A Genetic Algorithm for Retrieving Sequence Strategies

Fei Wu, Georges Gardarin
PRiSM Laboratory
University of Versailles
45 Avenue des Etats-Unis
78035 Versailles Cedex, FRANCE
Email: <Firstname>.<Lastname>@prism.uvsq.fr

Abstract. We consider the problem of finding sequence strategies in time-series databases. Users are often interested in finding out when a sequence will change from one state to another state so that some corresponding actions may be performed. A *sequence strategy* aims to predict the future trends of a sequence based on its current behaviors, and to give user some suggestions. A sequence strategy contains a set of CA (condition, action) rules. Since time-series databases are often of huge size, retrieving pertinent strategies is not a trivial work. We propose a new approach based on a genetic algorithm to solve this problem. To demonstrate the value of our method, experiments are conducted on the real French stock market.

Key words: time sequence, strategies, genetic algorithm

1 Introduction

Time-series data are of great importance in many database applications, such as data mining [DLM+98], [HDY99]. A time-series is a sequence of numbers, where each number is associated with a timestamp. Such sequence could represent stock values, exchange rates, weather data, sales, etc. The behaviors of a sequence can be expressed by several phases. For example, a stock sequence has three phases: rising, stable and descending. It is important for users to be able to determine the transition point from one phase to another phase in order to perform some actions. For instance, when a stock security changes from "stable" state to "rising" state, we will buy it. When it descends, we will sell it. A *sequence strategy* is used for determining such transition points and helps users to make decisions. To determine the states of a sequence, its *indicators* are calculated. For a stock sequence, its popular indicators are RSI[1], ROC[2], MOMENTUM[3], etc. Strategies are generated on sequence indicators. Temporal databases are often of huge size because of the large quantity of historical data. Thus mining strategies is not a trivial work.

This paper considers the problem of retrieving sequence strategies in time-series databases. More formally, a sequence strategy consists of a set of CA (condition, action) rules. An example of a strategy containing two rules is: "if the RSI of a stock is larger than 70%, then

[1] Relative Strength Index. RSI = 100 - (100 / (1 + (H/B))), where H(B) is the average of the closes of the up(down) days.
[2] ROC: Rate Of Change. ROC = (C/Cy)*100, where C is the current price and Cy is the price y days ago.
[3] Momentum(y) = C-Cy, where C is the current price and Cy is the price y days ago.

we will buy it; if the RSI of a stock is less than 30%, then we will sell it." Each strategy is associated with a gain, which indicates its fitness in the specified context. For the stock market, the fitness of a strategy is evaluated by calculating the profit by applying this strategy on a period of stock data.

The number of possible strategies grows exponentially in the number of indicators. It is very time-consuming to find the best strategy among the whole solutions. Thus what we need is rather a near-optimal solution than the best solution. To handle this problem, we proposed a Genetic Algorithm. Genetic algorithm has already been used in the data mining context [FR96], [Cho00]. To solve some financial problems, Genetic programming (GP), which is a branch of genetic algorithms, has been used. [NWD97] used GP to find the technical trading rules for the foreign exchange market. [IS99] used GP to predict the price data in Japanese stock market. The solutions are represented as a GP tree. In the contrast, we use a bitmap to represent a rule. In section 3.1, we will discuss that the bitmap structure is more suitable than a tree structure in our case.

The rest of this paper is organized as follows. Section 2 formally describes the problem of searching sequence strategies from temporal database and studies the search space generated by candidate strategies. Section 3 explains how we adapted the Genetic Algorithm to the given problem. Section 4 reports our experimental results. Finally section 5 concludes.

2 Problem Description and Search Space Study

This section formally presents the problem of mining sequence strategies in the time-series databases and estimates the search space.

2.1 Problem Statement

A sequence strategy contains a set of CA rules. A CA rule is composed of conditions and an action. If a set of conditions is satisfied, the corresponding action will be executed. Conditions are based on the sequence indicators. There are two kinds of indicators: categorical indicator and numerical indicator. For a categorical indicator, its possible elements are enumerated by a set of integers $e = \{e_1, e_2, ..., e_n\}$, $e_i \neq e_j$, $1 \leq i \neq j \leq n$. For a numerical indicator, its values are partitioned into m intervals. The intervals are then mapped to a set of consecutive floats $f = \{f_1, f_2, ..., f_m\}$, $f_q < f_p$, $1 \leq q < p \leq m$, such that the order of the intervals is preserved. In this way, a categorical indicator *ind_cat* is regarded as a set of pairs $\langle ind_cat, e_1\rangle$, $\langle ind_cat, e_2\rangle$, … , $\langle ind_cat, e_n\rangle$. A numerical indicator *ind_num* is treated as a set of pairs $\langle ind_num, f_q, f_p\rangle$, which means the value of *ind_num* is between f_q and f_p.

