A Trading Strategy for Continuous Double Auction based on Gjerstad-Dickhaut [1] Algorithm

Tiaobo Hao
MSc. Advanced Computing: Internet Technology with Security
University of Bristol
th12120@my.bristol.ac.uk

Yan Yan
MSc. Advanced Computing: Internet Technology with Security
University of Bristol
th12135@my.bristol.ac.uk

Zhanwen Xu
MSc. Advanced Computing: Creative Technology
University of Bristol
zx8176@my.bristol.ac.uk

Abstract

Trading strategies in auctions have been widely researched in resent years, which have promoted interest in modeling bidding behaviors with simulated agent models.

This paper provides the evolution of a trading strategy starting from Gjerstad-Dickhaut (GD) algorithm. In the following content, the implementation of GD, the process of modifying it, and experiments carried out will be demonstrated. The results of this strategy are tested in a Continuous Double Auction (CDA) along with comparisons among algorithms such as Give Away, Zero Intelligence Constrained (ZIC), Shaver, Sniper, and Zero Intelligence Plus (ZIP) [2].

1. Introduction

The aim of this assignment is to research into trading algorithms and attempt to implement one that out performs other trading robots in Bristol Stock Exchange, which is a simulation of a financial exchange running a limit order book (LOB) in a single tradable security. GD was chosen as a starting point because of its ability to learn from history and room for improvement.

2. Gjerstad-Dickhaut

Gjerstad-Dickhaut trading algorithm is a memory-based agent architecture. With existing auction history, GD trader can produce an order that maximizes its expected profit. In practice, GD will calculate the expected profits for all the bids and sells in history with length m, and pick the price with the highest expected profit if it is within allowance. The expected profit E of a price b can be formulated as:

$$E(b) = q(b) * r(b)$$

q(b) is the belief function which shows how likely a bid *b* will be accepted:

$$q(b) = \frac{T(b)}{m}$$

Where *T(b)* is the number of successful trades in auction history with a winning bid of *b* or less.

r(b) is the payoff function that shows the profit if a bid *b* is accepted:

$$r(b) = x - b$$

Where x is the limit price provided by the customer. Vice-versa for robots doing selling instead of bidding.

In implementation, when the market starts with no transaction list, GD acts as Give Away algorithm. Since BSE will respond to all traders when a new order was placed in LOB, GD can gain the relative information using respond() function. Therefore, if there was a transaction occurred, GD will update its transaction history in respond() function. When GD is generating an order (i.e. calculating a quote price), the program will invoke getquoteprice() to obtain one. In getquoteprice(), two other functions getP() and getE() will be called to calculate *q(b)* and *r(b)* of each price existed in transaction history.

The profit comparison between GD and other trading robots shows a reasonable result. **Table 1** lists the Win-Lose ratio between robots in the market of duration of 180 seconds with different order publish intervals. There are 20 of each robot in the test.

Interval		GVWY	SHVR	ZIC	ZIP
30	GD	60%	0	90%	80%
15		80%	0	100%	80%
6		90%	0	100%	80%

Table 1: Winning rate of GD

There are some characteristics of the behaviors of GD agents found during implementation.

- As GD is very dependent on auction history, the more transactions there are, the more effectively aggressive the GD agent becomes. However, the limit size of the history is still under consideration due to some trade-offs that is going to be mentioned later.
- When GD dominates the party of bidder or seller. GD appears to be very aggressive and always has higher profit. However, when various robots are evenly distributed, GD does not work that well. This is because GD normally has fewer transactions than other robots, as the quote price GD produces is less likely to be accepted than prices given by robots such as Give Away and Shaver although the quote price has the highest profit expectation. Since BSE does not involve trading cost, Shaver algorithm takes full advantage of this assumption.
- GD can only choose to produce a quote price that has appeared in the auction history, which makes its learning period very long. Moreover, when the auction history is not long enough, the decision made by GD is normally not the optimum.
- Multiple GD robots tend to choose the same quote price when their budgets are closed in a certain region. Because they share the same history, it is likely that the allowed quote price with the highest profit expectation is the same. However, this causes self-competition among GD agents.

The modifications made to GD are based on those characteristics.

