

QUBO Formulation of the Cardinality Constrained Portfolio Optimisation Problem

Sam Shailer

Abstract

In this paper I consider the Cardinality Constrained Portfolio Optimisation (CCPO) problem and how it can be formulated as a Quadratic Unconstrained Binary Optimisation (QUBO) problem. The CCPO problem follows the standard mean variance portfolio optimisation problem but adds a constraint limiting the number of assets in the final portfolio. The CCPO also constrains the values the weighting of assets can take. This project focuses on using the binary encoding scheme to represent the weights of each asset as a set of binary variables. This should allow the complete modeling of the CCPO as a QUBO without the need to simplify the problem.

Index Terms

Optimisation, QUBOs, Quadratic Unconstrained Binary Optimisation, Cardinality Constrained Portfolio Optimisation, Simulated Annealing, Portfolio Optimisation.

I certify that all material in this dissertation which is not my own work has been identified : Sam Shailer

CONTENTS

I	Introduction	1
II	Summary of Literature Review and Specification	2
II-A	Literature Review Summary	2
II-B	Specification Summary	3
III	Design	4
III-A	The QUBO Model	4
III-B	The CCPO Problem	5
III-C	Simulated Annealing	7
III-D	Pipe-lining	8
IV	Development	10
IV-A	Prototype	10
IV-B	QUBO Formulation	11
IV-C	Simulated Annealing	13
IV-D	Correlation to Covariance	14
IV-E	Portfolio Optimiser	14
V	Testing	17
V-A	Testing the QUBO formulation	17
V-B	Testing simulated annealing	18
VI	Description	19
VII	Evaluation	19
VIII	Critical Assessment of the Whole Project	20
IX	Conclusion	20
	References	21

I. INTRODUCTION

OPTIMISATION gives us the opportunity to find the most efficient solution to many of the worlds problems. This can range from modeling climate change with greater accuracy, optimising the medicine a patient is taking or decreasing the financial risk of an investment. While optimisation was developed and used before computers existed, the epoch of the digital age completely changed the scale at which it was possible to optimise problems. The ability to store vast amounts of information meant that much more detailed and complex solutions could be found. We have reached a point however where it's becoming a lot more computational difficult to sort through the amount of data we have.

The two solutions to this are to either increase the power of computer hardware or increase the efficiency of computer algorithms. The research completed in this paper looks at the later of the two solutions and how it can be applied to a specific problem within financial portfolio optimisation. The problem this paper explores is known as the **Cardinality Constrained Portfolio Optimisation (CCPO)** problem. This problem can be described as deciding which assets to include in a portfolio to make it efficient, given a list of assets and a set number of assets to be included. This problem is widely researched within the field of portfolio optimisation as it is often encountered by fund managers who have to deal with restrictions such as customer preference or asset transaction cost.

The model this paper investigates for formulating the CCPO problem is known as the **Quadratic Unconstrained Binary Optimisation (QUBO)** model. Quadratic optimisation models often include a matrix of variables acting on a Hessian matrix, a matrix describing the landscape of the problem, with the addition of the same variable matrix working on a matrix of constraints. The QUBO model is based around incorporating that constraint matrix into the Hessian matrix. By modeling optimisation problems in this way it has been shown that they can be solved efficiently using methods such as simulating annealing, tabu search and quantum annealing. Part of the project documented in this paper has also been to implement a simulated annealing algorithm and use it to find or approximate solutions to CCPO problems.

The difficulty with modeling straight portfolio optimisation using the QUBO model is the fact that assets don't tend to be binary by nature. Most of the research currently done in this field assumes that the decision being made is only whether to buy or not to buy an asset. However this can potentially lead to less efficient portfolios which only really give a rough idea of what the optimal portfolio would look like. In this project integer to binary encoding has been implemented to give an example of how this could be approached in the future. It will also show what difficulties arise when using integer to binary encoding for this type of problem. The variables encoded in this project will represent the integer percent of an asset to be bought.

The tests done in this project are aimed at gathering data on how viable it is to formulate the CCPO problem as a QUBO and solve it via simulated annealing. For this to be properly represented the tests undertaken use my simulated annealing algorithm, an externally developed simulated annealing algorithm and an externally developed tabu search algorithm. This ensures that results correctly determine whether firstly my simulated annealing algorithm is working efficiently and secondly whether it is viable to formulate the CCPO problem as a QUBO using the method shown in this paper. The results from my tests will show whether it is possible to model the CCPO problem as a QUBO and how efficient it is to model the CCPO problem as a QUBO. The data my testing will produce is a range of solutions to different CCPO problems which can then be converted into an efficient frontier and graphed. This will clearly show how well the project has gone and make it easy to compare my results to other peoples results.

This problem was suggested by Fujitsu, who have interests in formulating many real world problems to be solved using their quantum-inspired digital annealing units. Fujitsu's digital annealing units are specialist computer processors for digital annealing. The input to Fujitsu's digital annealing units is in the QUBO form. The main benefits offered by Fujitsu's digital annealing units are that they have a higher level of accuracy than any quantum technology in the market today and operate at room temperature, instead of absolute 0 which is required for most quantum computers. Doing this allows Fujitsu's digital annealing units to be seamlessly integrated into current data centers without the need for large specialist equipment. This offers Fujitsu a massive edge in the market currently

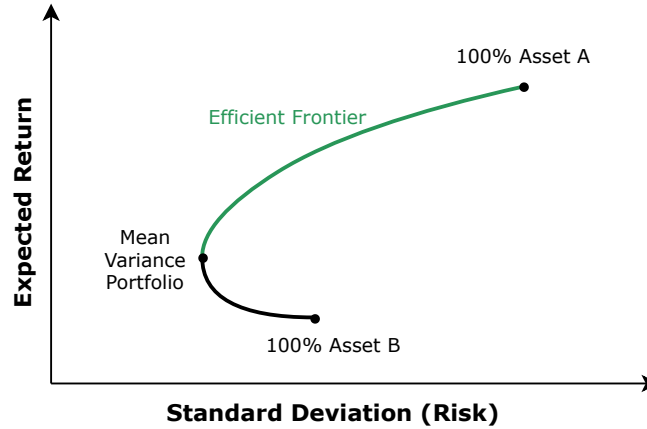
and large amounts of R&D is being done by Fujitsu to formulate problems so they can be solved on the digital annealing units. For this reason the project documented in this paper will hopefully help to shed light on methods that might help in their research.

II. SUMMARY OF LITERATURE REVIEW AND SPECIFICATION

A. Literature Review Summary

PORTFOLIO optimisation accounts for a large set of optimisation problems, in which the goal is to find the optimal combination of assets to achieve a desired outcome. This outcome could be to reduce the risk of an investment, increase the returns of an investment or even just find a valid portfolio given a number of constraints. In 1952 H. Markowitz [1] introduced the idea of diversification of portfolios. Previously it had been assumed that the relationship between assets in a portfolio was linear. This would mean that as a higher percentage of an investment was placed in one asset, and less in another, the total investment would adopt more of the characteristics of the asset being increased. It turns out this isn't actually the case, as the mere process of investing in another asset decreases the level of risk of the total investment. This causes a region where investing more in a less risky asset with lower returns actually increases the risk of the total investment while reducing the potential return. This theory developed by Markowitz is known as **Modern Portfolio Theory (MPT)**. In H. Markowitz's book [2] this theory is described mathematically and the equation it can be optimised. A set of portfolios can be produced where the risk is lower than just investing in one asset, while the return is higher. All portfolios in this set make up what's known as the efficient frontier as shown in fig. 1.

Fig. 1. Example of the possible portfolios between two assets[3]



Once the ability to optimise portfolios was discovered, lots of methods were researched and used to find efficient portfolios. The next problem was figuring out how to compare different portfolios to measure how well they would perform compared to other portfolios. While numerous methods have been developed to grade how optimal a portfolio is one of the most common used is the Sharpe ratio. Introduced by Sharpe in his 1966 and 1975 papers [4], [5] this formula compares the portfolio developed to a known benchmark, such as a market portfolio (e.g. the S&P 500). There are two main versions of the Sharpe ratio described in Sharpe's 1994 paper [6], these being the ex-ante and ex-post Sharpe ratios. The ex-ante Sharpe ratio calculates a ratio which indicates the expected differential return per unit of risk, and is described with the following formula.

$$S = \frac{\bar{d}}{\sigma_d} \quad (1)$$

Where S is the Sharpe ratio, \bar{d} is the difference in expected return of the optimised and market portfolios and σ_d is the standard deviation of d . The ex-post Sharpe ratio is described using a similar formula and calculates a ratio which indicates the historic differential return per unit of historic risk.