Let C be a boolean expression in a disjunctive normal form (DNF) expression: $C = C_1 \vee C_2 \vee \ldots$. Suppose there are in total *ind* indicators. Each $C_i = \alpha_1 \wedge \alpha_2 \wedge \ldots \wedge \alpha_{ind}$, where a sub-condition α_i corresponds to an indicator. If $\alpha_i = 0$, this means the corresponding indicator is not taken into account. Let $A = \{a_1, a_2, \ldots, a_k\}$ be a set of actions. A rule R is an implication of the form $C \Rightarrow a$, where $a \in A$. A strategy is composed of several rules $\{C1 \Rightarrow a_i; C2 \Rightarrow a_j; \ldots\}$, $1 \leq i, j \leq k$. An example of a strategy for stock market is: { ⟨RSI, 0.7, 0.9⟩ => *Buy*; ⟨RSI, 0.1, 0.3⟩ => *Sell* }. This strategy contains two rules. When the RSI of a stock is between 0.7 and 0.9, we will buy it. If the

RSI of a stock is in the range of 0.1 and 0.3, we will sell it. To each strategy is associated with a gain *G* that indicates the fitness of this strategy in the given context.

Given a period of data, the problem of mining sequence strategy is to find the strategy of a sequence that maximizes its gain.

2.2 Search Space Estimation

This section investigates the size of the solution space generated by candidate strategies.

A strategy contains *n* rules. The search space of a strategy is the number of combinations of all possible rules $C \Rightarrow a$. Condition C is of the form $C_1 \vee C_2 \vee \ldots$, and $C_i = \alpha_1 \wedge \alpha_2 \wedge \ldots \wedge \alpha_{ind}$ where *ind* is the total number of indicators. Suppose an indicator has *p* possible values. The total number of C_i is $\sum_{i=1}^{ind} p^i C_i^{ind} = (p+1)^{ind} - 1$, where C_i^{ind} is the number of combinations, i.e., $C_i^{ind} = \frac{ind!}{i!(ind-i)!}$. The total number of all possible C is $((p+1)^{ind} - 1) * ((p+1)^{ind} - 1) * \ldots$. For simplify, we suppose $C=C_1$. Given p =10, the following table presents the minimum number of conditions in the number of indicators.

nb. of indicators	search space
1	10
2	120
3	1330
4	14640
5	161050

Table 1: Size of the Search Space

Note that some constraints should be imposed when combining rules. For example, if two rules for a strategy concern the same indicator, their intervals (or elements) should not be intersected (or the same). That means, for $\langle y, v1, v2\rangle$, $\langle y, v3, v4\rangle$, we have $v2 < v3$. The rule $\langle y, 10, 30\rangle \Rightarrow Buy$ and the rule $\langle y, 20, 50\rangle \Rightarrow Sell$ are not allowed in a strategy.

3 A Genetic Algorithm for Retrieving Strategies

In this section, we describe our adaptation of the Genetic Algorithm for searching sequence strategies in time-series database.

Genetic Algorithm (GA) is a non-gradient optimization algorithm used for the search of local extremes (minimum or maximum) of functions with many variables and functional extremes [Hol75], [Gol89]. It is a randomized search algorithm that simulates the natural evolution process: the fittest individuals in the population are most likely to survive. This kind of search algorithm works on a population of solutions in place of only one solution. Two operators Mutation and Crossover are performed to generate the next generation. The Mutation operator allows adding new solutions in the population. The Crossover operator takes two solutions as parent and generates their children by exchanging their common parts. Every solution is

evaluated by a fitness function. At the end of each generation, the best solutions are chosen and kept for the next generation. This process continues until the number of generations (iterations) is reached to a user-defined threshold. The best solution found so far is returned as the final result. The **GA** for our task is illustrated in Figure 1.