3. History Limitation (MGD)

As tests were carried out with large amount of trades, it was found to be unnecessary and misleading to keep all the auction history and take them into consideration when calculating the quote price with the highest profit expectation. The reasons are:

- As the market runs, the running time of GD keeps increasing since the size of auctions history is increasing.
- It is possible for competitors to change their policy at any time or the market changes suddenly. In either case, GD needs only the fresh history that represents the recent transactions to adjust its decision for a quicker reaction.

Learning from MGD [3], we added a limitation to the size of the history for agents. However, the limit price in MGD was not considered in our implementation since BSE generates orders randomly and there does not exist unreasonable transactions.

Table 2 demonstrates how the size of history affects GD profit by showing the average profit of GD with different history size along with average number of trades:

Trade#	#100	#200	#400	#600	#800
412	197	192	184	191	188

869	413	405	398	395	404
1240	616	597	570	608	612

Table 2: GD profit with different history size

The five versions of GD robots were run in the same market simultaneously so they had the same source of transaction list. Judging from the table, in experiments with different average number of trades, robots with the history size of 100 tend to get higher profit than others.

4. Time based profit expectation

As mentioned earlier, GD has fewer trades than competitors. This is because GD does not adjust its quote price wisely with the market running. Due to the equal contributions of payoff and belief to the expected profit, GD tend to insist on a quote price, which is likely to have low possibility of being traded. In this stage, we managed to modify the current GD so that it can adjust its quote price according to different customer orders. We will refer the modified GD as HXY in later content.

The first thing we have done is to introduce a new factor *r'(b)* as a measurement of the profit rate at price *b*. *r'(b)* is defined as:

$$r'(b) = \frac{r(b)}{x}$$

Where x is the limit price given by customer order, and $r(b)$ is the payoff function.

By multiplying the profit rate with the profit expectation, our trader will try to produce a quote price with less profit rather than insisting on the best-single-trade profit when the profit rate is too low. This results the GD traders making different decisions according to different limit prices.

Another modification was made on the assumption that the original GD assumes the profit of one trade does not devalue as time goes on. In real market, the longer it takes for a trade, the fewer potential trades there will be for the agent, and therefore, the current profit devalues. Based on the fact that BSE is a market with limited time and fixed number of traders, we replace the original assumption by assuming the orders are decay-able, that is, we made the following 2 assumptions:

- The possibility of an order being accepted decreases as time goes on, because potential buyers/sellers get fewer.
- The expected profit of this order gets lower as time goes on, as it is getting more unlikely to be accepted.

Our solution is to build a mathematical model and find the best target value based on these assumptions.

Firstly, we define 2 functions:

$$f_b(t)$$
$$= \textit{The possibility of order of price b being accepted at time t.}$$

And

$$g_b(t)$$
$$= \textit{Expected profit of price b at time t.}$$

Based on our assumptions, both $f_b(t)$ and $g_b(t)$ decrease on the range [0, *e*] where *e* is the ending time of the market. So the time-based profit expectation $E_b(t)$ can be defined as:

$$E_b(t) = \int_t^e g(t) * f(t)$$

In order to simplify the problem, we simply assume $f_b(t)$ is linear and

$$g_b(t) = r(b) * f_b(t)$$

(Note that *r(b)* is a constant given *b*). Then we redefine the belief function *q(b)* as the approximation of the possibility that the order will be accepted during the whole market session. Also, we made a reasonable assumption that at time *e*, which is the time the market closes, both $g_b(t)$ and $f_b(t)$ equals to 0.

Figure 1 shows the images of $f_b(t)$.

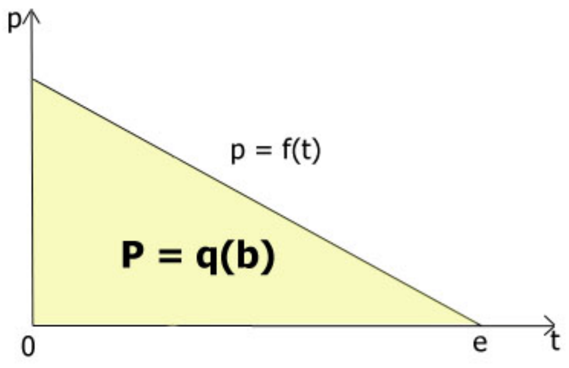

Figure 1: possibility-time relation

In this mathematical model, $P = q(b) = \textit{the area of the triangle} = \int_0^e f(t)$, where P is the total possibility of order of price b being accepted.

