DSA Assignment Report -
By Austin Bevacqua (20162896)

Information For Use

Introduction:
The CryptoGraph algorithm's main functionality involves reading in various files related to Cryptocurrency, performing analysis and displaying the results to the screen.

Installation / Requirements:
A valid installation of Java is required

- The algorithm requires a singular file which is acquired at the link
  https://www.binance.com/api/v3/exchangeInfo
  This file will be referred to as the "asset data file".
  Included in submission under name "assetdata.json"

- The program also optionally can import two other files
  https://www.binance.com/api/v3/ticker/24hr
  This file will be referred to as the "trade data file"
  Included in submission under name "tradedata.json"

  https://coinmarketcap.com/all/views/all/
  This file, once copied into .csv format, will be referred to as the "asset info file".
  Included in submission under name "asset_info.csv"

Terminology / Abbreviations:
Cryptocurrency - A digital currency in which users can make transactions for goods and services. The prices of these digital currencies change extremely frequently.
Coin - Many cryptocurrencies are also known as "coins", for example "Bitcoin".
Asset - For the purpose of this assignment, cryptocurrencies will be referred to mainly as "assets".
Trade - As cryptocurrencies can be traded for other cryptocurrencies, a transaction between two digital currencies will be referred to as a "trade".
Trading Network - Not every cryptocurrency can be directly traded with another cryptocurrency. Asset "Trading networks" exist to streamline the process of trading between assets, with the "Bitcoin trading network" allowing the Bitcoin asset to trade with over 240+ other assets.
BTC - An abbreviation for the cryptocurrency Bitcoin
ETH - An abbreviation for the cryptocurrency Ethereum.
Volume -The amount of an asset/trade that has circulated in a given time window. For example, 400 000 BTC might have been transferred in the past 24 hours; the 24 count for BTC is 400 000

Count - The number of trades between two assets that has been processed within a given time window.
Market Cap - The maximum number of a given asset that can exist in circulation at a given time.
Binance - One of many websites in which users can buy, sell and trade cryptocurrencies.

Walkthrough:
The algorithm has three starting options:

- Usage Information
- Report Mode
- Interactive Mode

Usage Information
To access usage information, the user must run the main file **CryptoGraph** with no parameters or > 4 parameters.

- E.g. “java CryptoGraph”

```
~~Incorrect usage.~~
    Interactive Mode: "java cryptoGraph -i"
    Report Mode: "java cryptoGraph -r <asset_file>"
    Report Mode: "java cryptoGraph -r <asset_file> <trade_file>"
    Report Mode: "java cryptoGraph -r <asset_file> <trade_file> <asset_info_file>"
```

Usage information will display to the user how to access the main functionalities of the program, report mode and interactive mode.

Report Mode
To access report mode, the user must supply the asset data file, trade data file and optionally the asset info file in the command line parameters, preceded by “-r”

- E.g “java CryptoGraph -r assetdata.json tradedata.json asset_info.csv”

```
Number of currencies on the BTC Network: 234
Number of currencies on the ETH Network: 101

Top 10 Trades by price:
1:BTCIDRT(1.683129173 7E8)
2:BTCBIDR(1.680191956E8)
3:HOTBTC(2.5E7)
4:BTCBKRW(1.316534580875113E7)
5:MBLBTC(7142857.142857143)
6:POEBTC(6666666.666666667)
7:ETHBIDR(5611916.77)
8:BTCNGN(5391974.80868017)
9:TNBBTC(5263157.894736842)
10:STMXBTC(5000000.0)

Top 10 Assets by price:
1:42(42-coin) - 63732.45
2:NANOX(Project-X) - 21273.84
3:YFI(yearn.finance) - 13618.85
4:RBTC(RSK Smart Bitcoin) - 12509.8
5:imBTC(The Tokenized Bitcoin) - 12233.88
6:HBTC(Huobi BTC) - 12227.74
7:WBTC(Wrapped Bitcoin) - 12212.79
8:BTC(Bitcoin) - 12199.63
9:BTCB(Bitcoin BEP2) - 12162.12
10:RENBTC(renBTC) - 12116.89

Number of currencies: 2505
Number of Trades: 818
```

Once in report mode, the algorithm will parse / import data from the files, and create a graph with the cryptocurrency information.

Report mode will then display basic analytical information to the screen, such as how many coins are on the BTC & ETH trading networks, the top 10 most valuable trades, and if asset_info.csv has been imported, the top 10 most valuable assets.

Interactive Mode

To access interactive mode, the user must run the program with the parameter "-i"

- E.g. "java CryptoGraph -i"

The user will then be prompted to enter the filename of an asset data file, which the program will import.

```
Enter a name for the ASSET DATA (e.g. assetdata.json)
```

- User Input: "assetdata.json"

Once successfully imported, the user will be prompted with a menu of options to perform analysis or to manipulate the data with. The menu also allows the user to import the other two optional files, which will allow more detailed analysis of trades and assets.

