Automated Identification and Visualisation of Sequence Rearrangements in Comparative Genomics Using Subgraph Isomorphism

Oliver Newport

Student No. 110254425

August 2017

MSc Computer Science

Supervisor: Anil Wipat

Word Count: 13,082

Table of Contents

Abstract

The field of comparative genomics is a rapidly growing area of scientific study, focusing on the comparison between the genomic features of different species. These comparisons are used for many purposes, such as to gain insights into the function of particular genes, or to identify targets for novel drug treatments. Comparative genomics commonly relies on the identification of specific rearrangement features that can be created through the sequence alignment of two DNA sequences. Currently, these rearrangement features must be identified for the most part manually using limited visualisation tools, making it a difficult and time consuming process. This project attempted to implement a system to automatically detect rearrangement features in sequence alignment data by applying subgraph isomorphism algorithms to a graph-based representation of sequence alignments. The system produced developed upon previous work carried out in pursuit of this goal, utilising an implementation of the VF2 subgraph isomorphism algorithm to detect rearrangement motifs in a graph-based representation of BLAST data. The identified rearrangement features are then able to be viewed in a visualisation of the graph-based representation.

Declaration

I declare that this dissertation represents my own work except where otherwise stated.

Acknowledgements

I would like to thank my supervisor for this project, Anil Wipat, for his assistance, as well as Keith Flanagan for his guidance throughout the process. Furthermore, I would like to acknowledge to previous work carried out by Eleni Mamalaki and Natalie Sandford, which was of great benefit for the completion of this project.

Chapter 1: Introduction

Comparative genomics is a branch of genomics in which the genomes of species are compared against one another in order to gain biological insights. These insights can include the evolutionary makeup of a species or the function of specific genes. With the number of species to have had their complete genomes sequenced still growing [1], the field of comparative genomics becomes ever more important, as does developing advances in the techniques incorporated by the field.

Identification of genomic rearrangement features is currently performed manually – a time consuming and difficult process – with there being no current ways of automated identification available. This problem is compounded by the limitations of tools that are currently used to perform this manual identification, with the visualisations they present being cluttered and difficult to analyse, especially when being used with large genomes.

This project seeks to build upon the work carried out by two previous students to develop a system to automatically identify genomic rearrangement features by applying subgraph isomorphism techniques to a graph-based representation of genomic sequence alignments.

1.1 Aim

To use subgraph isomorphism to automatically detect genomic rearrangement features in a graphical representation of sequence alignments, and to design and implement a graphical user interface (GUI) to display and analyse the results.

1.2 Objectives

The overall objectives of this project are to:

1. Implement and potentially improve upon an existing design for the graphical representation of sequence alignments.
2. Implement and potentially improve upon existing designs of subgraph motifs to represent specific genomic rearrangement features.
3. Identify the motifs implemented (2) in the graphical representation implemented in (1) using a subgraph isomorphism matching algorithm.
4. Research existing sequence alignment visualisation tools in order to ascertain their advantage and disadvantages, and the common features expected from this type of tool.
5. Design and implement a GUI to display (1) and the results of (3), along with suitable features identified in (4).

1.3 Project Management

It was determined that the prototyping development model would be most suitable for this project. The reasons behind this were twofold: a large amount of the design work had already been completed and tested, meaning a working implementation could be produced quickly; and the model allows for quick feedback on any implemented features, which is especially useful when designing a GUI for end users.

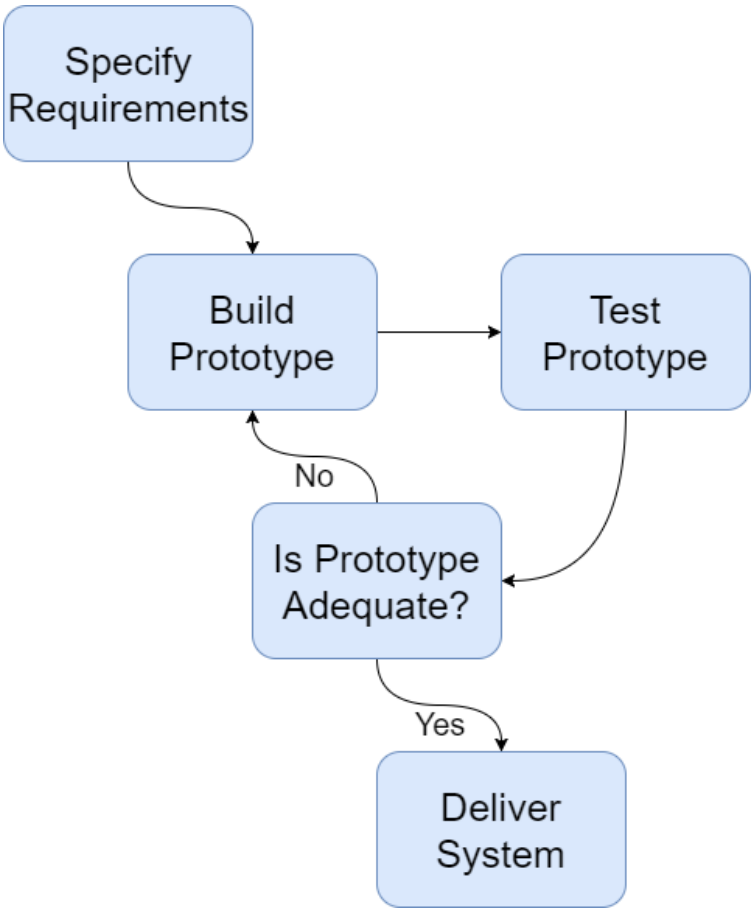


Figure 1.1 The prototyping development model.

Chapter 2: Background Research

2.1 Core Biological Concepts

In order to sufficiently understand the aims of this project some understanding of several key biological concepts are required. What follows is a summary of the concepts deemed relevant to this project.

2.1.1 DNA and Genes

Deoxyribonucleic Acid (DNA) acts as the genetic blueprint for all known forms of life. An entire copy of an organism's DNA is present in almost all of its cells, and is most commonly found in the nucleus of each cell, packaged into structures called chromosomes. A DNA molecule consists of two strands, each strand made up from basic monomer units known as nucleotides. The structure of a nucleotide consist of a single nucleobase (or base) bonded to a deoxyribose sugar and a phosphate group. Each strand in DNA is formed from the sugar of one nucleotide bonding to the phosphate of another, creating a backbone of alternating sugar-phosphate groups.

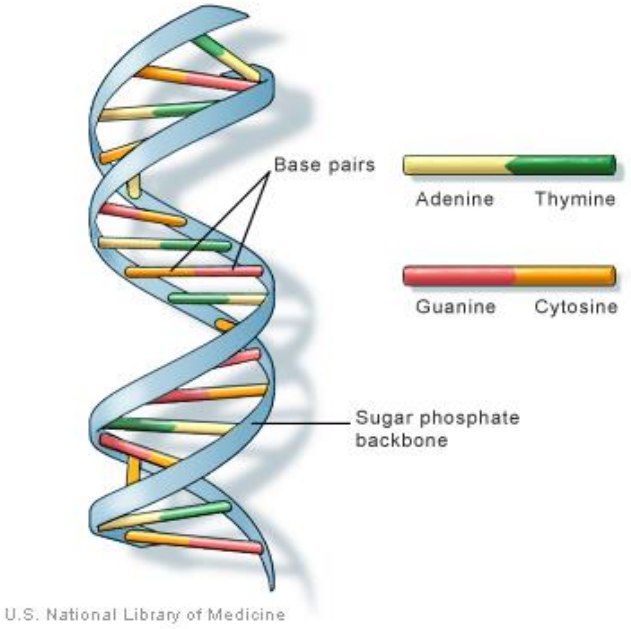


Figure 2.1 The basic ribbon structure of DNA and the base pairs [2].

Two DNA strands are joined together by hydrogen bonding between the bases of each nucleotide. Each base bonds to one other base on the opposite strand, creating base pairs. The strands twist together to form a more stable structure resulting in the well-known double-helix structure of DNA. There are 4 possible bases found in DNA, which are commonly referred to by their initial letter – adenine (A), thymine (T), cytosine (C) and guanine (G). There are certain rules that govern which bases can form pairs; A bonds with T, and C with G. DNA is able to make copies of itself through a process called DNA replication, which is critical during the process of cell division to ensure both cells have a complete copy of the organism's DNA.

DNA acts as a store of biological information through the specific sequence of the bases along its strands. The sequence of bases in DNA is used as the basis to produce messenger ribonucleic acid (mRNA) in a process called transcription. RNA is similar in structure to DNA but is found in single rather than double stranded structures. mRNA is produced from a single strand of DNA by unwinding the double-helix during its production, and therefore contains a sequence complementary to the DNA it was produced from. mRNA is used to synthesise proteins by specifying their amino acid sequence using codons made up of 3 base pairs each. Each three base sequence codes for one specific amino acid (there are potentially multiple codons for each amino acid as there are more possible base combinations than amino acids). This process is called translation.

Regions of DNA that encode for a specific biological function are known as genes. When proteins that are encoded by a gene are being produced that gene is said to be being expressed. Gene expression is a complex process, with expression of genes being controlled by regulatory regions of DNA which are not necessarily found nearby to the region of DNA that will encode for the proteins.

DNA sequences that are expressed to produce a specific protein are called exons. Other regions that are translated into RNA but do not code for proteins are called introns; they are spliced (removed) from the RNA transcript before translation occurs. The first sequencing of the human genome in 2001 [3] revealed that exons make up just 1.1% of the genome, with introns making up a 24% and the remaining 75% consisting of intergenic DNA that has no obvious purpose (although does contain some gene regulatory elements).

2.1.2 Comparative Genomics

In general terms, comparative genomics is a branch of genomics (the study of genomes) that uses the comparison of genomic features to further our biological understanding [4]. The field is particularly useful in identifying genomic features between species and understanding the evolutionary forces and mechanisms that create them. The basic principle of comparative genomics is that sequences that are conserved between species are likely to be constrained (similar in sequence due to evolutionary pressures) and therefore likely to have a biological function. The opposite is not necessarily true however, with DNA sequences that are not conserved between species still able to be functional (likely due to newer novel mutations or genes in a single species not yet being afforded time to proliferate).

DNA features such as exons are conserved between species with a specific pattern, making them simpler to model than regulatory elements [5], which are a particular feature that comparative genomics excels in identifying. For example, it has been used to detect highly conserved non-coding sequences positioned near genes important in development and growth across different breeds of dog [6]. In another case, comparative genomics was used to discover the presence of ultraconserved regions in the human genome [7]. 481 regions over 200 base pairs long were identified and found to be 100% conserved between human, rat and mouse genomes; as well as nearly completely conserved in dog and chicken genomes as well. The regions were found to be located around genes involved in RNA processing and the regulation of transcription and represent a novel class of genetic element.

Comparative genomics can also play an important role in the field of molecular medicine due to its ability to aid in the discovery of novel drug targets for infectious diseases. For example, given two strains of a species of bacterium, where one strain has developed antibiotic resistance and the other has not, comparative genomics can help to identify the causes of this resistance and potentially how it may be overcome. In another example a study by Odds [8] used comparisons of fungal genomes to identify potential targets for novel antifungals.

2.1.3 Genomic Rearrangement Features

There are several common features that can be identified and defined when two genomes are compared. These features can indicate events such as chromosomal rearrangement, where sections of one chromosome may break away and be reincorporated into a different section of the chromosome. In the case of prokaryotes (archaea and bacteria), they may indicate horizontal gene transfer between species (the transfer of genetic material between other nearby microorganisms - as opposed to vertical gene transfer from parent to offspring). The following features are explained in the context of a *subject* sequence, and a *query* sequence which is being compared against the *subject*.

Variation

A region in both the *subject* and *query* sequences where the sequences do not align.

Insertion

The presence of a region in the *subject* sequence that is not present in the *query* sequence, located within a region where both sequences are otherwise found to align. This can be caused by events such as chromosomal translocation, with the region that has broken away, inserting itself into the *subject* sequence.

Deletion

The opposite of an insertion: the presence of a region in the *query* sequence that is not present in the *subject* sequence, located within a region where both sequences are otherwise found to align. Duplications can be caused when a region of a chromosome has broken away and not been reinserted, such as in chromosomal translocation.

Duplication

The duplication of a region of either the *subject* or *query* sequence, such that the region is found multiple times in the opposing sequence. This can occur when a region has been separated from a chromosome, replicated, then reinserted. If the duplicated regions are directly adjacent to one another they are referred to as a tandem duplication. If they are separated it is known as a dispersed duplication.

Inversion

A region that is found to align between the *subject* and *query* sequences, but the matching region on the opposing sequence is reversed. This suggests the region has been separated from the genome and then reinserted in the reverse orientation.

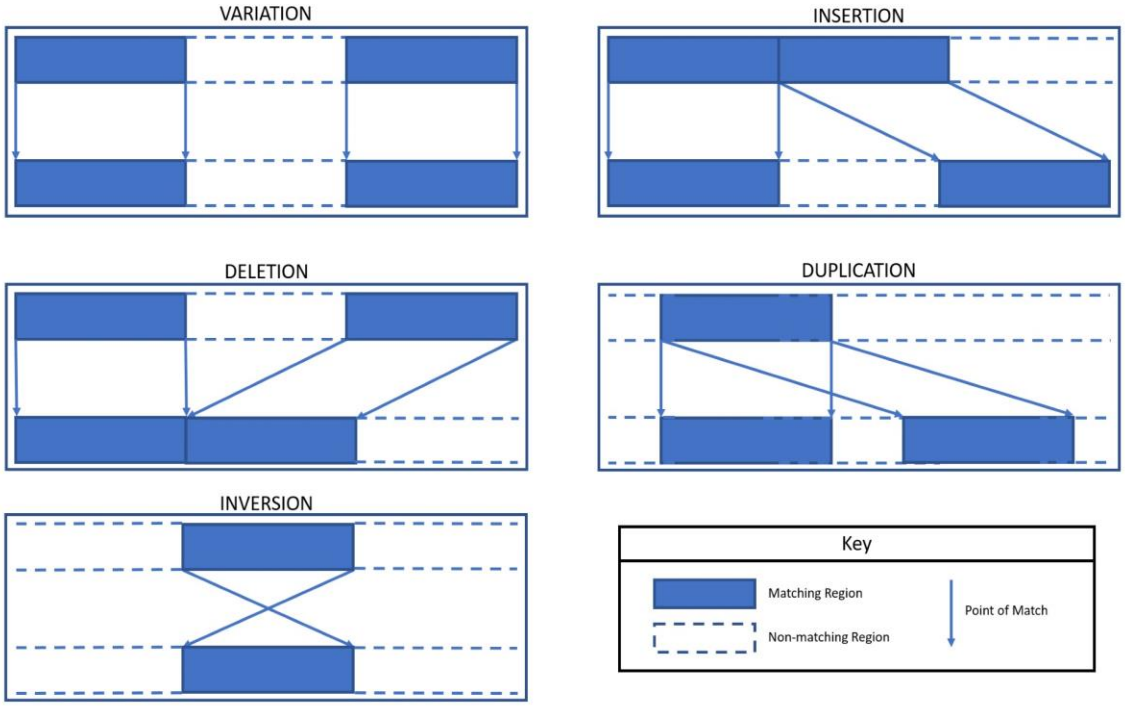


Figure 2.2 Illustrations showing common genomic rearrangement features between two DNA strands. For reference, the subject sequences is positioned below the query sequence. Taken from [9].