The **Quadratic Unconstrained Binary Optimisation (QUBO)** model is an optimisation model where the objective function is formulated as a matrix of coefficients and a matrix of variables. The QUBO model has been researched substantially over recent years due to its ability to model many different combinatorial optimisation problems found in industry [7]. The power of QUBO solvers, such as Fujitsu's digital annealing and D-Waves quantum annealing, can be used to efficiently solve many of the problems described in the paper above. The QUBO model is also closely related and computationally equivalent to the Ising model, therefore it lends itself well to being solved via quantum mechanical methods such as quantum annealing. Methods for formulation of objective functions using the QUBO model are explained well in the paper by Glover, Kochenberger Du. The formula below describes the QUBO model mathematically.

$$\begin{aligned} \text{minimise: } & f(x) = x^T Q x \\ \text{subject to (st):} & \end{aligned} \quad (2)$$

$$x_i \in \{0, 1\} \quad (3)$$

In MPT the variables representing assets are normally referring to the percent of the total investment placed in that asset. The QUBO model requires variables to be binary which can make it difficult to adapt portfolio optimisation problems to work with this format. Some research papers get around this problem by assuming the assets in the portfolio are binary by nature in the first place. An example of this could just be whether or not to buy the asset [8]. Other papers describing similar optimisation problems use integer to binary encoding. Integer to binary encoding can be done in a number of ways, involving one-hot encoding, binary encoding and unary encoding. Each of these encoding schemes offer different benefits mainly relating to the hardware they're running on. The problem with integer to binary encoding is that the number of variables needed to describe a problem increase a lot. For this reason there aren't many papers describing the formulation of portfolio optimisation using the QUBO model, and even less attempting to use binary encoding to model asset ratios. This is the motivation for using integer to binary encoding in this project.

B. Specification Summary

IN the literature review a set of functional and non-functional requirements were given. After completing the literature review, I had further discussions with Fujitsu who encouraged me to slightly change the requirements of the project based on research they were currently doing. Below is list of requirements some of which have stayed the same and others which have changed.

Summarised Functional Requirements

- The portfolio optimisation problem being researched will be the CCPO problem. New
- The portfolio optimisation problem will need to use real market data such as the historic open and close prices for stocks on a particular market.
- The quadratic optimisation model will need to incorporate the integer assets, co-variance matrix, potential return and constraints of the portfolio.
- The encoding scheme, level of precision and constants required to encode the integer variables as binary variables will need to be implemented.
- The quadratic optimisation model will need to be transformed into the Q matrix.
- A simulated annealing algorithm will need to be coded in order to find optimal solutions to the QUBO problems.
- The simulated annealing algorithm will need to accept both the number of variables in the Q matrix and the Q matrix.
- The simulated annealing algorithm must check that the input matrix has a width and height equal to the number of variables chosen.
- Another algorithm will be written that takes the data of the optimal portfolios for a particular portfolio optimisation problem and plots them on a graph.

Summarised Nonfunctional Requirements

- A range of different combinations of assets need to be tested.
- The simulated annealing algorithm must be tested with published data to make sure it works correctly
- The time taken for the algorithm to find an optimal solution to the portfolio optimisation problem needs to be reasonable.
- The number of assets in the portfolio optimisation problem must be kept low to reduce computation time but also large enough to produce usable results.
- The portfolios found need to be some what optimal.

These requirements differ from the requirements in the literature review firstly because they have been summarised so may not appear exactly the same. Secondly they differ because of input from Fujitsu. Fujitsu have been a prominent input in the development of this project. After the initial writing of the literature review they suggested the the project focus more around the CCPO problem described in this paper [9]. This was down to the fact Fujitsu encounters the CCPO problem often and are currently working on finding better methods of solving it, requiring all the support they can get.

III. DESIGN

THIS project will be written in python using the procedural programming paradigm. While python is an object oriented programming language it offers itself well to being written in a procedural way. The reason python has been chosen is that it is quick to develop and deploy applications and it offers a wide range of libraries which support working with the QUBO model. This project is also being designed as a proof of concept to show whether it is possible to formulate the CCPO problem as a QUBO using integer to binary encoding for asset weights. This means that speed and efficiency will not be evaluated so the fact python is known for being slower than the likes of C is not a problem in the scope of this project. Python is also known for its developer friendly design and readability which means if this project was to be used externally it will be quick and easy to understand the inner workings.

A. The QUBO Model

THE QUBO problem, as discussed previously, involves taking a problem to optimise and formatting it as a matrix of values and a matrix of binary variables. The benefits of using python is that there are numerous external libraries which offer support in formulating QUBOs. The two main libraries used for modeling QUBOs in python are Qubovet[10] and PyQUBO[11][12]. Both libraries offer very similar functionality, they both make it simple to model binary variables, write objective functions using these variables, add constraints to the objective functions, convert the objective function to a QUBO and convert the QUBO to other formats. After reading the documentation of both packages, being able to convert from a QUBO to a binary quadratic model (BQM) is important for solving via the D-Wave solvers. The BQM allows for easy encoding of both Ising models and QUBOs which is why its used by D-Wave, so later when testing my simulated annealing algorithm against other third party algorithms this will be very useful. The reason this is done in the module is that Ising models while very similar to QUBOs use spin variables (1, -1) instead of binary variables (1, 0). For the scope of this project Ising models are not used so it won't be discussed further. Unfortunately due to the relatively small community working with QUBOs there is not a large amount of documentation for either of these libraries and very little discussion on problem solving with either library. Therefore which features from each library would be used and which ones I would implement myself was also taken into account. The main difference between PyQUBO and Qubovet is that PyQUBO uses a C++ back-end. This makes converting the objective function to a QUBO and other formats faster than Qubovet which is written predominantly in python. For this reason I decided it would be best to use PyQUBO as while speed is not a goal of the project, when parameter tuning and testing is done with larger sample sizes any improvement in speed will be useful.

The next design consideration for how to implement a QUBO in python was deciding how to handle the constraints of the original objective function. The PyQUBO library offers built in constraint declaration and validation. This makes it easy to write simple constraints, adjust penalty values and validate whether the solver has broken the constraints. While initially from a design standpoint this seemed faster to implement and use, the abstraction from

how the library does this looked like it might make it harder to implement more complex constraints. The other issue with using external modules is that they have a large range of functionality which wasn't needed for my project. This meant that functions which appeared simple actually consisted of many lines of code to cover all their different use cases. This led to the design decision to implement constraints manually as then I would have a complete understanding of how they were working. This also meant that if the constraints weren't working then it was my problem and not an issue with the library.

The next consideration when designing the QUBO model in python was to decide how it would be exported. The way PyQUBO works is that it allows you to create an objective function and then compile that objective function into a model. The model is a data type unique to PyQUBO and stores the information about the QUBO (or Ising model) and then gives the user the ability to get the QUBO (or other models) from that model. As this project is working with QUBOs the main variable we want to store is the QUBO model. The QUBO returned by the ".to_qubo()" function of the PyQUBO library is of the form "dict[(str,str), float]", where the two strings are the variables and the float is their respective coefficient. The variables in the dictionary output by this function can also be in any order. While this function immediately gives a QUBO the formatting is not ideal for working with later on. This is because doing matrix multiplication with dictionaries would be challenging and making sure the variables are ordered correctly would be time consuming. A well known library called NumPy [13] has industry standard methods for using arrays and doing matrix multiplication and would be very useful for storing and working with the QUBO matrices. While PyQUBO unfortunately doesn't offer any methods for converting their QUBO datatype into a NumPy array, the BQM datatype mentioned earlier can be converted to a NumPy array using a simple function. This function is contained within the Dimod[14] package created by D-Wave, it takes in a BQM and the variable order for the array and returns a NumPy array of the coefficient matrix Q. For this reason the use of NumPy arrays and the "to_numpy_matrix()" function from Dimod will be used to export the QUBO Q matrix. The NumPy array output can also easily be converted back into the QUBO datatype so use outside of this project is still possible.

B. The CCPO Problem

THE Cardinality Constrained Portfolio Optimisation (CCPO) problem involves finding a proper subset of elements, given a set of elements, where the size of the subset is chosen in advance and the elements selected for the subset have been optimised in some way. In the scope of portfolio optimisation this means finding the efficient frontier of a standard mean variance portfolio where the portfolios must have a specified number of assets. This is explained in depth in the paper by T. Chang [9]. The standard mean-variance portfolio can be plotted as a continuous curve whereas the CCPO problem makes this curve discontinuous. The fact the efficient frontier may be discontinuous makes it considerably harder to solve. By using an exact approach to the problem based on weighting, as shown in fig. 2, it is easier to design a computationally effective heuristic algorithm that can solve the CCPO problem. The disadvantage of using this approach is that certain portions of the efficient frontier will be invisible.

