Structural Comparisons for Small Molecules

Mary McDowall

School of Computing Science
Sir Alwyn Williams Building
University of Glasgow
G12 8QQ

A dissertation presented in part fulfilment of the requirements of the Degree of Master of Science at The University of Glasgow

7th September 2015

Abstract

This project set out to create a software application which can be used to compare the structures of molecules. The method that was chosen was put forward by Marcus Ludwig *et al.* [17] and aimed to create a structure which is representative of the common structural features found within the molecules.

Alongside the presentation of their paper, the authors created a Java based implementation of their method. This was in the form of a plugin for the SIRIUS mass spectrometry framework [2]. However the most recent release of the framework does not include the plugin. This restricts the usability of this application, particularly for those using an alternative framework.

This project therefore attempts to create a Python based implementation of the method in [17]. The results received from this application, when tested with a specific set of molecules, were compared against the results received from the original application.

The application includes a basic parser which converts the SMILES chemical notation into a graph data structure. It also offers an alternative means of molecule comparison which can provide further information to the user about the structure of the molecules.

The algorithm has been implemented in full and, though there remains improvements that could be made, the current evaluation results appear mostly positive.

Education Use Consent

I hereby give my permission for this project to be shown to other University of Glasgow students and to be distributed in an electronic format. **Please note that you are under no obligation to sign this declaration, but doing so would help future students.**

Name: Mary McDowall Signature: [illegible]

Acknowledgements

I would like to thank Dr. Alice Miller for her advice throughout the project and Dr. Simon Rogers for his help with the problem domain and his guidance, particularly at the outset of the project.

Contents

Chapter 1

Introduction

This project uses computing science techniques to build efficient and usable software to help Glasgow Polyomics. Glasgow Polyomics, a research facility attached to the University of Glasgow, performs experiments and analysis in a number of different areas of biochemistry. Glasgow Polyomics seeks to combine the study and data from four separate fields: genomics, which is the study of all of the genetic material in an organism, transriptomics, which focuses on the RNA molecules found in cells, proteomics, which studies the proteins in a cell or organism, and metabolomics, which analyses the metabolites found in cells. Glasgow Polyomics sets out to combine the data and analysis from these different fields and so increase the understanding of the chosen organism [19].

A metabolite is an organic chemical structure which is involved in the chemical reactions within a cell or organism. They come in a huge variety of structures and types of molecule and are shared throughout different species. Increasing the knowledge of a living organism at the molecular level enables the effects of changes within it be clearly understood. This is of key importance when it comes to understanding the effects of diseases and to inform the development of ways to combat them [14]. This understanding is also important when experimenting with these potential therapeutic treatments; not only to see if the treatment has the desired effect but also to ensure that it does make dangerous changes to other parts of the system [16]. Metabolites are particularly effective for this type of experimentation as they have the benefit of being very sensitive to external changes.

One of the key techniques used in Glasgow Polyomics and in metabolomics in general is *mass spectrometry* (MS). MS is used to analyse the mass-to-charge ratio of different components that make up a sample and determine the abundance of that component within the sample [9]. In order to do this, the sample is broken apart into separate components using a number of different techniques, such as bombarding it with electrons which causes the fragments of the sample to become charged. It is then necessary to find out the quantity of each of these fragments of the sample by analysing the charged fragments. This produces a *fragmentation pattern* which is a graph displaying a series of peaks for each of the fragments of the sample. As each of these peaks signifies a different fragment, the next stage is to then identify the molecules which corresponds to each peaks.

When MS is used for a biological sample, each of the peaks within a fragmentation pattern corresponds to a different metabolite. The identification of the metabolite is of key importance if it is to be useful in further analysis. To aid in this, there are a number of databases which contain the structural information and properties of the known metabolites. If a metabolite is found in a database which matches the mass-to-charge ratio of a given peak then, if the other features of the compo-

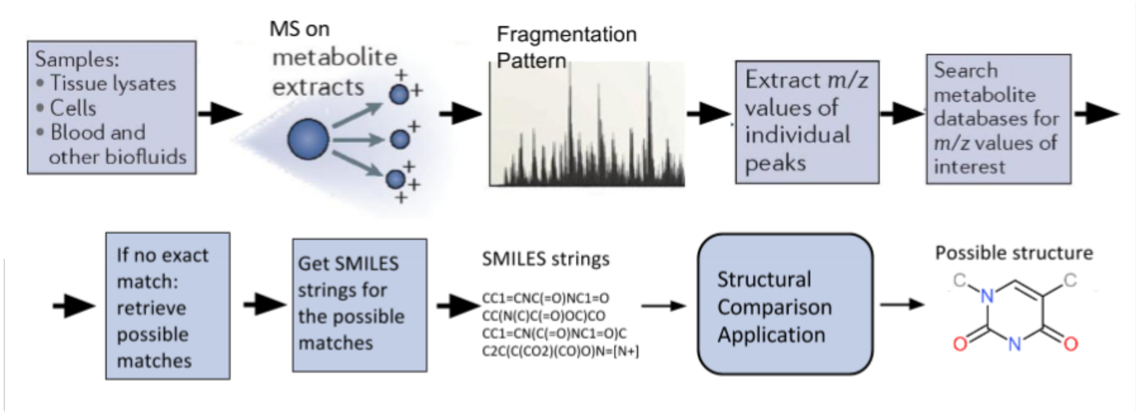

Samples:
• Tissue lysates
• Cells
• Blood and other biofluids
MS on metabolite extracts
Fragmentation Pattern
Extract m/z values of individual peaks
Search metabolite databases for m/z values of interest
If no exact match: retrieve possible matches
Get SMILES strings for the possible matches
SMILES strings
CC1=CNC(=O)NC1=O
CC(N(C)C(=O)OC)CO
CC1=CN(C(=O)NC1=O)C
C2C(C(CO2)(CO)O)N=[N+]
Structural Comparison Application
Possible structure
C
N
C
O
N
O

Figure 1.1: The steps involved in identifying an unknown metabolite from an MS fragmentation pattern when no match is found in the databases of metabolites

nent match with the metabolite, then that peak has been successfully identified and its structure and attributes are known. However, if no exact match can be found, then instead a list of potential matches will be retrieved. It then becomes a problem of comparing these potential matches so that the correct metabolite can be identified [18].

In their paper [17], Ludwig *et al.* present a technique which, when given a set of candidate molecules is able to uncover, at least partially the structure of the unknown molecule. This project sets out to develop an application which will be based upon the approach of Ludwig *et al.* that Glasgow Polyomics will be able to integrate into their current pipeline so that it will be possible for them to discover features of unknown metabolites and hopefully come closer to identifying them.

This dissertation will provide background information on relevant topics as well as studying the chosen method of comparing the structure of molecules. It will set the requirements of the application which includes what should be achieved by the application. The design decisions that were made and the methods used to implement the algorithm will be discussed. The application was tested for correctness by comparing the graphs it produced with those presented in the chosen paper and these results will be examined to help evaluate the success of the project.

Chapter 2

Background

Before exploring the chosen paper in detail, a number of different topics have to be explored. These include the basics of the graph data structure and molecule graphs. It was agreed with the client that the molecules that would be passed to the program would be using the SMILES notation so a thorough understanding of this was required.

2.1 Graph Theory

A graph is a set of vertices (or nodes) connected by edges and is used in many different disciplines. Within computing science a graph can be used to represent a computer network or a telecommunications network while in sociology a graph can be used to describe a social network. It has also been found in linguistics that graphs are a useful way of modelling the structure of a natural language. In biology and chemistry graphs have been used to help study molecules by providing a representation of the molecule structure.

The definitions provided within this section can be found in [11]

2.1.1 Graph Data Structures

For each graph G there is a list of vertices V and a list of edges E. A vertex v is *adjacent* to a vertex w if there is an edge between them while the *degree* of a vertex is the number of vertices adjacent to it. Each vertex and edge within the graph can have a *label* prescribed to it which signifies properties of the edge or vertex. An edge can also have a *weight* which is usually a real number. For a graph representation of a road network the weight of an edge could be used to signify the length of a road.

A graph is *directed* if for each edge one of the vertices one is designated the *origin* while the other is chosen as the *destination*. If a graph is *undirected* then the edges have no specified direction.

A *path* in a graph is a sequence of vertices where no vertex is visited more than once. For a path between two vertices, its length is the number of edges which are traversed. A *cycle* is a path which returns to the first vertex of the path similar to closing a circle.

A *subgraph* of a graph is a smaller graph such that its vertices and edges can be contained within the larger graph. For a set W which contains some of the vertices of a graph it is possible to create an *induced subgraph*. This induced subgraph will contain all of the vertices in W as well as any edges which have endpoints in W.

There are a number of ways to represent a graph and the relationships between its vertices. One way is to use an *adjacency matrix* where each of the vertices is a row and a column in the matrix and there will be either a 1 present at (v, w) if v and w are adjacent and a 0 if they are not. Another way in which a graph can be represented is by using an *adjacency list*. This consists of an array of lists, each one attributed to one of the vertices of the graph. Each vertex's list contains all the vertices which are adjacent to it. When choosing which representation of a graph will be used it is best to consider the *density* of the graphs in question. A graph is considered *dense* if the number of edges in the graph is close to $|V|^2$ where V is the number of vertices in the graph. If the number of edges is much less than $|V|^2$ then the graph is considered to be *sparse*. An adjacency list is more efficient in terms of memory, particularly for a sparse graph and it also has the benefit of more efficient running times for methods adding a vertex and retrieving the adjacent vertices [10]. Choosing to use the adjacency list representation also helps to ensure that searching and traversing the graph is more efficient.

