Contents

Chapter 1

Introduction

It is commonly known that the act of measuring something changes it. For instance, attempting to measure an electron requires emitting a photon, which in turn changes the electron's path. This fact has a powerful consequence on our knowledge of state, as it creates a discrepancy between what is measured versus what actually is there. In the case of an electron, this is trivial to account for, but what if one aimed to measure the state of a whole molecule?

Currently, this is impossible to do reliably and instead, researchers have developed techniques that measure unique attribute patterns and deduce the molecule from there. Yet this in turn requires deriving the patterns of every molecule, thus gradually synthesizing and documenting of molecules in databases[22]. Unsurprisingly, these databases remain incomplete and if one wishes for them to identify an unknown compound, they will return any number of possible matches based on their approximations. This project focuses on the implementation of techniques that narrow down the list of matches and ideally provide clues as to the identity of the desired compound.

1.1 Context

To be more specific to our situation, this project is primarily interested in *metabolites* (small molecules found in organic processes), measured via *Mass Spectrometry*. This process produces a *fragmentation spectrum* per molecule, which is then looked up in aforementioned databases to yield a list of possible candidate molecules. Our application uses this data to obtain a *Characteristic Substructure* (CS) from the candidates and then performs a *Peak Assignment* to evaluate how well the CS correlates with the fragmentation spectrum. All of these terms will be covered in detail in the *Background* Section (2).

1.2 Aims & Achievements

As mentioned previously, this project aims to aid in the identification of metabolites. It builds upon the Masters project[30] of former student Mary McDowall, making use of her implementation of the Characteristic Substructure algorithm and enhances it in numerous ways.

To begin, it improves upon the CS by being able to handle and characterise multiple candidate lists, instead of originally just one. Further, it adjusts the core of the algorithm to produce more representative structures, adding

various parameters and thresholds to control this. Moreover, it also corrects discovered mistakes in the original algorithm and increases the performance in terms of both speed and memory.

Next, it addresses some correctness issues in the initial parsing (converting strings into molecule graphs) of the chemical compounds and provides an algorithm to convert the CS back into a legible and usable format. Further, it visualises the CS for easier analysis.

Finally, the project also developed a wrapper for a tool known as *CFM-ID* that matches the CS to a fragmentation spectra, evaluating the results and plotting the selected peaks and their fragments for simple viewing.

On the whole, the project was a success and managed to meet the stakeholders' requirements (See *Section 3*), in spite of the unfamiliarity of the subject area and the issues with the initial code base.

Chapter 2

Background

As the application needed to be developed for *Metabolomics* research, this necessitated familiarisation with the actual subject area. This chapter will cover the respective concepts and key terms.

2.1 Metabolomics

The field of *metabolomics* is an emerging area concerned with the study of the chemical "fingerprints" that cellular processes leave behind[31]. Said fingerprints are typically referred to as *metabolites* which are small molecules of masses from 50-1000Da[1]. These molecules are of particular interest as they can effectively act as a readout of the physiological state of an organism. This information can then be used to assess the qualities of medical treatments as well as the behaviour of diseases[25]. Some examples of metabolites are caffeine or a vitamin such as B2.

2.2 SMILES Notation

Molecules can be defined using a number of notations, some of which are empirical, defining only composition, while others also consider structure. Caffeine can be expressed in the form $C_8H_{10}N_4O_2$, but this occludes all information about the molecule's bonds or structure. In this project we will use the Simplified Molecular Input Line Entry System (SMILES) form[35], adhering to the OpenSMILES standard[6]. For the purpose of this project, we will be interested in the part of the language that describes the atoms, bonds, rings and branches only (full grammar can be seen in *Appendix A*).

2.2.1 Atoms

Atoms are denoted using their respective atomic symbol. If the atom is part of the organic set (B, C, N, O, S, P, F, Cl, Br, I,*) or aromatic organic set (b, c, n, o, s, p) they can be described on their own like 'C' or 'n'. If they are not in either, they need to be of the form [<atom>], '[He]' being an example. Notably, Hydrogen is assumed to be bonded to every atom, corresponding to the atom's valence (available charge to form compounds with other atoms). Further additions that can be made to atoms (all of these require '[]', regardless of atom):

[1] A Dalton (Da) is the atomic mass unit, where 1 nucleon has the mass of 1Da [2].

Rule	Form	Examples
Explicit Hydrogen counts:	[<atom>H<number>]	[CH1], [CH], [Cl4]
Charge:	[<atom><charge><number>]	[C-] [C+] [C-1] [O6+]
Isotope:	[<isotope number><atom>]	[238U]
Together:	[<isotope><atom><H count><charge>]	[13CH4+]

Table 2.1: Atom structure examples

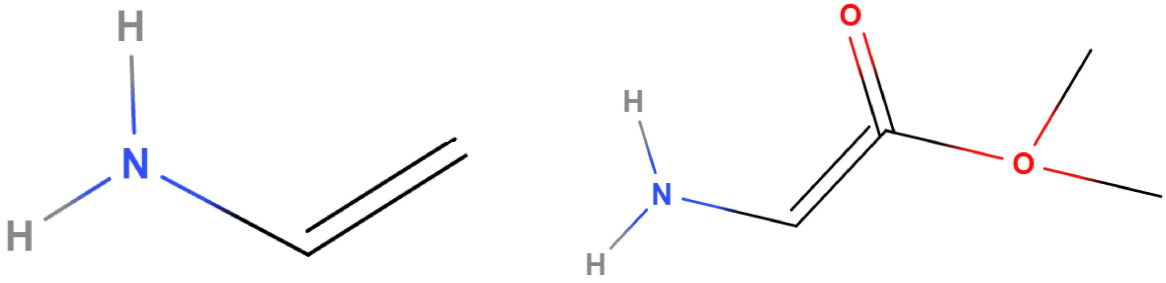
Note, that a charge or hydrogen count can be omitted if it is 1.

2.2.2 Bonds

We will use 4 bond types in this project, which are the single (-), double (=), triple (#) and quadruple ($) bonds. A typical SMILES string using these would be 'C=C-N'. However, a single bond can also be treated implicitly, so the string 'C=CN' is also valid.

2.2.3 Branches

A branch in a molecule is an atom with more than 2 bonds and is denoted using '(' and ')' and may be used repeatedly and recursively. For instance C(=C)N and C(=CN)(O(C)C)=O are of the form:



(a) C=CN

(b) C(=CN)(O(C)C)=O

Figure 2.1: Branch SMILES Examples

What is also of note, is that are no requirements on the order of the branches nor which atom the SMILES string starts. As such NC=C(=O)O(C)C describes the same molecule as in *Figure 2.1b*.

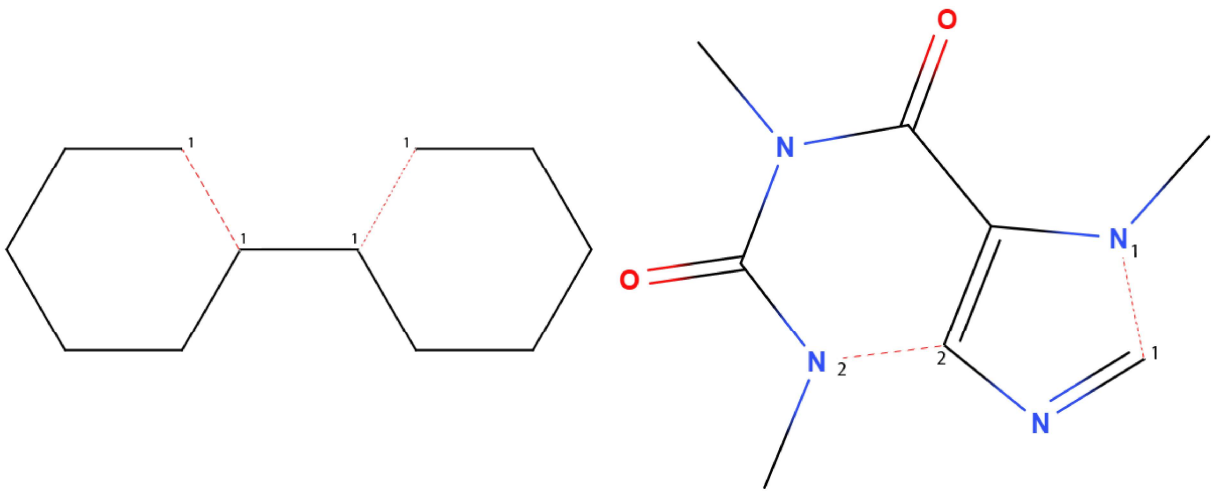
2.2.4 Rings

A ring in a molecule represents a cycle of bonds an atoms. In SMILES a ring is broken at a bond between a pair of atoms *A*, *B* and is represented using 2 integer IDs in the form of <atom A><bond><ring number><other atoms><atom B><bond><ring number>. Notably, the bond only needs to be specified for either *A* or *B* as it is the same for both. Further, any 2 adjacent atoms in the ring can be used to denote the ring and the actual IDs take the form of '0'-'9' or '%10'-'%99' (IDs >99 are illegal) and can be re-used. Lastly, an atom may be part of more than 1 ring, in which case the IDs can be defined in any order after the atom. To clarify, here is an example of each rule:

Rule	Correct Examples	Incorrect Examples
Simple Rings:	C1CCCCC1, C2CCCC2, C%10CCCC%10	C1CCCC2, C1CCCC1C1
Ring Bonds:	C=1CCCC1, C1CCCC=1, C=1CCCC=1	C=1CCCC-1
Multiple Rings:	C12CCCC1CC2, C1CCCCC1C1CCCCC1	C12CCCCC12
Together:	C(C=CC=1O)(C=CC(=O)O)=CC=1	-

Table 2.2: SMILES Ring Examples

Notably, the example C12CCCCC12 is illegal as a pair of atoms is not allowed to share more than 1 bond. Finally, here are some molecule diagrams using all of the above:



(a) C1CCCCC1C1CCCCC1 - dicyclohexyl

(b) CN1C=NC2=C1C(=O)N(C(=O)N2C)C - caffeine

Figure 2.2: Ring SMILES Examples

The dashed lines above indicate where the SMILES string has been split and indexed. Also note that as in both *Figure 2.2a* and *Figure 2.1*, a Carbon (C) atom is not labelled explicitly. It is also not uncommon for Hydrogen (H) atoms to not be shown at all and the default number of a stable valence to be assumed.

2.3 Mass Spectrometry

Mass spectrometry (MS) is an analytical method that determines the mass of the chemical constituents of a sample. It is frequently used in life sciences to identify compounds such as metabolites and displays high levels of sensitivity to even the slightest changes in molecular structure[20]. For this project, it is of interest as it produces some of the data the application uses.

A mass spectrometer is typically comprised of 3 main steps: ionisation, acceleration/separation and detection. The process begins by having parts of the sample ionised (bombarded with charged particles). They are then separated and focused and redirected towards a detector with magnetic fields. Lastly, the detector then measures the mass/charge ratio of the received ions and the relative abundance of each ratio encountered[12]. These measurements are then passed to a data processing system to obtain a result like the following:

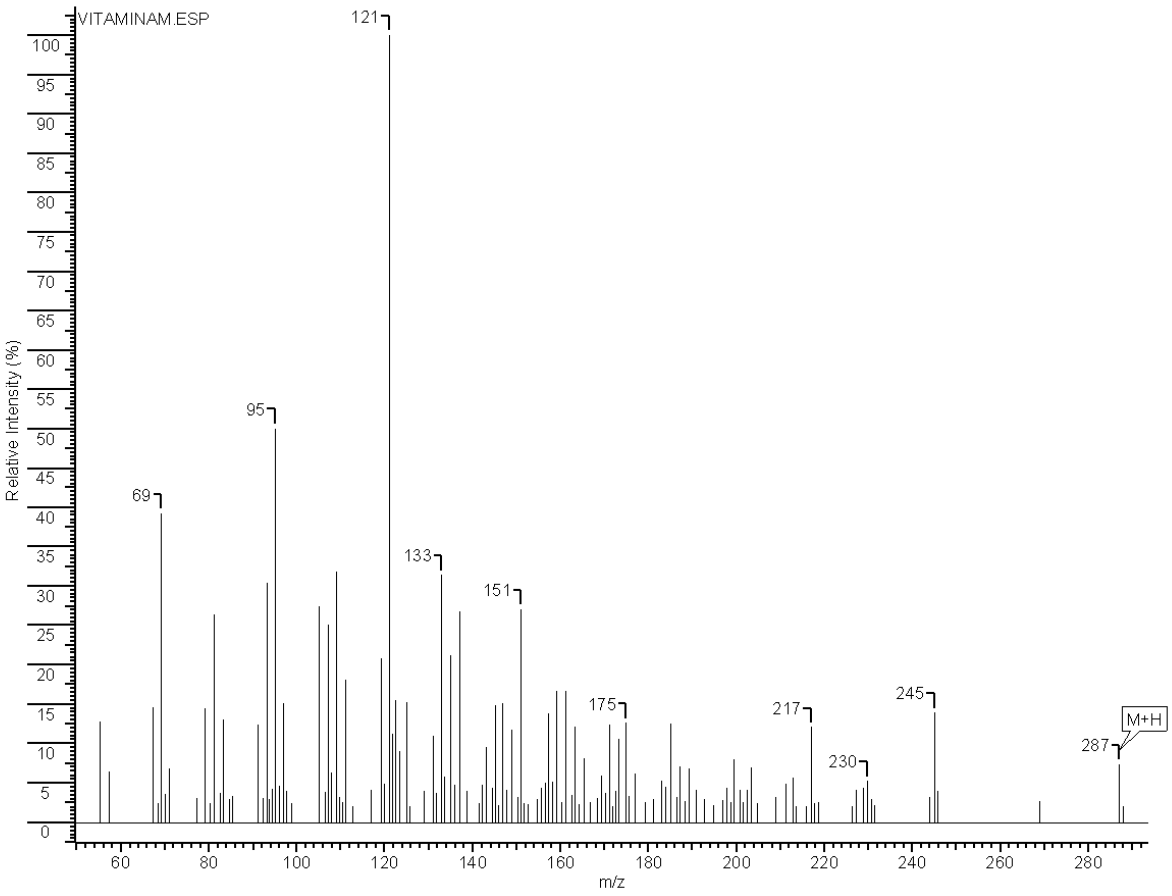


Figure 2.3: Liquid Chromatography MS Fragmentation Spectrum of Vitamin A[14]

What can be seen in *Figure 2.3* is a *fragmentation spectrum* (sometimes referred to as a *fragmenation pattern*) which shows the detected mass/charge ratios and how frequently they were measured, relative to each other. This spectrum is the canonical "fingerprint" of a molecule and is one of the inputs for the project's application. Each peak is considered to be a different fragment molecule of the original sample.

2.4 Spectral Peak Assignment

Using a spectrum like the above, it is possible to attempt to match a molecule to it. There a number of approaches to do this, most of which use a series of heuristics and rules to break the molecule into fragments and then match the respective ratios of the parts to the individual peaks. The particular one this project is interested in (and makes use of) is a modelling process known as *Competitive Fragment Modelling*(CFM)[8]. CFM makes use of machine learning and accordingly has 2 main stages: a supervised learning training phase in which it is given a (preferably large) number of spectra and their molecules to learn and a prediction stage where the trained model is used to predict a matching between a molecule and a fragmentation spectrum.

The training phase involves estimating a set of parameters w using an input vector X that contains a list of molecules and 3 fragmentation spectra per molecule at different *energy levels*[2]. The parameters are then esti-

[2]The energy level refers to the energy at which a MS is performed and typical ones chosen are 10V, 20V and 40V (alternatively referred to as low, medium and high).

mated using the *Maximum Likelihood* and *Expectation Maximization*[11].

The actual prediction phase takes in a molecule and 1-3 energy level fragmentation spectra. It then uses Markov state transitions, fragmenting the molecule based on bond-breaking rules and probabilities it determines throughout the search. At the end, it outputs potential assignments for each peak in every given fragmentation spectrum, together with measures of confidence in the assignment. The assignments themselves are a molecule fragment and its mass/charge ratio. Here is an example of what this might look like:

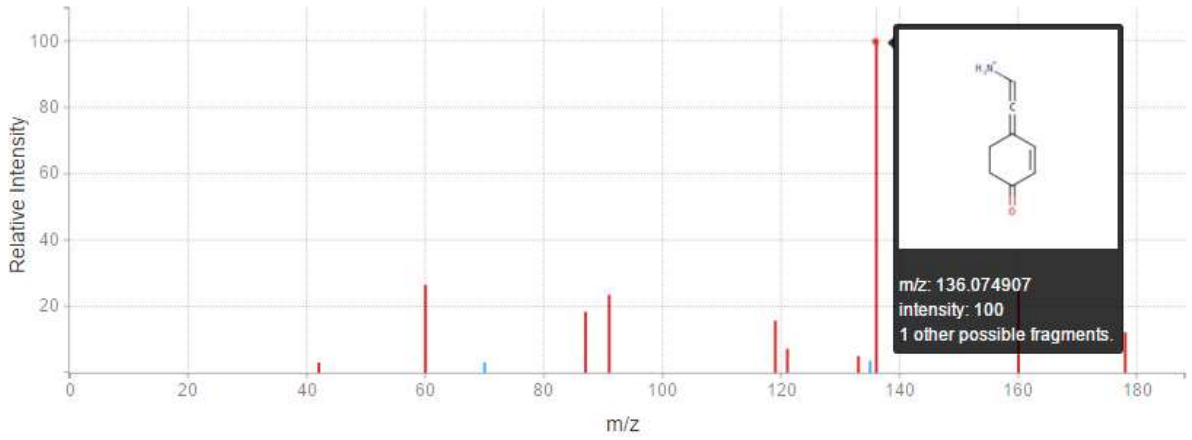


Figure 2.4: CFM-ID Web view of a peak annotation[37]

The plot in *Figure 2.4* shows matched peaks to be red and unmatched peaks to be blue. The most intense peak in the spectrum is highlighted and shows a potential matching fragment. More information about the machine learning algorithm and implementing software can be found under the respective paper[8].

For this project, this technique is interesting as this will allow us to assess how well the *Characteristic Substructure* can be explained with the peaks. Acquiring many matches of high confidence for large peaks, indicates a stronger correlation.

2.5 Graphs

When representing a molecule, they are typically implemented as *Graphs* and a lot of respective theory is necessitated for the *Characteristic Substructure Algorithm*. Accordingly, this section will detail the background for the algorithms and structures used.

2.5.1 Graphs & Molecules

A Graph is composed of a group of points (formally called *vertices*), where each vertex can have 0 or more connections to any other vertex (including itself). The connections between vertices are referred to as *edges*[26]. The formal notation for a Graph G is $G = (V, E)$, where V is a set of vertices and E is a set of edges. Accordingly vertices are denoted by $v \in V$ and edges are expressed by $e = (u, v) \in E$, where u and v are the vertices the edge e connects.

When accessing vertices next to each other, this is known as graph traversal. There are commonly 2 kinds of edges to traverse: *undirected* and *directed*. An undirected edge between v and u means that it is possible to

travel from $v \rightarrow u$ and $v \leftarrow u$. With a directed edge, traversal is only possible in one direction. Other relevant terms is that the starting vertex is known as the *origin* and the target vertex is the *destination*. Lastly, the vertices immediately adjacent to a vertex v are known as the *neighbours* of v.

