Implementations of knapsack algorithms for metabolite identification

Cristina Mihailescu

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

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

September 8, 2015

Abstract

Metabolite identification is an important problem in Bioinformatics. Given the mass of a molecule it needs to identify the number of various atoms that can compose it. This type of problem relates to the larger type of problems known as the Knapsack Problem. Therefore, three algorithms that can solve it were implemented: Exhaustive Search, Dynamic Programming Bellman's algorithm[3] and the algorithms used for the SIRIUS system[4] which have been introduced in [5].

The implementations have been evaluated against each other on in silico data and the results reflect that the SIRIUS-like knapsack algorithm produces better output at much faster speeds than more classical algorithms.

The system has several other functions such as generating test data from molecule formulas, running against a dataset read from file and filtering results. It can also be run from the console with intuitive commands that walk the user through entering the data and parameters needed.

Contents

Chapter 1

Introduction

Metabolites are small molecular end products found within biological samples, such as lactic acid and caffeine. They have various functions in an organism and their presence can provide valuable information of the processes within it. These processes are usually harder to asses in a direct fashion.

To identify a metabolite using some chemical techniques (discussed in more detail in section 2.2 Metabolite identification) the only data available is the mass of the molecule. A molecule is composed of several atoms that bind together. For n types of atoms with associated masses $a_i \in \{a_1...a_n\}$ and $c_i \in \{c_1..c_n\}$ th number of times each is encountered in the molecule , the mathematical formula for the mass is:

$a_1c_1 + a_2c_2 + .. + a_nc_n = M$

In Bioinformatics, masses are measured in daltons marked with the symbol Da. $1Da = 1u \approxeq 1g/mol$. where u is the symbol for the unified atomic mass unit, approximately equal to the mass of one nucleon (a proton or neutron).

The problem posed is of identifying the molecule, by decomposing the total mass M. The solution is the set c_i. This type of problem relates to the larger type of problems known as the Knapsack problem. This will be discussed in detail in section 2.1 Knapsack problem.

This project has been done in the context presented in chapter 2 Background.

The specific requirements are listed in chapter 3 Requirements and the design decisions derived from them in chapter 4 Design.

Chapter 5 Implementation goes into more detail regarding the various algorithm implementations with a discussion of the results in chapter 6 Results.

Future work, such as integration and personal reflection are found in chapter 7 Future work.

Finally, chapter 8 Conclusion brings the various discussion threads together to shed some light onto the problem introduced at the beginning of this chapter.

Chapter 2

Background

2.1 Knapsack problem

The Knapsack Problem is a well known problem of combinatorial optimization that often occurs when there are some resources to be shared, irrespective of the domain. The vocabulary used when discussing the problem is varied, with nomenclature preferred according to the specific context. I will attempt to present most. For this work I chose to follow the notations used in Pisinger [10].

The classical problem is usually exemplified through an analogy with a knapsack owning person. The stories vary, with the motivation being either a burglary or an upcoming hiking trip. Regardless of the owners motivation, the problem has the same staples:

- There is a finite resource: the backpack. Since we are referring either to the volume of the backpack or the total mass it can carry, the resource can also be called capacity and is commonly denoted by c.
- There are some items which can be chosen when filling the backpack. Each item has associated cost and profit.
- The cost is the relevant volume or mass that the item would consume from the resource if it is selected. It is sometimes referred to as weight and denoted by w_i for item i.
- The profit reflects the value it provides, which can sometimes be difficult to characterize. This is why the burglary scenario is often preferred as you can associate exact selling prices for the profit. It is denoted by p_i for item i.
- Choice for an object i is denoted by x_i. The values it can take depend on the scenario and will be discussed later in this section.

The owner of the backpack wishes to utilize it to its full potential. This is the same as saying that we need to maximise the profit sum without exceeding the capacity. Mathematically, using the notation described above, for n items, it can be written as in [10]:

$$\begin{aligned} &\text{maximise} \sum_{j=1}^{n} p_j x_j \\ &\text{subject to} \sum_{j=1}^{n} w_j x_j \leq c \\ &\text{where } x_j \geq 0 \end{aligned} \tag{2.1}$$

The decision problem for this classical Knapsack Problem is NP-complete, but the optimization problem is NP-hard. This means it is not possible to decide if a certain profit sum, within the capacity limit, can be achieved. Furthermore, there is no polynomial time algorithm that can achieve the optimal solution. As such, the only algorithm that can guarantee the optimal solution is through exhaustive search. Since the problem has been studied for more than a century and there is still active interest in it, there are several heuristics to either speed up the search or approximate correct solutions. These usually fall in one of these categories: branch-and-bound, dynamic programming, state space relaxation, preprocessing, hybridization of some of the former [10].