Fig. 2. The exact approach to the CCPO problem based on weighting from [9]

$$\text{minimise } \lambda \left[\sum_{i=1}^N \sum_{j=1}^N w_i w_j \sigma_{ij} \right] - (1 - \lambda) \left[\sum_{i=1}^N w_i \mu_i \right] \quad (16)$$

subject to

$$\sum_{i=1}^N w_i = 1, \quad (17)$$

$$\sum_{i=1}^N z_i = K, \quad (18)$$

$$\varepsilon_i z_i \leq w_i \leq \delta_i z_i, \quad i = 1, \dots, N, \quad (19)$$

$$z_i \in [0, 1], \quad i = 1, \dots, N. \quad (20)$$

The CCPO problem recommended for this project by Fujitsu is the one shown in fig. 2. As mentioned in the previous paragraph this approach may cause portions of the efficient frontier to be invisible. As discussed in [9] there are two reasons this approach is still used:

- firstly, by solving the problem using a heuristic approach it is possible to gain information about the invisible portions;
- secondly, attempting to design a computationally effective heuristic solver that can address the non-weighted approach is very difficult.

In the scope of this project the solvers used will be heuristic in nature and therefore the reasons above also apply.

The first design consideration to discuss for implementing the CCPO problem is how to handle variables. As can be seen in fig. 2 there a quite a large number of variables used. Below is a breakdown of what each variable is as whether it will need to be input into the program:

- N : The total number of assets available - Input
- σ_{ij} : The co-variance between two assets (asset i and asset j) - Input
- μ_i : The expected return of asset i - Input
- λ : The weighting parameter - Input
- K : The total number of assets to be included - Input
- ε_i : The lower bound of the weighting of asset i - Input
- δ_i : The upper bound of the weighting of asset i - Input
- w_i : The weights of the assets - Not input
- z_i : Whether an asset i is included or not - Not input

From this breakdown it is clear that the CCPO formulation program should have 7 inputs. Out of these inputs you can only add 1 for each asset to the total number of assets therefore N and K must be non-negative integers. The upper and lower bounds ε and δ can be either floats to represent the decimal fraction of an asset (e.g. $\varepsilon = 0.15$ and $\delta = 0.78$) or percents (e.g. $\varepsilon = 15\%$ and $\delta = 78\%$). The weighting parameter λ , as described in the paper, will be between 0 and 1 so should be stored as a float. The expected return of assets μ and the co-variances between assets σ will both need to be stored as collections of multiple values. There are various methods of doing this in python such as lists or NumPy arrays. Since after being input the shape will not need to change it makes sense to store them using NumPy arrays. NumPy arrays offer a faster and more compact method of storing multidimensional data in python compared to lists. They are also used in libraries such as PyQUBO and Dimod mentioned in the previous subsection which will make it easier to integrate my program with those existing libraries.

The next design consideration to discuss is how the weights of assets will be implemented in my program. As described in previous sections, the final QUBO model uses only binary variables whereas the original asset weights in the problem are fractions. This introduces the need for integer to binary encoding so that a set of binary variables can be used to represent their integer counterparts. The general equation for binary encoding is given below:

$$w = \sum_{d=0}^{D-1} 2^d x_{d+1} + C x_D \quad (4)$$

$$C = U - 2^D + 1 \quad (5)$$

$$(6)$$

Where D is the total number of variables needed to encode the integer variable, U is the upper bound of the asset weight. For the design of this part of the project the first consideration is how the integer variables will be represented. In the paper describing the problem the weights are stated to be a fraction between 0 and 1. Originally I thought that the asset weights could be represented as integers from 0 to 100. This would mean the equation above could be implemented without change and the results would be easier to understand. Using this method I thought that the value of weights in the results could then be divided by 100 to represent their decimal equivalent.

This method is demonstrated below where an integer variable constrained between 0 and 100 has been converted to a sum of binary variable ($D = 7$ and $U = 100$):

$$0 \leq w \leq 100 \quad (7)$$

$$w = x_1 + 2x_2 + 4x_3 + 8x_4 + 16x_5 + 32x_6 + 37x_7 \quad (8)$$

Upon looking at how the asset weights were used in figure 2 however, I realised that using this method would cause problems with how the weights interact with each other. I therefore decided the best way to represent the asset weights would be to follow the original paper and look to encode them as binary variables where the overall value would be between 0 and 1 as shown below:

$$0 \leq w \leq 1 \quad (9)$$

$$w = 0.01x_1 + 0.02x_2 + 0.04x_3 + 0.08x_4 + 0.16x_5 + 0.32x_6 + 37x_7 \quad (10)$$

C. Simulated Annealing

Simulated annealing is a metaheuristic algorithm for solving combinatorial optimisation problems. It was originally developed in 1983 by Kirkpatrick [15] and was based off the physical process of annealing. Many different versions of simulated annealing, such as fast simulated annealing (FSA [16]) and adaptive simulated annealing (ASA [17]), have now been developed however these can be significantly harder to implement compared to standard simulated annealing. In the scope of this project standard simulated annealing will therefore be used. The principles of simulated annealing were discussed in the literature review and the pseudocode for standard SA is given below:

Algorithm 1 Standard Simulated Annealing

```

Select the number of iterations  $iterations > 0$ 
Select an initial state  $i \in S$ 
Select an initial temperature  $T_0 > 0$ 
Set cooling rate  $0 < \alpha < 1$ 
for  $k = 0, k < iterations, k++$  do
    Generate state  $j$ , a neighbour of  $i$ 
    Calculate  $\delta = f(j) - f(i)$ 
    if  $\delta < 0$  then
         $i = j$ 
    else if  $\text{random}(0, 1) \leq \exp(-\delta/T)$  then
         $i = j$ 
    end if
     $T = \text{new\_temperature}(T_0, k, \alpha)$ 
end for

```

In the code above the values for the start temperature and cooling rate (α) can only be made through investigation so these will be covered when the algorithm is implemented. The methods for finding a neighbour and calculating a new temperature each iteration have multiple possible solutions. The choice of how to tackle these problems will be discussed in the following paragraphs.

The first consideration to make is how neighbours will be found. In simulated annealing finding a neighbour is the process of finding a new combination of variables based on the current combination. This can be as simple as the current variable having a value of 12 so a neighbour could be 13. In the paper by T. Chang[9] their version of a simulated annealing algorithm chooses the neighbour as either being an asset 1 or 2 moves away from the current asset. Choosing a neighbour like this however can lead to algorithms being applicable to only one problem (heuristic). This is one of the major benefits of the QUBO format, the function to optimise will always follow the same arithmetic and variables will always be binary so the neighbour function can be completely independent of the problem itself. This means only one SA algorithm is needed to solve any QUBO problem. This type of SA

algorithm can therefore be classed as metaheuristic. The possible methods for selecting a neighbour in a solution space containing only binary variables are listed below:

- 1) Flip a variable and select the result as the neighbour (1-opt)
- 2) Flip multiple variables and select the result as the neighbour (k-opt)
- 3) Collect a set of possible neighbours using the first or second method and pick the best result

This is further elaborated in the following paper by K. Katayama [18]. As stated in the paper the issue with k-opt is that it causes the neighbour space to grow exponentially. This can lead to neighbours being selected which are considerably far from the original solution and preventing the SA algorithm from converging to a feasible solution. The occasion in which k-opt might become beneficial is if all neighbours could be calculated and the best could be found, leading to a faster convergence. Unfortunately due to the neighbour space growing exponentially this is too computationally difficult. The much simpler alternative is to use 1-opt. This could be improved by finding the best neighbour in the space of possible neighbours. The issue with doing this is that by the nature of simulated annealing there needs to be a level of freedom so that the algorithm doesn't get stuck in a local minima. By only selecting the best neighbour there is the possibility that the algorithm will get stuck. For these reasons this project will implement the 1-opt neighbour selection method in the SA algorithm.

The next consideration to make is what annealing schedule to implement. The annealing schedule affects the rate at which the algorithm converges to a solution. In SA if a neighbour doesn't have a better value than the current solution then it depends on the temperature whether it is still selected. As the temperature decreases the probability a bad neighbour is picked also decreases until no bad neighbours are picked. The annealing schedule controls the rate at which the temperature decreases. If this is done too quickly then the search space is not being properly covered and it is likely the algorithm will end up getting stuck in a local optimum. If its done too slowly then a solution might not be found and the algorithm becomes less efficient than solving via brute force. A chapter in the book Encyclopedia of AI [19] covers in depth the different types of annealing schedules. In this chapter a study is done into how each annealing schedule performs. It was found that the non-monotonic adaptive cooling schedule was the best with the exponential annealing schedule not performing much differently. It is also important to note that the exponential annealing schedule is considerably simpler to implement. With its good performance and simple implementation the exponential annealing schedule will be used in this project. The formula for the exponential annealing schedule is given below.

$$T' \propto T_0 \times (\alpha^i) \quad (11)$$

Where T_0 is the initial temperature, T' is the new temperature to calculate and α is the cooling rate. The cooling rate for exponential annealing is generally between 0.8 and 0.9.

