphs, NemoCluster performs significantly better than a clique-based approach as network motifs are more adaptable in this situation. A sparse graph will contain few, if any, cliques but can contain plenty of motifs for a user to choose from. However, if the graph becomes too sparse, then NemoCluster's performance will decay as there are not enough motifs available. In all circumstances, network motifs with a z-score in the range [30,70] performed best with NemoCluster; this suggests that the frequency of the motifs must be right to be able to use this approach. This is likely the explanation why very sparse graphs do not achieve

In conclusion, the benchmarks of NemoCluster show that it either performs better than, or roughly equal, to Tectonic making it the better choice in many cases. In social networking applications, Tectonic is likely a better choice as cliques have a special meaning in that domain and require less time to find. In other fields, though, the fact that network motifs have more flexibility makes NemoCluster a more adaptable choice that can be molded to the specific input graph to achieve better performance.

7.4 Goals

In Chapter 1, two goals were laid out for this research, namely, to build a benchmarking suite for structural variant calling tools that uses microbial genomes instead of the human genome; and to build a network motif based clustering algorithm that can be used by a new

structural variant calling tool to cluster read-graphs.

The first goal was met in December, the tool was completed and a number of benchmarks were conducted, shown in 3 and published in [33]. The suite is available on GitLab and includes detailed documentation explaining how the software can be used, how new tools can be added, and how new genomes can be added for testing. In addition to this, there is a list of suggested improvements for both the software suite for future students to work on.

Building the clustering algorithm and analyzing its performance was the main goal of this thesis. NemoCluster has been shown to perform well in all scenarios that were benchmarked, except in very dense graphs; it either outperformed or was roughly equal to Tectonic. In addition to this, NemoCluster also is less susceptible to the negative properties of very dense or very sparse graphs as it does not depend on cliques. Overall, NemoCluster has met the goals set in Chapter 1 and it was also accepted for oral presentation at the ISBRA 2020 conference, though not for publication; which, as it was accepted for oral presentation, is likely due to the fact that it is not strictly a Bioinformatics algorithm and belongs in a Graph or Clustering journal or conference.

7.5 Future Work

One major feature of NemoCluster is that it allows the combination of different motifs to be used. In theory, this should allow the algorithm to be fine-tuned even further to achieve higher accuracy. Due to time constraints, it was not possible to thoroughly investigate this hypothesis.

During the testing of NemoCluster, it became apparent that certain motifs result in better performance than others. Network motifs with a z-score lower than 30 resulted in poor clusters, similarly, z-score values over 70 resulted in poor clusters, too. The question that

remains is, what exactly is the relationship between the clusters and the z-score? This is especially interesting in relation to motif combinations; is is possible to combine two motifs with z-score 20, which tend to perform poorly, and achieve good results as the combined score is in the good range?

REFERENCES

[1] Alexej Abyzov, Alexander E. Urban, Michael Snyder, and Mark Gerstein. Cnvnator: An approach to discover, genotype, and characterize typical and atypical cnvs from family and population genome sequencing. *Genome Research*, 21(6):974–984, 2011.

[2] Andrew Andersen and Wooyoung Kim. *NemoLib: A Java Library for Efficient Network Motif Detection*, pages 403–407. Bioinformatics Research and Applications: 13th International Symposium, ISBRA 2017, Honolulu, HI, USA, May 29 – June 2, 2017, Proceedings. Springer International Publishing, Cham, 2017.

[3] Christoph Bartenhagen. Rsvsim: an r/bioconductor package for the simulation of structural variations. `https://www.bioconductor.org/packages/release/bioc/html/RSVSim.html`, 2019.

[4] Austin R. Benson, David F. Gleich, and Jure Leskovec. Tensor spectral clustering for partitioning higher-order network structures. In *Proceedings of the 2015 SIAM International Conference on Data Mining*, pages 118–126, 2015.

[5] Colby Chiang, Ryan M. Layer, Gregory G. Faust, Michael R. Lindberg, David B. Rose, Erik P. Garrison, Gabor T. Marth, Aaron R. Quinlan, and Ira M. Hall. Speedseq: ultra-fast personal genome analysis and interpretation. *Nature Methods*, 12:966, Aug 2015.

[6] Yann Collet et al. Lz4 - extremely fast compression. `https://github.com/lz4/lz4`.

[7] Jennifer Commins, Christina Toft, and Mario A. Fares. Computational biology methods and their application to the comparative genomics of endocellular symbiotic bacteria of insects. *Biological Procedures Online*, 2009.

[8] Paramvir S. Dehal, Marcin P. Joachimiak, Morgan N. Price, John T. Bates, Jason K. Baumohl, Dylan Chivian, Greg D. Friedland, Katherine H. Huang, Keith Keller, Pavel S. Novichkov, Inna L. Dubchak, Eric J. Alm, and Adam P. Arkin. Microbesonline: an integrated portal for comparative and functional genomics. *Nucleic Acids Research*, 38(suppl_1):D396–D400, 11 2009.

[9] Scott Emmons, Stephen Kobourov, Mike Gallant, and Katy Boerner. Analysis of network clustering algorithms and cluster quality metrics at scale. *PLOS ONE*, 11(7):1–18, 07 2016.

[10] David Eppstein. Pseudoforest.svg. `https://en.wikipedia.org/wiki/File:Pseudoforest.svg`.

[11] Xian Fan, Travis E. Abbott, David Larson, and Ken Chen. Breakdancer: Identification of genomic structural variation from paired-end read mapping. *Current Protocols in Bioinformatics*, 45(1):15.6.1–15.6.11, 2014.

[12] Finishing the euchromatic sequence of the human genome. *Nature*, 431(7011):931–945, 2004.

[13] Ann Griswold. Genome packaging in prokaryotes: the circular chromosome of e. coli. *Nature News*, 2008.

[14] Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. Exploring network structure, dynamics, and function using networkx. In Gaël Varoquaux, Travis Vaught, and Jarrod Millman, editors, *Proceedings of the 7th Python in Science Conference*, pages 11 – 15, Pasadena, CA USA, 2008.

[15] RE Handsaker, V Van Doren, JR Berman, G Genovese, S Kashin, LM Boettger, and SA McCarroll. Large multiallelic copy number variations in humans. *Nature Genetics*, 15, 2015.

[16] SN Hart, V Sarangi, R Moore, S Baheti, JD Bhavsar, FJ Couch, and et al. SoftSearch: Integration of Multiple Sequence Features to Identify Breakpoints of Structural Variations. *PLoS ONE*, 8(12), 2013.

[17] Erez Hartuv and Ron Shamir. A clustering algorithm based on graph connectivity. *Information Processing Letters*, 76(4):175 – 181, 2000.

[18] Weichun Huang, Leping Li, Jason R. Myers, and Gabor T. Marth. Art: a next-generation sequencing read simulator. *Bioinformatics*, 28(4):593–594, 12 2011.

[19] Lawrence Hubert and Phipps Arabie. Comparing partitions. *Journal of Classification*, 2:193–218, 1985.