As mentioned before, there are different scenarios that lead to slightly different problems. A different setup can lead to more efficient algorithms. Some of the more popular setups are:

- *0-1 Knapsack Problem*

  There is only one item of each item type (described by weight and profit). Therefore, it can either be selected or not. This constraint is added to 2.1 as $x_j \in \{0, 1\}$

- *Bounded Knapsack Problem*

  There can be several items of each item type, with a bounded amount of each. Let m_j be the maximum number of items of type j for all $j \in \{0..n\}$. Then this constraint is added to 2.1 as $x_j \in \{0..m_j\}$.

- *Unbounded Knapsack Problem*

  There can be an infinite amount of items of each type. The only constraint added to 2.1 is $x_j \in \mathbb{Z}_{\geq 0}$.

- *Continuous Knapsack Problem*

  The items can be divided and fractions of items can be included in the knapsack. This problem can be either bounded or unbounded. For the unbounded, this can be written as $x_j \in \mathbb{Q}_{\geq 0}$. For the bounded case, another constrained needs to be added: $x_j \leq m_j$.

- *Multiple-choice Knapsack Problem*

  The items are organised in classes and the selection needs to be from disjoint classes. Equation 2.1 becomes rather cluttered, with an x_{ij} to stand for whether item j was selected from class i:

$$\begin{aligned} &\text{for } k \text{ classes } C_i, \text{ where } i \in \{1..k\} \\ &\text{maximise} \sum_{i=1}^{k} \sum_{j \in C_i} p_{ij} x_{ij} \\ &\text{subject to} \sum_{i=1}^{k} \sum_{j \in C_i} w_{ij} x_{ij} \leq c \\ &\text{where} \sum_{j \in C_i} x_{ij} = 1 \text{ for } i \in \{1..k\} \\ &\text{and } x_{ij} \in \{0,1\} \text{ for } i \in \{1..k\}, j \in C_i \end{aligned} \tag{2.2}$$

  The next to last line assures that there is only one item from each class.

- *Multiple Knapsack Problem*

  There are several knapsacks/independent resources that need to be filled simultaneously. They can have different capacities. If they have equal capacities and all items need to be packed, the problem is known as the Bin Packing Problem.

- *Multiple Constrained Knapsack Problem*

  This constraint, that could be identified by c in 2.1:

$$\sum_{j=1}^{n} w_j x_j \leq c \tag{2.3}$$

  is replaced by m constraints c_i with $i \in \{1..m\}$:

$$\sum_{j=1}^{n} w_{ij} x_j \leq c_i \text{ where } i \in \{1..m\} \tag{2.4}$$

- *Subset-sum Problem*

  The subset-sum problem has $p_j = w_j$ for all items j.

- *Change-making Problem*

  This is an unbounded setup, where the constraint c is set to equality:

$$\sum_{j=1}^{n} w_j x_j = c \tag{2.5}$$

  It can therefore be seen as a special case for the aforementioned *Unbounded Knapsack Problem*.

Some of these different formulations of seemingly the same problem can have much more efficient algorithms. For example, the aforementioned *Continuous Knapsack Problem* has a polynomial time algorithm. It is a greedy algorithm with preprocessing. The item types are considered in non-increasing order of their profit to weight ratio [7]. This is why it is very important to carefully consider each individual problem, rather than using a more general solving approach. Often, minor changes in the problem definition can lead to improved algorithms.

2.2 Metabolite identification

Bioinformatics has recently emerged as an interdisciplinary field. It aims to devise algorithms and tools to better manipulate and understand biological data. The data sets to be analysed are often very large and therefore require fast processing. Currently, the focus is mainly on genomics, despite growing research interest from the community in the fields often described as MS-omics: proteomics, lipidomics and metabolomics.

MS stands for Mass Spectrometry, a chemistry technique for analysing chemical data. The MS-omics pipeline consists of two phases: a wet-lab phase in which the sample is prepared and fed into a mass spectrometer and a data processing phase in which the data output is collected, processed and the desired solution generated. Usually, this involves the identification and/or qualification of a molecule. However, MS can only extract masses with associated intensity peaks, that correspond to the abundance of the molecule. The steps involved in this process are illustrated in fig. 2.1. Both figure and caption are taken exactly from [11].

