Objected Oriented Programming Mid Term Assignment Report

R1: Market Analysis

This segment is implemented by MovingAverage.cpp, OrderBook.cpp and MerkelBot.cpp

R1A:

In MerkelBot.hpp, an OrderBook object orderBook{"20200601.csv"} is created, which retrieves all lines in the 20200601.csv file. Please note that students have been specifically told by tutors not to include the csv file in the submission, therefore my csv file has been excluded from the .zip file.

To initialise orderBook{"20200601.csv"}, all lines in the CSV file are inserted into a map, with their timestamps becoming the map's unique key values. Each unique timestamp maps to its associated vector containing the relevant OrderBookEntry data of that timestamp. Then, separately, each of these unique timestamps is pushed into a Queue data structure, which enables every next timestamp to be called quickly in OrderBook::getNextTime() during the simulation.

R1B:

The algorithm that is used to predict likely future prices is based on the Simple Moving Averages methodology, which is a common stock trading technique. The algorithm calculates the average of the minimum ask prices in the past 10 timestamps, known as the shortMovingAverage, and calculates the average of the minimum ask prices in the past 50 timestamps, known as the longMovingAverage. These two moving averages are recalculated and updated in every timestamp cycle to account for the *next* past 10 and 50 timestamps, hence the name "*moving* average". The idea is that if the shortMovingAverage is greater than the longMovingAverage, then this signals that the future price is likely to increase further and vice versa.

The minimum ask price is used as a proxy for the closing price in this simulation because the lowest price offered is the first price that is processed by the matching engine.

There are five products in the CSV file: ETH/BTC, ETH/USDT, DOGE/USDT, DOGE/BTC and BTC/USDT. If, for instance, the shortMovingAverage of a product is greater than the longMovingAverage, then the algorithm decides to buy the product. If the shortMovingAverage is less than the longMovingAverage, then the algorithm decides to sell.

This algorithm is implemented in MovingAverage.cpp. The MovingAverage class takes in three arguments for initialisation: a vector containing OrderBookEntry objects, shortTime (set to 10) and longTime (set to 50). Each MovingAverage object is set to each of the five product names and stores the corresponding minimum prices and moving averages. In this simulation, there are five of these MovingAverage objects, which are all stored in the allProdMovingAverages vector in MerkelBot.cpp.

To get the minimum price in a *single* timestamp, the algorithm pushes all ask prices for the product in the relevant timestamp into the prodPrices vector, then finds the minimum value.

The calculation of the shortMovingAverage is implemented by MovingAverage::shortMovingAverage(), which takes the minimum prices in the last 10 timestamps and returns the average. The process is the same for MovingAverage::longMovingAverage(), but for the last 50 timestamps.

The MovingAverage::updatePrices function takes in a new vector containing OrderBookEntry objects from a new timestamp and updates the new prices for the product and recalculates its shortMovingAverage and the longMovingAverage accordingly.

[SEE NEXT PAGE]

R2: Bidding and buying functionality
This segment is implemented in MerkelBot.cpp

R2A:
The MerkelBot::decideBid() function decides whether to buy. It iterates through all the objects stored in allProdMovingAverages in MerkelBot.cpp and checks each and every product to see if its shortMovingAverage is ***greater*** than the longMovingAverage. If so, it decides to bid.

R2B:
If the MerkelBot::decideBid() function decides to buy the product, it checks if the wallet holds enough currencies for the bid and if it does, a Bid is inserted into the exchange or orderBook for processing and matching.

The target Bid price is set to the latest timestamp's minimum price for the relevant product multiplied by 1.2. This is to include a 20% premium over the minimum ask price in order to increase the chance of winning the bid.

The target Bid amount to be purchased is set at the value equivalent to 10% of the available paying currency that the bot owns divided by the target Bid price. For example, if the bot decides to buy ETH/BTC, the amount of ETH it targets to buy is: 10% of all the BTCs that it owns divided by the amount of BTC it offers to pay to buy 1 ETH.

