Project Report: Solution to analyse crypto-currency trading data (Java Language)

Calmen Chia Kai Fong

Department of Computing Curtin University

Lecturer: **Mr. Terence Tan Peng Lian**

School of Electrical and Computer Engineering

Curtin University

Miri, Malaysia

ABSTRACT

Writing applications to analyse crypto-currency trading data in Java under Linux Terminal can be quite challenging which requires programmer to understand how Linux Environment works. Linux Terminal works in Command Line Interface, which enables the user to use the program only with inputting commands through the keyboard. In this report, we are developing a program which provides features to search for a specific crypto currency which can be found in Binance. Using command line interface, the program enables user to retrieve the data faster as the option is selected through numbering system (0-9). The data for the crypto currencies are stored in JSON file format, which is a format that allow the ease for data retrieval from a server. There are two main JSON files in this project, which are assetFile.json and Trade.json. assetFile.json stores the data for all assets and the connections between them and Trade.json stores the data for different trades between two different assets. The project uses JSON Parser to retrieve the data from each of the .json file

and storing it in an array data structure, this data structure will then be connected to each other through another data structure which is known as graph. As a result, all data for crypto currencies are stored in a graph that will enable user to search for a specific assets and trades, showing every possible trade path and perform analysis on the data retrieved. This project is aimed for crypto currency traders ranged from daily trading to position trading style.

Contents

1. Information for Use

1.1 Introduction

This program was designed to show the analysis of crypto-currency trading data. It provides features in two different modes. The first mode is interactive mode and the second is report mode. In interactive mode, the program will run by displaying options as the menu to the user to choose.

There are eight options for the user to choose in interactive mode, the first options is load data, when user choose this option, the program will then prompt the user to choose to load the asset data, trade data or serialised a data. A serialised data is saving the complete graph into graph.dat which will allow the program to retrieve asset data and trade data directly from the option without loading asset and trade data. For option two, the program will prompt the user to enter an asset label, the program will then verify if the label is valid. If valid, the program will return the label along with its precisions. Otherwise, an error message will be displayed to user. For option three, the program will prompt user to input a specific trade labels such as "ETHBTC", which indicates the trade from ETH to BTC. If the trade label is valid, the program will return the trade data. Otherwise, an error message will be displayed to user. In option four, the program will prompt user to input a source asset and a destination asset. Once the assets are inputted, the program will then show all possible pathway to convert source to destination. For example, if source was "ETH" and destination was "BTC", the program will show all pathway to convert ETH to BTC. Option five will provides user a filtering feature. In this case, user will be prompted to enter the assets that they would like to ignore. Once they have inputted the assets to be ignored, the program will then output all asset data except the asset chosen to be ignored. The program will also provide another option for include. For example, if user would like to view only a certain number of assets, then the program will display only all the assets inputted by user. Asset overview in option six will display all the assets in the graph, along with which assets can it be converted from and which assets can it be converted to. For option seven trade overview, sorting of the object will be done and display the top ten price value, top ten volume value and top ten count value. In option eight, the program will provide a feature to save the graph in a data file so that the data can be retrieved later in sub-menu of option one, which is loading serialised data. For the last option nine, the program will then stop and exit from the terminal.

In the report mode, the program will display a broad analysis of the entire crypto currencies. The program will show the number of assets and number of trades available in the crypto currencies that was analyse in this project. The program will then show every asset in the graph and the
respective assets that is connected to it (Adjacent Asset).

1.2 Installation

Requirements:

- json-20200518.jar is required to allow the usage of JSON Parser
- junit-4.10.jar is required to allow writing and running JUnit test class
- Stack size of 512MB is required to run the program to avoid any crashes caused by stack overflow error during serialisation.

1.3 Terminology / Abbreviations

- Asset
- Trade
- cryptoGraph
- Serialised
- Filter

1.4 Walkthrough

Interactive Mode

Features	Status
Load data: • Asset data • Trade data • Serialised data	 Complete Complete Complete
Find and display asset	Complete
Find and display trade details	Complete
Find and display potential trade paths	Complete
Set asset filter	Complete
Asset overview	Complete
Trade overview	Complete
Save data (serialised)	Complete
Exit	Complete

Report Mode

Features	Status
Show number of assets	Complete
Show number of trades	Complete
Show highest price of trades	Complete
Show adjacency assets	Complete

1.5 Future Work

Some features which are suggested to be implemented to enhance the program are visualisation and plotting, finding the profitable trading paths from source to destination, specific filter categories, most profitable trades, calculating standard deviations for an assets and other common crypto currency trading indicators. For visualisation and plotting, a graph is plotted based on the price of a certain asset for the past periods of time which will be selected by the user. The plotted graph will also indicate the uptrend, sideways trend and down trend events of the asset which will inform the trader that the asset is going up making, trading in a horizontal channel or going down making respectively (Swiss Borg 2020). Providing features that enable user to analyse the price of the graph for a specific time helps to increase the usability of this program to different types of user or traders. For example, daily trader tends to do analysis on a short-term time frame to decide on buying and selling whereas swing traders and position traders tends to do analysis on weeks or sometimes months (Investopedia 2020). Finding profitable trading paths from selected asset to another asset also helps traders in analysing and decision making to buy or sell a particular trade. Specific filter categories such as filter by volume, highest price, lowest price will improve usability of the entire program which will enable trader to be more productive on analysing the crypto currency. Another feature that will be helpful in the program is the standard deviation of the trade would be providing standard deviation of a specific asset. This will be beneficial for swing and position traders as by analysing the standard deviation for the past thirty days, one can measure the risk of a particular trade (Airbag AI 2019). The last feature that is suggested to be implemented is to include common indicators that traders used in analysing the trend event of a trade. One of the common indicators used by traders is Relative Strength Index (RSI), which is a convenient indicator to most traders to identify if an asset is oversold or overbought. It is ranging from zero to one hundred, when the value exceeds seventy, it indicates the asset is overbought and giving a sell indication. But when the value decreases below thirty, it indicates that the asset is oversold and is giving a buy indication (Haas Online 2020). Another indicator that should be included is Simple Moving Average (SMA), which is an indication of an asset price over time. By analysing the trade activities, when the short-term average crosses above a long-term average will indicate the trader that it is a buy opportunity. SMA is also customisable to fit long or short-term horizons which makes it a useful indicator to be included (Haas Online 2020).