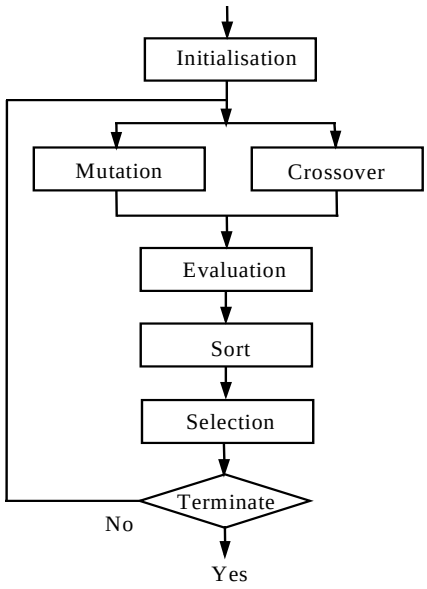

Figure 1: Genetic Algorithm Procedure

The main blocks of the procedure are explained as follows:

- **Initialization** - randomly generate an initial population BasePopu of strategies from the whole search space.

- **Mutation** - choose one strategy from the population, and transform this strategy to a new one by changing its conditions or its action.

- **Crossover** - randomly choose two strategies, that are not muted, from the population, and exchange their sub-conditions or actions according to some constraints in order to generate two new strategies. The new generated strategies are kept with their parents.

- **Evaluation** - for each strategy, evaluate its benefit by applying it to a period of data.

- **Sort** - sort all strategies according to their gains.

- **Selection** - choose the best BasePopu strategies from the result of Sort as the population of the next generation.

- **Termination** - check termination criterion for stopping searching. A possible criterion is the number of generations.

The heart of the **GA** is the generation of a new population of solutions based on Crossover and Mutation operations. These two transformations are the basis for the ability of the **GA** to find better solutions. In our case, the Mutation and Crossover on strategy transformations must follow specific constraints to derive pertinent strategies from existing ones; it will be explained in the next section. The best strategy found at every generation is a local best answer. The global best strategy is the best one among all local best solutions. At the end of each iteration, the local best strategy is compared to the global one. If the local best strategy is better than the global one, this local best strategy will be set as a global best strategy. All strategies are sorted, and only the first BasePopu strategies are kept for the next generation (iteration).

In the following sections we will describe our implementation of gene generation, mutation, crossover and evaluation procedure.

3.1 Gene Generation

Initially, a number of populations are randomly generated. Each individual in the population represents a strategy. For every strategy, the number of rules contained is generated randomly. For every rule, the indicators concerned and their corresponding values, i.e., a sub-condition are chosen randomly. A gene is either a sub-condition or an action.

A sub-condition item is represented as a bit-vector. Bitmap technique has already been used from database query problem [OG95], [CI99], data warehousing [WB98] to data mining domain [GPW98]. A bitmap represents either an element for a categorical indicator *x* or an interval for a numerical indicator *y*. For categorical indicators, a bit corresponds to an element in the set. For example, suppose a categorical indicator “State” has three elements: rising, stable and descending. The bitmap 001 indicates a state “descending”. For numerical indicators, a bit represents an interval. How to determine the intervals is another problem. We consider that this needs domain knowledge. In our application, we divide each numerical indicator equally. Suppose a numerical indicator whose minimum and maximum are 10 and 50 respectively. By dividing it into 4 intervals, 4 bits are needed. In this way, the bitmap 1000 defines an interval <10, 20> and the 0001 means <40, *max*>, where *max* in this example is 50. It is easy to see that the bitmap 0110 represents the range < 20, 40>. Some constraints must be imposed to make a bitmap valid. For categorical indicators, only one bit is allowed to be set in the vector. For quantitative indicators, its bits must be consecutive. That means a bitmap like 1101 is invalid.

Actions are in fact a list of enumeration. A set of integers is used to represent them. We use an array of integers to represent a condition. Each integer represents a sub-condition on an indicator. If an integer is 0, this means the corresponding indicator is not concerned in a condition. An example of a condition in Figure 2 shows that only indicator1 and indicator2 are

considered, but not indicator3. Suppose the interval borders of the first indicator are: 10, 20, 30, 40, 50, and the second are 10, 20, 30, 40. The condition in the Figure 2 is interpreted as *{<indicator1, 20, 40> ∧ <indicator2, 10, 30>}*.

