An Introduction To Writing Algorithms *for* Group Alpha Testing Suite

Adam Gleave
Nick Burscough
Artjoms Iskovs
Lawrence Esswood
Christopher Little
Alan Rusnak

1 Getting Started

Version 1.0 of the Testing Suite is almost entirely written in Java. As such, all algorithms that are to be submitted to the system must currently also be written in Java. This example documentation will remain largely independent of IDE, though Eclipse is a good choice due to its ability to streamline JAR creation (as algorithms are submitted to the system as JAR files).
To dive right into algorithm creation, we will now look through the workings of a hello-world-esque algorithm. The algorithm works in a completely random fashion, buying and selling according to the whims of its lord and master, `math.random()`. Obviously, this algorithm will fare extremely poorly in any real-world situation, and serves only to illustrate how we can interface with the testing framework. Following is the algorithm in its entirety, after which we will take a look at a breakdown of how it interfaces with the simulated market.

```java
// HelloRiches.java

import java.util.Iterator;
import orderBooks.Order;
import orderBooks.OrderBook;
import database.StockHandle;
import testHarness.ITradingAlgorithm;
import testHarness.MarketView;
import testHarness.clientConnection.Options;


public class HelloRiches implements ITradingAlgorithm {

        @Override
        public void run(MarketView marketView, Options options) {
                Iterator<StockHandle> stocks =
                    marketView.getAllStocks().iterator();
                StockHandle bestStockEver = stocks.next();
                OrderBook book =
                    marketView.getOrderBook(bestStockEver);
                double propensityToSell = 4;
                while(!marketView.isFinished()) {
                        marketView.tick();
                        propensityToSell += Math.random()-0.5;
                        if(propensityToSell < 5) {
                                //BUY BUY BUY
                                Iterator<? extends Order> iter =
                                    book.getAllOffers();
                                if(!iter.hasNext()) continue;
                                int bd = iter.next().getPrice();
                                marketView.buy(bestStockEver, bd, 1);
                        } else {
                                //SELL SELL SELL
                                Iterator<? extends Order> iter =
                                    book.getAllBids();
                                if(!iter.hasNext()) continue;
                                int bd = iter.next().getPrice();
                                marketView.sell(bestStockEver, bd, 1);
                        }

                }
        }
}
```

2 Breakdown of HelloRiches.java

2.1 Imports

```java
import orderBooks.Order;
import orderBooks.OrderBook;
```

The orderbook construct holds all bids and offers for a stock. Each stock has an associated order book. An order is simply one of these bids or offers. We will see later how we can use these classes to place our algorithm's offers onto the market and also bid for existing orders.

```java
import database.StockHandle;
import testHarness.ITradingAlgorithm;
import testHarness.MarketView;
import testHarness.clientConnection.Options;
```

A stock handle essentially does what it says on the tin and allows us to hold a reference to a particular stock so that we interact with it (buying or selling etc). Market View and Options exist to allow the testing framework to test market situations and vary simulation options such as tick size. Finally, the ITrandingAlgorithm class is implemented by the algorithm to ensure correct interfacing with the framework (i.e. the presence of a `run()` method.

2.2 Methods

```java
@Override
public void run(MarketView marketView, Options options) {
```

As we are implementing the `ITradingAlgorithm` interface, we must `@Override` the `run()` method. The `MarketView` and `Options` will be decided by the testing server and handed to the algorithm.

```java
Iterator<StockHandle> stocks = marketView.getAllStocks().iterator();
StockHandle bestStockEver = stocks.next();
OrderBook book = marketView.getOrderBook(bestStockEver);
```

First, we define a new `Iterator` that iterates over all possible stocks - (`marketView.getAllStocks().iterator()`), and then use it to arbitrarily pick a stock for our algorithm to work with. In this example we simply choose the stock that is returned from `stocks.next()`, i.e. the first stock of the iterator. Finally, we extract the `OrderBook` for this specific stock so that we can begin attempting to interface with the market.

```
double propensityToSell = 4;
while(!marketView.isFinished()) {
       marketView.tick();
       propensityToSell += Math.random()-0.5;
```