Assume $f(t) = kt + h$, so $f(0) = h$.

Therefore,

$P = \frac{1}{2} * h * e \Leftrightarrow h = \frac{2P}{e}$, and

$f(e) = ke + \frac{2P}{e} = 0 \Leftrightarrow k = \frac{-2P}{e^2}$.

So,

$$f(t) = -\frac{2P}{e^2} * t + \frac{2P}{e}$$

Hence *g(t)* can be formulated as:

$$g(t) = r(b) * (-\frac{2P}{e^2} * t + \frac{2P}{e})$$

Where *r(b)* is the profit at price *b*.

With substitution with *f(t)* and *g(t)*,

$$E_b(t) = \int_t^e g(t) * f(t) * dt$$

$$= \int_t^e r(b) * (\frac{4P^2}{e^4} t^2 - \frac{8P^2}{e^3} t + \frac{4P^2}{e^2}) * dt$$

$$= \frac{4r(b)P^2}{e^2} * \int_t^e \left(\frac{1}{e^2} t^2 - \frac{2}{e} t + 1\right) * dt$$

$$= \frac{4r(b)P^2}{e^2} * (\frac{e}{3} - \frac{t^3}{3e^2} + \frac{t^2}{e} - t)$$

$$= \frac{4r(b)P^2}{3e^4} * (e^3 - t^3 + 3et^2 - 3e^2 t)$$

Where $E_b(t)$ is the time based profit expectation of price b.

Lastly, we joined the multiplier with profit rate, and named it "valuator" *V(b)*:

$$V(b) = r'(b) * E_b(t)$$

Price in history with the highest **value** is selected as the quote price.

Table 3 lists the Win-Lose ratio of HXY with Time-Based Expectation and **Figure 2** shows the result among competitors in this stage.

Interval	30	15	6
	HXY		
GD	70%	70%	60%
GVWY	50%	80%	100%
SHVR	10%	10%	10%
ZIC	80%	100%	100%
ZIP	70%	80%	90%

Table3: Winning rate of HXY against others

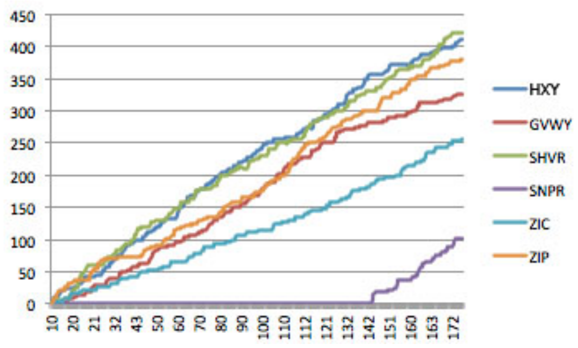

Figure 2: profit-time market run

Comparing **Table 3** with **Table 1**, some improvements can be seen on HXY_TBE competing with other agents. Statistically, HXY_TBE has averagely 2% more profit than the original GD.

5. Best-guess price

Another disadvantage of GD is that GD can only choose quote prices that have been successfully traded before in the history. As there are not many transactions in the history in early market, the quote price does not make much statistical sense. In this modification, we managed to enable HXY to calculate the optimum quote price based on the values of historical prices.

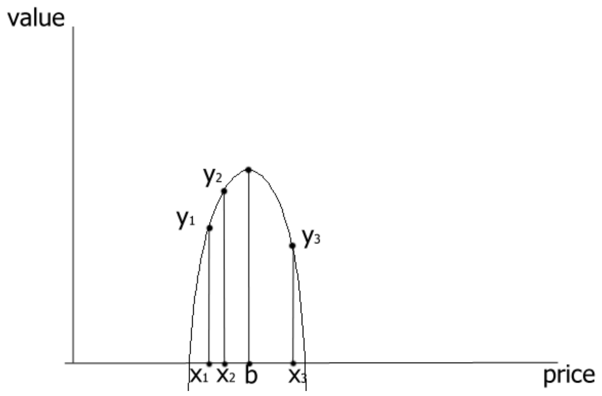

Figure 3: dig out the history

The theory is based on the result of the time-based expectation that we discussed in the previous section. Assume we found the price x_2 having the best expectation y_2 among all recorded values. x_1 and x_3 are the closest recorded prices near x_2, having the expectation of y_1 and y_3.