Using this notation, molecules can be represented by graphs quite easily. Atoms are represented by the vertices and are labelled with various properties (e.g charge), while bonds are represented by the edges. As bonds have chemical limitations[5] on their behaviour these need to be reflected in the graph. Accordingly, all edges are undirected, a pair of vertices cannot be connected with more than 1 edge, nor can a vertex have an edge to itself. Here is an example of a molecule as a graph:

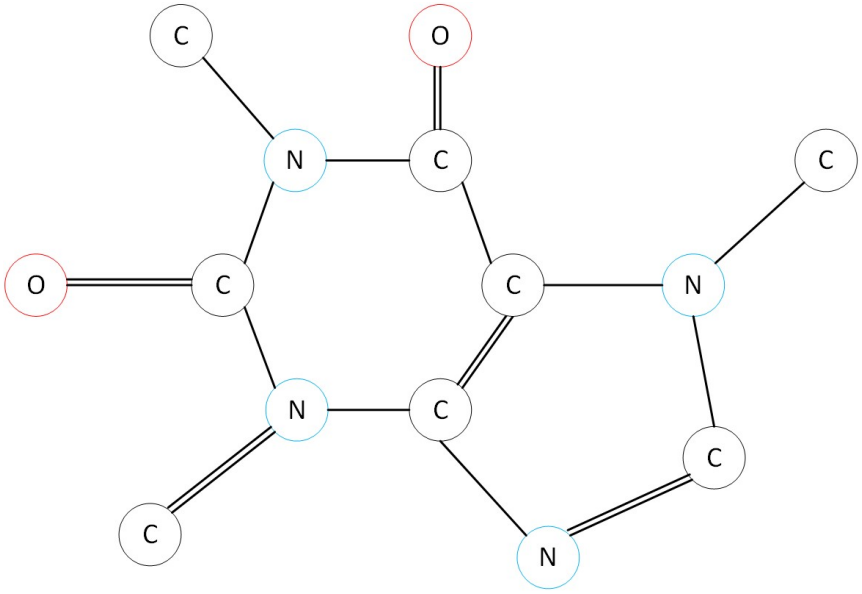


Figure 2.5: A graph representation of caffeine

Here, single bonds are shown with 1 line, whereas double bonds are shown with 2. Additionally, this graph is *hydrogen-suppressed*, meaning that Hydrogen atoms are not shown and will not be explicitly used by the *Characteristic Substructure Algorithm*.

2.5.2 Graph Structures

When using graphs, it is also important to be able to refer to structures in the graph. The following are the most relevant expressions.

A *path* is a sequence of edges, connecting a series of vertices which are normally assumed to be distinct (implying a path cannot have the same vertex in it repeatedly). An example from *Figure 2.5* would be the path "O=C-N-C=O".

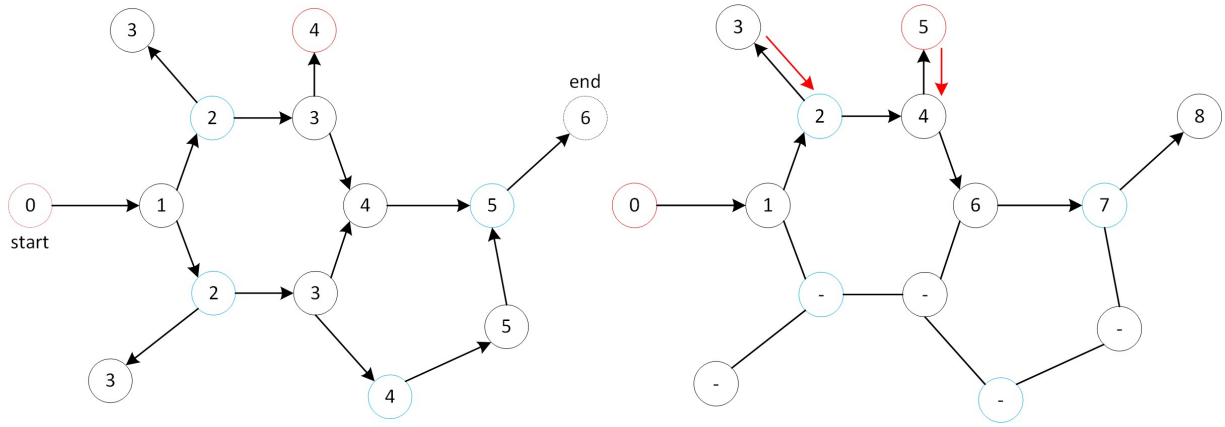
A *cycle* is a path where the first and final vertex of the series share an edge, resulting in them forming a loop. Any ring in a molecule is represented by a cycle in a graph.

A *subgraph* refers to a graph whose vertices and connected edges are a subset of another graph. Formally graph G is subgraph of graph H if $V(G) \subseteq V(H)$ and $E(G) \subseteq E(H)$. For example, the aforementioned path is also a subgraph.

An *vertex induced subgraph* (sometimes just referred to as *induced subgraph*), is a graph H which takes a subset of vertices V_S, from some graph G and all edges from G which have both endpoints in V_S. This can be expressed as $H = G[V_S]$. An example subgraph can be induced from a ring of vertices from *Figure 2.5* creating a graph of both the vertices and the ring edges. E.g "C1-N-C=N-C=1".

2.5.3 Graph Traversal

Graphs can be traversed in a number of ways. When searching for a path from a vertex to another, one commonly uses *Breadth-First Search* (BFS) or *Depth-First Search* DFS). BFS performs starts from vertex and visits all of its neighbours, then visits the neighbours of those neighbours, etc. DFS visits a vertex's neighbour, then the neighbour of that neighbour, etc. without necessarily visiting all the neighbours of the previous vertices first. For more clarity, here an example traversal of the previous molecule:



(a) Breadth-First Search Example

(b) Depth-First Search Example

Figure 2.6: Graph Traversal Examples

The above 2 traversals show the steps each algorithm would take when finding a path from the "start" to the "end" vertex and the integers show during which steps the vertices are encountered. More details on these algorithms can be found here[34]. This project employs variants of these traversal algorithms to identify various structures in the molecules.

2.5.4 Graph Isomorphism

A graph G is *isomorphic* to another graph H if they have the same number of vertices which are all connected in the same way. More formally, two graphs are isomorphic if there is a bijection $f : V(G) \to V(H)$ that preserves edge adjacency $e = (u, v) \in V(G)$ iff $\exists f(e) = (f(u), f(v)) \in V(H)$. As the project treats molecules as graphs, this condition is used to identify if 2 molecules are the same. An added factor is that all of the molecules' labels have to match as well. Another interesting point is that the complexity of determining whether 2 graphs are isomorphic is neither polynomial nor NP-Complete[24], but in an ambiguous 3rd category[3].

[3]More recent news (November, 2015) suggest that a quasi-polynomial algorithm that can solve all problem classes has been found. This is still pending verification[10].

2.6 Characteristic Substructure Algorithm

The *Characteristic Substructure Algorithm* (CSA) is an algorithm written by Ludwig *et Al.*[28] and has the purpose of building a *representative* molecule (the CS) from the subgraphs of a list of molecules M. The definition of "representative" also needs to be specified in the form of choosing a frequency threshold $f_m \in [0, .., 1]$ which denotes that for any graph G in the CS, it must be subgraph in at least $f_m \times |M|$ of all M. So given an M of 5 molecules and a f_m of 0.8 (80%) all $G \in CS$ would have to be in at least 4/5 molecules.

The core algorithm can be broken down into 3 steps.

1. Find all *representative paths* P from M.
2. Find all *representative path structures* S from P.
3. Choose the most representative structure in S to be the CS. Then, choose each path structure in the order of descending representativeness and if possible, add it to the CS.

These stages are iterated over repeatedly. Prior to each iteration, the algorithm chooses a path length l to get the representative paths. Typically, it will decrement l by 1, unless it manages to complete step 3 (it finds at least one representative path structure), where it will decrease by intervals of l_{step}. Finally, when $l \leq l_{end}$, the algorithm terminates. Notably, the choice of the starting values of l, l_{step} and l_{end} is left open to the user.

In regard of the individual steps, a *representative path* refers to any path p in M s.t $frequency(M, p) \geq f_m \times |M|$. Paths are considered to be equal (and contribute to frequency) if they have the same atom and bond types. As mentioned earlier, the choice of path length is given by l.

The *representative path structures* are subgraphs induced using the vertices from the representative paths. This adds any edges between the vertices that were not already in the path, resulting in (potentially) different graphs. At this point, a graph isomorphism algorithm needs to be used to count isomorphic path structures and ascertain which of them satisfy $f_m \times |M|$, assigning the result to S_r.

The 3rd step begins with ordering the *representative path structures* $s \in S_r$ by descending values of $frequency(M, s)$. It then chooses s_0 (most frequent to be the characteristic) and then proceeds to attempt to add $s_1 ... s_{i-1}$ where $i = |S_r|$. Where and how many times a path structure s is added is determined by a number of factors. The first factor is, whether there exist $k > 1$ identical s in at least $H_{iso} \times |M|$ molecules. Here, k denotes the number of times a path structure is in the same molecule and H_{iso} is another configurable constant. After choosing how many times to add s to the CS, the algorithm then selects the k (for multiple) or 1 (single) most frequently occurring positions of s next to the CS in every of M. Thus, if a molecule has both vertices in common with the CS and with s, then these common vertices are considered as a potential position.

A full example of these steps can be seen in Appendix *Section B*, whereas a fine-grained implementation of the algorithm & data structures can be found in *Section 4.3*. Additionally, here is a pseudo-code of the overarching

CSA algorithm:

Algorithm 1: Characteristic Substructure Algorithm

Data: list of molecules M, frequency f_m, lengths l_{start}, l_{step}, l_{end} and multi-structure constant H_{iso}
Result: Characteristic substructure CS

$l = l_{start}$
$CS = \emptyset$
while $l \geq l_{end}$ **do**
  $P =$ find all paths length l in $\forall m \in M$
  $P_r \subseteq P \; s.t. \; \forall p \in P_r, \; freq(p, M) \geq f_m \times |M|$
  $S = m[p], p \in P_r, m \in M$
  $S_r \subseteq S \; s.t. \; \forall s \in S_r, iso\_freq(s, M) \geq f_m \times |M|$
  $S_r = sort(s \in S_r, iso\_freq(s, M))$
  if $S_r \neq \emptyset$ **then**
    if $CS = \emptyset$ **then**
      $CS = S_r$
    else
      for $s \in S_r$ **do**
        if $iso\_count(s, m) > 1, m \in M$ *for at least* $H_{iso} \times |M|$ **then**
          $k = min(iso\_count(s, m) > 1), m \in M$ for the upper $H_{iso} \times |M|$ counts
          $L_{pos} = find\_k\_best\_possible\_positions(s, k)$
          $CS.add(s, l_{pos}), l_{pos} \in L_{pos}$
        else
          $l_{pos} = find\_best\_possible\_positions(s)$
          $CS.add(s, l_{pos})$
  if $CS \neq \emptyset$ **then**
    $l = l - l_{step}$
  else
    $l = l - 1$
return CS

In the above algorithm, the function calls correspond to the following:

Function	**Action**
$freq(p, M)$	Returns the frequency at which a path p can be found over all molecules M.
$iso\_freq(s, M)$	Returns the frequency at which a path structure s can be found over all molecules M.
$iso\_count(s, m)$	Counts the number of times a path structure s occurs in the same molecule m.
$find\_k\_best\_possible\_positions(s, k)$	Finds the k most fitting locations to add a path structure k times.

Table 2.3: CS Algorithm Functions

2.7 Contextualisation

This section elaborates on the initial context of *Section 1.1* to explain how all of these concepts tie into the process of identifying metabolites. Observe the following diagram:

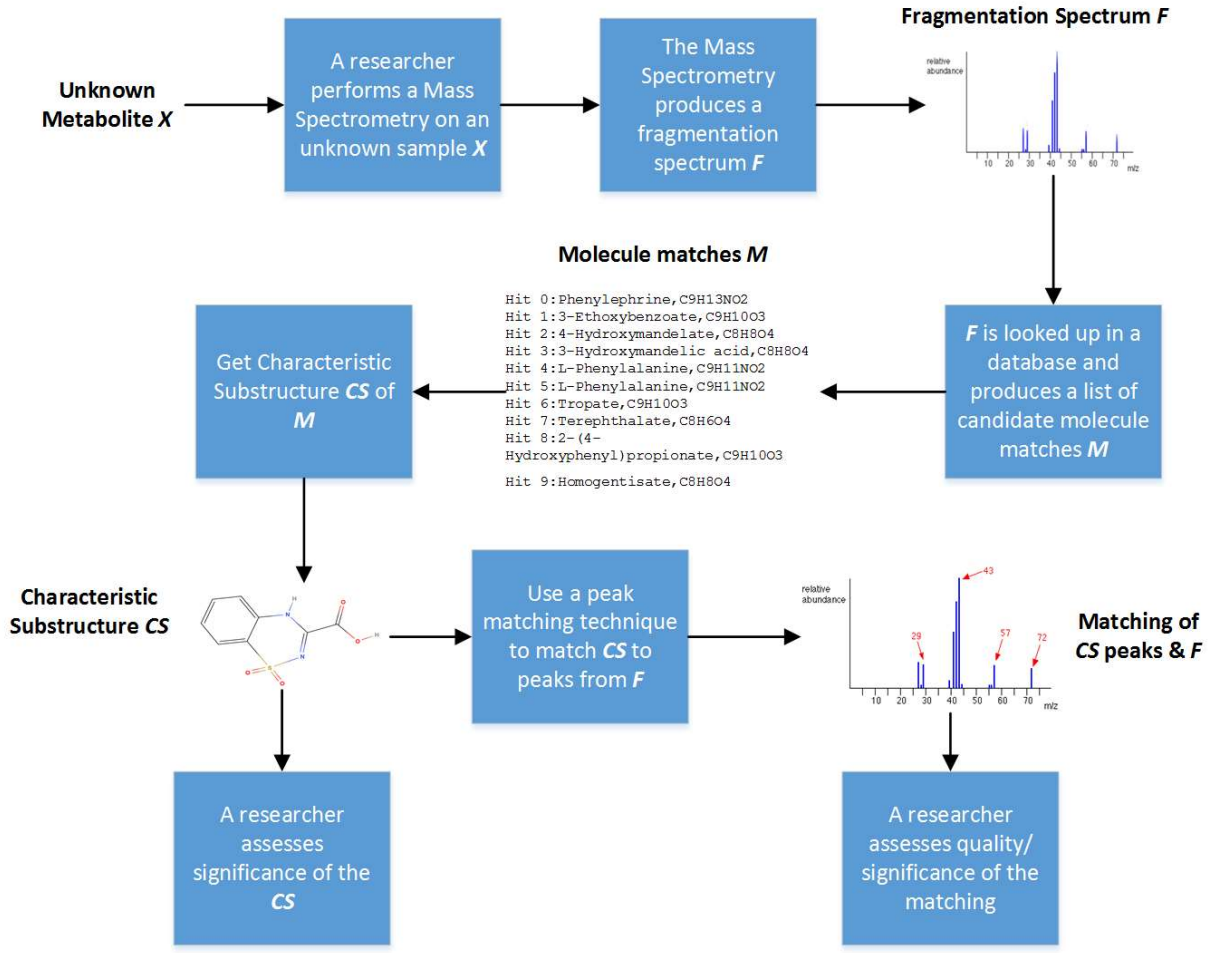


Figure 2.7: Research & Application Flow

This figure describes how the techniques and algorithms described are used collectively. Starting at the top left, a researcher performs an MS measurement on an unknown metabolite, getting a fragmentation spectrum. They will then lookup the spectrum in a database such as PubChem[22] HumanDB[14] which will return a list of hits. These ”hits” are a list of suggested results ranked from most to least likely. An application then gets the CS from that list, producing a result for a researcher to assess. Further, the CS can then be compared to the original fragmentation spectrum (either manually or with an application) to assess how well the CS matches the measurement.

This concludes the background material. The next section covers the requirements, detailing how this project will be part of this system.

Chapter 3

Requirements

As stated earlier, the general aim of this project is to provide various extensions to the CS algorithm. These extensions were elicited over a series of iterations (See Appendix C) with the stakeholder(s) of this project. The stakeholder representative for this project was Dr. Simon Rogers, acting as a liaison for the *Glasgow Polyomics* Research Group[1]. This group specialises in biochemistry and bioinformatics-related fields one of which is *Metabolomics* and is interested in using this application individually for research as well as integrating it into a larger system.

3.1 Prior Work

The first requirement of the project was the familiarisation with the existing content. A previous project by Mary McDowall, had implemented a parser from SMILES to a molecule graph as well as the CS algorithm. The source code for this project and paper[30] were made available for study and use in October, 2015 to complete the requirements.

3.2 Functional Requirements

The functional requirements are a series of direct enhancements to the CSA, allowing cross-checking with fragmentation spectra and a collection optional improvements if time permits. Here is a list of the main requirements:

1. Enable the usage of *multiple molecule lists* as an input and adding a threshold to regulate list membership for the CS. This would mean for instance, that if a CS algorithm received 3 input lists and we define the list threshold to be 2/3, then any path or path structure has to be found in at least 2/3 lists to be considered representative.
2. Upon completing Req. #1, add a mechanism that observes which input molecule the CS algorithm considers best matching. This requires taking 2 sets of molecule lists S_{M1} and S_{M2} and one singular 3rd list of molecules M_T. Then, attempt to build a characteristic substructure CS_1 from S_{M1} and another CS_2 using S_{M2}, both using the same molecule $m_t \in M_T$. Perform this iteratively for every $m_t \in M_T$ and observe which molecule affects CS_1 and CS_2 the most. The purpose of this task is to identify a molecule that correlates most with 2 groups S_{M1} and S_{M2}.

3. Independent of Req. #2, find a means such as *in silico fragmentation*[1] to ascertain how well a CS can explain a fragmentation pattern F. In this case "explain" means to use a technique that attempts to match a the CS to the F and then expresses a measure of success or confidence. This is intended as a feedback mechanism for whether the produced CS is meaningful against the original measurement(s).

Respectively, here are the optional requirements:

- Create a parser that converts the molecule graph back into a SMILES string. This is to make the created CS easier to read, potentially even plotting the molecule.
- Plot the fragmentation spectra with annotated peaks for easier legibility.

3.3 Non-Functional Requirements

The non-functional requirement can be summarised by improving the performance of the existing CS algorithm. There were indications of memory issues with the calculating the CS that caused issues in the previous project[30]. Although there are no explicit computational requirements, it is generally desirable for analysis tools to be performant and reasonably scalable.

[1] This is an evaluation technique that takes a in a fragmentation pattern and then queries one or more databases to find matches and then ranks suitability vs. the pattern. More under[38].

Chapter 4

Design & Implementation

This section covers how the actual algorithms are designed and implemented. Further, it also explains the overall application structure and various design decisions made to meet the requirements.

4.1 The Existing Implementation

As the core requirements build entirely on the existing CS algorithm and parser, this implementation needs to be reviewed and adjusted for the project's requirements. The entire code is implemented in Python 2.7 and has only 1 library dependency: `NetworkX`[7]. This package is useful for graph functions and appears to be performing graph isomorphism checks via the *VF2*[13][1] algorithm. Additionally, the code is well-documented and does run, producing output CS graphs for input SMILES files. Here is the structure of a command it can be run with:

```
python source\characteristic_substructure.py -c data\folder\FileName.txt
```

Here, the `source\characteristic_substructure.py` is the main file `-c`, is a flag that specifies the interest in just the CS (as opposed to outputting all representative structure as well) and `data\folder\FileName.txt` is the path for a file of SMILES strings.

The structure of the existing application can be described with the following simplified UML:

[1] Strictly speaking, VF2 is designed for subgraph isomorphisms (a significantly harder problem), but has shown good performances for graph isomorphisms as well[17].