Main Menu:

```
CURRENT GRAPH:
    Asset Data File: assetdata.json
    Trade Data File: <NOT FOUND>
    Asset Info File: <NOT FOUND>

Select an option from 0-8:
(1)Import Files
(2)Display an asset
(3)Display a trade
(4)Find trade routes
(5)Set asset filter
(6)Asset overview
(7)Trade overview
(8)Save
(0)Exit
```

The user will select an option, which will print information to the screen or print out another sub-menu. Once the user is finished, they can exit the program by inputting "0".

<User Input>:

<1>

```
(1) Asset Data Json <assetdata.json> (Will overwrite everything)
(2) Trade Data Json <tradedata.json>
(3) Asset Info csv <asset_info.csv>
```

<assetdata.json>

- Main menu is then shown

<2> (Display an asset)

```
Enter the name of the asset:
```

<BTC>

```
Asset BTC found!
Asset is involved in : 234 trades
Asset IS NOT on the BTC Network
Asset IS on the ETH Network

Name: BTC
Full Name: Bitcoin
Price: 12199.63
Price Change 4.2%
Volume: 3.46577063E10
Market Cap: 2.2597E11

CURRENT GRAPH:
    Asset Data File: assetdata.json
    Trade Data File: tradedata.json
    Asset Info File: asset_info.csv

Select an option from 0-8:
(1)Import Files
(2)Display an asset
(3)Display a trade
(4)Find trade routes
(5)Set asset filter
(6)Asset overview
(7)Trade overview
(8)Save
(0)Exit
```

<3>

```
3
Enter the name of the trade:
```

<LINKBTC>

```
Trade LINKBTC found!
FROM: LINK
TO: BTC
Price Change: 1.315E-5
Price Change: 1.35%
Average Price: 9.9321E-4
Volume: 2021396.8
Count: 52309

CURRENT GRAPH:
    Asset Data File: assetdata.json
    Trade Data File: tradedata.json
    Asset Info File: asset_info.csv

Select an option from 0-8:
(1)Import Files
(2)Display an asset
(3)Display a trade
(4)Find trade routes
(5)Set asset filter
(6)Asset overview
(7)Trade overview
(8)Save
(0)Exit
```

<4>

```
Enter the name of the ORIGIN asset
```

<DOGE>

```
Enter the name of the DESTINATION asset
```

<LINK>

```
Enter the maximum number of nodes to traverse (2-5)
```

<3>

```
25 number of routes found
A direct path does not exist between DOGE & LINK
Would you like to:
(1) Display all paths
(2) Display top 10 paths by cumulative price change
(0) Go to main menu
```

<2>

```
1: DOGE -> USDT -> BKRW -> LINK (126.51699999999998%)
2: DOGE -> USDT -> AUD -> LINK (124.46199999999997%)
3: DOGE -> BTC -> USDT -> LINK (99.16%)
4: DOGE -> USDT -> USDC -> LINK (93.309%)
5: DOGE -> USDT -> TRY -> LINK (88.066%)
6: DOGE -> BTC -> BKRW -> LINK (87.18500000000003%)
7: DOGE -> BTC -> AUD -> LINK (84.884%)
8: DOGE -> BTC -> USDC -> LINK (78.44600000000001%)
9: DOGE -> BUSD -> AUD -> LINK (66.123%)
10: DOGE -> BUSD -> BKRW -> LINK (64.11800000000001%)
```

- Main menu shown

<5> (Set Asset Filter)

```
Do you want to
(1) ADD to the filter
(2) REMOVE from the filter?
```

<1>

```
Enter the name of the asset
```

<ETH>

- Main menu shown

<6> (Asset Overview)

```
There are 2505 assets in the graph
Number of assets on the BTC Network: 234
Number of assets on the ETH Network: 101

Sort data by:
(1) Price
(2) Price Change %
(3) Volume
(4) Market Cap
(0) None
```

<3>

```
Top 10 Assets by volume:
1:USDT(Tether) - 4.9302683259E10
2:BTC(Bitcoin) - 3.46577063E10
3:ETH(Ethereum) - 1.639223632E10
4:BCH(Bitcoin Cash) - 2.430374003E9
5:LTC(Litecoin) - 1.879191876E9
6:TRX(TRON) - 1.796328043E9
7:EOS(EOS) - 1.683478978E9
8:XRP(XRP) - 1.62585081E9
9:LINK(Chainlink) - 1.401628985E9
10:XMR(Monero) - 1.271327017E9

Sort data by:
(1) Price
(2) Price Change %
(3) Volume
(4) Market Cap
(0) None
```