R2C:
The matching engine is implemented by the OrderBook::matchAsksToBids function. If the bot wins the bid after matching, the OrderBookEntry's orderType changes from bid to bidsale and is then pushed into the sales vector, which contains all successful matches for the product. The amount associated with the bot's successful bidsale is then processed by the wallet. Note that it is assumed that the the crypto exchange charges a flat trading fee of 0.2%. This fee is added to the price the bot pays for the product, which is implemented by the line **sale.price = (double) sale.price*1.002** in MerkelBot::processSales() when processing the wallet.

R2D:
The withdrawal of a bid or ask is implemented by the MerkelBot::decideWithdraw() function in MerkelBot.cpp. Note that the withdrawal decision is considered only *after* the decisions on both bidding (MerkelBot::decideBid()) and selling (MerkelBot::decideAsk()) have been called.

To decide whether to withdraw, the function first checks whether the bot is trying to buy the Bid product with a currency that it is also trying to sell at the same time. For example, suppose that the bot is trying to buy ETH/BTC and sell BTC/USDT at the same time. Situations like this must not be allowed to happen in this simulation as the bot risks selling BTC it may no longer own as all its BTC is already used up to pay for ETH/BTC, a concurrent bid transaction. This causes BTC to go into negative territory in the bot's wallet.

To resolve this, MerkelBot::decideWithdraw() checks for such a clash and, if it occurs, decides to withdraw either Bid or Ask randomly. Consider the example above again, the bot randomly picks either the ETH/BTC Bid or the BTC/USDT Ask to withdraw based on a random number generated by **int rand = std::rand() % 2**. If rand equals 0, then the bot withdraws the BTC/USDT Ask, else it withdraws the ETH/BTC Bid.

Note that the withdrawal decision is considered before matching is implemented by OrderBook::matchAsksToBids.

Special note to marker: The bot outputs std::cout lines for every withdrawal it implements. To check for this, after the simulation completes, search or the keyword "have withdrawn" in the console to find out which transactions the bot has withdrawn from the orderBook. The std::cout lines also show which are the two products that have clashed.

[SEE NEXT PAGE]

R3: Offering and selling

R3A:

The MerkelBot::decideAsk() function decides whether to sell. If the bot does decide to sell, it checks the wallet to see if there is enough of the selling currency to be offered on the exchange. The function follows a similar process as MerkelBot::decideBid() as described in R2A, except that the decision to insert an Ask is triggered if the product's shortMovingAverage is ***lesser*** than its longMovingAverage.

R3B:

The process is the same as the one described in R2B for MerkelBot::decideBid() except for the target price and the target amount.

The target Ask price is set at 5% more than the minimum ask price for the product. This is to limit potential losses and ensure the bot does not lose too much currency in a fire sale.

The target Ask amount for a single timestamp is set at 10% of the total amount of the Ask currency that the bot currently holds.

R3C:

The matching engine is implemented by the OrderBook::matchAsksToBids function. If the bot wins the sale after matching, the OrderBookEntry's orderType changes from ask to asksale and is then pushed into the sales vector, which contains all successful matches for the product. The amount associated with the bot's successful asksale is then processed by the wallet.

The flat trading fee of 0.2% is ***deducted*** from the proceeds the bot receives as a result of the asksale, which is implemented by **sale.price = (double) sale.price*0.998** in MerkelBot::processSales() when processing the wallet.

R3D:

The exact same process described in R2D also applies to the offering and selling of currencies.

R4: Logging

This segment is implemented by Logging.cpp

R4A:

The Logging::logAssets function takes in the current timestamp and the balance of currencies. Then, using the assets map, the current timestamp is mapped to the balance of currencies. This means the balance at the end of each timestamp is recorded in the assets map. When the simulation ends, all these records are published into the assets.txt file by the function Logging::publish(). This shows how the balance of the bot's currencies change over time.