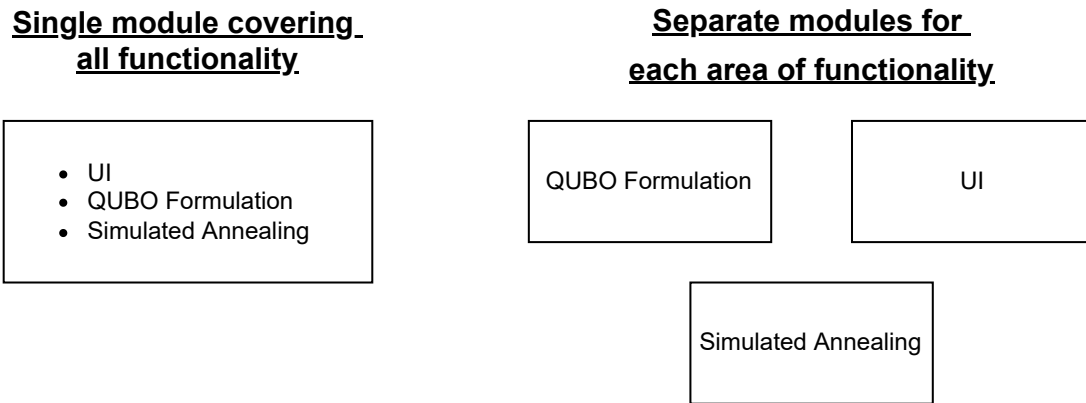
D. Pipe-lining

Pipe-lining is the process of taking all the different parts of the project and putting them together to provide the required functionality. The first design consideration to make for pipe-lining is how the project will be structured. Since python has been selected as the language for this project it would be possible to produce all the functionality in a single module and then split processes into separate functions. The advantage of using this method is that all the code would be in the same place so it is easy search through. It also means less modules would need to be imported and functions could be added faster. The disadvantages to using this method are the decrease in readability of the final code base. With all the code in one place it would become increasingly hard to understand what everything is doing and keep track of all the functionality.

The other method for structuring the project is to group each set of functionality into separate modules. Both methods are shown graphically in fig. 3. The advantages to using this method are firstly that code becomes a lot more readable. Its easy to tell which module specific functions will be in and can support with decoupling each processes. The second advantage to using this method is that functions from different modules can be reused. If the project is used again in the future each module and their functions could easily be used independently of each

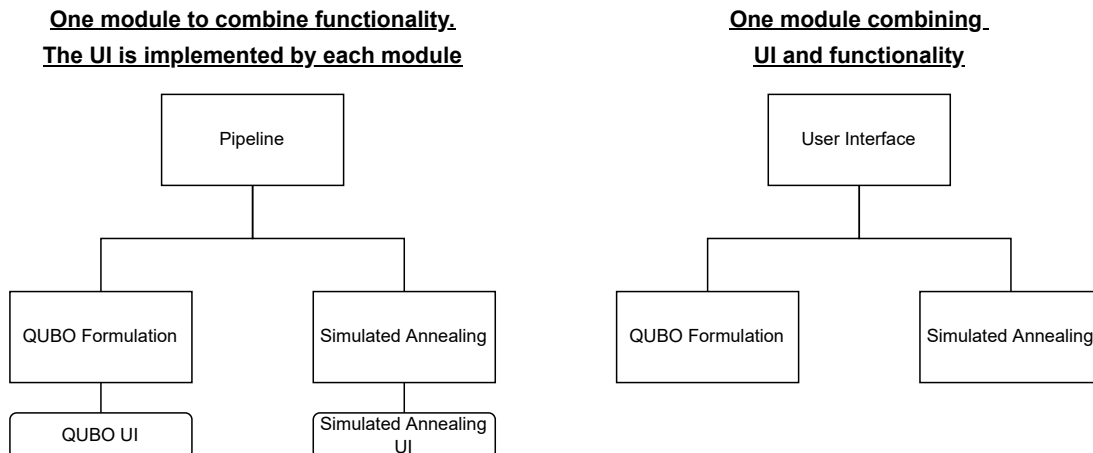
other. The only disadvantage to this method is that functionality will be more dispersed. Based on the advantages and disadvantages of each method I have chosen to implement separate modules for functionality in this project.

Fig. 3. Different project structures



The next design consideration for pipe-lining is how the user will interact with different functionality. The purpose of pipe-lining is to allow the user to easily use different features of the project. Based on the previous design consideration, the two main options for how this could be designed are: to include user interface (UI) elements within the separate modules, where the modules UI elements relate to the functionality of the module and a separate module just combines the other modules together; or to develop a module which puts the functionality of all the other modules together and also makes the UI to allow the user to interact with all the different components of the project. These two methods are shown in fig. 4. The advantage of the first method is that it helps to keep functionality self contained. It would also allow modules to be used separately without needing to be interfaced. The disadvantage to this method is that it could limit how modules interact with each other and impose strict uses on the modules. The advantages to the second method is that it keeps the functionality of modules self contained and makes them more reusable. Based on the characteristics of these two methods and the project as a whole I think it would be beneficial to use the second method for implementing a UI.

Fig. 4. Different UI methods



IV. DEVELOPMENT

THE development section will be broken up into stages covering the development of the prototype, the development of the QUBO formulation module, the development of the simulated annealing module and the development of the user interface.

A. Prototype

TO quickly understand how the CCPO problem could be implemented in python and discuss the direction of the project with Fujitsu, I chose to develop a prototype. When prototyping I had initially decided to use the Qubovert library for formulating the QUBOs. This library offered simple ways of adding constraints to the model and inbuilt simulated annealing functionality.

The first stage to developing the prototype was deciding what the requirements were. Based on the full project specifications and a previous meeting with Fujitsu, I decided that the main interest was to implement the CCPO problem in python using binary assets. This means that instead of asset weights being optimised it would just find which assets should be selected. The benefits of prototyping this is that it would improve my understanding of how the mathematical formula in fig. 2 can be implemented in python without the difficulty of implementing integer to binary encoding. Since the Qubovert module also has solvers built in it also shouldn't be too difficult to solve the QUBO.

The second stage to developing the prototype was to actually implement the cardinality constrained objective function. I firstly generated a boolean variable for each of the assets stating whether it had been selected or not. I then decided to break the objective function into 2 separate functions and combine them later. The first function was $\sum_{i=1}^N \sum_{j=1}^N w_i w_j \sigma_{ij}$ which calculates the risk of the portfolio and the second function was $\sum_{i=1}^N w_i \mu_i$ which calculates the expected return. I implement these using for loops instead of sums and stored each of the functions as separate variables. It was then simple to combine them into one function and add the weighting parameter to get the required objective function.

The third stage was to add the cardinality constraint and solve the produced QUBO. I used Quboverts built in constraint functions to add the cardinality constraint so that only K variables would be included. Once the model was finished it could be converted to a QUBO using Quboverts methods and solved by simulated annealing. To find an efficient frontier the problem needed to be solved multiple times using different weighting parameter. This was implemented using a for loop where the weighting parameter would increase in increments of 0.1. To change the weighting parameter in the model, it had to be recomputed after each iteration with the updated weighting parameter.

The prototype worked with relative success. Using the 4 asset sample (shown in fig. 5) for the paper by T. Chang [9], the prototype was able to assert which assets should be included in the optimal portfolio for different weighting parameter when the cardinality constraint was 2. This test case was well documented in the paper and offers a good way to determine whether the algorithm was successful.

Fig. 5. Four asset example [9]

Asset	Return (weekly)	Standard deviation	Correlation matrix			
			1	2	3	4
1	0.004798	0.046351	1	0.118368	0.143822	0.252213
2	0.000659	0.030586		1	0.164589	0.099763
3	0.003174	0.030474			1	0.083122
4	0.001377	0.035770				1

By implementing a prototype I was able to see which methods could be used to implement the final system. The concept of splitting the objective function will most likely continue into the final project. The use of Quboverts constraint functions did feel that they might hinder the development of the final system, as the constraints will probably be quite complex, so I will probably implement the constraint functions manually. In a meeting with Fujitsu about the prototype they didn't seem very familiar with Quboverts but did know of PyQubo. This is another reason which lead to choosing PyQubo over Quboverts as it would be easier to explain how my system was working to the representatives from Fujitsu.

