CryptoGraph –Software Design Document

Contents

Introduction

Cryptograph allows the user to explore cryptocurrency assets and trades in an automated way. Asset and trade information is stored in such a way that both direct and indirect trade paths between assets can be identified. The cost of these paths can also be calculated, including commissions, and sorted in a way that exposes the most efficient trade paths between assets. While asset and trade information can be extracted from .json files in a specific format (based on those that are returned from Binance API requests), it is also possible to get the up to date trade information for each trade via a call to the Binance API at the time of using the program.

The program is implemented in object oriented Python code. The program makes use of data structures implemented without the use of any Python built-ins. Linked lists are heavily used, with specially implemented methods for additional functionality such as sorting. A graph data structure has been extended for this program to make the class CryptoGraph. An instance of this class holds all assets as vertices, and the edges store attributes describing the trades, such as price.

Data is required to run the program. Three files were specified by the client, which are returned from the Binance API: exchangeInfo, 24hr, and trades?symbol=ETHBTC.

In my exploration of the available data, I found that:

- exchangeInfo.json contains useful information about potential trades. Each trade between two assets is called a symbol. This file contains information as to whether trading is allowed, and contains field that specify the baseAsset and quoteAsset.
- But exchangeInfo.json does not contain the prices of actual trades, or the volumes. We would like to know this information for our weighted edge graph
- the 24hr.json file does contain the prices and volumes of trades over the last 24hrs, including the useful AvgWeightedPrice field.
- none of these files contained asset information directly, except for the field toAsset and fromAsset in the exchangeInfo.json file.

As such it was decided to use the file exchangeInfo.json to extract the asset names, and the allowable trades between them. 24hr.json was used to give the edge weight to the graph. However, a problem was encountered in that not all allowable trades had been traded in the last 24 hours. In these cases, a trade would have the price of 0.00 for the averageWeightedPrice field in the 24hr.json file, and as such the edge weight for that trade would be recorded at 0.00. The solution to this problem will be discussed later in this report.

TO understand the following sections it is important to have a basic understanding of the structure of the program. There are two main objects:

The Binance Data object – this is constructed from the specified data files, and has methods to output asset, details, trade details, and to build a graph for deeper analysis.

The CryptoGraph object – this is constructed from the Binance Data object, and this object is used for the more complex analyses, such as trade path analysis.

Installation

The program requires Python 3 to be installed, and has been tested with version 3.8.

Python libraries required are json, requests, sys, os, and pickle but all should be included in a vanilla install of Python 3.

The program can be run from the terminal, in two ways, by moving to the program directory, and typing, for interactive mode:

```
python cryptoGraph.py -i
```

Or for report mode:

```
python cryptoGraph.py -r <exchange info file> <24hr trades file>
```

Please note, on Linux machines you may need to use python3 instead of python.

The program has been extended beyond the initial client requirements to allow up to date information to be retrieved from the Binance API. However, as this was not requested, the program is also fully functional offline, and handles all Connection Errors that may be raised. It is provided with example data files that should be used in an offline situation. These are json files that are the result of GET requests to the Binance API. These files are:

1) 24hr.json, from https://www.binance.com/api/v3/ticker/24hr
2) exchangeInfo.json, from https://www.binance.com/api/v3/exchangeInfo
3) price.json, from https://api.binance.com/api/v3/ticker/price

Terminology/Abbreviations

Terms:

Asset: A cryptocurrency, specified by a 3-5 letter code.

Exchange Info json file: this should be of the format returned by a GET request to the Binance API, using the URL: https://www.binance.com/api/v3/exchangeInfo

24hr json file: this should be of the format returned by a GET request to the Binance API, using the URL: https://www.binance.com/api/v3/ticker/24hr

Function and Static Method - these terms are considered interchangeable in this report.

Walkthrough

Demonstrate all functionality, indicating anything that isn't implemented or working properly.

Report Mode

When run in report mode, using the command

```
python cryptoGraph.py -r <exchange info file> <24hr trades file>
```

the program will build a graph in memory from the user specified data files. It will then run a predetermined set of methods on the data and return useful information. The sequence of events is as follow:

1. program creates a BinanceTradingData object from the two file paths.
2. program creates the skeleton of a graph from the BinanceTradingData object, using the fromAsset and toAsset fields of the exchangeInfo file to add edges. Edges are only added if trading between these two assets is allowed. A list of vertices is also maintained in the graph, which are added as part of creating a new edge. This object is called validTrades in the program.
3. The graph is then populated with edge attributes, from the 24hr trades file.
4. The vertices of the graph are sorted by the number of outward edges they have, and the top 5 with the most 'outward allowable trades' are printed.
5. The edges of the graph are sorted by the volume24hr attribute. The top 5 trades by volume in the last 24hrs are then printed to the screen.
6. The edges of the graph are sorted by percentage price change over the last 24hrs. The top 5 trades by percentage price change in the last 24hrs are then printed to the screen.