2. Traceability Matrix

		Requirements	**Design/Code**	**Test**
1	**Load Data** • **Asset Data** • **Trade data** • **Serialised data**	System displays usage if called without arguments.	cryptoGraph.main()	UI Test: Live demonstration
		System displays interactive menu with "-i" argument.	cryptoGraph.interactiveMode()	UI Test: Live demonstration
		In interactive mode, user enters command and system responds.	cryptoGraph.interactiveMode()	UI Test: Live demonstration
		System returns appropriate error messages from wrong	UserInterface.userInput()	UI Test: Live demonstration

		user input.		
		Load the Base Asset data (assetFile.json)	ParseJSON.readBaseAsset()	UnitTestAsset.testReadAsset()
		Load the Quote Asset data (assetFile.json)	ParseJSON.readQuoteAsset()	UnitTestAsset.testReadAsset()
		Load Edge data	ParseJSON.readEdge()	UnitTestEdge.testReadEdge()

		Load Asset Info (asset_info.csv)	cryptoGraph.readAssetInfo()	UnitTestCryptoGraph.testReadAssetInfo()
				UnitTestAssetInfo.testLength()
				Unit TestAssetInfo.testDataRead()

		Load Filters data to help in creating Edge	ParseJSON.readEdge()	UnitTestEdge.testReadFilters()

		Store the data retrieved in a graph data structure	cryptoGraph.storeGraph()	UnitTestDSAGraph.testCreatingVertex()
				UnitTestDSAGraphEdge.testLabel()
				UnitTestDSAGraphEdge.testFromTo()
				UnitTestDSAGraphEdge.testDirectedFalse()
				UnitTestDSAGraphEdge.testDirectedTrue()
				UnitTestDSAGraphEdge.testVisitedTrue()
				UnitTestDSAGraph.testAddingEdge()
				UnitTestDSAGraph.testAddingEdgeError()
				UnitTestDSAGraph.testVisited()

				UnitTestCryptoGraph.testStoreGraph()
		Load the trade data	ParseJSON.readTrade()	UnitTestTrade.testReadTrade()
		Load the serialised data	Serialisation.load()	UnitTestSerialisation.testLoad()
2	**Find and display asset**	Prompt user for asset symbol (eg: “BTC”)	UserInterface.userInput	UnitTestUserInterface.main()
		Convert any case letter to uppercase (case insensitive)	cryptoGraph.toUpper()	UnitTestCryptoGraph.testToUpper()

		Retrieve asset from the graph data structure	DSAGraph.getVertex()	UnitTestDSAGraph.testHasVertex()
				UnitTestDSAGraph.testGetVertex()
				UnitTestDSAGraph.testGetVertexError()

3	**Find and display trade**	Prompt user for trade symbol (eg: “ETHBTC”)	UserInterface.userInput	UnitTestUserInterface.main()
		Convert any case letter to uppercase (case insensitive)	cryptoGraph.toUpper()	UnitTestCryptoGraph.testToUpper()

		Retrieve trade from trade array	cryptoGraph.findTrade()	UnitTestCryptoGraph.testFindTrade()
4	**Find and display potential paths**	Prompt user for source and destination symbol	UserInterface.userInput	UnitTestUserInterface.main()
		Convert any case letter to uppercase (case insensitive)	cryptoGraph.toUpper()	UnitTestCryptoGraph.testToUpper()

		Display all the potential paths	cryptoGraph.displayPotentialPath()	DSAGraph.testGetAdjacent()
				UnitTestDSAGraph.testDFSStartEnd()
				UnitTestDSAGraph.testDFSStartMiddle()
				UnitTestDSAGraph.testDFSMiddleMiddle()

5	**Set asset filter**	Prompt user for the mode of filter (1. Include, 2. Exclude)	UserInterface.userInput	UnitTestUserInterface.main()
		Include Filter	cryptoGraph.includeFilter()	UI Test: Live Demonstration
		Exclude Filter	cryptoGraph.excludeFilter()	UI Test: Live Demonstration

6	**Asset Overview**	Get all the asset(s) that can be converted to a particular asset	DSAGraph.getParentVertex()	DSAGraph.testGetParentVertex()
				DSAGraph.testGetAdjacent()

		Get all asset(s) that the particular asset can be converted to	DSAGraph.getChildVertex()	DSAGraph.testGetChildVertex()
				DSAGraph.testGetAdjacent()

7	**Trade Overview**	Convert price from trade array to double array	cryptoGraph.getPriceArr()
			Sort.insertionSortDouble()
				UnitTestSort.testSortDouble()

		Convert volume from trade array to double array	cryptoGraph.getVolumeArr()
			Sort.insertionSortDouble()
				UnitTestSort.testSortDouble()

		Convert count from trade array to long array	cryptoGraph.getCountArr()
			Sort.insertionSortLong()
				UnitTestSort.testSortLong()