Here we begin to set up the logic of our algorithm - how we are going to determine our money making strategy for this market tick. A tick, in this context, refers to the smallest possible snapshot of time that our algorithm can witness in our market. We can do as much computation as we need to (provided we do not time-out our simulation) within a tick and then advance in time by calling `marketView.tick()`. For this example algorithm, our strategy is going to be based on the rolling of `Math.random()` affecting our inclination towards selling the stock we have bought. In this code snippet specifically, we see that, so long as the simulation has not finished, we will advance to the next tick of time and then alter our inclination towards buying/selling with equal weight.

```
if(propensityToSell < 5) {
       //BUY BUY BUY
       Iterator<? extends Order> iter = book.getAllOffers();
       if(!iter.hasNext()) continue;
       int bd = iter.next().getPrice();
       marketView.buy(bestStockEver, bd, 1);
} else {
       //SELL SELL SELL
       Iterator<? extends Order> iter = book.getAllBids();
       if(!iter.hasNext()) continue;
       int bd = iter.next().getPrice();
       marketView.sell(bestStockEver, bd, 1);
}
```

Directly following from the previous snippet, after altering our propensity to sell, we then check if we lie within the buying range or the selling range. Note that the initial value of `propensityToSell` is set to 4 to force the market strategy of the first tick to be a buy (as we of course need to buy before we can sell). Upon making our buy/sell decision, we use the `OrderBook` object `book` to get the iterator for either all current bids or all current offers for our chosen stock. Then, we check to see if the iterator is non-empty, setting an int to the value of either the best off or the best bid if this is the case. Finally, we use our `marketView` to buy or sell respectively.

(And then progress to the next `tick()`, as per our loop)

3 Key Commands/Concepts

As seen in the simple example, there are a few key methods for interfacing with the testing framework that are essential for writing your own algorithms. Namely:

`marketView.tick()` Used to progress the similation by a single, discrete unit of time, the granularity of which, can be changed by editing the simulation options.

`marketView.buy(<Stock>,<Price>,<Volume>)` Used to place a buy order into the order book of a specified stock. If your price matches that of a sell order present in the book, a trade will occur.

`marketView.sell(<Stock>,<Price>,<Volume>)` The counterpart to the `buy()` method, this is used to place a sell order onto the order book of the specified stock. If the price and volumes match a buy order, a trade will occur.

`book.getAllOffers()` Returns an `Iterator<? extends Order>` containing all of the offers (sell orders) from the `book` object. We can use this to, for example, calculate the standard deviation for sell orders at this point in time, as well as perform other methods of statistical analysis that we might like to use to try and predict how the market will change in subsequent ticks.

`book.getAllBids()` Returns a similar iterator, this time containing all of the buy orders from the specified `book` object.

`marketView.getAllStocks.Iterator()` A good way of setting up the ability to look through all available stocks from the current data set (at the current `tick`. Returns an Iterator containing all Stocks that are in the simulation.