[20] P. K. Jana and A. Naik. An efficient minimum spanning tree based clustering algorithm. In *2009 Proceeding of International Conference on Methods and Models in Computer Science (ICM2CS)*, pages 1–5, 2009.

[21] R. A. Jarvis and E. A. Patrick. Clustering using a similarity measure based on shared near neighbors. *IEEE Transactions on Computers*, C-22(11):1025–1034, 1973.

[22] Wooyoung Kim and Lynnette Haukap. NemoProfile as an efficient approach to network motif analysis with instance collection. *BMC Bioinformatics*, 18(12):423, October 2017.

[23] Andrea Lancichinetti, Santo Fortunato, and Filippo Radicchi. Benchmark graphs for testing community detection algorithms. *Physical Review E*, 78(4), Oct 2008.

[24] Ryan M. Layer, Colby Chiang, Aaron R. Quinlan, and Ira M. Hall. Lumpy: a probabilistic framework for structural variant discovery. *Genome Biology*, 15, 2014.

[25] Eric Lehman, F Thomson Leighton, and Albert R Meyer. *Mathematics for Computer Science*. MIT Press, May 2015.

[26] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data`, June 2014.

[27] Heng Li and Richard Durbin. Fast and accurate short read alignment with burrows–wheeler transform. *Bioinformatics*, 25(14):1754–1760, 05 2009.

[28] Geoff Macintyre, Bauke Ylstra, and James D. Brenton. Sequencing structural variants in cancer for precision therapeutics. *Trends in Genetics*, 32(9), 2016.

[29] Jurgen F. Nijkamp, Marcel A. van den Broek, Jan-Maarten A. Geertman, Marcel J. T. Reinders, Jean-Marc G. Daran, and Dick de Ridder. De novo detection of copy number variation by co-assembly. *Bioinformatics*, 28(24):3195–3202, 10 2012.

[30] Duarte C. Oliveira and Herminia de Lencastre. Multiplex pcr strategy for rapid identification of structural types and variants of the mec element in methicillin-resistant staphylococcus aureus. *Antimicrobial Agents and Chemotherapy*, 46(7):2155–2161, 2002.

[31] Chandra Shekhar Pareek, Rafal Smoczynski, and Andrzej Tretyn. Sequencing technologies and genome sequencing. *Journal of Applied Genetics*, 52(4):413–435, 2011.

[32] Tobias Rausch, Thomas Zichner, Andreas Schlattl, Adrian M. Stütz, Vladimir Benes, and Jan O. Korbel. Delly: structural variant discovery by integrated paired-end and split-read analysis. *Bioinformatics*, 28(18):i333–i339, 09 2012.

[33] N. Rohde and W. Kim. Benchmarking of structural variant detection tools for microorganisms. In *2019 IEEE International Conference on Bioinformatics and Biomedicine (BIBM)*, pages 2005–2012, 2019.

[34] Jeffrey A. Rosenfeld, Christopher E. Mason, and Todd M. Smith. Limitations of the human reference genome for personalized genomics. *LoS ONE*, 7(7), 2012.

[35] Lorenzo Tattini, Romina D'Aurizio, and Alberto Magi. Detection of genomic structural variants from next-generation sequencing data. *Frontiers in Bioengineering and Biotechnology*, pages 3–92, 2015.

[36] Kathrin Trappe, Anne-Katrin Emde, Hans-Christian Ehrlich, and Knut Reinert. Gustaf: Detecting and correctly classifying svs in the ngs twilight zone. *Bioinformatics*, 30(24):3484–3490, 07 2014.

[37] Charalampos E. Tsourakakis, Jakub W. Pachocki, and Michael Mitzenmacher. Scalable motif-aware graph clustering. *CoRR*, abs/1606.06235, 2016.

[38] Danila Vella, Simone Marini, Francesca Vitali, Dario Di Silvestre, Giancarlo Mauri, and Riccardo Bellazzi. Mtgo: Ppi network analysis via topological and functional module identification. *Scientific Reports*, 8(1), Mar 2018.

[39] J. Wang, C. G. Mullighan, J. Easton, S. Roberts, S.L. Heatley, and et al. CREST maps somatic structural variation in cancer genomes with base-pair resolution. *Nature Genetics*, 8, 2011.

[40] James L. Weber and Eugene W. Myers. Human whole-genome shotgun sequencing. *Genome Research*, 7:401–409, 1997.

[41] Sebastian Wernicke and Florian Rasche. FANMOD: a tool for fast network motif detection. *Bioinformatics*, 22(9):1152–1153, 02 2006.

[42] Kai Ye, Marcel H. Schulz, Quan Long, Rolf Apweiler, and Zemin Ning. Pindel: a pattern growth approach to detect break points of large deletions and medium sized insertions from paired-end short reads. *Bioinformatics*, 25(21):2865–2871, 06 2009.

[43] Hao Yin, Austin R. Benson, Jure Leskovec, and David F. Gleich. Local higher-order graph clustering. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '17, page 555–564, New York, NY, USA, 2017. Association for Computing Machinery.

[44] Bruno Zeitouni, Valentina Boeva, Isabelle Janoueix-Lerosey, Sophie Loeillet, Patricia Legoix-né, Alain Nicolas, Olivier Delattre, and Emmanuel Barillot. Svdetect: a tool to identify genomic structural variations from paired-end and mate-pair sequencing data. *Bioinformatics*, 26(15):1895–1896, 07 2010.

Appendix A
SV TOOL BENCHMARKING DATA

Shown below are the numerical results discussed in Chapter 3. In addition to the precision values discussed there, recall values are also shown for reference. The results are divided by the tools, Section A.1 shows the results for Breakdancer Max, Section A.2 shows the results for CNVNator, Section A.3 shows the results for Gustaf, Section A.4 shows the results for LUMPY, and Section A.5 shows the results for SVDetect. The only exception to this division is the execution times, shown in Section A.6, which shows all tools for each data-set. All results reference the data-sets discussed in Chapter 3, refer to Table 3.1 for details.

A.1 Breakdancer Max

Tables A.1 - A.4 show the precision values discussed in Chapter 3 and Tables A.5 - A.8 show supplemental recall values for reference only. Breakdancer Max does not call duplications; therefore, results for data-set 3 (duplications only) are omitted.

A.2 CNVNator

Tables A.9 - A.11 show the precision values discussed in Chapter 3 and Tables A.12 - A.14 show supplemental recall values for reference only. CNVNator does not call insertions or inversions; therefore, results for data-set 4 (insertions only) and data-set 5 (inversions only) are omitted.

Table A.1: Precision Results for Breakdancer Max on Data-set 1 (All Variants).

Threshold (bp)	**Coverage**			
	20x	**30x**	**40x**	**50x**
20	0.010582	0.004950	0.009217	0.009091
40	0.031746	0.029703	0.027650	0.022727
80	0.058201	0.064356	0.059908	0.059091
160	0.058201	0.064356	0.059908	0.059091
320	0.063492	0.069307	0.064516	0.063636
640	0.084656	0.089109	0.087558	0.086364
1280	0.095238	0.099010	0.096774	0.095455
2560	0.105820	0.113861	0.110599	0.109091
5120	0.132275	0.138614	0.138249	0.140909
10240	0.153439	0.158416	0.156682	0.163636

Table A.2: Precision Results for Breakdancer Max on Data-set 2 (deletions only).