There are two main algorithms which relate to the traversal of a graph. These are *depth first search* and *breadth first search*. The depth first search starts at one vertex and travels to the end of a path before proceeding to the next vertex which is adjacent to the starting vertex. In this way the search will proceed as deep into the graph as it can before altering its course. The breadth first search instead focuses on visiting each of the vertices which are adjacent to the starting vertex before proceeding to the subsequent 'layer.' This search can be thought to radiate out from the starting vertex in steps

2.1.2 Molecule Graphs

When discussing the structure and composition of molecules, structural formula are often used to display the atoms which are involved and the ways in which they are bonded together. A *molecule graph* is a graph structure where the vertices are used to represent the atoms of a molecule while the edges are used to represent the bonds. These graphs are undirected and each of the vertices is labelled with their atom element and each edge is labelled according to the bond type, whether it is single, double, triple or aromatic. This is demonstrated in Figure 2.1 which shows the molecule graph of aspirin. Within these graphs it is usually chosen that the hydrogen atoms will not be displayed, instead it is assumed that each of the other atoms is bonded to enough hydrogen atoms so that their degree matches their lowest normal *valence* which is number of bonds which the atom can form with hydrogen atoms [3]. This number differs between elements with carbon able to form 4 separate hydrogen bonds while oxygen can only form 2 bonds with separate hydrogen atoms.

2.1.3 Graph and Subgraph Isomorphism

An important problem which many algorithms attempt to solve is that of *isomorphism*. If two objects are isomorphic they can be thought of as structurally the same within the rules for those objects. If two graphs are found to be isomorphic then they can be thought of as being the same in terms of structure and relationships between the vertices. Within graph theory, isomorphism has

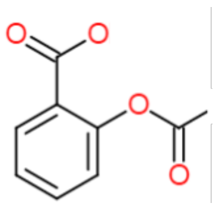


Figure 2.1: The molecule graph of aspirin (acetyl salicylic acid). Unlabelled vertices represent carbon atoms

two forms which are closely related. The *graph isomorphism* problem is concerned with the direct comparison of two graphs whereas an algorithm for *subgraph isomorphism* problem will look to find if one graph can be considered a subgraph of a second graph. For the purposes of this project the main focus will be on whole graph isomorphism. Formally, an isomorphism between two graphs G and H means that there is a bijection $\phi : V_G \rightarrow V_H$ such that for any vertices u, v in V_G where there is an edge (u, v) joining them there must be an edge $(\phi(u), \phi(v))$ in H.

There are a number of different algorithms which have been created to solve this problem. These include LAD-filtering, VF algorithm and the improved VF2 algorithm. For further details on these algorithms see [8][4]. There are a number of applications which implement these algorithms such as LAD and VFlib which are written in C and C++ respectively. When considering which application to choose it was important to find out if the algorithm considered the labels of the vertices and edges.

This consideration meant that the LAD application, which provides favourable results even when compared to the VF2 algorithm was not feasible. Though it provided matching for labelled graphs, those graphs had to also be directed. As the molecule graphs are undirected this would not provide an accurate comparison of the molecules; if there was a cycle present in the molecule graph then if the vertices were encountered in a different order in two graphs then they would be considered different when structurally they are the same.

2.1.4 Maximum Common Subgraph Problem

In order to gain insight into the features of an unknown molecule it is possible to compare its molecule graph with the molecule graphs of other molecules. One of the ways in which this can be done is to find the largest subgraph which can be found in all of the molecule graphs. This is known as the *maximum common subgraph problem*. In molecules which are very similar this can produce very illuminating information, as this common structure can help to show the properties which the molecules have in common due to the similar-structure, similar-property principle. However the solutions for this problem are very strict in that they require that all of the molecules contain the entirety of the maximum common subgraph [20]. This can mean that if a collection of molecules is quite heterogeneous then the maximum common subgraph can be very small [17].

2.2 Simplified Molecular-Input Line-Entry System

Storing molecular structures as merely images in a database would add to the difficulty of searching and comparing between molecules. For this reason the Simplified Molecular-Input Line-Entry System or SMILES notation is used which preserves the information presented within the structural formula while also being in a compact and searchable format.

The SMILES notation is used to display molecules as strings which contain the information on the atoms in the molecule and the bonds that are between them. This method of notation was first put forward by David Weininger in [22]. This set out the basic premise for the chemical representation. The SMILES notation has been maintained and improved by the Daylight Chemical Information Systems, Inc [6]. In order to provide a non-proprietary specification for the SMILES notation the Blue Obelisk project has set about creating OpenSMILES [15] which details the rules for the language and demonstrates variations of SMILES strings that are acceptable.

Within the SMILES notation each atom in a molecule is represented by its atomic symbol. If the atom is from the *organic set* [B, C, N, O, P, S, F, Cl, Br, I, *] (where * is an unknown atom) then it can be simply written as its atomic symbol. For these organic atoms the attached hydrogen atoms are not explicitly stated, instead it is assumed that there are enough hydrogen atoms attached to make up the lowest normal valence of the element, such as 4 covalent bonds for carbon or 2 covalent bonds for oxygen.

If an atom does not belong to the organic set or does not have enough associated hydrogen atoms to meet the normal valence number then it is put inside square brackets and the number of hydrogen atoms must be explicitly stated. This hydrogen count follows after the atomic symbol so [CH2] is a carbon atom with two attached hydrogen atoms. The charge of an atom is also placed inside the brackets if it is different from normal so [CH2+2] is a carbon atom which has two attached hydrogen atoms and a +2 charge. If the atom within the molecule is an isotope then the atomic mass is put before the atomic symbol within the brackets so [12C] and [13C] are specific carbon isotopes whereas [C] is an unspecified isotope. When taken together the information within the square brackets takes the form of:

[Isotope Number Atomic Symbol Hydrogen Count Charge]

When representing single bonds within a SMILES string either a –or simply the concatenation of atoms can be used. To represent double, triple and quadruple bonds the symbols =, #, $ are used respectively between the atoms. To demonstrate branches within a molecule the bonds and the atoms are placed inside round brackets. This means that the structure in Figure 2.2 could be written as CCC(CC)CO or as CCC(CO)CC.

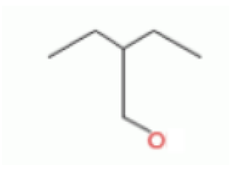


Figure 2.2: A molecule with a branch

If there is a branch coming off of another branch then it is possible to nest the parentheses within each other. The nesting of branches can be performed indefinitely. If there is however a compound in which two molecules are disconnected then this is demonstrated by a period between the atoms from each of the molecules.

If there is a ring present in the molecule then a bond is chosen within the ring which will be 'broken' so that it can be written as a linear string. The two elements which have been disconnected are each marked with a digit after the atomic symbol. So the ring in Figure 2.3 could be written as C1CNCNC1.

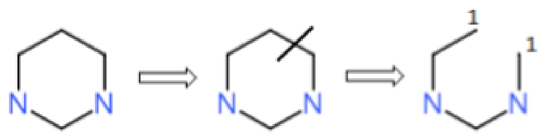


Figure 2.3: A ring which has one bond broken so it can be written linearly

If the ring structure within the molecule is an aromatic ring then there two different ways that it can be represented in SMILES. The ring can either be written with normal atomic symbols and alternating single and double bonds between the atoms or the atomic symbols can be written in the lower case without assigning any particular double or single bonds. The latter of these two options is preferred as it does not require an arbitrary choice to be made in the placement of double bonds.

An atom can be present in multiple distinct ring closures, which is demonstrated by having multiple numbers following the atomic symbol, each number reflecting a different ring closure.

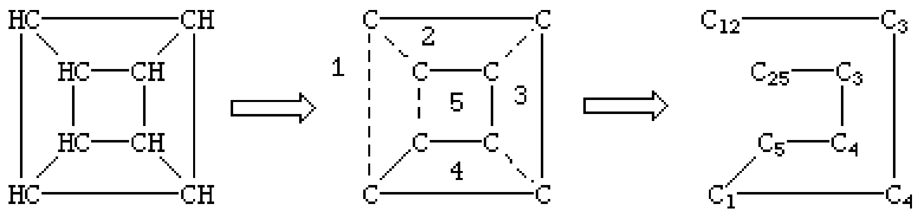


Figure 2.4: Multiple ring closures [6]

Chapter 3

Finding Characteristic Substructures for Metabolite Classes

Ludwig *et al.* present an alternative method for finding common structural properties of a set of molecules which is an alternative to the maximum common subgraph. Their method consists of creating a structure, known as the *characteristic substructure*, which is a combination of the structures which appear most frequently within the molecules. This method is able to find a much larger structure than the maximum common subgraph which can therefore offer more information about the molecules in general. This paper offers an interesting means of comparison for molecules though it has not been studied in depth in the wider metabolomics community.

The basic steps of the algorithm that the authors present involves first finding the paths within the molecules and then, using these paths, inducing subgraphs of the molecules. The subgraphs which appear frequently in the molecules are added together to form the characteristic substructure. These steps can be seen in Figure 3.1 and Algorithm 1.