4 A (more in depth) example

Here we see the full source code of the Depth algorithm, followed by a breakdown of how it functions.

```
package sampleAlgos;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collections;
import java.util.HashMap;
import java.util.Iterator;
import java.util.LinkedList;
import java.util.List;
import java.util.Map;
import java.util.Map.Entry;
import orderBooks.BuyOrder;
import orderBooks.Match;
import orderBooks.Order;
import orderBooks.OrderBook;
import orderBooks.SellOrder;
import testHarness.ITradingAlgorithm;
import testHarness.MarketView;
import testHarness.clientConnection.Options;
import valueObjects.TickOutOfRangeException;
import database.StockHandle;

public class Depth implements ITradingAlgorithm {
@Override
public void run(MarketView market, Options options) {
new DepthImpl(market, options).run();
}

private class DepthImpl {
private final double decayFactor;
private final double tradeThreshold;
private final int volumePercentage;
private final MarketView market;
private final List<String> configuredStocks;
private List<StockHandle> stocks;
private Map<StockHandle,Double> pressure;
private Map<StockHandle,Integer> volume;
private List<StockDouble> normalizedPressure;

public DepthImpl(MarketView market, Options options) {
decayFactor = Double.parseDouble(options.getParam("decayFactor"));
tradeThreshold = Double.parseDouble(options.getParam("tradeThreshold"));
volumePercentage = Integer.parseInt(options.getParam("volumePercentage"));
String rawUserStocks = options.getParam("stocks");
if (rawUserStocks != null) {
List<String> userStocks;
userStocks = Arrays.asList(rawUserStocks.split(","));
configuredStocks = Collections.unmodifiableList(userStocks);
} else {
configuredStocks = null;
}
this.market = market;}
```

```java
public void run() {
List<StockHandle> marketStocks = market.getAllStocks();
if (configuredStocks == null) {
stocks = marketStocks;
} else {
stocks = new LinkedList<StockHandle>();
for (StockHandle s : marketStocks) {
if (configuredStocks.contains(s.getTicker())) {
stocks.add(s);
}
}
}

while (!market.isFinished()) {
tick(market.tick());
}
}

private void tick(Iterator<Match> matches) {
updatePressures();
updateVolumes();
updateNormalizedPressures();
trade();
}

private double orderPressure(Iterator<? extends Order> orders) {
LinkedList<Integer> priceVolume = new LinkedList<Integer>();

if (orders.hasNext()) {
int bestPrice = 0;
Order o;
o = orders.next();
bestPrice = o.getPrice();

 while (orders.hasNext()) {
o = orders.next();
int deltaPrice = Math.abs(bestPrice - o.getPrice());
priceVolume.addFirst(deltaPrice * o.getVolume());
}
}

double pressure = 0;
for (int pv : priceVolume) {
pressure = pressure * decayFactor;
pressure += pv;
}

return pressure;
}

private double calculatePressure(StockHandle s) {
OrderBook ob = market.getOrderBook(s);

Iterator<BuyOrder> bids = ob.getAllBids();
Iterator<SellOrder> offers = ob.getAllOffers();

return orderPressure(bids) - orderPressure(offers);
}

private void updatePressures() {
pressure = new HashMap<StockHandle, Double>();
for (StockHandle s : stocks) {
pressure.put(s, calculatePressure(s));
}
}
```

```
private int orderVolume(Iterator<? extends Order> orders) {
int volume = 0;
while (orders.hasNext()) {
volume += orders.next().getVolume();
}
return volume;
}

private int calculateVolume(StockHandle s) {
OrderBook ob = market.getOrderBook(s);

Iterator<BuyOrder> bids = ob.getAllBids();
Iterator<SellOrder> offers = ob.getAllOffers();

return orderVolume(bids) + orderVolume(offers);
}

private void updateVolumes() {
volume = new HashMap<StockHandle, Integer>();
for (StockHandle s : stocks) {
volume.put(s, calculateVolume(s));
}
}

private class StockDouble implements Comparable<StockDouble> {
public final StockHandle stock;
public final double val;

public StockDouble(StockHandle stock, double val) {
this.stock = stock;
this.val = val;
}

public int compareTo(StockDouble sv) {
return Double.compare(Math.abs(val), Math.abs(sv.val));
}
}

private void updateNormalizedPressures() {
normalizedPressure = new ArrayList<StockDouble>();
for (StockHandle s : stocks) {
int v = volume.get(s);
double p = pressure.get(s);
double normalized = p / v;
StockDouble sd = new StockDouble(s, normalized);
normalizedPressure.add(sd);
}

Collections.sort(normalizedPressure);
}

private void trade() {
// iterate over stocks in order of descending pressure
// make an aggressive trade on the first one we can that is
// above threshold
for (StockDouble sd : normalizedPressure) {
if (Math.abs(sd.val) < tradeThreshold) {
break;
}
StockHandle s = sd.stock;
OrderBook o = market.getOrderBook(s);

int totalVolume = volume.get(s);
int orderVolume = totalVolume * volumePercentage / 100;


try {
```

```
if (sd.val > 0) {
int bestOffer = (int)o.getLowestOffer().getValue(0);
int totalPrice = bestOffer * orderVolume;
// buy at most what we have the capital to purchase
totalPrice = Math.min(totalPrice, market.getFundsLessCommission().intValue());
orderVolume = totalPrice / bestOffer;
if (!market.buy(s, bestOffer, orderVolume)) {
System.err.println("error buying");
}
} else {
int bestBid = (int)o.getHighestBid().getValue(0);

Iterator<Entry<StockHandle, Integer>> portfolio = market.getPortfolio();
int amountWeOwn = 0;
while (portfolio.hasNext()) {
Entry<StockHandle,Integer> position = portfolio.next();
if (position.getKey().equals(s)) {
amountWeOwn = position.getValue();
}
}
if (amountWeOwn > 0) {
orderVolume = Math.min(orderVolume, amountWeOwn);
if (!market.sell(s, bestBid, orderVolume)) {
System.err.println("error selling");
}
}
}
break;
} catch (TickOutOfRangeException e) {
// no data in order book -- try again
continue;
}
}
}
}
}
```