As can be seen in **Figure 3**, it is likely that the price x_2 has the highest value y_2 in the history, but it might not be the optimum quote price. We tried to find the optimum b by naively mapping the current quote price and its two neighbors onto a quadratic function *f(x)* regardless of other historical prices:

$$f(x) = a(x-b)^2 + c$$

With the substitution of the three prices and values:

$$y_1 = a(x_1 - b)^2 + c \qquad (1)$$

$$y_2 = a(x_2 - b)^2 + c \qquad (2)$$

$$y_3 = a(x_3 - b)^2 + c \qquad (3)$$

$\frac{(1)-(2)}{(2)-(3)}$ Makes:

$$\frac{y_1 - y_2}{y_2 - y_3} = \frac{(x_1 - b)^2 - (x_2 - b)^2}{(x_2 - b)^2 - (x_3 - b)^2}$$

And therefore, the best-guess price b is calculated as:

$$b = \frac{(y_2 - y_3)(x_1{}^2 - x_2{}^2) - (y_1 - y_2)(x_2{}^2 - x_3{}^2)}{2((y_1 - y_2)(x_3 - x_2) - (y_2 - y_3)(x_2 - x_1))}$$

With this algorithm, HXY can produce quote prices that were never recorded and this facilitates faster learning in the early market from the history.

In implementation, we pass through x_1,x_2, x_3,y_1,y_2,y_3 to a new function *getbgprice()* to calculate best-guess price b. If program found that x_2 is already the minimum or maximum price or $y_1 = y_2 = y_3$, quadratic function would not applied.

Now, program can find out a best-guess price if there are few choices from transaction history. However, it won't be necessary if the history is long enough to cover most possible price.

6. Profit reduction

There has always been a trade off between the profit of a single trade and the total number of trades done by an agent during a market session. Prices with less profit are more likely to be accepted, prices with more profit are more difficult to be accepted. As a matter of fact, the Shaver has a great performance due to its relatively large amount of accepted orders even though such strategy is doubtful in the real world. In order to explore into this trade off, experiments were carried out to find a best-balanced solution.

Naive profit reduction was chosen to be our first approach to increase the number of transactions by HXY. This method simply increases the Profit Reduction Rate (PRR) to its limit associated with the remaining time of the market. Within this algorithm, the PRR begins from 0% then linearly increases to the Limit Profit Reduction Rate at the end time.

Moreover, another method was raised, which adjust quote price based on the current best price providing by LOB. The calculation is different between bids and asks operation. First, for the type of order is ‘Bid’, there are two situations considered. If the best bids price is lower than limit price of the order, the new quote price is the Geometric Mean (GM) of best bids price and old quote price. If the best bids price is greater than limit price of the order, the new quote price is the geometric mean of limit price and the old quote price. Similarly for ‘Ask’ order, if the best asks price is greater than limit price, the new quote price is the geometric mean of best asks price and quote price; if the best asks price is lower than limit price, the new quote price is the geometric mean of limit price and quote price. The purpose of this method is to obtain a new quote price that closer to the smaller element in two factors and geometric mean calculates a meaningful quote price under effect of current best price.

Table 4 shows the total profit of versions of HXY agents with different profit reduction methods and number of total trades in the market.

#Trades	300	1000	2000
	HXY		
PRR: 0	117.13	406.28	814.47
PRR: 1/2	117.13	381.28	797.82
PRR: 1/3	115.33	399.23	795.67
PRR: 1/4	115.80	396.62	792.66
PRV: GM	126.51	426.24	856.73

Table 4: profits with different profit reduction methods

From the above table, the strategy of GM has achieved the highest overall profit.

7. Experiments setup and results

After all previous modifications have been applied to HXY; a series of experiments should be processed. Next, we will discuss about the improvements we have achieved from those modifications by providing different experiment results.