Threshold (bp)	**Coverage**			
	20x	**30x**	**40x**	**50x**
20	0.000000	0.000000	0.000000	0.000000
40	0.083333	0.000000	0.000000	0.000000
80	0.208333	0.172414	0.147059	0.138889
160	0.208333	0.172414	0.147059	0.138889
320	0.208333	0.172414	0.147059	0.138889
640	0.208333	0.172414	0.147059	0.138889
1280	0.208333	0.172414	0.147059	0.138889
2560	0.291667	0.241379	0.205882	0.194444
5120	0.291667	0.241379	0.205882	0.194444
10240	0.416667	0.344828	0.294118	0.277778

Table A.3: Precision Results for Breakdancer Max on data-set 4 (insertions only).

Threshold (bp)	**Coverage**			
	20x	**30x**	**40x**	**50x**
20	0.000000	0.000000	0.000000	0.000000
40	0.000000	0.000000	0.000000	0.000000
80	0.013158	0.012048	0.011628	0.021739
160	0.013158	0.012048	0.011628	0.021739
320	0.013158	0.012048	0.011628	0.021739
640	0.026316	0.024096	0.023256	0.032609
1280	0.026316	0.024096	0.023256	0.032609
2560	0.026316	0.024096	0.034884	0.043478
5120	0.026316	0.024096	0.034884	0.043478
10240	0.039474	0.036145	0.058140	0.076087

Table A.4: Precision Results for Breakdancer Max on data-set 5 (inversions only).

Threshold	**Coverage**			
(bp)	**20x**	**30x**	**40x**	**50x**
20	0.019608	0.018519	0.017544	0.000000
40	0.156863	0.074074	0.052632	0.050000
80	0.313725	0.296296	0.263158	0.233333
160	0.392157	0.370370	0.350877	0.333333
320	0.392157	0.370370	0.350877	0.333333
640	0.392157	0.370370	0.350877	0.333333
1280	0.392157	0.370370	0.350877	0.333333
2560	0.392157	0.370370	0.350877	0.333333
5120	0.392157	0.370370	0.350877	0.333333
10240	0.392157	0.370370	0.350877	0.333333

Table A.5: Recall Results for Breakdancer Max on Data-set 1 (All Variants).

Threshold	**Coverage**			
(bp)	**20x**	**30x**	**40x**	**50x**
20	0.022727	0.011364	0.022727	0.022727
40	0.068182	0.068182	0.068182	0.056818
80	0.125000	0.147727	0.147727	0.147727
160	0.125000	0.147727	0.147727	0.147727
320	0.136364	0.159091	0.159091	0.159091
640	0.181818	0.204545	0.215909	0.215909
1280	0.204545	0.227273	0.238636	0.238636
2560	0.227273	0.261364	0.272727	0.272727
5120	0.284091	0.318182	0.340909	0.352273
10240	0.329545	0.363636	0.386364	0.409091

Table A.6: Recall Results for Breakdancer Max on Data-set 2 (deletions only).

Threshold (bp)	**Coverage**			
	20x	**30x**	**40x**	**50x**
20	0.000000	0.000000	0.000000	0.000000
40	0.090909	0.000000	0.000000	0.000000
80	0.227273	0.227273	0.227273	0.227273
160	0.227273	0.227273	0.227273	0.227273
320	0.227273	0.227273	0.227273	0.227273
640	0.227273	0.227273	0.227273	0.227273
1280	0.227273	0.227273	0.227273	0.227273
2560	0.318182	0.318182	0.318182	0.318182
5120	0.318182	0.318182	0.318182	0.318182
10240	0.454545	0.454545	0.454545	0.454545

Table A.7: Recall Results for Breakdancer Max on data-set 4 (insertions only).

Threshold (bp)	**Coverage**			
	20x	**30x**	**40x**	**50x**
20	0.000000	0.000000	0.000000	0.000000
40	0.000000	0.000000	0.000000	0.000000
80	0.045455	0.045455	0.045455	0.090909
160	0.045455	0.045455	0.045455	0.090909
320	0.045455	0.045455	0.045455	0.090909
640	0.090909	0.090909	0.090909	0.136364
1280	0.090909	0.090909	0.090909	0.136364
2560	0.090909	0.090909	0.136364	0.181818
5120	0.090909	0.090909	0.136364	0.181818
10240	0.136364	0.136364	0.227273	0.318182

Table A.8: Recall Results for Breakdancer Max on data-set 5 (inversions only).

Threshold	**Coverage**			
(bp)	**20x**	**30x**	**40x**	**50x**
20	0.045455	0.045455	0.045455	0.000000
40	0.363636	0.181818	0.136364	0.136364
80	0.727273	0.727273	0.681818	0.636364
160	0.909091	0.909091	0.909091	0.909091
320	0.909091	0.909091	0.909091	0.909091
640	0.909091	0.909091	0.909091	0.909091
1280	0.909091	0.909091	0.909091	0.909091
2560	0.909091	0.909091	0.909091	0.909091
5120	0.909091	0.909091	0.909091	0.909091
10240	0.909091	0.909091	0.909091	0.909091

Table A.9: Precision Results for CNVNator on Data-set 1 (All Variants).

Threshold	**Coverage**			
(bp)	**20x**	**30x**	**40x**	**50x**
20	0.0	0.000000	0.000000	0.0
40	0.0	0.000000	0.000000	0.0
80	0.2	0.166667	0.166667	0.2
160	0.4	0.333333	0.333333	0.4
320	0.4	0.333333	0.333333	0.4
640	0.8	0.666667	0.666667	0.8
1280	0.8	0.666667	0.666667	0.8
2560	0.8	0.666667	0.666667	0.8
5120	0.8	0.666667	0.666667	0.8
10240	1.0	1.000000	1.000000	1.0

Table A.10: Precision Results for CNVNator on Data-set 2 (deletions only).

Threshold	**Coverage**			
(bp)	**20x**	**30x**	**40x**	**50x**
20	0.0	0.000000	0.000000	0.00
40	0.0	0.000000	0.000000	0.00
80	0.0	0.000000	0.000000	0.00
160	0.0	0.000000	0.000000	0.00
320	0.0	0.000000	0.000000	0.00
640	0.0	0.000000	0.000000	0.00
1280	0.5	0.333333	0.333333	0.25
2560	0.5	0.333333	0.333333	0.25
5120	0.5	0.333333	0.333333	0.25
10240	0.5	0.333333	0.333333	0.50

Table A.11: Precision Results for CNVNator on data-set 3 (duplications only).

Threshold	**Coverage**			
(bp)	**20x**	**30x**	**40x**	**50x**
20	0	0	0	0
40	0	0	0	0
80	0	0	0	0
160	0	0	0	0
320	0	0	0	0
640	0	0	0	0
1280	0	0	0	0
2560	0	0	0	0
5120	0	0	0	0
10240	0	0	0	0

Table A.12: Recall Results for CNVNator on Data-set 1 (All Variants).