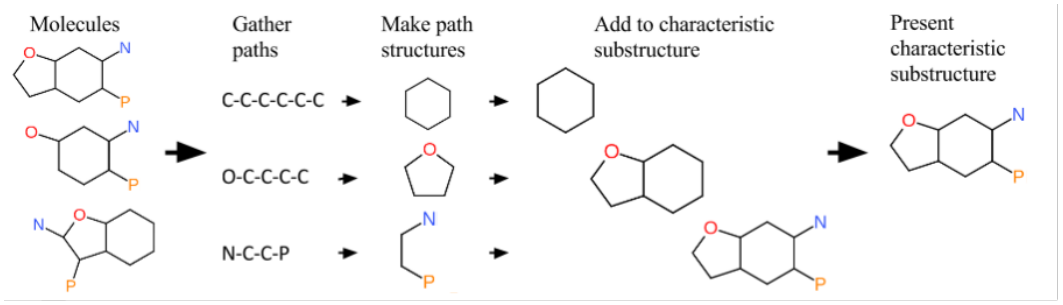


Figure 3.1: First the paths which appear frequently are found, then path structures which induced from these paths are made. The path structures which appear with a high enough frequency in the molecules are added to the characteristic substructure.

The initial parameters of the algorithm include the maximum and minimum length of path which should be considered. The algorithm begins by considering the maximum length and then decrements until the minimum length is reached. For each length, the paths within the molecules of that length are found. The frequency of these paths will be compared to a threshold which is also set in the parameters. The suggested threshold is 0.8 which means that if a path appears in at least 80%

of the molecules then it can be considered *representative*.

Once the representative paths for a certain length have been found, the next step is to find all of the subgraphs which can be induced by these representative paths. Within the paper, these subgraphs are called *path structures*. The frequency of these path structures within the molecules is also compared against the threshold and those that are high enough are designated as *representative path structures*. If there are no path structures which are representative then the length of the paths to be considered is decremented by 1 and the process begins again for the shorter length.

If there are representative path structures then they are sorted into frequency order with the most frequently appearing path structure coming first. These structures are then added to the characteristic substructure. If no characteristic substructure has yet been created then the most frequent path structure is used as the base for the characteristic substructure. The molecules which contain a subgraph isomorphic to this path structure are remembered.

When adding a path structure to the characteristic substructure it is important to consider how the location of this newest path structure relates to the other path structures in the molecules which have already been added to the characteristic substructure. The different locations where the new path structure could be added to the characteristic substructure are compared in terms of their frequency within the molecules. The location of the path structure in the characteristic substructure which is most frequent is chosen and the molecules which contain this newly added section of the characteristic substructure are remembered.

This process continues until all of the representative path structures, induced by representative paths of the current length, are added to the characteristic substructure. Following this the length of path which is to be considered is decreased by a larger step so that only path structures which add more useful information are considered.

Algorithm 1 Create the characteristic substructure for a set of molecules

```
Let $l$ be a certain length and let $l_{start}$ and $l_{end}$ be preset integers
while $l_{start} \geq l > l_{end}$ do
    find all the paths of length $l$ in the set of molecules
    find the paths of length $l$ which are representative
    find all the path structures which can be induced by these representative paths
    if there are path structures which are representative: then
        sort the representative path structures into order of their relative frequency
        if there is no characteristic substructure then
            the most frequent path structure becomes the characteristic substructure
            remember the location of the path structure within the molecules
        else there is a characteristic substructure:
            compare the location of the path structure within the molecules
            find the path structures most frequent location
            add the path structure to the characteristic substructure at this location
            remember the location of the path structure within the molecule
            decrease $l$ by a preset step
    else there are no representative path structures
        decrease $l$ by 1
```

When considering path structures which appear multiple times in the same molecule Ludwig *et al.* state that the multiple copies of the path structure should be considered as one whole structured, connected (or disconnected) as they are within the molecules, to ensure that no information is lost. When considering where to add these path structures into the characteristic substructure these larger path structures are treated in a similar manner to the representative path structures when they are being added to the characteristic substructure.

Algorithm 2 Adding multiple path structures to the characteristic substructure

```
Let $P$ be a path structure which can be found multiple times in the same molecule
if $P$ appears in a preset percentage of the set of all the molecules then
    for $k$, where $k$ is an integer $\geq 1$ do
        for each molecule that contains $k$ copies of $P$ do
            create a graph which contains $k$ copies of $P$ as they can be found in the molecule
        compare the frequency of these graphs within the molecules
        the most frequent graph is added to the characteristic substructure
```

This method ensures that there is not a choice being made between the different positions of the path structure within the molecule and instead consider them all together.

3.1 MoleculePuzzle

As part of this process Ludwig *et al.* implemented a plugin called MoleculePuzzle. This was written in Java for SIRIUS which is a system that handles the results from mass spectrometry. MoleculePuzzle uses the algorithm that is presented but focuses primarily on allowing the user to modify and create their own structures. As this was created as a plugin for a larger application, it means that it is not possible to use the MoleculePuzzle application within another pipeline without heavy modification to the source code. This plugin has also not been included in the newest release of the SIRIUS pipeline. This means that in order to use the plugin within SIRIUS the user would have to use an older version which lacks the improvements and features of the newer version.

Chapter 4

Requirements

The requirements for a software development project set down what the project must do and sets constraints on the system. They are agreed upon by the developer and the client. Through interviews with Dr Simon Rogers, who acted as liaison for Glasgow Polyomics within the university, the main requirements of the application were gathered. The requirements are presented in full in Appendix A.

One of the main requirements of the project was that the information about the molecules would be entered using a file of SMILES strings. This necessitates the use of a *SMILES parser*. To *parse* a linear representation is to create a structure representing it with the use of a set of grammar rules [13]. For this situation a SMILES parser would take in a SMILES string and then, using the grammar rules described in Section 2.2, turn it into a graph data structure.

There is one platform requirement which specifies that the application should either be written in Python or R so that it can be easily integrated into the clients system. There are a number of benefits of choosing to write the application in Python. Within the scientific community there has been a strong rise in popularity for Python as the programming language of choice. This is partly due to the wide library of resources which include many packages such as NumPy and SciPy which are tailored directly to scientific programming and MatPlotLib which is a graphing library. Another benefit of Python is that it is quite easy to integrate it alongside other languages such as R which is used for a lot of analytical scientific programming and C which can be used to speed up intensive tasks.

By choosing to program this application in Python it means that, if it is desired, it can be easily integrated into a Python pipeline or a larger R programme. Many of the packages that are available in Python make working with graphs far easier such as NetworkX which is an package that can be used to create and compare graphs. As Python is also noted for its human readability it ensures that if non-programmers wish to look at the code to understand the algorithm that was used the syntax of the language will not stand in the way of understanding.

To verify the requirement that the application correctly implements the algorithm detailed in Chapter 3, the results of the application will be compared with the results which the authors received when running a known set of molecules. A discussion of this comparison is carried out in Chapter 7. So that the results presented in [17] can be replicated, care will be taken to follow the method set down to calculate the characteristic substructure as closely as possible. If the application is able to produce graphs which bear a close resemblance to those provided then it ensures that the application is reliable as a means of comparison for molecules.

One of the quality requirements for the application set a desired response time for the application. The response time for the application while performing the evaluation on the chosen sets of molecules will be recorded alongside the graphs that are obtained. To improve the response time of the application there will be a focus during the design process to choose efficient ways to implement the algorithm and the graph data structure.

The use cases presented in Appendix A help to demonstrate the ways in which user will interact with the application. The design of the user interaction will aim to make the carrying out of these use cases as simple as possible. Careful consideration will be given to the way in which the application will be used when designing how best to interact with the user and present the results.

Chapter 5

Design

The project could be separated into three distinct sections: the molecule graph representation, the parsing of the SMILES strings and the implementation of the characteristic substructure algorithm. An overall structure was designed, as shown in Figure 5.1 with the three sections kept separate. The in depth design decisions were performed in stages, as many decisions on a component could not be made until the previous component had been implemented.

While designing the overall structure a focus was given to separating out the concerns. This means that each class deals with a different area of the application, allowing these smaller sections to be more easily understood.

5.1 Representing Graphs and Molecules

The design of the graph representation was done in such a way that if an alternative implementation was required it would be possible to alter this class without requiring alteration to the less abstract classes. When considering how best to do this sources such as [10] and [1] were consulted. The Graph Abstract Data Type described on [10, p. 626] has separate classes for the vertices and the edges and the graph object itself. This allowed the vertex and edge objects to be relatively simple and ensured that majority of the logic for the graph data structure was contained inside the one data structure in the graph object. Alternative designs for a graph abstract data type involved having more logic included within the vertex and edge objects. However this created more coupling within the code as data had to be sent back and forth between objects. It was therefore decided that the former structure would form the basis of the design for the graph data structure.

When considering where to store the information about molecule graphs there were two distinct options. The molecular information could be included in the graph class while the atom and bond information could be included in the vertex and edge classes respectively, or instead there could be distinct molecule, atom and bond classes which held their respective information and inherited the properties of the graph, vertex and bond classes. It was decided that the first structure would limit flexibility as if changes were to be made to the graph data structure this would also involve altering the storage of the molecular data. So the the graph data structure and the molecule graph information would be kept separately.