B. QUBO Formulation

TO start developing the QUBO formulation module it was important to figure out where to start. While the system developed in the prototype resembles the CCPO problem, in order to model the variable weights correctly most of the functions become significantly more complicated. This is due to the difference in how multiple binary variables will interact in the objective function. As discussed in the literature review the QUBO model must be quadratic. This means that multiplication can only occur between 2 variables at a time. In the prototype it was possible to implement the objective function first and then the constraints. Unfortunately it isn't possible to do this with the main system as constraints may need to be hard coded into the encoding scheme. Due to this I decided it would be best to firstly implement the integer to binary encoding so that I could then figure out how the encoded variables would be used in the objective function.

To encode the integer variables as binary variables a list of binary variables had to be generated. To calculate the number of binary variables required it is first important to discuss how the constraints on the weights of the assets is implemented. The asset value constraints denoted by ε_i and δ_i are added to the system by hard coding the possible weights an asset can take into the system. Since the binary encoding scheme described earlier does not allow the integer variable being encoded to take a negative value a constraint on the lower bound can be implemented by adding the value of the lower bound to the encoded equation. To constrain the upper bound the difference between the upper and lower bound must first be calculated. Then when the integer variable is being encoded the upper bound is replaced with the difference. Since the lower bound is added to the final equation this will allow the value of the final equation to reach the upper bound but never exceed it. Since the value for the upper bound is replaced with the difference for encoding only the number of variables needed relies on the difference instead of the upper bound. The equation to calculate the number of variables needed is given below.

$$D = \lceil \log_2(\text{diff}) \rceil \quad (12)$$

Where D is the number of variables and 'diff' is the difference between the upper and lower bound. This tells us how many binary variables are needed to represent each assets possible weights. To implement the cardinality constraint an auxiliary variable is also needed for each of the assets, to represent whether it has been selected or not. This makes the total number of binary variables required for the problem $D + N$.

The `encode_variables()` function takes in that list of binary variables it also takes the lower bound, difference between lower and upper bound, the total number of assets and the number of variables needed per asset. In discussions with the representatives from Fujitsu the following mathematical encoding scheme for the cardinality constrained problem was put forward fig. 6. This encoding scheme incorporates the inequality constraints of the assets into the encoded variables. The `encode_variables()` function attempts to implement this encoding scheme and return an array of asset weights modeled using binary variables.

The `initialise_variables()` function firstly produces the original list of binary variables using the method described above. It then creates a separate list of just the auxiliary variables which just requires taking the N variables from the end of the original list. Where N is the total number of assets. After that it calls the `encode_variables()` function passing it the relevant parameters and getting back the array of encoded asset weights. The `initialise_variables()` function then returns the array of encoded asset weights and the array of auxiliary variables.

Now that the asset weights have been encoded the objective function for the CCPO problem can start to be developed. As with the prototype I decided to separate the objective function into 2 parts. The `create_hamiltonian_1()` function

Fig. 6. Binary encoding scheme

A. Binary encoding

$$w'_i = \sum_{r \in V} v_r b_{i,r} + \epsilon'_i z_i \quad (8)$$

with $m = \lceil \log_2(\delta' - \epsilon') \rceil$ the number of binary variables $b_{i,r}$ required per integer w'_i and $V = [1, 2, 4, \dots, 2^{m-1}, \delta' - \epsilon' - (2^{m-1} - 1)]$. Since we will need an extra binary variable z_i to express when an asset is selected or not, we encode the minimum proportion ϵ'_i there. We thus only need to cover values from 0 to $C = \delta' - \epsilon'$ with each $b_{i,r}$.

implements the first part of the objective function. This function takes in the array of encoded asset weights, the total number of assets, the weighting parameter and the covariance matrix. The array of encoded asset weights was designed in a way that the equation relating to each assets weight could be treated as a single variable in the array. This allows the implementation to be very similar to the prototype. The main difference is that the weighting parameter is added within the function and then the first Hamiltonian is returned. The `create_hamiltonian_2()` function is very similar to the previous one except it takes in the array of mean returns instead of the covariance matrix. It then implements the second part of the objective function, applies 1- the weighting parameter and returns the second Hamiltonian.

The `create_cc_model()` function takes in the encoded asset weights, the list of auxiliary variables, the first and second Hamiltonian and the total number of assets that should be included in the final portfolio. This function is used to finalise the objective function and compile it to the PyQUBO model. It calls the `add_constraints()` function to add the necessary constraints to the final objective function. Finally it returns the model.

The `add_constraints()` function takes in the encoded asset weights, the auxiliary variables, the current final objective function and the total number of assets to be held in the final portfolio. This function sets the penalty values. These values are hard coded in for the time being. Parameter tuning will be undertaken to make sure their values apply significant penalties if the constraints are violated. The required constraints are then produced using the functions described in the following paragraphs and added to the final objective function. The final objective function is then returned.

The `total_investment_constraint()` function is implemented with the goal of making sure that 100 percent of the investment is invested. It takes the encoded asset weights and a penalty value as parameters. It then implements the following equation to create the total investment constraint.

$$C = p \times (1 - \sum_{i=0}^N w_i)^2 \quad (13)$$

Where C is the constraint equation and p is the penalty value. To implement this the python sum function is used to add all the encoded weights. This `total_investment_constraint()` function returns the constraint equation produced.

In discussions with Fujitsu known methods for implementing the cardinality constraint were discussed. Below is the proposed method for implementing the cardinality constraint mathematically. The `chosen_constraint()` function implements equation 20 in fig. 7. What should be noted about this constraint is that if the lower bound is greater than 0 a selected asset will still have the lower bound added if none of the assets encoded variables are 1. This is important for calculating the weights of assets when the QUBO has been solved. The `chosen_constraint()` function takes in the encoded asset weights, the auxiliary variables and the penalty value. It uses a for loop to iterate through all the assets but handles the second summation slightly differently to the equation. Instead of iterating through

Fig. 7. Cardinality constraint

For all other encodings than onehot encoding, the cardinality constraint is expressed by the two following Hamiltonians:

$$H_{chosen} = \sum_{i=1}^N \left(\sum_{r \in V} b_{i,r} (1 - z_i) \right) \quad (20)$$

$$H_K = \left(\sum_{i=1}^N z_i - K \right)^2 \quad (21)$$

Equation (20) ensures that for any binary variable $b_{i,r}$ from the corresponding w'_i to be chosen, its corresponding z_i has to be chosen or else H_{chosen} will be positive and thus constraint violated.

each individual binary variable, the total weight equation for each asset is multiplied by $(1 - z_i)$ which should have the same affect. The constraint equation is then returned.

The `cardinality_constraint()` function implements equation 21 in fig. 7. It takes in the auxiliary variables, a penalty value and the total number of variables to be held in the final portfolio. It uses the `sum()` function which is a standard python function to add all the auxiliary variables together and implement the constraint. It then returns the constraint equation.

The `calc_variables()` function takes the total assets in the sample and the upper and lower bounds for asset weighting. It then calculates and returns the difference between upper and lower bounds, the number of binary variables required to model each asset and the total number of binary variables needed in the system.

Finally the `cc_qubo()` function is the main function of the QUBO formulation model. It takes in the total number of assets to be held in the final portfolio, the covariance matrix, the array of mean returns, the weighting parameter and the lower and upper bounds for asset weighting. The function combines the functionality of the other functions mentioned in order to formulate a QUBO based on the inputs provided. After calling all the required functions and producing a model for the CCPO problem, the model then needs to be converted to a Binary Quadratic Model (BQM). As mentioned previously this is a very similar model to the QUBO but the Dimod package from D-Wave provides functionality to convert a BQM into a NumPy array, which is the format of the QUBO we want to return. I would like to note here that D-Wave have now classed the function for this conversion as deprecated. There doesn't appear to be a good reason for this and even after emailing D-Wave directly I haven't gotten a reasonable answer. It is still in the current version of the Dimod package however and will be used in this project.

C. Simulated Annealing

THE simulated annealing module comprised of two main parts, the algorithm itself and plotting how the algorithm performed. The `sampler()` function combined these two parts and handled rerunning the SA algorithm and graphing the process depending on how many times were input. If only one run is done then 4 detailed graphs describing how all the values changed in relation to iteration or temperature are produced. These are incredibly useful for seeing how changing variables such as initial temperature affect the annealing schedule. If multiple runs are done then 1 graph showing how the result value changes per run will be shown. There is also the option not to show any graphs. This function returns a list of dictionaries, a new dictionary is added for each run and stores the number of the run, the state of variables for the final result of the run and the final value of the run.