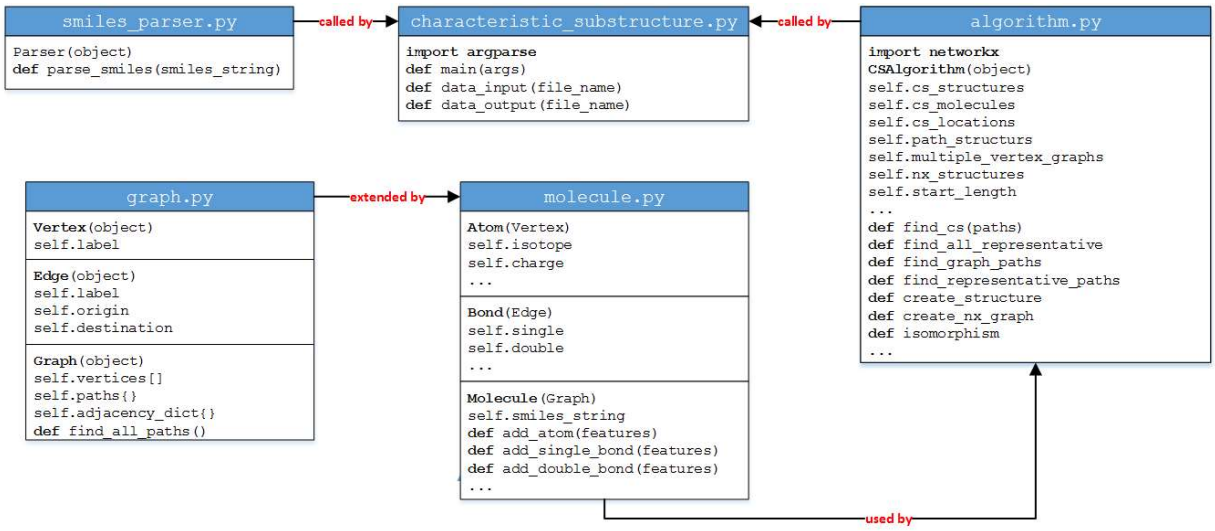


Figure 4.1: Initial Application UML

On the whole this follows a clear structure, with a `molecule` just being a `graph` with more labels. The `smiles_parser` is one parsing function and the `characteristic_substructure` file handles the argument parsing and file IO. Unfortunately, this also means that all of the CS algorithm logic is in the `algorithm.py` file, comprising around 80% of the code. As this is supposed to be extended further, some of the functions will be broken down and separated.

4.1.1 Encountered Issues

Throughout the development a number of issues were encountered, primarily in regard of the correctness and performance of the CS Algorithm (CSA), leading to a variety of amendments and re-implementations.

In the matter of correctness, the CSA would sometimes to produce a **different** output for the **same** input. This means that for some molecule list M the algorithm could produce wildly different CS, which should not be the case for a deterministic algorithm. There were 2 main reasons for this, one of which will be discussed here and another that will be explained in *Section 4.9*. In regard of the original implementation, the reason is due to a mistake in the algorithm that is meant to find paths for the CSA in conjunction with the behaviour of the Python `dictionary`. This path-finding algorithm in the `graph` class uses a variant of a DFS, but forgets to label its vertices as unvisited when it backtracks, resulting in future calls from neighbours to be cut short (Example in *Appendix D*). This results in the algorithm finding a subset instead of all paths, affecting the results of the CSA. This, in conjunction with the application placing its vertices in a `dictionary` (which does not retain their insertion order) resulted in a different list of paths being returned each time.

In regard of performance, some of the application's design decisions led to very high memory usage and some excessive computation. The probably most significant one was that all created graphs were in the same scope as the `CSAlgorithm`, resulting in them never being deleted in spite of being only necessary for a single iteration of l. As a result, attempting to run the application on some test sets such as "lipids.txt"[2] resulted in it consuming over 6GB of memory before crashing. For the run-time, the algorithm determined the best position to attach a given path structure s to the CS by copying the CS for every molecule s was in and then adding the s at each possible position. It would then count the most common position by matching these CS against each other via graph isomorphisms. This could result in run-time performances such as $O((|M| \times isomorphism(CS))^2$ for

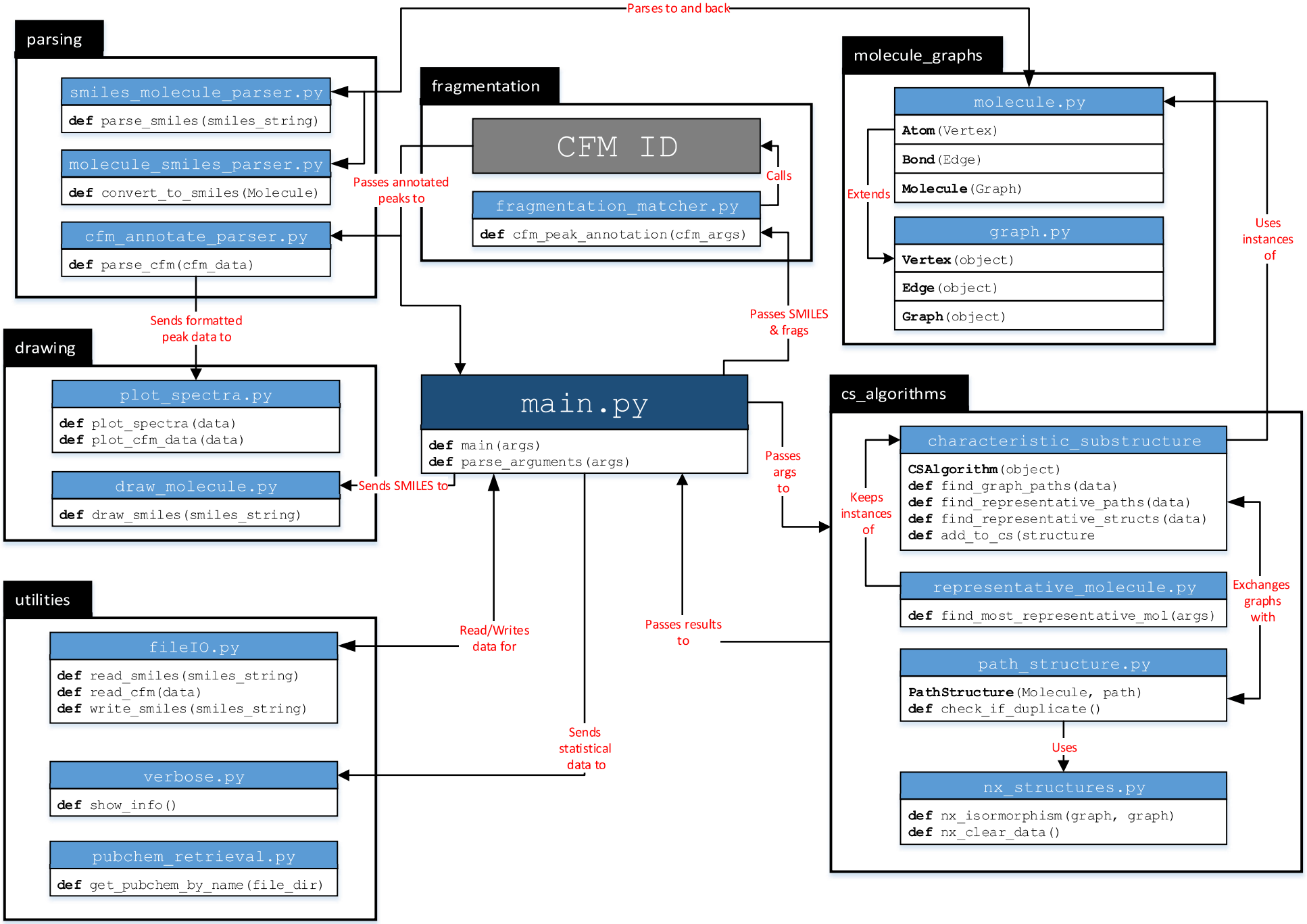
[2]This is a data from the original paper and can be found in the application's test data.

every representative path structure. A much easier alternative is to count and compare the position vertices in the molecules, requiring no isomorphism whatsoever.

The above issues led to the CSA gradually being reimplemented, in addition to the required features.

4.2 Design Overview

Having reviewed the requirements and existing work, the following application structure is be used:


parsing
smiles_molecule_parser.py
def parse_smiles(smiles_string)
molecule_smiles_parser.py
def convert_to_smiles(Molecule)
cfm_annotate_parser.py
def parse_cfm(cfm_data)
Parses to and back
fragmentation
CFM ID
fragmentation_matcher.py
def cfm_peak_annotation(cfm_args)
Calls
Passes annotated peaks to
molecule_graphs
molecule.py
Atom(Vertex)
Bond(Edge)
Molecule(Graph)
Extends
graph.py
Vertex(object)
Edge(object)
Graph(object)
Uses instances of
Sends formatted peak data to
Passes SMILES & frags
drawing
plot_spectra.py
def plot_spectra(data)
def plot_cfm_data(data)
draw_molecule.py
def draw_smiles(smiles_string)
Sends SMILES to
main.py
def main(args)
def parse_arguments(args)
Passes args to
cs_algorithms
characteristic_substructure
CSAlgorithm(object)
def find_graph_paths(data)
def find_representative_paths(data)
def find_representative_structs(data)
def add_to_cs(structure
Keeps instances of
representative_molecule.py
def find_most_representative_mol(args)
Exchanges graphs with
path_structure.py
PathStructure(Molecule, path)
def check_if_duplicate()
Uses
nx_structures.py
def nx_isormorphism(graph, graph)
def nx_clear_data()
utilities
fileIO.py
def read_smiles(smiles_string)
def read_cfm(data)
def write_smiles(smiles_string)
Read/Writes data for
Passes results to
verbose.py
def show_info()
Sends statistical data to
pubchem_retrieval.py
def get_pubchem_by_name(file_dir)

As can be seen in the diagram, the program is divided into 6 packages and 1 controlling main file. Note, that the individual files do not detail every single function, solely the most important ones. This structure will now be explained in terms of the files' and functions' purposes.

The `main.py` file takes commands (as used in the original application) and parses all arguments from the user/caller of the application. It then chooses which process run(e.g finding a CS and/or matching a fragmentation pattern) and sets the necessary configurations. It then calls the respective packages and functions, taking the outputs and passing them to be saved to a file or otherwise processed.

The `cs_algorithms` package handles all CS related algorithms, with the core steps being in `characteristic_substructure.py`. When finding a CS, it makes use of `Molecule` objects and their paths, passing them to `path_structure.py`, which keeps all instances of path structures and maps their vertices to their counterparts in the respective `Molecules`. Additionally, `path_structure.py` also communicates with `nx_structures.py` in order to handle graph isomorphisms. After finding the representative path structures, the construction of the CS is then handled by `characteristic_substructure.py`, with the result being passed to `main.py`.

The `molecule_graphs` package performs the same purpose it did in the original implementation, by keeping the definitions of molecules. It does however, also make use of the `molecule_smiles parser` and `smiles_molecule_parser`, parsing a `Molecule` back to a SMILES string and back on demand.

The `fragmentation` package is responsible for matching a fragmentation spectrum to a provided SMILES string. The `fragmentation_matcher.py` wraps around an external C++ application called *CFM-ID* (*See Section 4.7* and launches it, collecting the outputs. It then passes these back to `main.py` for display and saving, as well as to `cfm_annotate_parser.py`, which parses and formats the results, so they can be plotted by `plot_spectra.py`.

The `parsing` package is used as an abstraction for the aforementioned functions.

The `drawing` package plots both individual spectra and annotated ones using parsed CFM data. Additionally, it also has `draw_molecule.py`, which draws any SMILES string it is passed and saves and resizes the resulting image.

Lastly, `utilities` covers the essentials and miscellaneous features of the application. The `fileIO.py` is responsible for reading and writing a variety of formats, but leaves most of the processing to the respective parsers. The `verbose.py` is linked to a command line setting `-v` or `--verbose` that evolved primarily out of the need to understand and debug the application. As a result, it holds strings and timers to display statistical information about what the application is currently doing. The `pubchem_retrieval.py` file handles the retrieving of data from the PubChem database[22] and is primarily used to acquire testing and evaluation data.

This concludes the overview of the application. The following sections will detail how every respective part was designed and implemented individually.

4.3 Re-designing the CS Algorithm

The following sections detail the implementation of all of the core algorithms, starting with finding the paths from the molecules:

4.3.1 Finding Representative Paths

Finding all paths in a graph can be achieved with a DFS that iterates through every possible route from a given vertex, from every vertex. As can be imagined, this has a potentially *very* high run-time complexity depending on the graph it is used on. For instance, attempting the above on a *complete graph*[3], can be lower bounded by a run-time complexity of $\Omega(n!)$[4] for a complete graph of size n. Fortunately, the metabolite graphs are much more similar to *planar graphs*[5] and tend to have an edge count $|E| << |V|^2$, reducing the risks. In terms of memory, there is the tradeoff of finding all paths (which is faster, but more memory-intensive) vs. finding just those of size l (less memory used, but graphs need to be re-traversed). As the metabolite graphs are typically small (< 200 vertices) and sparse, this decision does not have too great of an impact and we chose to find all paths.

With these decision in mind, here is the actual algorithm:

Algorithm 2: Find all paths

Data: A single molecule m
Result: A list of paths L_S, and a dictionary D_P mappings to the vertices

def *find_all_paths(*m*)*:
    $L_S, D_P = \emptyset$
    $vertex\_stack, path\_stack = \emptyset$
    for $v \in V(M)$ **do**
        path_visit$(v, vertex\_stack, path\_stack, L_S, D_P)$
    return L_S, D_P

def *path_visit(*v, $vertex\_stack$, $path\_stack$, L_S, D_P*)*:
    $v.visited = true$
    $path\_stack.add(v.label)$
    $vertex\_stack.add(v)$
    for $n \in neighbours(v)$ **do**
        if $not\ n.visited$ **then**
            $path\_stack.add(e = (v, n))$
            path_visit$(n, vertex\_stack, path\_stack, L_S, D_P)$
    $path = path\_stack.output()$
    if $not\_duplicate(L_S, path)$ **then**
        $L_S[len(path)].add(path)$
        $D_P[path].add(vertex\_stack.output())$
    $path\_stack.pop()$
    $vertex\_stack.pop()$
    $v.visited = false$

As is shown above, the algorithm uses 2 functions. The first function *find_all_paths()* initialises the data structures and starts the search from every vertex in the molecule. The second function *path_visit()* performs a recursive DFS, maintaining 2 stacks of both the vertices and the path (composed of vertex and edge labels). It traverses all possible neighbours recursively, each time adding a path to L_S based on its length (this is for the CSA's steps through l) and maps the vertices to a specific path in D_P (to build path structures from the path later). An example entry in L_S would be (5:["C-C-C=S=N", "N-C-C-S-O"]), where 5 is the length and the subsequent

[3] A graph where every vertex is directly connected to every other vertex.

[4] This can be calculated by treating the problem combinatorially. If there are n vertices and they can be travelled/arranged in any order, then there are $n!$ combinations and it will take at least that long to find all paths.

[5] A graph sparse enough $s.t$ no edges may intersect or overlap.

list are all paths of length 5. A similar example for D_P is {"C-C-S": v_a,v_b,v_c} where the v_a and the others are vertices in the graph.

One last point of interest is function *not_duplicate()* which is not featured because of its length. Regardless, it serves a very useful purpose. Normally, this algorithm would find **every** path, but a path and its reverse (From $A \rightarrow B$ and $A \leftarrow B$) will create the same path structure. Accordingly, nothing is gained from having both and only considering $A \rightarrow B$ halves all stored paths and created path structures (also halving all isomorphism checks made). This is another improvement that was made with this revision.

Finding the *representative paths* ($s.t\ freq(p) \geq f_m \times |M|$) is fairly straightforward with the given data structures. For any length l take the respective path list from L_S (this is aggregated from all molecules in the implementation) and then for each $p \in L_S$ count in how many molecules' D_P it occurs. If this count is $\geq f_m \times |M|$, then p is representative and $p \in P_r$.

4.3.2 Finding Representative Path Structures

Finding the representative path structures begins with having all representative paths and inducing a subgraph from each one. This would look something like this:

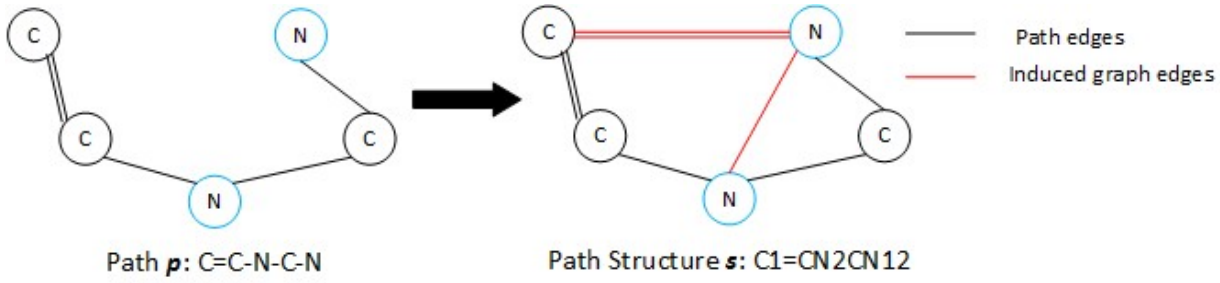


Figure 4.2: Inducing a Path Structure

In the above example the vertices $v \in p$ (held in the aforementioned D_P) in some molecule m are used to induce a new graph s of all edges these vertices share. As a result, it is possible for the path structure to have more edges than the original path and these edges are unlikely to be the same across all molecules. This means that identical paths p_a and p_b have no guarantee that their path structures s_a and s_b are isomorphic. Accordingly, for every $p \in P_r, M$ the algorithm will have to match each induced s_{new} against all existing $s \in S$. Theoretically, this can be upper-bounded space-complexity-wise by creating $O(|S|)$ and time-complexity-wise as $O(|S|^2 \times iso())$, where $iso()$ denotes the cost of an isomorphism. Especially in terms of run-time, this part is the bottleneck of the algorithm, if one considers that $|P_r| \leq |S| \leq |M||P_r|$.

Of relevance is that each path structure keeps a mapping between its vertices and the vertices of the 1 or more molecules it occurs 1 to *k* times in. In the implementation the dictionary for this has the structure of: {`path_struct:[molecule:{struct_vertex: [molecule_vertex]}]`}. This rather reference-intensive mapping will be used when adding structures to the CS, to identify possible locations to attach an s to the CS. Notably, finding the set of representative path structures from this can be done by counting how often (taking the length of all mapped molecules per struct) it occurs in and again comparing this to $f_m \times |M|$, adding it to S_r when applicable.