5 Breakdown of Depth.java

5.1 Imports

```
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collections;
import java.util.HashMap;
import java.util.Iterator;
import java.util.LinkedList;
import java.util.List;
import java.util.Map;
import java.util.Map.Entry;
import orderBooks.BuyOrder;
import orderBooks.Match;
import orderBooks.Order;
import orderBooks.OrderBook;
import orderBooks.SellOrder;
import testHarness.ITradingAlgorithm;
import testHarness.MarketView;
import testHarness.clientConnection.Options;
import valueObjects.TickOutOfRangeException;
import database.StockHandle;
```

Here we see very standard imports for any Java program with a need for multiple data storage methods - arrays, lists and hashmaps are all present along with a few other constructs that are more efficient for very specific tasks. Also, we will be using most of the methods from `orderBooks`, along with interfaces that we saw in the previous example, such as `ITradingAlgorithm`.

5.2 Methods

```
public class Depth implements ITradingAlgorithm {
       @Override
       public void run(MarketView market, Options options) {
              new DepthImpl(market, options).run();
       }
```

We declare the class for our algorithm and override the `run()` method from `ITradingAlgorithm`. The new `run` method initialises a new `DepthImpl`, passing in the `MarketView` and `Options` and then calls the `run()` method of `DepthImpl`.

```
private class DepthImpl {
        private final double decayFactor;
        private final double tradeThreshold;
        private final int volumePercentage;
        private final MarketView market;
        private final List<String> configuredStocks;
        private List<StockHandle> stocks;
        private Map<StockHandle,Double> pressure;
        private Map<StockHandle,Integer> volume;
        private List<StockDouble> normalizedPressure;
```

Here we declare our `DepthImpl` class, which is the class we will actually be running since we called the `run()` method in here from the `run()` method of `Depth`. The essence of our algorithm is the calculation of perceived market 'pressures', in an attempt to work out the variations in supply and demand and their impacts on the orderbook so that we might guess at how the orderbook is likely to change in subsequent ticks. Thus, we initialise data structures for holding the various pressure paramaters, as well as the standard things we might want to keep track of in any trading algorithm i.e. a list of stocks we will be working with.

```
public DepthImpl(MarketView market, Options options) {
        decayFactor =
            Double.parseDouble(options.getParam("decayFactor"));
        tradeThreshold =
            Double.parseDouble(options.getParam("tradeThreshold"));
        volumePercentage =
            Integer.parseInt(options.getParam("volumePercentage"));
        String rawUserStocks = options.getParam("stocks");
        if (rawUserStocks != null) {
                List<String> userStocks;
                userStocks = Arrays.asList(rawUserStocks.split(","));
                configuredStocks =
                    Collections.unmodifiableList(userStocks);
        } else {
                configuredStocks = null;
        }
        this.market = market;}
```