8	**Save data (Serialised)**	Save the graph	Serialisation.save()	UnitTestSerialisation.testSave()

		data structure created into a file		
9	**Exit**	Stop the interactive mode and exit	cryptoGraph.interactiveMode()	UI Test: Live Demonstration
10	**Report Mode**	System displays items in report mode with "-r" argument and file(s) required	cryptoGraph.reportMode()	UI Test: Live Demonstration
		Display the number of assets analysing	DSAGraph.getCount()	UnitTestDSAGraph.testGetCount()
		Display the number of connections between assets	DSAGraph.getEdgeCount()	UnitTestDSAGraph.testGetEdgeCount()
		Display the highest price of trade	Sort.insertionSortDouble()	UnitTestSort.testSortDouble()
		Display the asset(s) that each	DSAGraph.displayAsList()	UnitTestDSAGraph.testDisplayAsList()

		asset can be converted to		
11	**Graph Operation**	Check if connection (edge) existed	N/A	UnitTestDSAGraph.testHasEdge()
		Check the value of the vertex	N/A	UnitTestDSAGraph.testVertexValue()
		Get the connection (edge) object	N/A	UnitTestDSAGraph.testGetEdge()
		Check if two assets (vertex) is connected	N/A	UnitTestDSAGraph.testIsAdjacent()
		Check for connection label (graph edge label)	N/A	UnitTestDSAGraph.testGraphEdgeLabel()
		Check toString for graph edge	N/A	UnitTestDSAGraph.testGraphEdgeToString()

12	**DSAGraphVertex**	Check for visited status of each vertex	N/A	UnitTestDSAGraphVertex.testVisitedTrue()
				UnitTestDSAGraphVertex.testVisitedFalse()

		Check for label of vertex	N/A	UnitTestDSAGraphVertex.testGetLabel()
				UnitTestDSAGraphVertex.testGetValue()

		toString of vertex	N/A	UnitTestDSAGraphVertex.testToString()
		Adjacency vertices of the vertex	N/A	UnitTestDSAGraphVertex.testGetAdjacent()
		Number of adjacent of a vertex	N/A	UnitTestDSAGraphVertex.testGetSize()
13	**DSALinkedList**	Insert First	N/A	UnitTestDSALinkedList.main()
		Insert Last	N/A	UnitTestDSALinkedList.main()
		Peek First	N/A	UnitTestDSALinkedList.main()
		Peek Last	N/A	UnitTestDSALinkedList.main()
		Remove First	N/A	UnitTestDSALinkedList.main()
		Remove Last	N/A	UnitTestDSALinkedList.main()
		Iterator	N/A	UnitTestDSALinkedList.main()
		Iterator next	N/A	UnitTestDSALinkedList.main()
		Iterator hasNext	N/A	UnitTestDSALinkedList.main()
14	**DSAStack**	Insert (Push)	N/A	UnitTestStack.testStackPush()

		Remove (pop)	N/A	UnitTestStack.testStackPop()
				UnitTestStack.testStackPopEmpty()
				UnitTestStack.testStackOverPop()

		Get (Top)	N/A	UnitTestStack.testStackTop()
				UnitTestStack.testStackTopPop()
				UnitTestStack.testStackTopPopsEmpty()
				UnitTestStack.testStackTopEmpty()

15	**DSABinarySearchTree**	Insert	N/A	UnitTestDSABinarySearchTree.testInsert()
				UnitTestDSABinarySearchTree.testInsertInvalid()
				UnitTestDSABinarySearchTree.testInsertExistingKey()

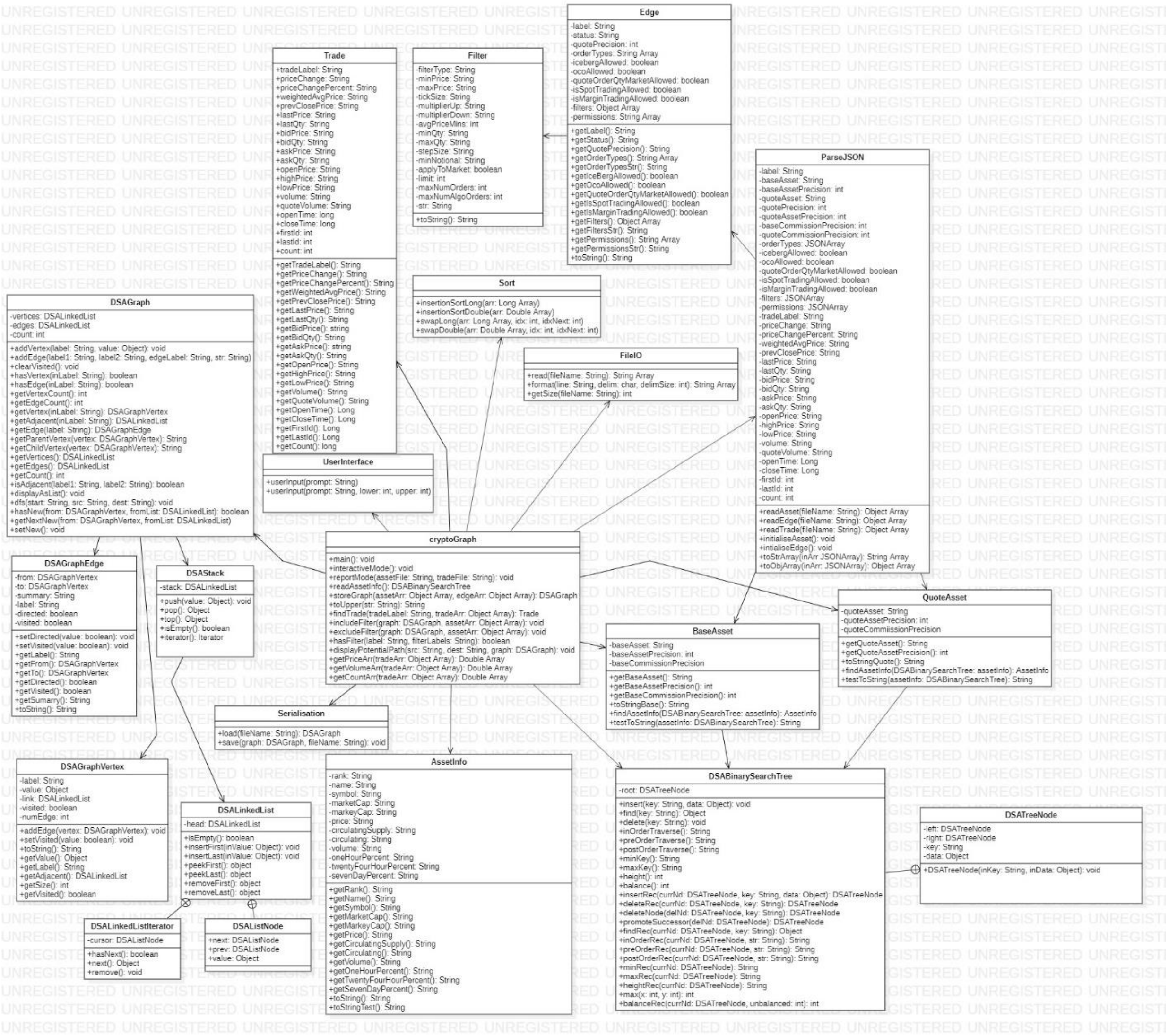
		Find	N/A	UnitTestDSABinarySearchTree.testFind()
				UnitTestDSABinarySearchTree.testNotFound()