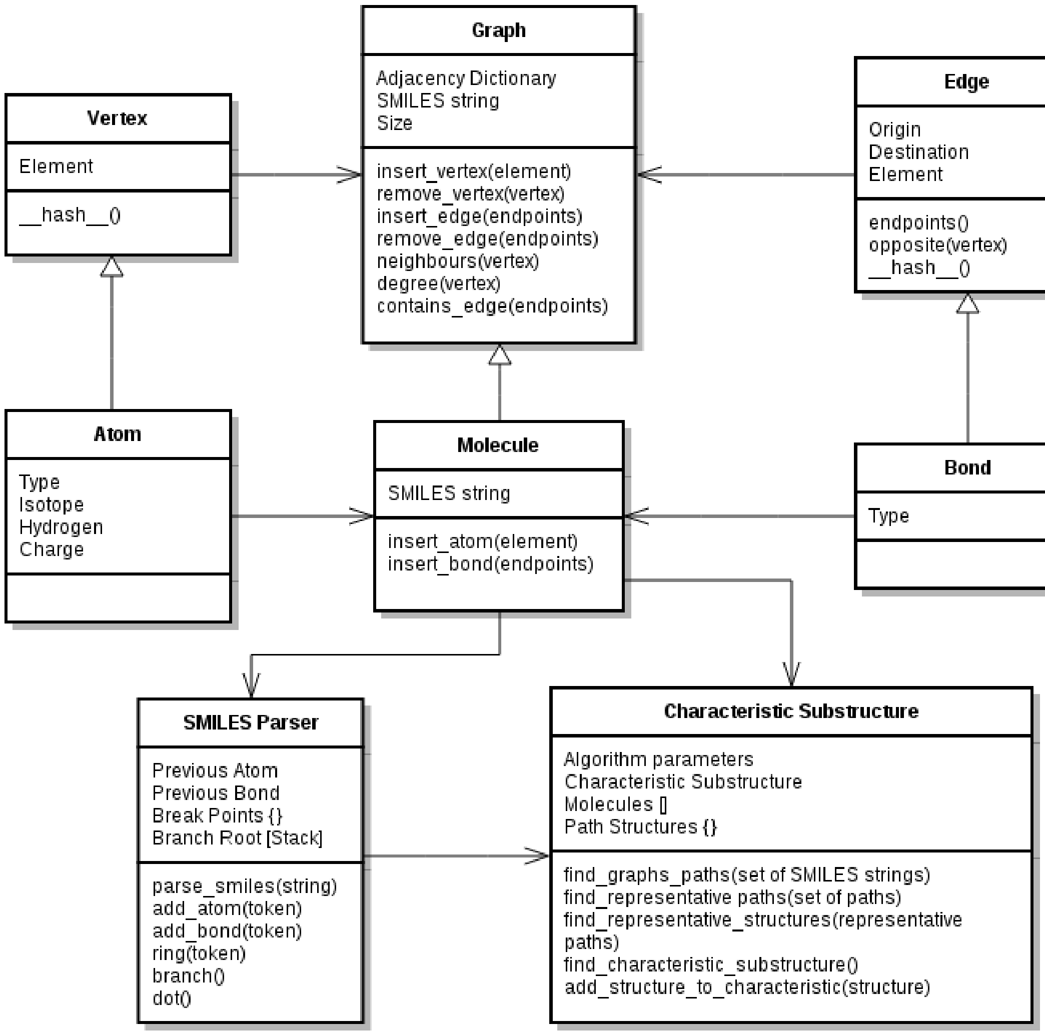


Figure 5.1: Preliminary Class Design

This separation of concerns had the advantage of allowing relative freedom when it came to designing the atom and bond classes. When taking in information from the SMILES string an atom could be either a normal atom or an atom which was part of an aromatic ring. For the bonds, they could be either single, double, triple, quadruple or aromatic bonds. One method of demonstrating the difference in the type of atom and bond would be by having a distinct class for each variant of the atoms and bonds. Alternatively, one could have a single atom and a single bond class with parameters indicating the type of atom or bond that it is. It was decided that this second option would produce more concise code which avoided repetition thus increasing the codes maintainability.

5.2 Inputting the SMILES Strings

The next component of the application to be designed in full was the method for taking in the SMILES strings and creating a graph structure based on them. External applications were considered to perform this task. There are a number of chemical toolkits which provide language conversion features including SMILES parsers.

The RDKit and OpenBabel are both chemoinformatics toolkits which are predominantly written in C++ with Python wrappers. The parsers which are contained within these toolkits, while being well tested are quite large (when compared to the other SMILES parser which will be considered) as each possible scenario is handled individually and focuses on providing more information to the molecule graph than the algorithm in [17] requires.

Frowns is another chemoinformatics toolkit which is written almost entirely in Python. The SMILES parser implements an alternative method for parsing the SMILES string where the string is split first into identifiable sections based on the type of SMILES element that it is. For each type of element a different operation is performed on it which creates quite a concise parser as there are a limited number of elements within the SMILES notation. However this toolkit is no longer actively supported.

When considering the benefits and drawbacks for each of the external SMILES parsers, and when considering the simplicity of the Frowns SMILES parser, it was decided that the creation of a specific parser for this application would be feasible. The Frowns parser would form the basis for the design of the parser, with the SMILES string broken apart using a *regular expression*. A regular expression can be thought of as a pattern which the given string is tested against to see if it matches. A regular expression can also be used to split a string apart based on the elements contained within it.

The Frowns parser uses a regular expression which breaks the SMILES string apart into separate components. Within Python it is possible to create groups of acceptable strings in the regular expression and provide them with names. This might take the form of all the possible bond symbols within SMILES which consists of –, =, $ and #. With these named groups it becomes possible to check what type of element each of the components is. In Listing 5.1 the regular expression which finds all of the bond symbols in a SMILES string can be seen. Each bond type belongs to a different named group.

```
'(?P<bond>(?P<single>-)|
          (?P<double>=)|
          (?P<triple>\#)|
          (?P<quadruple>\$)|
          (?P<aromatic_bond>\:))'
```

Listing 5.1: Regular expression to take in bond symbols with named groups.

Once this is done the parser calls will send the component to the appropriate method, depending on the group which it belongs to.

When designing the parser for the application, the idea of having simply functions contained in a source file that would be called when needed was examined. However when considering the different parameters and data structures each method would require access to it became clear that it would be better to have these functions as methods in a class where the object would contain the parameters and structures individual to it. During the design of the functions, it became clear that it would be necessary to hold a reference to the previous atom in the SMILES string when moving onto the next component as well as maintaining a note of where rings and branches began. For this reason it was decided that a parser object would be created which would hold this information.

5.3 Characteristic Substructure Algorithm

When originally designing this section of the application a similar problem was encountered as was found in the design of the SMILES parser; whether a set of functions would suffice or if a unique algorithm object would be required. As the algorithm requires the storage of the relationships between the molecules as well as the initial parameters that the user might set this component was designed as an algorithm class object.

The design of this class was focused around an intimate understanding of the algorithm that is presented in Chapter 3. When originally considering Algorithm 1 it was clear that the class would require a method for each of the stages of the algorithm that required finding particular data within the molecules whether that be paths, path structures or the location of the path structure. Data structures would also be required to hold the information about the characteristic substructure and the path structures as can be seen in Figure 5.1.

An iterative design process was used for this class. As each new section of the algorithm was implemented, the design would be reconsidered to see if the information was being stored in a suitable format. Also, as the complexity of each stage of the algorithm came to light during the implementation, more methods would be required to ensure that, if possible, each method carried out only one process.

Chapter 6

Implementation

As was found in the design phase of the project, the implementation of the application occurred in three stages. A focus was first given to creating a robust graph representation. The SMILES parser could not be implemented until there was a molecule class which could be used to test it. Similarly, the characteristic substructure algorithm could not be implemented until a method for taking in the molecule information had been created.

6.1 Graph and Molecule Graph

The implementation of the graph data structure followed on from the study in to graph theory and consideration of the many design alternatives that were available.

6.1.1 Graph as Adjacency Dictionary

As the metabolite graphs do not generally contain enough edges to be counted as dense it was decided that the adjacency list representation would be used. The Python *dictionary* allows a variation on the adjacency list representation. A dictionary is composed of key, value pairs where each key is unique. The Python dictionary is an example of a *hash table* which uses small integers as keys. This means that the keys of a dictionary can be of any type as long as they can be turned into small integers, called *hashing*. Any immutable object, such as strings or tuples and custom objects can be used as dictionary keys. With the high flexibility of the Python dictionaries it becomes possible to create an *adjacency dictionary* representation. This would consist of each vertex in the graph being a key in a dictionary, the value of which would be another dictionary. Within this inner dictionary each of the adjacent vertices would act as keys while the edge objects which joined them would be the values. Using the adjacency dictionary rather than the adjacency list allows access to an edge object in $O(1)$ time while ensuring that the insertion of vertices and edges maintains their $O(1)$ time [10]. The adjacency dictionary was included within the graph class along with the methods to add vertices and edges to it.

6.1.2 Depth First Search

As was stated in Section 2.1, a depth first search of a graph involves traveling to the end of a branch of the graph before return back to the starting node to travel down another branch.