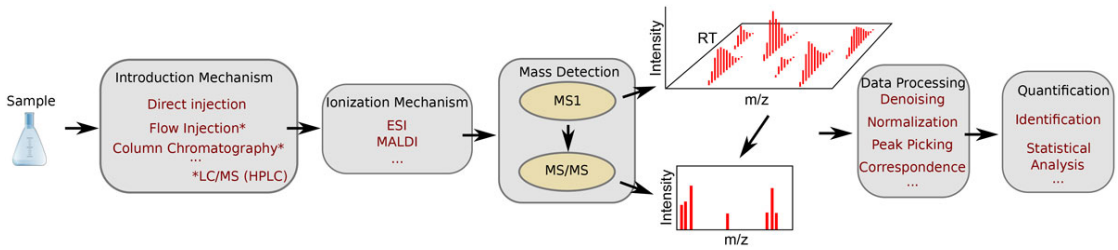

Figure 2.1: "The MS-omics pipeline. A sample is introduced to an ionization mechanism with or without a preliminary separation technique, where particles receive a charge enabling the detector to estimate the mass-to-charge ratio (m/z) and intensity of each analyte. If the system has tandem mass spec capabilities, some precursor ions (MS1) are selected for fragmentation (MS/MS). Data processing techniques prepare the data to be quantified via statistical methods and identified via matches to theoretical databases."

Each step needs to be performed correctly and as fast as possible since there are knock-on effects. While the instrumentation and protocols for MS processing have advanced considerably and continue to, the tools move at a much slower pace [6]. A paper that describes itself as

a tutorial in mass spectrometry for Computer Scientists [11] identifies the key issue impeding computational progress as the lack of cross-disciplinary experts. It is often difficult and prolonged to assimilate MS-omics coming from a background in Mathematics, Computer Science or Statistics. However, it can be even more difficult to acquire these technical skills from a biological context.

Another issue for advance in the field of MS-omics is the lack of labeled data [11]. This makes it difficult to assess the correctness of individual algorithms and to judge and compare optimizations. The available evaluation means consist of qualitative metrics and in silico simulated data. None of these approaches reflect properly the quality of a tool, but can often be a good indication or starting point.

As stated in the Introduction, this project is aimed more narrowly at metabolites. A metabolite is a small molecular end product found within a biological sample. Most metabolites are under 1000 Da, for one database 96.5 % being this small in 2006 [8, 4]. The metabolome - a complete set of metabolites for a sample - can provide an interesting snapshot of the cell from which it originates.

Glasgow Polyomics is a research facility at the University of Glasgow. As stated on their website [1], they apply "state-of-the-art technologies to measure the genome, transcriptome, proteome, and metabolome from any biological system". Their aim is to "seek ways to obtain new biological inference by combining multi-level 'polyomics' datasets."

For the metabolome analysis, after the MS phase over a sample, the system outputs masses (assumed correct). To identify the corresponding molecule, a database search for isotope patterns would be employed. However, for metabolites, there are many uncharacterised patterns. At the moment, this involves a manual switch to a different tool to look up the metabolite. This is not only time-consuming, but it also impedes a computational streamline of the entire pipeline.

The problem posed is therefore to identify candidate molecule formulas given a mass. Let the mass be M and the elements that can compose the molecule some of CHNOPS, with associated masses $a_1 ... a_6$. There are several other elements that can be found in a metabolite, but CHNOPS make up the vast majority. This now looks like a Knapsack Problem:

$$M - \epsilon \leq a_1 c_1 + a_2 c_2 + .. + a_6 c_6 \leq M + \epsilon \tag{2.6}$$

where ϵ is a very small tolerance measure to accommodate both equipment errors and computation roundings.

Not all possible formulas are potential molecules. There is a set of 7 golden rules [9] that help filter the set of candidates. These constitute the current state-of-the-art, but are not completely accurate. They indeed filter out most impossible formulas. However, they also eliminate some valid molecules. Since these molecules are extremely rare in nature, the compromise is considered worth while. A key observation is that some experiments might require the entire set of candidates with no filtering.

The 7 Golden rules are as follows [9]:

- 1. Restrictions for the number of elements

  This rule mainly concerns CHNOPS. For masses under 500, 1000, 2000 and 3000 there are maximum numbers each element can appear in a molecule. The default restriction for any element is naturally that the maximum number has to be less than or equal $mass_{total}/mass_{element}$.