		Traverse	N/A	UnitTestDSABinarySearchTree.testTraverseInOrder()
				UnitTestDSABinarySearchTree.testTraversePreOrder()
				UnitTestDSABinarySearchTree.testTraversePostOrder()

		Min	N/A	UnitTestDSABinarySearchTree.testMin()
		Max	N/A	UnitTestDSABinarySearchTree.testMax()
		Height	N/A	UnitTestDSABinarySearchTree.testHeight()
		Balance	N/A	UnitTestDSABinarySearchTree.testBalance()

		Remove	N/A	UnitTestDSABinarySearchTree.testDeleteOneChild()
				UnitTestDSABinarySearchTree.testDeleteTwoChild()
				UnitTestDSABinarySearchTree.testDeleteNoChild()

3. Class Diagram


Edge
-label: String
-status: String
-quotePrecision: int
-orderTypes: String Array
-icebergAllowed: boolean
-ocoAllowed: boolean
-quoteOrderQtyMarketAllowed: boolean
-isSpotTradingAllowed: boolean
-isMarginTradingAllowed: boolean
-filters: Object Array
-permissions: String Array
+getLabel(): String
+getStatus(): String
+getQuotePrecision(): String
+getOrderTypes(): String Array
+getOrderTypesStr(): String
+getIceBergAllowed(): boolean
+getOcoAllowed(): boolean
+getQuoteOrderQtyMarketAllowed(): boolean
+getIsSpotTradingAllowed(): boolean
+getIsMarginTradingAllowed(): boolean
+getFilters(): Object Array
+getFiltersStr(): String
+getPermissions(): String Array
+getPermissionsStr(): String
+toString(): String
Trade
+tradeLabel: String
+priceChange: String
+priceChangePercent: String
+weightedAvgPrice: String
+prevClosePrice: String
+lastPrice: String
+lastQty: String
+bidPrice: String
+bidQty: String
+askPrice: String
+askQty: String
+openPrice: String
+highPrice: String
+lowPrice: String
+volume: String
+quoteVolume: String
+openTime: long
+closeTime: long
+firstId: int
+lastId: int
+count: int
+getTradeLabel(): String
+getPriceChange(): String
+getPriceChangePercent(): String
+getWeightedAvgPrice(): String
+getPrevClosePrice(): String
+getLastPrice(): String
+getLastQty(): String
+getBidPrice(): string
+getBidQty(): String
+getAskPrice(): string
+getAskQty(): String
+getOpenPrice(): String
+getHighPrice(): String
+getLowPrice(): String
+getVolume(): String
+getQuoteVolume(): String
+getOpenTime(): Long
+getCloseTime(): Long
+getFirstId(): Long
+getLastId(): Long
+getCount(): long
Filter
-filterType: String
-minPrice: String
-maxPrice: String
-tickSize: String
-multiplierUp: String
-multiplierDown: String
-avgPriceMins: int
-minQty: String
-maxQty: String
-stepSize: String
-minNotional: String
-applyToMarket: boolean
-limit: int
-maxNumOrders: int
-maxNumAlgoOrders: int
-str: String
+toString(): String
Sort
+insertionSortLong(arr: Long Array)
+insertionSortDouble(arr: Double Array)
+swapLong(arr: Long Array, idx: int, idxNext: int)
+swapDouble(arr: Double Array, idx: int, idxNext: int)
FileIO
+read(fileName: String): String Array
+format(line: String, delim: char, delimSize: int): String Array
+getSize(fileName: String): int
ParseJSON
-label: String
-baseAsset: String
-baseAssetPrecision: int
-quoteAsset: String
-quotePrecision: int
-quoteAssetPrecision: int
-baseCommissionPrecision: int
-quoteCommissionPrecision: int
-orderTypes: JSONArray
-icebergAllowed: boolean
-ocoAllowed: boolean
-quoteOrderQtyMarketAllowed: boolean
-isSpotTradingAllowed: boolean
-isMarginTradingAllowed: boolean
-filters: JSONArray
-permissions: JSONArray
-tradeLabel: String
-priceChange: String
-priceChangePercent: String
-weightedAvgPrice: String
-prevClosePrice: String
-lastPrice: String
-lastQty: String
-bidPrice: String
-bidQty: String
-askPrice: String
-askQty: String
-openPrice: String
-highPrice: String
-lowPrice: String
-volume: String
-quoteVolume: String
-openTime: Long
-closeTime: Long
-firstId: int
-lastId: int
-count: int
+readAsset(fileName: String): Object Array
+readEdge(fileName: String): Object Array
+readTrade(fileName: String): Object Array
+initialiseAsset(): void
+intialiseEdge(): void
+toStrArray(inArr JSONArray): String Array
+toObjArray(inArr: JSONArray): Object Array
DSAGraph
-vertices: DSALinkedList
-edges: DSALinkedList
-count: int
+addVertex(label: String, value: Object): void
+addEdge(label1: String, label2: String, edgeLabel: String, str: String)
+clearVisited(): void
+hasVertex(inLabel: String): boolean
+hasEdge(inLabel: String): boolean
+getVertexCount(): int
+getEdgeCount(): int
+getVertex(inLabel: String): DSAGraphVertex
+getAdjacent(inLabel: String): DSALinkedList
+getEdge(label: String): DSAGraphEdge
+getParentVertex(vertex: DSAGraphVertex): String
+getChildVertex(vertex: DSAGraphVertex): String
+getVertices(): DSALinkedList
+getEdges(): DSALinkedList
+getCount(): int