The `sim_anneal()` function handled the implementation of the algorithm which closely followed the psuedocode in the design section. Initially the number of iterations for the algorithm would have to be input but upon running a number of trials it became clear that the algorithm would converge when the temperature got significantly low (e.g. 0.001). This lead to the number of iterations being calculated by rearranging equation 11 for i given T' . The

`neighbour()` and `new_temp()` functions are used by `sim_anneal()` and implement the methods described previously for finding a neighbour and setting a new temperature. The `objective_value()` function uses the QUBO matrix and the set of binary variables representing the current solution to calculate the current value of the objective function. Another benefit of using the QUBO model is that to calculate the value of the objective function is incredibly quick and only requires basic matrix multiplication.

The functions `plot_value_change()`, `plot_temp_change()`, `plot_prob_change()` and `plot_result_change()` all take in a list of the values they need to plot and plot them on a graph with respect to the number of iterations. The `plot_value_change_t()` function takes in the list of value and the list of temperatures and plots a graph showing how the graph changes with respect to temperature.

D. Correlation to Covariance

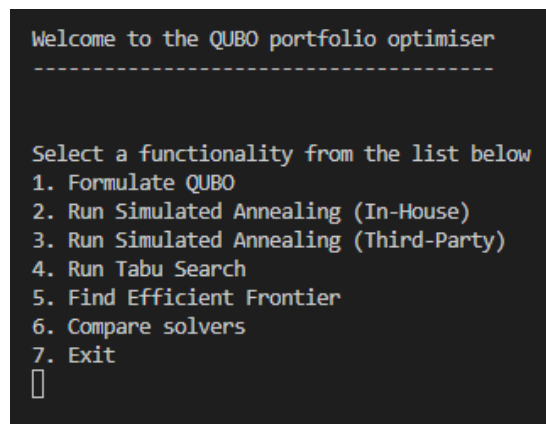
WHILE testing different problems and using different data sets it became clear that a lot of the data sets used correlation matrices and standard deviation matrices. This required often conversions from a correlation matrix to a covariance matrix so I decided to implement a small module which would provide this functionality. The `convert()` function takes in an array of correlations and an array of standard deviations and then uses them to produce a covariance array. The function returns the covariance array.

I also decided to try implementing in depth error handling in this module. It took a reasonable amount of time even though the module is very small in the scope of this project did not seem worthwhile. If the modules were reused this might be helpful but I decided it was more import to develop the system further rather than handle every error each module might throw.

E. Portfolio Optimiser

THE portfolio optimiser module combines the functionality of all the other modules and provides a user interface via the command line. The `main()` function prints the main menu and uses python's pattern matching to select which functions to use based on the user input. The main menu will reappear after each feature has ended until the user exits. Below is a screen shot showing the design and options of the main menu.

Fig. 8. Portfolio Optimiser Menu



```

Welcome to the QUBO portfolio optimiser
-----

Select a functionality from the list below
1. Formulate QUBO
2. Run Simulated Annealing (In-House)
3. Run Simulated Annealing (Third-Party)
4. Run Tabu Search
5. Find Efficient Frontier
6. Compare solvers
7. Exit
  
```

The `get_qubo_parameters()` function interfaces with the user to gather data in order to formulate a QUBO. This function is used for functionality where a new QUBO needs to be formulated such as formulating a new QUBO or finding the efficient frontier. This function also uses the `convert()` function from the correlation to covariance module to allow the user to input a correlation matrix to produce a QUBO. The `find_qubo()` function extends the functionality of the `get_qubo_parameters()` function by requesting the user to input the weighting parameter and then using the `cc_qubo()` function from the qubo formulation module to formulate the QUBO. The QUBO is then output to the console and the user is asked if they want to. The `savetxt()` function from NumPy is used to save the QUBO array, where each row is separated by a newline and each column is separated by a comma.

The `input_qubo()` function gets a previously formulated QUBO from the user. The QUBO doesn't have to have been formulated by this system but it must follow the same formatting used when saving a QUBO, as detailed above.

The `inhouse_sa()` function handles the functionality around the simulated annealing module. It takes the QUBO and then gets the initial temperature, number of reads, cool-down ratio and whether to graph the process from the user. It then inputs these values into the `sampler()` function from the simulated annealing module and stores the output. The time it takes the `sampler()` function to return a result is recorded. The best result from the output of the `sampler()` function is then returned along with its corresponding variables and the speed the algorithm ran.

The `thirdparty_sa()` function takes in the QUBO matrix as a parameter and requests the number of reads from the user. It then passes the QUBO matrix and number of reads to the `thirdparty_sa_solver()` function and receives the state of the binary variables for the best result and the speed at which the algorithm ran. It then calculates the value of the objective function using the `objective_value()` function and the state of the binary variables for the best result. The `thirdparty_sa()` function returns the binary variables for the best result, the value of the best result and the speed the algorithm ran.

The `thirdparty_sa_solver()` function uses the Neal module from D-Wave which provides simulated annealing for the BQM model. This means the QUBO matrix must be converted to a BQM before it's sent to the sampler. The results received back from the `sample()` method are a `sampleset` object defined in Dimod which stores, for each run, the number of the run, the state of the binary variables for the result of that run and the value of the result. The `sampleset` object stores runs in order of which run had the lowest value. This makes getting the best run simple as it's just the first run in the sample set. The time the `sample()` function took to run is also calculated. The `thirdparty_sa_solver()` then returns the state of the binary variables for the best result and the speed of the algorithm.

The `tabu_solve()` function works similarly to the `thirdparty_sa()` and `thirdparty_sa_solver()` functions. It takes in the QUBO matrix as a parameter and retrieves the number of reads through user input. The Tabu module also developed by D-Wave provides a `sampler()` function for the tabu search algorithm which also takes in a BQM and the number of reads. The `sampleset` object retrieved from the `sampler()` function is the same as the Neal `sampleset` so is handled in the same way. The speed the algorithm ran is calculated and the value of the best result is found. The `tabu_solve()` function returns the result variables, the value of the result and the speed of the algorithm.

To make sure that the simulated annealing algorithm developed for the project worked sufficiently well the `solver_comparison()` function was implemented to run all 3 solvers described above and output the value of their results and their speed. Being able to see these values side by side will help to determine how well the simulated annealing algorithm built for the project worked.

The `print_results()` function is used to print the solution to a QUBO that has been input. To print the results extra information is needed on top of the original QUBO such as the total number of assets in the original model, the upper bound of the original model and the lower bound of the original model. This function handles retrieving that data from the user and calling the `find_results()` function which prints out the results. This function also gives the user the option to save the results. This is also done using the same `find_results()` function but the standard output stream is changed to the user defined file so the print statements in `find_results()` will output to the file instead of the console. The standard output stream is then returned to the original output.

The `find_results()` function takes in all necessary variables passed to it by the `print_results()` function and prints necessary information about the results. This information includes:

- 1) The state of the variables for the solution.
- 2) The value of the solution.
- 3) The variables for each asset.
- 4) The weighting of each asset as a percent.
- 5) Whether each asset was selected.

The weighting for each asset is calculated in a very similar method to how the original weight equations are calculated in the QUBO formulation module. However in this function the asset numbers also need to be printed out. This leads to an extra variable being added to keep track of which auxiliary variable the current asset is constrained by.

One of the requirements for this project was to plot the data of the optimal portfolios for a particular optimisation problem on a graph. This graph as discussed is known as the efficient frontier and in relation to the CCPO problem the portfolios on the frontier can be found by changing the value of the weighting parameter. This unfortunately leads to a new QUBO needing to be calculated for every different weighting value. All of this functionality is covered in the `fund_efficient_frontier()` function. The function takes in the necessary parameter to formulate a QUBO without the weighting parameter. It then gets the number of portfolios to find on the efficient frontier and how many times the SA algorithm should run for each portfolio. The values of the weighting parameter are then stored in an array, where the total number of values is equal to the number of portfolios to calculate and the values are evenly spaced between 0 and 1. The values are then iterated through, for each iteration the QUBO matrix for that weighting value is formulated, the QUBO is solved using the `thirdparty_sa_solver()` function and the risk and return are calculated for the best portfolio found. The user is kept continuously updated on how many portfolios have been found out of the total number to find. The reason the `thirdparty_sa_solver()` function was used is that when tested it performed faster than the other two solvers and found marginally better solutions. This meant it was the clear option to use for finding the efficient frontier. Once all the portfolios are found the user can then select to plot the efficient frontier, the graph has the risk on the x axis and the return on the y axis which is standard practice for plotting efficient frontiers. The user can then choose to save the results, which are stored as pairs of risk and return. Storing them in this way makes it easy to graph again if required.