- 2. Lewis and Senior chemical rules check

  This rule regards valences and has 3 conditions:

  - sum of all valences must be even
  - sum of all valences $\geq 2 * valence_{max}$
  - sum of all valences $\geq 2 * \#atoms - 1$

  The second condition can be hard to judge for all elements

- 3. Isotopic pattern filter

  Naturally synthesised compounds abide by the elemental isotope patterns. This can help filter when there are several masses for the same formula.

- 4. Hydrogen/Carbon element ratio check

  This rule states that for most molecules $6 > mass_H/mass_C > 0.1$

- 5. Heteroatom ratio check

  This rule bounds NOPS to carbon ratios. It is similar to rule 4, with appropriate values bounding the ratios.

- 6. Element probability check

  This rule bounds NOPS to carbon ratios. It is similar to rule 4, with appropriate values bounding the ratios.

- 7. TMS check

  This rule is only relevant for Gas Chromatography Mass Spectrometry (GC/MS). Certain TMS groups may have ionized the molecule during the chemical process and need to be subtracted to identify the molecule.

For Knapsack algorithms the first rule can prove especially useful. Turning an Unbounded problem into a Bounded one opens access to faster algorithms.

2.3 Previous work

An extremely relevant paper describes the SIRIUS implementation [4] for metabolite identification using both masses and isotope patterns. The SIRIUS implementation has not been evaluated on masses larger than 1000 Da. The algorithms for mass decomposition have been described in detail in [5]: *Round Robin* and *Find All*.

In 2013 a student at the University of Glasgow attempted to reproduce the results described in SIRIUS using those 2 algorithms mentioned earlier [2]. His implementation in Java worked fine for very small molecules (under 100 Da) and very slow for medium molecules (100 - 500 Da). Without using isotope patterns or the 7 Golden rules to filter the results, there were many false positive hits for each query. His implementation would however mention which formula has the nearest mass.

Chapter 3

Requirements

There are two sides to this project: exploratory and functional. Early on it became obvious that given the time limit only one can be successfully achieved. However, some of the requirements overlap.

The main requirement was to implement at least one knapsack algorithm that given a mass will return candidate molecules. For the exploratory side, several algorithms need to be implemented and evaluated against a benchmark. For the functional side, the algorithm(s) should be easy to interact with from the console, against a dataset or callable from a larger system.

The 7 Golden Rules needed to be implemented to filter obtained formulas (optionally) or to prune the search space and hopefully optimise and speed up the process.

Speed was also an important concern as very high precisions at near real time for relatively large batches of data is required.

To ease testing and make all experiments reproducible, there should be some functionality to generate in silico data and store it in files, read input from file and using one of the solving algorithms to produce output in other files.

Since the integration of the functionality comes more as an after thought it was deemed not essential at this time. Moreover, the pipeline in which this algorithm is to be integrated was still work in progress. However, the system should allow for future integration.

Chapter 4

Design

4.1 Language

The system at Glasgow Polyomics is written in R, with some Java classes, wrapped in Django/ Python. The system through which the algorithm implementation would interface with the larger system is written in Python, using Django. This would allow for easy calls to either python or R scripts, but would not exclude Java classes.

For similar Bioinformatics algorithms, the current trend is to use either Python or R. For number intensive tasks and/or statistics, the natural choice is R. This language is supposed to be more accurate with high precision numbers and faster with appropriate use of data structures. After succinct deliberations, a first implementation was attempted in R. This endeavour revealed that R is less intuitive for procedural algorithms and more suitable for mapping functions onto structured data. The data structures needed for this algorithms were less structured than needed to take advantage of R's speed ups. There was no 'natural' manner to organise the data into matrices or data frames. Despite the advertised accuracy, R often displayed inaccurate and unexpected rounding. Furthermore, the Polyomics group has stated that they are trying to move away from R as it is a rather abstruse medium. All these coupled with my personal unfamiliarity with the language put R at a serious disadvantage.

The other obvious choice was Python. None of the problems mentioned for R were maintained for Python. It would also be presumably easier to integrate a script into a Django system.

4.2 Selected algorithms

As presented in section 2.1 Knapsack problem there are several algorithms already devised to solve the problem more or less optimally.

When describing the algorithms I will use N to mean the number of decision points. A decision point is whether to include or exclude an item in the choice for the knapsack. Therefore N is bounded by the maximum number of items the knapsack can hold. For a maximum capacity c and elements $e_1..e_k$, N is bounded by $c/min(mass_{e_1}, ..., mass_{e_k})$. Since for the smallest mass is the mass of Hydrogen (approx 1.00782503214 Da), N will be always slightly less than c.