R4B:

This is implemented by the Logging::logBidsOffers function. It takes in every Bid or Ask that has been inserted in the exchange orderBook as an argument. Then, all Bids are pushed into the bids vector and all Asks are pushed into the asks vector. To publish all Bids and Offers that have been made by the bot, the Logging::publish() function first publishes all of the elements in the bids vector, then all elements in the asks vector to the bidsoffers.txt file.

R4C:

This is implemented by the Logging::logSales function. It takes in as arguments every successful bidsale or asksale OrderBookEntry object, as well as the associated product's mean Bid price and mean Ask price during the timestamp that the sale took place. The function binds the sale OrderBookEntry object with the mean Bid Price and mean Ask Price with a tuple data structure, then pushes it into the bidSales vector if it is a bidsale or pushes it into the askSales vector if it is an asksale.

To publish all successful asksales and bidsales, the Logging::publish() function first prints out all tuples in the bidSales vector, then prints out all tuples in the askSales vector into the successTrades.txt file.

[SEE NEXT PAGE]

Trading algorithm
The trading algorithm is explained in section R1B.

Challenge Requirement

The exchange mechanism in OrderBook.cpp has been optimised. Before optimisation, the whole simulation took around 30 ***minutes*** to complete. However, after optimisation, the entire simulation now only takes between 20 to 27 ***seconds*** to complete. There is a timer that has been coded in main.cpp within the merkelrex() function to track how much time the bot has used to complete the entire simulation. The function then outputs a std::cout line onto the console at the end of the simulation to show the total time taken.

The main reason for the extremely slow processing speed before optimisation is that the bot had to use for-loops to repeatedly iterate over 1 million OrderBookEntries (there are over 1 million lines in the CSV file) in the orderBook each time it carries out simple tasks like getting the next timestamp (OrderBook::getNextTime()), getting all known products (OrderBook::getKnownProducts()), retrieving orders according to specified criteria (OrderBook::getOrders()). Even worse, each time a Bid or Ask is inserted to the orderBook, over 1 million OrderBookEntries had to be sorted from ascending order according to timestamps. All these cause the running time to explode as the size of the input increases, meaning a dataset of over 1 million entries is bound to cause serious problems to the processing speed.

To resolve this, first, the CSV lines are retrieved and stored in a temporary vector called entireOrders. Then, a map named orders that maps timestamps (unique keys) to a vector of OrderBookEntries is created. All elements in entireOrders are then pushed into the orders map, meaning all OrderBookEntries are now arranged according to their timestamps (see constructor function of OrderBook.cpp). This means in every timestamp, the bot is able to just load only the current timestamp's orders quickly by using orders[timestamp]. All other OrderBookEntries are no longer retrieved as they are not needed at all for that current timestamp. Therefore, the orders[timestamp] vector that the bot needs to iterate through at a given time is much shorter. This causes OrderBook::getOrders(), OrderBook::getKnownProducts() as well as the sorting algorithms used in the matching engine and insert function to run much faster.

To ensure OrderBook::getNextTime() does not need to iterate the entire orderBook just to get the next timestamp, the timestamps (unique keys) in the orders map are each pushed into a Queue data structure, which utilises the First In First Out method for organising data. Hence, all the bot needs to do to get the next time is just to .pop() the Queue and push the old timestamp to the back of the Queue, then retrieve the new first element of the Queue. This has helped to speed up the simulation significantly.

Additional note:

Comparing the initial balance (BTC: 10 and USDT: 1000) and the balance at the end of the simulation (BTC: 0.957243, DOGE: 100083.820478, ETH: 366.156708, USDT: 0), in US Dollar terms, there is a profit of **USD 782.95**, representing a return of roughly **0.8%**, after taking into account trading fees.

Note that the USD exchange rate is based on 1st June 2020, the same date as the CSV data:

1 BTC = USD 9758.85
1 DOGE = USD 0.002606
1 ETH = USD 245.17
1 USDT = USD 1.0016