<0>

- Main menu shown

<7> (Trade Overview)

```
There are 818 trades in the graph

Sort data by:
(1) Price
(2) Price Change
(3) Price Change %
(4) Volume
(5) Count
(0) None
```

<3>

```
Top 10 Trades by price change percent:
1:HOTBTC(25.0%)
2:ASTBTC(22.64%)
3:ANKRUSDT(22.0%)
4:ANKRBNB(21.695%)
5:ANKRBTC(21.311%)
6:ADADOWNUSDT(17.769%)
7:ADAUPUSDT(16.892%)
8:OCEANUSDT(14.616%)
9:OCEANBTC(13.842%)
10:OCEANBNB(13.796%)

Sort data by:
(1) Price
(2) Price Change
(3) Price Change %
(4) Volume
(5) Count
(0) None
```

<0>

- Main menu

<8> (Save)

- NOT IMPLEMENTED, shows main menu again

<0>

- Exits algorithm

Future Work:

- The algorithm could be enabled to scalp the data of the Binance website, making the process simpler through the reduction of file IO and the information more easily be able to be up to date.
- The algorithm could use visualisation modules to create a visual overview of the trading networks and markets
- A limitation of the algorithm in its current state is that files retrieved different times will cause errors when performing analysis. The algorithm could be updated to perform checks to ensure compatibility between files.
- The algorithm could be updated to enable the saving of data once imported, whether through Java serialization or through file IO.

Traceability Matrix

		Requirements	Design / Code	Test
1	Menu / Modes	1.1 Algorithm displays usage information if run with no arguments	CryptoGraph.main()	UsageTest.txt - 1.1

		1.2 Algorithm displays usage information if run with >4 arguments	CryptoGraph.main()	UsageTest.txt - 1.2
		1.3 Algorithm enters report mode if argument “-r <file> <file> (<file>)” is entered	CryptoGraph.main()	UsageTest.txt - 1.3
		1.4 In report mode, the algorithm notifies the user if data files are incorrect.	Menu.reportMode()	UsageTest.txt - 1.4
		1.5 Algorithm enters interactive mode if the “-i” argument is ran	CryptoGraph.main()	UsageTest.txt - 1.5
		1.6 In interactive mode, the user can enter input to perform actions	Menu.interactiveMode()	UsageTest.txt - 1.5
2	File Input / Output	2.1 The system can read in the Asset Data Json file.	FileIO.readAssetData()	UnitTestFileIO - Test 2.1
		2.2 The Asset Data Json file is parsed and formatted into a Graph.	JsonParser.parseAssetData()	UnitTestFileIO - Test 2.2
		2.3 Given an existing graph with Asset Data, the algorithm can read in a Trade Data Json file	FileIO.readTradeData()	UnitTestFileIO - Test 2.3
		2.4 The algorithm will find existing trades (paths between nodes) and update them with information from the Trade Data Json.	JsonParser.parseTradeData()	UnitTestFileIO - Test 2.4
		2.5 The algorithm can, given an existing graph with Asset Data, read in the Asset Info csv.	FiileIO.readAssetInfo()	UnitTestFileIO - Test 2.5
		2.6 The algorithm will find / create nodes and add information extracted from the Asset Info csv file.	FileIO.readAssetInfo()	UnitTestFileIO - Test 2.6
		2.7 If any file can not be found or is not of the expected format, an error will be displayed to the screen	FileIO.readAssetInfo()	UnitTestFileIO - Test 2.7
3	Report /	3.1 In interactive mode, the user	Menu.interactiveMode()	UnitTestFileIO