2.2 Sequence Alignment

Sequence alignment is a commonly used bioinformatics technique that allows the comparison of DNA sequences in order to determine regions of similarity. It is particularly useful for use in comparative genomics as a means to detect conserved regions of DNA, as well as to identify the rearrangement features described previously.

Sequence alignment has two main methods that can be used to compare sequences:

- Global alignment: most suited to being performed on closely related sequences that are of the same or similar length, this technique attempts to align sequences across the whole of their length, inserting gaps where appropriate to keep the sequences aligned. The Needleman-Wunsch algorithm was developed to perform global alignments in 1970 [10]. The algorithm uses the concept of dynamic programming, which is the breaking down of a complex problem into a set of smaller, less complex problems.
- Local alignment: sequences are compared to detect smaller, or local, regions of similarity across the whole of both sequences. This is the technique that is most relevant to this project, as it is the type of alignment performed by BLAST and allows for the detection of specific rearrangement features. The Smith-Waterman algorithm [11], which is based on the Needleman-Wunsch algorithm, can be used to carry local alignments. Like the Needleman-Wunsch algorithm it uses the concept of dynamic programming.

These alignment methods can be used to perform either pairwise or multiple alignments. Pairwise alignments are simply alignments performed on two distinct sequences, whereas multiple alignments attempt to align three or more sequences simultaneously. Compared to pairwise alignments, multiple alignments are very computationally intensive and hard to visualise. This project will be focusing only on visualising pairwise alignments, with the incorporation of multiple alignments being a possible (though likely significant due to the novel way this project is visualising and detection rearrangements) extension for the future.

2.2.1 Basic Local Alignment Search Tool (BLAST)

BLAST is a bioinformatics tool developed by the National Centre for Biotechnology Information. The tool uses the BLAST algorithm [12] to find local (as opposed to global) regions of similarity between genomic sequences. A user can upload a specific sequence, which is compared against a sequence database and returns matches that have are deemed statistically significant. Users can also upload a subject sequence and one or more query sequences that will be compared against the subject sequence for local alignments.

BLAST has several options that allow users to compare both nucleotide and protein amino acid sequences: blastn for comparing two nucleotide sequences, blastp for comparing two protein amino acid sequences, blastx for comparing a nucleotide query against a protein subject, and tblastn for comparing a protein query against a nucleotide subject. Each of these options can be used to compare a single sequence against database entries or to compare user entered subject and query sequences.

The service be accessed online through Web BLAST, which is hosted by the NCBI; as well as through standalone command-line applications developed by the NCBI and found in the BLAST+ package. NCBI also offers a BLAST API, written in C++, which allows developers to make call BLAST from their own applications.

BLAST Output Format

As the purpose of this project is based upon analysing the output of BLAST queries, understanding its format is required in order to parse output correctly. BLAST delivers its output in both a standard BLAST report as well as a tabular output known as a hit table. The standard report is designed to be human-readable but is subject to change, resulting in the hit table format being more appropriate for the purposes of parsing [13]. Each row of the hit table represents a sequence in both the query and subject sequences that align, and will be referred to as 'matches'.

```
NC_000012.12:55966769-55972789  NC_000076.6:c128705051-128697939   83.961  1141    136 33  3739    4865    4847    5954    0.0         1050
NC_000012.12:55966769-55972789  NC_000076.6:c128705051-128697939   77.647  765     112 41  648     1405    637     1349    7.64e-117   411
NC_000012.12:55966769-55972789  NC_000076.6:c128705051-128697939   80.043  466     68  18  14      465     1       455     3.68e-90    322
NC_000012.12:55966769-55972789  NC_000076.6:c128705051-128697939   88.936  235     24  2   1961    2195    2422    2654    3.73e-80    289
```

Figure 2.3 Example of the format of matches present in the standard BLAST hit table.

The default BLAST hit table, delivered in the .blast file format which will be the standard used for the purposes of this project, consists of 12 columns which are summarized in Table 2.1.

COLUMN	DESCRIPTION
query id	Query sequence identifier
subject ids	Subject sequence identifier
% identity	The percentage of exactly matching bases in the match, before any gaps are inserted to improve the alignment
alignment length	The length of the match
mismatches	The number of bases that do not match
gap opens	The number of gaps added to the match in order to make the sequences align
q. start	The starting position of the match in the query sequence
q. end	The ending position of the match in the query sequence
s. start	The starting position of the match in the subject sequence
s. end	The ending position of the match in the subject sequence
evalue	E-value. A measure of the statistical significance of each match. The lower the value the more significant the match. As an example an E-value of 0.05 indicates a 5 in 100 chance of the match occurring by chance
bit score	Bit-score. A score assigned to each match. Higher score signifies a better match. Calculated from alignment of similar or identical residues along with any gaps added to align the sequences

Table 2.1 Description of each of the columns from the standard BLAST hit table.

2.2.2 Existing Sequence Alignment Visualisation Tools

There are a number of tools currently available that allow users to visualise genome sequence alignments. As this project is focussed on developing the GUI of a new visualisation method it is useful to be familiar with similar existing tools in order to identify common features that would be expected by users of these tools. What follows is a summary of the key features, advantages and disadvantages of three widely used existing visualisation tools [14].

Artemis Comparison Tool

The Artemis Comparison Tool[1] (ACT) is a bioinformatics tool developed by the Sanger Institute. It is based on Artemis, a genome browser and annotation tool. The tool is available as a standalone Java application, with other ACT-related web-based tools also being available: WebACT and DoubleACT. ACT is used as a tool for displaying the pairwise alignment of two or more DNA sequences. The tool is useful for identifying and analysing regions of similarity and difference between genomes, as well as exploring the concept of syntenic regions of chromosomes (areas where the order of genes on a chromosome are conserved between species).

To display alignments, ACT represents the forward and reverse strands of the subject sequence as a bar at the top of the screen. The same is true for the query sequence, which is presented at the bottom of the screen. These are referred to as 'The DNA Views'. The middle of the screen is comprised of the 'Comparison View', where matching sections between the two sequences are joined by red bands. Matching sequences which are inverted are represented by blue bands. The intensity of each band represents the percentage identity (how identical the matching sequences are to one another), with a darker shade indicating a greater identity.

An ACT entry consists of two sequence files alongside a comparison file. ACT allows for several kinds of sequence feature files, including EMBL or GenBank feature tables. The same is true for comparison files, with multiple formats of BLAST output being able to be read by the program as well as MSPcrunch output. Sequence files allow for ACT to annotate the DNA views with additional information such as regions of the sequence that code for RNA or proteins. This allows users to gain a greater insight into the importance of any identified rearrangement features by allowing them to determine whether they are located in a functional region of the DNA sequences.

One significant downside to ACT is that is that other than inversions, it does not automatically identify any specific types of rearrangement features. The user must identify these features manually using the comparison view, which for complex or longer comparisons can be particularly difficult, as the comparison view can become incredibly cluttered and hard to decipher.

[1] http://www.sanger.ac.uk/science/tools/artemis-comparison-tool-act

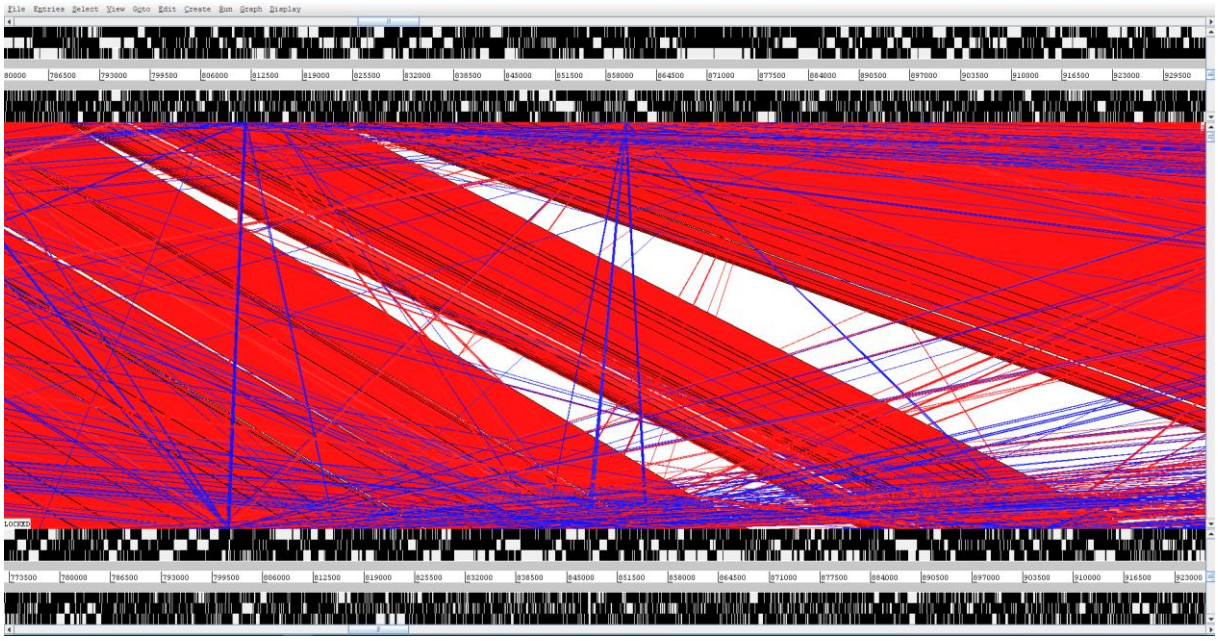


Figure 2.4 Screenshot of an ACT alignment visualisation, demonstrating how with complex alignments the viewer can become cluttered and difficult the read.

Mauve

Mauve[2] is a tool for multiple genome alignments developed by the Darling Lab. Like ACT, it has also been developed using Java. The tool allows for the alignment of multiple whole genomes simultaneously through the use of the Mauve algorithm [15] or the updated version, progressiveMauve, both of which perform global alignments. Users can input several different kinds in sequence files to be aligned, including the FastA and Genbank flat file formats.

The algorithm detects regions of sequence homology and assigns these regions to uniquely coloured blocks. Each genome that has been aligned is represented as a central horizontal line positioned in parallel across the screen, with the blocks positioned along the line relative to their sequence position in the genome. Blocks above the line indicate the sequences have been aligned in the forward complement, with blocks below the line representing sequences aligned in the reverse complement. Blocks in the first sequence are connected by lines to similarly coloured blocks in the other sequences, indicating sequences that are homologous.

Similarly to ACT, if the aligned sequences were input using the GenBank format containing annotated sequence features Mauve will display these features alongside their respective genomes as a series of boxes. The boxes of different types of features are displayed in different colours: for instance protein coding regions are displayed in white, whereas tRNA coding regions are red. Mauve also allows users to search an alignment for the location of a specific gene.

[2] http://darlinglab.org/mauve/mauve.html

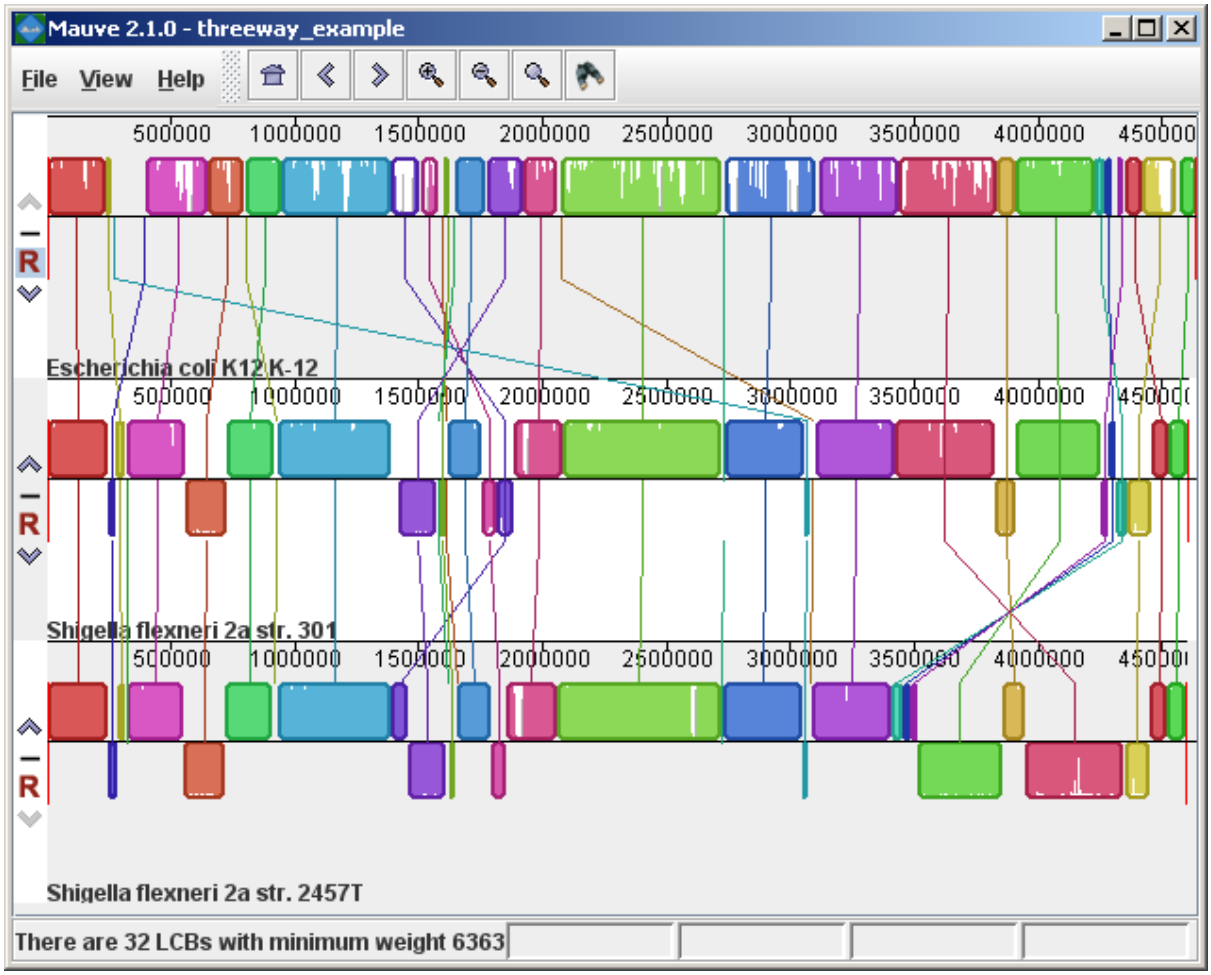


Figure 2.5 An example of a three-way alignment in Mauve.