4.3.3 Adding Structures to the Characteristic Substructure

The final step of adding to the CS can be broken down into making the most representative $s \in S_r$ the base for every other s, establishing how many times (k) s is in M and determining where it can be added. If s occurs $k > 1$ times in at least $H_{iso} \times |M|$ molecules (where H_{iso} is a user-defined constant), it can be added k times. This makes the distinction between adding a path structure once or multiple times. For an example, consider the situation below:

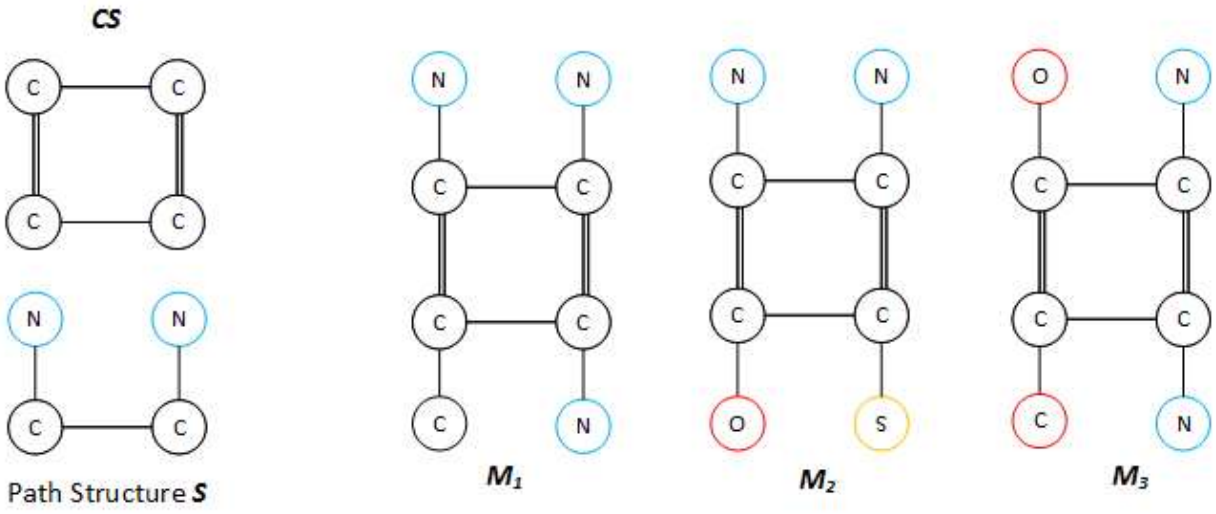


Figure 4.3: Adding a path structure to the CS

Here, a structure "N-C-C-N" is being added. For molecule M_1 there are 2 possibilities and for M_2,M_3 there is one in a different positions. In total, there are 2 equally popular positions in M (see highlighted below).

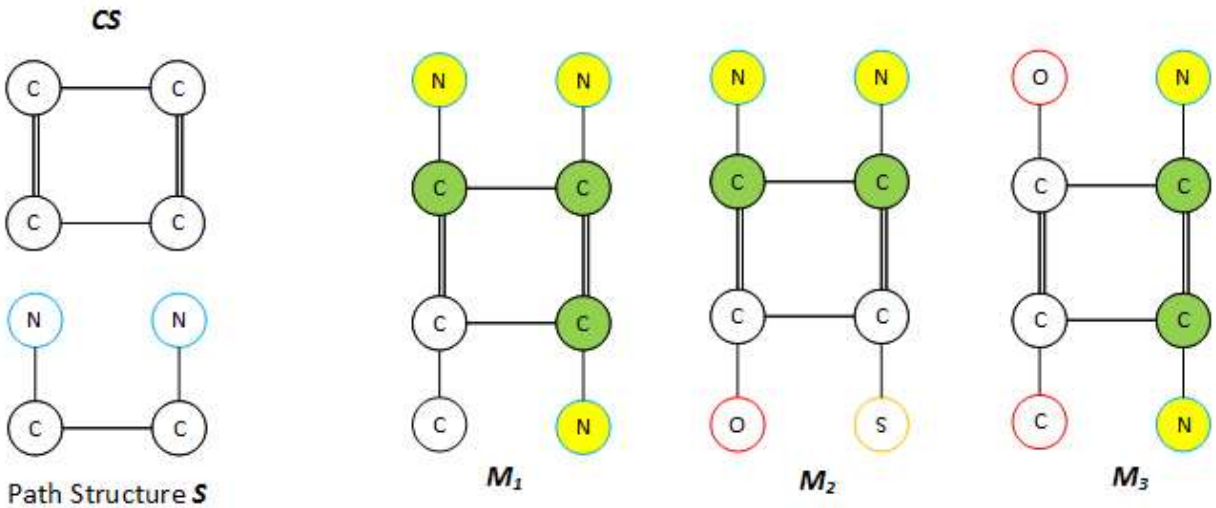


Figure 4.4: Highlighted positions

Here, the green vertices are the possible locations and the yellow vertices the additional vertices of the path structure. Accordingly $k = 2$ and if $H_{iso} \leq 1/3$, then S will be added twice, as it is in M_1. If $H_{iso} > 1/3$, then

it will only be added as it is in M_2 or M_3. At this point, one might ask: "how to choose between M_2 or M_3?" and this is actually a very crucial aspect the original paper[28] does not address, which will be discussed in *Section 4.9*.

As for determining these positions, this requires both the mapping from the previous section and for the CS to have a mapping as well. This second mapping is created when the first s_0 is added/used to create the CS. There, the entire mapping of the path structure to its molecules is updated to be the CS's mapping to these molecules. Accordingly, when another s_{next} wishes to know if there are any possible locations, it needs to iterate through the molecules it is in, to determine if any of its mappings intersect with those of the CS. We can quantify the upper bound of run-time complexity of this to be $O(|V(M)|^2)$, where $|V(M)|$ is the sum of all vertices in M and we make the assumption that the CS and s_{next} are mapped to every vertex in every graph. In reality, this is almost never the case and with the use of Python `dictionaries`, the lookup can even be bounded $O(|V(M)|)$. After finding the intersecting groups of locations L, these need to be counted by frequency. A simple way to do this is to compare every group $l \in L$ to every other group in L. This implies a run-time cost of $O(L^2)$, which can be reduced with sorting the vertices, although L is typically too small to make a noticeable difference. To clarify this process somewhat, here is an example:

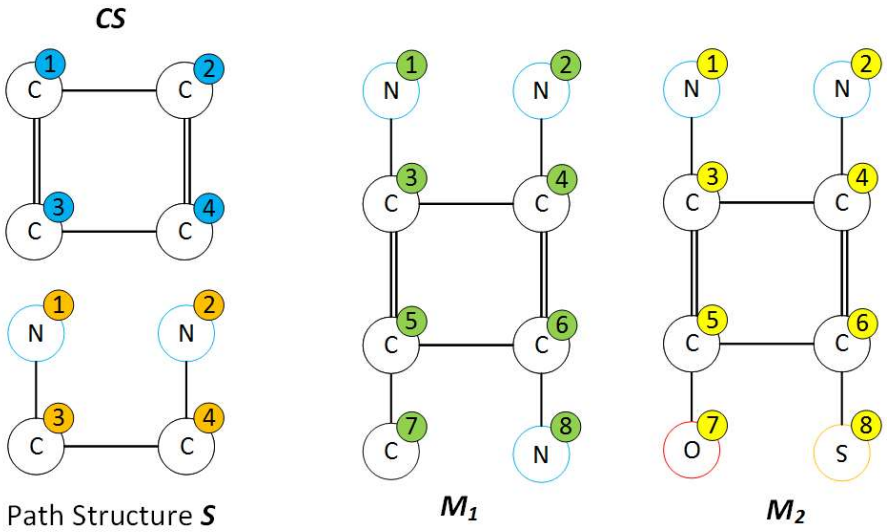


Figure 4.5: Indexed Example

The above molecules, CS and S are indexed with IDs to distinguish them. Here is how they hold mappings of each other:

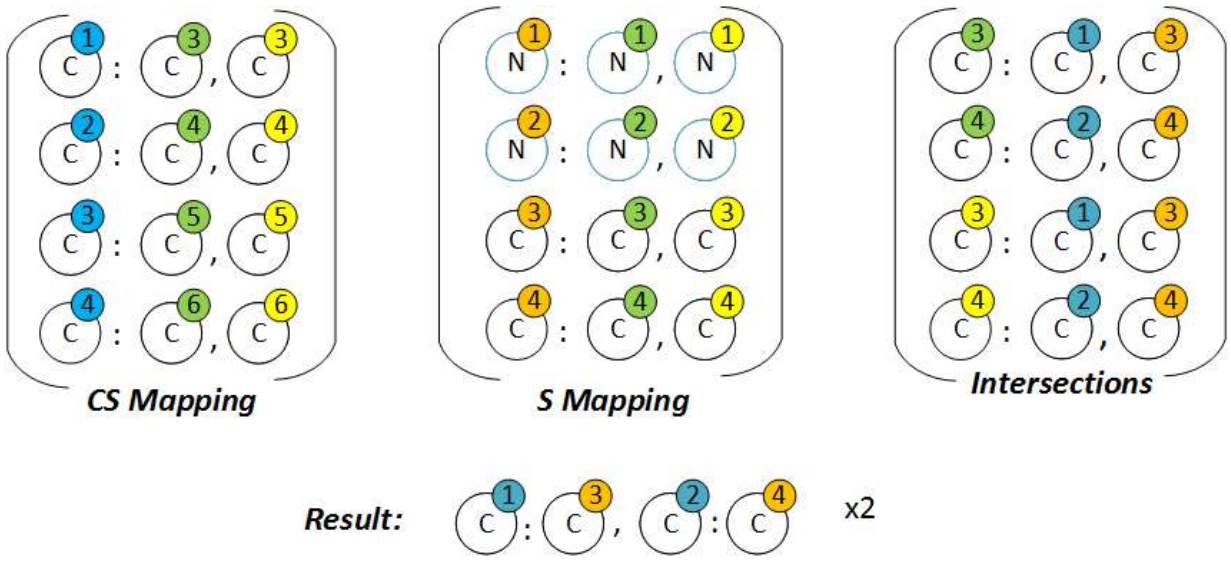


Figure 4.6: Mappings between graphs

Here, the mappings show which molecule vertices the CS' and s' vertices correspond to. By finding those they have in common, a location is obtained and when performed over every molecule, the most frequent one is determined. After that, the all non-intersecting vertices of s and all edges in s are copied to the CS.

The reimplementation of the CS algorithm reflects all of the aforementioned design and performance decisions. The respective source code for these steps can be found in `graph.py` for the *find_all_paths()* algorithm and `characteristic_substructure.py` for the path structure and CS algorithms.

4.4 Enabling Multiple List Inputs

With the CS corrected and understood, the capability to use multiple molecule input lists was added. In the CS algorithm, this addition requires to keep track of which list a molecule is part of and can be done by adding this as an attribute to the `Molecule` objects themselves. Further, the setting of a parameter threshold f_{list} or f_l, which then assesses whether a path or path structure is in at least $f_l \times |L_M|$ lists and is used as a second condition together with $f_m \times |M|$. After this, the algorithm can behave the same for adding path structures to the CS.

In the implementation, this logic is integrated into `characteristic_substructure.py` (checking thresholds) and `fileIO.py` (parsing lists). A list itself can be recognised 2 ways: either by having SMILES strings in multiple files and specifying these and/or by separating a list of SMILES strings with a new line. Here is an example:

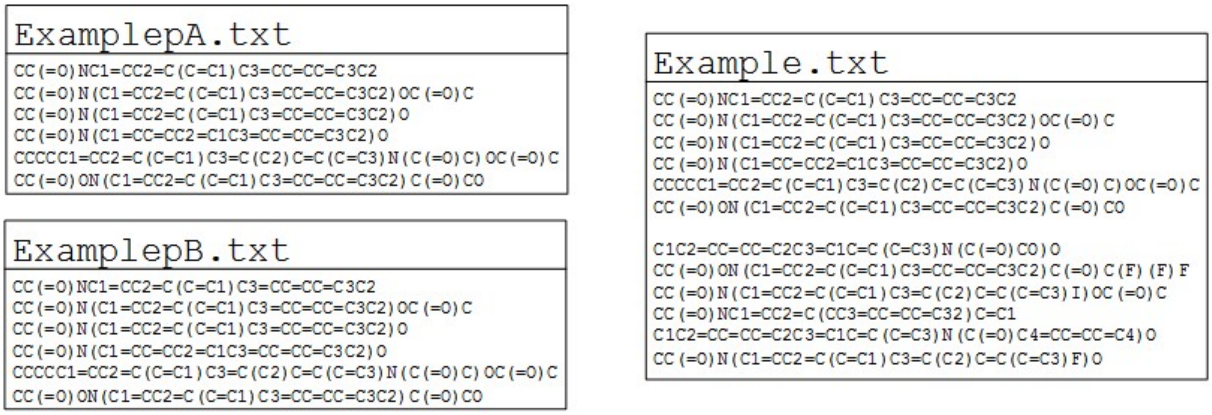


Figure 4.7: Example of how lists are formatted as the input to the application.

In the above, `ExamplepA.txt` and `ExampleBp.txt` are treated synonymous to `Example.txt` by the application. When calling `main.py`, this corresponds to:

```
python main.py cs ...\ExampleA.txt ...\ExampleB.txt
python main.py cs ...\Example.txt
python main.py cs ...\Example.txt -lt 0.7
```

Here, the `cs` command specifies a characteristic substructure and the file names are for the SMILES strings. The 3rd command also has a flag `-lt` which is the shortened version of `--list_threshold` and takes a floating decimal from 0 to 1.

4.5 Finding the Best-fitting Molecule

Determining the best-correlating molecule between 2 CS_1 and CS_2 is best done by observing how much they contribute to the representativeness of either CS underlying paths and path structures. As path structures depend on paths, we will only consider the latter for this assessment. Measuring this level of "contribution" can be done in any number of ways. In this project, it was decided to iterate through each molecule $m_t \in M_T$, (where M_T is the list of molecules we want to fit) and add it to the molecule sets of S_{M1} and S_{M2}. We then find the representative path structures for both of these and then take the weighted sum for each one as follows:

$$\alpha_i = \sum_{i=0}^{|S_r|} len(s_r) \times freq(s_r) \tag{4.1}$$

where $i \in [1, 2]$, α_i is the sum, corresponds to which molecule set we are using, $len()$ is the length of a representative path structure $s_r \in S_r$ and $freq()$ is how representative it is (some value from [0..1]). This sum accounts for how many path structures there are, favouring larger ones with $len()$ and more representative ones with $freq()$. With this, we would have 2 scores α_1 and α_2 for S_{M1} and S_{M2}, whose sum could then represent the quality of a match. Yet, it may also be of interest to consider these scores relative to each other to reflect that the chosen molecule m_T actually correlates with both sets S_{M1}, S_{M2} and not just one. An example of this would be for instance, $\alpha_1 = 0$ (indicating no structures whatsoever for S_{M1}) and $\alpha_2 = 100$ (indicating some

structures in S_{M2}) is considered equally good vs. $\alpha_1 = 50$, $\alpha_2 = 50$. Thus, we will take the mean($\bar{x}$) and standard deviation(σ) as follows[4]:

$$\bar{x} = \frac{\alpha_1 + \alpha_2}{2}, \qquad \sigma = \sqrt{\frac{(\bar{x} - \alpha_1)^2 + (\bar{x} - \alpha_2)^2}{2}} \tag{4.2}$$

and calculate the final score of a correlation to be $score = \bar{x} - \sigma$. The effectiveness/accuarcy of this assessment technique will be discussed in *Section 5.2*.

This assessment technique is implemented in `representative_molecule.py`. It begins by parsing the SMILES files and then iterates through each $m_T \in M_T$, finding the path structures of $S_{M1} + m_T$ and $S_{M2} + m_T$ and scoring them as described earlier. It then builds CS_1 and CS_2 with the best scoring molecule, outputting the SMILES strings for all 3. Here are an example input and output of the program:

```
Input:
python main.py rm file_A.txt file_B.txt file_C.txt

Output:
COC1=C(O)C=CC(C=CC(O)=NCCC2=CC=C(O)C=C2)=C1 106.0
C1NC2=CC(=C(C=C2S(=O)(=O)N1)S(=O)(=O)N)Cl 89.4
C1=C(C(=O)NC(=O)N1)I 89.0
CC(=NOS(=O)(=O)O)SC1C(C(C(C(O1)CO)O)O)O 104.466666667
CC1CCC2C(C(=O)OC3C24C1CCC(O3)(OO4)C)C 101.533333333

Best candidate: COC1=C(O)C=CC(C=CC(O)=NCCC2=CC=C(O)C=C2)=C1
Score: 106.0
CS A: OCC(OCC1O2)C(C1O)(COC1C)CC12
CS B: C(C=C(C=1)O)=C(C=1)CC(O)=O
```

In the input, the token `rm` chooses to find the representative molecule. Of the 3 files, `file_A.txt` and `file_B.txt` are the SMILES sets S_{M1} and S_{M2} whereas `file_C.txt` is M_T. As S_{M1} and S_{M2} can be composed of any number of SMILES lists, it is possible to input any number into the command line as well, with the restriction that the number lists for S_{M1} and S_{M2} should be divisible by 2, to allow for sensible splitting. The output shows the SMILES string and scoring for each $m \in M_T$ and then the best candidate and respective CSs that were built with it.

4.6 Converting a Molecule Graph to SMILES

A `Molecule` graph can be converted into a SMILES into 2 steps: creating a *spanning tree*[6] and then parsing this tree. The construction of this tree is done by performing a BFS from a vertex, detecting any rings and removing

[6] A tree is an undirected graph where any 2 vertices are connected by exactly 1 path. This implies that there are no cycles in the graph and every vertex is reachable from any other vertex. Extending from this, the spanning tree T is a tree that spans a graph G, including all of its vertices[3].

an edge and labelling the vertices. Here is a pseudo-code of this process:

Algorithm 3: Creating a spanning tree

Data: A graph G
Result: A spanning tree T

$r = v_0 \in G$
$Q = \emptyset$
$Q.add(r)$
while $Q \neq \emptyset$ **do**
    $v = Q.pop()$
    $v.visited = true$
    for $n \in neighbours(v)$ **do**
        if $notn.visited$ **then**
            $e = (v, n)$
            $e.visited = true$
            $Q.add(n)$

Where r is an arbitrarily chosen root vertex, Q is a queue and T can be derived from the vertices and edges labelled "visited". This is a fairly simple algorithm with a run-time complexity of $O(V + E)$ (as every vertex and edge are visited once) and a space complexity of $O(V + E)$ (although it is typically less as no copy of the graph is produced). Here is an example of what a `Molecule` graph would look like as a result:

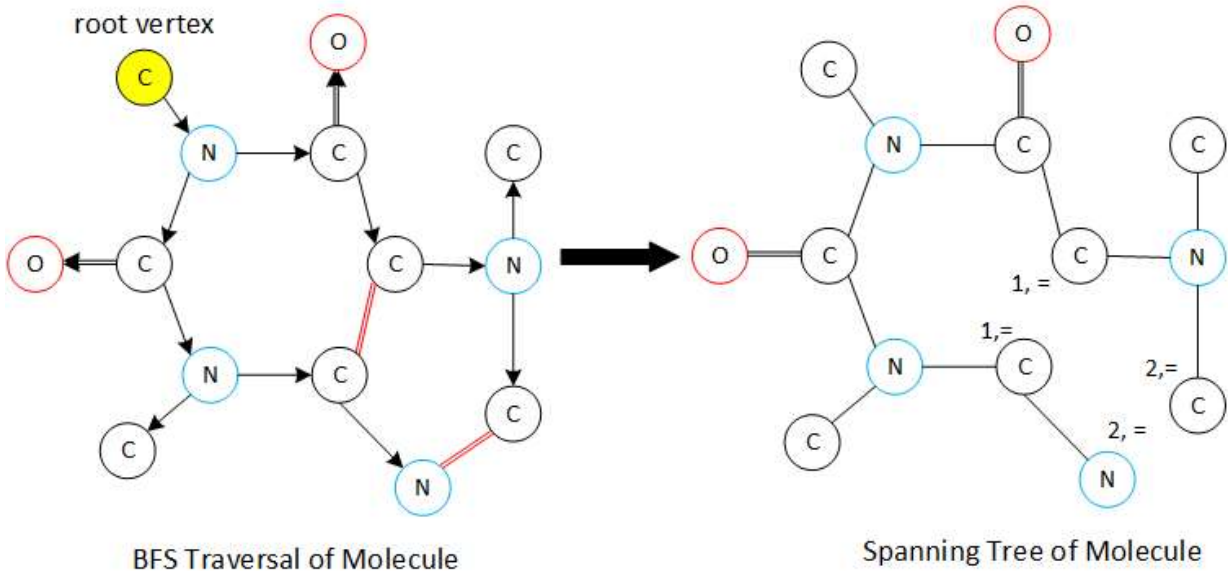


Figure 4.8: How a spanning tree is created from a Molecule graph.