This concludes the functionality of the report mode, and the program exits.

Interactive Mode

The program can be run in interactive mode using the command:

```
python cryptoGraph.py -i
```

When run in this mode, a menu is presented to the user that allows them to explore the data in a user choice driven way. There is a recommended order to the steps the user should follow, and while reminders are implemented in the code if a user makes a choice out of sequence, I will detail the recommended sequence of steps here:

```
INTERACTIVE MODE MENU
1. Load data from user specified files
2. Update price data to latest using Binance API
3. Find and display an asset
4. Find and display trade details
5. Find and display potential trade paths
6. Find the top 5 trade paths by cost
7. Set asset filter
8. Asset overview
9. Trade overview
10. Check for Possible Profitable Trade Paths...
11. Save data (serialised)
12. Reload graph from saved serialised graph object
13. Exit
Please make a user_choice:
```

Figure 1. Interactive Mode Menu

1) Firstly, the user should select menu option 1, to `Load data from user specified files`. The user will be asked to specify the path to an exchange info json file, and a 24hr json file. The format requirements are discussed in the Terminology/Abbreviations section of this report.

   The CryptoGraph program will then perform the following operations:

   a) create a BinanceTradingData object that holds the raw file information, and has methods to perform basic summaries of this raw data, and has a method to generate a basic graph from the data. (This object will be used as the basis for the reports initiated by steps 3 and 4 of the interactive menu.)

   b) The program will then create a basic graph from the BinanceTradingData object. Using the same methods as the report mode already discussed, the program uses the fromAsset and toAsset fields of the exchangeInfo file to add edges to the graph. Edges are only added if trading between these two assets is allowed. A list of vertices is also maintained in the graph, which are added as part of creating a new edge. This object is called validTrades in the program.

   c) The graph is then populated with edge attributes, from the 24hr trades file.

   d) The program returns to the main menu (NB: this step will not be raised again, suffice it to say that the main menu will be presented until the user enters '13' to exit the program.

2) Step 2 is optional – the use can choose to update the edge weights with up to date cost information for each trade. This step makes a GET request to the Binance API, using the URL: https://api.binance.com/api/v3/ticker/price . It then steps through each edge in the graph, and updates the price for that edge from the data returned from the Binance API call. Please note, that without doing this, some edges have a weight of zero, if they have not been traded in the last 24 hours. As such, it is recommended to always carry out this step. A price.json file has been included, to use without connection to the internet.

3) Find and display an asset. As very little asset information was provided in the files from the client, the output of this step is limited to a summary of the allowable trades for each asset. The program asks for an asset code from the user, and then gets the possible asset trades from the Binance Data object. Please note: this step requires step 1 to have been performed, but it only uses the Binance Data object, and not the graph object.

4) Find and display trade details: Asks the user for a 'from asset', and a 'to asset'. These are simply concatenated into a symbol code. The Binance trading data is searched for this symbol code, and the attributes displayed to the user. Once again, this step requires step 1 to have been performed.
   Step 4 of the program also optionally compares the 24hr weighted average price for the trade, with the up to date trade price loaded in step 2. If step 2 has not been performed, the program will requests the user to run the step if they would like to view this additional information.

THE NEXT STEPS RUN ON THE CRYPTOGRAPH OBJECT

5) The user is once again asked to specify a 'fromAsset' and a 'toAsset'. The program then performs a recursive depth first search on the cryptograph object, and finds all paths that start with the 'fromAsset' and end with the 'toAsset'. The program makes no distinction between direct paths and indirect paths, but it makes sure that vertices are not visited twice in a path search by marking the individual vertices as visited. This is important, as it is one of the reasons why it was chosen to implement a graph structure with both a vertices list, and an edges list.
   Once a path is found that traverses the graph from the 'toAsset' to the 'fromAsset', the program uses this list of assets to find edges, and calculates the total cost of the path by using the edge weights. It also includes commission for each trade step, and as such is a realistic estimate of the cost of a trade if performed through Binance. **NB: it does not take into account the discount for using the Binance Coin**. Trade paths are added to a container which is implemented as a sortable linked list, and returned to the main program for display to the user.
6) This step follows the same method as step 5, but in addition carries out a sort of the paths container linked list. The paths are sorted using a searchByAttribtue method that I have implemented in my sortable linked list data structure, and searched by the attribute 'cost'. The first 5 elements are obtained using a recursive function, and returned and printed to the user.
7) Set asset filter. This step asks the user for repeated input of asset codes. Upon input of each asset code, the program searches the graph for vertices with that name. If it is found, it is removed from the graph structure. Using this approach, that vertex will not be available as a step when finding paths through the graph, and as such is an effective way to implement an asset filter for those functions. This step operates on the CryptoGraph object, and as such does not affect the Binance Data object. As such, the asset exclusions performed by this step will have no affect of the reports output from steps 3 and 4 of the interactive menu.
8) Asset Overview. As there is not much asset information provided by the client, this step makes a summary of what we do have: the possible trades for each asset. Once again, the Sortable Linked List functionality is used: the graph vertices list is sorted by the number of edge links each vertex has, and the first of these assets is printed to the user. **This output will be influenced by the asset filter in step 7.**
9) Trade Overview: Makes heavy use of sortable linked lists. Sorts the CryptoGraph object's edge list by volume of trades in the last 24 hours, and displays top 5 to the user. Then sorts the edge list by percent price change in the last 24 hours and displays the top 5 to the user. **This output will be influenced by the asset filter in step 7**, as whenever a vertex is removed from the graph, the edge list is modified to reflect this change.
10) Check for possible profitable trade paths: **This step is an extension to the customer requirements**. This step will take a trade pair, and report to the user if the cheapest trade path for that pair is not the direct path. Using this feature, the user could identify trade paths where a profit could be made by trading between crypto currencies. At present, this step asks