This is our constructor for the `DepthImpl` class. When it is called, we simply initialise the variable in `DepthImpl` to the assosciated values that have been passed in view the options. Then, so long as our simulation actually has data on at least one stock, we pass the stocks into an unmodifiable list called `configuredStocks`.

```
public void run() {
        List<StockHandle> marketStocks = market.getAllStocks();
        if (configuredStocks == null) {
                stocks = marketStocks;
        } else {
                stocks = new LinkedList<StockHandle>();
                for (StockHandle s : marketStocks) {
                        if (configuredStocks.contains(s.getTicker()))
                            {
                                stocks.add(s);
                        }
                }
        }

        while (!market.isFinished()) {
                tick(market.tick());
        }
}
```

Here we configure the stocks that we wish to work with, eventually piping them into a `LinkedList` called `stocks` and then we call our own `tick` method, passing in the actual `tick()` method from the `MarketView` so that we can advance our snapshot of the market.

```
private void tick(Iterator<Match> matches) {
        updatePressures();
        updateVolumes();
        updateNormalizedPressures();
        trade();
}
```

This is the main logic of our algorithm. First, we update our view of the market pressures by calling `updatePressures()`, then we update volumes (the amount of stock willing to be bought and sold), then we normalize the pressures by weighting them according to volume, and finally we check if there are any trades that are worthwhile, and if there are, we place aggresive buy/sell orders in an attempt to profit.

```java
private double orderPressure(Iterator<? extends Order> orders) {
        LinkedList<Integer> priceVolume = new LinkedList<Integer>();

        if (orders.hasNext()) {
                int bestPrice = 0;
                Order o;
                o = orders.next();
                bestPrice = o.getPrice();

                while (orders.hasNext()) {
                        o = orders.next();
                        int deltaPrice = Math.abs(bestPrice -
                            o.getPrice());
                        priceVolume.addFirst(deltaPrice *
                            o.getVolume());
                }
        }

        double pressure = 0;
        for (int pv : priceVolume) {
                pressure = pressure * decayFactor;
                pressure += pv;
        }

        return pressure;
}
```

orderPressure works by taking an iterator of orders (for example, all of the sell orders for a specific price) and then returning a 'pressure' value. This value represents the amount of pressure on the market that either supply (if it was a list of sell orders) or demand (if it was a list of price orders) is exerting on the price of the assosciated stock. If there is an excess in demand pressure, we would expect price to rise over time and conversely, for the price to fall over time with excess supply pressure.

```
private double calculatePressure(StockHandle s) {
        OrderBook ob = market.getOrderBook(s);

        Iterator<BuyOrder> bids = ob.getAllBids();
        Iterator<SellOrder> offers = ob.getAllOffers();

        return orderPressure(bids) - orderPressure(offers);
}
```

Using the previous (`orderPressure()`) method, here we attempt to calculate the overall pressure on the price of a stock by calculating the difference between demand pressure and supply pressure.

```
private void updatePressures() {
        pressure = new HashMap<StockHandle, Double>();
        for (StockHandle s : stocks) {
                pressure.put(s, calculatePressure(s));
        }
}
```

Pressure is calculated for every single stock in `stocks` and then the pressure values contained within the `stock` object are updated to their new values. Keep in mind that this is done each tick so that our algorithm always has an up to date view of how the price is being pressured for every stock we care about.

```
private int orderVolume(Iterator<? extends Order> orders) {
        int volume = 0;
        while (orders.hasNext()) {
                volume += orders.next().getVolume();
        }
        return volume;
}
```

A simple getter-esque function to return the cumulative volume across a list of orders.

```java
private int calculateVolume(StockHandle s) {
        OrderBook ob = market.getOrderBook(s);

        Iterator<BuyOrder> bids = ob.getAllBids();
        Iterator<SellOrder> offers = ob.getAllOffers();

        return orderVolume(bids) + orderVolume(offers);
}
```

Sums the cumulative volume across buy orders and the cumulative volume across sell orders for a specific stock.

```java
private void updateVolumes() {
        volume = new HashMap<StockHandle, Integer>();
        for (StockHandle s : stocks) {
                volume.put(s, calculateVolume(s));
        }
}
```