In the above, the left graph shows how the BFS traversal would work from the highlighted root vertex and the red edges are those that are unvisited. The parsing process for SMILES can then be started as a recursive DFS from any vertex. Each recursive step can be summarised by:

1. Convert all of the `Atom`'s properties (e.g charge) into a string.
2. Check each neighbour for rings and add the respective ID label (tracked by a global array).

3. Determine whether to create a branch based on the neighbour count.

4. Traverse any unvisited neighbours.

In the implementation, the spanning tree algorithm is part of the `Graph` in `graph.py` and the SMILES parser is in its own file, accordingly named `molecule_smiles_parser.py`.

4.7 Comparing a Fragmentation Pattern with the CS

Comparing a fragmentation spectrum with the CS, initially required determining a suitable technique. A suggestion made by the stakeholders led to the investigation of *in silico fragmentation*[38]. This technique involves building a fragmentation tree where the root is the intact molecule and its children are determined by breaking the molecule at certain bonds, producing fragments. The algorithm uses a BFS to generate children as it goes down the tree to a predefined search depth (as the size of the tree grows exponentially). Finally, the silico fragments are matched against the original fragmentation pattern, assessing the quality by determining how far each suggested fragments' mass/charge deviates from that of the peak.

It was initially considered to implement this technique, but further research revealed that this is just the foundation of an algorithm that has many additional heuristics and optimizations. Instead, existing software was considered, the first one being the Java-based software *Metfrag*[33] which implements the aforementioned method. However, it did not meet the requirements well and was of difficult feasibility. *Metfrag* is more geared towards larger scale database analyses, does not support SMILES notation and generally, is not very convenient for comparing just one molecule and one spectra. In the end, we chose to use *CFM-ID* by Felicty Allen *et Al.*, which uses *peak assignment*; a technique described earlier in *Section 2.4*. The *CFM-ID* software is usable via a web-server[37] as well as a collection of C++ applications[16], where for portability purposes, this project chose to make use of the latter.

CFM-ID is composed of a collection of .exe files and their source code. In our case, we are interested in 2 of these: `cfm-train.exe` and `cfm-annotate.exe`. The former trains a model using some peak data and their respective molecules and the latter assigns peaks to molecules based on this training. As the project does not have large quantities of suitable training and test data, it was decided to forgo the training phase and use a pre-trained model for metabolites measured via *capillary electrophoresis mass spectrometry* (MS)[32]. As our evaluation data is obtained from *liquid chromatography MS*[23] this was not a perfect match, but sufficient for testing the application. In order to annotate a peak, `cfm-annotate.exe` requires the following arguments[7]:

```
cfm-annotate.exe <smiles_string> <spectrum_file> <id> <ppm_mass_tol>
 <abs_mass_tol> <param_file> <config_file> <output_file>
```

And here is a table of what these arguments correspond to:

[7] This project is using *CFM-ID* v2.1. This may affect some bugs and interface specifics in different versions.

Parameter	Examples	Purpose
`smiles_string`	CC=C, O1COS1	Any SMILES string (exception of . separated fragments)
`spectrum_file`	see *Appendix E*	A file with peak 1-3 peak spectra in it. This will be our data.
`id`	cytosines, bananas	A name given to the result. Can be anything.
`ppm_mass_tol`	5	OPTIONAL - A mass tolerance in ppm. Determines how far a peak and fragment mass may deviate to match.
`abs_mass_tol`	0.03	OPTIONAL - A mass tolerance in Da. Will choose maximum of this and ppm.
`param_file`	-	A file defining options for the annotation process (thresholds, heuristics, etc.)
`config_file`	-	The training model data.

Table 4.1: `cfm-annotate.exe` parameter list.

Of the above arguments, the most important ones are the SMILES string and spectrum data, which will be used to compare a CS to the molecule list M's original MS measurement. A typical spectrum file is composed of a list of mass/charge ratios and their intensity. Additionally, an example output of `cfm-annotate.exe` can be seen under *Appendix F*.

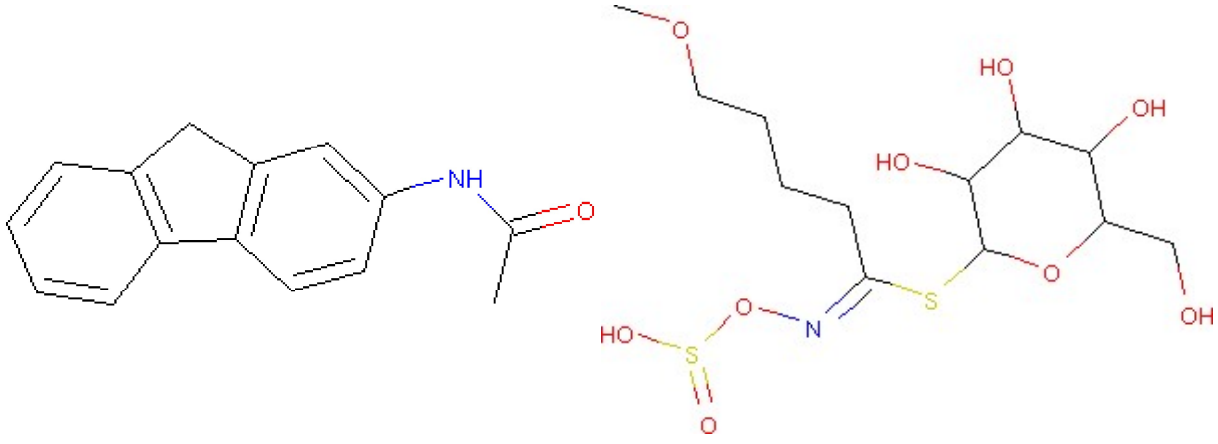
The project's application wraps around `cfm-annotate.exe` using Python's `subprocess` module. This takes any given parameters and launches the .exe with it, collecting the output when it terminates. As we have the means to convert a `Molecule` into a SMILES string, it is possible to pass `cfm-annotate.exe` data and observe the results. In the implementation, the wrapper is implemented in `fragmentation_matcher.py` and can be passed a spectrum file (See *Appendix E*) for an example from `main.py`. The respective command is an optional flag for creating a characteristic substructure and looks like the following:

```
python main.py cs -f <spectrum file> <any number of SMILES files>
```

Here `-f` is short for `--fragmentation_pattern` and can be placed anywhere in the command together with exactly 1 spectrum file.

4.8 Plotting Data

An additional requirement was to plot the CS and any fragmentation spectra. The plotting of a molecule is relatively complex and with some investigation a 3rd party library `rdkit`[27] was chosen to perform this task. There a few minor image processing steps involved (requiring the fairly standard image library `PIL`[29]), but generally this was straightforward to implement. Here are 2 example CS molecules drawn with this library:



(a) CS of `acetylaminofluorenes.txt` dataset. (b) CS of `glucosinolates.txt` dataset.

Figure 4.9: `rdkit` molecule drawings.

Notably, the datasets mentioned under *Figures 4.9a, 4.9b* are sourced from the original CS paper[28]. As for the images, they depict the molecules in a legible fashion, although a vector graphic would be more aesthetic.

In the implementation this process is managed by `draw_molecules.py` which writes images like the above to an output folder for viewing. This image output can be called by the following commands from `main.py`:

```
python main.py cs -img <smiles files>
python main.py rm -img <smiles files> <special smiles list>
```

As can be seen, the `-img` flag enables images to be drawn of both the individual CS algorithm as well as representative molecule algorithm from *Section 4.5*. Additionally, as this feature does create 2 library dependencies, it is optional, implying that the rest of the application will run without it, provided no attempt is made to draw an image.

The plotting of spectra has also been implemented, using the standard graph plotting library `matplotlib`[21] and data processing library `numpy`[15]. It can be called by the application to either plot a regular fragmentation spectra or an annotated fragmentation spectra using some parsed data produced by `cfm-annotate.exe`. Here is an example plot of an annotated spectrum:

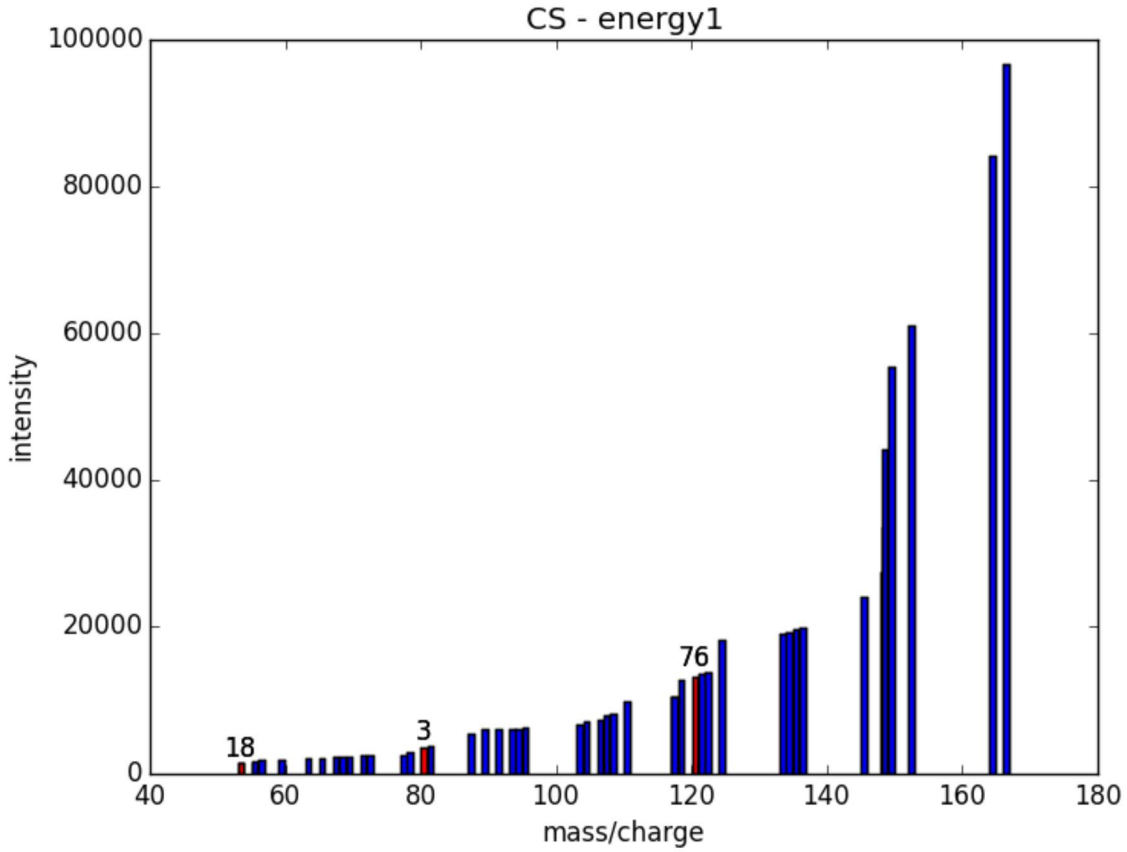


Figure 4.10: Example spectrum produced from annotated peaks.

As is displayed by the example, the application plots peaks of the original spectrum in blue and highlights any sufficiently matching peaks in red, together with the ID they are referred to in the cfm-data. In the above example, the peak of ID #76 is mapped to the SMILES string CC#[CH+]N1C=CC=CC1.

The actual implementation of the plotting involves a few steps, starting with the parsing of the `cfm-data`, which converts the raw output string `cfm_annotate_parser.py` into a collection of `dictionaries` and lists to interpret. The `plot_spectra.py` then takes this reformatted data and plots a graph as shown above. This feature is used automatically with any call to annotate a spectrum, so there is no explicit command for it. However, it is possible to state a threshold that defines what constitutes a sufficiently close matching in the plotting process (to avoid showing too few or too many annotated peaks). This is the respective command:

```
python main.py cs -f <fragmentation pattern> -cfm <threshold> <smiles files>
```

Here, `-cfm` or `--cfm_minimum_score` is the flag and it can be followed by any positive number. It should be noted though, that typical confidence measures from `cfm-annotate.exe` range from 0-100, so it is advised to use values within that range for meaningful results.

4.9 Ambiguities & Potential Problems with the CSA

During the implementation of all the above mechanisms a number of ambiguities became apparent with the original characteristic substructure algorithm's description [28]. The most important one is what causes the random outputs in the input problem mentioned first in *Section 4.1.1*: there is no specification on how to break ties.

Whenever data is ordered or sorted, there is (for most datasets) a possibility that some values $v_i, v_j \in D, v_i = v_j$, for $j \neq i$ for some dataset D. The process of choosing v_i or v_j to be first (by some either meaningful or arbitrary metric) in the ordering is known as *breaking ties*. In the case of the CSA, there are 3 situations in which data needs to be ordered.

The first is when determining how many paths $p \in P$ are representative and this has no impact, given that any paths of equal frequency $freq(p_r1) = freq(p_r2)$ will still be used to build path structures of potentially different representativeness. However, once the algorithm has decided which path structures are representative, it needs to order them by representativeness to add them to the CS. There, it will likely have to break ties, which means that path structures of equal $freq(s_r)$ have no deterministic definition of which is added first. In the implementation, this leads to Python's sorting choosing this arbitrarily (presumably by some random state information), resulting in a different order of representative structures each run-time.

This would be relatively inconsequential, were it not for the fact that the order in which a path structure s_r is added to the CS, affects what future path structures $s_{r+1}, s_{r+2}....s_{r+k}$ can be added to it. A simple example of this can be seen below:

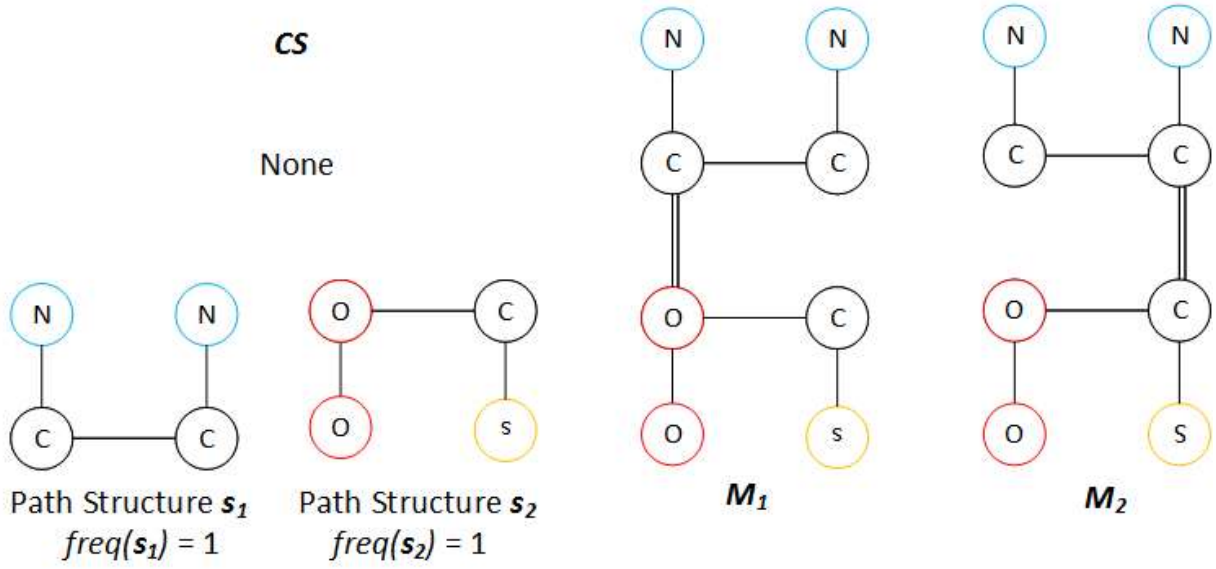
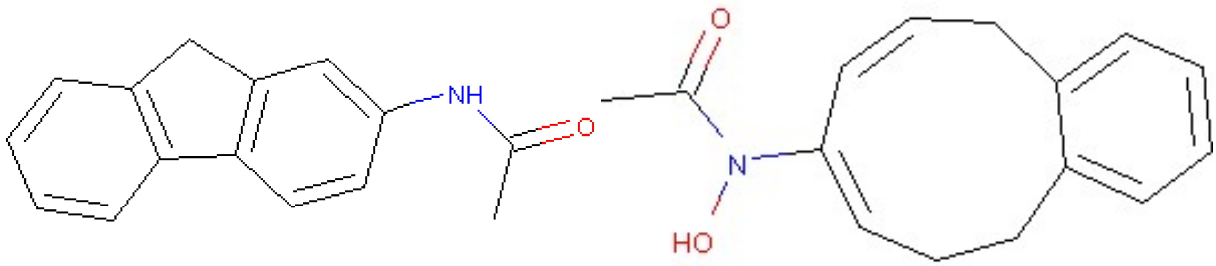


Figure 4.11: Breaking a tie between s_1 and s_2.

In *Figure 4.11* there are 2 possible path structures[8], both with a representativeness of 1. If s_1 is chosen, s_2 can't be added because it shares no vertices with s_1 and vice versa. As a result, there can be 2 possible CS, based on how the ties are broken. The above situation is a very simple case, but the real data the algorithm has to break ties many times. For instance, a count of the tie-breaks for the `adenines.txt` dataset (approx 40 molecules of 15-25 atoms each) yielded 103. As such, there is really no guarantee as to whether a produced CS is the

[8]It might be tempting to think both M_1 and M_2 are the same, but the difference in that bond's position makes them non-isomorphic.

correct characteristic substructure (as far as there is a "correct" one). Here is an example of the earlier used `acetylaminofluorenes.txt` dataset:



(a) CC(N(C(=CC(=C1C2=CC=3)CC2=CC=3)C=C1)O)=O (b) CC(N(C(C=CCC1=CC=2)=CCCC1=CC=2)O)=O

Figure 4.12: Result of different tie breaks in CS algorithm.

The above are both the results of the same CS algorithm, the molecule on the left being the correct molecule (according to the data set) and the one on the right being incorrect. This illustrates a reliability issue with this algorithm that will have to be addressed in future work. For now, the command line does enable an option to enforce order (but does not guarantee it will find the correct CS):

```
python main.py -nr <smiles_strings>
```

The flag `-nr` or `--not_random` orders the path structures by their creation, which depends on the path and molecule creation order. This is static, as the order in which molecules are instantiated is determined by the input files themselves. Lastly, the 3rd tiebreak that can result is the choosing of positions for a path structure to be added to the CS (See *Figure 4.4* for an example). This is also regulated by the same flag.

4.10 Miscellaneous Enhancements

In addition to the above implementations, a list of minor additions was made to make the application easier to use. First, is the `fileIO.py`, that will arrange for any major output data such as CS, peak annotations, or images, to be saved to an output folder in a project directory named `output_data`. The default name for this folder is the name of the first SMILES file in the command line arguments. Alternatively, it is also possible to choose a name via the `-o` or `output_name` flag.