The algorithms selected for this project were:

- Exhaustive search

  This is the only algorithm that guarantees the optimal answer. However, the time trade-off makes it infeasible for solving in real time even medium sized problems. It works fine on small masses and can be used as a benchmark for qualitative evaluation.

  It can be implemented through backtracking or nested loops either recursively or iteratively. The time complexity is $O(2^N)$ as at each decision point both branches are considered.

- Dynamic Programming with Bellman's algorithm [3]

  This algorithm can produce good, reasonably fast, solutions for integer Subset Sum problems. The setup of the problem posed for this project differs from the Subset Sub one in just one aspect: the masses are not integers. Moreover, precision is extremely important, ideally at 1 ppm. To increase precision a blowup factor b can be used to scale up the masses. The choice for b will be discussed in chapter 5 Implementation.

  As stated in the previous paragraph, this algorithm requires integer approximations of the capacity and weights. Let these be $int(c)$ and $int(w_i)$ for $i \in \{1..n\}$. It then builds a matrix of size $N * int(c)$ using a greedy approach. Each row considers a new item that can be added. Each column holds capacities less than the column index. For each entry a_{ij} will hold approximately the maximum capacity less than j, that can be achieved from items $x_1..x_i$. The reason this is approximately optimal and not the actual optimal solution is because for each entry a_{ij} only the previous row and the current item are considered. There is a 0 row initialised with null values and then for each row iteratively, matrix entries are computed using an easy recursive formula. Adapted for this problem definition, the formula is 4.1.

$$f_i(j) = \begin{cases} f_{i-1}(j) & \text{for } j = 0, ..., w_i - 1; \\ max(f_{i-1}(j), f_{i-1}(j - w_i) + w_i) & \text{for } j = w_i, ..., c. \end{cases} \tag{4.1}$$

When computing the values for a row, only the previous row is needed and as such, only two rows need to be kept in memory at any time. This makes the space memory rather efficient. The complexity for this algorithm is that of building the table: $O(N*int(c))$. While this is a fast, pseudo-polynomial algorithm, it does not provide any guarantees. Getting optimal answers can be even harder for non-integer values that require high precision.

- SIRIUS-like knapsack

  This approach includes 2 algorithms [5]. Like Bellman's algorithm it rounds the masses to integers as well. Again, a blowup factor b can be used to scale up masses and the choice will be discussed in chapter 5 Implementation.

  The first algorithm *Round Robin* builds an Extended Residue Table of size $(min(mass_{e_1}, ..mass_{e_k}) - 1) * n$, where n is the number of items, which is very small for this problem definition, as low as 6 for CHNOPS. If the element masses are not scaled up this algorithm would not work as the minimum mass is that of Hydrogen (rounded up at 1). The table indicates which values are decomposable. An example can be seen in fig. 4.1 with both figure and caption taken from [5]. The *Round Robin* algorithm presented in [5] for computing this table is both fast and easy to implement. The pseudocode taken from the original paper[5] is found at the end of this section. The formula used to compute an entry is:

  ERT$(r, i) = min\{n \mid n \equiv r (\text{mod } a_1)$ and n is decomposable over $a_1, ..., a_i\}$

  where r is the row and a_i are the item masses.

  The algorithm for the knapsack solving, *Find All*, is refferred to as a mass decomposition algorithm. It has complexity $O(n * min(mass_{e_1}, ..mass_{e_n}) * \gamma(c))$, where $\gamma(c)$ is the number of decompositions of c. Larger c implies more possible decompositions. Since the minimum mass is that of Hydrogen (slightly larger than 1) and n is fixed as the very low number of 6 for CHNOPS the time is largely depended on the total capacity c. The *Find All* algorithm presented in [5] is therefore both fast and easy to implement. The pseudocode taken from the original paper[5] is found at the end of this section.