BLAST Ring Image Generator

The BLAST Image Ring Generator[3] (BRIG) is another tool that allows for the visualisation of multiple genome alignments [16]. The tool was developed by researchers based in the Australian Infectious Diseases Research Centre, and is Java-based like ACT and Mauve. The tool also requires BLAST to be installed locally to perform sequence alignments, and as such takes in sequence files as input which can be of the GenBank, EMBL or FastA formats.

BRIG visualises alignments of a set of query sequences against a reference (subject) sequence and is designed to be used for comparing prokaryotic genomes. Sequences are represented by a series of rings, each of which represents a particular sequence. The intensity of each section of the query sequence rings indicates the sequence identity of that region to the reference sequence. The tool is useful for identifying differences in gene content between the query sequences and the subject sequences. However, this is limited by the fact that users can only see what sequences are present in the subject sequence and absent in the query sequences, not vice-versa. This means that to get a complete comparison the tool would have to be run multiple times, alternating the subject sequence. As well as this, query sequences can only be compared against the subject sequence and not each other.

[3] http://brig.sourceforge.net/

Summary

While each of these tools provides a unique way to visualise sequence alignments, they all lack (with the exception of inversions in ACT) the ability to automatically identify sequence rearrangement features. Each tool requires manual identification, and while each provides a variety of features that may aid in this task, there is clearly a lack of a tool that carries out automated feature identification.

2.3 Subgraph Isomorphism

The subgraph isomorphism problem is problem found in graphical theory that seeks to determine whether when given two graphs, G1 and G2, G1 contains one or more subgraphs that are isomorphic to G2. A subgraph is a graph created from a subset of the sets of vertices and edges of another graph. Two graphs are said to be isomorphic if both contain the same number of vertices connected in the same way (i.e. the graphs are equivalent despite the visual arrangement of their vertices).

The subgraph isomorphism of unlabelled graphs or graphs with repeated labels is an NP-complete (nondeterministic polynomial time) problem [17]. The time required to solve a problem that is NP-complete increases exponentially when increasing the size of the problem. Due to the nature of NP-complete problems algorithms that attempt to solve the subgraph isomorphism problem often rely on rely on certain techniques to improve their efficiency. These can include finding and employing suitable heuristics (imperfect methods or shortcuts that lead to a suitably accurate approximate solution), reducing the number of subgraph isomorphism calls, or relaxing the isomorphism conditions.

There are several popular algorithms available that solve the subgraph isomorphism problem. One of the earliest algorithms was developed by Ullmann in 1976 [18], which is still used today, however there are several more modern algorithms available. Two of these are the VF2 algorithm and the RI algorithm.

2.3.1 The VF2 Subgraph Isomorphism Algorithm

The VF2 subgraph isomorphism algorithm was published by Cordella et al in their 2004 paper [19]. It can be used to solve both isomorphism and subgraph isomorphism problems and is suitable for matching large graphs containing thousands of vertices and branches. The algorithm is an improvement upon a previous iteration released in 1999 [20], the VF algorithm, having increased performance over this previous version. These improvements stem from the reorganisation of data structures required by the algorithm, significantly reducing its memory requirements.

```
PROCEDURE Match(s)
    INPUT:  an intermediate state s; the initial state s0 has M(s0)=∅
    OUTPUT: the mappings between the two graphs

    IF M(s) covers all the nodes of G2 THEN
      OUTPUT M(s)
    ELSE
      Compute the set P(s) of the pairs candidate for inclusion in M(s)
      FOREACH p in P(s)
        IF the feasibility rules succeed for the inclusion of p in M(s) THEN
          Compute the state s´ obtained by adding p to M(s)
          CALL Match(s')
        END IF
      END FOREACH
      Restore data structures
    END IF
END PROCEDURE Match
```

Figure 3 Pseudocode for the VF2 algorithm taken from [19].

2.3.2 The RI Subgraph Isomorphism Algorithm

A more recent subgraph isomorphism algorithm was developed by Bonnici et al. in 2013 [21]. The algorithm, called RI is able to efficiently reduce the search space by removing potential partial paths without resorting to computationally intensive pruning rules. It does this by ordering the vertices of the pattern graph in a way as to introduce edge constraints as soon as possible, and independent of the target graph. The algorithm maintains this ordering throughout, meaning it is static and target independent, unlike the VF2 algorithm which is dynamic and target dependent.

In tests performed in [21], RI outperformed the VF2 algorithm in all instances. While selecting a faster subgraph isomorphism algorithm would obviously be beneficial for the purposes of this project, this project will use an implementation of the VF2 algorithm rather than RI. This decision was made due to the availability of existing implementations of the VF2 algorithm (see section 2.5), as well as previous work demonstrating that the algorithm is capable of performing the calculations necessary for this project sufficiently quickly. Because of this the gains in performance utilising RI would likely be minimal and not worth the extra time dedicated to its implementation.

2.3.3 Related Literature

The complexity and interconnectivity of many biological systems such as metabolic pathways and protein-protein interactions means they are suited to being represented graphically. Detecting specific subgraphs found in these graphs is necessary for understating the networks they represent. Subgraphs that occur regularly can be referred to as network motifs. Motifs can either be designed or detected algorithmically from the graphs themselves [22].

While these techniques have not been applied to the identification of rearrangement features, there are existing studies that make use of these techniques. For example, protein amino acid structures have been modelled in three dimensions using graphs, with vertices representing amino acids and edges the chemical bonds between them. Subgraph isomorphism techniques were then used to detect motifs that identify areas in the proteins where molecular interactions could potentially occur, such as ligand binding sites [23].

2.4 Previous Work

There has already been a meaningful amount of previous work that can be carried forward into this project. Initial work designing the subgraph motifs for each type of rearrangement feature and research into appropriate subgraph isomorphism algorithms was carried out for a BSc dissertation by Eleni Mamalaki in 2013 [24]. The project succeeded in developing a novel graph-based representation of alignments, as well as designing appropriate motifs to represent insertion, deletion and variation rearrangement features within the graph. The project was also successful in implementing the VF2 subgraph isomorphism algorithm to identify these motifs within the graph.

Further development of the concept was carried out for another BSc dissertation produced by Natalie Sandford in early 2017 [9]. This project successfully improved upon the motifs designed in the previous project and was able to produce further motifs to detect inversion and duplication rearrangement features. The nature of these motifs will be explained in detail in the design section in chapter 3.

While both previous projects made great progress in the development of the graphical representation of alignments and the application of subgraph isomorphism algorithms to detect motifs, neither managed to produce an application suitable for use as a bioinformatics tool by an end user. This is where this project aims to continue this previous work from, as well as seeking to improve upon the initial designs where possible.

2.5 Languages, Tools and Technologies

The following is a brief overview of the languages and external libraries that will be used in the development of the system for this project.

2.5.1 Java

Java[4] is an object-orientated programming language developed by Sun Microsystems and owned by the Oracle Corporation. Java applications are compiled into bytecode designed to run on a Java virtual machine. This bytecode can then be run on any architecture that supports a Java virtual machine without requiring recompilation. Java incorporates many standard class libraries to provide many useful facilities to developers. Of particular note is the Swing library which provides classes that implement GUI components and allows for relatively easy construction of functional GUIs.

2.5.2 Graph Libraries

Two external graphical Java libraries were chosen to aid in the development of this project: JGraphT and JGraphX.

JGraphT

JGraphT[5] is a free, open-source Java library that provides implementations of key graph theory concepts and algorithms. The library supports multiple types of graph: including weighted and unweighted graphs, directed and undirected graphs, as well as simple and multigraphs. On top of these graph implementations the library contains various algorithms that can be applied to graphs, such as Dijkstra's algorithm to find the shortest path between nodes in a graph.

Most significantly JGraphT contains an isomorphism package which provides an implementation of Cordella et al.'s VF2 subgraph isomorphism algorithm, which provides a significant advantage to this project as it forgoes the need to develop a custom implementation of the algorithm. The library also provides the ability to easily visualise JGraphT graphs using the JGraphX visualisation library through the *JGraphXAdapter* class, another piece of functionality which may benefit this project in particular.

JGraphX

JGraphX[6] (formerly JGraph), similarly to JGraphT is a free, open-source Java library for graph visualisation, implemented using Java Swing – the official GUI widget library for Java. Due to its implementation using Swing, developers are able to utilise the powerful and flexible features provided by Swing, allowing JGraphX to easily be implemented into Java applications. The package names of the library use those of mxGraph, a JavaScript graph visualisation developed in tandem with JGraphX. While this project is focusing on developing a standalone Java application, mxGraph bears future consideration were a web-based implementation of this work to be developed.

[4] https://www.java.com/en/
[5] http://jgrapht.org/
[6] https://github.com/jgraph/jgraphx

Chapter 3: Design

The following chapter will outline the proposed design of the overall system, including the minimum functionality that is intended to be implemented. The design of the graphical representation of sequence alignments will also be summarised alongside the design of the subgraph motifs.

3.1 System Outline

A basic outline of the system can be seen in Figure 3.1, which demonstrates the proposed functionality to be implemented as well as the relevant external library that will be used to achieve this. The purpose of the system is to take a BLAST hit table file, created from performing a BLAST alignment on a subject and query sequence, as input. The system them parses the file into data objects which can be used to create a graphical representation of the sequence alignment using the JGraphT library. A subgraph isomorphism algorithm is then run against the graph to detect specific motifs which represent different genomic rearrangement features. Finally, the JGraphT graphical representation is converted into a JGraphX format, which can be displayed in a GUI along with the results of the subgraph matching algorithm.

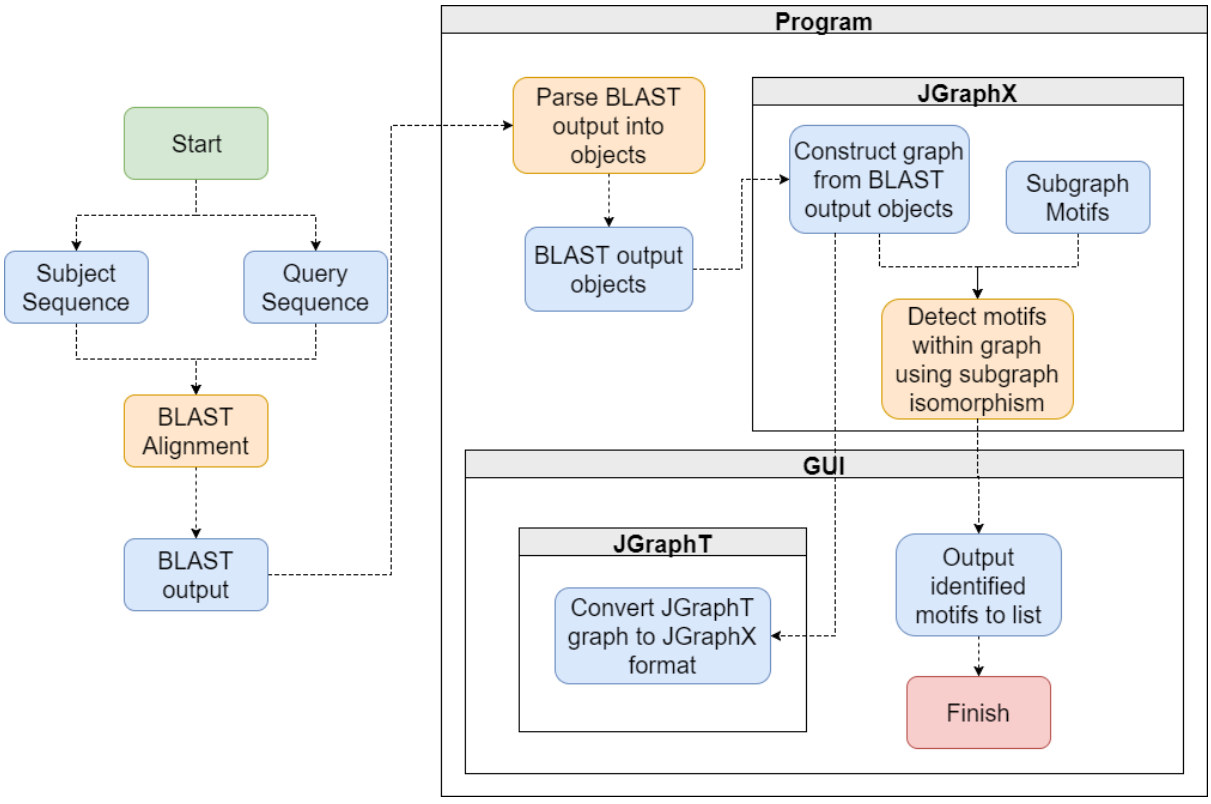


Figure 3.1 Flow chart outlining the core functionality of the system.