At the initial, all sub-condition items are generated. Then they are randomly joined to construct a condition according to some constraints. An action is randomly selected for this condition in order to construct a rule.

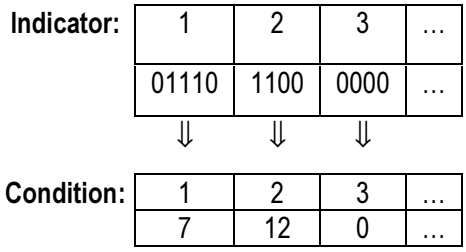

Figure 2: An Example of Condition

A tree structure can also be used to represent a rule [NWD97]. However, a bitmap is more suitable than a tree in our case. For example, to interpret the condition in the Figure 2, a corresponding tree structure is illustrated in Figure 3.

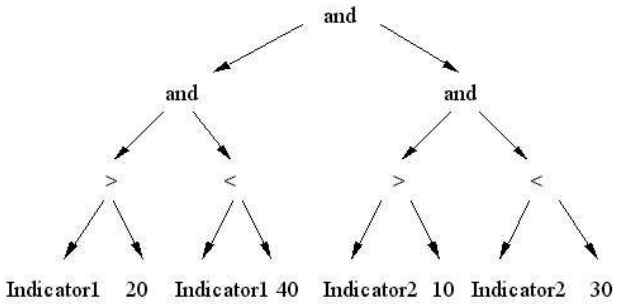

Figure 3: GP tree

3.2 Mutation Operator

Mutation is one of the major genetic operations. A certain percentage of the strategies in the population will do Mutation. In this procedure, a sub-condition item of a condition or an action is randomly selected and another sub-condition item (or another action) is randomly chosen to replace it. An example is depicted in Figure 4. The operation Mutation may bring some new genes that do not exist in the current population. We could also remove a sub-condition by setting its corresponding integer to 0.

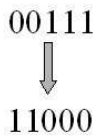

Figure 4: Mutation Operator

3.3 Crossover Operator

While part of the population go through the Mutation procedure, the rest perform the Crossover operation. Two strategies are randomly chosen. Their common sub-condition items are exchanged in order to generate two new strategies. Figure 5 shows two examples by exchanging the common indicators of two conditions. Indicators to exchange are randomly chosen too. While the Mutation operator works on one indicator, the Crossover operator may lead to the change of several indicators in one step.

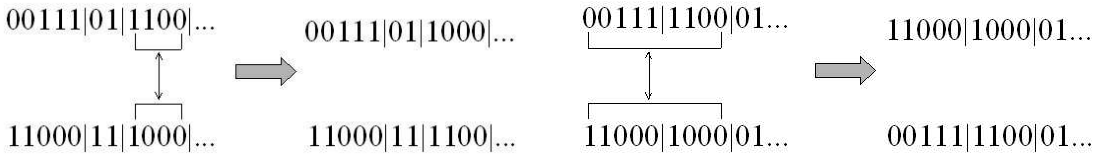

Figure 5: Crossover Operator

3.4 Selection Process

Unlike the Mutation operator, new strategies generated during the Crossover procedure do not replace their parents. Thus the population will become crowded at the reproduction stage of Crossover as it consists of both parents and offspring. Then, before the next generation, a selection is necessary to perform in order to restrict the population. The choice of survivals is determined by their relative fitness to the environment.

To evaluate a strategy, we apply it on a period of data. The benefit we obtain at the end of the period signifies the fitness of this strategy. Consider a strategy on stock sequence:{ ⟨RSI, 0.7, 0.9⟩ => *Buy*; ⟨RSI, 0.1, 0.3⟩ => *Sell* }. Given a part of stock data ordered by time in table 2. The column "Day" indicates the open days; "Close" is the closing price of an operational day, and the RSI is an indicator. We begin from the first day. The RSI is 0.8. According to the example strategy, we will buy this security the next day: day 2. Suppose the default trading quantity is 100 share. $7*100 = $700 is spent on the second day. On the fourth day, the RSI decreases to 0.22, so all shares we hold will be sold on the fifth day, which makes us earn $15*100 = $1500. The gain of the example strategy is then evaluated as $1500 – $700 = $800.