Threshold	**Coverage**			
(bp)	**20x**	**30x**	**40x**	**50x**
20	0.000000	0.000000	0.000000	0.000000
40	0.000000	0.000000	0.000000	0.000000
80	0.011364	0.011364	0.011364	0.011364
160	0.022727	0.022727	0.022727	0.022727
320	0.022727	0.022727	0.022727	0.022727
640	0.045455	0.045455	0.045455	0.045455
1280	0.045455	0.045455	0.045455	0.045455
2560	0.045455	0.045455	0.045455	0.045455
5120	0.045455	0.045455	0.045455	0.045455
10240	0.056818	0.056818	0.056818	0.045455

Table A.13: Recall Results for CNVNator on Data-set 2 (deletions only).

Threshold	**Coverage**			
(bp)	**20x**	**30x**	**40x**	**50x**
20	0.000000	0.000000	0.000000	0.000000
40	0.000000	0.000000	0.000000	0.000000
80	0.000000	0.000000	0.000000	0.000000
160	0.000000	0.000000	0.000000	0.000000
320	0.000000	0.000000	0.000000	0.000000
640	0.000000	0.000000	0.000000	0.000000
1280	0.045455	0.045455	0.045455	0.045455
2560	0.045455	0.045455	0.045455	0.045455
5120	0.045455	0.045455	0.045455	0.045455
10240	0.045455	0.045455	0.045455	0.090909

Table A.14: Recall Results for CNVNator on data-set 3 (duplications only).

Threshold	**Coverage**			
(bp)	**20x**	**30x**	**40x**	**50x**
20	0	0	0	0
40	0	0	0	0
80	0	0	0	0
160	0	0	0	0
320	0	0	0	0
640	0	0	0	0
1280	0	0	0	0
2560	0	0	0	0
5120	0	0	0	0
10240	0	0	0	0

A.3 Gustaf

Tables A.15 - A.18 show the precision values discussed in Chapter 3 and Tables A.19 - A.22 show supplemental recall values for reference only. Gustaf does not call insertions; therefore, results for data-set 4 (insertions only) are omitted.

Table A.15: Precision Results for Gustaf on Data-set 1 (All Variants).

Threshold	**Coverage**			
(bp)	**20x**	**30x**	**40x**	**50x**
20	0.039568	0.033033	0.028871	0.026066
40	0.043636	0.036364	0.031746	0.028640
80	0.043956	0.037037	0.032258	0.029268
160	0.045802	0.039088	0.034483	0.031169
320	0.046332	0.039604	0.035191	0.031662
640	0.084746	0.073801	0.066007	0.059880
1280	0.093333	0.080769	0.072664	0.066456
2560	0.126263	0.109170	0.098039	0.088652
5120	0.200000	0.175000	0.160920	0.147368
10240	0.326087	0.300000	0.280374	0.263158

A.4 LUMPY

Tables A.23 - A.26 show the precision values discussed in Chapter 3 and Tables A.27 - A.30 show supplemental recall values for reference only. LUMPY does not call insertions; therefore, results for data-set 4 (insertions only) are omitted.

Table A.16: Precision Results for Gustaf on Data-set 2 (deletions only).

Threshold	**Coverage**			
(bp)	**20x**	**30x**	**40x**	**50x**
20	0.192308	0.156250	0.138889	0.119048
40	0.192308	0.156250	0.138889	0.119048
80	0.200000	0.161290	0.142857	0.121951
160	0.208333	0.166667	0.147059	0.128205
320	0.208333	0.166667	0.147059	0.128205
640	0.208333	0.166667	0.147059	0.128205
1280	0.208333	0.166667	0.147059	0.128205
2560	0.291667	0.233333	0.205882	0.184211
5120	0.318182	0.269231	0.233333	0.212121
10240	0.476190	0.400000	0.344828	0.312500

Table A.17: Precision Results for Gustaf on data-set 3 (duplications only).

Threshold	**Coverage**			
(bp)	**20x**	**30x**	**40x**	**50x**
20	0.073171	0.055046	0.046154	0.040000
40	0.073171	0.055046	0.046154	0.040000
80	0.075000	0.055556	0.047619	0.041958
160	0.075000	0.056075	0.048387	0.042254
320	0.075949	0.056604	0.049587	0.043165
640	0.078947	0.059406	0.052174	0.045113
1280	0.082192	0.062500	0.055556	0.048387
2560	0.082192	0.062500	0.055556	0.048387
5120	0.082192	0.062500	0.055556	0.048387
10240	0.082192	0.062500	0.055556	0.048387

Table A.18: Precision Results for Gustaf on data-set 5 (inversions only).

Threshold	**Coverage**			
(bp)	**20x**	**30x**	**40x**	**50x**
20	0.606061	0.512821	0.350877	0.312500
40	0.606061	0.512821	0.350877	0.312500
80	0.625000	0.526316	0.370370	0.333333
160	0.714286	0.625000	0.487805	0.444444
320	0.714286	0.645161	0.526316	0.476190
640	0.769231	0.689655	0.606061	0.588235
1280	0.833333	0.800000	0.769231	0.769231
2560	0.869565	0.833333	0.800000	0.800000
5120	0.909091	0.909091	0.869565	0.869565
10240	0.952381	0.952381	0.909091	0.909091

Table A.19: Recall Results for Gustaf on Data-set 1 (All Variants).

Threshold (bp)	**Coverage**			
	20x	**30x**	**40x**	**50x**
20	0.125000	0.125000	0.125000	0.125000
40	0.136364	0.136364	0.136364	0.136364
80	0.136364	0.136364	0.136364	0.136364
160	0.136364	0.136364	0.136364	0.136364
320	0.136364	0.136364	0.136364	0.136364
640	0.227273	0.227273	0.227273	0.227273
1280	0.238636	0.238636	0.238636	0.238636
2560	0.284091	0.284091	0.284091	0.284091
5120	0.318182	0.318182	0.318182	0.318182
10240	0.340909	0.340909	0.340909	0.340909

Table A.20: Recall Results for Gustaf on Data-set 2 (deletions only).

Threshold (bp)	**Coverage**			
	20x	**30x**	**40x**	**50x**
20	0.227273	0.227273	0.227273	0.227273
40	0.227273	0.227273	0.227273	0.227273
80	0.227273	0.227273	0.227273	0.227273
160	0.227273	0.227273	0.227273	0.227273
320	0.227273	0.227273	0.227273	0.227273
640	0.227273	0.227273	0.227273	0.227273
1280	0.227273	0.227273	0.227273	0.227273
2560	0.318182	0.318182	0.318182	0.318182
5120	0.318182	0.318182	0.318182	0.318182
10240	0.454545	0.454545	0.454545	0.454545

Table A.21: Recall Results for Gustaf on data-set 3 (duplications only).