+isAdjacent(label1: String, label2: String): boolean
+displayAsList(): void
+dfs(start: String, src: String, dest: String): void
+hasNew(from: DSAGraphVertex, fromList: DSALinkedList): boolean
+getNextNew(from: DSAGraphVertex, fromList: DSALinkedList)
+setNew(): void
UserInterface
+userInput(prompt: String)
+userInput(prompt: String, lower: int, upper: int)
cryptoGraph
+main(): void
+interactiveMode(): void
+reportMode(assetFile: String, tradeFile: String): void
+readAssetInfo(): DSABinarySearchTree
+storeGraph(assetArr: Object Array, edgeArr: Object Array): DSAGraph
+toUpper(str: String): String
+findTrade(tradeLabel: String, tradeArr: Object Array): Trade
+includeFilter(graph: DSAGraph, assetArr: Object Array): void
+excludeFilter(graph: DSAGraph, assetArr: Object Array): void
+hasFilter(label: String, filterLabels: String): boolean
+displayPotentialPath(src: String, dest: String, graph: DSAGraph): void
+getPriceArr(tradeArr: Object Array): Double Array
+getVolumeArr(tradeArr: Object Array): Double Array
+getCountArr(tradeArr: Object Array): Double Array
DSAGraphEdge
-from: DSAGraphVertex
-to: DSAGraphVertex
-summary: String
-label: String
-directed: boolean
-visited: boolean
+setDirected(value: boolean): void
+setVisited(value: boolean): void
+getLabel(): String
+getFrom(): DSAGraphVertex
+getTo(): DSAGraphVertex
+getDirected(): boolean
+getVisited(): boolean
+getSumarry(): String
+toString(): String
DSAStack
-stack: DSALinkedList
+push(value: Object): void
+pop(): Object
+top(): Object
+isEmpty(): boolean
+iterator(): Iterator
QuoteAsset
-quoteAsset: String
-quoteAssetPrecision: int
-quoteCommiissionPrecision
+getQuoteAsset(): String
+getQuoteAssetPrecision(): int
+toStringQuote(): String
+findAssetInfo(DSABinarySearchTree: assetInfo): AssetInfo
+testToString(assetInfo: DSABinarySearchTree): String
BaseAsset
-baseAsset: String
-baseAssetPrecision: int
-baseCommissionPrecision
+getBaseAsset(): String
+getBaseAssetPrecision(): int
+getBaseCommissionPrecision(): int
+toStringBase(): String
+findAssetInfo(DSABinarySearchTree: assetInfo): AssetInfo
+testToString(assetInfo: DSABinarySearchTree): String
Serialisation
+load(fileName: String): DSAGraph
+save(graph: DSAGraph, fileName: String): void
DSAGraphVertex
-label: String
-value: Object
-link: DSALinkedList
-visited: boolean
-numEdge: int
+addEdge(vertex: DSAGraphVertex): void
+setVisited(value: boolean): void
+toString(): String
+getValue(): Object
+getLabel(): String
+getAdjacent(): DSALinkedList
+getSize(): int
+getVisited(): boolean
DSALinkedList
-head: DSALinkedList
+isEmpty(): boolean
+insertFirst(inValue: Object): void
+insertLast(inValue: Object): void
+peekFirst(): object
+peekLast(): object
+removeFirst(): object
+removeLast(): object
DSALinkedListIterator
-cursor: DSAListNode
+hasNext(): boolean
+next(): Object
+remove(): void
DSAListNode
+next: DSAListNode
+prev: DSAListNode
+value: Object
AssetInfo
-rank: String
-name: String
-symbol: String
-marketCap: String
-markeyCap: String
-price: String
-circulatingSupply: String
-circulating: String
-volume: String
-oneHourPercent: String
-twentyFourHourPercent: String
-sevenDayPercent: String
+getRank(): String
+getName(): String
+getSymbol(): String
+getMarketCap(): String
+getMarkeyCap(): String
+getPrice(): String
+getCirculatingSupply(): String
+getCirculating(): String
+getVolume(): String
+getOneHourPercent(): String
+getTwentyFourHourPercent(): String
+getSevenDayPercent(): String
+toString(): String
+toStringTest(): String
DSABinarySearchTree
-root: DSATreeNode
+insert(key: String, data: Object): void
+find(key: String): Object
+delete(key: String): void
+inOrderTraverse(): String
+preOrderTraverse(): String
+postOrderTraverse(): String
+minKey(): String
+maxKey(): String
+height(): int
+balance(): int
+insertRec(currNd: DSATreeNode, key: String, data: Object): DSATreeNode
+deleteRec(currNd: DSATreeNode, key: String): DSATreeNode
+deleteNode(delNd: DSATreeNode, key: String): DSATreeNode
+promoteSuccessor(delNd: DSATreeNode): DSATreeNode
+findRec(currNd: DSATreeNode, key: String): Object
+inOrderRec(currNd: DSATreeNode, str: String): String
+preOrderRec(currNd: DSATreeNode, str: String): String
+postOrderRec(currNd: DSATreeNode, str: String): String
+minRec(currNd: DSATreeNode): String
+maxRec(currNd: DSATreeNode): String
+heightRec(currNd: DSATreeNode): String
+max(x: int, y: int): int
+balanceRec(currNd: DSATreeNode, unbalanced: int): int
DSATreeNode
-left: DSATreeNode
-right: DSATreeNode
-key: String
-data: Object
+DSATreeNode(inKey: String, inData: Object): void