Day	Close	RSI
1	8	0.8
2	***7***	***0.6***
3	10	0.45
4	12	0.22
5	***15***	***0.31***

Table 2: A Part of Stock Data

3.5 Algorithm

Figure 6 gives the pseudo-code of our adapted **GA** algorithm. Every strategy contains two variables: a vector of integer arrays, and the gain of this strategy. Each vector represents a rule.

The parameter *Iteration* determines the number of generations to propagate. The parameter *BasePopu* indicates the number of strategies retained in a population. We use the array structure *Popu* to implement the whole population of strategies after the Crossover. Parameter *Part* is the percentage of the *BasePopu* population doing the Crossover. Evidently, BasePopu * (1 + Part) = Popu. Since the population dynamically increases after the Crossover, a selection procedure is executed before entering the next generation. The offspring of the Crossover procedure are placed from (BasePopu + 1) until Popu. After the Crossover operation, a sort procedure is performed. Thus the best BasePopu solutions can be selected automatically. At the end of each generation, the local best strategy is compared to the global best strategy. If the local best one is better than the global one, we will set this local best strategy as the global best solution.

```
Procedure GA(Iteration, BasePopu, Part) {

 Generate the initial population: strategies[BasePopu] ;
 sort(strategies) ;
 OptimalStra = strategies [0]; // Keep the best strategy

 FOR (i = 0; i < Iteration; i++) DO {
       number = 0;

       // Apply Crossover to Part of the population
       WHILE number < Part * BasePopu DO {

             // Randomly choose two strategies for Crossover
             s1 = strategies [random(BasePopu)] ;
             s2 = strategies [random(BasePopu)] ;
             Crossover(s1, s2);
             s1.treated = s2.treated = TRUE;
             number++;
             strategies[BasePopu + number] = s1;
             number++;
             strategies[BasePopu + number] = s2;
       }

       // Apply Mutation to the rest
       FOR (i = 0; i < BasePopu; i++) DO {
             IF NOT strategies[i].treated {
                   Mutation(strategies[i]);
             }
       }

       FOR (i=0; i < Popu ; i++) DO
             evaluate(strategies [i]);

       sort(strategies);

       IF (strategies [0].gain > OptimalStra.gain)
             OptimalStra = strategies [0];
 }

 return OptimalStra;

}
```

Figure 6: Implementation of our Genetic Algorithm

4 Experimental Results

Our GA algorithm was implemented in JAVA 1.2. The experiments were done on a PC running Windows NT 4.0 with Pentium II 300M Hz and 64M memory. The data set is extracted from the French stock market Règlement Mensuel. The stock data used for evaluating a strategy is Total Fina Company from the 1st December 1999 to the 1st May 2000. The data is divided into two parts. The first part is used to retrieve strategies, and the second part is used to evaluate the best strategy returned. The first part contains data from the 1st December 1999 to the 1st February 2000. The second part contains stock data from the 1st February 2000 to the 1st May 2000. After having found the best strategy from the first part of data, this strategy is applied on the second part of data in order to evaluate the gain the generated strategy can get. Eight indicators are taken into account in our experiment. The interval of each indicator is defined by users. In our experiment, each indicator is divided into 10 intervals. The following table explains the GA parameters used.

BasePopu	Number of population within a generation
Part	Percentage of population to do Crossover
Iteration	Number of iterations (generations), stopping_condition

Table 3: Parameters

For our GA algorithm, three parameters are needed to tune. They are (1) the stopping_condition, i.e., the number of iterations (generations), (2) the number of individuals in a population BasePopu and (3) the percentage Part of population that will do Crossover. Table 4 to 6 compare the best strategies found, time consummation and evaluation results by tuning these three parameters. Table 4 changes the number of iterations from 30 to 100 by setting BasePopu to 10 and 40 and Part to 0.2. Table 5 increases the BasePopu from 40 to 100 by fixing the number of iterations to 50 and Part to 0.2. The results of changing Part from 0.2 to 0.9 are shown in table 6, where Iteration and BasePopu are set to 20 and 40 and 30 and 50 respectively.