Residue Table RT

r	n_r
0	0
1	16
2	12
3	8
4	9

Extended Residue Table ERT

r	$a_1 = 5$	$a_2 = 8$	$a_3 = 9$	$a_4 = 12$
0	0	0	0	0
1	∞	16	16	16
2	∞	32	17	12
3	∞	8	8	8
4	∞	24	9	9

Figure 4.1: Residue table and extended residue table for the MCP instance 5, 8, 9, 12

Algorithm 1 ROUND ROBIN

initialize $n_o = 0$ and $n_r = \infty$
for i=2,..k **do**
  $d = gcd(a_1, a_i)$;
  for p=0,...,d-1 **do**
    Find $n = min\{n_q | q \mod d = p, 0 \leq q \leq a_1 - 1\}$;
    if $n < \infty$ **then** repeat $a_1/d - 1$ times
      $n \leftarrow n + a_i; r = n \mod a_1$;
      $n \leftarrow min(n, n_r); n_r \leftarrow n$;
    end if
  end for
end for

Algorithm 2 FIND-ALL(**MASS** M, **INDEX** I, **COMPOMER** C)

if $i = 1$ **then**
  $c1 \leftarrow M/a_1$; output c; return;
end if
$lcm \leftarrow lcm(a_1, a_i); \ell \leftarrow lcm/a_i$; ▷ least common multiple
for $j = 0, ..., \ell - 1$ **do**
  $c_i \leftarrow j; m \leftarrow M - ja_i$; ▷ start with j pieces of a_i
  $r \leftarrow m \mod a_1$; lbound $\leftarrow$ ERT$(r, i-1)$;
  while $m \geq$lbound **do** ▷ m is decomposable
    FIND-ALL$(m, i-1, c)$; ▷ over$(a_1, ..., a_{i-1})$
    $m \leftarrow m - lcm; c_i \leftarrow c_i + \ell$;
  end while
end for

The 7 Golden Rules introduced in section 2.2 Metabolite identification were selected for post-search filtering. For the first 2 algorithms versions with early pruning using the first rule have been devised and the implementations are presented in chapter 5 Implementation. The SIRIUS algorithm for knapsack does not allow for bounding the number of items selected. At no point during processing do the individual numbers reflect, in an obvious or fast to compute manner, the final numbers associated with the items. The complexity would increase more than the gain.

4.3 Structure

There is no structure enforced on this project. To make it easy to use, to integrate and for future code reuse I have opted for a modular approach.

For each algorithm implementation there is an individual script. These are callable from the main script which can be run to perform either of the actions required (run from console, run against a file, call from within the django system).

The element masses and associated data are held in relevant data structures that ease access in a separate script.

There are also independent scripts that can be replaced or used for other programs: a 7 rules filtering script, a script for generating in silico data, and one for the Round Robin pre-processing.

Chapter 5

Implementation

There is one script *runnable.py* that can be used with either of the solving algorithms (or subset) to: run from the console for one datum (i.e. mass), to run against a data set (read from all files in *input_files*, with all results output to relevant files in *output_files*) or, in the future, to be called from the Django system.

There is a script *periodic_table.py* containing two sets of elements: *CHNOPS* and *elements*. *CHNOPS* can be used for testing or to speed up the search when we are confident the resulting molecules will only contain this most common 6 elements. The other more inclusive set, *elements*, contains most elements we would reasonably expect for metabolite mass spectrometry. Retrieving masses is one of the operations most needed. Using the elements as keys inside a formula definitions is extremely useful for any of the implementations. Therefore, each element is held in a frozendict which can retrive in O(1). A frozen dict is immutable, but the elements never change. Therefore, it can be hashed and used as keys in the formula definitions.

Several functions (e.g. computing the mass from a formula) are used across several scripts. These either concern formula manipulation or input-output handling. To make it easier to update and add to the list, these are kept in a separate script *helper_functions*.

The script generating in silico data parses metabolite formulas taken from https://metlin.scripps.edu/metabo_advanced.php and a few non valid molecules, computes masses, generates associate random tolerance measures in the range 1-10 and writes the pairs in one of 3 files (”Small.txt”,”Medium.txt”,”Large.txt”). Masses under 500 Da are considered small, masses in the range 500 - 1000 Da are considered medium and masses over 1000 Da are large. No results were obtained for the Large dataset. More information can be found in chapter 6

There is a folder holding all input files. Another folder for all output files. The output files are organised in folders by input file name. Each output file has in the name the timestamp and algorithm used to produce the output.

The 7 golden rules script contains independent rules implementations so that any can be rewritten easily and they can also be used individually.

All algorithms presented in section 4.2 Selected algorithms have been implemented in individual scripts. Each algorithm provides a function search with parameters: mass, tolerance, delta, restrict. Tolerance is read in for each mass individually and accommodates the equipment measuring errors. Delta is used to allow for computational errors and it is set at 2 levels of precision higher than the required precision of 1 ppm. Restrict is a boolean which indicates whether to search only for CHNOPS. Given that most metabolites contain only CHNOPS and that there is a significant speed up for fewer items, restrict has been set to be True.