Another minor feature of note, is the `-v` or `--verbose` flag. This enables the application to be toggled between a "results-only" mode (ideal for calling by other programs, supporting easier integration) and an "exploratory mode". The latter mode will output all sorts of information such as time taken for a step, which structures are added to the CS, how many tie breaks there are and much more. There are also more flags/configurations available, all of which can be seen under *Appendix G*.

Chapter 5

Evaluation

The application was evaluated in different ways for each features, so they will be covered individually. These evaluations included test manual walkthroughs of the application code, feature testing, performance testing, testing with stakeholder data and an acceptance meeting were the features were reviewed. However, on the whole this is a difficult application to assess as it is for research and there thus is never necessarily a right or even existing answer/output.

5.1 CS Algorithm

The CS algorithm had to be tested as it was re-implemented and this was done by through some feature testing of the individual steps, which eventually revealed the issue discovered in the previous chapter. It was also tested repeatedly via series of manual walkthroughs such as the one shown in *Appendix B*. Additionally, it was also tested on individual features such as the finding of all paths and the representativeness tests. Further, the application was also tested against the data-set of *Ludwig et Al.* and did manage to reproduce the correct molecule (as shown in the earlier section). However, due to the breaking ties issue, it is a bit difficult to reason whether this means the algorithm works as it was intended.

In regard of performance, it would not really be realistic to compare this application to the previous one, as they now perform different functions and the differences in correctness also reflect in that more computations need to be completed by the re-implemented CS. A simple example of a performance difference though, is that the new algorithm consumes 1.3GB of memory for the largest data set `lipids.txt` (22 molecules, 123 atoms avg.), whereas the original application would run out of memory.

5.2 Best-Fitting Molecule & Fragmentation Spectra

The core features of the application were evaluated individually using some *CFM-ID* test data as well as some data provided by the stakeholders (a report of which can be seen under *Appendix H*). This was submitted to the stakeholders but has not received any feedback as of time of submission, but nevertheless, shows the presence of the functionalities.

In regard of the scoring technique for the best-fitting molecule, this was used in the aforementioned report to observe how well feruloyltyramine correlated with the classes of ethyl and ferulic acid. When this molecule was placed into the evaluating list with a random group of other molecules, it generally outperformed (implying it

had the best score), but did at other times produce a low score. With more investigation, it was revealed that the scoring method generally favoured larger molecules as these would have more paths and thus a likelier chance to increase the representativeness of the path structures. As such, it is not really suitable and a more sophisticated approach should be considered.

5.3 Other Features

Some of the other features such as the SMILES parser were evaluated for correctness in order to ascertain a valid and correct SMILES string was produced. These tests were performed using strings and graph from PubChem[22], ascertaining whether a built graph would turn into the correct SMILES. Another test (given both features were available) was to pass the SMILES string through the `smiles_molecule_parser.py` and then observe whether `molecule_smiles_parser.py` would produce the same SMILES string, which it did.

Chapter 6

Conclusion

6.1 Summary

On the whole the project was a success. The individual requirements have been met with the respective implemented features. Additionally the characteristic substructure algorithm is now correctly working and more performant than it originally was, making it more scalable. However, more time would have been necessary to fully test every aspect of the application given its complexity and potentially diverse input data.

6.2 Challenges

Many challenges were encountered during this project. These begun with having no familiarity with the field of metabolites and continues with the complexity of the task at hand. A lot of research had to be made that did not necessarily prove relevant and it would have likely been impossible, were it not for an occasional meeting with the project stakeholder. Another matter, was the deceptively functional state of the original application, which as well-designed, but just buggy enough to be noticed at the wrong times. Having to fully understand and rework the characteristic substructure algorithm cost the project a lot of time that could have been used to refine and test more features.

6.3 Future Work

There are many improvements that can be made to this project. To begin the `molecule_smiles_parser` does not output the formally used canonical SMILES string. For better understanding of results, this could be improved[36]. Next there is the fact that the `rdkit` library will sometimes refuse to draw a SMILES string because it is no chemically correct in terms of valence (the CS algorithm does not adhere to these rules). That and the fact that it servers a myriad of other purposes (such as machine learning functions), makes it a bit of a heavyweight library for the function it serves. An alternative would be desirable.

In regard of performance, the crucial memory problems have been addressed, but it would nevertheless be an improvement to replace `NetworkX` with an isomorphism algorithm that does not need to produce graph copies. Some possibilities in this direction is to generate keys such as the Universal Chemical Key (UCK)[18] and take their MD5 hash to guarantee isomorphism within very high certainty. Alternatively, as many metabolites

are planar graphs, the according isomorphism algorithms of this style could be used[19] to used to improve performance.

Lastly and probably most importantly, a technique or agreement should be devised on how to break ties in the CS algorithm in order to give it more stability. As the entire application makes use of it, resolving this matter could prove very positive to its utility and functionality. A suggestion in that regard might be to explore possible states where computationally affordable, choosing the sequence that permits as many representative structures to be added as possible (within the tie-breaking).

Bibliography

[1] Glasgow polyomics reasearch page. http://www.polyomics.gla.ac.uk/.

[2] *NIST Dalton Definition*. http://physics.nist.gov/cgi-bin/cuu/Value?tukg.

[3] Spanning tree. http://mathworld.wolfram.com/SpanningTree.html.

[4] Standard deviation. http://mathworld.wolfram.com/StandardDeviation.html.

[5] Chemical applications of graph theory. part i. fundamentals and topological indices. *Journal of Chemical Education*, 65(7):574, 1988.

[6] *OpenSMILES Standard*, 2007. http://www.opensmiles.org/opensmiles.html.

[7] *NetworkX library and documentation*, 2016. https://networkx.github.io/index.html.

[8] Felicity Allen, Russ Greiner, and David Wishart. Competitive fragmentation modeling of esi-ms/ms spectra for putative metabolite identification. *Metabolomics*, 11(1):98–110, 2014.

[9] Felicity Allen, Allison Pon, Michael Wilson, Russell Greiner, and David S. Wishart. Cfm-id: a web server for annotation, spectrum prediction and metabolite identification from tandem mass spectra. *Nucleic Acids Research*, 42:94–99, 2014.

[10] László Babai. Graph isomorphism in quasipolynomial time. *CoRR*, abs/1512.03547, 2015.

[11] Serafim Batzoglou Chuong B Do. What is the expectation maximization algorithm. *Nature*. http://www.nature.com/nbt/journal/v26/n8/full/nbt1406.html.

[12] Jim Clark. Fragmentation patterns in the mass spectra of organic compounds, February 2014. http://www.chemguide.co.uk/analysis/masspec/fragment.html.

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

[14] Human Metabolome Database. http://www.hmdb.ca/spectra/ms_ms/520.

[15] Numpy Developers. *Numpy Documentation*, 2016. http://www.numpy.org/.

[16] Felicity Allen et Al. Cfm c++ repository. https://sourceforge.net/p/cfm-id/wiki/Home/.

[17] P. Foggia, C. Sansone, and M. Vento. A performance comparison of five algorithms for graph isomorphism. In *in Proceedings of the 3rd IAPR TC-15 Workshop on Graph-based Representations in Pattern Recognition*, pages 188–199, 2001.

[18] Robert Grossman, Pavan Kasturi, Donald Hamelberg, and Bing Liu. An empirical study of the universal chemical key algorithm for assigning unique keys to chemical compounds. *Journal of Bioinformatics and Computational Biology*, 2(1):155–171, 3 2004.

[19] J. E. Hopcroft and J. K. Wong. Linear time algorithm for isomorphism of planar graphs (preliminary report). In *Proceedings of the Sixth Annual ACM Symposium on Theory of Computing*, STOC '74, pages 172–184, New York, NY, USA, 1974. ACM.

[20] Franziska Hufsky, Kerstin Scheubert, and Sebastian Bcker. Computational mass spectrometry for small-molecule fragmentation. *TrAC Trends in Analytical Chemistry*, 53:41 – 48, 2014.

[21] John D. Hunter and Technology Fellowship. *Matplot Lib Documentation*. John Hunter Technology Fellowship, 2016. http://matplotlib.org/.

[22] National Center Biotechnology Information. Pubchem database, March 2016. https://pubchem.ncbi.nlm.nih.gov/about.html.

[23] Sean Gottlieb Jennifer Betancourt. Liquid chromatography. *UC Davis Chemistry Wiki*, 2008. http://chemwiki.ucdavis.edu/Core/Analytical_Chemistry/Instrumental_Analysis/Chromatography/Liquid_Chromatograp

[24] Michael R. Garey; David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness (Series of Books in the Mathematical Sciences)*. W. H. Freeman, 1979.

[25] Douglas B. Kell and Royston Goodacre. Metabolomics and systems pharmacology: why and how to model the human metabolic network for drug discovery. *Drug Discovery Today*, 19(2):171 – 182, 2014. System Biology.

[26] Gross L. *Handbook of Graph Theory*. CRC Press Inc, 2003.

[27] KNIME Labs. *RDkit library documentation*, 2016. http://www.rdkit.org/docs/Overview.html.

[28] Marcus Ludwig, Franziska Hufsky, Samy Elshamy, and Sebastian Böcker. Finding Characteristic Substructures for Metabolite Classes. In Sebastian Böcker, Franziska Hufsky, Kerstin Scheubert, Jana Schleicher, and Stefan Schuster, editors, *German Conference on Bioinformatics 2012*, volume 26 of *OpenAccess Series in Informatics (OASIcs)*, pages 23–38, Dagstuhl, Germany, 2012. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[29] Fredrik Lundh and Contributors. *Python Image Library Documentation*, 2008. http://effbot.org/imagingbook/pil-index.htm.

[30] Mary McDowall. Structural comparisons for small molecules. Master's thesis, University of Glasgow, School of Computing Science, September 2015.

[31] Dr. Tomislav Mestrovic. What are metabolites. *News Medical*, 2015. http://www.news-medical.net/life-sciences/What-are-Metabolites.aspx.

[32] Juliet Precissi. Capillary electrophoresis. *UC Davis Chemistry Wiki*, 2000. http://chemwiki.ucdavis.edu/Core/Analytical_Chemistry/Instrumental_Analysis/Capillary_Electrophoresis.

[33] et Alia S. Wolf. Metfrag web interface. http://msbi.ipb-halle.de/MetFrag/.

[34] Steve S. Skiena. *The Algorithm Design Manual*. Springer, 1997.

[35] David Weininger. Smiles, a chemical language and information system. 1. introduction to methodology and encoding rules. *J. Chem. Inf. Comput. Sci.*, 28(1):31–36, February 1988.

[36] David Weininger, Arthur Weininger, and Joseph L. Weininger. SMILES 2. Algorithm for Generation of Unique SMILES Notation. *Journal of Chemical Information and Computer Science*, 29(2):97–101, 1989.

[37] wishartlabs. Cfm peak annotation. http://cfmid.wishartlab.com/assign.

[38] Sebastian Wolf, Stephan Schmidt, Matthias Müller-Hannemann, and Steffen Neumann. In silico fragmentation for computer assisted identification of metabolite mass spectra. *BMC Bioinformatics*, 11(1):1–12, 2010.

Appendices

Appendix A

SMILES Grammar

The full grammar of the SMILES[6] language. Notably, this project is only interested in a subset.

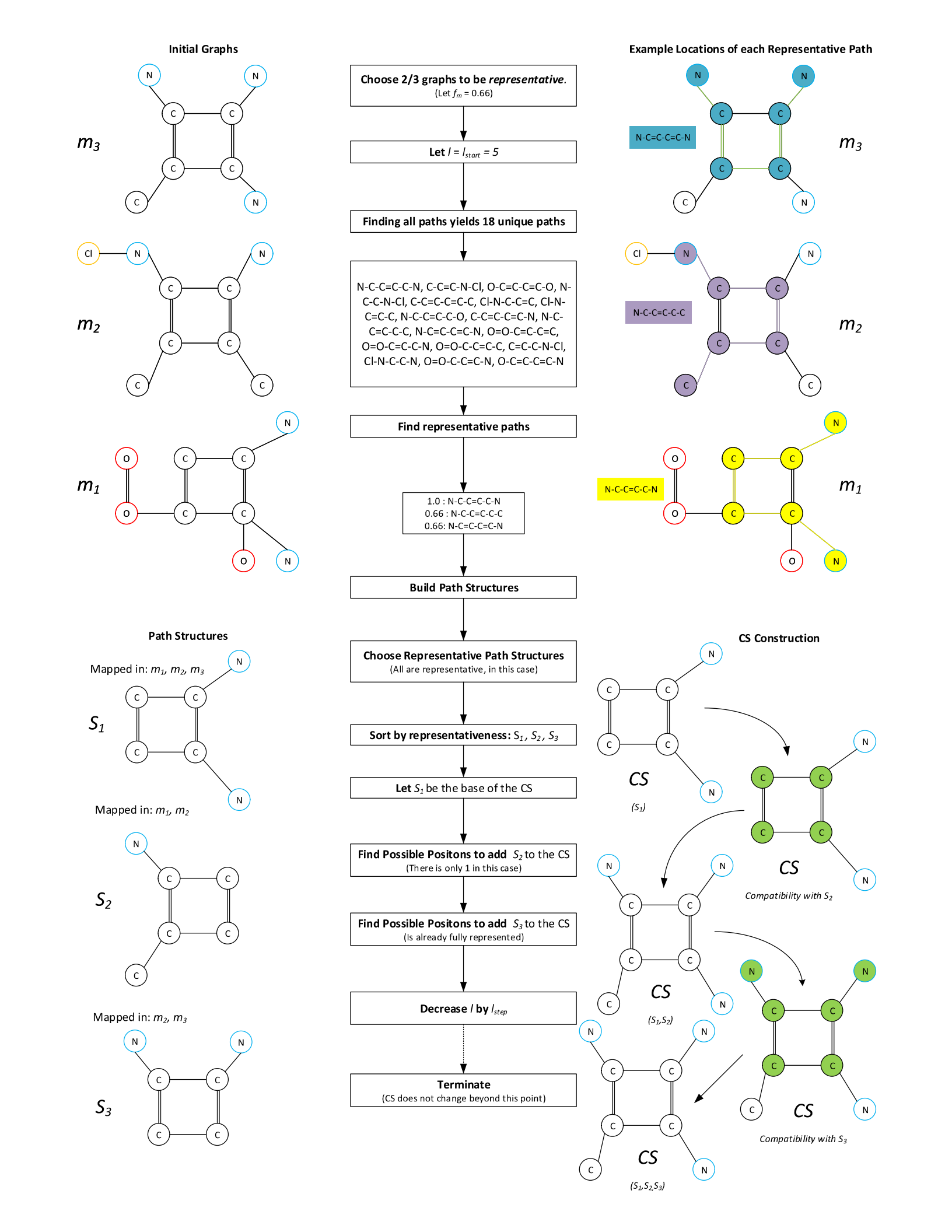
```
SMILES    ::= Atom ( Chain | Branch )*
Chain     ::= ( Bond? ( Atom | RingClosure ) )+
Branch    ::= '(' Bond? SMILES+ ')'
Atom      ::= OrganicSymbol
            | AromaticSymbol
            | AtomSpec
            | WILDCARD
Bond      ::= '-'
            | '='
            | '#'
            | '$'
            | ':'
            | '/'
            | '\'
            | '.'
AtomSpec ::= '[' Isotope? ( 'se' | 'as' |
 AromaticSymbol | ElementSymbol |
  WILDCARD ) ChiralClass? HCount? Charge? Class? ']'
OrganicSymbol
          ::= 'B' 'r'?
            | 'C' 'l'?
            | 'N'
            | 'O'
            | 'P'
            | 'S'
            | 'F'
            | 'I'
AromaticSymbol
          ::= 'b'
            | 'c'
            | 'n'
            | 'o'
            | 'p'
            | 's'
```

```
WILDCARD ::= '*'



ElementSymbol
         ::= [A-Z] [a-z]?
RingClosure
         ::= '%' [1-9] [0-9]
           | [0-9]
ChiralClass
         ::= ( '@' ( '@' | 'TH' [1-2] | 'AL' [1-2] |
          'SP' [1-3] | 'TB' ( '1' [0-9]? | '2' '0'? |
           [3-9] ) | 'OH' ( '1' [0-9]? | '2' [0-9]? |
            '3' '0'? | [4-9] ) )? )?
Charge   ::= '-' ( '-' | '0' | '1' [0-5]? | [2-9] )?
           | '+' ( '+' | '0' | '1' [0-5]? | [2-9] )?
HCount   ::= 'H' [0-9]?
Class    ::= ':' [0-9]+
Isotope  ::= [1-9] [0-9]? [0-9]?
```

Appendix B

CS Algorithm - Visual Walkthrough

This section has a visual example of the main steps in the characteristic substructure algorithm.


Initial Graphs
m_3
m_2
m_1
Choose 2/3 graphs to be *representative*.
(Let f_m = 0.66)
Let l = l_{start} = 5
Finding all paths yields 18 unique paths
N-C-C=C-C-N, C-C=C-N-Cl, O-C=C-C=C-O, N-C-C-N-Cl, C-C=C-C=C-C, Cl-N-C-C=C, Cl-N-C=C-C, N-C-C=C-C-O, C-C=C-C=C-N, N-C-C=C-C-C, N-C=C-C=C-N, O=O-C=C-C=C, O=O-C=C-C-N, O=O-C-C=C-C, C=C-C-N-Cl, Cl-N-C-C-N, O=O-C-C=C-N, O-C=C-C=C-N
Find representative paths
1.0 : N-C-C=C-C-N
0.66 : N-C-C=C-C-C
0.66: N-C=C-C=C-N
Build Path Structures
Choose Representative Path Structures
(All are representative, in this case)
Sort by representativeness: S_1, S_2, S_3
Let S_1 be the base of the CS
Find Possible Positons to add S_2 to the CS
(There is only 1 in this case)
Find Possible Positons to add S_3 to the CS
(Is already fully represented)
Decrease l by l_{step}
Terminate
(CS does not change beyond this point)
Example Locations of each Representative Path
N-C=C-C=C-N
N-C-C=C-C-C
N-C-C=C-C-N
Path Structures
Mapped in: m_1, m_2, m_3
S_1
Mapped in: m_1, m_2
S_2
Mapped in: m_2, m_3
S_3
CS Construction
CS
(S_1)
Compatibility with S_2
(S_1,S_2)
Compatibility with S_3
(S_1,S_2,S_3)

Appendix C

Requirements Documents

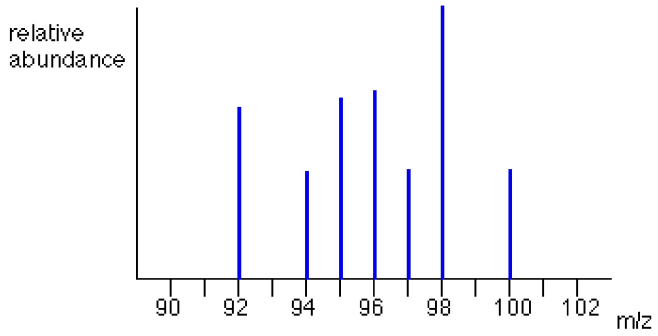
The requirements were collected during a series of meeting between the author, Dr. Alice Miller and Dr. Simon Rogers. There were 2 major iterations of requirements gathering, followed by a cycle of analysis/feedback to ascertain/disambiguate parts. Here are the documents that were made for these steps:

Metabolite Substructures Project – Problem Statement

This CS project focuses on designing and implementing one or more algorithms to analyse data obtained from samples analysed via Mass Spectrometry (MS). Specifically, these samples are intended to be metabolites and the analyses should aid in determining possible and probable molecules found within a sample.