Algorithm 3 Depth first search of a graph

```
function DFS
    for each of the vertices in the graph do
        set the vertex as unvisited
    for each of the vertices in the graph do
        if the vertex is unvisited then
            call the VISIT function on this vertex
function VISIT($v$)
    set the vertex $v$ as visiting
    for each vertex which is adjacent to vertex $v$ do
        if the adjacent vertex is unvisited then
            call the VISIT function on this adjacent vertex
    set the vertex $v$ as visited
```

As can be seen in Algorithm 3 the depth first search involves including a status flag for each of the vertices. This keeps a memory of which vertices have been visited during the search so that each segment of the graph is visited only once. It is important to note that the *visiting* and *visited* state of an element change on either side of the recursive call of the algorithm. This is known as the *parenthesis structure* of the algorithm. What this means is that if we were to write"$(v$" when we are visiting v and then"$v)$" when v has been visited then the visits to all the vertices would be properly nested in the parenthesis [5]. This parenthesis structure makes it possible to use this traversal method to document all the possible paths in a graph as is required by Algorithm 1. As can be seen in Algorithm 3. To utilise this parenthesis structure two *stacks* are used, one stack to recorded the path string itself and another to record the vertices which are involved in the path. A stack employs the last-in-first-out structure and and item is *pushed* to a stack if it is added to the end and *popping* retrieves the last element, removing it from the stack.

As we visit each vertex of graph the string representing its label is pushed onto one stack while the vertex object itself is pushed to the other stack. At the completion of each recursive call a copy of the two stacks is made. The elements are then popped from their respective stacks so that, when the depth first search travels down a subsequent branch, the vertices of the previous branch are not included in the path.

In order to draw all of the paths found within a graph, particularly a graph which is disconnected, each of the vertices of the graph must be used as the starting vertex of the depth first search in turn. Though this can lead to repetition of paths, it is necessary in order to get a complete set of paths.

6.2 SMILES Parser

When implementing the SMILES parser, careful consideration had be given to the best way in which to act when encountering each of the different symbols within the SMILES notation.

When an atom symbol is encountered a new vertex would be made and it would be stored as the previous atom. This is so that when the next atom is encountered an edge may be formed between the two vertex objects. This bond will be single if no bond symbols have been encountered between the two atoms.

However the presence of rings and branches complicate the simple addition of vertices and edges.

6.2.1 Branches in Molecules

As has been discussed in Section 2.2, within SMILES the start of a branch is signified by an open round bracket and the end of the branch by a closing round bracket. When starting a branch, the vertex which is at the base will have to be remembered so that it can be returned to once the branch has been completed. For a single, simple branch this would only require the vertex to be stored as a variable. However, within SMILES it is possible to nest branches as much as is desired. Therefore an alternative method of storage is required. A stack, as a last-in-first-out data structure allows the branch roots to be retrieved in the correct order. As an open bracket is encountered the atom directly previous to it is pushed to the queue. The parser then continues along the branch, adding atoms and bonds to the molecule graph as it encounters them. Once the branch is completed with a closing bracket the branch root is popped from the stack and the parser may continue with the neighbouring atom.

6.2.2 Rings in Molecules

A ring is shown, within SMILES, by a number after an atom or a bond. This number will appear next to a second atom further along the SMILES string and these two atoms should have a bond between them closing the ring. A dictionary is used in order to successfully match the closures of the rings. When a digit is first encountered it is used as the key for the dictionary. The value for this number is a tuple containing the vertex object and bond type that immediately proceeded the number. This then means that when the digits counterpart is encountered the vertex can be retrieved and a bond can be made between the two vertices of the type specified in the dictionary.

6.3 Characteristic Substructure Algorithm

A wide variety of data structures and algorithms were required to fully implemented the method put forward to find the characteristic substructure. Not only did the molecules and the paths that they contained have to be stored but also the relationships between the molecules and the path structures which these paths induced.

6.3.1 Dictionaries to Store Data in Algorithm

Dictionaries were used quite frequently during the implementation of the main algorithm. Due to the key, value nature of the dictionary it was useful for storing the relationships between objects. During the running of the algorithm it was necessary to remember which molecules contained copies of the path structures. This was so that the frequency of the appearances of the path structures could be checked. This took the form of a dictionary where the path structures are the keys and a list of the molecules that contained them were the values.

However in order to compare the locations of these path structures, prior to adding them to the characteristic substructure, it was necessary to note the vertices in the the molecules which mapped to the vertices of the path structure. This then took the form of a dictionary mapping the path structure vertices as keys and the molecule vertices as the values. This mapping dictionary was then stored as the value to the molecule object within the larger path structure dictionary. The entire nested dictionary then has, for each path structure, the molecules which contain it and for each of these molecules the dictionary containing the mapping from the path structure vertices to the molecule vertices.

6.3.2 Isomorphism of Graphs

There is a necessity at a number of points in the algorithm to compare whether two structures are isomorphic. This ability is required before accessing the frequency of the path structures in the molecules. In order to get an accurate count of the path structures, the list that is maintained must not contain any duplicate path structures. As each path structure is created, before adding it to the dictionary of path structures, it is compared to the the other structures in the dictionary. If the new structure is found to match another then the molecule and vertices mapping which is associated with it is added to the dictionary entry of the matching structure. This ensures that for each unique path structure there is a complete list of all the molecules that contain it.

To find the most frequent location of a structure when adding it to the characteristic substructure it was necessary to compare path structures. For each of the molecules, a graph object would be created which reflected the way in which the characteristic substructure and path structure connect in the molecule. To find out which of these graph objects reflects the most frequent location for the connection, each of the graph objects is compared to the others and if there are duplicates then the frequency of that matching location is increased.

As the different isomorphism algorithms are quite complex to implement well it was decided that an external implementation would be used. As the LAD application was not able to consider labelled undirected graphs, implementations of the VF2 algorithm were considered as it is regarded as a fast isomorphism algorithm [8].

VFLib and NetworkX both implement the VF2 algorithm, however as VFLib is written in C++ it would have to be recompiled for each platform that the application was to be used on. In comparison, NetworkX which is written in Python is quite simple to install using the pip package manager. As a there was a focus on portability it was decided that using the NetworkX VF2 implementation would be a better choice.

In order to use the NetworkX isomorphism algorithm, the molecules with the program had to be given as representation as a NetworkX graph. As each of the nodes in a NetworkX graph is identified by an integer which reflects the order that the nodes were added to the graph, a similar method of identifying the vertices of the application's graphs was required. Each vertex was therefore given a *position* property which reflected the number of vertices in the graph when this new vertex is added. Due to this addition of the position attribute, it became possible to use the mapping, which the NetworkX isomorphism algorithm returned connecting the tested graphs, to connect the molecules to the path structures that they contained.

6.3.3 Adding a Structure to the Characteristic Substructure

To know where to add a structure to the characteristic substructure, the location of the path structure within each of the molecules that contain it had to be considered. If a section of the molecule which is isomorphic to the path structure is already mapped to the characteristic substructure then it is unnecessary to add that part of the path structure to the characteristic substructure. To find out whether a path structure vertex v should be added to the characteristic substructure, a check is performed to see if the molecule vertex u, which the path structure vertex maps to, also maps to a vertex in the characteristic substructure. If u is not represented in the characteristic substructure then v will be added into the characteristic substructure adjacency dictionary. Whether a new vertex object has been or not, the neighbours of v will also be compared against the characteristic substructure. This is to determine if a new bond object needs to be made or not. If both vertex and neighbour already exist in the characteristic substructure then no new bond is required as it is already represented. If both the vertex and the neighbour will need to be added to the characteristic substructure then the bond object from the path structure can be added into the characteristic substructure adjacency dictionary. If one vertex is in the characteristic substructure and one is not, then a new bond object will have to be made which has these two vertex objects as its endpoints.

6.3.4 Remembering Location in a Molecule

Within the algorithm in [17] there is a need to remember the locations of subgraphs in molecules which are isomorphic to the characteristic substructure. An initial plan was that for any vertex in a molecule which mapped to a vertex in the characteristic substructure, the vertex object in the molecule would be swapped with the vertex object from the characteristic substructure. This was successfully implemented so that the new vertex object gained the neighbours and connecting edges of the original vertex. However the complexity of performing this swapping action grew as the number of data structures which were required to hold the molecules vertices increased. There was also the consideration that the molecule object was now being used to contain information relevant to the algorithm, so the clear separation of concerns from the original design had been some what lost.

To tackle this an alternative solution was sought. Upon consideration of the successful implementation of the dictionary which held all of the path structures and their relationships to the molecules, it became clear that a similar nested dictionary could be used to contain the information about the characteristic substructure.

This new data structure would contain the molecules which contained all or part of the characteristic substructure as the keys. These would then have the mapping from the molecule vertices to the characteristic substructure vertices as the value. In this way the pertinent information about the relationship was held within the class in which it was to be used thus increasing communicational cohesion.

6.4 Additional Data from the Algorithm

Through investigation into a real world problem provided by the client which involved proprietary data, it was discovered that there was a call to present the representative structures and molecules which contain them. By gathering together the representative path structures of each length it is possible to present a compromise between the characteristic substructure which is not necessarily contained within a single molecule and the maximum common subgraph which must be contained in every molecule. So that the structures of the molecules could be compared, each of the representative structures was numbered when it was displayed and then for each molecule a membership array was provided which signifies which structures are contained in that molecule. Within these arrays if there would be a 0 at position 2 if structure 2 was not found in the associated molecule and a 1 at position 2 if it was.

6.5 User Interface

When considering the best way for the user to interact with the system consideration must be given to how the application is to be used. If the application is to be integrated into a larger application pipeline then allowing the application to be called by simply using command line arguments will allow the operation of the application to be automated. The initial format for these command arguments simply involved allowing the user to specify the SMILES input file.