the user for a specific trade pair to check, but the future vision for this feature would be to implement it in a loop that check all possible trades, and would report to the user if any potentially profitable indirect trade paths exist. This loop could be run at predefined time intervals, and the user notified when an opportunity presented itself. Whilst theoretically sound, in practise I did not identify any profitable trade paths: this is not to say that this rare situation could never occur, and if it did, large profits could be made.

11) Save the trade graph by serialisation. This step writes the CryptoGraph object to file by serialisation. It was necessary to increase the recursion limit for this step to work. There is no option to save the BinanceData object, as this object can be recreated by reading from the json files, and is not modified by the program. The CryptoGraph object **is** modified by the program using the Asset Filter functionality in step 7, and as such it may be useful to save it.
12) Reload graph from saved serialised graph object. Replaces the current graph object by the last saved object. Users can experiment with asset filters, and then reload the graph to a previous state. Multiple save points have not been implemented, to keep the program simple.
13) Exit. Exits the program using the sys.exit() method.

Class Descriptions

The program has two main code files, described here:

cryptoGraph.py:

In general, this file holds the main code for the program, and the key classes for the functionality specific to the program.

Classes:

- BinanceTradingData
- CryptoGraph
- TradePath

Static Methods:

- getCurrentSymbolPrice
- getAllSymbolPrices
- getFirstXElements
- _getFirstXElementsRec
- loadData
- serialize
- deserialize
- displayUsage
- runInteractiveMenu
- runReportMode
- main

dataStructures.py

In general, this module file holds modified version of the data structures implemented during the semester and submitted for the practicals. Whilst my intention was to have the original data structures in these files, and inherit from them and specify new classes in crytoGraph.py, this became impractical when I needed to modify how classes called by other classes were implemented. As such, it is an informal division, but still serves to organise the code in a logical way, and to minimise the detail included in the main crytoGraph.py to what is especially relevant for the running of the program.

Classes:

- DSAGraphVertex
- DSAGraphEdge
- DSAGraphWithEdges
- DSAListNode
- DSALinkedList
- DSALinkedListDE
- SortableList

As dataStructures.py contains parents classes that are inherited in cryptoGraph.py, I will start by explaining the classes in dataStructures.py:

dataStructures.DSAGraphVertex

This class is based on previously submitted work, my submission for Practical 06 – Graphs. Each instance of this class describes one vertex in the graph, and specifies its links.

Attributes are:

self._label - used to hold the main identifier for the vertex

self._value - used for a variety of purposes, as a container for the vertex data. Not used in our implementation.

self._edges - a SortableList to hold references to the other vertices that this vertex links to. **PLEASE NOTE in the CryptoGraph program this is not used.** Instead, we store the edge information in a list of edge objects that belongs to the graph class.

self.edgeCount - a counter attribute, that helps us to store the total number of vertices that this vertex links to. Used for sorting functions in the CryptoGraph program, and more efficient to increment when adding edges than to iterate through the lists of vertices just to count them when performing analysis.

self._visited - used when traversing the graph, to avoid loops.

Methods are:

setVisited() - a basic setter

clearVisited() - a basic setter

__repr__() - a standard repr method to return the label as the identifier for the object.

dataStructures.DSAGraphEdge

This class is specific to the CryptoGraph program, but it was more practical to include it in the dataStructures file because it is needed for the DSAGraphWithEdges class. Each instance of this class describes one edge in the graph. I chose to create this class, because it seemed most logical (and similar to life) to use edges to represent the trades between assets. I feel like this presented a logical way to conceptualise the problem in terms of real world objects – we certainly have more information for edges (trades) than for vertices (assets).