Threshold	**Coverage**			
(bp)	**20x**	**30x**	**40x**	**50x**
20	0.272727	0.272727	0.272727	0.272727
40	0.272727	0.272727	0.272727	0.272727
80	0.272727	0.272727	0.272727	0.272727
160	0.272727	0.272727	0.272727	0.272727
320	0.272727	0.272727	0.272727	0.272727
640	0.272727	0.272727	0.272727	0.272727
1280	0.272727	0.272727	0.272727	0.272727
2560	0.272727	0.272727	0.272727	0.272727
5120	0.272727	0.272727	0.272727	0.272727
10240	0.272727	0.272727	0.272727	0.272727

Table A.22: Recall Results for Gustaf on data-set 5 (inversions only).

Threshold	**Coverage**			
(bp)	**20x**	**30x**	**40x**	**50x**
20	0.909091	0.909091	0.909091	0.909091
40	0.909091	0.909091	0.909091	0.909091
80	0.909091	0.909091	0.909091	0.909091
160	0.909091	0.909091	0.909091	0.909091
320	0.909091	0.909091	0.909091	0.909091
640	0.909091	0.909091	0.909091	0.909091
1280	0.909091	0.909091	0.909091	0.909091
2560	0.909091	0.909091	0.909091	0.909091
5120	0.909091	0.909091	0.909091	0.909091
10240	0.909091	0.909091	0.909091	0.909091

Table A.23: Precision Results for LUMPY on Data-set 1 (All Variants).

Threshold	**Coverage**			
(bp)	**20x**	**30x**	**40x**	**50x**
20	0.096774	0.096774	0.096774	0.096774
40	0.098361	0.098361	0.098361	0.098361
80	0.098361	0.098361	0.098361	0.098361
160	0.099174	0.099174	0.099174	0.099174
320	0.100000	0.100000	0.100000	0.100000
640	0.187500	0.187500	0.187500	0.187500
1280	0.203704	0.203704	0.203704	0.203704
2560	0.267327	0.267327	0.267327	0.267327
5120	0.371795	0.371795	0.371795	0.371795
10240	0.596154	0.596154	0.596154	0.596154

A.5 SVDetect

Tables A.31 - A.35 show the precision values discussed in Chapter 3 and Tables A.36 - A.40 show supplemental recall values for reference only. SVDetect can call all types of variants benchmarked in these experiments.

A.6 Execution Times

This section displays the execution times for all tools on the five data-sets discussed in Chapter 3. In the following tables, Breakdancer Max was abbreviated to BDM and

Table A.24: Precision Results for LUMPY on Data-set 2 (deletions only).

Threshold (bp)	**Coverage**			
	20x	**30x**	**40x**	**50x**
20	0.238095	0.238095	0.238095	0.238095
40	0.238095	0.238095	0.238095	0.238095
80	0.238095	0.238095	0.238095	0.238095
160	0.238095	0.238095	0.238095	0.238095
320	0.238095	0.238095	0.238095	0.238095
640	0.238095	0.238095	0.238095	0.238095
1280	0.238095	0.238095	0.238095	0.238095
2560	0.333333	0.333333	0.333333	0.333333
5120	0.350000	0.350000	0.350000	0.350000
10240	0.500000	0.500000	0.500000	0.500000

Table A.25: Precision Results for LUMPY on data-set 3 (duplications only).

Threshold	**Coverage**			
(bp)	**20x**	**30x**	**40x**	**50x**
20	0.190476	0.181818	0.190476	0.181818
40	0.190476	0.181818	0.190476	0.181818
80	0.190476	0.181818	0.190476	0.181818
160	0.190476	0.181818	0.190476	0.181818
320	0.190476	0.181818	0.190476	0.181818
640	0.190476	0.181818	0.190476	0.181818
1280	0.190476	0.181818	0.190476	0.181818
2560	0.190476	0.181818	0.190476	0.181818
5120	0.190476	0.181818	0.190476	0.181818
10240	0.190476	0.181818	0.190476	0.181818

Table A.26: Precision Results for LUMPY on data-set 5 (inversions only).

Threshold (bp)	**Coverage**			
	20x	**30x**	**40x**	**50x**
20	0.863636	0.863636	0.863636	0.863636
40	0.863636	0.863636	0.863636	0.863636
80	0.863636	0.863636	0.863636	0.863636
160	0.863636	0.863636	0.863636	0.863636
320	0.863636	0.863636	0.863636	0.863636
640	0.904762	0.904762	0.904762	0.904762
1280	0.904762	0.904762	0.904762	0.904762
2560	0.904762	0.904762	0.904762	0.904762
5120	0.904762	0.904762	0.904762	0.904762
10240	0.950000	0.950000	0.950000	0.950000

Table A.27: Recall Results for LUMPY on Data-set 1 (All Variants).

Threshold (bp)	**Coverage**			
	20x	**30x**	**40x**	**50x**
20	0.136364	0.136364	0.136364	0.136364
40	0.136364	0.136364	0.136364	0.136364
80	0.136364	0.136364	0.136364	0.136364
160	0.136364	0.136364	0.136364	0.136364
320	0.136364	0.136364	0.136364	0.136364
640	0.238636	0.238636	0.238636	0.238636
1280	0.250000	0.250000	0.250000	0.250000
2560	0.306818	0.306818	0.306818	0.306818
5120	0.329545	0.329545	0.329545	0.329545
10240	0.352273	0.352273	0.352273	0.352273

Table A.28: Recall Results for LUMPY on Data-set 2 (deletions only).

Threshold (bp)	**Coverage**			
	20x	**30x**	**40x**	**50x**
20	0.227273	0.227273	0.227273	0.227273
40	0.227273	0.227273	0.227273	0.227273
80	0.227273	0.227273	0.227273	0.227273
160	0.227273	0.227273	0.227273	0.227273
320	0.227273	0.227273	0.227273	0.227273
640	0.227273	0.227273	0.227273	0.227273
1280	0.227273	0.227273	0.227273	0.227273
2560	0.318182	0.318182	0.318182	0.318182
5120	0.318182	0.318182	0.318182	0.318182
10240	0.454545	0.454545	0.454545	0.454545

Table A.29: Recall Results for LUMPY on data-set 3 (duplications only).

Threshold	**Coverage**			
(bp)	**20x**	**30x**	**40x**	**50x**
20	0.181818	0.181818	0.181818	0.181818
40	0.181818	0.181818	0.181818	0.181818
80	0.181818	0.181818	0.181818	0.181818
160	0.181818	0.181818	0.181818	0.181818
320	0.181818	0.181818	0.181818	0.181818
640	0.181818	0.181818	0.181818	0.181818
1280	0.181818	0.181818	0.181818	0.181818
2560	0.181818	0.181818	0.181818	0.181818
5120	0.181818	0.181818	0.181818	0.181818
10240	0.181818	0.181818	0.181818	0.181818

Table A.30: Recall Results for LUMPY on data-set 5 (inversions only).