The value of the objective function also needs to be calculated in the module frequently so the `objective_value()` function from the simulated annealing module was also implemented in this module. While the principle of DRY (don't repeat yourself) was followed as closely as possible in the implementation of this project, the decision was made to implement this function in 2 separate modules to keep decoupling. In this module the `objective_value()` function is used with the results from the third party simulated annealing and Tabu search algorithms. If either of these modules were to change how their results are formatted then the `objective_value()` function in this module could be adapted without affecting how it worked in the simulated annealing module.

The `risk_return()` function takes in the state of the binary variables for the result, the covariance matrix, the mean returns, the lower bound of asset weights, the difference between asset weight bounds and the number of binary variables needed per asset. It uses this data to firstly calculate what the decimal weights of each asset are. It can then use these weights to calculate the risk and return of the final portfolio. The following formula is implemented to calculate the portfolios risk using a matrix of asset weights (w) and the covariance matrix (cov):

$$\sigma = \sqrt{w \cdot cov \cdot w^T} \quad (14)$$

Where σ is the risk of the portfolio and w^T is the transpose of w . The portfolios expected return is calculated using the following, w and the mean returns matrix (ret):

$$R = w \cdot ret \quad (15)$$

Where R is the expected return of the portfolio.

As a lot of this module takes input from the user robustness and error handling were very important. The module has been implemented so that as many possible exception cases could be handled as possible. This means that every input should have the appropriate pattern matching and error handling associated with it. If a users input causes problems then an appropriate warning message is printed and the user is asked to reenter a valid input. For the sake of yes/no inputs (Y/N) 'Y' will be taken only if the input is 'Y' any other input will result in an 'N'. Most of the error handling was input dependent so a standard validation function wasn't possible. A few inputs did follow the pattern that they must be an integer value above 0 so the `validate_int_above_zero()` function was implemented to handle these input cases.

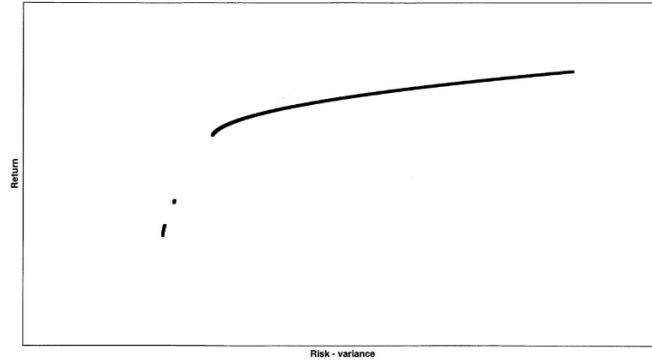
V. TESTING

A. Testing the QUBO formulation

THE best way to test how the QUBO formulation was working was to plot the efficient frontier. If efficient frontiers could be successfully plotted then this would indicate QUBOs with a range of weighting parameter values could be successfully formulated for a given problem. To determine whether an graph resembles the efficient frontier without knowing what the efficient frontier looks like is quite difficult. This becomes even harder for the CCPO problem as the efficient frontier is discontinuous so just because the graph isn't smooth doesn't mean it's not the efficient frontier.

For the majority of testing of the QUBO formulation system the sample data set shown in fig. 5 was used. This is because there is a small number of assets so computing time is relatively low. This means multiple tests can be run in quick succession and problems can be detected faster. The other reason this sample set is used is because the paper by T. Chang[9] also provides a set of graphs detailing the efficient frontier for different problems using this sample set. The visible efficient frontier given in the paper is shown below.

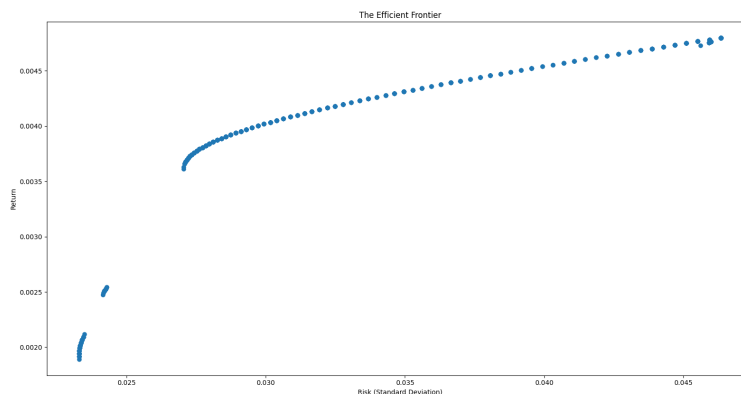
Fig. 9. Cardinality constrained efficient frontier - visible portions. [9]



This frontier was arrived at with the lower bound set to 0, the upper bound set to 1 and a cardinality constraint of 2.

Using the data set in my system, the aim was to reproduce the visible portions of the cardinality constrained efficient frontier as closely as possible. By tuning the penalty values of the QUBO formulation eventually the following values were converged upon. A penalty value of 30 was selected for the `total_investment_constraint()` function and a penalty value of 90 was selected for the `cardinality_constraint()` function and `chosen_constraint()` function. The number of portfolios to find was set to 1000 and the number of reads for each portfolio was set to 500. Below is the graph of the efficient frontier produced.

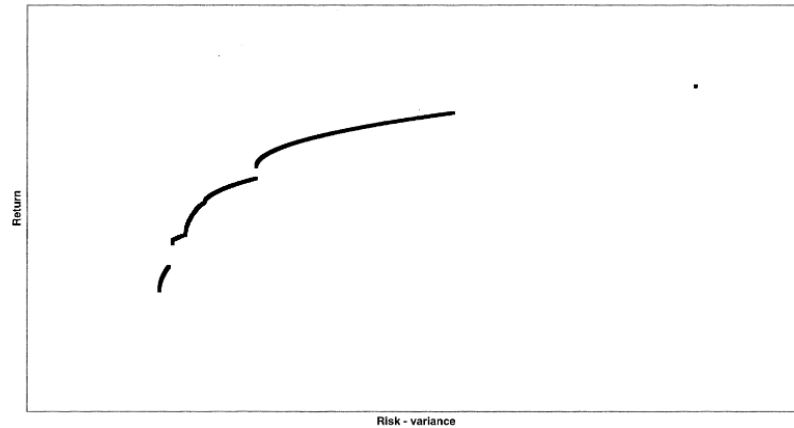
Fig. 10. Produced cardinality constrained efficient frontier



While the proportions of the graph are slightly different it is clear to see that the frontiers match up almost perfectly. All sections expected to be found using a heuristic approach have been found. This shows that the QUBO formulation was a success and all the constraints were correctly imposed on the model.

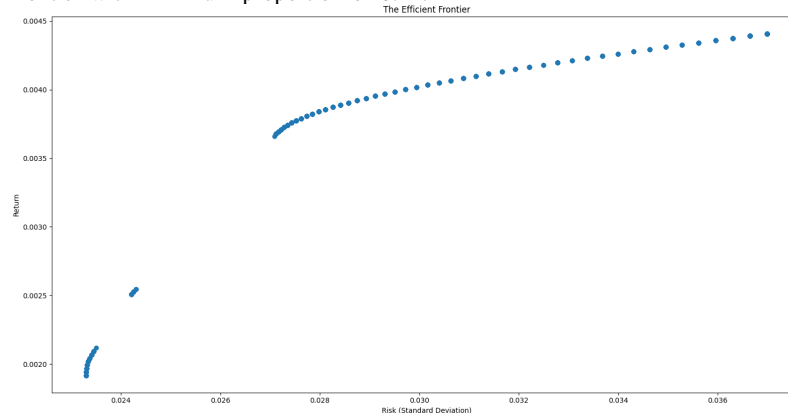
Using the same set up but changing the lower bound to 0.24 the paper by T. Chang [9] produced found the following efficient frontier. This frontier does include the invisible portions to heuristic solvers.

Fig. 11. Efficient frontier with minimum proportion of 0.24. [9]



The frontier produced by my system is shown below.

Fig. 12. Produced efficient frontier with minimum proportion of 0.24.



While this graph may not appear different to the one above the scales at the bottom show that the top tail is missing. This is again exactly what we expect to see based on the graphs produced in the paper. This shows that the formulation of the CCPO problem with minimum proportion constraints was a success.

B. Testing simulated annealing

THE other main functionality to test in the project is the simulated annealing algorithm. We know for the above testing that the QUBOs being formulated can be used to find an optimal portfolio. Since the generation of the efficient frontier uses the simulated annealing functions from Dimod we also know that they are able to find valid portfolios when 500 reruns are done. The more reruns done just increases the probability that the solution found is as close to optimal as possible.