3.1.1 Requirements Specification

The following are the functional and non-functional requirements for the system which will be used during the validation stage to determine the success of the system implementation.

Functional Requirements

1. The system should be able to parse alignment data from a .blast file into data objects.
2. The system should be able construct a graph from the data parsed from the .blast file. The graph must be:
    - 2.1. Semantic, to allow comparison of vertices and edges based on the data they represent.
    - 2.2. Directed, to illustrate the forward direction of the sequences.
3. The system must run a subgraph isomorphism matching algorithm against the graph to detect motifs which identify the following genomic rearrangement types:
    - 3.1. Variations.
    - 3.2. Insertions.
    - 3.3. Deletions.
    - 3.4. Inversions in both query and subject sequences.
    - 3.5. Duplications in both query and subject sequences.
4. The system should display the constructed graph in a GUI.
5. The system should display a list of the outputs of the subgraph matching algorithm in a GUI, separated by rearrangement feature type.
6. The system should visualise selected subgraph matches in the graph.

Non-Functional Requirements

1. The system should be written in the Java programming language.
2. The system should run on the Windows operating system.
3. The system should have good performance for all inputs of reasonable size.
4. The system should produce consistent output for multiple iterations over the same data.
5. The system should produce accurate output, identifying the majority of rearrangement features.

3.2 Graph Design

The designs of the graphs and subgraph motifs that will be used in this project are taken from previous work carried out by Eleni Mamalaki and Natalie Sandford in their respective dissertations. What follows is a summary of the designs implemented in these dissertations and the refinements made to them for the purposes of this project.

3.2.1 Graphical Representation of Alignment Data

Before steps can be taken to identify rearrangement types the basic graphical representation of the BLAST output must be established. The basis of this design comes from work by Mamalaki, where the subject and query sequences can be represented as a series of ordered vertices aligned left to right and connected by edges. Each vertex represents a portion of the sequence found in the matches from BLAST output data. One vertex represents the section of the matching sequences found in the subject sequence, and the other vertex the section from the query sequence. These will henceforth be referred to as the subject and query vertices. Corresponding subject and query nodes are connected by an edge to indicate they are part of a match.

As well as holding data (e.g. sequence identity, E-value) about the match it represents each vertex is assigned a direction, forwards or backwards, indicating the direction the matching sequence was found to align against the opposing sequence in.

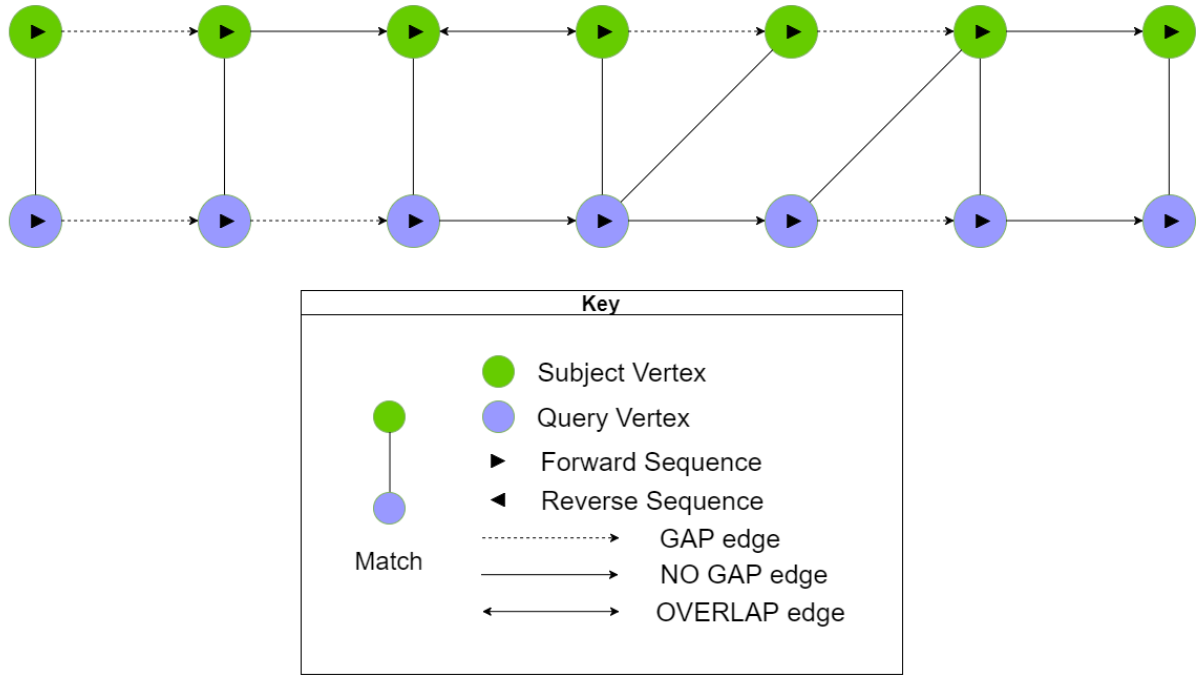


Figure 3.2 The proposed design for the graphical representation of alignment data developed by Mamalaki and Sandford.

Edge Types

Several different type of edges are required to accurately represent the alignment data as well as to allow the creation of appropriate subgraph motifs. There are in total five different edge types: 'MATCH', 'NO GAP', 'GAP', 'OVERLAP' and 'DEFAULT'. MATCH edges are simply the vertical edges of the graph which connect a pair of subject and query nodes to indicate they are matching sequences.

The NO GAP, GAP, OVERLAP and DEFAULT edges are each different types of horizontal edges that join together the individual sets of subject and query vertices to represent their respective sequences. The NO GAP edge indicates that there is no gap between the end of the sequence of one vertex and the start of the sequence of the next vertex in the sequence. The same is true for the GAP edge, only that there is instead a gap between the sequences. The OVERLAP edge indicates that the sequences of two adjacent nodes overlap with one another (i.e. the sequence of one node does not end till after the start of the other). The fifth edge type, DEFAULT, is used to represent any type of edge that is not a MATCH edge and is utilised in the subgraph motifs for duplications.

3.2.2 Subgraph Motif Designs

The following is a description of the motifs that represent each type of genomic rearrangement that will be identified using subgraph isomorphism. The motifs for representing variations, insertions and deletions were initially designed by Mamalaki and remained unchanged in work by Sandford. Revised motifs for representing inversions and duplications were created by Sandford due to limitations identified in the design of the motifs for these features by Mamalaki.

Variation

The variation motif contains four vertices: two subject vertices and two query vertices. The starting subject and query vertices are connecting by a MATCH edge, as are the ending subject and query vertices, creating two pairs of matches. The two subject vertices are then joined by a GAP edge, as are the two query vertices, to represent the regions of non-matching sequences between matching regions in the two sequences.

Insertion

The insertion motif is composed in a similar way to the variation motif, with the addition of a NO GAP edge between the subject vertices in the place of a gap edge.

Deletion

Like the insertion motif, the deletion motif is similar to the variation motif, although the no GAP edge is this time added between the query vertices rather than the subject vertices.

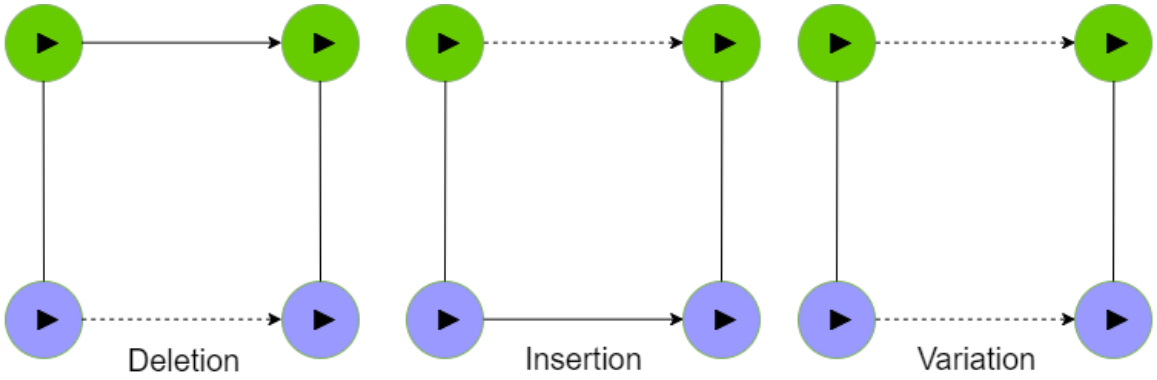


Figure 3.3 Subgraph motif designs for deletions, insertions and variations.

Inversion

There are two different motifs to identify inversions: one to represent inversions in the subject sequence and another to represent inversions in the query sequence. The structure of each motif is simple, involving one subject vertex and one query vertex connected by a MATCH edge. To identify this motif as an inversion rather than a basic match, the motif utilises the direction attribute associated with each vertex, with the two vertices having opposite directions.

Duplication

The design of the motifs to represent duplications is the most complicated of any of the rearrangement feature types, requiring four different motifs. Two motifs are required to detect duplications in each sequence: one to detect duplications where the duplicated sequences are found adjacent to one another (tandem duplications), and one where the duplicated sequences are separated from one other by other sequence regions (dispersed duplications). These features will be referred to henceforth simply as adjacent duplications and duplications respectively. To detect duplications in both subject and query sequences four motifs are necessary.

The motif to represent duplications in the subject sequence requires two subject vertices and a single query vertex. The query vertex represents the sequence that has been duplicated and is located in two regions of the subject sequence, which are represented by the subject vertices. The query vertex is joined by a MATCH edge to each of the subject vertices. The motif to represent adjacent duplications in the subject is similar, yet the two subject vertices are also joined by a DEFAULT edge to represent any type of edge that is connecting the adjacent duplicated vertices.

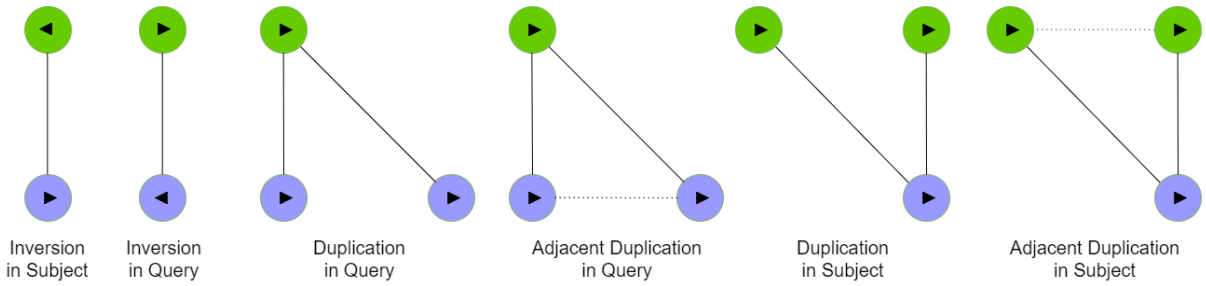


Figure 3.4 Subgraph motif designs for inversions and duplications.

3.2.3 Refinements to the Existing Design

While the existing designs for the alignment representation and subgraph motifs were suitable, some refinements were able to be made to improve them.

The Adjacent Edge Type

While the insertion and deletion motifs were appropriate for detecting these features when the sequences called for a no gap edge to be added, the same is not true for when an overlap edge was present. As such it became apparent a sixth edge type would need to be added for use in the motifs. This edge, the type of which will be named 'ADJACENT', will be used to represent edges that can be of either the NO GAP or OVERLAP types. As such the insertion and deletion motifs will be updated, with the NO GAP edge present in both being replaced by an ADJACENT edge.

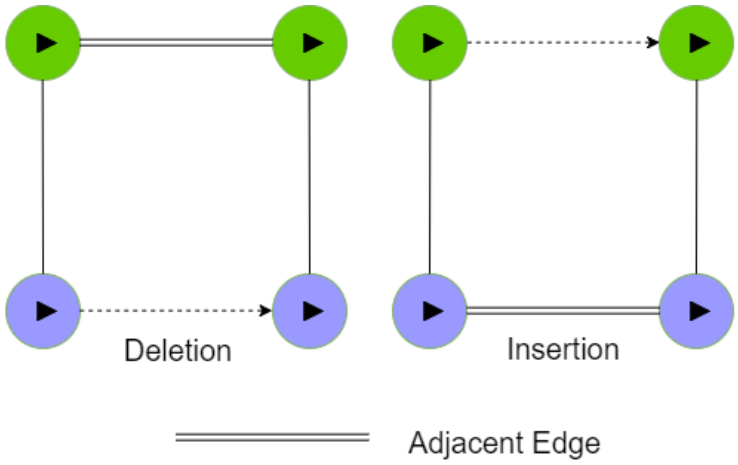


Figure 3.5 Revised subgraph motifs for deletions and insertions incorporating the ADJACENT edge type.

Post Processing of Duplication Subgraphs

The existing set of duplication motifs have been shown in previous work to be suitable for detecting duplication features in the current graph design. However, issues with the design arise when a sequence has been duplicated in the opposite sequence more than two times. Because the motifs only contain two vertices to represent areas which have been duplicated,

the matching algorithm detects a new subgraph for every time an area has been duplicated. This requires some manual identification by the user to determine whether a sequence has in fact been duplicated multiple times. Two possible ways to solve this issue were identified:

- Design further motifs to represent sequences that have been duplicated multiple times.
- Analyse the subgraphs output for the current motifs to detect whether a sequence has been duplicated multiple times and combine the subgraphs to accurately represent this.

The first of these solutions is highly impractical: it is unknown how many times a sequence may be duplicated so in order to compensate for this and ensure all duplications are identified an excess number of motifs would have to be used. These motifs would also be increasingly more complex as the number of duplications increases, resulting in an exponential increase in computation time for each motif due to the NP-complete nature of the subgraph isomorphism problem. Coupled with the necessity for additional motifs to detect adjacent duplications the number of motifs required increases exponentially, making the solution untenable.

