Lachlan Greenland - 90021176

Curtin College
Data Structures and Algorithms
Assignment Report

Table of Contents

Information for Use

Introduction

The cryptoGraph package allows the user to parse csv files that contain asset and trade information on the cryptocurrency market. The files must be in a specific format for the parsing and the analysis to be accurate. A variety of options are available to the user for different ways of viewing the data.

An added feature is option 10 which makes a request to the Coin Market Cap API, updating all assets loaded into the running program with the most current information.

Installation

To install ensure all dependancies are installed. These can be found in the README.md file. Unzip the file and run the following command for usage information:

```
$ python3 cryptoGraph.py
```

There is a test subfolder which contains unit tests of relevant classes. This can be run from the directory where cryptoGraph.py is with the command:

```
python3 -m unittest discover test
```

Terminology

Most terminology used is self-explanatory. The choice was made to attempt to write descriptive variable names at the cost of lines running longer than necessary. This does change within the test subdirectory. For example, in the test directory on creation of a queue data structure it is referenced as:

```
q = Queue()
```

Walkthrough

Upon running the program in interactive mode with:

```
$ python3 cryptoGraph.py -i
```

The user is presented with a menu offering 10 options, chosen by inputting the corresponding number next to these options. The first option allows the user to load data into the program either from a csv file or obj file. As these are parsed differently, the user must specify which file is being loaded. It is necessary for an assets file to be loaded into the program for any functionality to work, apart from exiting the program.

Once data in the form of assets and trades has been loaded into the program Options 2 and 3 are available and quite similar. Inputting either the asset symbol e.g. BTC for Bitcoin, or BTCETH for the trade from Bitcoin to Ethereum, the program will either show more information on the asset/trade or say there is no information on that asset or trade.

Option 4 again gets input from the user of a starting asset and a finishing asset for two different types of traversals through the data, constructed into a graph. The first finds the shortest path if it exists irregardless of arc weight, whereas the second traversal shows the best traversal for maximising arc weight.

Option 5 allows the user to remove an asset from being analysed. Once processed all evidence of the asset entered will be removed from the system including all trades it is involved with. This is useful for Options 6 and 7 to remove popular cryptocurrencies from the overview and giving a broader assessment of the market. These overviews refresh each time they are selected to keep updated with the assets being filtered out.

Option 8 saves the CurrentMarket() class object created in menu.py. This saves all of the data that has been parsed by the dataGrab class object, and if loaded back in to the program, all filters will be remembered.

Option 9 quits the program and Option 10 has been discussed in the Introduction.

Future Work

The package does require more work in several areas for improved neatness, readability, efficiency, and bulk of certain files. Starting with the menu class, although it keeps the main function small, it does a lot of input checking, which would have been better to move into another class. Also, the currentMarket object is a very large file and has unavoidable repetition due to the slight differences between trade data and asset data. Separating into a trade market and asset market could split up this bulk, but would also not allow for one object to be saved and upon reloading, instantly repopulating all options for use. A cleaner and more streamlined way of checking whether data has been loaded into the current market to prevent crashes would also be a great improvement. The design choice for only adding trades that had matching asset data was made with the API in mind, however in the future this should change to accepting all csv input trades, with the current traversal algorithms being used.

Requirements

1. Starting options

Usage options have been created if the program is not run correctly. Inspiration on formatting was drawn from other command line based programs like grep and awk that output instructions on usage. More information on running the program can be found in the README.md and Walkthrough

2. Load data

This requirement was complete through application of the csv module and the dataGrab class. The option to load either asset data, trade data, or serialised data is given to the user, and from the choice, the appropriate methods in dataGrab are used to process the data into asset and trade objects.

3. Find and display asset details

This requirement was complete through the implementation of a Hash Table. It allows for O(1) look-up and the formatted output is a method within the Asset class.

4. Find and display trade details

This requirement was completed through the implementation of a Hash Table. It allows for O(1) look-up and the formatted output is a method within the Trade class.

5. Find and display potential trade paths

This requirement was completed through the implementation of a Graph data structure, which makes use of a Hash Table, Double Linked List, Queue, and Stack. The graph is structured as an adjacency list and does a breadth first and depth first traversal, printing the paths to the command line. It must be noted that only trades with matching asset data were added to the graphs, thus affecting the total paths that can be followed.