Table 5 shows the performance of original GD against different robots in 1 vs. 1 market, that is, there are only two types of robots existing in the market at a time. In this experiment, the amount of GD and another robot is equivalent in both buyers and sellers (e.g. 20 GD buyers and sellers, 20 GVWY buyers and sellers) and 100 market sessions will be run. There are about 300-400 transactions in each session, which should be large enough for GD.

GD vs.	Wins	Transactions involved	Avg. Profit per session
GVWY	91	28864	7348
SHVR	61	27124	7033
SNPR	81	18950	4232
ZIP	79	27910	6949
ZIC	99	26594	6865

Table 5. The performance of original GD in 1 vs.1 market. N.B. each selling or buying operation counts 1 in Transaction involved

In contrast, **Table 6** shows the performance of HXY (in completely same situation as above experiments, and more discussions will be put forward later).

HXY vs.	Wins	Transactions involved	Avg. Profit per session
GVWY	83	35554	7681
SHVR	77	31442	7612
SNPR	86	19168	4393
ZIP	82	33540	7447
ZIC	100	33340	7795

Table 6. The performance of HXY in 1 vs. 1 market. N.B. each selling or buying operation counts 1 in Transaction involved

From **Table 5** and **6** we can see that the number of wins against the same type of robot did not change much, however the number of transactions involved and average profit rose significantly. In our opinion, the reason is that we applied time-based profit expectation and profit reduction method as shown in section 4, 5 and 6. Although we sacrificed a part of single-trade profit, we can have transactions more frequently thus we dealt more orders than GD did in a certain period of time. As a result, HXY obtained more profits than GD from the market.

In order to be more intuitive to compare GD and HXY and to simulate real situations further, we set up all different types of robots together in one market and ran 100 times. This experiment showed the competition between each robot in a better way.

Robots	Wins	Transactions involved	Avg. Profit
GD	20	32397	8006
HXY	34	36677	8237
GVWY	14	38808	7909
SHVR	31	36833	8066
SNPR	0	8414	1808
ZIP	1	34398	7461
ZIC	0	17175	4495

Table 7. The performances of different types of robots. N.B. each selling or buying operation counts 1 in Transaction involved

From **Table 7**, it can be said that HXY performed better than original GD. In our opinion, the improvement from GD to HXY is that HXY pays more attention on long-term benefits while GD always attempts to maximize the profit of a single order. HXY made a balance between one-time income and overall income, which brought more profit to us.

Figure 4 shows the profits of different robots in typical market run.

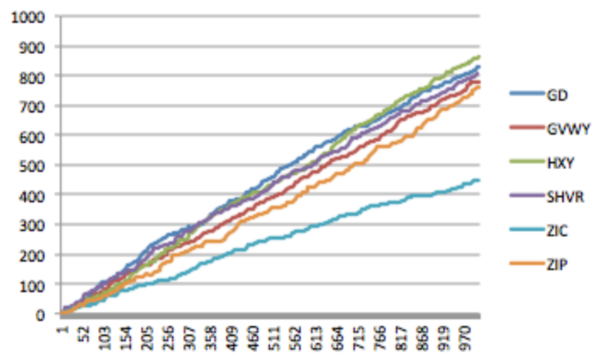

Figure 4: profits summary, x-axis is the number of transactions and y-axis is the unit of profit

8. Conclusion

In conclusion, a new strategy based on GD, HXY, that out performs other robots in BSE has been implemented through several improvements and trade-offs.

We started as the original GD algorithm because of its ability of learning from history. In the decision of history size, comparisons were made to find out the best option for the market.

We then reconsidered the potential profit of trades at certain prices. Instead of using the measurement of original profit expectation, we implemented a valuator that valuates different prices based on time and profit rate.

In order to further improve the performance of HXY in the early market, we enabled the agent to find out the optimum quote price with quadratic function mapping. By doing so, the problem that GD can only choose its quote price from the history was solved as well.

Next, we carried out experiments on the trade-off between single-trade profit and trading efficiency. By comparing the profit earned by different versions of trading robots with different profit reduction methods, a conclusion was drawn that using GM as the profit reduction method can maximize the overall profit.