Thus the second solution was chosen to be implemented. Post processing of the data does raise issues relating to the goal of this project being to detect rearrangement features using subgraph isomorphism. However, because the duplications have already been detected using subgraph isomorphism before this processing is carried out, it was deemed that this does not compromise on the aims of this project.

3.3 GUI Design

Before beginning the implementation process it is important to prioritise and finalise the initial design of the GUI: both in terms of the features that are aiming to be included and the overall layout of the application. This approach helps to make the implementation process more efficient by providing a minimum set of features to work towards and deliver functionality for, while allowing lower priority features to be added later should time allow.

Other than the basic visualisation of the graphical representation of an alignment, the following features were decided to be included to provide the base functionality of the application, with further features being added should time allow:

- Allow users to manually select a BLAST output file from their file system.
- Allow users to filter out hits from the BLAST table based on parameters such as alignment length or E-value.
- Provide detailed information of each node when it is clicked on by the user.
- The ability to automatically scroll the graph visualisation to a specific sequence location on either the subject or query sequence.
- Display a list for each rearrangement type of each location the motif has been detected in the graph.
- Scroll the visualisation to show and highlight a match when it is selected by the user from the list.
- Allow users to select a new input file to be analysed without restarting the application.

It was decided to keep the design of the GUI similar to those presented by other tools such as ACT and Mauve. In both these tools the main screen is taken almost entirely up by the visualisation of the alignments, with the majority of the functionality of the applications being accessed through the menu bars rather than cluttering the screen and taking space away from the visualisation.

Figure 3.7 shows the initial design concept of the GUI that implements the features listed above. The graphical representation of the alignment is positioned at the top of the screen, and provides users with the ability to scroll horizontally along the sequences. The output of the subgraph isomorphism algorithm are presented below in the bottom left corner. The located subgraphs for each motif are presented in scrollable lists, each separated into different tab based on the rearrangement type. In the bottom right corner is an area for the details of a particular vertex (e.g. sequence ID, E-value) to be displayed when selected by the user. Finally, the menu bar at the top of the screen will be where additional functionality will be accessed, such as scrolling to a particular sequence location and selecting new alignment data.

Figure 3.7 Proposed design concept for the GUI. The numbered sections refer to: (1) the visualisation of the alignment data; (2) the lists displaying the identified arrangement features identified by subgraph isomorphism; (3) detailed information of a match that is displayed when the user selects a vertex; (4) the menu bar where additional functionality can be accessed from.

Chapter 4: Implementation

The implementation of the system was carried out using the Eclipse Integrated Development Environment (IDE) developed by the Eclipse foundation. Some consideration was given to the use of the NetBeans IDE from Sun Microsystems due to its powerful inbuilt GUI design features. However, due to the relatively simple nature of the GUI design it was deemed not worth giving up the familiarity of Eclipse.

4.1 Implementation Breakdown

The nature of the project naturally lent itself into being broken down into several logical components: the construction of a graph based representation of BLAST data, detecting subgraphs within the constructed graph using subgraph isomorphism, and visualising the results of this analysis in a GUI. By breaking down the code into logical units, the code will naturally tend towards having low coupling and high cohesion, resulting in improved readability and maintainability.

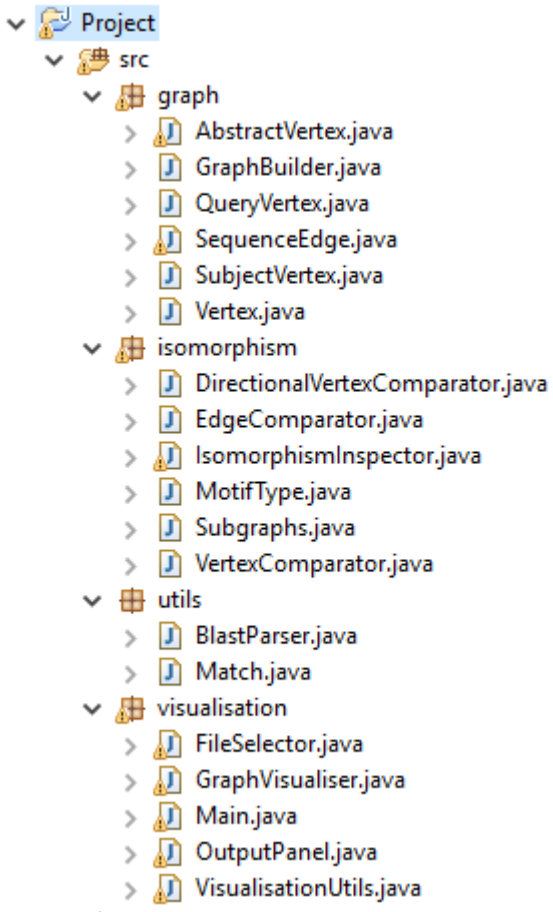


Figure 4.1 Breakdown of the project structure.

As such the system implemented using four packages that reflect these components. The *utils* package contains classes for parsing BLAST output through the *BlastParser* class; and representing these results as Java objects through the *Match* class. The *graph* package contains classes for the construction of graphs using JGraphT through the *SequenceEdge*,

SubjectVertex and *QueryVertex* component classes, with the *GraphBuilder* class managing the construction of the alignment graph using these components. The *isomorphism* package contains classes for running the subgraph isomorphism algorithms on the graph, containing the *EdgeComparator* and *VertexComparator* classes, the *Subgraphs* class, the *IsomorphismInspector* class, and the *MotifType* enumerated type. Finally the *visualisation* package contains classes to display the alignment graph graphically as well as the individual component classes that are used to build the GUI.

4.1.1 Utils Package

The *utils* package contains the first classes that were implemented into the project, the *Match* and *BlastParser* classes. They are used to perform the initial step of converting BLAST output data into usable data objects.

Match

The *Match* class is used to represent a single line, or match, in the BLAST hit table output as an object. The class contains member variables to store the values of each column of the BLAST hit table, including the subject and query IDs, the sequence identity, alignment length, query and subject start and end sequence locations, E-value and Bit-score. The constructor of the class takes in a single line from the BLAST hit table which has been split into an array of strings as a parameter, and assigns values to the member variables from the values in the array at the index which corresponds to the relevant row in the BLAST hit table.

BlastParser

The *BlastParser* class is responsible for parsing data from a BLAST hit table file and creating *Match* objects to represent each row in the table. The class contains a single static method *parse()*, which takes in a string of the location of the file to be parsed as a parameter. The default BLAST hit table format contains a number of lines that contain information about the specific query before the hit table itself begins. These lines are required to be skipped by the parser, and for convenience each begins with a '#' character to easily allow for this. The parser then takes each line of the hit table, splits the line into an array of strings using the whitespace characters around each value, and creates a new *Match* object to represent that line.

The *parse()* method also takes in an integer value which determines the minimum alignment length allowed as set by the user. If the length of the *Match* object is greater than the minimum it is added to an *ArrayList* of *Match* objects which is returned by the method, otherwise it is ignored.

```
public static ArrayList<Match> parse(String file, int minLength) throws FileNotFoundException {

    Scanner sc = new Scanner(new FileInputStream(file));
    ArrayList<Match> matches = new ArrayList<Match>();

    String line;

    while (sc.hasNextLine()) {
        line = sc.nextLine();
        if (!line.startsWith("#")) {
            Match match = new Match(line.split("\\s"));
            if (match.getLength() >= minLength) matches.add(match);
            break;
        }
    }

    while (sc.hasNextLine()) {
        line = sc.nextLine();
        if (!line.isEmpty()) {
            Match match = new Match(line.split("\\s"));
            if (match.getLength() >= minLength) matches.add(match);          }
    }

    sc.close();
    System.out.println(matches.size());
    return matches;
}
```

Figure 4.2 Code for the parse() *method of the* BlastParser *class.*

4.1.2 Graph Package

The *graph* package contains classes used to construct the graph-based representation of BLAST data described in the design chapter utilising implementations found in the JGraphT library.

Sequence Edge

The *SequenceEdge* class is used to represent the edges of the constructed graph. The class extends from the *DefaultEdge* class found in the JGraphT library. The *DefaultEdge* class provides methods to retrieve the source and target vertices of the edge (i.e. the vertices the edge connects). The *SequenceEdge* class contains the *EdgeType* enumerated type and stores a value of the type as a member variable. The *EdgeType* enum contains the values *GAP*, *NO_GAP*, *OVERLAP*, *MATCH*, *DEFAULT* and *ADJACENT*, which are used to identify the specific type of a given edge.

```
public class SequenceEdge extends DefaultEdge {

    private EdgeType type;

    public SequenceEdge(EdgeType type) {
        super();
        this.type = type;
    }

    public enum EdgeType {
        GAP, NO_GAP, OVERLAP, MATCH, DEFAULT, ADJACENT;
    }

    public EdgeType getType() {
        return type;
    }
}
```

Figure 4.3 Code for the SequenceEdge *class, containing the* EdgeType *enum.*

Vertex

The *Vertex* interface defines the basic public methods required by vertex objects.

AbstractVertex

The *AbstractVertex* class is an abstract class that implements the *Vertex* interface. The class is declared as abstract as there is never a situation where it would need to be instantiated within the project as all nodes will either be of the *SubjectVertex* or *QueryVertex* types. The *SubjectVertex* and *QueryVertex* classes are functionally identical other than based in their class, the *AbstractVertex* class was implemented for convenience to prevent the need for repeating code across two separate classes.

SubjectVertex and QueryVertex

The *SubjectVertex* and *QueryVertex* classes both extend from the *AbstractVertex* class. Both classes contain only a basic constructor which calls the constructor of the superclass, *AbstractVertex*. These classes exist to allow the subgraph isomorphism algorithm to compare vertices on a semantic level and not to provide any further functionality over that provided by the *AbstractVertex* class. The use of semantics will be discussed further in the discussion of the *isomorphism* package.

GraphBuilder

The *GraphBuilder* class is responsible for the construction of the JGraphT graph from the data parsed from the BLAST output. The class creates a *ListenableDirectedGraph*, meaning the graph is listenable: it can have listeners added to it to detect changes to its vertex and edge sets; and directed: edges have a specific direction associated with them. The *ListenableDirectedGraph* identifies its vertices and edges using type declarations, which in this case are objects that implement the *Vertex* interface and *SequenceEdge* objects respectively.

```
public GraphBuilder(ArrayList<Match> blastData) {

    graph = new ListenableDirectedGraph<Vertex, SequenceEdge>(SequenceEdge.class);
    this.blastData = blastData;
    subjectVertices  = new ArrayList<SubjectVertex>();
    queryVertices = new ArrayList<QueryVertex>();

    populateGraph();

    Collections.sort(subjectVertices);
    Collections.sort(queryVertices);

    addHorizontalEdges();
}
```

Figure 4.4 Code for the constructor of the GraphBuilder *class, demonstrating the order of operations which are used to build the graph.*

The graph is constructed by creating a pair of vertices, one subject vertex and one query vertex, joined by an edge to represent each *Match* object parsed from the BLAST hit table. When they are created, each vertex is added to a collection of their respective type. This is carried out by the *populateGraph()* method. Both sets of vertices are then sorted into the order they appear in their respective whole sequence using the start and end positions of the sequences they represent. This allows the horizontal edges to be added to connect the vertices so they can represent their respective sequence. Before each edge is added it compares the start and end points of the two vertices to determine whether a gap is present between the matches in the sequence, then applies the appropriate *EdgeType* to the edge.

```
private EdgeType gapCheck(Vertex current, Vertex next) {

    int currentStart = current.getStart();
    int currentEnd = current.getEnd();

    int nextStart = next.getStart();
    int nextEnd = next.getEnd();

    if (((currentEnd + 1) == nextStart) || ((currentEnd + 1) == nextEnd) ||
            ((currentStart + 1) == nextStart) || ((currentStart + 1) == nextEnd)) {
        return SequenceEdge.EdgeType.NO_GAP;
    }
    else if (((currentEnd + 1) < nextStart) && ((currentStart + 1) < nextStart) &&
            ((currentEnd + 1) < nextEnd) && ((currentStart + 1) < nextEnd)) {
        return SequenceEdge.EdgeType.GAP;
    }
    else return SequenceEdge.EdgeType.OVERLAP;
}
```

Figure 4.5 Code for the gapCheck() *method, illustrating how the type for each horizontal edge is determined.*

4.1.3 Isomorphism Package

The *isomorphism* package contains classes that allow for subgraph motifs to be searched for in graphs implemented by the classes contained in the *graph* package.

MotifType

The *MotifType* enumerated type contains elements that are used to identify each type of rearrangement motif that can be searched for.

Subgraphs

The *Subgraphs* class contains provides static methods to construct graphs to represent each type of rearrangement motif that can be searched for. Each method constructs a single *ListenableDirectedGraph* from *Vertex* and *SequenceEdge* objects that represents a given rearrangement motif. These graphs can then be compared against the alignment graph produced by the *GraphBuilder* class using the subgraph isomorphism algorithm. The class contains no constructor as it does not have any need to be instantiated, instead the methods to create each motif graph are called statically by the *IsomorphismInspector* class.

```
public static ListenableDirectedGraph<Vertex, SequenceEdge> inversionInSubject() {

    ListenableDirectedGraph<Vertex, SequenceEdge> inversionInSubject =
            new ListenableDirectedGraph<Vertex, SequenceEdge>(SequenceEdge.class);

    SubjectVertex subject = new SubjectVertex("Subject", 2, 1, 0.0, 0, 0.0, "0.0");
    QueryVertex query = new QueryVertex("Query", 1, 2, 0.0, 0, 0.0, "0.0");

    inversionInSubject.addVertex(subject);
    inversionInSubject.addVertex(query);

    SequenceEdge edge = new SequenceEdge(SequenceEdge.EdgeType.MATCH);
    inversionInSubject.addEdge(subject, query, edge);

    return inversionInSubject;
}
```

Figure 4.6 Example of the construction of a subgraph, in this case for the inversion in subject motif.

IsomorphismInspector

The *IsomorphismInspector* class is responsible for carrying out the process of detecting rearrangement motifs within the alignment graph.