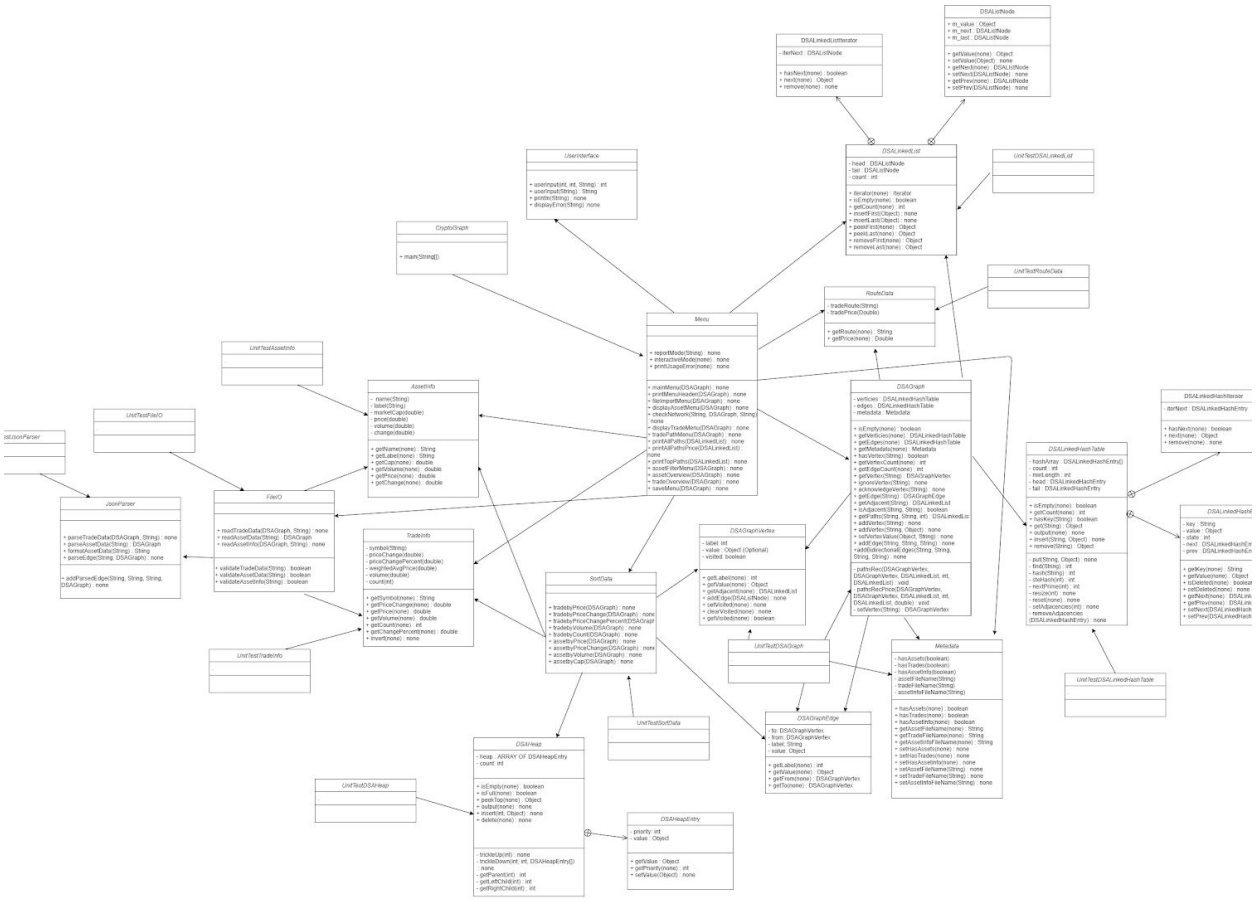
	Interactive Mode	will first be prompted to import an Asset Data json file.		
		3.2 After the Asset file has been successfully imported, a list of options will be presented to the user. The user will use user input to select an option	Menu.mainMenu()	MenuTests.txt
		3.3 In report mode, the algorithm will print out basic information related to the data.	Menu.reportMode()	MenuTests.txt
		3.4 If an Asset Info csv file has been imported, the system will display the top 10 most valuable assets.	SortData.assetbyPrice()	UnitTestSortData
4	Find and display asset	4.1 The user specifies through user input which asset they want to display	UserInterface.userInput()	UserInterfaceTest.txt
		4.2 If the asset inputted does not exist, an error is displayed to the screen	Menu.displayAssetMenu()	MenuTests.txt
		4.3 If the asset does exist, first the number of trades the asset it involved in will be displayed to the screen	DSAGraph.getAdjacent()	UnitTestDSAGraph.graphTest
		4.4 Then, whether it is on the BTC network will be displayed to the screen	DSAGraph.getAdjacent()	UnitTestDSAGraph.graphTest
		4.5 Whether it is on the ETH network will be also displayed to the screen	DSAGraph.getAdjacent()	UnitTestDSAGraph.graphTest
		4.6 If an asset info file exists and has been imported, general information relating to the asset will be displayed to the screen.	Menu.displayAssetMenu()	MenuTests.txt
5	Find and display trade details	5.1 The user specifies through user input which trade they want to display	UserInterface.userInput()	UserInterfaceTest.txt
		5.2 If the trade inputted does not exist, an error is displayed to the	Menu.displayTradeMenu()	MenuTests.txt