6. Set asset filter

This requirement was completed by allowing the user to input an asset name to be taken out of the Current Market object. As this object contains all the data entered for analysis, it effectively removes the asset chosen from any further analysis. The various removals are done by iterating through the linked lists, retrieving through the Hash Table and remaking the graph, if Option 4 is selected again.

7. Asset overview

This requirement was completed through the Double Linked List and numpy module. Insertion sorting was used for each overview category displayed as the data in the array being sorted was already in some order. As each sort would complete it would display to the screen for the user to scroll to the top of the command line to view all information.

8. Trade overview

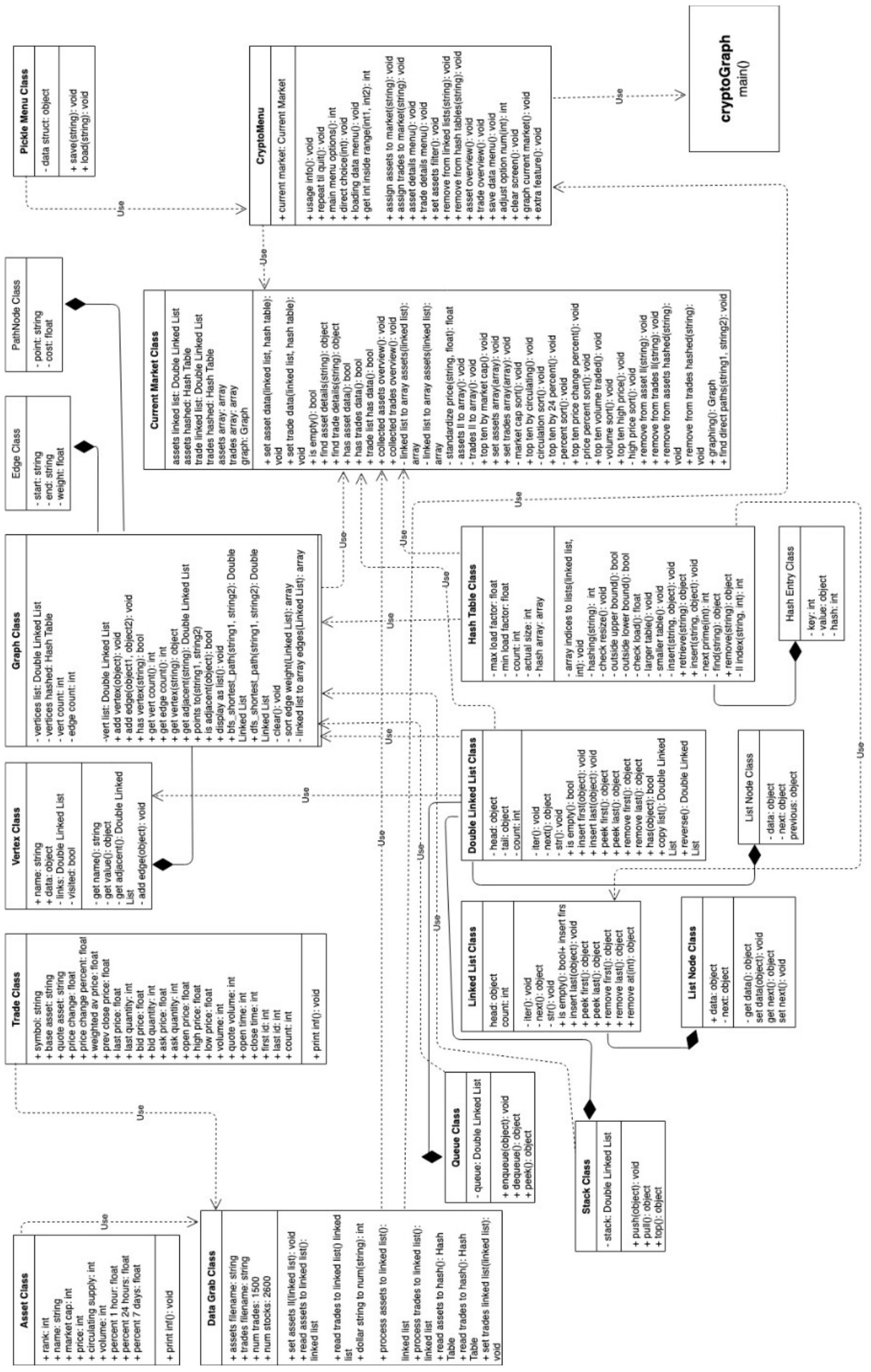
This requirement was completed through the Double Linked List and numpy module. Insertion sorting was used for each overview category displayed as the data in the array being sorted was already in some order. As each sort would complete it would display to the screen for the user to scroll to the top of the command line to view all information.