Finally, we put all agents (same number for each algorithm) altogether into market runs to examine the performance of HXY in competition. From statistics, HXY has led to the most profitable agents.

8. Reference

[1] Steven Gjerstad and John Dickhaut. Price Formation in Double Auctions. *Games and Economic Behaviour*, 22(1):1 – 29, 1998.

[2] Dave Cliff. Minimal-Intelligence Agents for Bargaining Be- haviours in Market-Based Environments. Technical report, June 1997.

[3] Gerald Tesauro and Rajarshi Das. High Performance Bidding Agents for the Continuous Double Acution.

9. Appendix

```
class Trader_HXY(Trader):
        def __init__(self, ttype, tid, balance):

                self.tid = tid
                self.ttype = ttype
                self.balance = balance
                self.blotter = []
                self.orders = []
                # this is the transaction history recorded by this trader
                # only trading price will be recorded for each transaction
                # most recent trading price will be inserted at the begining
                # of the list
                self.history_transac = []



        def getorder(self, time, countdown, lob):

                # Get the acceptance possibility of a price existing
                # in the transaction history.
                # Params. price: target price
                def getP(price):
                        otype = self.orders[0].otype
                        # History limitation is set to 1/5 of the total length
                        m = int(len(self.history_transac)*0.2)
                        if otype == 'Bid':
                                success = 0.0
                                for i in range(0,m):
                                        value = self.history_transac[i]
                                        if value<=price:
                                                success+=1
                                if m == 0.0:
                                        return 0.0
                                else:
                                        return success/m

                        if otype == 'Ask':
                                success = 0.0
                                for i in range(0,m):
                                        value = self.history_transac[i]
                                        if value>=price:
                                                success+=1

                                if m == 0.0:
```

```
                                return 0.0
                        else:
                                return success/m




                # Calculate expectation of a price
                # Params. price: target price
                #        profit: the profit gains by target price
                #        profit_rate: the profit rate (profit/limit price) of a target price
                def getE(price,profit,profit_rate):
                        possibility = getP(price)
                        return possibility**3*profit*profit_rate






                # Calculate the best-guess price based on known expectation
                # of each price in transaction history
                # Params. knownlist: a list of [expectation,price] for each target price
                #                  i.e. [[e1,p1],[e2,p2]...]
                #        knownbest: the highest expectation with its price in knownlist
                #                  i.e. [e,p]
                def getbgprice(knownlist,knownbest):
                        knownlist.sort(key=lambda x:x[1]) # sort by price
                        position = knownlist.index(knownbest) # position of known best price
#                        print 'kb position',position,'list len',len(knownlist)
                        kbprice = knownbest[1] # get price of know best E-price item
                        kbE = knownbest[0] # get E of know best E-price item
                        bgprice = kbprice # initialize the hidden best price to known best price
                        if position!=0 and position!=len(knownlist)-1: # if known-best price is not at first
and last position in list
                                left = knownlist[position-1] # get E-price item by shifting left of 1
                                right = knownlist[position+1] # get E-price item by shifting right of 1
                                leftprice = left[1] # get price of left E-price item
                                leftE = left[0] # get E of left E-price item
                                rightprice = right[1] # get price of right E-price item
                                rightE = right[0] #get E of right E-price item

                                # now calculate the best gest price (bgprice)
                                part1 =
(kbE-rightE)*(leftprice**2-kbprice**2)-(leftE-kbE)*(kbprice**2-rightprice**2)
                                part2 =
2.0*((leftE-kbE)*(rightprice-kbprice)-(kbE-rightE)*(kbprice-leftprice))

                                if part2 !=0:
```

```
                    bgprice = part1/part2
                    # according to assumption, bgprice should be between leftprice and
rightprice
                    if not(bgprice>leftprice and bgprice<rightprice):
                        print 'bgprice error'

            return bgprice



        # Calculate the quote price using geometric mean method
        # a gmprice will be returned if there was a best price
        # in LOB, otherwise the quote price will remain the same
        # Params. price: the quote price we have got so far
        def getgmprice(price):
            otype = self.orders[0].otype