- Exhaustive search

  This was implemented with the obvious backtrack algorithm. I chose a recursive approach with a recursive depth of N (the number of decision points). The complexity as stated before is $O(2^n)$. There are two reasons why such an implementation might not be desired: it is not *pythonic* and the default recursion depth in Python is 999. However, there are reasons to recommend it: it has less code and it is easier to implement, read, update.

  Given that the depth can be set higher and for most practical purposes this implementation can not be used for masses warranting more than 999 depth, I chose to forego the non-*pythonic* issue.

- Exhaustive search with 7 rules pruning

  This is a tweaked implementation of the former. It prunes paths that exceed the bounds enforced in the first rule of the 7 golden rules. It also performs 7 golden rules filtering before accepting a solution.

- DP Bellman

  This algorithm was implemented following the description in chapter 4 Design. As discussed there, the masses are rounded to integers. To increase precision a blowup factor b was used to scale up the masses. The complexity $O(N * int(c))$ increases linearly in b as $c' = b * int(c)$. Several powers of 10 were experimented ($10^1..10^4$). The only reasonable effect was that of 10^1. This does not help maintain the precision needed of ideally 1 ppm. Qualitative improvement was only noticeable for $b \geq 10^3$. However the time was comparative to the exhaustive search with less results.

- DP Bellman with 7 rules pruning

  This is a tweaked implementation of the former. It prunes paths that exceed the bounds enforced in the first rule of the 7 golden rules. It also performs 7 golden rule filtering before accepting a solution.

- SIRIUS-like knapsack

  The algorithm were implemented following the description and algorithms in chapter 4 Design. *Round Robin* was implemented as a separate script that produces an Exdended

Residue Table with a set precision (discussion in the next paragraph) and writes it a .txt file. Therefore, the complexity and time of *Round Robin* were not deemed important as it is run once before the search and the same table can be used every time.

As discussed in chapter 4, the masses are rounded to integers as well. To increase precision a blowup factor b was used to scale up the masses. The complexity of the knapsack algorithm *Find All* is as stated before $O(n * min(mass_{e_1}, ..mass_{e_n}) * \gamma(c))$, where $\gamma(c)$, which is linear in c. Therefore it would increase linearly in b as $c' = b * int(c)$. Several powers of 10 were experimented ($10^1..10^7$). Due to better speeds than Bellman, b can be set at 10^5 and still keep most processing times at worst around a second. A blowup factor of 10^6 would have maintained the ideal precision of 1ppm.

Chapter 6

Results

All algorithms were tested on 2 files *Small.txt* and *Medium.txt*. There are 17 pairs (mass, tolerance) in *Small.txt* and 7 pairs in *Medium.txt*. Both files contain 2 fake pairs that should be identified by the knapsack algorithms and filtered out by the 7 rules.

6.1 Qualitative

The exhaustive search produced output with very few false positives for both files. The version with early pruning produced the same results as post filtering. However, after pruning, only 12 out of the 15 small valid molecules and 4 out of 5 medium valid molecules were maintained. After careful inspection, it seems that indeed the formulas filtered do not comply with some of the 7rules and are rather unusual metabolites.

The Dynamic Programming Bellman's script produced only 3 molecules for the small set and none for the medium. The post pruning version correctly filtered out two of those while the early pruning version wrongly filtered out all.

The SIRIUS-like knapsack script produced large outputs both for the small and medium dataset. The number of false positives grows with mass. For example masses under 100 have produced only one formula, masses in the range 100-200 a few formulas per mass, while masses in the medium dataset produced hundreds of results per mass. After filtering, the small dataset output was more appropriate while the large dataset still contained tens of false positives. The reason for this is that the tolerance for knapsack had to be set slightly more lax than for the previous two algorithms.