9. Save data

This requirement was complete through application of the pickle module and the Pickling class. The Current Market object is serialised and saved into the current working directory, after a name has been input by the user. A .obj extension is automatically added.

Class Diagram

- Also included .jpg in directory



Pickle Menu Class
- data struct: object
+ save(string): void
+ load(string): void
CryptoMenu
+ current market: Current Market
+ usage info(): void
+ repeat til quit(): void
+ main menu options(): int
+ direct choice(int): void
+ loading data menu(): void
+ get int inside range(int1, int2): int
+ assign assets to market(string): void
+ assign trades to market(string): void
+ asset details menu(): void
+ trade details menu(): void
+ set assets filter(): void
+ remove from linked lists(string): void
+ remove from hash tables(string): void
+ asset overview(): void
+ trade overview(): void
+ save data menu(): void
+ adjust option num(int): int
+ clear screen(): void
+ graph current market(): void
+ extra feature(): void
cryptoGraph
main()
Use
PathNode Class
- point: string
- cost: float
Edge Class
- start: string
- end: string
- weight: float
Current Market Class
assets linked list: Double Linked List
assets hashed: Hash Table
trade linked list: Double Linked List
trades hashed: Hash Table
assets array: array
trades array: array
graph: Graph
+ set asset data(linked list, hash table): void
+ set trade data(linked list, hash table): void
+ is empty(): bool
+ find asset details(string): object
+ find trade details(string): object
+ has asset data(): bool
+ has trades data(): bool
+ trade list has data(): bool
+ collected assets overview(): void
+ collected trades overview(): void
+ linked list to array assets(linked list): array
- linked list to array assets(linked list): array
- standardize price(string, float): float
- assets ll to array(): void
- trades ll to array(): void
+ top ten by market cap(): void
+ set assets array(array): void
+ set trades array(array): void
- market cap sort(): void
+ top ten by circulating(): void
- circulation sort(): void
+ top ten by 24 percent(): void
- percent sort(): void
+ top ten price change percent(): void
- price percent sort(): void
+ top ten volume traded(): void
- volume sort(): void
+ top ten high price(): void
- high price sort(): void
+ remove from asset ll(string): void
+ remove from trades ll(string): void
+ remove from assets hashed(string): void
+ remove from trades hashed(string): void
+ graphing(): Graph
+ find direct paths(string1, string2): void
Graph Class
- vertices list: Double Linked List
- vertices hashed: Hash Table
- vert count: int
- edge count: int
-vert list: Double Linked List
+ add vertex(object): void
+ add edge(object1, object2): void
+ has vertex(string): bool
+ get vert count(): int
+ get edge count(): int
+ get vertex(string): object
+ get adjacent(string): Double Linked List
+ points to(string1 , string2)
+ is adjacent(object): bool
+ display as list(): void
+ bfs_shortest_path(string1, string2): Double Linked List
+ dfs_shortest_path(string1, string2): Double Linked List
- clear(): void
- sort edge weight(Linked List): array
- linked list to array edges(Linked List): array
Hash Table Class
- max load factor: float
- min load factor: float
- count: int
- actual size: int
- hash array: array
- array indices to lists(linked list, int): void
- hashing(string): int
- check resize(): void
- outside upper bound(): bool
- outside lower bound(): bool
- check load(): float
- larger table(): void
- smaller table(): void
- insert(string, object): void
+ retrieve(string, object): void
+ insert(string, object): void
- next prime(int): int
- find(string): object
+ remove(string): object
- ll index(string, int): int
Hash Entry Class
- key: int
- value: object
- hash: int
Vertex Class
+ name: string
+ data: object
- links: Double Linked List
- visited: bool
- get name(): string
- get value(): object
- get adjacent(): Double Linked List
- add edge(object): void
Double Linked List Class
- head: object
- tail: object
- count: int
- iter(): void
- next(): object
- str(): void
+ is empty(): bool
+ insert first(object): void
+ insert last(object): void
+ peek first(): object
+ peek last(): object
+ remove first(): object
+ remove last(): object
+ has(object): bool
+ copy list(): Double Linked List
+ reverse(): Double Linked List
List Node Class
- data: object
- next: object
previous: object
Trade Class
+ symbol: string
+ base asset: string
+ quote asset: string
+ price change: float
+ price change percent: float
+ weighted av price: float
+ prev close price: float
+ last price: float
+ last quantity: int
+ bid price: float
+ bid quantity: int
+ ask price: float
+ ask quantity: int
+ open price: float
+ high price: float
+ low price: float
+ volume: int
+ quote volume: int
+ open time: int
+ close time: int
+ first id: int
+ last id: int
+ count: int
+ print inf(): void
Linked List Class
head: object
count: int
- iter(): void
- next(): object
- str(): void
+ is empty(): bool+ insert firs
+ insert last(object): void
+ peek first(): object
+ peek last(): object
+ remove first(): object
+ remove last(): object
+ remove at(int): object
List Node Class
+ data: object
- next: object
- get data(): object
set data(object): void
get next(): object
set next(): void
Queue Class
- queue: Double Linked List
+ enqueue(object): void
+ dequeue(): object
+ peek(): object
Stack Class
- stack: Double Linked List
+ push(object): void
+ pull(): object
+ top(): object
Asset Class
+ rank: int
+ name: string
+ market cap: int
+ price: int
+ circulating supply: int
+ volume: int
+ percent 1 hour: float
+ percent 24 hours: float
+ percent 7 days: float
+ print inf(): void
Data Grab Class
+ assets filename: string
+ trades filename: string
+ num trades: 1500
+ num stocks: 2600
+ set assets ll(linked list): void
+ read assets to linked list(): linked list
+ read trades to linked list() linked list
+ dollar string to num(string): int
+ process assets to linked list(): linked list
+ process trades to linked list(): linked list
+ read assets to hash(): Hash Table
+ read trades to hash(): Hash Table
+ set trades linked list(linked list): void