Currently the command line interface allows the user to set the name of the input file which should either be a text (.txt) or a SMILES (.smi) file. The user can specify whether they would like to receive the characteristic substructure or the representative structures through the use of flags. If no flag is specified then the application creates both sets of data. The user can also specify the threshold for the frequency of the paths and path structures in the molecules, allowing them to receive structures which, though they appear less frequently, may present interesting information.

As the volume of data to be returned upon completion of the application could be quite large it was decided that the best means of delivery would be in a text document. When displaying the results of the characteristic substructure search, the completed structure would be displayed first, and then each of the path structures which had went into the characteristic substructure would be placed in a numbered list. It would be useful to know which molecules contain which components of the characteristic substructure so a membership array would follow each molecule in a manner similar to the presentation of the representative structures.

6.6 Testing

Testing is an important feature of a software development project it not only ensures that the application performs as it is expected to, it also gives the user the opportunity to verify the correctness of the application.

At each stage of the implementation process different types of testing were going on. As each new method or feature was added chosen molecules which contained particular attributes would be run through the method. This was to check that the output that was being returned was as expected and also that there were no unexpected problems with the many different types of molecules which could be entered. The molecules would contain the different types of bonds, branches and rings as well as structures which are repeated in the molecule.

This type of on the spot testing highlighted the unexpected behaviour in the copying of nested dictionaries. Within Python the copy function for dictionaries produces a *shallow copy*. This means that though a new dictionary entry is created to hold the objects, the object references which are contained in the dictionary point to the same objects as were contained in the original dictionary. This then means that the dictionary which is nested inside the original dictionary will be altered if any changes are made to in the copied version of the dictionary. Therefore in order to create a copy of an adjacency dictionary that contains the correct vertex objects, a shallow copy of each nested dictionary must be added into a new larger dictionary. In this way the inner references point to the correct object but the original dictionaries will not be altered by changes in the copy.

A unit test is designed to test a individual element or method. Unit tests were used to test the creation of the vertex, edge, atom and bond objects. They were also used to ensure that the more complicated methods within the graph and molecule classes worked correctly.

Feature testing or integration testing is used to ensure that the different parts of the system work together correctly. As the SMILES parser relies upon the Molecule object as well as its own methods it was important to see that, when given a particular SMILES string, the molecule object which was created had the correct structure. A small number of feature tests were used for the main algorithm to test that, given a known molecule or molecule set, the expected outcome was achieved. However the main testing of the algorithm class was conducted with the evaluation data as these sets of molecules had a target outcome.

Chapter 7

Evaluation

In the evaluation of this project, the focus was placed on how accurately the implementation reflected the original algorithm. This would then lead to conclusions on the accuracy and reliability of the results of the application. The evaluation comparisons were conducted on a quad-core Intel® Core^{TM} i5-3550 with 8 GB RAM under Windows 7 operating system.

7.1 Correctness of Implementation

Within the paper, the molecules and the characteristic substructures which they produced were presented. This then made it possible to test the correctness of this implementation of the algorithm. It is possible to see if the characteristic substructures produced by this implementation resemble those presented in [17].

7.1.1 Data Used for Comparison

The paper offered the characteristic substructures for 15 different compound classes. These classes ranged in size from glucosinolates which consisted of 66 molecules to the 2-acetylaminofluorenes which had only 12 molecules in it. The molecules themselves also varied in size with the largest being the lipids which are on average 136 atoms per molecule.

The molecules within the class were given by their PubChem CIDs which could then be used to retrieve the SMILES string from the PubChem database using the PubChem Python wrapper PubChemPy.

Of the 15 characteristic substructures in the paper, 3 of those depicted were unclear in their structure making it infeasible to use them for structural comparison.

7.1.2 Results of Comparison

Consult Appendix B to view the graphs created by the application for the evaluation. The graphs presented in the original paper have been included for comparison.

The results for this evaluation are overall quite positive. Though many of the results do not match the given graph exactly, they each bear a close resemblance with many of the graphs obtained through the application containing the original graph as a subgraph or themselves being a subgraph of the original. This is true of different sized molecules as both benzothiadiazines with 31 atoms per molecule and uraciles with an average of 15 atoms per molecule were very close matches. The results also show that the application is now quite accurate for molecules with many repeated structures in them. This can be seen in the glucosinolates compound class, whose characteristic substructure included 14 structures that used the multiple structure method in Algorithm 2. This class also shows the robustness of the application for sets containing a large number of molecules.

There are three compound classes which had to be run at a lower frequency threshold in order to find structures which resembled those presented in the paper. These were benzothiadiazines, chlorothiazides and guanines. Benzothidiazines and chlorothiazides produced structures at the normal threshold but they only shared a few features in common with the presented characteristic substructure. Before a characteristic substructure of the guanine class it was required that the threshold be lowered so that the path structures only had to appear in half of the molecules present before they were considered representative.

The application failed to run to completion for one set of molecules. This was the lipids compound class which consisted of molecules that comprised of 136 atoms on average. This average is more than three times greater than the next largest molecules which has an average of 41 atoms per molecule. While running the application using the molecules from the lipids class the application will raise a Memory Error which means that Python has used all of the memory that has been allocated to it.

7.1.3 Improvements During Evaluation

Performing the evaluation highlighted one of the problems which had been present in the implementation of Algorithm 2. Due to the slightly ambiguous wording of the this section of the paper, there was uncertainty about how many structures were to be added to the characteristic substructure when considering the molecules which contain k copies of a structure P. Either a copy of P should be added in the most frequent locations for each value of k or the most frequent location of all the possible k values should be added. Initially, the first alternative was implemented. When the peroxide molecules were run through the application the characteristic substructure which was returned contained 71 atoms, far more than 6 atoms present in the given characteristic substructure. Therefore the alterations were made to the application so that instead the second method of choosing which multiple structures to add was implemented. Following this, the results for the peroxides class were far more similar to the one provided as can be seen in Appendix B.

The application initially had difficulty with the glucosinolates compound class which contains a large number of structures that appear multiple times in a molecule. When running this class not only was it slow (take upwards of 10 minutes) but it would also create an error when comparing these repeated structures. When trying to discover if a structure composed of repeated elements was unique it would sometimes be compared against that had already been added to the characteristic substructure. Adding a path structure to the characteristic substructure made the path structure vertices unsuitable for use with the isomorphism algorithm. This was due to the fact that the position was stored in the vertex object itself, so moving the vertex object altered the position attribute.

To resolve this problem the position attribute was removed from the vertex objects and instead a list was added to each graph which contained all of the vertex objects. The position of the vertex in the structure would then be the vertex's index in the list and would be specific to the graph rather than the vertex. It was necessary to use another data structure along with the graphs adjacency dictionary because when the keys of a dictionary in Python are presented they are in an arbitrary order and so the order of the keys cannot be used as a means of identifying a vertex's position.

7.2 Shortcomings of Application

The application is currently quite memory intensive, particularly for sets involving large molecules. As this causes the application to crash, this problem will have to be resolved to ensure that the application is robust. The NetworkX isomorphism algorithm requires the creation of a NetworkX graph which replicates the original graph. It would therefore be beneficial to find an isomorphism application which can be applied directly to the graph objects or alternatively creating a new NetworkX graph at each comparison rather than holding the graphs in a dictionary. This would be detrimental to speed but might be necessary to achieve a fully functional application.

Due to the lack of a method which can turn an adjacency dictionary into a SMILES string, in order to compare the results of the applications the graph adjacency dictionaries had to be drawn by hand and then an external chemical structure drawing tool had to be used. This limits the practicality of the results presentation. Though the adjacency dictionary demonstrates the structure of a graph it is not as immediately clear for constructing a molecule as a SMILES string.

The small differences in the characteristic substructures could be attributed to one of the assumptions made during its implementation being faulty. Before adding a path structure to the dictionary that maps it to its associated molecules, the path structure is first compared to all of the other path structures in the dictionary. This is to ensure that there are no duplicates within the dictionary and that the molecules map to the correct path structures. However, though this action appears logical when considering how best to compile a list of representative path structures, there is no mention of the uniqueness of the path structures within the method outlined in [17].

The implementation would have benefited from having the path structures stored as separate objects which could then maintain data structures reflecting the relationships with the molecules. If this method were used the code would be more modular and so easier to maintain and modify. This could have a negative impact on the speed of the implementation as the use of global variables can be expensive in terms of performance, however it could help in turns of memory efficiency as it would reduce the need to store multiple relationships between structures. It would also limit the amount of nesting required in the data structures as each path structure object could hold a dictionary of the molecules that contain it.

Chapter 8

Conclusion

8.1 Challenges

One of the challenges faced when conducting this project was the wide range of topics that had to be studied and understood before the design process could begin. This included not only graph theory and the SMILES notation but also Python itself which had previously been known to only a rudimentary level.

Another challenge to be overcome was the ambiguous wording that was occasionally used in the explanation of the characteristic substructure method. As the steps of the algorithm could be interpreted to have a number of different meanings, careful consideration had to be given to the intended outcome and assumptions had to be made on what would make logical sense in the context.