Threshold	**Coverage**			
(bp)	**20x**	**30x**	**40x**	**50x**
20	0.863636	0.863636	0.863636	0.863636
40	0.863636	0.863636	0.863636	0.863636
80	0.863636	0.863636	0.863636	0.863636
160	0.863636	0.863636	0.863636	0.863636
320	0.863636	0.863636	0.863636	0.863636
640	0.863636	0.863636	0.863636	0.863636
1280	0.863636	0.863636	0.863636	0.863636
2560	0.863636	0.863636	0.863636	0.863636
5120	0.863636	0.863636	0.863636	0.863636
10240	0.863636	0.863636	0.863636	0.863636

Table A.31: Precision Results for SVDetect on Data-set 1 (All Variants).

Threshold (bp)	**Coverage**			
	20x	**30x**	**40x**	**50x**
20	0.034483	0.034884	0.034884	0.034483
40	0.034483	0.034884	0.034884	0.034483
80	0.034483	0.034884	0.034884	0.034483
160	0.103448	0.116279	0.104651	0.080460
320	0.114943	0.139535	0.139535	0.137931
640	0.195402	0.220930	0.220930	0.218391
1280	0.195402	0.220930	0.220930	0.218391
2560	0.252874	0.279070	0.267442	0.275862
5120	0.264368	0.302326	0.279070	0.298851
10240	0.287356	0.337209	0.313953	0.321839

Table A.32: Precision Results for SVDetect on Data-set 2 (deletions only).

Threshold (bp)	**Coverage**			
	20x	**30x**	**40x**	**50x**
20	0.000000	0.000000	0.000000	0.000000
40	0.000000	0.000000	0.000000	0.000000
80	0.000000	0.000000	0.000000	0.000000
160	0.263158	0.263158	0.263158	0.263158
320	0.263158	0.263158	0.263158	0.263158
640	0.263158	0.263158	0.263158	0.263158
1280	0.263158	0.263158	0.263158	0.263158
2560	0.368421	0.368421	0.368421	0.368421
5120	0.368421	0.368421	0.368421	0.368421
10240	0.526316	0.526316	0.526316	0.526316

Table A.33: Precision Results for SVDetect on data-set 3 (duplications only).

Threshold	**Coverage**			
(bp)	**20x**	**30x**	**40x**	**50x**
20	0.095238	0.047619	0.086957	0.086957
40	0.095238	0.047619	0.086957	0.086957
80	0.095238	0.047619	0.086957	0.086957
160	0.095238	0.047619	0.086957	0.086957
320	0.095238	0.047619	0.086957	0.086957
640	0.095238	0.047619	0.086957	0.086957
1280	0.095238	0.047619	0.086957	0.086957
2560	0.095238	0.047619	0.086957	0.086957
5120	0.095238	0.047619	0.086957	0.086957
10240	0.095238	0.047619	0.086957	0.086957

Table A.34: Precision Results for SVDetect on data-set 4 (insertions only).

Threshold	**Coverage**			
(bp)	**20x**	**30x**	**40x**	**50x**
20	0.000000	0.000000	0.000000	0.000000
40	0.000000	0.000000	0.000000	0.000000
80	0.000000	0.000000	0.000000	0.000000
160	0.023256	0.046512	0.047619	0.046512
320	0.023256	0.046512	0.047619	0.046512
640	0.023256	0.046512	0.047619	0.046512
1280	0.023256	0.046512	0.047619	0.046512
2560	0.023256	0.046512	0.047619	0.046512
5120	0.023256	0.046512	0.047619	0.046512
10240	0.069767	0.093023	0.095238	0.093023

Table A.35: Precision Results for SVDetect on data-set 5 (inversions only).

Threshold	**Coverage**			
(bp)	**20x**	**30x**	**40x**	**50x**
20	0.000000	0.000000	0.000000	0.000000
40	0.000000	0.000000	0.000000	0.000000
80	0.000000	0.000000	0.000000	0.000000
160	0.764706	0.764706	0.764706	0.764706
320	0.823529	0.823529	0.823529	0.823529
640	0.823529	0.823529	0.823529	0.823529
1280	0.823529	0.823529	0.823529	0.823529
2560	0.882353	0.882353	0.882353	0.882353
5120	0.882353	0.882353	0.882353	0.882353
10240	0.882353	0.882353	0.882353	0.882353

Table A.36: Recall Results for SVDetect on Data-set 1 (All Variants).

Threshold	**Coverage**			
(bp)	**20x**	**30x**	**40x**	**50x**
20	0.034091	0.034091	0.034091	0.034091
40	0.034091	0.034091	0.034091	0.034091
80	0.034091	0.034091	0.034091	0.034091
160	0.102273	0.113636	0.102273	0.079545
320	0.113636	0.136364	0.136364	0.136364
640	0.193182	0.215909	0.215909	0.215909
1280	0.193182	0.215909	0.215909	0.215909
2560	0.250000	0.272727	0.261364	0.272727
5120	0.261364	0.295455	0.272727	0.295455
10240	0.284091	0.329545	0.306818	0.318182

Table A.37: Recall Results for SVDetect on Data-set 2 (deletions only).

Threshold	**Coverage**			
(bp)	**20x**	**30x**	**40x**	**50x**
20	0.000000	0.000000	0.000000	0.000000
40	0.000000	0.000000	0.000000	0.000000
80	0.000000	0.000000	0.000000	0.000000
160	0.227273	0.227273	0.227273	0.227273
320	0.227273	0.227273	0.227273	0.227273
640	0.227273	0.227273	0.227273	0.227273
1280	0.227273	0.227273	0.227273	0.227273
2560	0.318182	0.318182	0.318182	0.318182
5120	0.318182	0.318182	0.318182	0.318182
10240	0.454545	0.454545	0.454545	0.454545

Table A.38: Recall Results for SVDetect on data-set 3 (duplications only).

Threshold	**Coverage**			
(bp)	**20x**	**30x**	**40x**	**50x**
20	0.090909	0.045455	0.090909	0.090909
40	0.090909	0.045455	0.090909	0.090909
80	0.090909	0.045455	0.090909	0.090909
160	0.090909	0.045455	0.090909	0.090909
320	0.090909	0.045455	0.090909	0.090909
640	0.090909	0.045455	0.090909	0.090909
1280	0.090909	0.045455	0.090909	0.090909
2560	0.090909	0.045455	0.090909	0.090909
5120	0.090909	0.045455	0.090909	0.090909
10240	0.090909	0.045455	0.090909	0.090909

Table A.39: Recall Results for SVDetect on data-set 4 (insertions only).

Threshold	**Coverage**			
(bp)	**20x**	**30x**	**40x**	**50x**
20	0.000000	0.000000	0.000000	0.000000
40	0.000000	0.000000	0.000000	0.000000
80	0.000000	0.000000	0.000000	0.000000
160	0.045455	0.090909	0.090909	0.090909
320	0.045455	0.090909	0.090909	0.090909
640	0.045455	0.090909	0.090909	0.090909
1280	0.045455	0.090909	0.090909	0.090909
2560	0.045455	0.090909	0.090909	0.090909
5120	0.045455	0.090909	0.090909	0.090909
10240	0.136364	0.181818	0.181818	0.181818

Table A.40: Recall Results for SVDetect on data-set 5 (inversions only).