Table 4 shows that increasing the number of iterations can help to find better solutions. This is because more new genes (sub-conditions) will be added into the population during the reproduction, and more rule combinations are generated. However, by comparing the table 4(a) and 4(b), we can see that it is more efficient when the population is small. It is also possible that the best strategy retrieved from the first part of data is not a good one. Let us take an example in table 4(a). The model returned by iteration 50 is better than that of iteration 40, which has gain 1712 against 1469. While if we apply these two models on the second part, the gain of the model of iteration 50 is –19547, but that of the iteration of 40 is 840. That is why every model retrieved should be evaluated on another data in order to prove its correctness. We observed that the cost of GA algorithm augments linearly in the number of the iterations.

Iteration	Gain	Time (Milliseconds)	Evaluation
30	1469	3856	840
40	1469	5057	840
50	1712	6199	-19547
60	6375	7430	6290
70	6375	8552	6290
80	6375	9654	6290

Iteration	Gain	Time (Milliseconds)	Evaluation
30	2873	15302	-37685
40	3574	20018	3142
50	3574	24896	3142
60	3574	29803	3142
70	3574	34530	3142
80	3574	39272	3142

90	6375	10986	6290
100	6375	12358	6290

(a) BasePopu=10, Part=0.2

90	3574	44304	3142
100	3574	49000	3142

(b) BasePopu=40, Part=0.2

Table 4: Changing Iteration

Table 5 aims to study the impact of BasePopu on the GA result. We can see that the number of population impacts the quality of the best strategy returned. In fact, having more candidates allows to better cover the search space, which avoids our algorithm to be limited to a local minimum. When the number of population is small, the answer strategy is not very satisfying (BasePopu < 30). By generating more candidates, a strategy whose benefit is "3575" is retrieved and its gain on the second part is "3142". Notice that when BasePopu is equal to 60, the best result returned lost "9052" on the evaluation. However although the last model generated (BasePopu=100) has only "2086" gain, its evaluation reaches "4710".

BasePopu	Gain	Time (Milliseconds)	Evaluation
10	1712	6199	-19547
20	3233	12217	-19344
30	2315	18176	905
40	3575	24966	3142
50	4867	31004	-17672
60	5097	37093	-9052
70	2230	43022	2801
80	3076	48330	1456
90	3705	55680	-97619
100	2086	62310	4710

Table 5: Tuning BasePopu (Iteration=50, Part=0.2)

Table 6 shows that increasing the Part can also provide better solutions. Two tests have been conducted. One test sets BasePopu to 20 and Iteration to 30 and the other with BasePopu 40 and Iteration 50. When the population is small (table 6a), the results are improved by increasing the Part. However, by passing more individuals to do the Crossover operation (Part >= 0.8), the quality of the result decreases. The reason is that when more candidates doing Crossover, less new genes (sub-conditions) will be introduced in the population. While when the population is big (table 6b), a good model is also found with the Part set to 0.9. This is because when we have enough good genes, the Crossover is more efficient than the Mutation operator. That also explains the reason that the models generated in table 6b are better than that of table 6a.

Part	Gain	Time (Milliseconds)	Evaluation
0.2	3233	7311	-19344
0.3	1620	8302	584
0.4	2814	8492	-28613
0.5	1495	8933	-151
0.6	4756	10094	1430
0.7	4756	10234	1430
0.8	4756	10766	1430

Part	Gain	Time (Milliseconds)	Evaluation
0.2	3575	25196	3142
0.3	4992	27299	2545
0.4	5045	30033	2177
0.5	7642	33789	5825
0.6	7642	35491	11604
0.7	7143	37244	6022
0.8	5431	38920	-11511

0.9	2787	11807	-19593

(a) BasePopu=20, Iteration=30

0.9	9465	40158	7300

(b) BasePopu=40, Iteration=50

Table 6: Changing Part

5 Parallelism Discussion

With the development of parallel database system, efficient parallel algorithms are widely demanded. Our GA algorithm can be easily parallelized in two ways:

- Inter-Parallelism: The Mutation and Crossover operations are independent of each other. Therefore they can be performed parallely. The idea is to first divide the whole solutions into two groups. Group1 contains those that will do Mutation and the rest that will do Crossover is in group2. Then the Mutation is performed on the group1 and the Crossover is run on the group2 at the same time. It is also possible to parallelize the evaluation phase, since read operation is considered. In similar way, the population is partitioned into several groups, and the finesses of the individuals in these groups are calculated parallely. After these groups finish computing, all strategies are gathered together and the next generation is chosen.