		screen		
		5.3 The basic trade information will be displayed to the screen (the source and destination of the trade)	Menu.displayTradeMenu()	MenuTests.txt
		5.4 If a Trade Data Json has been imported, additional information about the trade will be displayed.	Menu.displayTradeMenu()	MenuTests.txt
6	Find and display potential trade paths	6.1 The user will be prompted to enter the names of two assets.	UserInterface.userInput()	UserInterfaceTest.txt
		6.2 If asset one or asset two does not exist, an error message is printed to the screen	Menu.tradePathMenu()	MenuTests.txt
		6.3 The user will be prompted to enter a maximum size for the paths between assets, with the maximum being length 5.	UserInterface.userInput()	UserInterfaceTest.txt
		6.4 The algorithm will find all paths between nodes, outputting a heap once completed.	DSAGraph.displayPaths()	UnitTestDSAGraph.graphTest
		6.5 Whether a direct path exists between the nodes is printed to the screen	DSAGraphNode.getEdge()	UnitTestDSAGraph.graphTest
		6.5 The total number of paths are printed to the screen	DSALinkedList.getCount()	DSALinkedListTest
		6.6 The user will be prompted if they want to see ALL paths, or, if Trade Data json has been imported, the top 10 paths	UserInterface.userInput()	UserInterfaceTest.txt
		6.7 If the user selects all paths, every path is printed to the screen	Menu.printAllPathsPrice()	MenuTests.txt
		6.8 If the user selects top 10 paths, the top 10 highest trades routes by cumulative price change will be printed to the screen.	Menu.printTopPaths()	MenuTests.txt

7	Set asset filter	7.1 The user will be given the choice of adding an asset to the filter or removing an asset from the filter.	UserInterface.userInput()	UserInterfaceTest.txt
		7.2 The user will then specify which asset they want to perform an action on.	UserInterface.userInput()	UserInterfaceTest.txt
		7.3 If the asset does not exist, an error is displayed to the screen	Menu.assetFilterMenu()	MenuTests.txt
		7.4 If the asset is found and the user wants to ADD the asset to the filter, the asset is set as “ignored”	DSAGraph.ignoreVertex()	UnitTestDSAGraph.graphTest
		7.5 If the asset is found and the user wants to REMOVE the asset to the filter, the asset is set as “not ignored”	DSAGraph.acknowledgeVertex()	UnitTestDSAGraph.graphTest
8	Asset overview	8.1 The algorithm will list the number of assets, in total, that exist within the graph	DSAGraph.getVertexCount()	UnitTestDSAGraph.graphTest
		8.2 If an asset info file exists, the algorithm will ask the user if they want to sort the values by price, volume, market cap, ect...	UserInterface.userInput()	UserInterfaceTest.txt
		8.3 The algorithm will display the top 10 assets by the user-selected sorting criteria.	Menu.assetOverview()	UnitTestSortData
9	Trade overview	9.1 The algorithm will list the number of trades, in total, that exist within the graph.	DSAGraph.getEdgeCount()	UnitTestDSAGraph.graphTest
		9.2 If a trade info file exists, the algorithm will ask the user if they want to sort the values by price, volume, count, ect...	UserInterface.userInput()	UserInterfaceTest.txt
		9.3 The algorithm will display the top 10 trades by the user-selected sorting criteria.	Menu.tradeOverview()	UnitTestSortData
10	Save data (serialised)	10.1 The user will be asked to input a file name for the data to	Unimplemented	Unimplemented

		be saved as		
		10.2 The graph and any stored data will be saved into a serialised file.	Unimplemented	Unimplemented

Class Diagram



Class Descriptions - Overview

Class Name	Type	Description	Test File / Harness
CryptoGraph	Main	The main entry point of the program. Verifies command line arguments.	UsageTest.txt
FileIO	Static Class	Handles the reading and writing of files into the	UnitTestFileIO

		algorithm	
JsonParser	Static Class	Parses Json data files into a graph.	UnitTestJsonParser
UserInterface	Static Class	Handles all user input and output	UserInterfaceTest.txt
Menu	Static Class	A static class which handles menu items in the report and interactive modes of the algorithm.	MenuTests.txt
SortData	Static Class	Handles the sorting of asset or trade files by specific characteristics.	UnitTestSortData + SortDataTests.txt
DSAGraph	Data Structure	An implementation of a directed, weighted graph using the Linked Hash Table data structure.	UnitTestDSAGraph.graphTest
DSALinkedList	Data Structure	A data structure which represents a list of items, where the item order has no place in memory, but are linked together through classfields.	UnitTestDSALinkedList
DSALinkedHashTable	Data Structure	A data structure which combines the functionality of a linked list with a hash table. Entries are added to an array with a hash hey, but can be iterated over.	UnitTestDSALinkedHashTable
DSAHeap	Data Structure	A data structure which enters objects into an array with a priority. The data structure ensures that entries with the highest priority get removed first.	UnitTestDSAHeap
DSAGraphVertex	Model Class	Used by DSAGraph to store information related to the Graph Vertices.	UnitTestDSAGraph.vertexTest
DSAGraphEdge	Model Class	Used by DSAGraph to store information related to the Graph Edges.	UnitTestDSAGraph.edgeTest

Metadata	Model Class	Stores information regarding the imported files to the graph.	UnitTestDSAGraph.metadataTest
TradeInfo	Model Class	An object which stores information regarding specific trades between assets. Used with Trade Info Json.	UnitTestTradeInfo
AssetInfo	Model Class	An object which stores information regarding specific assets. Used with Asset Info csv.	UnitTestAssetInfo
RouteData	Model Class	An object which contains "routes". Used by DSAGraph to store trade routes when performing path finding algorithms such as findPaths()	UnitTestRouteData

More detailed class descriptions are mentioned within justifications.