Attributes are:

self.fromVertex = a string with the asset code of the from vertex for this edge

self.toVertex = a string with the asset code of the to vertex for this edge

self.weight = this is a standard attribute for an edge implementation, I used it to hold the cost of the trade between to toAsset and the fromAsset.

self.volume24hr = a later addition, specific to the CryptoGraph program, to hold an attribute of a trade edge. Can be used for sorting, for the summary functions of the program.

self.percentPriceChange24hr = a later addition, specific to the CryptoGraph program, to hold an attribute of a trade edge. Can be used for sorting, for the summary functions of the program.

self._visited = to assist in preventing loops when traversing the graph

Methods are:

setVisited and clearVisited = these are basic setters, similar to the implementation in DSAGraphVertex.

dataStructures.DSAGraphWithEdges

This class is based on previously submitted work, my submission for Practical 06 – Graphs. This class forms the basis for the CryptoGraph object (which inherits from DSAGraphWithEdges).

Attributes are:

self._vertices = a Sortable List object, to store the vertices that make up the graph. Specifically, self._vertices is a SortableList of DSAGraphVertex objects.

self._edges = a SortableList of DSAGraphEdge objects.

self.edgeCount = a basic counter.

self.vertexCount = a basic counter.

Methods are:

self.addVertex = adds a vertex

self.getVertex = returns a vertex object

self.addEdge = this is the primary way that data is added to the graph in the cryptoGraph program. a from label, a to label and a weight can be specified, although the weight defaults to 0.0. This is the case when the 'skeleton graph' is constructed from the asset data – we do not assign edge weights at that stage, as the weights are contained in a different file.

self.removeVertex = removes a vertex

self.getEdge = returns an edge object

self.hasVertex = a check to see if the graph contains a vertex with a matching label

self.getAdjacent = this method has been changed since the version of this class submitted for practical 6. The graph with edges implementation stores edge information in the DSAGraphWithEdges._edges list. In the previous implementation, edges were stored by using an adjacency list for each vertex. This is deprecated and is not used in this edge version of the graph. Self.getAdjacent now iterated though the self._edges list and looks for edges where the fromVertex = the label we are looking for. It then returns a list of strings, each of which is the label for a vertex.

dataStructures.DSAListNode

This class is based on previously submitted work, my submission for Practical 04 – Linked Lists. Each instance of this class is an item in a list. It is used as the list element for all three list implementations

in the program (although this could be simplified in a future version by refactoring the code to use a single implementation).

Attributes are:

self.value = the object that is the item in the list. In the CryptoGraph program, linked lists are used extensively. As such, this can be many different object types. It could be a DSAGraphVertex, a DSAGraphEdge, a string representing an asset code, or another linked list (in a few places we have linked lists of linked lists).

self.next = reference to the next item in the list

self.previous = reference to the previous item in the list

dataStructures.DSALinkedListDE

This class is based on previously submitted work, my submission for Practical 04 – Linked Lists. Each instance of this class is a list.

This class has one additional method:

removeValue(self, value) = this allows the user to remove a value from the linked list, and will update the next and head references. This functionality is used by the **Asset Filter** option, item number 7 in the interactive mode of the CryptoGraph program.

The other methods will not be described here, as this is a direct implementation of the structure used in the practicals. This implementation is only used as a template for two other classes, TradePath and SortableList, and the extra functionality will be described there.

dataStructures.SortableList(DSALinkedListDE)

This class extends DSALinkedListDE to be sortable. It uses a recursive mergeSort algorithm that is based on the one from https://www.geeksforgeeks.org/merge-sort-for-linked-list/ but adds the functionality to sort by any attribute, and with high value first or low value first (my additions).

Additional Methods are:

```
def sortByAttribute(self, attribute="edgeCount", order="high"):
```

This is a wrapper method. This is particular to the implementation of linked lists: the SortableList object only refers to a .head item. So we need to update the head (and all of the links underneath that in the structure.

we call self.mergeSort(self.head, attribute, order) and this returns our new head.

Attribute is the attribute to sort by, which could be weight or price, colume24hr, or percentPriceChange24hr.

Order is ‘high’ or ‘low’, and specifies whether the first item in the returned list will be the highest value, or the lowest value.

helper methods for this are

```
def getMiddle(self, head):
```

This method returns a node just less than halfway through the list.

```
def mergeSort(self, h, attribute, order):
```

This method then breaks the list in two, and performs a recursive merge sort on each half. When the lists are halved enough times to be one item, we have reached the base case, and the single item is returned up the chain of recursive calls.

The two halves that were divided are merged back together with a separate method:

```
sortedMerge(self, a, b, attribute, order):
```

This method will merge two lists back together. It allows for one of the sublists to be empty, in which case it will return the other sublist.