4. **Class Description**

Class	**Description**
ParseJSON	This is a static class that will provide the data for the entire program. The Trade data and the Asset data are stored in Trade.json and assetFile.json respectively. Therefore, we will need a JSON Parser in this case to read the file which is in JSON format. The purpose of this class is to read the Asset Data, Edge Data and the Trade Data by storing them into each object array and return it to the caller method (In cryptoGraph). The program will then use the array retrieved to create the graph for the entire project to do all the analysis work. Which is our purpose of this project assignment. ParseJSON.readAsset(String fileName) will return us the data for asset (vertex in graph) whereas ParseJSON.readEdge(String fileName) will return us the data for edge (connection in graph) and ParseJSON.readTrade(String fileName) will return us the trade data for analysis (Trade overview and report mode).
Trade	This class is created to handle all the trade data. This is because each of the trade data consists of a lot of data (21 data) in Trade.json. Therefore, it is easier if we

	could organise these data into a class, and then obtain each of the data (field) using object-oriented programming techniques. The object for each trade data is created through the constructor of the Trade class by using all the fields retrieved from Trade.json and every field in trade class has its own getter (Accessor) so that the field data for each object can be retrieved by calling the getter of the field. Once the trade object is created, store it in the object array (Object[] tradeArr) so that we can store all the trade data into one array, and each element in the tradeArr represents a trade object which has all the fields required for the trade data.
BaseAsset and QuoteAsset	These class is used to handle the asset data from assetFile.json. This is because the Asset data alone is not a single field in assetFile.json. There are 6 data in the assetFile.json. Hence, to organise the data, we will need to create the class for Asset to store all the fields in this class. The data can be retrieved by calling the getter (Accessor) of the Asset class using the object of asset. The object of Asset is created in ParseJSON.readAsset(String fileName). Over here, the data from the file is retrieved and used to create the asset

	object. Every time an object of asset is created, we will store it in object array of asset (assetArr). Therefore, we will now have the asset data stored in the array and it can then be used to create the vertex of the graph.
Edge	This is the class to store all the edge data. The edge data consists 10 data. In this case, we need to create a class to store all the edge fields and retrieve it by edge object. The edge object is first created using the fields retrieved from assetFile.json in ParseJSON.readEdge(String fileName). When each of the edge object is created, it is then stored into object array (edgeArr). By using the array, we then can retrieve each of the fields of edge using the element in the array as the element in the array is an edge object.
Filter	This class is created to handle the filter data. Which is one of the fields in Edge Data. Unlike other fields in the Edge Data, filter data itself has a lot of fields data (15 data). Therefore, it is better to organise the data into one class and create an object to store it under the filter fields in asset data. Hence, when we would like to retrieve specific data fields in filter, we could retrieve it by calling the getter (Accessor)

	using the filter object. All the fields for filter object in each asset data are stored in an object array and this object array is returned as filter object as one of the fields of Edge. Since Filter is a part of Edge data, the filter class can also be included as private inner class in the Edge class instead of a separate class. But in this project, we are retrieving each fields of filter data in ParseJSON.toObjArray(JSONArray inArr). So, we will need the filter class to be public in this case.
DSAGraph	This is the class for the data structure that store the whole data assetFile.json and Trade.json from the assignment (Asset Data and Edge Data). DSAGraph contains the methods that enables us to create Graph Data Structure for the entire crypto currency asset. Each element in the assetArr and edgeArr is passed to the graph representing vertex and edge respectively. The asset will be the vertex and the edge will represent the connection between two vertices. Asset is created by a list of vertices, which is a field of DSAGraph and Edge is created by another class which is DSAGraphEdge and is then stored in edges list, which is a field of DSAGraph too. DSAGraph also contains the method to

	retrieve a particular asset (vertex) and method for traversal to search for the trade paths (Depth First Search). This class is created to help us to retrieve data from DSAGraphEdge and DSAGraphVertex through association relationship. At the same time, we are allowed to create all the vertices (created through DSAGraphVertex) and connection between them (created through DSAGraphEdge).
DSAGraphEdge	This is the class for graph edge which shows the connection between two vertices. DSAGraphEdge will handle the edge data between two assets that connects them. Every edge object from edgeArr will be passed and used to create DSAGraphEdge object which will then be stored in the list. DSAGraphEdge will be helpful for us to identify the relationship between two vertices, which is “from Vertex” and “to Vertex”. Each path for two vertex has a unique String representation which is edgeLabel. edgeLabel is basically the combination of the label of “from Vertex” and “to Vertex” without any spaces in between. By using edgeLabel, we could get the edge object and retrieves the edge data. Therefore, we could get the information of each asset and their relationship with other