Justifications

Firstly, I will explain the overall structure and implementation of the program, and then explain the decisions made for every feature of the CryptoGraph algorithm.

Overall Structure

I decided to centralise all my data within a Graph data structure, using the DSAGraph created for DSA prac 6 as a template. I decided to use the cryptocurrency assets as vertices within the graph, and the trades between these assets would be represented with edges in the graph. To achieve this, all the data from the files would need to be parsed and stored in a graph Object

Graph Object

My vertices and edges were originally stored within linked list data structures, as they were frequently iterated over and requirement for dynamic storage was essential for this algorithm, as data may vary wildly depending on the imported file. However, I found that I was frequently trying to access specific elements within these linked lists for operations such as FIleIO, displaying asset information, finding paths between nodes, ect…, which on average was an O(n/2) time complexity for every access of the data.

The hash data structure solves this problem, as access time is on average case O(1), however the requirement of needing to frequently iterate through the dataset, a feature not well supported by hash tables, was deemed more important than access time.

The solution to this problem, I found, was to combine these data structures into a single data structure called the LinkedHashTable (File: DSALinkedHashTable). Java has a similar implementation documented as a LinkedHashMap (Link to source code: https://docs.oracle.com/javase/8/docs/api/java/util/LinkedHashMap.html). My implementation of the LinkedHashTable is mostly based on the DSAHashTable algorithm submitted for DSA prac 7, which would insert elements into a list based on a hashed key. Where my algorithm differs from a hash table is that the individual elements have references to each other in the form of the next and prev class fields; identically to a linked list. The hash table also has references to the head and tail nodes, where iterating over the elements starts at the head node and gets the “next” element until the end of the dataset. When elements are inserted into the LinkedHashTable, references to the previous and next nodes are created, maintaining a linked list-like structure for iteration.

Using this data structure increases complexity when inserting elements, and increases storage needed for the extra variable overheads, however the benefit of O(1) accessing of nodes in the algorithm (using the hash functions) and the ability to iterate over all elements (Using the linked list-like functions) largely makes up for this.

Additionally, DSAGraphVertex and DSAGraphEdge were chosen to be separate files in the algorithm rather than private inner classes. This is due them being required by other functions to access information within the algorithm. For example, the SortData class requires the DSAGraphVertex and DSAGraphEdge classes to perform analysis on the overall graph.

For the graph type, I decided to implement a weighted and directed graph.

Weighted:
A weighted graph allows me to store information and data within the edges, allowing for information such as trade volume to be stored.

Directed:
Binance data only gives information for trades going in one direction, for example if ETH -> BTC exists, BTC -> ETH will not exist in the data. My interpretation of this was that for every trade that exists in one direction, there is an un-stored trade that exists in the other direction. Although BTC -> ETH does not exist within the data files, I have interpreted that the trade still exists. Cryptocurrencies trades work similarly to real exchanges of currency; when you trade AUD for USD, the inverse trade USD for AUD is simultaneously occurring; one trade cannot exist without the other. Furthermore, just like physical currencies, we can work out the rate and price of the other inverse trade using simple math. If 1 AUD is worth 0.7 USD, we can do the formula 1 / 0.7 to work out how much 1 USD is worth in AUD. We can do the same for the data supplied; if one BTC is worth 30 ETH, we can work out how much 1 ETH -> BTC is by calculating 1 / 30. This was used to calculate the value of the inverse trades when importing the data.
Through this implementation, all data is bidirectional, seemingly making having a directed graph redundant. However, as the inverse trades contain different information to the original trade, we store this information separately as a new trade; having all edges in the graph be unique.

File Input / Output - assetData.json

I decided that the assetdata.json file would be the basic file on which the initial graph would be constructed upon. The assetdata file, in the form of a .json (Javascript Object Notation), specifies trades within the graph; including the BASE asset (origin) and QUOTE asset (destination).

Figure 1 - Information within the unmodified assetdata.json file

To parse this data, I used a FileIO BufferedReader to read the entire line, and then various regex commands to remove all the unnecessary information from the imported line; such as the timezone, all information within arrays, and the “]” “[“ characters.