Background

MS is a measuring process by which one obtains the mass-to-charge ratios of the chemical components within a given sample. There are a variety of ways this is conducted, where most involve steps of sample separation, charging, accelerating and then detection. At the end of the process one arrives at a fragmentation pattern like the following:



Example Fragmentation Pattern

Each peak corresponds to the frequency of a mass-to-charge ratio indicating the presence of molecule with these properties. Notably, there are multiple peaks as there can be different variations (e.g isotopes) of the original molecule and because the sample is broken down both inevitably and intentionally during the measuring process.

A metabolite is a small molecule found in cells and organisms and is the product of a metabolism. It has uses in analysing cellular behaviour and accordingly plays a role in observing the behaviour of organisms such as diseases.

Proposed Algorithms

Finding Common subgroups

1. *Simple version*
    a. We are given a group of molecules with the assumption that they have common molecule (could be a fragment). Determine what that common molecule could be.
2. *Detailed version*
    a. We are given a method that groups candidate molecules into a group $G = \{m_1, m_2, \dots, m_n\}$ (m is a molecule) with the assumption that there is a common

subgroup[1] S. We assume that S must be a subgraph and use a subgraph isomorphism algorithm between all members of G to determine what the likeliest value of S is. In order to determine the likelihood, consider the # of vertices in S vs. the # of members in G that the subgraph isomorphism with S satisfies.

3. *Questions*
    a. Isn't finding an isomorphic subgraph likely to yield a trivial result? Mostly as slight differences in bonds (e.g with isomers) will be discounted even though they are chemically almost identical?
    b. What is this mystery method that groups molecules? Do we actually assume that the subgroup is a physical substructure or just hazy thing they have in common?

Matching Fragmentation Patterns & Molecules

1. *Simple version*
    a. We are given a molecule and would like to see how well it matches with a given fragmentation pattern. Achieve this by breaking it into fragments and seeing how well they match with the peaks in the pattern.
2. *Detailed version*
    a. Given a molecule M and a fragmentation pattern $F = \{p_1, p_2, \dots, p_n\}$, break M into a set of smaller molecules $S = \{s_1, s_2, \dots, s_m\}$ and take their mass-to-charge ratios. Then, determine for how many $s_i \in S, \exists p_j \in F$ such that $mz(s_i) = p_j$ within an acceptable margin of error (where $mz()$ gets the mass-to-charge ratio).
3. *Questions*
    a. Is the main challenge here to break the molecule in such a way, that it is likely to satisfy the highest possible fraction of the fragmentation spectra?
        i. Should this use heuristics, and/or should the search be exhaustive?
        ii. Is it assumed that $m = n$ or can certain spectra/remainder atoms/charges be ignored?
    b. By what rules can molecules be broken down?
        i. Are there probabilities of certain bonds breaking over others?
    c. What is an acceptable margin of error when comparing the mass-to-charge ratio of a molecule fragment and a fragmentation peak?

[1] This is a single molecule fragment that group could have in common.

Fragmentation Pattern

1. Given a list *l* and its fragmentation pattern (a list of tuples with a m/z and a relative Hz) *f*, find a characteristic substructure *c* of *l* and then compare it to *f*, finding the best-matching fragment (another molecule).
    a. This can be done via a tool called *CFM-ID*, which will take a molecule and fragmentation pattern and return a fragmentation tree and scores. From those, one selects the best-scoring fragment.
2. While developing the substructure *c*, for each path structure considered viable, assess how well it matches *f*? Consider additional opportunities to compare the fragmentation patterns.
    a. Will be treated as optional until the previous part is completed.

List Matching

Given sets of lists s_1, s_2 and list l, find 2 characteristic substructures with the following:

- For characteristic substructure c_1, it chooses paths/structures from s_1 and a single molecule $m \in l$.
- For c_2, it chooses paths/structures from s_2 and the same $m \in l$.

The interest here should lie in the choice of m.

I will verify whether this is what we want before designing an algorithm as this may not be simple or efficient to compute, especially when selecting the right m.

Appendix D

Incorrect Finding of Paths Example

The below is an example of how the path finding only yields a subset of all results, by forgetting to mark its vertices as unvisited when backtracking.

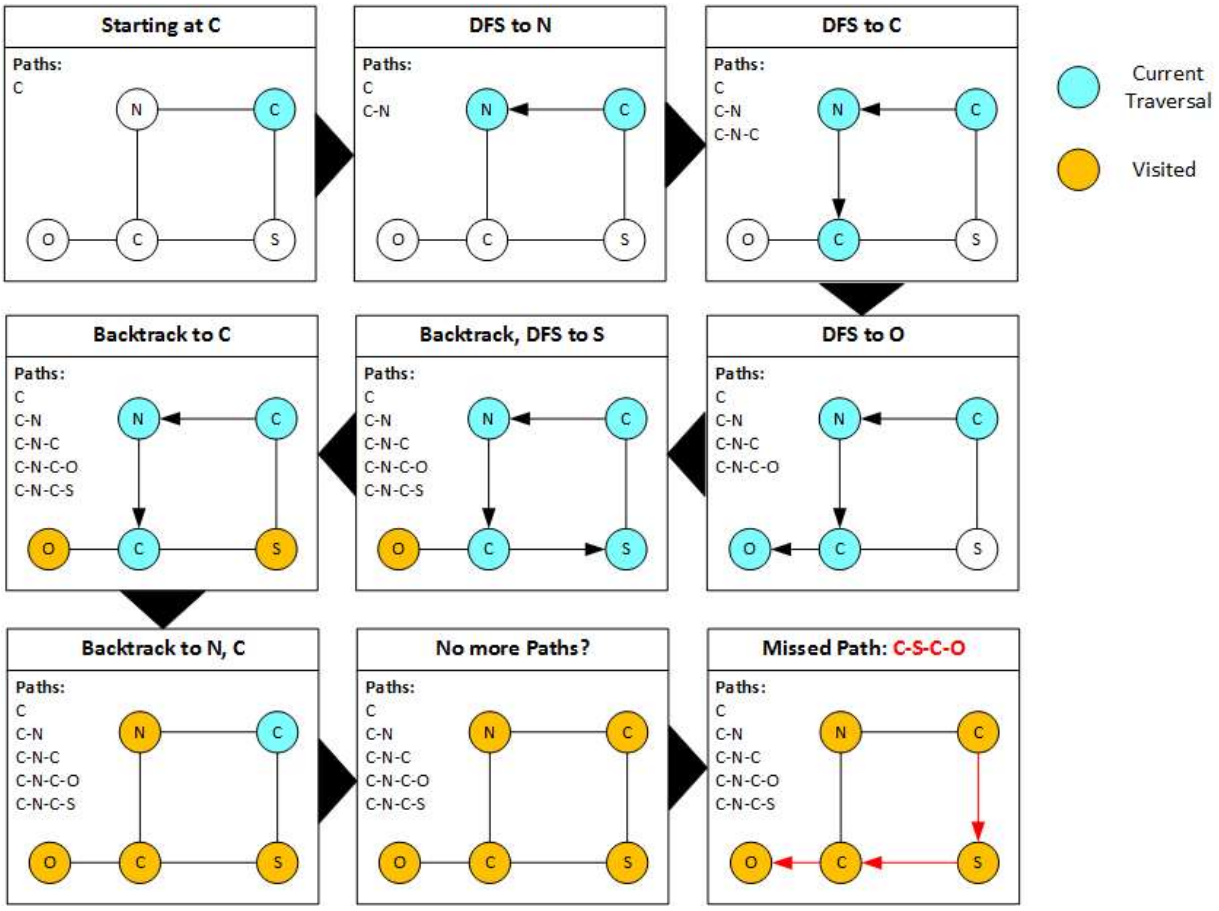


Figure D.1: Pathfinding Subset Example

Appendix E

Configuring CFM

This project uses CFM-ID v2.1 and has written a wrapper around the `cfm-annotate.exe`. The application requires the additional files `lpsolve.dll` and `ISOTOPE.DAT` to be in the same folder as the application. The former is a publicly available DLL and the latter should arrive bundled with the application itself. Further, the application also needs a `param_config.txt` which specifies the bond breaking behaviour. If not provided, it will look for this file in the running directory. Additionally, the training data for the application is typically in a `param_outputX.log` file, where "X" is some number (in our case, it is `param_output0.log`). One other specific file the application needs is the peak data (`spectrum_file`) that needs to have a layout like this example:

```
low
81.0333699384 121220.59375
77.03881151 133290.625
51.0234085408 165148.796875
95.0493939466 281643.1875
53.0388854061 1210836
93.0698532916 1619275.25
91.0541537703 1633952.75
103.0545234711 1977487
121.0649668164 10389399

medium
81.0333699384 121220.59375
77.03881151 133290.625
51.0234085408 165148.796875
95.0493939466 281643.1875
53.0388854061 1210836
93.0698532916 1619275.25
91.0541537703 1633952.75
103.0545234711 1977487
121.0649668164 10389399

high
81.0333699384 121220.59375
77.03881151 133290.625
51.0234085408 165148.796875
95.0493939466 281643.1875
```

```
53.0388854061 1210836
93.0698532916 1619275.25
91.0541537703 1633952.75
103.0545234711 1977487
121.0649668164 10389399
```

Where the 1st column is the mass data and the 2nd one is the intensity. Note that the application insists on the "low, medium, high" labelling, but will return these as "energy0, energy1, energy2" in its outputs. More details on the configuration can be seen under[16].

Appendix F

CFM Example Outputs

The following is an example output of the `cfm-annotate.exe` application when testing the `ethyl_166.122631425_` peak data with the SMILES string C(C(=CC=1)O)=C(C=1)CC(O)=O.

```
TARGET ID: CS
energy0
51.02340854 0.9419713197 17 (0.94197)
53.03888541 6.906334206
77.03881151 0.7602595254
81.03336994 0.6914148019 18 5 (0.40157 0.28985)
91.05415377 9.319696282 16 (9.3197)
93.06985329 9.235979148
95.04939395 1.60642893
103.0545235 11.27913781
121.0649668 59.25877798
energy1
51.02340854 0.9419713197 17 (0.94197)
53.03888541 6.906334206
77.03881151 0.7602595254
81.03336994 0.6914148019 18 5 (0.40155 0.28986)
91.05415377 9.319696282 16 (9.3197)
93.06985329 9.235979148
95.04939395 1.60642893
103.0545235 11.27913781
121.0649668 59.25877798
energy2
51.02340854 0.9419713197 17 (0.94197)
53.03888541 6.906334206
77.03881151 0.7602595254
81.03336994 0.6914148019 18 5 (0.40155 0.28987)
91.05415377 9.319696282 16 (9.3197)
93.06985329 9.235979148
95.04939395 1.60642893
103.0545235 11.27913781
121.0649668 59.25877798

19
```

```
0 153.0546206 O=C(O)Cc1cccc([OH2+])c1
1 135.0440559 OC(O)=[C+]C1=CC=CC=C1
2 107.0491413 [CH2+]C1=CC=CC(O)=C1
3 109.0647913 CC1=CC=CC([OH2+])=C1
4 135.0440559 OC#CC1=CC=CC([OH2+])=C1
5 81.0334912 [CH+]=C(O)C#CC
6 69.0334912 C#CC(=C)[OH2+]
7 95.01275576 C#CC#CC(O)=[OH+]
8 111.0440559 C#CC#CCC(O)[OH2+]
9 125.0233204 [CH2+]C(O)=C=C=C=C(O)O
10 127.0389705 CC([OH2+])=C=C=C=C(O)O
11 69.0334912 C=CC#C[OH2+]
12 127.0389705 C=C(C#C[OH2+])C=C(O)O
13 125.0233204 [CH+]=C(C#CO)C=C(O)O
14 99.04405588 [CH+]=C(C)C=C(O)O
15 111.0440559 C#CC(=C)C=C(O)[OH2+]
16 91.05422664 [CH2+]C1=CC=CC=C1
17 51.02292652 C#CC#[CH2+]
18 81.0334912 [CH+]=C(C)C#CO

0 1 O
0 2 O=CO
0 3 O=C=O
0 4 O
0 5 C#CC(O)O
0 6 C#CC=C(O)O
0 7 C=C(C)O
0 8 C#CO
0 9 C=C
0 10 C#C
0 11 C#CC=C(O)O
0 12 C#C
0 13 C=C
0 14 C=C=C=O
0 15 C#CO
1 16 O=C=O
1 17 C#CC=C(O)O
2 5 C#C
2 18 C#C
3 16 O
3 5 C=C
3 17 C=C(C)O
3 18 C=C
4 5 C#CC=O
4 18 C=C=C=O
6 17 O
7 17 O=C=O
8 17 C=C(O)O
9 5 O=C=O
10 5 O=CO
11 17 O
```

```
12 18 O=CO
13 18 O=C=O
14 18 O
15 17 C=C(O)O
```

The first section above shows the annotated peaks, having up to 2 suggested matches (their IDs) with their measures of confidence. These IDs then map to a series of molecule fragments with their mass/charge ratios and SMILES strings. The last data collection then details the state transitions in the algorithm itself to explain the peak annotation process.

Appendix G

Program Command Line

The program has to major modes, as is given below:

```
usage: main.py [-h] {cs,rm} ...

This application finds characteristic substructures, matches them to
fragmentation spectra,and finding a representative molecule from multiple
lists.

positional arguments:
  {cs,rm}
    cs        Find characteristic substructure and optionally use
              fragmentation comparison.
    rm        Find a the best-matching molecule from a list, when building 2
              CS.
```

Choosing “cs” gives the following options:

```
usage: main.py cs [-h] [-r] [-t THRESHOLD] [-lt THRESHOLD] [-iso THRESHOLD]
                  [-f FILE_NAME] [-nr] [-ls LENGTH] [-le LENGTH] [-s LENGTH]
                  [-cfm SCORE] [-img] [-o OUTPUT_FOLDER_NAME] [-v]
                  smiles_files [smiles_files ...]

positional arguments:
  smiles_files          1 or more files containing SMILES strings.

optional arguments:
  -h, --help            show this help message and exit
  -r, --representative_struct
                        Output all of the representative path structures as
                        well.
  -t THRESHOLD, --threshold THRESHOLD
                        the relative frequency of structures in the molecules.
                        Choose percentage from 0..1. Default is 0.8.
  -lt THRESHOLD, --list_threshold THRESHOLD
```

```
                        relative frequency of structures among a set of lists.
                        Choose percentage from 0..1. Default is 0.8.
  -iso THRESHOLD, --isomorphism_factor THRESHOLD
                        sets how many path structures need to be present for
                        adding multiple structures. Choose percentage from
                        0..1. Default is 0.8.
  -f FILE_NAME, --fragmentation_pattern FILE_NAME
                        Add a file for a fragmentation pattern of the format
                        specified by the README. Will be tested against the
                        deduced characteristic substructure anda peak graph +
                        fragments will be returned.
  -nr, --not_random     Ensures tie-breaking is consistent by using a
                        secondary ordering of arbitrary IDs, dependent on
                        molecule generation order.
  -ls LENGTH, --length_start LENGTH
                        Specify starting path length of CS algorithm.
  -le LENGTH, --length_end LENGTH
                        Specify finishing path length of CS algorithm.
  -s LENGTH, --step LENGTH
                        Specify the stepping length for the algorithm, once it
                        has found a CS.
  -cfm SCORE, --cfm_min_score SCORE
                        Specify a minimum score to highlight matched
                        fragments. Default is 10.
  -img, --image_of_CS   Select if an image of the characteristics substructure
                        should be outputted.
  -o OUTPUT_FOLDER_NAME, --output_name OUTPUT_FOLDER_NAME
                        Add a folder name to use for the output folder. Will
                        otherwise default to first SMILES file name.
  -v, --verbose         prints additional info & statistics while processing.
```

Similarly, choosing “rm” gives the following options:

```
usage: main.py rm [-h] [-t THRESHOLD] [-lt THRESHOLD] [-nr] [-ls LENGTH]
                  [-le LENGTH] [-s LENGTH] [-o OUTPUT_FOLDER_NAME] [-v] [-img]
                  smiles_files [smiles_files ...] LIST_FILE_NAME

positional arguments:
  smiles_files          2 or more files containing SMILES strings. Number of
                        files has to be even.
  LIST_FILE_NAME        File containing list of SMILES strings to try adding
                        to the 2 CS as individual molecules.

optional arguments:
  -h, --help            show this help message and exit
  -t THRESHOLD, --threshold THRESHOLD
                        the relative frequency of structures in the molecules.
                        Choose percentage from 0..1. Default is 0.8.
  -lt THRESHOLD, --list_threshold THRESHOLD
                        relative frequency of structures among a set of lists.
                        Choose percentage from 0..1. Default is 0.8.
```

```
-nr, --not_random     Ensures tie-breaking is consistent by using a
                      secondary ordering of arbitrary IDs, dependent on
                      molecule generation order.
-ls LENGTH, --length_start LENGTH
                      Specify starting path length of CS algorithm.
-le LENGTH, --length_end LENGTH
                      Specify finishing path length of CS algorithm.
-s LENGTH, --step LENGTH
                      Specify the stepping length for the algorithm, once it
                      has found a CS.
-o OUTPUT_FOLDER_NAME, --output_name OUTPUT_FOLDER_NAME
                      Add a folder name to use for the output folder. Will
                      otherwise default to first SMILES file name.
-v, --verbose         prints additional info & statistics while processing.
-img, --images_of_molecules
                      Select if an image of the chosen molecule, CS1 and CS2
                      should be outputted.
```

Appendix H

Stakeholder Test Data Report

Metabolite Substructures Project – Data Report

Input Data

The input data is a collection of files that take the following format:

General Format	Example of an ethyl measurement
File name: <compound name>_<parent peak mass>_<parent peak retention time>	ethyl_166.122631425_696.858.txt
<child_peak data> $mass_1$,$intensity_1$ $mass_2$,$intensity_2$ $mass_3$,$intensity_3$ ... <chemical compound info> $name_1$,$compound_1$ $name_2$,$compound_2$ $name_3$,$compound_3$ ...	81.0333699384,121220.59375 77.03881151,133290.625 51.0234085408,165148.796875 95.0493939466,281643.1875 53.0388854061,1210836 93.0698532916,1619275.25 91.0541537703,1633952.75 103.0545234711,1977487 121.0649668164,10389399 Hit 0:Phenylephrine,C9H13NO2 Hit 1:3-Ethoxybenzoate,C9H10O3 Hit 2:4-Hydroxymandelate,C8H8O4 Hit 3:3-Hydroxymandelic acid,C8H8O4 Hit 4:L-Phenylalanine,C9H11NO2 Hit 5:L-Phenylalanine,C9H11NO2 Hit 6:Tropate,C9H10O3 Hit 7:Terephthalate,C8H6O4 Hit 8:2-(4-Hydroxyphenyl)propionate,C9H10O3 Hit 9:Homogentisate,C8H8O4

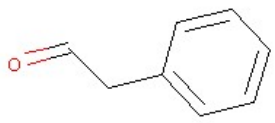
The file name describes the main peak of the MS measurement, defining its mass and retention time. In the file, the first section contains the fragmentation pattern followed by a list of possible molecule candidates.

For this specific data, it is primarily of interest to see individual characteristic substructures for Ethyl and Ferulic individually as well as together with a 3rd list of feruoyltyramine.