8.2 Future Work

The main work that would be done, given more time, would be to attempt alternative implementations of the algorithm in [17]. The focus would be on reducing the currently high memory usage and altering the assumptions that had been made to try and achieve results that more closely match those provided by Ludwig *et al.*

The creation of a SMILES writer which could turn a molecule object into a SMILES string would make the results far easier for the general user to understand and incorporate the results into other programs. This would require a method within the molecule class that creates a *spanning tree* of the molecule which would visit each of the vertices in turn and break any rings, while a note is taken of where these rings existed. With the introduction of a SMILES writer it would also be possible to incorporate an external tool that could represent the SMILES string as a correctly formatted chemical structure. This would allow the results to be presented in a format that is helpful for the understanding of the user.

It would be beneficial to introduce exception handling to the SMILES parser. This would increase the robustness and reliability of the code as it would ensure that only valid SMILES strings were passed into the algorithm. Further feature and unit testing for the algorithm class would help to discover if there are any discrepancies within the implementation of the algorithm.

8.3 Conclusion

Overall the project has been completed with quite a high level of success. The aim of creating an application which compares the structure of molecules which have been entered as SMILES strings has been met successfully. A SMILES parser has been designed and implemented which allows SMILES strings to be converted into the desired graph structure. A graph data structure has been created which is extendable so that molecule objects and other features such as a path writer can be added. The chosen method of structural comparison has been implemented in such a way that it meets the time constraint set in Appendix A while requiring only one additional package to be installed and so is quite portable

The results from the evaluation show that, though the two applications do not produce exact matches in every case, the graphs received bear a close enough structural resemblance to still prove useful for the identification of unknown molecules.

The project also adds an additional method of comparison with the retrieval of all of the representative structures in the molecules. This allows the user to view all of the possible structures contained in the molecules without the alterations that are made during the creation of the characteristic substructure. This allows the possibility of finding interesting structures which are common to a select number of the molecules.

Appendix A

Requirements Documents

A.1 Initial Requirements

Problem Statement: This software will find the common structural features for a set of molecules.

Background: The software will implement the method put forward by Ludwig *et al.* to find the characteristic substructure which is comprised of structures that appear frequently in the molecules. For further details see [17]. This will be used at Glasgow Polyomics to help identify unknown molecules.

Functional Requirements:

1. The data must be entered in the SMILES string format
2. The application must compare the structural features of the molecules
    (a) The application should implement the chosen algorithm correctly
    (b) The application could offer alternative methods to compare the molecules structures
3. The results should contain the characteristic substructure as a SMILES strings
4. The results could display the list of molecules which contain the characteristic substructure
5. The results could display the characteristic substructure as a molecular structure

Quality and Platform Requirements:

1. The application should aim to be as portable as possible
    (a) The application should have as few dependencies as possible
2. The application should give a response with 1 minute for a set of 30 molecules or fewer
3. The application should be written in Python or R to integrate well within the customers current system

A.2 Use Cases

Use Case: Get characteristic substructure

Actors: Member of Glasgow Polyomics

Goal: Get a structure that demonstrates the common structural features of a set of molecules.

Preconditions: The actor has created a SMILES string for each of the molecules in the set. These have been gathered into a .txt or .smi file in the correct SMILES file format with one SMILES string per line followed by new line symbol.

Summary: When the actor wants the characteristic substructure of the molecules they pass the SMILES file to the application.

Steps:

Actor actions	*System responses*
1. Go to application folder using command line.	
2. Enter a call to run the algorithm.	
3. Include file path to SMILES file as command line argument.	
	4. Display the characteristic substructure in the command line.
	5. Create a file containing the characteristic substructure and a list of the molecules in the same folder as the input file.

Use Case: Get characteristic substructure with less frequent structures

Related Cases: Extension of *Get characteristic substructure* (extension point: step 3. Include file path...)

Summary: When the actor wants a characteristic substructure of the molecules which contains structures that appear in fewer of the molecules, they set a new frequency threshold when they call the application

Steps:

Actor actions	*System responses*
3. Include file path to SMILES file as command line argument.	
4. Include new frequency threshold for the characteristic substructure as a command line argument after the file name.	
	5. Display the characteristic substructure in the command line.
	6. Create a file containing the characteristic substructure and a list of the molecules in the same folder as the input file. The frequency threshold used is included in the file name.

A.2.1 Additional Use Case

There was a request from the client for further functionality which would allow a list of the most frequent structures present in the molecules to be produced. This would add a new use case.

Use Case: Get representative structures

Related Cases: Extension of *Get characteristic substructure* (extension point: step 3. Include file path...)

Summary: When the actor wants to find out all of the structures which appear frequently within the molecules they pass the SMILES file to the application.

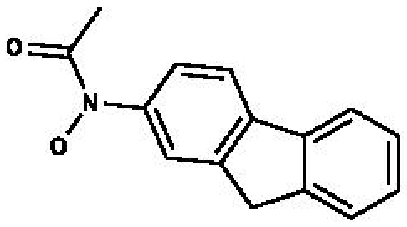
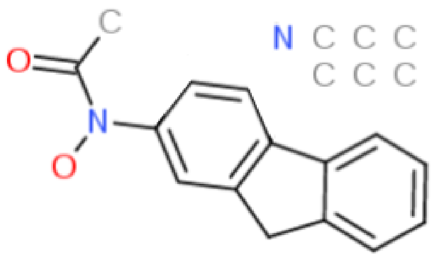
Steps:

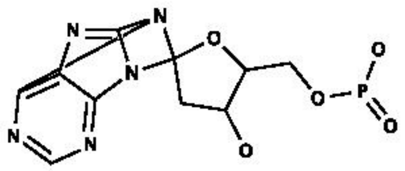
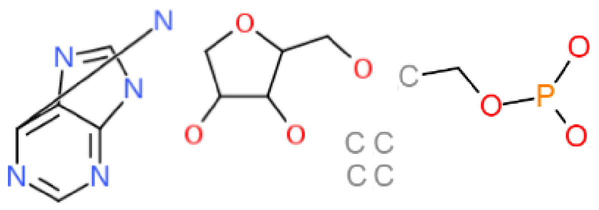
Actor actions	*System responses*
3. Include file path to SMILES file as command line argument.	
4. Include a flag in the command line arguments to indicate that the representative structures are required	
	5. Display the list of representative structure in the command line.
	6. Create a file containing the list of representative structures and a list of the molecules in the same folder as the input file.

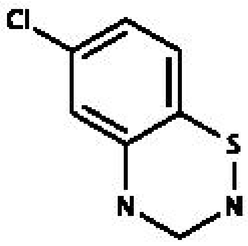
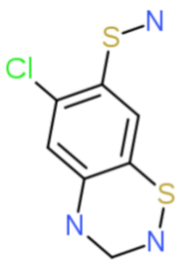
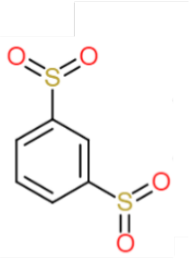
Appendix B

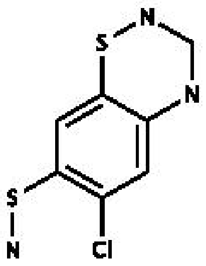
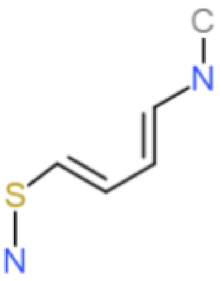
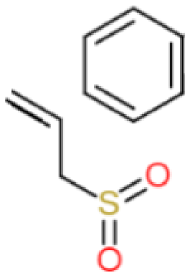
Correctness Evaluation Results

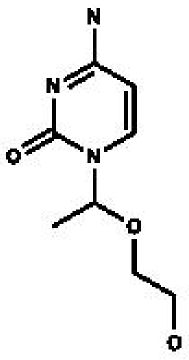
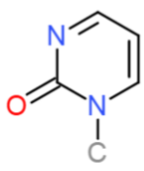
Note that molecule graphs taken from [17] have implicit hydrogens.

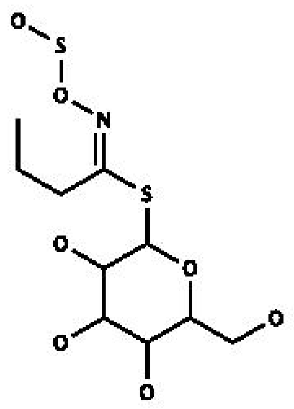
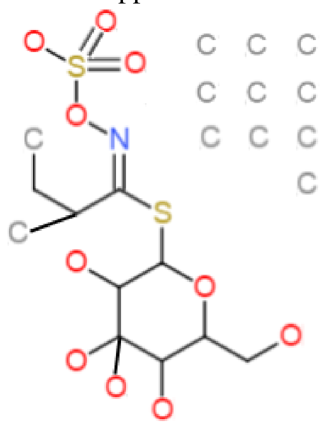
2-acetylaminofluorenes characteristic substructures	
Presented in paper	Obtained in application
	
Molecules in class: 12 Response time: 3.3 secs Threshold of path structures: 0.8	Atoms in molecules (avg.): 20 Calls to multiple structure method: 5

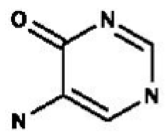
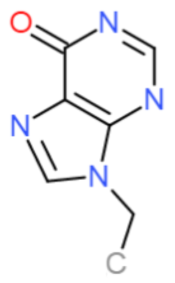
Adenines characteristic substructures	
Presented in paper	Obtained in application
	 N N N N N O O O O C O P O O C C C C
Molecules in class: 44 Response time: 14.2 secs Threshold of path structures: 0.8	Atoms in molecules (avg.): 25 Calls to multiple structure method: 3