Class Descriptions

Double Linked List

The Double Linked List is a data structure serving many purposes because of its ability to expand easily and in a time complexity of O(1) as well as being able to iterate through the structure to perform any operations needed on the data contained within. For some operations within the program the best time complexity that can occur is O(n), for example when parsing the files given, each line of data must be taken in one at a time. For these sequential operations the linked list is perfect. It also serves as the foundation for several other classes including for the Graph class, Queue class, and Stack class, because the Double Linked List, at the cost of being more expensive memory-wise, does provide O(1) operations for peek first, peek last, retrieve first, and retrieve last operations.

Graph

The Graph that has been created is based around the adjacency list. All vertices in the Graph are stored in a Hash Table for O(1) time complexity when retrieving them. This is to aide in the iteration necessary to occur when going through the arcs of a Vertex stored in a Double Linked List. The method display_as_list() was one of the most helpful when trying to visualise the structure of the graph for different traversals. A breadth first and a depth first traversal were implemented within the Graph class, both returning the paths in a Double Linked List. The bfs was used to find the shortest path, because it is certain to find a path if one exists. The dfs was used due to to the way it stored pathways for later popping off, when necessary to backtrack. When these pathways are added in a sorted order based on edge weight, the traversal chooses the path that maximises or minimises this factor. In this case the traversal maximises the price change percent between trades.

Hash Table

The Graph class used in the program uses a basic, but effective hashing algorithm to ensure very few clashes, which there have been done. Clashes are also avoided by adding a Linked List at each array index. This is due to the Hash Table constantly resizing based on the number of entries in the table. This does require much more space than is ultimately necessary to store only the data that needs to be stored, but for the time complexity of O(1) for retrieval operations, it is deemed worthy. The hashing algorithm for this Hash Table does require a string as a key, as well as on initialisation, also requires an initial sized passed as an argument, which would be looked at for future iterations of the package.

Linked List

The Linked List is only used in the Hash Table to cut down on the extra memory that it takes up to ensure minimal clashes. As a Single Linked List it’s retrieval operations are only O(1) from the head of the list, whereas for anything else the worst case goes to O(n), having to iterate through the entirety of the list to the last node.