Runs each tick to ensure our view of cumulative volumes for each stock is correct by recalculating based on the current tick's data and then updating the relevant data structures.

```
private class StockDouble implements Comparable<StockDouble> {
        public final StockHandle stock;
        public final double val;

        public StockDouble(StockHandle stock, double val) {
                this.stock = stock;
                this.val = val;
        }

        public int compareTo(StockDouble sv) {
                return Double.compare(Math.abs(val),
                    Math.abs(sv.val));
        }
}
```

A custom datatype that we will use to hold a handle to a specific stock along with a double that will eventually hold the normalized pressure of that stock. Here we can see implementations of all the methods that we need to implement for the datatype to actually be useful e.g. `compareTo` to allow comparisons to other `StockDoubles`.

```
private void updateNormalizedPressures() {
        normalizedPressure = new ArrayList<StockDouble>();
        for (StockHandle s : stocks) {
                int v = volume.get(s);
                double p = pressure.get(s);
                double normalized = p / v;
                StockDouble sd = new StockDouble(s, normalized);
                normalizedPressure.add(sd);
        }

        Collections.sort(normalizedPressure);
}
```

For each stock in our list of relevant stocks, `stocks`, create a `StockDouble` containing a handle to the stock and the normalized pressure of that stock at the current tick. Add all of these elements to a sorted list of `StockDoubles`.

```
private void trade() {
        // iterate over stocks in order of descending pressure
        // make an aggressive trade on the first one we can that is
        // above threshold
        for (StockDouble sd : normalizedPressure) {
                if (Math.abs(sd.val) < tradeThreshold) {
                        break;
                }
                StockHandle s = sd.stock;
                OrderBook o = market.getOrderBook(s);

                int totalVolume = volume.get(s);
                int orderVolume = totalVolume * volumePercentage /
                    100;

                try {
                        if (sd.val > 0) {
                                int bestOffer =
                                    (int)o.getLowestOffer().getValue(0);
                                int totalPrice = bestOffer *
                                    orderVolume;
                                // buy at most what we have the
                                    capital to purchase
                                totalPrice = Math.min(totalPrice,
                                    market.getFundsLessCommission().intValue());
                                orderVolume = totalPrice / bestOffer;
                                if (!market.buy(s, bestOffer,
                                    orderVolume)) {
                                        System.err.println("error
                                            buying");
                                }
                        } else {
                                int bestBid =
                                    (int)o.getHighestBid().getValue(0);

                                Iterator<Entry<StockHandle, Integer>>
                                    portfolio = market.getPortfolio();
                                int amountWeOwn = 0;
                                while (portfolio.hasNext()) {
                                        Entry<StockHandle,Integer>
                                            position = portfolio.next();
                                        if
                                            (position.getKey().equals(s))
                                            {
```

```
                                        amountWeOwn =
                                            position.getValue();
                                }
                        }
                        if (amountWeOwn > 0) {
                                orderVolume =
                                    Math.min(orderVolume,
                                    amountWeOwn);
                                if (!market.sell(s, bestBid,
                                    orderVolume)) {
                                        System.err.println("error
                                            selling");
                                }
                        }
                }
                break;
        } catch (TickOutOfRangeException e) {
                // no data in order book -- try again
                continue;
        }
    }
}
```

The final part of our algorithm - the method that decides whether or not it is worth making any trades in the current tick. Our propensity to trade throughout the whole algorithm is based on our `tradeThreshold`. Our trade logic works by iterating over our sorted list of `StockDoubles` and, for each one, if there is upwards pressure on the price (excess demand), attempting to buy stock so that we might sell later to make profit, otherwise (in the case of excess supply and hance a falling price), checking if we own any of that stock and if we do, attempting to make a sale. The algorithm is buy low, sell high in a very pure form, using the 'pressures' of supply and demand to try and determine where the market price will go (and hence when to buy, and when to sell).

Documentation By Nick Burscough, for Team Alpha.