This method is also recursive. Please bear in mind, that it updates the links between all the items in the lists, but it only returns the head value, as that is the only pointer we have in our linked list.

Please note: This merge sort implementation does not update the tail references of node, and as such we lose our doubly ended doubly linked functionality that was present in the parent class DSALinkedListDE. Whilst it does not affect the functioning of the Cryptograph program, this is something that definitely needs to be resolved in future work, so that the SortableList module is self contained and fully working, to prevent errors occurring in future development that may make naïve use of that module.

cryptoGraph.BinanceTradingData() object

This class is only instantiated once when running the crytoGraph program. The purpose is to hold references to the trade data files, and to provide methods for creating a graph structures from these. Basically, an instance of this object holds the raw data, the methods to parse it, the methods to display it, and the methods to create a graph from it.

The reason for implementing this class, was that I felt that the raw trade data is something that needed to have its own methods. Initially, I reasoned that the BinanceTradingData object was a snapshot of the state of the market at a particular point in time, and that we would need methods to make calls to the API to update this. In practise, this has not been implemented yet, but could be at a future date. Also, a justification for keeping the BinanceTradingData separate from the main graph, is that the graph can be rebuilt from the trading data at any point. Further justification exists in that some basic functionality of the program can be implemented by only looking at this BinanceTradingData object, and not referring to the more complex graph object to be discussed later.

Attributes and default values for this class are

```
trades_24hr_filepath='24hr.json',
exchangeInfo_filepath='exchangeInfo.json'
localPrices_filepath ='price.json'
```

and default paths are specified in the constructor.

There is only ever one active instance of this object in the program, and is stored in the variable called validTrades. This name was chosen to represent how the structure of the graph is initially built: we have a validTrades object which shows all valid trades between asset vertices, and then we move on to populating the edges with weights and attributes.

Methods are:

```
displayTradeDetails(self, symbol):
```

This method searches the 24 hour trades file for symbols that match the symbol parameter. This is an O(N) operation as we are iterating over a dictionary returned by the json.loads method. It then prints this to the user.

Next the method makes a call to the Binance API to get the latest Symbol price, and compares this to the weighted average price of the last 24 hours from the 24hrs.json file. It will print the difference to the user. This operation is only O(1) so it can be considered to be just as efficient as recording the average weighted price from our previous operation iterating over the dictionary in a variable, and referencing that variable.

```
createSkeletonGraph(self)
```

This method takes no arguments, and creates a CryptoGraph object from the BinanceTradingData object. The method instantiates an empty CryptoGraph object, and then iterates through the exchange info dictionary that is returned from the json.loads function. **This may be a point of**

inefficiency, we run json.loads again in this method, perhaps it would have been better to parse the json files once, and store their attributes in the cryptograph object directly to avoid reparsing… Given the client-imposed limitation to avoid built-in data structures except when directly parsing files, I was unsure of the best approach to implement in this regard.

The method iterates through the json dictionary and examines each trade symbol. If trading is allowed, it adds this edge to the graph (please note that as part of adding an edge, the vertices are added to the graph, but this will be discussed in the implementation of the CryptoGraph object and its parent class, DSAGraphWithEdges).

This method returns the graph object to the calling method.

```
getAssetTrades(self, assetName):
```

This method takes an asset name, and iterates over the exchange Info file to get all the possible trades for that asset. The asset trades are stored in a linked list, however in retrospect this may not be the more efficient data structure, as we do not need to know any ordering information. The reason I chose to use a LinkedList data structure was the lack of a requirement to resize the list if it became full, which is a requirement of array based data structures. In retrospect, this may not be a significant concern as I could set the size to be equivalent to the longest path likely to occur, so resizing would be infrequent if at all. I retrospect, a queue implemented with an array would have been a better choice to hold the data returned from this method.

cryptoGraph.CryptoGraph object

```
class CryptoGraph(DSAGraphWithEdges):
```

This class inherits from the DSAGraphWithEdges class in dataStructures.py that was submitted as part of Practical 6 – Graphs. It was decided to keep the class definitions in the dataStructures.py module as more general data structures that could be imported, and to have the functionality that was immediately specific to the CryptoGraph program contained within the cryptoGraph.py file, for easy access for code maintenance and modularity. It should be noted that the parent class DSAGraphWithEdges is not instantiated at all when running the CryptoGraph program, but the decision to separate CryptoGraph specific code from more general data structures was maintained.

This child class has the following new methods that do not exist in DSAGraphWithEdges:

```
loadEdgeAttributesFrom24hr(self, binanceDataObject)
```

This method loads trade attribute data into the edges of the graph. This may seem confusing or an unusual structure to implement, but my rationale is as follows.