            # limit price of the order
            limitprice = self.orders[0].price
            # quote price calculated by gemotric mean method
            gmprice = 0
            if otype =='Bid':
                # best bid price in LOB at present
                bidbestprice = lob['bids']['best']
                if bidbestprice!=None:
                    if bidbestprice<limitprice:
                        gmprice = (bidbestprice*price)**(0.5)
                    if bidbestprice>limitprice:
                        gmprice = (limitprice*price)**(0.5)
                    return gmprice
                else:
                    return price
            if otype =='Ask':
                # best ask price in LOB at present
                askbestprice = lob['asks']['best']
                if askbestprice!=None:
                    if askbestprice>limitprice:
                        gmprice = (askbestprice*price)**(0.5)
                    if askbestprice<limitprice:
                        gmprice = (limitprice*price)**(0.5)
                    return gmprice
                else:
                    return price

        # Calculate the quote price
        def getquoteprice():
            otype = self.orders[0].otype
            limitprice = self.orders[0].price
            # if the type of oder is to bid..
            if otype == 'Bid':
                profit_price = [] # structure: a list of [expectation,price]
                temp = [] # temp list to avoid adding expectation of same price to profit_price
list

                # get each target price from transaction history
                for price in self.history_transac:
                    # ensure a target price produces profit and its expection
                    # has not been calculated yet
                    if price<limitprice and price not in temp :
                        temp.append(price)
                        profit = limitprice-price # profit made by target price
                        profit_rate = float(profit)/float(limitprice) # profit rate of
the target price

                        E = getE(price,profit,profit_rate) # expectation of the target
price

                        if E!=0:
                            profit_price.append([E,price]) # add [E,price] to the
list
                profit_price.sort(reverse=True) # sort list from higher E to lower E

                # if profit_price contains elements, which means there existing some
                #  price in history that can make profit
                if len(profit_price)!=0:
                    # get the first element in the list, which contains the highest E
                    # and corresponding price
                    knownbest = profit_price[0]
                    # price with the highest E we have got so far, if best-guess
                    # method not applied
                    kbprice = knownbest[1]

                    # calculate the best-guess price
                    bgprice = getbgprice(profit_price,knownbest)

                    # calculate the geometric mean price
                    gmprice = getgmprice(bgprice)
                    return gmprice

                # if there was nothing in the list, that means none of existing price
                # can make profit, then we quote same price as limit price
                else:
                    return limitprice

            # if the type of order is to ask, similar as 'Bid' above
```

```python
        if otype == 'Ask':
            profit_price = [] # [profit expectation,price]
            temp = []
            for price in self.history_transac:

                if price>limitprice and price not in temp :
                    temp.append(price)
                    profit = price - limitprice
                    profit_rate = float(profit)/float(limitprice)
                    E = getE(price,profit,profit_rate)
                    if E!=0:
                        profit_price.append([E,price])
            profit_price.sort(reverse=True)
            if len(profit_price)!=0:
                knownbest = profit_price[0]
                kbprice = knownbest[1]

                # calculate the best-guess price
                bgprice = getbgprice(profit_price,knownbest)

                # calculate the geometric mean price
                gmprice = getgmprice(bgprice)
                return gmprice
            else:
                return limitprice

    if len(self.orders) < 1:
        #no orders: return NULL
        order = None
    else:
        # get quote price
        quoteprice = getquoteprice()
        # pass order to LOB with quote price
        order=Order(self.tid, self.orders[0].otype, quoteprice, self.orders[0].qty, time)

    return order


# Every time a new order was placed in LOB, market_session() will invoke
# respond() function of each trader. Therefore our trader will update
# transaction history here using the LOB passed by market_session()
# This methond is rewrite from Trader class
def respond(self, time, lob, trade, verbose):

    # update transaction history
    def updatehistory():
        # trading price
        price = trade['price']
        # trading quantity
        count = trade['qty']
        # each transaction will be recored multiple times
        # if the quantity > 0. However in BSE, the quantity
        # of transaction will always be 1
        while count!=0:
            # most recent transaction will be inserted at the beginning
            self.history_transac.insert(0, price)
            count -=1

# if there was a transaction occurred in the market, then
# we update transaction history
if trade!=None:
        updatehistory()
```