Threshold	**Coverage**			
(bp)	**20x**	**30x**	**40x**	**50x**
20	0.000000	0.000000	0.000000	0.000000
40	0.000000	0.000000	0.000000	0.000000
80	0.000000	0.000000	0.000000	0.000000
160	0.590909	0.590909	0.590909	0.590909
320	0.636364	0.636364	0.636364	0.636364
640	0.636364	0.636364	0.636364	0.636364
1280	0.636364	0.636364	0.636364	0.636364
2560	0.681818	0.681818	0.681818	0.681818
5120	0.681818	0.681818	0.681818	0.681818
10240	0.681818	0.681818	0.681818	0.681818

CNVNator to CNVN for compactness.

Table A.41: Execution times (in seconds) for All Tools on Data-set 1 (All Variants).

Coverage	**Tool**				
	BDM	**CNVN**	**Gustaf**	**LUMPY**	**SVDetect**
20x	0.616	5.394	1770.992	1.728	5.102
30x	0.909	5.582	3479.103	2.523	8.434
40x	1.199	5.762	6230.691	3.296	13.064
50x	1.490	5.819	8315.805	4.321	16.968
Mean	1.000	5.636	4226.988	3.163	9.882
Std. Dev.	0.375	0.192	2900.484	1.106	5.202

Table A.42: Execution times (in seconds) for All Tools on Data-set 2 (deletions only).

Coverage	Tool				
	BDM	**CNVN**	**Gustaf**	**LUMPY**	**SVDetect**
20x	0.523	5.206	1383.104	1.429	4.571
30x	0.769	5.491	2562.825	2.097	6.837
40x	1.014	5.556	4284.811	2.678	9.659
50x	1.259	5.745	6332.868	3.097	11.266
Mean	0.846	5.496	3131.673	2.576	7.636
Std. Dev.	0.316	0.223	2154.158	0.724	2.972

Table A.43: Execution times (in seconds) for All Tools on Data-set 3 (duplications only).

Coverage	Tool				
	BDM	**CNVN**	**Gustaf**	**LUMPY**	**SVDetect**
20x	†	5.362	1692.933	1.665	5.503
30x	†	5.634	3415.278	2.447	8.593
40x	†	5.759	6163.518	3.284	11.826
50x	†	5.937	8129.319	4.040	15.134
Mean	†	5.669	4125.600	3.123	9.591
Std. Dev.	†	0.241	2858.015	1.028	4.147

† not applicable

Table A.44: Execution times (in seconds) for All Tools on Data-set 4 (insertions only).

Coverage	**Tool**				
	BDM	**CNVN**	**Gustaf**	**LUMPY**	**SVDetect**
20x	0.535	†	†	†	4.800
30x	0.787	†	†	†	7.179
40x	1.042	†	†	†	9.554
50x	1.289	†	†	†	11.898
Mean	0.867	†	†	†	7.911
Std. Dev.	0.324	†	†	†	3.055

† not applicable

Table A.45: Execution times (in seconds) for All Tools on Data-set 5 (inversions only).

Coverage	**Tool**				
	BDM	**CNVN**	**Gustaf**	**LUMPY**	**SVDetect**
20x	0.534	†	1476.086	1.394	4.482
30x	0.786	†	2940.812	2.065	6.560
40x	1.037	†	4212.602	2.587	8.728
50x	1.288	†	6295.830	3.313	10.839
Mean	0.865	†	3275.635	2.562	7.262
Std. Dev.	0.324	†	2042.818	0.812	2.742

† not applicable

Appendix B

NEMOCLUSTER EXPERIMENTAL DATA

Shown below are visualizations of the Receiver Operating Characteristic (ROC) curves that were not shown in Chapter 6. This is included for reference only and will not be discussed here as it was already addressed in Chapter 6.

Fig. B.1 through B.6 present the RoC curves for NemoCluster and Tectonic. This type of curve shows the true-positive rate (TPR), or recall, as a function of the false-positive rate (FPR). Ideally, a program should never drop below the central diagonal (the line $y = x$). Above the central diagonal, the TPR is higher than the FPR, meaning that a program makes more correct decisions than incorrect ones.

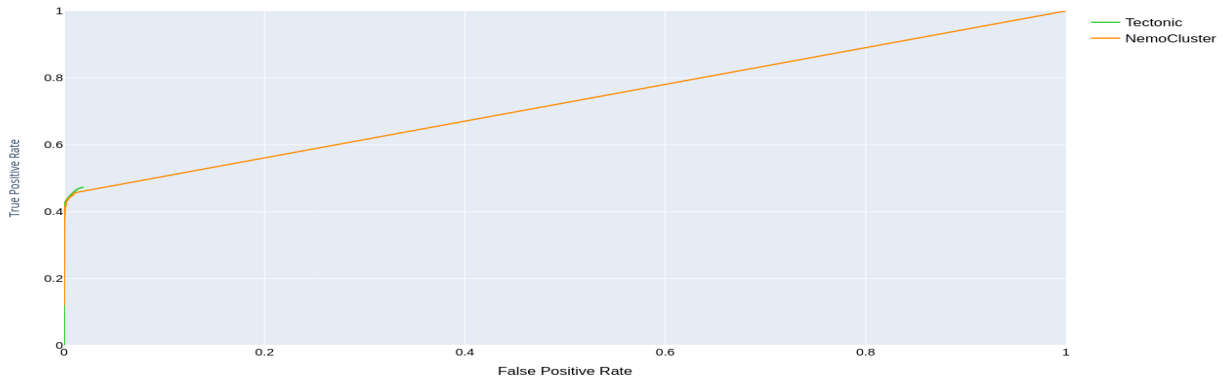

Figure B.1: ROC Curve of Tectonic and NemoCluster on the Amazon network from [26].

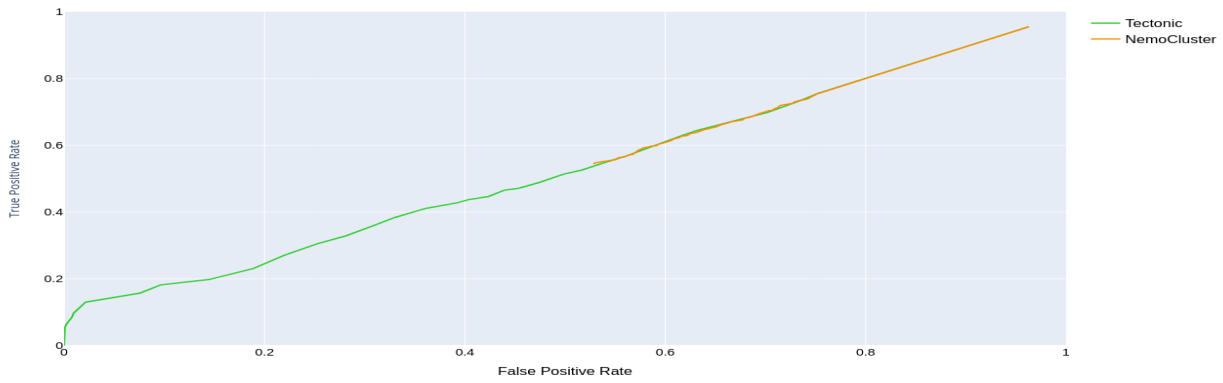

Figure B.2: ROC Curve of Tectonic and NemoCluster on the Email core network from [26].