- Intra-Parallelism: Another possible parallelism of our GA algorithm is to run several GA algorithms parallely. For each GA is generated a different population. At the end of the evaluation phase, candidate strategies generated by each GA algorithm are gathered together. Suppose there are *n* GA algorithms currently running and each algorithm works on *Popu* solutions. In total we have *n*Popu* solutions. These candidates are sorted and *n*BasePopu* are selected to survive. Then these *n*BasePopu* strategies are randomly distributed as the next generation for each GA algorithm. This procedure helps to introduce new genes from another population to a generation in addition to the Mutation operation.

6 Conclusion and Future Work

This paper introduces the problem of discovering sequence strategy in temporal databases. A strategy is composed of several rules. The conditions of a rule are based on the sequence indicators. Since the search space grows exponentially in the number of the indicators, retrieving an optimal strategy is not an easy work to tackle. We proposed a genetic algorithm to solve this problem. Two versions of parallel GA algorithms are also discussed. Another interesting problem is to add time notion, for example: If the RSI of a stock is in the range of 0.1 and 0.3, we will sell it within 5 days. It will be interesting to test our genetic algorithm on other time series data sets. Our genetic algorithm is currently integrated to a financial trading system ContextBourse implemented in JAVA.

References

[CB94] Munir Cochinwala and John Bradley. *A Multidatabase System for Tracking and Retrieval of Financial Data*. In Proceedings of 20th International Conference on Very Large Data Bases, September 12-15, 1994, Santiago de Chile, Chile, pp 714-721.

[Cho00] Sunil Choenni. *Design and Implementation of a Genetic-Based Algorithm for Data Mining*. In Proceedings of 26th International Conference on Very Large Data Bases, September 10-14, 2000, Cairo, Egypt, pp 33-42.

[CI99] Chee Yong Chan, Yannis E. Ioannidis. *An Efficient Bitmap Encoding Scheme for Selection Queries.* In Procedding of ACM SIGMOD International Conference on Management of Data, June 1-3, 1999, Philadephia, Pennsylvania, USA. pp 215-226.

[DLM+98] Gautam Das, King-Ip Lin, Heikki Mannila, Gopal Renganathan and Padhraic Smyth. *Rule Discovery from Time Series*. In Proceedings of the Fourth International Conference on Knowledge Discovery and Data Mining (KDD-98), August 27-31, 1998, New York City, New York, USA, pp 16-22.

[FR96] Ian W. Flockhart and Nicholas J. Radcliffe. *A Genetic Algorithm-Based Approach to Data Mining*. In Proceeding of the Second International Conference on Knowledge Discovery and Data Mining (KDD-98), August 4-8, 1996, Portland, Oregon, pp 299-302.

[GPW98] Georges Gardarin, Philippe Pucheral, Fei Wu. *Bitmap Based Algorithms For Mining Association Rules.* In Proceeding of the 14th Base de Données Avancées (BDA), 26-30 October 1998, Hammamet, Tunisie, pp 157-176.

[Gol89] D. E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*, Addison-Wesley, Reading, MA, 1989.

[HDY99] Jiawei Han, Guozhou Dong and Yiwen Yin. *Efficient Mining of Partial Periodic Patterns in Time Series Database*. In Proceedings of the 15th International Conference on Data Engineering, 23-26 March 1999, Sydney, Australia, pp 106-115.

[Hol75] J.H. Holland. *Adaptation in Natural and Artificial Systems*. University of Michigan Press, Ann Arbor, MI, 1975.

[IS99] Hitoshi Iba and Takashi Sasaki. *Using Genetic Programming to Predict Financial Data*. In Proceedings of the 1999 Congress on Evolutionary Computation (CEC99), pp 244-251.

[NWD97] Christopher Neely, Paul Weller and Robert Dittmar. *Is Technical Analysis in the Foreign Exchange Market Profitable? A Genetic Programming Approach*. Journal of financial and Quantitative Analysis, vol. 32, no. 4, 1997.

[WB98] Ming-Chuan Wu, Alejandro P. Buchmann. *Encoded Bitmap Indexing for Data Warehouses.* In Proceeding of the 14th International Conference on Data Engineering (ICDE), February 23-27, 1998, Orlando, Florida, USA, pp: 220-230.