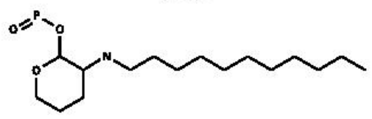
Benzothiadiazines characteristic substructures		
Presented in paper	Obtained in application	
		
	Threshold: 0.6	0.8
Molecules in class: 25 Response time: 15 secs	Atoms in molecules (avg.): 31 Calls to multiple structure method: 0	

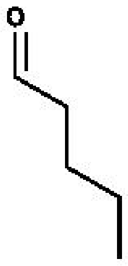
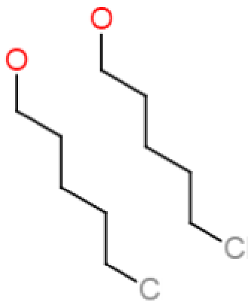
Chlorothiazides characteristic substructures		
Presented in paper	Obtained in application	
		
	Threshold: 0.7	0.8
Molecules in class: 24 Response time: 30.5 secs Threshold of path structures: 0.7	Atoms in molecules (avg.): 41 Calls to multiple structure method: 0	

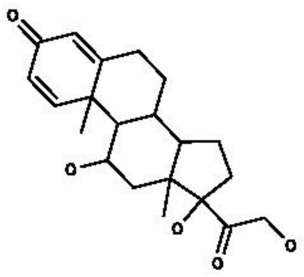
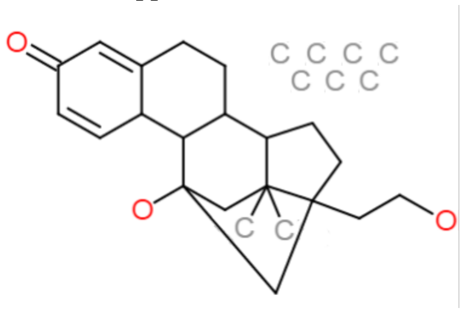
Cytosines characteristic substructures	
Presented in paper	Obtained in application
	
Molecules in class: 22 Response time: 6.9 secs Threshold of path structures: 0.8	Atoms in molecules (avg.): 21 Calls to multiple structure method: 0

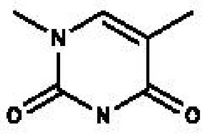
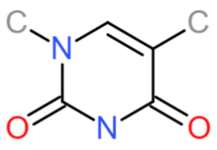
Glucosinolates characteristic substructures	
Presented in paper	Obtained in application
	
Molecules in class: 66 Response time: 123 secs Threshold of path structures: 0.8	Atoms in molecules (avg.): 26 Calls to multiple structure method: 14

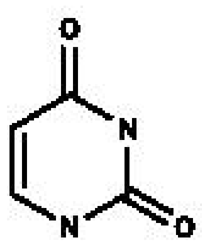
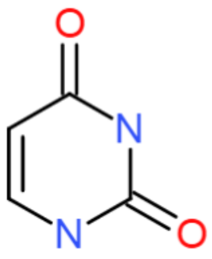
Guanines characteristic substructures	
Presented in paper	Obtained in application
	 Threshold of path structures: 0.5
Molecules in class: 18 Response time: 2.3 secs	Atoms in molecules (avg.): 24 Calls to multiple structure method: 0

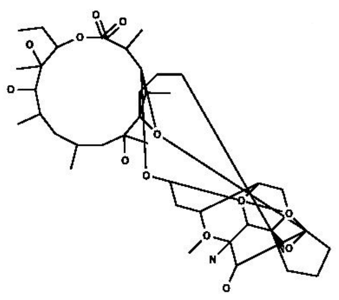
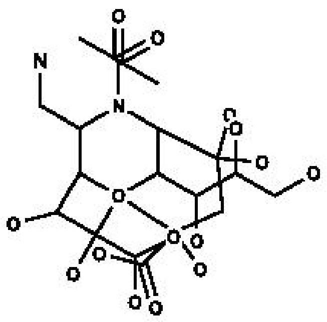
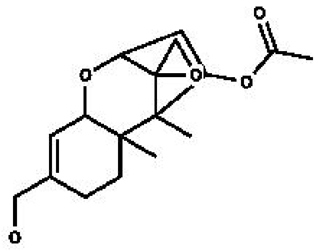
Lipids characteristic substructures	
Presented in paper	Obtained in application
	Program terminated with“Memory Error” when the 26th multiple structure is being added to characteristic substructure
Molecules in class: 22 Response time: Threshold of path structures:	Atoms in molecules (avg.): 136 Calls to multiple structure method:

Peroxides characteristic substructures	
Presented in paper	Obtained in application
	
Molecules in class: 24 Response time: 8.5 secs Threshold of path structures: 0.8	Atoms in molecules (avg.): 26 Calls to multiple structure method: 1

Pregnadienes characteristic substructures	
Presented in paper	Obtained in application
	
Molecules in class: 26 Response time: 20.6 secs Threshold of path structures: 0.8	Atoms in molecules (avg.): 32 Calls to multiple structure method: 9

Thymines characteristic substructures	
Presented in paper	Obtained in application
	
Molecules in class: 24 Response time: 0.9 secs Threshold of path structures: 0.8	Atoms in molecules (avg.): 18 Calls to multiple structure method: 0

Uraciles characteristic substructures	
Presented in paper	Obtained in application
	
Molecules in class: 24 Response time: 0.4 Threshold of path structures: 0.8	Atoms in molecules (avg.): 15 Calls to multiple structure method: 0

Graphs not used for comparison		
Erythromycins	Neuraminic Acids	Tricothcenes
		

Bibliography

[1] Alfred V. Aho. Computer representations of graphs. In *Handbook of Graph Theory*, pages 57–67. CRC Press.

[2] Sebastian Böcker, Matthias Letzel, Zsuzanna Lipták, and Anton Pervukhin. SIRIUS: Decomposing isotope patterns for metabolite identification. *Bioinformatics*, 25, 2009.

[3] Nathan Brown. Chemoinformatics - an introduction for computer scientists. *ACM Computing Surveys*, 41(8), 2009.

[4] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento. An improved algorithm for matching large graphs. In *3rd IAPR-TC15 Workshop on Graph-based Representations in Pattern Recognition, Cuen*, pages 149–159, 2001.

[5] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, 2 edition, 2001.

[6] Daylight Chemical Information Systems, Inc. 3. SMILES - a simplified chemical language. `http://www.daylight.com/dayhtml/doc/theory/theory.smiles.html`, 2008.

[7] Reinhard Diestel. *Graph Theory*. Springer, 1997.

[8] P. Foggia, C. Sansone, and M. Vento. A performance comparison of five algorithms for graph isomorphism. In *3rd IAPR-TC15 Workshop on Graph-based Representations in Pattern Recognition, Cuen*, pages 188–199, 2001.

[9] Gary L. Glish and Richard W. Vachet. The basics of mass spectrometry in the twenty-first century, 2003.

[10] Michael T. Goodrich, Roberto Tamassia, and Michael H. Goldwasser. *Data Structures and Algorithms in Python*. Wiley, 2013.

[11] Jonathan L. Gross and Jay Yellen. *Handbook of Graph Theory*. CRC Press, 2004.

[12] Jonathon L. Gross and Jay Yellen. Fundamentals of graph theory. In *Handbook of Graph Theory*, pages 2–19. CRC Press, 2004.

[13] Dick Grune and Ceriel J.H. Jacobs. *Parsing Techniques - A Practical Guide*. Ellis Horwood, Chichester, England, 1990.

[14] Haleem J. Issaq, Que N. Van, Timothy J. Waybright, Gary M. Muschik, and Timothy D. Veenstra. Analytical and statistical approaches to metabolomics research. *Journal oof Seperation Science*, 32, 2009.

[15] Craig A. James. OpenSMILES specification. `http://opensmiles.org/opensmiles.html`, 2012.

[16] Douglas B. Kell. Systems biology, metabolic modelling and metabolomics in drug discovery and development. *Drug Discovery Today*, 11, 2006.

[17] Marcus Ludwig, Franziska Hufsky, Samy Elshamy, and Sebastian Böcker. Finding characteristic substructures for metabolite classes. In *German Conference on Bioinformatics 2012*, pages 23–38, 2012.

[18] Gary J. Patti, Oscar Yanes, and Gary Siuzdak. Metabolomics: the apogee of the omics trilogy. *Nature Reviews Molecular Cell Biology*, 13, 2012.

[19] Glasgow Polyomics. `http://www.polyomics.gla.ac.uk/`.

[20] John W. Raymond and Peter Willett. Maximum common subgraph isomorphism algorithms for the matching of chemical structures. *Journal of Computer-Aided Molecular Design*, 16:521–533, 2002.

[21] David S.Wishart. Current progress in computational metabolomics. *Briefings in Bioinformatics*, 8, 2007.

[22] David Weininger. SMILES, a chemical language and information system. 1. Introduction to methodology and encoding rules. *Journal of Chemical Information and Modeling*, 28, 1988.

[23] David Weininger. SMILES. 2. Algorithm for generation of unique smiles notation. *Journal of Chemical Information and Modeling*, 29, 1989.

[24] David Weininger. SMILES. 3. Depict. Graphical depiction of chemical structures. *Journal of Chemical Information and Modeling*, 30, 1990.