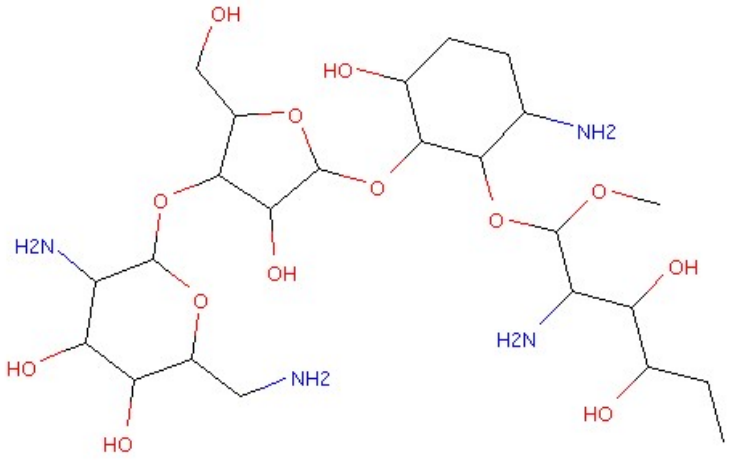
Characteristic Substructures

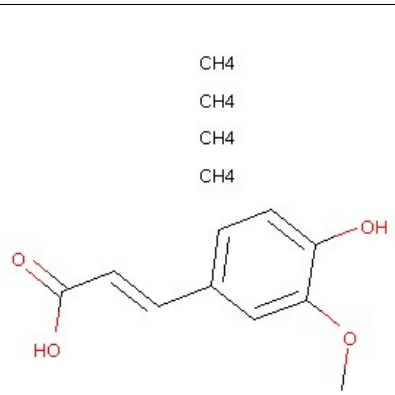
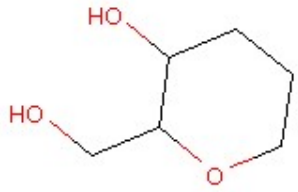
Ethyl

Compound	Molecule

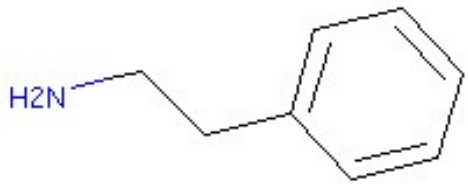
mass=166.122631425 retention=696.858	
SMILES: C(=C(CC=O)C=C1)C=C1 Molecule_threshold=0.5	

Ferulic

Compound	**Molecule**
mass=307.1764492284 retention=565.722	
SMILES: O(C(C(C1OC(C(OC(C(C(C(CC)O)O)N)OC)C(C2)N)C(C2)O)O)C(CO)O1)C(C(C(C1O)O)N)OC1CN.C.C.C.C.C.C.C.C.C.C.C.C.C.C.C.C.C.C.C.C.C (cropped trailing C atoms) Molecule_threshold=0.5	

mass=194.0811588123 retention=372.653	
SMILES: O=C(O)C=CC(C=CC=1O)=CC=1OC.C.C.C.C Molecule_threshold=0.3	
All	
SMILES: C(C(C(O)CC1)OC1)O Molecule_threshold=0.2 List_threshold=1	

Ferulic & Ethyl

Molecule

SMILES: C(CC(C=CC=1)=CC=1)N Molecule_threshold=0.2 List_threshold = 0.6 (at least 2 out of 3)

Ferulic, Ethyl & Feruoyltyramine

None. Regardless of list or molecule threshold chosen, the algorithm does not yield a substructure. I have determined why, and it relates to an ambiguity in the original paper in

regard of adding a path structure to the characteristic structure multiple times. There are some ways around this I would like to discuss.

Notes

- Thresholds range from 0-1 (0%-100% coverage).
- Only used molecules & fragmentation patterns that had hits from the mass bank.
- Used name of molecules as formulas tended yield many results. Pubchem did not find results for: “Prowl(TM)”, “Na-Benzenolarginine”. A pubchem ID or other identifier might make this easier.

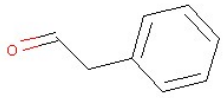
Molecule Fragments

Using the CFM-ID peak assignment software and the following assumptions:

- Ion mode: positive
- Medium energy spectra
- Mass tolerance of 10 ppm
- A variety of internal configurations influencing bond breaking rules, search depth, etc.

Here are the best matching fragments for each substructure & fragmentation pattern.

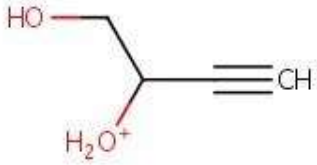
Ethyl

Peak Data & mass of fragment	**Molecule & SMILES**	**Confidence score**
121.064968 59.2587778 121.065339	 O=CCC1=CC=CC=C1	59.259

Ferulic #1

72.08081554 2.722792654 72.08132432	 CCC=C[NH3+]	2.4008

Ferulic #2

87.04392403 11.73790802 87.04460446	 C#CC([OH2+])CO	6.6488

Notes

- Unclear if it is suitable for LCMS.
- Results depend heavily on configuration of search. Typically would also require training a model.
- Not sure which spectra would be applicable to characteristic structures made from multiple lists.

Appendix I

Example Program Outputs

I.1 CS & Fragmenation Spectrum of Ethyl

The following is the `verbose` output of taking the CS of the stakeholder data: `ethyl_166.122631425_696.858`

```
Verbose mode activated.
Set threshold to: 0.5
Set not_random to: True
Read in 1 list(s) with a total of 10 SMILES strings.
Started path search in molecules.
Found 188 unique paths.
Path distribution: [0, 3, 5, 9, 17, 22, 30, 33, 35, 23, 11]
Finished path search in 14ms.
Starting CS algorithm at length: 10.
CS search length: 10.
CS search length: 9.
Found 4 rep. paths in 0ms.
Found 2 rep. structures in 20ms.
Rep. tie-breaks: 1
Added path struct C-C=C-C=C-C-C-C-O to CS.
Added path struct C=C-C=C-C=C-C-C=O to CS.
CS search length: 5.
Found 9 rep. paths in 0ms.
Found 9 rep. structures in 105ms.
Rep. tie-breaks: 5
Added path struct C-C=C-C=C to CS.
Added path struct C=C-C-C-O to CS.
Added path struct C-C-C-C=C to CS.
Added path struct C=C-C-C=O to CS.
Added path struct C-C-C-C-O to CS.
Added path struct C-C=C-C-O to CS.
Added path struct C-C-C-C=O to CS.
Added path struct C=C-C=C-O to CS.
Characteristic Substructure:
C(C=CC=1OC=CC=C)=C(C=1)CC(O)=O
TARGET ID: CS
```

```
energy0
51.02340854 0.9419713197 6 (0.94197)
53.03888541 6.906334206 5 (6.9063)
77.03881151 0.7602595254
81.03336994 0.6914148019 56 54 47 37 51 50 58 (0.25235 0.20041 0.19551 0.031235 0.0
91.05415377 9.319696282 44 (9.3197)
93.06985329 9.235979148 45 (9.236)
95.04939395 1.60642893 48 29 53 55 57 52 (0.67854 0.63898 0.14302 0.14274 0.0019242
103.0545235 11.27913781
121.0649668 59.25877798 46 23 36 49 (31.725 17.146 8.1511 2.2366)
energy1
51.02340854 0.9419713197 6 (0.94197)
53.03888541 6.906334206 5 (6.9063)
77.03881151 0.7602595254
81.03336994 0.6914148019 37 56 54 47 51 50 58 (0.20785 0.18404 0.14616 0.14258 0.00
91.05415377 9.319696282 44 (9.3197)
93.06985329 9.235979148 45 (9.236)
95.04939395 1.60642893 48 29 53 55 57 52 (0.8809 0.34632 0.18569 0.1853 0.0066262 0
103.0545235 11.27913781
121.0649668 59.25877798 46 23 36 49 (34.294 14.326 8.2214 2.4178)
energy2
51.02340854 0.9419713197 6 (0.94197)
53.03888541 6.906334206 5 (6.9063)
77.03881151 0.7602595254
81.03336994 0.6914148019 56 37 54 47 58 51 50 (0.19901 0.16331 0.15806 0.15418 0.00
91.05415377 9.319696282 44 (9.3197)
93.06985329 9.235979148 45 (9.236)
95.04939395 1.60642893 48 53 55 29 57 52 (1.04 0.21925 0.21878 0.11636 0.010124 0.0
103.0545235 11.27913781
121.0649668 59.25877798 46 23 49 36 (49.261 5.0956 3.4731 1.4287)

61
0 205.0859207 C=CC=C[OH+]c1cccc(CC(=O)O)c1
1 69.0334912 CC=C=C=[OH+]
2 135.0440559 OC(O)=[C+]C1=CC=CC=C1
3 137.0597059 OC([OH2+])=CC1=CC=CC=C1
4 139.075356 OC([OH2+])CC1=CC=CC=C1
5 53.03857658 C=CC#[CH2+]
6 51.02292652 C#CC#[CH2+]
7 151.0389705 OC(O)=[C+]C1=CC(O)=CC=C1
8 153.0546206 OC(O)=CC1=CC([OH2+])=CC=C1
9 155.0702706 OC(O)CC1=CC([OH2+])=CC=C1
10 165.0546206 C=[O+]C1=CC=CC(C=C(O)O)=C1
11 179.0702706 C#C[OH+]C1=CC=CC(CC(O)O)=C1
12 189.0546206 [CH2+]C#COC1=CC=CC(C=C(O)O)=C1
13 145.0647913 C#CC#C[OH+]C1=CC=CCC1
14 157.0647913 C#CC#C[OH+]C1=CC=CC(C)=C1
15 159.0804414 C#CC#C[OH+]C1=CC=CC(C)C1
16 161.0960915 C=CC#C[OH+]C1=CC=CC(C)C1
17 187.075356 C#CC#C[OH+]C1=CC=CC(CCO)=C1
18 131.0491413 C#CC#C[OH+]C(C)=CC#C
```

```
19 133.0647913 C#CC=C(C)[OH+]C#CC=C
20 135.0804414 C#CC=C(C)[OH+]C#CCC
21 117.0334912 C#CC#C[OH+]C#CC=C
22 119.0491413 C#CC#C[OH+]C#CCC
23 121.0647913 C#CC#C[OH+]C=CCC
24 165.0546206 C#CC#C[OH+]CC#CCC(O)O
25 163.0389705 C#CC#C[OH+]CC#CC=C(O)O
26 179.0702706 C#CC#C[OH+]C(=C)C=CCC(O)O
27 189.0546206 C#CC#C[OH+]C#CC#CCCC(O)O
28 97.06479133 C=CC#C[OH+]CC
29 95.04914126 C#CC#C[OH+]CC
30 113.0597059 C#CC=CCC(O)[OH2+]
31 109.0647913 C=CC#C[OH+]C(=C)C
32 107.0491413 C#CC#C[OH+]C(=C)C
33 95.01275576 C#CC#CC(O)=[OH+]
34 97.02840582 C=CC#CC(O)=[OH+]
35 99.04405588 CC=C=C=C(O)[OH2+]
36 121.0647913 C#CC(C)[OH+]C#CC=C
37 81.0334912 C#CC#C[OH+]C
38 113.0597059 C#CC(=C)CC(O)[OH2+]
39 111.0440559 C#CC(=C)C=C(O)[OH2+]
40 99.04405588 [CH+]=C(C)C=C(O)O
41 105.0334912 C#CC#C[OH+]C#CC
42 189.0546206 C#CC#C[OH+]CC=C(C#C)C=C(O)O
43 179.0702706 C#CC#C[OH+]CCC(=C)C=C(O)O
44 91.05422664 [CH2+]C1=CC=CC=C1
45 93.06987671 [CH4+]C1=CC=CC=C1
46 121.0647913 [OH2+]C=CC1=CC=CC=C1
47 81.0334912 [C+]#CC=C(C)O
48 95.04914126 [OH2+]C1=CC=CC=C1
49 121.0647913 C=[O+]C1=CC=CC(C)=C1
50 81.0334912 C#COC(=[CH+])C
51 81.0334912 C#C[OH+]C#CC
52 95.04914126 [CH+]=C(C)OC#CC
53 95.04914126 C=C=C=C=CC[OH2+]
54 81.0334912 C#CC#CC[OH2+]
55 95.04914126 C#CC(=C)C=C[OH2+]
56 81.0334912 [CH+]=C(C)C#CO
57 95.04914126 C#C[OH+]C(C)C#C
58 81.0334912 C#CC(=C)[O+]=C
59 93.0334912 C#CC#C[OH+]C=C
60 79.01784114 C#CC#C[O+]=C

0 1 OC(O)=CC1=CC=CC=C1
0 2 CCC=C=O
0 3 CC=C=C=O
0 4 C=C=C=C=O
0 5 OC(O)=CC1=CC(O)=CC=C1
0 6 OC1=CC=CC(CC(O)O)=C1
0 7 C#CCC
0 8 C#CC=C
```

0 9 C#CC#C
0 10 C#CC
0 11 C#C
0 12 C
0 13 C=C(O)O
0 14 OCO
0 15 O=CO
0 16 O=C=O
0 17 O
0 18 C=CC(O)O
0 19 C#CC(O)O
0 20 C#CC(=O)O
0 21 C=CCC(O)O
0 22 C#CCC(O)O
0 23 C#CC=C(O)O
0 24 C#CC
0 25 C=CC
0 26 C#C
0 27 C
0 28 CC#CC#CC(=O)O
0 29 CC#CC#CC(O)O
0 30 C#CC#COC=C
0 31 C#CC#CC(O)O
0 32 C#CC=CC(O)O
0 33 C=C(C)OC#CCC
0 34 C=CC#COC(=C)C
0 35 C#CC#COC(=C)C
0 36 C#CC=C(O)O
0 37 C=C(C#CC)C=C(O)O
0 38 C#COC#CC=C
0 39 C#COC#CCC
0 40 C#CCOC#CC=C
0 41 C=C(C)C=C(O)O
0 42 C
0 43 C#C
1 6 O
2 44 O=C=O
3 44 O=CO
3 45 O=C=O
4 44 OCO
4 45 O=CO
4 46 O
7 47 C#CC(=O)O
8 47 C#CC(O)O
9 48 C=C(O)O
9 47 C=CC(O)O
10 49 O=C=O
11 50 C#CC=CC(O)O
12 51 CC#CC#CC(=O)O
12 52 C#CC#CC(=O)O
13 6 OC1=CC=CC=C1

13 48 C#CC#C
13 29 C#CC#C
14 44 C=C=C=C=O
15 44 CC=C=C=O
15 45 C=C=C=C=O
15 6 CC1=CC(O)=CC=C1
15 29 C#CC#CC
15 5 C#CC#COC(=C)C
16 44 CCC=C=O
16 45 CC=C=C=O
16 5 CC1=CC(O)=CC=C1
16 6 CC1C=CC=C(O)C1
16 49 C#CC
16 23 C#CC
16 29 C=CC#CC
16 36 C#CC
17 46 C=C=C=C=O
17 5 O=C=CC1=CC(O)=CC=C1
17 6 OC=CC1=CC(O)=CC=C1
17 23 C#CC#CO
17 29 C=C=C=C=C=CO
17 53 C#CC#COC=C
17 54 C#CC#COC(=C)C
17 36 C#CC#CO
17 37 C=C(C#CC)C#CO
17 55 C#COC#CC=C
17 56 C#CCOC#CC=C
18 47 C#CC#C
19 47 C#CC=C
20 29 C#CC
20 5 C#CC=C(C)O
20 47 C#CCC
21 6 C=C=C=C=O
22 6 CC=C=C=O
23 29 C#C
23 37 C#CC
23 6 CCC=C=O
24 29 C#CC(=O)O
24 23 O=C=O
25 5 O=C=C=C=CC(=O)O
26 29 C#CC=C(O)O
27 29 C#CC#CC(=O)O
27 37 CC#CC#CC(=O)O
27 5 OC#CC#CC#CC(O)O
28 51 C
29 5 C=C=O
30 53 O
31 50 C=C
32 50 C#C
33 6 O=C=O
34 6 O=CO

```
35 54 O
36 6 CCC=C=O
36 57 C#C
38 55 O
39 6 C=C(O)O
40 56 O
41 5 [C-]#CC#[O+]
42 5 C#CC(=C=C=O)C=C(O)O
42 29 C#CC#CC(=O)O
43 5 C=C(C=C=O)C=C(O)O
43 29 C#CC=C(O)O
23 41 C
23 1 C#CC=C
23 5 CC=C=C=O
29 6 CC=O
36 29 C#C
36 5 CC=C=C=O
36 58 C#CC
36 41 C
37 5 [C-]#[O+]
23 59 C=C
23 60 C=CC
36 59 C=C
36 1 C#CC=C
37 6 C=O
59 6 C=C=O
60 6 [C-]#[O+]
41 6 C#CC=O

energy0
energy1
energy2
Total calculation time: 11.183s.
```

Additionally, here is the plot produced from the peak annotation:

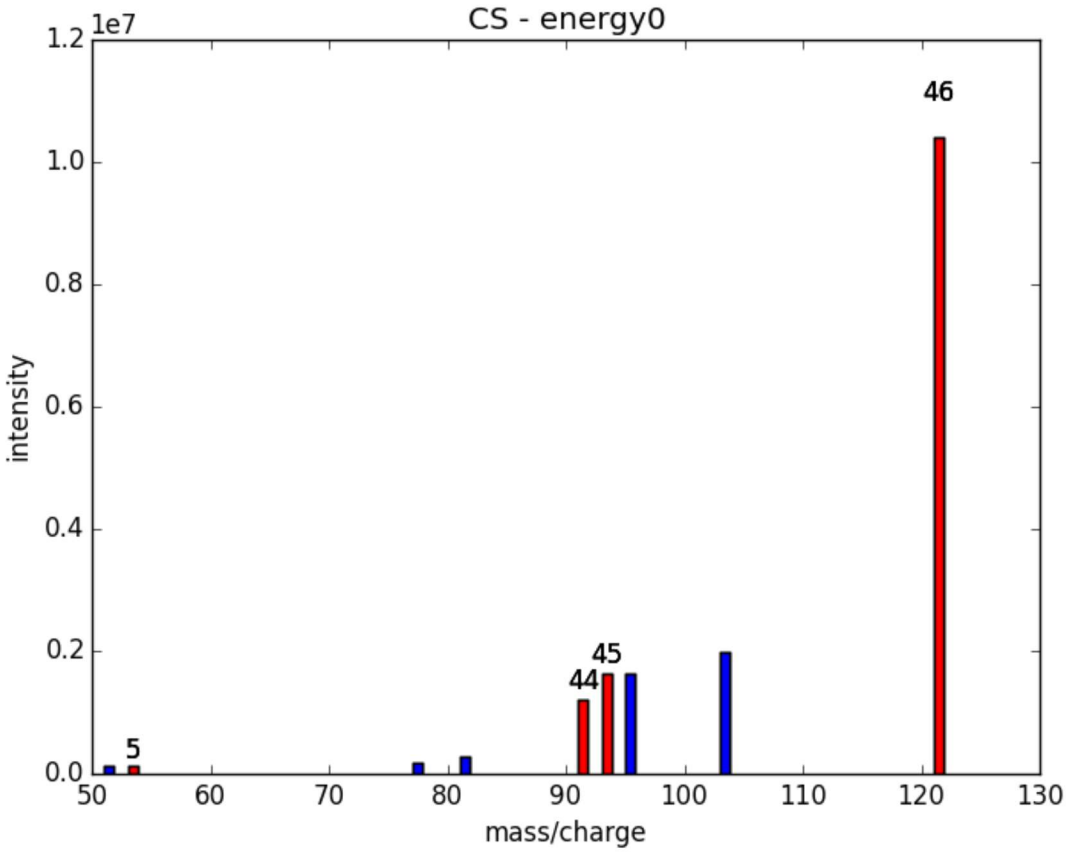


Figure I.1: Example spectrum produced from annotated peaks.