1) we need to store a reference to the trade data files that is not a global variable
2) the files need operations performed on them, and the first step is building the skeleton of a graph that defines allowable trades as edges

3) The information for these edges is contained in another json file, so rather than populate it at the time of creation, I saw this as a separate process (even though it could be done within nested for lops, with both files open for reading simultaneously).
4) I also saw the loading of the edge attributes as something that may want to be done separately to the creation of the graph. Prices are liable to update much more regularly than the options of allowable trades in the Binance crypto currencies. As such, we make the skeleton CryptoGraph, and then population of the attributes happens separately, and can be done multiple times. A reference to the original data (BinanceDataObject) is then retained separately as well.

Given this structure, the way this method works, is that is belongs to the CryptoGraph class, and is fed the Binance data as an argument. The attributes are populated using an algorithm with two nested for loops. Sequentially searching through the list is O(N) complexity. Then searching through the list of dictionaries returned from json.loads is also O(N). This equates to $O(N^2)$ complexity. Having populated the edges at time of creation would not have reduced this complexity.

Even python built-in lists only allow quick index based searching. If we wanted to reduce the complexity of this, we would need to store the edges as a hash table instead of a linked list. This suggests a different choice of implementation for the CryptoGraph._edges attribute that we need to examine:

As the order of the edges in the graph is not important, could we sacrifice the ordered properties of a linked list data structure for the quick access properties of a hash table. We could implement this so that on searching for a symbol code, we would get the object with O(1) complexity (average scenario, can also be O(N) worst case).

Despite this initially promising lead, we do in fact need to retain some sorting to our linked lists. This is because the program heavily depends upon sorting the edges according to different attributes. How about using a Binary Search Tree? This would allow faster search times, and also allow us to maintain the tree in sorted state. I would argue that given that we need to sort and resort by **different attributes**, linked lists are a good choice.

```
loadCostFromLocalJson(self, binanceDataObject)
```

This method replaces some if the functionality of the previously discussed method, loadEdgeAttributesFrom24hr(). In my early implementations of this program, that method would load the edge weights with the avgWeightedPrice from the 24hr file.

This proved problematic for a single reason: while trades between two assets might be allowed in principle, there was not necessarily any trades between those assets in the last 24 hours, and a value of 0.00 was recorded in the input files. There were a few ways around this, such as removing trade edges that had not been traded in the last 24 hours, but I felt that this was not a true picture of the state of allowable trades. I examined the Binance API documentation and found that up to date price information was available for all possible trades by a GET request to https://api.binance.com/api/v3/ticker/price

I decided to make this functionality core to my program, so I downloaded a file price.json and have included it in the software package. This is a list of dictionaries, one dictionary for each asset.

Data is added to the edges with once again $O(N^2)$ complexity. This is not very efficient, but is a limitation of the data being spread across numerous files, and the limitations imposed.

```
loadEdgeWeightsFromCurrent(self)
```

EXTENSION: This method is a natural extension of the functionality provided in loadCostFromLocalJson(). This method updates the edge weights with data from GET request to the Binance API. To avoid possibly misleading results, the price data for all trades is downloaded in a single GET request and thus represents a snapshot of the markets at a particular time. The list of edges, and the price.json are iterated over in a nested for-loop algorithm, so the time complexity is again $O(N^2)$.

```
getAllPaths(self, startNode, endNode)
```

This is a wrapper method for _getAllPathsRec. These methods are used to traverse through the graph to find all possible paths between two user specified assets. It establishes a TradePath() object to store a single path, and a pathContainer object to store all the possible paths found.

In addition to establishing the containers, this wrapper method uses the arguments startNode and endNode, which are just text strings of the asset codes, to get the actual objects from the CryptoGraph object. It then sets all vertices in the validTrades graph to not-visited (False). This is itself is an O(N) operation!

It then calls the recursive method with the start node, end node, path, and path container.

The recursive method appends all valid paths to the pathContainer. This wrapper method returns the pathContainer to the calling function in the main code.

```
_getAllPathsRec(self, u, d, path, pathContainer)
```

Firstly the method sets the origin vertex to be ‘visited’, and adds it to the path list.

The base case is if the origin label == the destination label. That means we have successfully reached the end of the path. The path is deep copied (as it is a linked list of linked lists, and we need to preserve everything). A call to the calculateTotalCost() method is made to calculate the total cost for the complete path and add it as an attribute to the completePath object (this object is of type TradePath(), please refer to implementation for details).

If we have not reached the base case, another recursive method call is made, for each of the adjacent vertices for the origin node of the current call (if it has not already been visited… this avoids loops).