The end result of all this regex and parsing is a heaving modified .json String in which all individual elements (Trades) are separated by the “{“ or “}” characters and each individual point of data within those elements (“symbol”, “baseAsset”...) are separated by commas.

I then created a method JsonParser.parseAssetData() which would import this modified line and iterate through every element. For each valid (non blank) element, it would retrieve the “symbol”, “baseAsset” and “quoteAsset” which would be used to create new edges and nodes within a graph.

“Symbol” - The name / label of the trade (DSAGraphEdge.label)
“baseAsset” - The origin/”from” of the trade (DSAGraphEdge.from)
“quoteAsset” - The destination/”to” of the trade (DSAGraphEdge.to)

If the origin / destination of the trade had not yet been added to the graph, it was added when creating the edge.

JsonParser

I chose this approach to FileIO as I felt that creating a custom Json parser for this algorithm would be more useful than using an external parser. Creating a custom parser allows me significantly more control over how the data is created, stored and parsed. It also allows me to easily modify which elements I want to extract from the dataset and how I want to extract that data. In the case of the assetdata file, the custom json parser created the trade for any imported file, but would then also create an inverse trade which would also be stored in the graph.

File Input / Output - tradedata.json + asset_info.csv

Importing the file tradedata.json was done in an extremely similar fashion to assetdata.json, however the nature of the file bought restrictions.

The file contains a large amount of data for specific trades, including the count, volume, average price, ect… However, the tradedata.json does not specify the base asset and quote assets in regards to trades, only the LABEL of the trade. This means that we cannot construct trades from

this file, we can only add to pre-existing trades. Therefore, this file cannot be imported unless the assetdata file has already been imported.

Parsing the file was an extremely similar process to parsing the assetdata file; first regex commands to remove unnecessary data; split by “{“ “}”, and then split by commas. Individual elements/trades were imported into the TradeInfo object, in which data would be parsed and a new TradeInfo object would be constructed.

The TradeInfo object stores a range of values imported from the tradeinfo file, all stored in private class fields within the object. Once created, the JsonParser algorithm will find the corresponding trade that already exists within the graph (Using the label of the trade as the key in a hash function) and sets the VALUE of the trade to be the newly constructed TradeInfo object. This implementation of importing the values into the graph makes accessing the values simple, as they are stored within the trades as a class field.

Inside the graph exists an Object named Metadata. This object stores general information relating to the graph, such as which files are imported and the names of the imported files. Once the tradeinfo file is imported, the graph’s metadata values are updated to represent that this file has been imported. Once changed, the user has access to more detailed analysis of trades within the graph, being able to order them by count, volume, price, ect....

asset_info.csv Is imported into the graph with a near identical implementation, except the file is already in .csv format, and the information is added into the graph vertices as a AssetInfo object. Metadata is also updated to reflect this file being imported. As .csv values are, as their name suggests, separated by commas, it was difficult to parse data which was formatted like “$400,200”, as this would split into “$400 and 200”. Using a regex command derived from Scott Robinson (2020), I was able to split this data appropriately and parse it into the graph.

File Input / Output - Incorrect Files

If the user is unfamiliar with the algorithm, it is likely that they will, at some point, mix up which files need to be imported in which order. To prevent the algorithm trying to parse one .json file as another, basic validation has been added to each FileIO function. The validation ensures that the file exists, and if the file does exist, it contains information specific to the file.
For example, the tradedata.json file should contain the String “”{"symbol":””, which is unique to the file. If the imported file does not contain this String, then the incorrect file has been imported.

Report Mode

In report mode, there is no user interactivity. The user simply specifies the files to import in the command line and the algorithm will attempt to read these files and perform analysis on the imported data.

As the assetdata file is required for the creation of the graph, if the specified assetdata is invalid, no data will be displayed to the screen, but instead an error message. If the files do get successfully imported, the algorithm will print out extremely basic information related to the

number of trades and assets in the graph. If additional files tradedata + asset_info.csv have been successfully imported, the algorithm will print out analysis on the top 10 most valuable assets / trades.

Interactive Mode

As specified earlier, the assetdata.json file is required to construct the graph. Therefore, when the user runs the program in interactive mode, they are required to import an assetdata file before any other options become available. Once the file is successfully imported, a range of options will be available to the user. All options are accessed through basic integers ranging 1-9, with the 0 digit being reserved for exiting menus. This is to make navigation of the program simple to understand and consistent. Furthermore, all user input is validated through the UserInterface class, preventing the user from inputting an integer out of range or a String instead of an integer. Most menu options have an additional sub-menu. Some of the options within these sub-menus cannot be accessed until additional files have been imported, for example, in the asset overview sub-menu, the user cannot order the assets by values unless an additional asset info file has been imported. This is to make the algorithm more user-friendly; as the user should know exactly what functions they have access to.