The test below shows the results of finding a QUBO on the previous frontier where the weighting parameter is equal to 0.5. For the developed simulated annealer the initial temperature was set to 1000, the cooldown ratio was

set to 0.9 and the number of reads was set to 500. The number of reads for the other two externally developed algorithms was also set to 500. The initial temperature and cooldown ratio were arrived at by running numerous runs of the algorithm, plotting the results and seeing which values worked best. The data to take from this test is the speed at which the algorithms perform and the value of the solution found. This results from this test show that

Fig. 13. Solver comparison

```
Results
-----

In-house result was: -390.001564963855, in-house speed was: 0.8841424999991432

Third-party result was: -390.00156521261994, third-party speed was: 0.26902740000514314

Tabu result was: -390.00156521262, Tabu speed was: 10.580309000011766
Press enter to continue
```

the in-house simulated annealer found a solution which was effectively just as good as the Dimod SA algorithm and Tabu solver. The in-house SA algorithm was only 0.6151 seconds behind the Dimod implementation so while not quite as fast it was still a useable speed. The Tabu solver was significantly slower than the other solvers. This was a common theme over multiple runs of the experiment and made it clear that the Tabu solver was not viable for implementation when speed was required. The solution from the Tabu solver was also not much different from the other two so there did not seem to be any advantage in using this method.

VI. DESCRIPTION

THE QUBO model is becoming more widely adopted due to its new found ability to model lots of different combinatorial optimisation problems. Fujitsu research uses the QUBO model for their digital annealing system so they are trying to form as many problems as possible as QUBOs. Fujitsu also encounters the CCPO problem often so being able to show it is possible to formulate it as a QUBO makes it both easier for them to solve their CCPO problems and add further functionality to their digital annealing system.

The final project is able to formulate QUBOs for the CCPO problem, solve these QUBOs and graph the solution process. It is also able to find the efficient frontier for a given CCPO problem. The main focus of this project was to prove that it was possible to use the QUBO model to successfully represent the CCPO problem. The methods for doing this have been described in depth throughout this paper and the results show that it is entirely possible to formulate QUBOs which represent the CCPO problem.

VII. EVALUATION

THE following section will cover how the original set of specifications proposed at the start of the paper have been successfully implemented throughout the project. It will then discuss what the key points taken away from the project are. Finally whether the project can be classed as successful will be considered.

The project has been developed focusing around the implementation of the CCPO problem so the first functional requirement can be considered a success. The system has the capability of converting multiple different formats of real stock market data into a use able format for the system. The tests conducted above also used real stock market data and other tests have been carried out with larger data sets of up to 31 assets which were also successful. The second functional requirement can therefore also be classed as successful.

The system is able to formulate a QUBO for the CCPO problem based on the co-variance matrix and expected return. This QUBO also contains the constraints required for the CCPO problem. The QUBO produces can then be solved and the integer weights of each asset can be determined. So the third functional requirement can also be classed as a success. The integer weights of the assets were successfully encoded using the binary encoding scheme and the model was successfully exported as the Q matrix which means all functional requirements relating to the QUBO formulation were successful.

The simulated annealing algorithm was implemented and has shown to be successful in finding optimal portfolios given a QUBO formulated by the system. The `sample()` function of the simulated annealing module takes in the Q matrix and the total number of variables for the system and outputs a solution to the problem. While the algorithm itself doesn't validate matrix size and number of variables, the portfolio optimiser module validates all inputs to the simulated annealing function which for the scope of this project covers this functionality. All functional requirements relating to the implementation of the simulated annealing algorithm can therefore be classed as successful. The portfolio optimiser module provides the functionality to graph the efficient frontier to that functional requirement can also be classed as successful.

Many different combinations of assets were tested on the system and ran with success. The entire system and simulated annealing algorithm were tested with published data. In the testing section the published data shows that the solutions found by the system were optimal. It is also shown that the simulated annealing algorithm worked correctly in a relatively small amount of time. The set of 4 assets provided a large enough problem to demonstrate the system and the CCPO problem while also keeping the computational time relatively low so that good quality results could be found. The non functional requirements for the system can therefore also all be classed as successful.

The key points to take away from the project are that the CCPO problem can be formulated as a QUBO where the assets have integer weights. At the time of producing the literature review very little research appeared to have been done in using integer to binary encoding to model a portfolio optimisation problem as a QUBO where the asset weights remained integer percents, instead of assets weights being an option to only choose that asset. Based on the success against the original specification and the fact the project was able to implement systems with very little research surrounding them I would class it as a success.

VIII. CRITICAL ASSESSMENT OF THE WHOLE PROJECT

THE development of the project was not without any issues and some areas couldn't be tested as fully to make sure that the core concepts of the project were successful. Areas that could be improved in the project mainly relate to this issue. The robustness of each module could be further improved, especially if they were to be used again separate from the portfolio optimiser module. Unit tests could also have been more thoroughly implemented to make sure modules continued to behave as expected.

If these system was to be used within a commercial setting or to formulate very large QUBOs it could be rewritten in C. C is closer to machine code and would offer faster formulation times and cleaner memory handling. It doesn't provide the same support as python however and it would take considerably longer to implement an equivalent system in the C language. This project fulfills its purpose well though and proves a very important point about whether it was actually possible to do.

IX. CONCLUSION

IN conclusion this project has been a dive into a relatively unexplored section of optimisation. It has combined different concepts which are at the cutting edge of research in this field and hopefully gives an insight into how they can be used practically. The system developed in this project will help to show what can be done with the CCPO problem and how it can be formulated as a QUBO. This should help to provide evidence as to why this field is an important area of study.

REFERENCES

- [1] H. Markowitz, "Portfolio selection," *The Journal of Finance*, vol. 7, no. 1, pp. 77–91, 1952.
- [2] H. Markowitz, *Portfolio selection : efficient diversification of investments*. New Haven London: Yale University Press, 1959.
- [3] V. Yankov, "In search of a risk-free asset," *n/a*, 2014.
- [4] W. F. Sharpe, "Mutual fund performance," *The Journal of business*, vol. 39, no. 1, pp. 119–138, 1966.
- [5] W. F. Sharpe, "Adjusting for risk in portfolio performance measurement," *The Journal of Portfolio Management*, vol. 1, no. 2, pp. 29–34, 1975.
- [6] W. F. Sharpe, "The sharpe ratio," *The Journal of Portfolio Management*, vol. 21, no. 1, pp. 49–58, 1994.
- [7] G. Kochenberger, J.-K. Hao, F. Glover, M. Lewis, Z. Lü, H. Wang, and Y. Wang, "The unconstrained binary quadratic programming problem: a survey," *Journal of combinatorial optimization*, vol. 28, no. 1, pp. 58–81, 2014.
- [8] D. Venturelli and A. Kondratyev, "Reverse quantum annealing approach to portfolio optimization problems," *Quantum Machine Intelligence*, vol. 1, no. 1, pp. 17–30, 2019.
- [9] T.-J. Chang, N. Meade, J. E. Beasley, and Y. M. Sharaiha, "Heuristics for cardinality constrained portfolio optimisation," *Computers & Operations Research*, vol. 27, no. 13, pp. 1271–1302, 2000.
- [10] J. T. Iosue, "Qubover." [Online]. Available: <https://pypi.org/project/qubover/>
- [11] M. Zaman, K. Tanahashi, and S. Tanaka, "Pyqubo: Python library for qubo creation," *IEEE Transactions on Computers*, 2021.
- [12] K. Tanahashi, S. Takayanagi, T. Motohashi, and S. Tanaka, "Application of ising machines and a software development for ising machines," *Journal of the Physical Society of Japan*, vol. 88, no. 6, p. 061010, 2019.
- [13] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. Fernández del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant, "Array programming with NumPy," *Nature*, vol. 585, p. 357–362, 2020.
- [14] A. Condello, "Dimod." [Online]. Available: <https://pypi.org/project/dimod/>
- [15] S. Kirkpatrick, C. D. Gelatt Jr, and M. P. Vecchi, "Optimization by simulated annealing," *science*, vol. 220, no. 4598, pp. 671–680, 1983.
- [16] H. Szu and R. Hartley, "Fast simulated annealing," *Physics letters A*, vol. 122, no. 3-4, pp. 157–162, 1987.
- [17] L. Ingber *et al.*, "Adaptive simulated annealing (asa)," *Global optimization C-code*, Caltech Alumni Association, Pasadena, CA, 1993.
- [18] K. Katayama and H. Narihisa, "Performance of simulated annealing-based heuristic for the unconstrained binary quadratic programming problem," *European Journal of Operational Research*, vol. 134, no. 1, pp. 103–119, 2001.
- [19] J. F. D. Martín and J. M. R. Sierra, "A comparison of cooling schedules for simulated annealing," in *Encyclopedia of Artificial Intelligence*. IGI Global, 2009, pp. 344–352.