All complete paths are added to the pathContainer linked list, which is not returned but is immutable in memory so it can be accessed by the same memory pointer and returned by the calling method getAllPaths()

```
getEdgeValue(self, fromVertex, toVertex, attribute='weight')
```

This method is used to get the attribute of a graph edge object. It is only implemented for simplicity of code, as the same result can be obtained using the getEdge method that is implemented in the DSAGraphWithEdges parent class. This could be removed with refactoring at a later stage.

```
getTopFiveTradePathsByCost(self, fromAsset, toAsset)
```

This method calls the 'getAllPaths()' method, which returns all possible paths between two assets as a SortableList object. It then sorts these paths by the attribute 'cost', order 'low' with the lowest values first.

It then calls the 'getFirstXElements' function (it is a static method) to return the first 5 elements of this SortableList object. These are returned as a list which is then iterated through in the main function for display.

There is some redundancy here that leads to over complexity. This will be discussed in the Static Methods section.

```
__repr__(self)
```

This returns a python f string that includes the edge count and vertices count, which was useful in debugging.

cryptoGraph.TradePath object

```
class TradePath(DSALinkedListDE):
```

This class inherits from DSALinkedList DE, which is a double ended doubly linked list implementation largely unmodified from what was submitted for DSA Practical 04 – Linked Lists. Each instance of this class stores a path traversal through the graph, as a sequence of vertices (technically in Python these are memory references to the graph vertices). It is used to store traversal paths from one asset to another, as determined in menu options 5 and 6 of the cryptograph interactive menu.

Additional attributes:

The init method has been replaced to include the attribute self.cost. This is a float-type field used to store the cumulative path weight for paths through the graph.

Additional Methods:

In addition to the methods inherited from the parent class, the TradePath class implements:

```
calculateTotalCost(self, tradeGraph):
```

This method requires a reference to the tradeGraph object. Its functionality is determined by the design of our system, where the paths are found as a sequence of asset codes, but the costs of the trades are stored in the edges. As such, for each pair of assets, we need to find the DSAGraphEdge object that matches and get its cost.

The method looks at each pair of vertices in the TradePath two at a time, and uses the getEdgeValue method of the DSALinkedListDE parent. There is an issue with efficiency in this part of the program: we are not calculating cost as we go, when making the path… so we need to do revisit each of the vertex pairs again at the end and perform getEdge() which is an O(N) operation, where N is the number of edges. This is quite inefficient as (worst case) we must check every edge in the graph. In a typical validTrades graph object, there were found to be 290 vertices and 1065 edges. We need to do this for each vertex pair so the method is $O(N^2)$ in total. This is in addition to finding the path in the first place! It may have been more efficient to find paths through the graph by traversing the edges, and storing the costs as we go. Another option could have been to store the edges as a hash table, so that getEdge would be an O(1) operation.

Static Methods (functions)

```
getFirstXElements(inList, x):
```

For example, if x is set to 5, the getFirstXElements function appends each of the 5 items to a list and then returns it, which is then again iterated over for display. It would perhaps more efficient to directly print the top 5 paths to the user as they are found by the getFirstXElements() function. Whilst being more specific to the current requirements of the CryptoGraph program, refactoring the code in this way would reduce the general useability of the getFirstXElements() function.

At the moment this is implemented as a static method, though it could just as well be included as a method in the SortableList class.

```
getCurrentSymbolPrice(symbol):
```

EXTENSION: Takes a trade symbol as a string, and performs a GET request to the Binance API. Returns the current trade price as a float. This method implements exception handling for network or connection errors, and handles HTTP response errors as well.

```
getAllSymbolPrices():
```

EXTENSION: Gets all current trade prices with a GET request to the Binance API. This method implements exception handling for network or connection errors, and handles HTTP response errors as well.

```
serialize(path, myObject)
```

Basic function that takes a file path and an object, and pickles the object to that path.

In the CryptoGraph program, the path is hardcoded and the same path always used. As such only one 'snapshot' of the graph object can be saved.

```
deserialize(path)
```

Reads the CryptoGraph object back in from the serialized file. Incorporates exception handling.

```
displayUsage()
```

A basic function to print usage information to the console.

```
runInteractiveMenu()
```

The function that contains the menu code and a section to run each of the choices and call the appropriate methods/functions.

This could possibly be refactored to have the run code for each user menu selection contained in a method. My approach to refactoring would be to have Menu class, with MenuItem classes for each menu option. For now, the code works, but there could be issues with variable scopes. I would prefer a higher degree of modularisation to maximise cohesion and minimise coupling.

This steps followed are described in more detail in the Walkthrough section of this design document.

```
runReportMode(exchangeInfo_filepath, trades_24hr_filepath)
```

This method is called when the program is run with the command line argument -r. It runs a selection of the summary methods and print the results to the console. This is described in more detail in the Walkthrough section of this design document.

```
main():
```

The main function handles the command line arguments, and performs one of the following:

1) displays the correct command line usage to the user
2) Runs the interactive menu
3) Runs report mode
4) Alerts the user that they have specified an incorrect number of arguments, and also displays the command line usage

Future Work

Missing items or suggested enhancements.