6.2 Speed

Various tables with speeds are recorded below.

algoritm	time in min:sec
Exhaustive Search	6:18
Exhaustive Search with pruning	15:01
DP Bellman	2:08
DP Bellman with pruning	3:20
SIRIUS-like knapsack	0:32

Figure 6.1: Speeds on the entire Small dataset (17 masses)

algoritm	time
Exhaustive Search	<3.5 hours
Exhaustive Search with pruning	<4.5 hours
DP Bellman	<7 min
DP Bellman with pruning	<1 min
SIRIUS-like knapsack	< 8min

Figure 6.2: Speeds on the entire Medium dataset (7 masses)

The speeds for the previous project at the University of Glasgow[2], reproducing SIRIUS algorithms, are found in 6.4. It is obvious that the knapsack implementation for this project is much faster both for small and medium sized molecules.

blowup factor b	seconds
100	0.013
1000	0.113
10000	0.441
100000	4
1000000	60

Figure 6.3: Speeds for generating the Extended Residue Table

Mass	Precision	Range	Time
100	low	1	42s
500	low	1	2624s
500	low	0.01	128s
100	normal	0.01	26s
100	normal	0.1	237s
500	normal	0.01	4975s

Figure 6.4: Speeds for various masses and precisions (low = 0.0005 and 0.0001) in previous project [2]

Chapter 7

Future work

7.1 Integration

At the beginning of the project there were hopes of integrating it in the system at Glasgow Polyomics, through the pipeline that was being built at the same time as another MSc project. However this aim could not be accomplished in this time frame.

Integration is independent of the specific algorithm used and it could therefore have been done in parallel with the knapsack algorithms implementation once the exhaustive search script was completed. However, this would have taken time away from refactoring which I believe has lead to such time optimizations when compared to other implementations such as the one by Benjamin [2].

This project can be easily integrated through the *runnable.py* script. The only required implementation is that of a function definition that parses parameters, makes the appropriate call to the best algorithm (presumably the SIRIUS-like knapsack implementation) and then packages the results in the desired format.

7.2 Reflection

At the beginning of the project, the language R was chosen and several weeks were spent learning and adjusting to the languages specifics. As mentioned in section 4.1 Language, R was not suitable for the project in the end. In the future I would rather consider such esoteric languages with limited support only if there is a much stronger case in its favour.

Chapter 8

Conclusion

Several algorithms that solve the problem posed in the Introduction were implemented and evaluated against each other. The results reflect that the SIRIUS-like knapsack algorithm produces better output at much faster speeds than more classical algorithms.

The system has several other functions such as generating test data from molecule formulas, running against a dataset read from file, filtering results with the 7 golden rules discussed in the Background section. It can also be run from the console with intuitive commands that walk the user through entering the data and parameters needed. As discussed in the previous chapter, it is very easy to integrate in the system for which it was designed.

Overall, I feel that I have achieved the initial goals of the project.

Bibliography

[1] Glasgow Polyomics at the College of Medical, Veterinary and Life Sciences, University of Glasgow. `http://www.polyomics.gla.ac.uk/`. Accessed: 2015-08-30.

[2] Benjamin Behm. *Implementation of knapsack problem solving algorithms for metabolite identification*. 2013.

[3] Richard Bellman. 6. dynamic programming i. 1958.

[4] Sebastian Böcker, Matthias C Letzel, Zsuzsanna Lipták, and Anton Pervukhin. Sirius: decomposing isotope patterns for metabolite identification. *Bioinformatics*, 25(2):218–224, 2009.

[5] Sebastian Bocker and Zsuzsanna Lipták. A fast and simple algorithm for the money changing problem. *Algorithmica*, 48(4):413–432, 2007.

[6] Salvatore Cappadona, Peter R Baker, Pedro R Cutillas, Albert JR Heck, and Bas van Breukelen. Current challenges in software solutions for mass spectrometry-based quantitative proteomics. *Amino acids*, 43(3):1087–1108, 2012.

[7] George B Dantzig. Discrete-variable extremum problems. *Operations research*, 5(2):266–288, 1957.

[8] Minoru Kanehisa, Susumu Goto, Masahiro Hattori, Kiyoko F Aoki-Kinoshita, Masumi Itoh, Shuichi Kawashima, Toshiaki Katayama, Michihiro Araki, and Mika Hirakawa. From genomics to chemical genomics: new developments in kegg. *Nucleic acids research*, 34(suppl 1):D354–D357, 2006.

[9] Tobias Kind and Oliver Fiehn. Seven golden rules for heuristic filtering of molecular formulas obtained by accurate mass spectrometry. *BMC bioinformatics*, 8(1):105, 2007.

[10] David Pisinger. Algorithms for knapsack problems. 1995.

[11] Rob Smith, Andrew D Mathis, Dan Ventura, and John T Prince. Proteomics, lipidomics, metabolomics: a mass spectrometry tutorial from a computer scientist's point of view. *BMC bioinformatics*, 15(Suppl 7):S9, 2014.