The class uses the *VF2SubgraphIsomorphismInspector* class provided by the JGraphT library to detect rearrangement motifs. The constructor of the *VF2SubgraphIsomorphismInspector* class takes in two graphs as parameters, *graph1* and *graph2*. *Graph2* is checked against the *graph1* to see whether *graph1* contains any subgraphs isomorphic to *graph2*. The constructor also takes in two Comparator objects: an edge comparator and a vertex comparator. These comparators allow the vertices and edges to be compared semantically when searching for subgraphs, meaning that the properties of the objects can be used to determine whether or not they are the same. If comparators are not used, the algorithm will match subgraphs based entirely on basic vertex and edge relationships, with other properties such as the specific type of an edge not being taken into account. Finally, the constructor takes in a Boolean

parameter, *cacheEdges*, which determines whether edges are cached for faster access. Some informal testing determined using this feature provided no performance increase (in some cases actually decreasing performance), so it was not used.

The *VF2SubgraphIsomorphismInspector* class contains the *getMappings()* method which returns an iterator over all the mappings between the two graphs. Each mapping represents the position in the query graph where the subgraph has been located. The *IsomorphismInspector* class uses two methods – *mappingsToArray()* and *mappingToSubgraph()* – to convert each mapping into an individual *ListenableDirectedGraph*.

```
ListenableDirectedGraph<Vertex, SequenceEdge> subgraph =
        new ListenableDirectedGraph<Vertex, SequenceEdge>(SequenceEdge.class);

Set<Vertex> vertices = graph.vertexSet();

for (Vertex v : vertices) {
    Vertex vertex = mapping.getVertexCorrespondence(v, true);
    if (vertex != null) {
        subgraph.addVertex(v);
    }
}

//Search for and add edges
for (Vertex v : subgraph.vertexSet()) {
    for (Vertex v2 : subgraph.vertexSet()) {
        if (graph.containsEdge(v, v2)) {
            subgraph.addEdge(v, v2, graph.getEdge(v, v2));
        }
    }
}

return subgraph;
```

Figure 4.7 Code from the mappingToSubgraph() *method demonstrating how each mapping is converted to a subgraph.*

The class contains methods to search for an individual rearrangement type as well as to automatically perform a search for all the rearrangement types. The *inpect()* method performs the search for a given arrangement type and returns an *ArrayList* of the detected subgraphs produced by the *mappingsToArray()* and *mappingToSubgraph()* methods. The *inspectAll()* method carries out the *inpect()* method for all rearrangement types and returns all the located subgraphs in a *HashMap* which uses the motif type as a keys to identify each *ArrayList* of subgraphs, which are stored as the values of the map.

For duplication motifs, the class contains a final method, *combineDuplications()* which carries out the post processing to duplication motifs discussed in section 3.2.3. Subgraphs representing regions that have been duplicated in more than two locations are combined by searching the collection of subgraphs for those which share the same duplicated vertex.

Vertex Comparators

The original implementation of the design by Sandford contained a single comparator class that was used to compare vertices. This comparator compared vertices on their type, subject or query, and then on their sequence direction. Two vertices were deemed to be the same if both of these were the same. However, using this comparator to compare vertices for all motif types is unnecessary, with comparison of the direction attribute only being necessary to identify inversions. It could also potentially hinder the identification of other motifs. For example, a duplication where one sequence has also been inverted would not be identified.

To account for this, two different comparator classes were implemented: the *VertexComparator* class and the *DirectionalVertexComparator* class. The *VertexComparator* class compares vertices by simply comparing whether they are a subject vertex or a query vertex. The DirectionalVertexComparator class is similar to the comparator implemented by Sandford, comparing vertices on both their type and direction. It is used as the comparator for identifying inversion motifs, while the *VertexComparator* class is used for identifying all other motif types.

EdgeComparator

The *EdgeComparator* class was implemented to allow for the semantic comparison of edges when searching for matching subgraphs. The method for comparison of edges is more complicated than that of comparing vertices due to the presence of the DEFAULT and ADJACENT edge types.

Because the DEFAULT edge type can represent any other type of edge other than the MATCH type, if one of the edges being compared has the DEFAULT type, provided the edge it is being compared against is does not have the MATCH type, the two edges are deemed to be equal. Similarly, the ADJACENT type represents edges of both the NO GAP and OVERLAP types. Therefore if one of the edges being compared has the ADJACENT type, the two edges are deemed equal if the other is of either the NO GAP or OVERLAP types.

If neither of the edges being compared has the DEFAULT of ADJACENT type, the edges are simply compared on type alone, with edges sharing the same type being declared as equal.

4.1.4 Visualisation Package

The *visualisation* package contains classes responsible for converting the JGraphT graph into a JGraphX graph that can be displayed on screen, as well as for constructing the individual components that make up the GUI of the application. The package also contains the *main* method that the program is run from.

GraphVisualiser

The *GraphVisualiser* class has several key functions: to convert the JGraphT alignment graph into a JGraphX graph, and manage how the graph produced is displayed on screen by controlling the style and layout of the graph components. The class extends from the *mxGraphComponent* class from the JGraphX library, which allows it to be added as a Java Swing component to an application.

In order to understand how the functions of the class are implemented, some explanation of how the JGraphX library visualises graphs is required. The *mxGraph* class is used to represent a graph object in the JGraphX library, with an *mxGraph* object being displayed visually by an *mxGraphComponent* object, provided to it in the constructor of the class. The actual structure of the graph (i.e. the vertices and edges and their relationships) represented by an *mxGraph* object is in fact managed by an underlying class called *mxGraphModel*, similar to the way the *JList* and *JTable* classes in the Swing toolkit manage their data. Changes to the data of the graph, such as addition and removal of vertices and edges, are performed through the *mxGraphModel* API.

Each part of an mxGraph is represented by object of the *mxCell* class. An *mxCell* is used to represent both vertices and edges. Each cell holds an object (the vertex or edge object that it is representing in the model), the specific geometry of the cell (where it is displayed on screen using x and y coordinates) and the visual style of the cell.

To convert the JGraphT graph constructed by the *GraphBuilder* class, each vertex and edge must be converted into its own cell and added to the model. While the JGraphT library does contain a method to automatically convert to the JGraphX format – the *JGraphXAdapter* class – doing so means it is harder to perform later tasks. For example, highlighting specific subgraphs becomes tricky as with automatic conversion the ability to track which cell corresponds to which vertex is lost. Therefore vertices and edges are converted manually, with the cell-vertex/edge pairs being added to *HashMap* objects so they can be easily referred to later. This is carried out by the *buildModel()* method, which uses the *addVertex()* and *addEdge()* methods to convert the vertices and edges to cells and add then to the graph model.

```
public GraphVisualiser(mxGraph graph, GraphBuilder builder) {

    super(graph);

    this.builder = builder;
    graphT = builder.getGraph();
    model = new mxGraphModel();
    graphX = graph;

    graphX.setModel(model);

    buildModel();
    buildStyleSheet();
    graphVisualisation();
}
```

Figure 4.8 Constructor for the GraphVisualiser *class, showing the sequence in which the elements of the graph are constructed.*

For convenience, the JGraphX library allows an *mxGraph* to have a stylesheet allocated to it. The implementation of these stylesheets is conceptually similar to the application of CSS stylesheets to HTML, in that the developer can define and name specific combinations styles and add them to the stylesheet. There are many different style available that can be applied

to an mxCell, such as the colour of vertices or whether an edge is dashed. The complete collection of styles that can be applied can be found in the *mxConstants* class: the Strings that define styles begin with "STYLE_" followed by the name of the specific style. The styles of an *mxCell* can then be set using simply the name given to the style combination in the stylesheet.

The *GraphVisualiser* class creates and applies the stylesheet during the creation of the mxGraph using the *buildStyleSheet()* method. The stylesheet defines the different styles for both subject and query vertices as well as for the different types of edges, ensuring edges that indicate a gap in the sequence are displayed as dashed, for instance. The stylesheet also defines a highlighted style to allow for specific vertices and edges to be highlighted a different colour.

Finally, the *graphVisualisation()* method is used to both apply the relevant styles from the stylesheet to each cell, as well as set the geometry of each cell (i.e. the x, y coordinates of each cell). The method also alters the permissions of the graph, ensuring users cannot move or edit the vertices or edges once they have been drawn to the screen.

OutputPanel

The *Output* panel class is used to display the results of the subgraph matching algorithm run on the graph. The class itself extends from *JPanel* so it can be easily added to the GUI. Within the panel the identified subgraphs for each rearrangement type are displayed using a *JList*. Each list is separated to occupy a single tab of a *JTabbedPane*.

FileSelector

The *FileSelector* class is a custom *JDialog*, constructed from Swing components, which allows users to browse for or manually enter the alignment file they want to be analysed. The dialog also gives the option to users to specify a minimum match length to impose on the data should they choose.

VisualisationUtils

The *VisualisationUtils* class contains static helper methods used to carry out operations on the JGraphX visualisation called from other elements of the GUI. More specifically is contains the *highlightMotif()* and *scrollToPos()* methods which highlight a subgraph in the visualisation that has been selected from the *OutputPanel*; and scroll the visualisation to a specified sequence position, respectively.

Main

The *Main* class contain the *main()* method, the method through which all Java applications begin their execution. The main method is also responsible for the construction if the complete GUI using the *OutputPanel* and *FileSelector* classes as well as other Swing components.

Chapter 5: Validation

Testing was performed incrementally throughout the implementation process at logical points to determine each section of the system was functioning as expected. The process of testing was made easier through the use of the powerful debugging tools provided by Eclipse, which allows developers to place break-points in the code, view runtime assignment of variables and line-by-line step-through of code. JUnit, a framework for writing repeatable automated tests, was also used to create test classes to test the functionality of certain classes when appropriate (e.g. the edge and vertex comparator classes).

```
@Test
public void test()
{
    assertNotEquals(0, comparator.compare(matchEdge, defaultEdge));
    assertNotEquals(0, comparator.compare(matchEdge, gapEdge));
    assertNotEquals(0, comparator.compare(matchEdge, noGapEdge));
    assertNotEquals(0, comparator.compare(matchEdge, overlapEdge));
    assertNotEquals(0, comparator.compare(matchEdge, adjacentEdge));

    assertEquals(0, comparator.compare(defaultEdge, gapEdge));
    assertEquals(0, comparator.compare(defaultEdge, noGapEdge));
    assertEquals(0, comparator.compare(defaultEdge, overlapEdge));
    assertEquals(0, comparator.compare(defaultEdge, adjacentEdge));

    assertNotEquals(0, comparator.compare(gapEdge, noGapEdge));
    assertNotEquals(0, comparator.compare(gapEdge, overlapEdge));
    assertNotEquals(0, comparator.compare(gapEdge, adjacentEdge));

    assertNotEquals(0, comparator.compare(noGapEdge, overlapEdge));
    assertEquals(0, comparator.compare(noGapEdge, adjacentEdge));

    assertEquals(0, comparator.compare(overlapEdge, adjacentEdge));
}
```

Figure 5.1 Example of a Junit test to ensure the correctness of the EdgeComparator *class.*

Due to refinements made to previous designs in the design chapter, the implementation of the new graph-based alignment representation and subgraphs motifs would have to be tested as well. This was carried out using both real and artificial alignment data.

5.1 Testing using Artificial Data Sets

The system was first tested with artificially designed comparison data in order to confirm the ability of the subgraph motifs to detect rearrangement features. The artificial data was designed to incorporate each type of rearrangement feature that the system is required to identify. Figure 5.1 illustrates how each rearrangement type was incorporated into the data by presenting the data in the style of an ACT visualisation. The complete artificial data is included in the appendices.

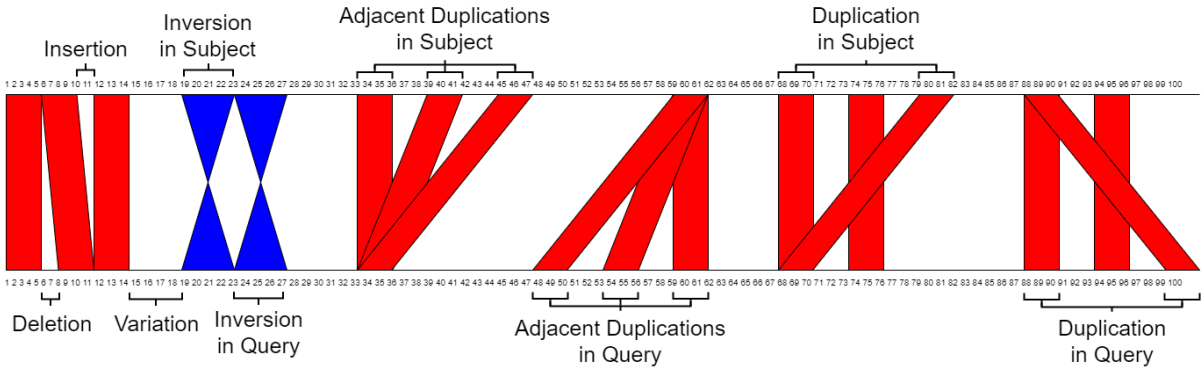


Figure 5.2 Breakdown of the structure of the artificial data, presented in the style of an ACT visualisation.

5.1.1 Results

The results of running the artificial data though the system can be seen in Table 5.1.

REARRANGEMENT TYPE	EXPECTED	ACTUAL
VARIATION	8	6
INSERTION	1	1
DELETION	1	1
INVERSION IN SUBJECT	1	1
INVERSION IN QUERY	1	1
DUPLICATION IN SUBJECT	2	4
ADJACENT DUPLICATION IN SUBJECT	2	2
COMBINED DUPLICATIONS IN SUBJECT	2	2
DUPLICATION IN QUERY	2	4
ADJACENT DUPLICATION IN QUERY	2	2
COMBINED DUPLICATIONS IN QUERY	2	2

Table 5.1 Results for the location of rearrangements motifs in artificial data.

Unfortunately, the system still has issues accurately identifying all occurrences of the variation rearrangement feature, a problem which was identified in previous work by Sandford. The cause of this is vertices that are nested between duplicated vertices, resulting in the structure of the graph not matching the variation motif.