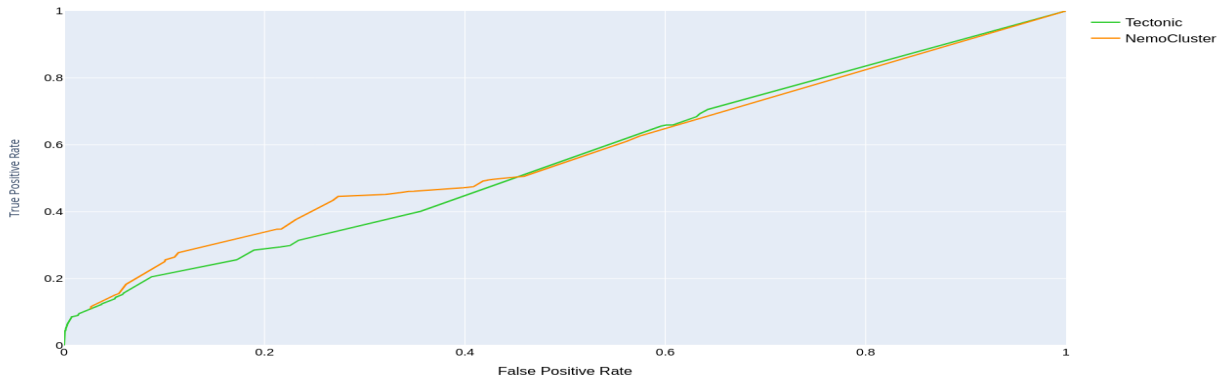

Figure B.3: ROC Curve of Tectonic and NemoCluster on the Protein network from [38].

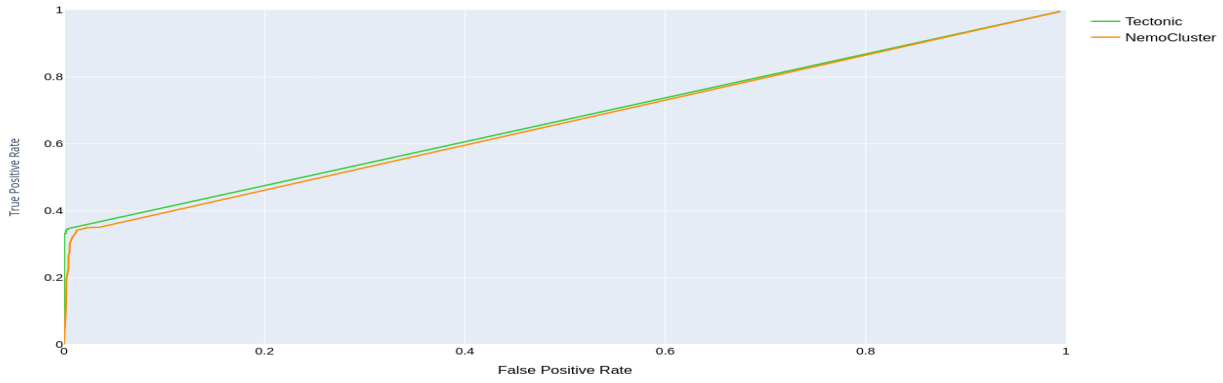

Figure B.4: ROC Curve of Tectonic and NemoCluster on the 250 Vertex synthetic networks (average over 10 networks).

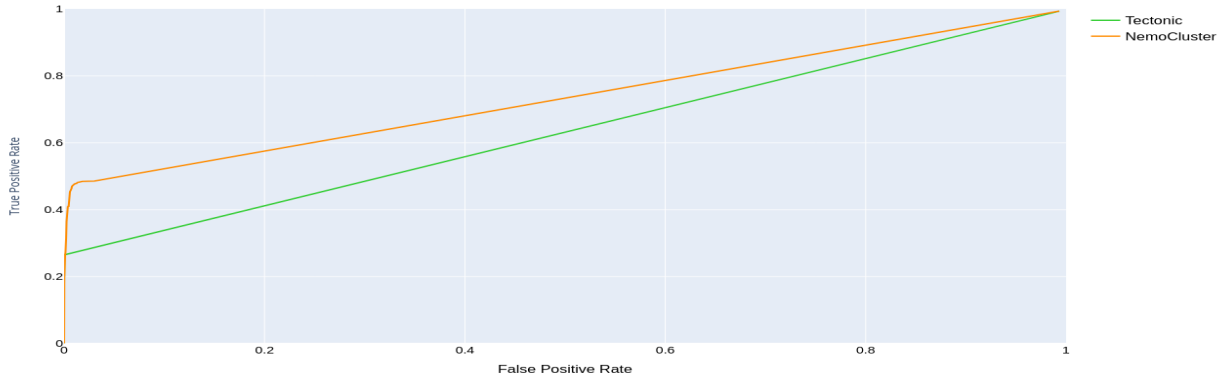

Figure B.5: ROC Curve of Tectonic and NemoCluster on the 500 Vertex synthetic networks (average over 10 networks).

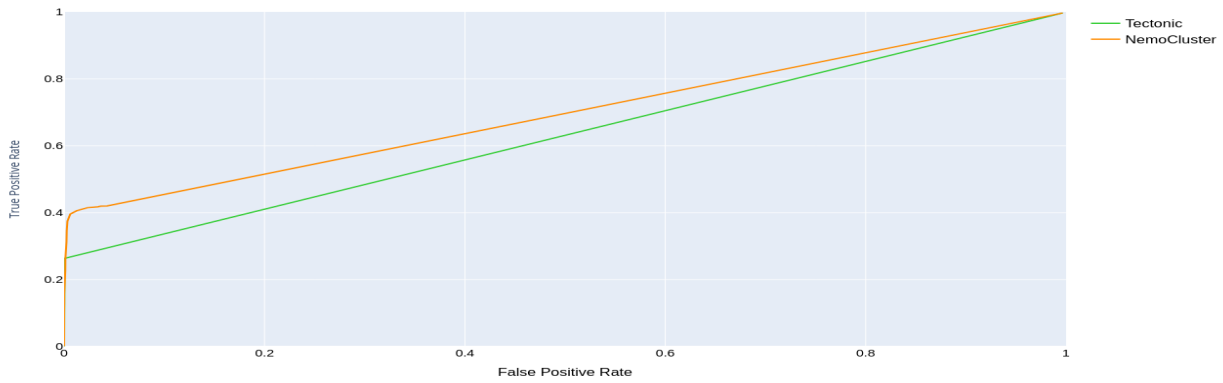

Figure B.6: ROC Curve of Tectonic and NemoCluster on the 1000 Vertex synthetic networks (average over 10 networks).