Pickling

The Pickle Menu Class is used as a containing for the pickling module, which is a requirement for the package. It allows objects to be saved to the working directory. The program can also load these objects into the environment, and as the Current Market object is a container for all the processed data structures, all features of the program will function immediately on loading this object.

Queue

The Queue class used for the program is an implementation of the Double Linked List with wrap around methods imitating a queue. The list allows the queue to have an O(1) time complexity for all of its operations, apart from the has() function which must iterate through in a worst case situation of O(n) to find whether the queue has the data passed in.

Stack

The Stack class is very similar to the Queue class where it contains a Double Linked List that covers these operations with its own methods imitating the nature of a stack being last in first out. Again, all these operations have a time complexity of O(1). As a count is kept of the data contained in the structure, a count can be provided in O(1), like in all other classes implemented in this package.

Asset

The Asset class greatly simplifies the data being parsed. 1 object contains 10 different object variables that can be used within the program. Also, the variable names give important context to the operations being done on any of these objects as well as improving readability.

Current Market

The Current Market class is an object in the package that aims to contain data already parsed into their optimal data structures for the corresponding operations. For example, a user can look up an asset for more information, and the current market is the access point for the menu. Instead of cluttering the menu with hash table retrieves and sorting operations for freshly filtered data, the current market provides functionality to collect all these operations together into one that makes sense in the context of the user. Another example of the abstraction the current market object allows is its find_direct_paths() method, which combines the bfs and dfs searches for the user compare the differences between the two.

Data Grab

The Data Grab class is an object used to interact with csv files entered by the user. It’s methods are all involved in separating the data into either Trade or Asset objects, removing any extra clutter to turn them into usable numbers or strings, and set up the correct data structures to transfer then into the Current Market object.

Crypto Menu

The Menu class is the point where all command line inputs are taken in and checked, and all instructional output to the command line is located. The Current Market object is a class variable within the Crypto Menu, which is good for behind the scenes control of the data analysis. If no assets or trades have been loaded and a user inputed any of the menu options, error checking will happen between the Current Market and the menu to ensure the program never exists a safe state.

Trade

The Trade class is very similar to the Asset class in terms of its purpose. It allows simplification of the data and improvements in readability. Serving as a container for trade information between a base asset and a trade asset, this is what is used to construct weights of arcs. A difficulty found when implementing the program to use the API was the lack of separation between the base asset and the quote asset. Instead all up-to-date trade data was only given by the symbols together, which is impossible to parse accurately, without a lot of time overhead. A solution that was considered was to continually run slices of each trade combination symbol through a hash retrieve to add to the graph, but in the end it is only updating the existing graph instead of expanding it with new assets.

Justifications

The Double Linked List was one of the most used data structures in the package, second to the Hash Table. It was useful, partially because of the simplicity it allows when expanding or reducing it. Also the time complexity for many operations are very good and in many cases where it was used, it was necessary to have an O(n) operation as all data needed to be parsed at once. The Stack and Queue classes both took advantage of the Double Linked List time operations for their corresponding functionality. The other negative side of a Double Linked List is the larger overhead it carries compared to a Single Linked List due to each List Node containing a pointer to the next node as well as the previous node.

The Hash Table is unparalleled in its retrieval operation. The implementation of it in this package in built on an array, which has a retrieval time of O(1) for any index, however at each index of the array is a linked list to prevent collisions from erasing previously added data. This moves the retrieval operation to in the worst case over O(1), but the likelihood of this occurring is low due to the reliable hash algorithm being used.

The Graph data structure is used in this package as a tool for visually presenting relationships between assets through trades that occur in the market. A hash table is used for much faster retrievals of a Vertex class in the Graph, however to allow for iterative traversals, each vertices arcs are contained in a Double Linked List. The traversals that are completed are time intensive, particularly the breadth first search as it is a complete solution. In other words if only one path exists between two vertices, this path will be found and displayed to the user. The depth first search algorithm was applied to the problem of finding a path through the graph that maximised the arc weight. It is a non-recursive implementation as the scale of the data being traversed could lend to a stack overflow.

Moving finally to the sorting algorithm that was used to sort through asset and trade data that displays an overview of data in top ten lists, insertion sort was used. It was used, because the asset data was already mostly in order, and insertion sort is much more efficient on data that is almost ordered as it recognises points in the array do not not to be switched. Thus the sorting operations are completed much faster than through a bubble or selection sort, which requires comparisons at each step.