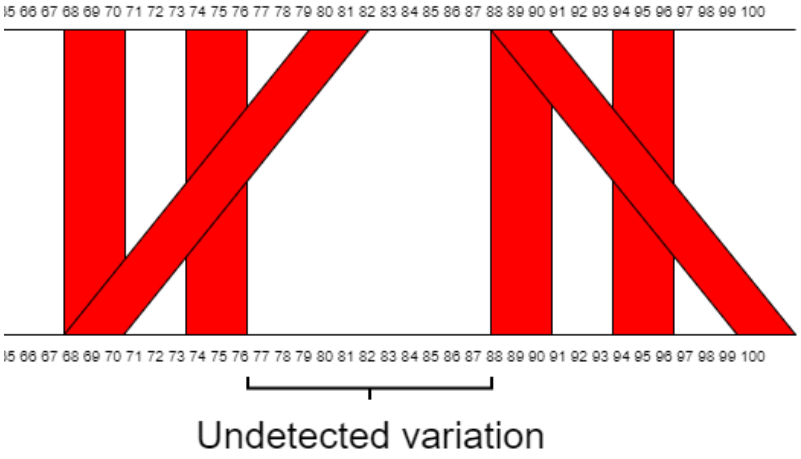


Figure 5.3 Example of how a vertex nested between duplicated vertices can prevent variations from being identified.

Another issue that can be identified is in the identification of duplication motifs, with the same subgraph being identified twice for each occurrence of the motif. The exact reason for this behaviour is unclear, and is potentially being caused by a bug in the *VF2SubgraphIsomorphismInspector* class of the JGraphT library. Interestingly, this issue did not occur during testing performed by Sandford. This may be down to the previous implementation storing the identified subgraphs in a *HashSet* as opposed to an *ArrayList*, which is used in the implementation for this project. A *HashSet* does not allow duplicate elements, so the duplicate subgraphs could potentially have been missed due to not being able to be added to the set. Fortunately, this issue does not compromise the overall accuracy of the system, as the duplicate subgraphs are removed during the process of combing the duplications. This results in the overall amount of duplications being identified being correct.

5.2 Testing using Real Data Sets

While testing with artificial data sets is useful, it is important to ensure the system is able to identify rearrangement features when presented with real alignment data – the kind of which will be input by end users of the system. Therefore, once testing with artificial data sets confirmed the accuracy and reliability of the system, data sets containing the alignment of real genomes was used.

This testing was performed with two different strains of the *Staphylococcus aureus* species of bacterium: more specifically *Staphylococcus aureus subsp. aureus NCTC 8325*[7] and *Staphylococcus aureus subsp. aureus Mu50*[8], which were used as the subject sequence and the query sequence respectively. A BLAST alignment was carried out on the two sequences to obtain a .blast file, which was then input into the system.

While using artificial test data provides the benefit of being able to know the specific number and type of rearrangements, using real alignment data unfortunately does not have that advantage. Being able to comprehensively determine whether the system has identified every rearrangement feature present in real alignment data would require a tool to identify these feature beforehand – the exact goal this project aims to accomplish. Despite this it is still of benefit to test using real data as it allows the speed of the matching algorithm to be measured.

5.2.1 Results

The alignment of *Staphylococcus aureus subsp. aureus NCTC 8325* and *Staphylococcus aureus subsp. aureus Mu50* produced a .blast file containing 8560 matches. Table 5.2 shows the frequency of occurrence of each rearrangement feature as well as the average speed the matching algorithm took to identify the subgraphs for each motif.

As can be seen from the results in Table 5.2, the total time to for the system to analyse a complete genome sequence alignment containing a large number of matches came in to well under a minute. This includes the time taken to construct the graphical representation as well as convert it into the JGraphX format to be visualised. While the time required will differ depending on the hardware running the application, it is unlikely to excessively increase the run time, meaning the performance of the system seems more than appropriate. It should also be noted that alignments with a fewer amount of matches will take exponentially less time to run due to the NP-complete nature of the subgraph isomorphism problem.

[7] https://www.ncbi.nlm.nih.gov/nuccore/NC_007795.1?report=fasta
[8] https://www.ncbi.nlm.nih.gov/nuccore/NC_002758.2?report=fasta

REARRANGEMENT TYPE	FREQUENCY	MEAN RUN TIME (MILLISECONDS)
VARIATION	294	1423
INSERTION	7	562
DELETION	7	1428
INVERSION IN SUBJECT	3646	3767
INVERSION IN QUERY	0	139
COMBINED DUPLICATIONS IN SUBJECT	900	16835
COMBINED DUPLICATIONS IN QUERY	758	5510
CONSTRUCT JGRAPHT GRAPH		3670
CONVERT TO JGRAPHX GRAPH		5128
TOTAL		38462

Table 5.2 Performance breakdown when using alignment data from Staphylococcus aureus subsp. aureus NCTC 8325 *and* Staphylococcus aureus subsp. aureus Mu50*.*

As mentioned previously, while it is difficult to know the accuracy to which the system is identifying the frequency of rearrangement features, it is possible to confirm positive identifications using ACT. For example, Figure 5.3 illustrates the identification of an inversion in both ACT and the system developed for this project.

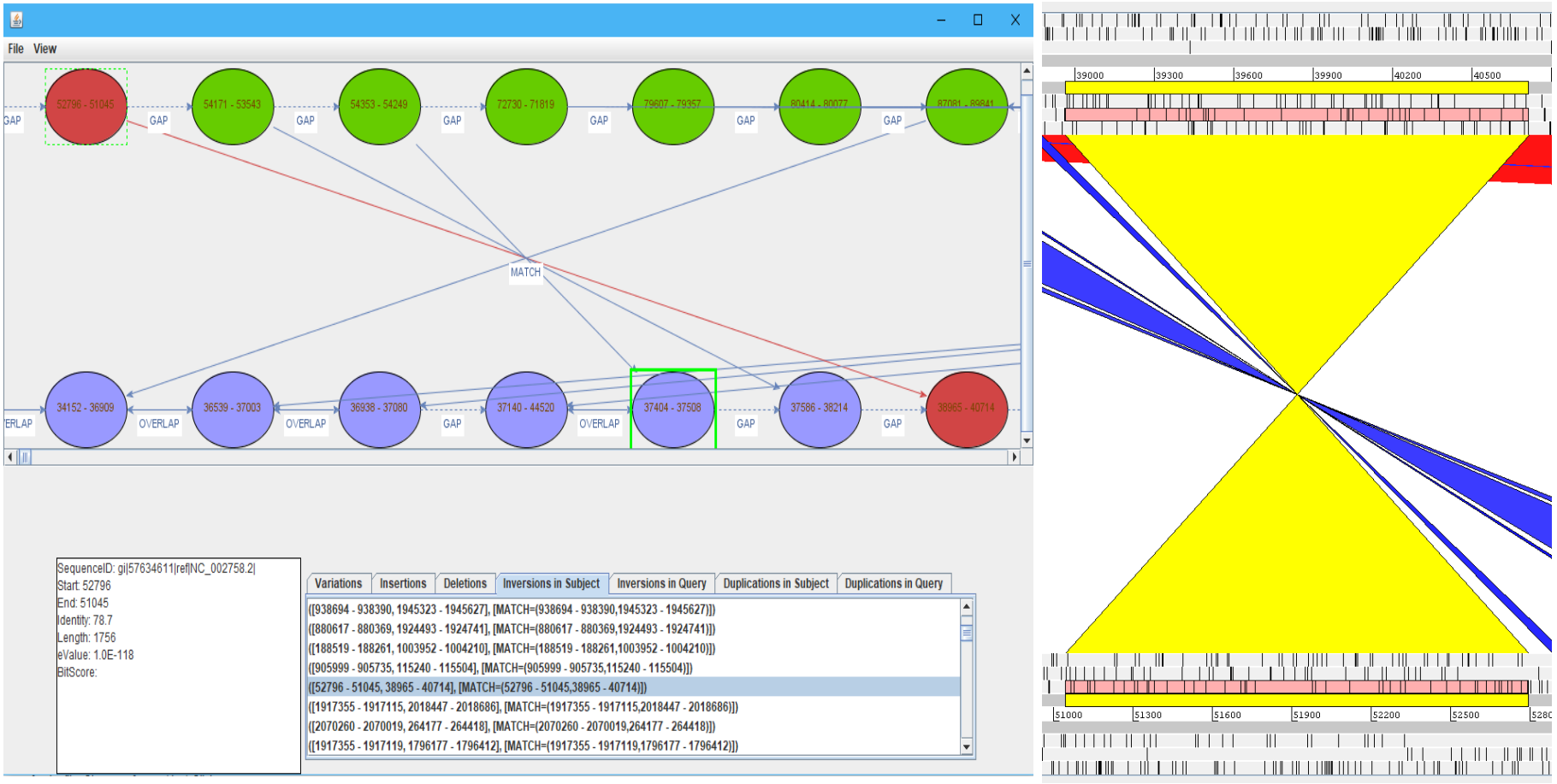


Figure 5.4 An inversion in the subject sequence identified in both the system (left, highlighted in red) and in ACT (right, highlighted in yellow) between positions 52796 and 51045 in the subject sequence and positions 38965 and 40714 in the query sequence.

Chapter 6: Evaluation

6.1 Performance with Large Data Sets

When the system is presented with a set alignment data of several thousand rows – the kind that is produced from the alignment of relatively similar bacterial genomes, as used in the testing performed above – several issues arise. Firstly, the graphical representation presented in the GUI can become incredibly cluttered and unreadable. The cause of this issue is the amount of matching edges that are being displayed, and this issue is compounded when there are multiple matching regions of one sequence located farther along the opposing sequence. The large amount of match edges can obscure the sequence vertices, making them difficult to select or determine whether they have been highlighted to indicate their presence in a rearrangement feature.

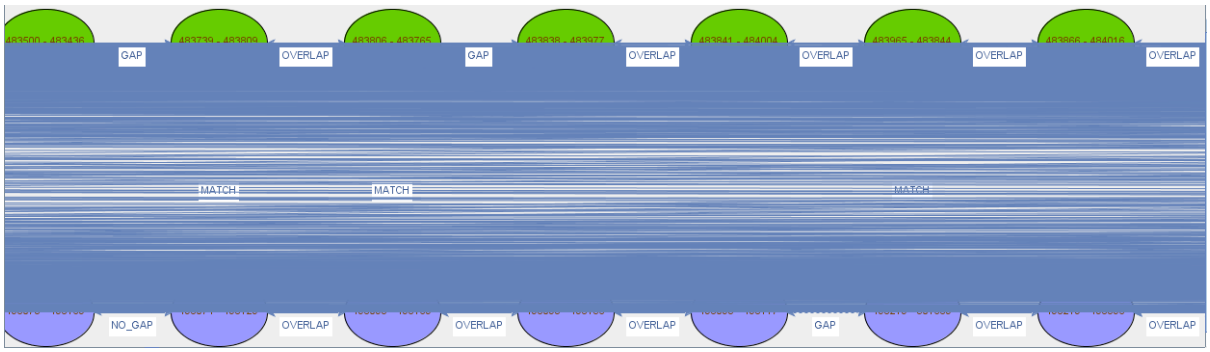


Figure 6.1 Illustration of how using large data sets can impact the readability of the visualisation.

Secondly, especially large data sets produce long alignment graphs with a great number of vertices and edges, causing the performance of the system to slow down tremendously and often become unusable due to application freezing. This is likely caused by JGraphX not being designed to draw the necessary large amounts of cells to screen at one time. JGraphX provides the ability to 'collapse' cells so they are no longer visible, with collapsing cells that away from the region being displayed on screen having the possibility to improve performance. However, this would compromise the visualisation of the alignment as it would cause match edges that connect to vertices off screen from displaying correctly if their target vertices are collapsed.

However, while the GUI struggles to handle large alignment data sets, the underlying system of automatically identifying rearrangement features still performs well. It may therefore be of benefit to redesign the system in such a way that the detailed information of each identified rearrangement can be viewed by the user without relying on using the graphical representation presented in the GUI.

6.1.1 Setting Minimum Match Length

Problems caused by the input of large data sets can be somewhat circumvented by setting a minimum length to the matches incorporated into the graph. While the BLAST algorithm does make efforts to remove matches that are unlikely to be of significance, many smaller matches are still identified in the output.

When a minimum alignment length of 100 base pairs (indicating matches that are likely to be of scientific interest) was applied to the real alignment data used for testing in section 5.2, the number of matches was reduced from 8560 to 2150.

REARRANGEMENT TYPE	FREQUENCY	MEAN RUN TIME (MILLISECONDS)
VARIATION	426	719
INSERTION	11	328
DELETION	21	356
INVERSION IN SUBJECT	660	487
INVERSION IN QUERY	0	68
COMBINED DUPLICATIONS IN SUBJECT	144	1181
COMBINED DUPLICATIONS IN QUERY	72	516
CONSTRUCT JGRAPHT GRAPH		776
CONVERT TO JGRAPHX GRAPH		1973
TOTAL		6404

Table 6.1 Performance breakdown when using alignment data from Staphylococcus aureus subsp. aureus NCTC 8325 *and* Staphylococcus aureus subsp. aureus Mu50*, with a minimum alignment length of 100 base pairs.*

As can be seen in Table 6.1, the performance in terms of run time improves dramatically when setting a minimum match length for this particular alignment. This is expected due to the significant reduction in the number of matches. The reduced number of matches alleviates many of the problems in performance discussed previously, so it would likely be of benefit to encourage users to set a minimum match length before entering large data sets.

Setting a minimum match length does raise certain questions as it leads to the creation of a differently structured graph and therefore differing amounts of each rearrangement type, as illustrated when comparing the frequencies of each motif in Table 5.2 and Table 6.1. This is something that users will have to consider themselves based on the specific alignment data they are entering – judging whether matches below a certain length can safely be ignored as scientifically insignificant based on the species being compared or length of the alignment.