	assets (Assets that they can be converted from and Assets they can be converted to). This could be private inner class for DSAGraph, but we have made it separated public here in the assignment so that we can retrieve the edge data without the graph object. This is because we would like to simplify the code in DSAGraph and not making DSAGraph.java looks more complicated in this sense to maintain readability.
DSAGraphVertex	The class is designed to handle the data of the different assets (305 different asset). The assets are stored as vertex in the entire graph data structure. DSAGraphVertex would allow us to retrieve the vertex information using the object of DSAGraphVertex and each of the DSAGraphVertex object is represented by a unique string label. The label is the abbreviation of a particular crypto currency asset. For example, the abbreviation for Bitcoin is BTC, Ethereum is ETH and etc. The purpose of DSAGraphVertex is to enable us to deal with each fields of a single asset data. DSAGraphVertex could be done by declaring it as private inner class in DSAGraph. But making it public here so that we could use the accessor in it

	from cryptoGraph to retrieve the information of the asset data. Otherwise if it was put in DSAGraph, the file will be filled with extra code and it is not ideal for readability and the purpose of object-oriented programming which is modularity for easier troubleshooting as well.
DSAStack	This is an ADT that performs the “Last-In First-Out” behavior. It is used in the DFS (Depth-First Search) method in the program that allows us to travel and find for all the possible paths (Option 4 in interactive mode).
DSALinkedList	This is the class for Linked List data structure. This is the most used data structure in the program. Which is used by four classes through association relationship, which are DSAGraph, DSAGraphVertex, DSAGraphEdge and DSAStack. In DSAGraph, LinkedList is used as “vertices” to represents the number of assets available for the program. In DSAGraphVertex, there is another linked list for each vertex to represents the adjacency list of a particular vertex. For example, the list of vertices that a particular vertex is connecting to. In DSAGraphEdge, the linked list is used to store all the edge we have between two vertices. This can be

	stored as array instead of linked list but we will need to insert element into the connection of list every time so is more efficient to use insertFirst with O(1) Big-O notation. We would also save more operation in creating a duplicated array and copying all data from the initial array which will takes up a lot of operations (computational expensive). In DSAStack, Linked List is suitable to be used since it has the insertFirst and removeFirst method to achieve the "Last-In First-Out" behavior. Array can be used in DSAStack too but inserting and removing in the stack is computational expensive in this case since we will need to create another duplicate array with extra size whereas it can be done with O(1) notation in Linked List.
DSALinkedListIterator	This class allow the iterator behavior for the linked list. The purpose of having iterator in linked list allow us to traverse through a linked list. In java, there is for-each loop which reserves the same features as default for loop, which allow us to traverse through element of array. Whereas in for-each loop, it allows us to traverse through every node in the linked list. This is a private inner class for DSALinkedList

	to limit its usage out of this class (Encapsulation).
DSAListNode	This class will allow us to create each data stored in the linked list. In other words, it is equivalent to the element in the index of the array. Each List Node will store the reference of its next node and its previous node.
Serialisation	This class enable us to store the graph data structure, which is created by all other data from the assetFile.json into another file (graph.dat). This is done in Serialisation.save(DSAGraph graph, String fileName). Other than saving, it also allows us to retrieve the graph data structure so that we could continue to where we left off from previous running of the program. This could be done in Serialisation.load(String fileName).
UserInterface	This is the class that handles the user input in the entire program. Anything throughout the program which required user input data will be done by this class. Exception handling is also done in this class to ensure that the user is inputting valid data without crashing the program.
Sort	This class handle the sorting part of the program. There are two methods in this

	class which handle the sorting for long data type and double data type.
cryptoGraph	This is the driver class for the entire program. This is where the main method located in the program. The class serves a purpose to provide menu for user and calling other methods from different classes to perform their tasks and return the value. The returned value is then output in this cryptoGraph class. It also serves the purpose to create the graph data structure for the entire program through cryptoGraph.storeGraph(Object[] assetArr, Object[] edgeArr).
FileIO	This is a static class that deals with reading the asset_info.csv and format it in String array. This is because the csv file contains data that has ‘,’ but are being treated as one data field. This can be represented by “xx,xx,xx,xx” and this is being treated as one data instead of 4 data.
AssetInfo	This class is used to handle the fields for the newly added file (asset_info.csv). The class is provided with getters (Accessor) to retrieve each data fields by using object of AssetInfo
DSABinarySearchTree	This class is used to store all the data in the newly added file (asset_info.csv). This data structure will store all the asset information

	in AssetInfo Object. Then, the retrieval of the data is done by searching the symbol of the asset (key) and returned the object of AssetInfo which represents the data for a particular asset read from the asset_info.csv.

5. Justification

Class	Justification
ParseJSON	The data structure used to store all the data for asset, edge and trade is Array. The reason of using array instead of Linked List to store these data is because each data for asset, edge and trade respectively consists of a lot of fields (sub-data). Therefore, in term of memory complexity, we should choose Array instead of Linked List as Array consumes lower memory compare to Linked List. This is because Linked List will need to reserve more memory space for the reference of its next pointer and previous pointer (GeeksforGeeks 2020). Other than that, since the amount of data to be inserted from the file is known and consistent, we could declare the array of enough size according to the number of data. Hence, the array size is fixed and there is no need for us to expand it as the array size declared is enough for the data. Therefore, the O(1) Big-O Notation of Linked List in term of inserting and removing the item is not going to be useful for our purpose here as we do not need any removal operation.