Asset Information:

One of the requirements from the client is to be able to display asset information. However the initial data provided by the client did not specifically provide much asset information, with the only mentions of individual assets being their presence as toAssets or fromAssets as part of trades. To be fair, if I had more familiarity with cryptocurrency terminology at the start of the project I would have investigated the Binance API in more depth from the start. But this shortfall was not identified until late in the project development. Asset information is currently limited to the potential trades with other coins that a particular asset can make.

A strategy to include further asset information, would be to extend the DSAVertex class to hold more attributes, and load these in from the assets.csv file that was provided by the client later in the project. This is a simple extension to the program, and given the well structured and documented nature of the program, it should be easy to add in at a later date.

Data Structure Choice:

Linked lists have been used extensively in this implementation, but is it possible that in some cases a different data structure would have been a more efficient choice, to reduce the complexity as measure by Big O analysis. This has been discussed in some detail in the class description section of this design document, but a few points a worthy of reiteration:

The prime example that comes to mind is the graph edges. These are currently implemented as a linked list. But a more efficient approach may have been to implement the edges of a graph as a Hash table for quick lookup of a particular edge to retrieve its attributes. Searching for a particular edge in our current LinkedList implementation is a O(N) operation, but with a hash table it would be an O(1) operation!

The second example is when storing a temporary collection of items to then display these to the user, such as for the various ‘top 5’ summaries. Instead of writing these items to a linked list we could use an array based Queue.

Asset Filter Section:

The asset filter operates on the graph, and removes vertices and edges containing the chosen assets. However, this does not affect the results from ***Option 3: Find and Display an Asset***. This is due to the fact that Option 3 is based on reporting from the BinanceData object, which is a container for the json files.

The **Asset Filter** updates the graph, and effectively removes vertices from paths when doing graph traversals. But is does not affect the results from ***Option 3*** or from ***Option 4: Find and display trade details***.

In this way, the reporting is really broken up into two categories, those that report on the raw data, and those that report on the data contained in the graph. As a suggestion for the future, I think that this should be unified, so that asset exclusions exclude assets from all reporting.

Classes and Inheritance

There are two Linked List implementations in the code, DSALinkedListDE and SortableList. These could be combined into a single class that is suitable in all roles. The justification for the current two class implementation, is that it is inappropriate to conceptualist as some of the lists as 'sortable' : they should never be sorted! A good example of this is the TradePath class – these paths are sequential, and we should not imply in the code that they could be sorted and the order rearranged.

Containerisation

A common problem with sharing python scripts is that different python versions or library version, can pose untested situations. To ensure that the program has been fully tested, it could be distributed as a containerised executable, developed with a tool such as Docker.

Structure of the program

The function runInteractiveMenu needs to be modularised further. This became most apparent when developing unit tests. If each of the user choice items was contained in a separate function or method, these user choices could be tested more effectively. To my mind this is the biggest weakness of the approach I have taken.

Optimisations

During testing, an issue was identified when calculating the cost of trade paths. Using the data provided for the task, the only apparent trade price information was contained in the 24hrs.json file, that summarises all trades in the last 24 hours. This data was used to extract edge weights for the graph, so that the edge weight was equal to the average trade price for all trades of that symbol in the last 24 hours.

However, some trades are uncommon, and while they are permissible, may not have occurred in the last 24 hours. To this end, rare trades would be assigned a cost of 0.0.

This was problematic when calculating the cost of trade paths, as that component of the path would be assigned a weighting of 0.0, thereby making the overall trade cost = 0.0.

I attempted three solutions to this problem:

1) leverage the Binance API to get current trade price for each symbol individually

```
https://api.binance.com/api/v3/ticker/price?symbol=ETHBTC
```

   This proved to be very slow in practise.

2) leverage the Binance API to get current trade price for each symbol individually, but only for those symbols with no trades in the last 24 hours

   This also proved to be very slow:

```
if cost == 0.0:
     symbol = fromLabel + i._label
     cost = self.getSymbolPrice(symbol)
```

3) leverage the Binance API to get current trade prices for all symbols in one get request, and use this to update all edges in the graph iteratively in Python

This approach proved to be quickest, and was implemented.

References

The code makes heavy re-use of code based on the lecture materials for the Curtin University of Technology Unit Data Structures and Algorithms, COMP5008. Code submitted for Practical 04 – Linked Lists and Practical 06 Graphs has been included in slightly modified form in the file dataStructures.py

No materials have been formally consulted outside of the lecture materials except two algorithms described here:

https://www.pythoncentral.io/find-remove-node-linked-lists/

https://www.geeksforgeeks.org/merge-sort-for-linked-list/