Finding asset / trade options on the menu work in a similar fashion. The user can select an asset / trade to display the information on, however, unless the appropriate additional file has been imported, only basic information will be displayed to the screen. As mentioned earlier, retrieving this data is on average an O(1) operation. If the data cannot be found in a graph, an error is displayed to the screen and the algorithm resumes as the main menu.

Asset Filter

Asset filter was achieved by having a class field within each vertex labeled “ignored”. If this boolean class field was set to true, then the vertex would be ignored in all pathfinding algorithms. This value is false by default, but can be changed by using the menu’s asset filter functionality. When accessed, the user will be prompted to enter an asset name, and then specify if they want to add or remove the asset from the filter. Once selected, the boolean class field is changed appropriately.

Potential Trade Paths

The trade paths option is access through the main menu of the algorithm. Once the user enters the find trade paths sub-menu, they are prompted to enter the name of two assets in the graph. The algorithm verifies that these assets exist, and if they do, it continues with the algorithm. If not, an error is shown to the screen.

The recursive pathfinding algorithm works similarly to a breadth-first search; in which once a node is visited, every adjacent node is then visited after. The algorithm imports a starting node and a destination node. The algorithm sets the starting node as the “current node”, gets every element adjacent to the current node, and then does a recursive call for each element adjacent to the current node, setting each visited node as the new “current node”. A linked list of all nodes visited in the current recursion stack is also passed and added to, to keep track of the

current route. Once the current node has become the destination node, we have completed a path.

When a path is completed, every vertex that was stored in the current linked list “route” is concatenated into a singular string, and then this string is added to an overall linked list which stores all valid routes. Once the algorithm is complete, this linked list will contain all valid routes from the source node to the destination node.

I have implemented a maximum recursion depth of 5 for the find all paths algorithm. With this recursion limit, there exists over 200,000 trades paths between the assets BTC and ETH, and this takes 2-3 seconds to process on my machine. Allowing the algorithm to go past recursion depth 5 will exponentially increase the number of trade paths available, exponentially increasing the processing time and memory usage of the algorithm. The user can select the recursion depth of the algorithm from anywhere between 2-5 when running the program, which correlates to the maximum length of the trading paths between two assets. I allowed the user to select their depth as the user may find having extremely large trading paths between two nodes seems impractical in real-life applications, and that they may only want to view trading paths which involve 3 or 4 assets.

I have also implemented the algorithm to, if an additional trading path exists, calculate the cumulative price change over a trading route. If the file exists, every trade will contain a price change percentage. Whenever a new node is visited, the value of this price change % between these two nodes is retrieved and added to the cumulate price change, which is then passed on to the next level of recursion. When a route is added, instead of just adding the route string to the linked list; the route string + the cumulative price change are created as a new RouteData object, and then this object is stored in the linked list.

RouteData was chosen to be a seperate class file rather than an inner class as it is used in the Menu class to calculate the most valuable paths.

Sorting Trade Paths

If an additional file is present, the user has the option to sort the trades by cumulative price change once all trades are calculated. If the user selects this option, then all of the trades stored in the linked list are inserted into a heap data structure, with the cumulative price change being the priority value. This data structure was chosen as heaps are able to maintain sorted order as items are inserted, and items are removed in descending sorted order. Therefore, if all elements are added to a heap, and then 10 elements are removed from the heap, the 10 elements removed will be the item with the 10 highest priority, or in this case, highest cumulative price change.

Sorting Assets / Trades

Sorting asset or trade data is done in a near identical way. All assets are added to a heap object, with the data we wish to sort by being chosen as the priority. All sorting algorithms were placed inside of class SortData. Functions within the class are near-identical; however due to

the nature of not being able to import functions / function pointers as method parameters in Java, a different function would have to be written for every sort, as seperate functions were used to retrieve data for each possible asset / trade data.

Serialization

Serialization was unable to be implemented in the algorithm. Whenever Serialization was attempted, the algorithm would throw a StackOverFlowException. I have assumed this is due to the serialization module storing the LinkedHashTable; where information is stored similar to a linked list. It is assumed that as the module tries to store the references to other nodes inside of a singular node, it will store those nodes, which will store its connected nodes, which will store its connected nodes ect… I attempted to create my own saveObject module, but with a lack of knowledge about serialization this was unsuccessful.

References:

- Curtin University, Data Structures and Algorithm Lecture Slides 1-8
- Scott Robinson, 2020, “Regex: Splitting by Character, Unless in Quotes” https://stackabuse.com/regex-splitting-by-character-unless-in-quotes/