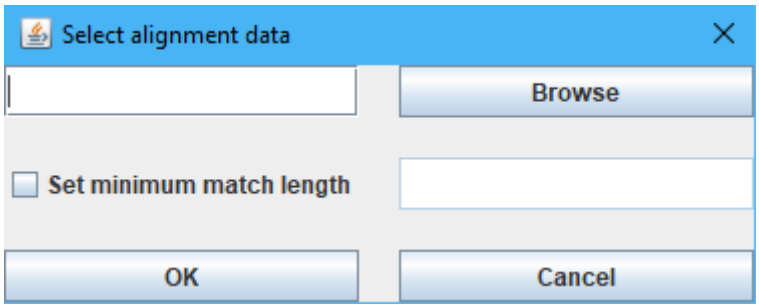
6.2 GUI Evaluation

Evaluating the success of the implementation of the GUI can be carried out by comparing which features are present in the final implementation to those laid out in the design section:

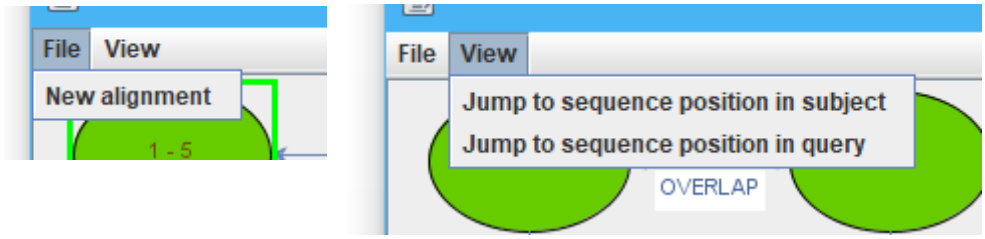
- **Allow users to manually select a BLAST output file from their file system.**
  Users are able to manually enter the location an alignment file from their system from a dialogue menu on start up. Users also have the ability to browse their file system to select a file, with files being filtered to only display .txt and .blast files for ease of use.
- **Allow users to filter out hits from the BLAST table based on parameters such as alignment length or E-value.**
  Users can set a minimum match length to filter out matches under a specified length. Unfortunately the ability to filter out by other values such as E-value was not added so this feature has only been partially implemented.
- **Provide detailed information of each node when it is clicked on by the user.**
  Users can select each node and view its sequence ID, start and end position, sequence identity, length, E-value and Bit-score.
- **The ability to automatically scroll the graph visualisation to a specific sequence location on either the subject or query sequence.**
  From the View menu item users can select to jump to a specific sequence position on both the subject and query sequence. The view will scroll to the first vertex that contains that sequence position or the nearest vertex to the sequence position if it is located between vertices.
- **Display a list for each rearrangement type of each location the motif has been detected in the graph.**
  The identified subgraphs are displayed in separate lists for each of the rearrangement types – variation, insertion, deletion, inversion (subject and query) and duplication (subject and query).
- **Scroll the visualisation to show and highlight a match when it is selected by the user from the list.**
  Each subgraph from the results lists can be selected, causing the visualisation to scroll to the first sequence vertex present in the subgraph. Each vertex and edge in the subgraph is then highlighted in red to allow them to be easily recognised by the user.
- **Allow users to select a new input file to be analysed without restarting the application.**
  Users can select a new alignment file to be input from a selection in the File menu. The new visualisation for the new alignment is opened in a new window.

Figure 6.2 illustrates some of these features in action in the final implementation of the system.

A.



B.



C.

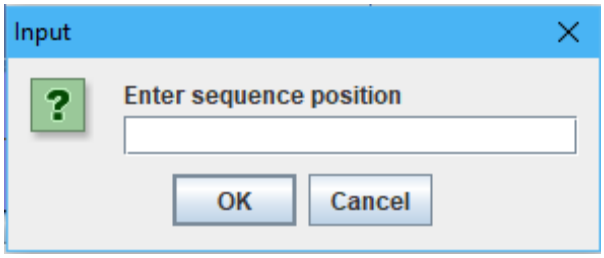


Figure 5.2 (A) The new alignment dialogue panel, which allows users to select an alignment file, as well as optionally set a minimum match length. (B) The options provided by the File and View menu items. (C) The input dialogue to jump to a select sequence position.

While the majority of the planned features were implemented successfully, the issues raised with the visualisation of large data sets described in section 6.1 do show that further work could be performed to improve upon the implementation of the GUI.

6.3 Graphical Representation Evaluation

Overall the graphical representation has worked effectively, although there are still several problems outstanding that have not been able to be solved – most notably in the masking of variations caused by nested vertices. Overcoming this problem would likely require a large redesign of the graphical representation and potentially sacrifice its current easily readable format.

Another issue with the design of the graphical representation may come from defining the gap edge type too strictly. Only one base is required to be present between the sequences of adjacent vertices for an edge to be assigned the GAP type. This could potentially lead to situations where insertions or deletions may be being identified as variations when they perhaps should not be. It may therefore be of benefit for future implementations to consider defining the GAP edge type less strictly either within the code or, most likely a better solution, allow users to set their own threshold for the number of bases allowed before there is considered to be a gap between matches.

6.4 Evaluation of Objectives

In order to determine a measure of the success of this project it is helpful to look back at the objectives laid out in section 1.2 and determine the degree to which they were achieved by the final implementation:

1. **Implement and potentially improve upon an existing design for the graphical representation of sequence alignments.**
   Achieved. The existing graphical was implemented successfully, with the design not requiring any improvements to be made. Testing using artificial data confirms the accuracy of the implementation.

2. **Implement and potentially improve upon existing designs of subgraph motifs to represent specific genomic rearrangement features.**
   Achieved. The existing subgraph motifs were implemented successfully, with the insertion and deletion motifs being improved through the addition of the adjacent edge type. Testing using artificial data confirms the accuracy of the implementation.

3. **Identify the motifs implemented (2) in the graphical representation implemented in (1) using a subgraph isomorphism matching algorithm.**
   Mostly achieved. All the motifs were able to be detected in both artificial and real data sets. The accuracy of the matching algorithm was improved upon from previous implementations by the implementation of an additional comparator to distinguish between directional comparisons and regular comparisons, as well as modifications to certain edge types. The current implementation still lacks the ability to identify some variations masked by nested matches within duplications, meaning this objective has only been partly achieved.

4. **Research existing sequence alignment visualisation tools in order to ascertain their advantage and disadvantages, and the common features expected from this type of tool.**
   Achieved. Research was carried out into three different tools: ACT, Mauve and BRIG.

5. **Design and implement a GUI to display (1) and the results of (3), along with suitable features identified in (4).**
   Achieved. The implementation of the GUI incorporates all the features that were laid out in the design section.

Chapter 7: Conclusion

When assessing the results of the project against its overall aim and objectives, this project demonstrates that it is possible to utilise subgraph isomorphism to detect rearrangements features with a high degree of accuracy. While certain minor problems remain in regards to the graphical representation and in particular the identification of variations, the system developed is able to accurately identify the vast majority of rearrangement features and display them to the user in a clear and easily understandable format.

Looking back, one particular aspect of the project I enjoyed was it allowed me to apply the knowledge I gained during my undergraduate studies in biochemistry, and in particular to see a small part what goes into the creation of the bioinformatics tools that I used during that time. It was also both challenging and engaging attempting to continue the work carried out by others, particularly in regards to adapting and developing upon another person's code.

Overall, this project once again shows the potential for this technique to be a viable option in the study of comparative genomics, and provides the base upon which could be developed another effective bioinformatics tool.

7.1 The Future

There are still many potential avenues for the extension and improvement of the system that was implemented during this project, some of which are outline below.

- Adapt the system to be able to parse other formats of alignment data, such as MUMmer or MSPcrunch.
- Increase the range of parameters that users can use to filter alignment data that has been input.
- Allow users to perform a BLAST alignment through the system by incorporating the BLAST API, preventing the need for alignment data to be produced beforehand.
- Produce a web-based version of the system.
- Investigate ways to extend the existing graphical representation to represent multiple sequence alignments.
- Extend the system to be able to take in sequence feature files, such as the GenBank format, and display the features alongside the graphical alignment representation in a similar way to ACT or Mauve.

References

[1] M. Land, L. Hauser, S.-R. Jun, I. Nookaew, M. R. Leuze, T.-H. Ahn, T. Karpinets, O. Lund, G. Kora, T. Wassenaar, S. Poudel and D. W. Ussery, "Insights from 20 years of bacterial genome sequencing," *Funct Integr Genomics,* vol. 15, no. 2, pp. 141-161, 2015.

[2] "https://ghr.nlm.nih.gov/primer/basics/dna," [Online].

[3] J. Venter, "The sequence of the human genome.," *Science,* vol. 291, no. 5507, pp. 1304-51, 2001.

[4] X. Xia, Comparative Genomics, 2013.

[5] J. Alföldi and K. Lindblad-Toh, "Comparative genomics as a tool to understand evolution and disease," *Genome Research,* vol. 23, pp. 1063-1068, 2013.

[6] K. Lindblad-Toh, C. M. Wade, T. S. Mikkelsen, E. K. Karlsson and e. al, "Genome sequence, comparative analysis and haplotype structure of the domestic dog," *Nature,* vol. 438, pp. 803-19, 2005.

[7] G. Bejerano, M. Pheasant, I. Makunin, S. Stephen, W. J. Kent, J. S. Mattick and D. Haussler, "Ultraconserved Elements in the Human Genome," *Science,* vol. 304, no. 5675, pp. 1321-1325, 2004.

[8] F. C. Odds, "Genomics, molecular targets and the discovery of antifungal drugs.," *Rev Iberoam Micol.,* vol. 22, no. 4, pp. 229-37, 2005.

[9] N. Sandford, "Computational classification of genomic rearrangements using a graphical representation," 2017.

[10] S. B. Needleman and C. D. Wunsch, "A general method applicable to the search for similarities in the amino acid sequence of two proteins," *Journal of Molecular Biology,* vol. 48, no. 3, pp. 443-453, 1970.

[11] T. F. Smith and M. S. Waterman, "Identification of Common Molecular Subsequences," *Journal of Molecular Biology,* vol. 147, pp. 195-197, 1981.

[12] S. Altschul, W. Gish, W. Miller, E. Myers and D. Lipman, "Basic local alignment search tool.," *J Mol Biol.,* vol. 215, no. 3, pp. 403-10, 1990.

[13] T. Madden, The NCBI Handbook [Internet]. 2nd edition., NCBI, 2013.

[14] D. J. Edwards and K. E. Holt, "Beginner's guide to comparative bacterial genome analysis using next-generation sequence data," *Microb Inform Exp,* vol. 3, p. Published online, 2013.

[15] A. C. Darling, B. Mau, F. R. Blattner and N. T. Perna, “Mauve: Multiple Alignment of Conserved Genomic Sequence With Rearrangements,” *Genome Research,* vol. 14, pp. 1394-1403, 2004.

[16] N. Alikhan, N. Petty, N. B. Zakour and S. Beatson, “BLAST Ring Image Generator (BRIG): simple prokaryote genome comparisons,” *BMC Genomics,* vol. 12, p. 402, 2011.

[17] M. Garey and D. Johnson, Computers and Intractability: A Guide to the Theory of NP-Completeness, W. H. Freeman & Co., 1979.

[18] J. R. Ullmann, “An Algorithm for Subgraph Isomorphism,” *Journal of the Assocmtlon for Computing Machinery,* vol. 23, no. 1, pp. 31-42, 1976.

[19] L. Cordella, P. Foggia, C. Sansone and M. Vento, “A (sub)graph isomorphism algorithm for matching large graphs,” *IEEE Transactions on Pattern Analysis and Machine Intelligence,* vol. 26, no. 10, pp. 1367-1372, 2004.

[20] L. Cordella, P. Foggia, C. Sansone and M. Vento, “Performance evaluation of the VF graph matching algorithm,” *Conference: Image Analysis and Processing,* 1999.

[21] V. Bonnici, R. Giugno, A. Pulvirenti, D. Shasha and A. Ferro, “A subgraph isomorphism algorithm and its application to biochemical data,” *BMC Bioinformatics,* vol. 14, p. Published online, 2013.

[22] R. Milo, S. Shen-Orr, S. Itzkovitz, N. Kashtan, D. Chklovskii and U. Alon, “Network Motifs: Simple Building Blocks of Complex Networks,” *Science,* vol. 298, no. 5594, pp. 824-827, 2002.

[23] N. W. Lemons, B. Hu and W. S. Hlavacek, “Hierarchical graphs for rule-based modeling of biochemical systems,” *BMC Bioinformatics,* vol. 12, p. 45, 2011.

[24] E. Mamalaki, “Automated Identification of Sequence Rearrangements in Comparative Genomics,” 2013.

Appendices

Artificial Data

Displayed below is the complete BLAST hit table for the artificial data used for testing in section 5.1.

1	seq|Newman_Test	seq|MRSA_Test	100.0	5	0	0	1	5	1	5	0	0
2	seq|Newman_Test	seq|MRSA_Test	99.32	5	0	0	8	12	5	10	0	0
3	seq|Newman_Test	seq|MRSA_Test	98.21	5	0	0	12	17	12	17	0	0
4	seq|Newman_Test	seq|MRSA_Test	95.38	5	0	0	20	25	25	20	0	0
5	seq|Newman_Test	seq|MRSA_Test	90.57	5	0	0	30	25	25	30	0	0
6	seq|Newman_Test	seq|MRSA_Test	88.5	5	0	0	35	40	35	40	0	0
7	seq|Newman_Test	seq|MRSA_Test	87.9	5	0	0	35	40	45	50	0	0
8	seq|Newman_Test	seq|MRSA_Test	92.4	5	0	0	35	40	55	60	0	0
9	seq|Newman_Test	seq|MRSA_Test	100.0	5	0	0	60	65	80	85	0	0
10	seq|Newman_Test	seq|MRSA_Test	86.2	5	0	0	70	75	80	85	0	0
11	seq|Newman_Test	seq|MRSA_Test	98.0	5	0	0	80	85	80	85	0	0
12	seq|Newman_Test	seq|MRSA_Test	96.2	5	0	0	90	95	90	95	0	0
13	seq|Newman_Test	seq|MRSA_Test	90.3	5	0	0	90	95	110	115	0	0
14	seq|Newman_Test	seq|MRSA_Test	85.32	5	0	0	100	105	100	105	0	0
15	seq|Newman_Test	seq|MRSA_Test	85.32	5	0	0	120	125	120	125	0	0
16	seq|Newman_Test	seq|MRSA_Test	85.32	5	0	0	140	145	120	125	0	0
17	seq|Newman_Test	seq|MRSA_Test	85.32	5	0	0	130	135	130	135	0	0