DSAGraph	The entire crypto currency asset and edge data is stored in a graph data structure instead of a others such as tree or hash table is because in a tree data structure, there are only two child nodes can be stored under a parent node. But in this project, there are assets which can be converted to more than two assets. For example, BTC (Bitcoin) can be converted into sixteen different assets. Therefore, graph is suitable in this case to represents the relationship between vertices. On the other hand, hash table is inefficient to be used when we would like to represent all possible paths (Option 4 in Interactive mode). This is because we will need to do a linear search on the hash table to find for each adjacent vertex in the hash table whereas we could just display all the connection of the paths in graph using DFS (Depth First Search).
DSAGraphEdge	The object of DSAGraphEdge is stored in the linked list. This can be stored as array instead of linked list but since we will need to insert an element into the connection of list every time in cryptoGraph.storeGraph(), so it is more efficient to use insertFirst with O(1) Big-O notation. We would also save more

	operation to avoid creating a new array with larger size and copying all data from the existing array which will takes up a lot of operations (computational expensive). In this case, it will be O(n) of Big-O Notation if we were to store it in array.
DSAGraphVertex	The object of DSAGraphVertex is stored in a linked list instead of an Array. This is because if it was stored in the Array, we will first need to identify the length of the Array. Which will require us to traverse all the data in assetFile.json. Any unrepeated base asset in the label will be counted as new asset (Vertex). Therefore, we can create the Array we needed with the number of sizes we acquired from traversing the assetFile.json. Only then we can start storing the vertex in an array. In this case, the Big-O Notation for storing the vertex in an array would be O(n) since we need to traverse all the data first to initialise the array with the size return before storing the Vertex whereas for Linked List, we do not care about the size as Linked List is flexible in expanding and shrinking. Therefore, we could have Big-O Notation of O(1) in storing each of the vertex.

DSAStack	The Stack ADT was implemented by Linked List instead of Array because Linked List will gives us the O(1) Big-O Notation in insertion and also removal since we are implementing a Double and Doubly Linked List whereas for Array Implementation of Stack, every time an item is removed, we would need to create a new array and copy all items in the existing array to be stored in the newly created array. In this case, the implementation of Stack will result in an O(n) Big-O Notation for removal.
DSABinarySearchTree	DSABinarySearchTree is used to store the in the asset_info.csv for a retrieval of asset data. Binary Search Tree is used instead of Hash Table because when using hash table, the size could go up to 5227 due to the resize nature in hash table to avoid higher rate of collisions. While the actual data in asset_info.csv is only 2600, which means there are 2627 memory of array index is being wasted. Therefore, a binary search tree would provide us a memory efficient data structure in this case. In fact, unlike hash table, since binary search tree is referenced based, it will be no need to add more memory than it was supposed to have (StackOverflow 2010). In terms of

	speed performance, Binary Search Tree will always provide constant O(logN) time complexity in retrieving the data, while although Hash Table will provide time complexity O(1) on average, but it will be computational expensive when resizing happens. Resizing will cost us O(n) time complexity. Therefore, by analysing the space complexity and consistent time complexity that binary search tree could provide us, binary search tree will be more dominant than hash table in this case.
Graph traversal to find all possible paths	Depth First Search is chosen instead of Breath First Search because Depth First Search allow us to go in deeper into the graph until we search for the destination asset (vertex) while Breath First Search operates in going wider to the graph which may waste the paths that does not lead us to the destination asset (vertex). Depth First Search is also generally faster than Breath First Search (GeeksForGeeks 2020). Therefore, Depth First Search is the ideal algorithm that we will use here to search for source asset to destination asset in a large graph.

6. References

AI, Airbag. 2019. “How to measure Risk for Crypto Trading”. Medium. https://medium.com/@airbag.ai/how-to-measure-risk-for-crypto-trading-2eb738d1330e

Ganti, Akhilesh. 2019. “Position Trader”. Investopedia. https://www.investopedia.com/terms/p/positiontrader.asp#:~:text=The%20average%20time%20frames%20for,than%2010%20trades%20a%20year.

GeeksForGeeks. 2020. Linked List vs Array. https://www.geeksforgeeks.org/linked-list-vs-array/#:~:text=Elements%20are%20stored%20consecutively%20in,stored%20randomly%20in%20Linked%20lists.&text=The%20requirement%20of%20memory%20is,next%20and%20previous%20referencing%20elements.

GeeksForGeeks. 2020. Difference Between DFS and BFS. https://www.geeksforgeeks.org/difference-between-bfs-and-dfs/

How to Apply Technical Analysis to Cryptocurrencies. 2020. SwissBorg. https://swissborg.com/blog/how-to-apply-technical-analysis-to-cryptocurrencies

Kuepper, Justin. 2020. “Day Trading: An Introduction”. Investopedia. https://www.investopedia.com/articles/trading/05/011705.asp#the-basics-of-day-trading

Mitchell, Cory. 2020. “Guide to Swing Trading”. Investopedia.

https://www.investopedia.com/terms/s/swingtrading.asp

Devoted. 2010. “Advantages of Binary Search Tree over Hash Tables”. Data-Structures. November 8, 2010. https://stackoverflow.com/questions/4128546/advantages-of-binary-search-trees-over-hash-tables

Vernon, Jeff. “Cryptocurrency Indicators: What they Mean and Which Ones to Use”. HaaSOnline https://www.haasonline.com/cryptocurrency-indicators-what-they-mean-which-to-use/#:~:text=Some%20of%20the%20most%20popular,equal%20application%20in%20both%20realms.