University of London
Imperial College of Science Technology and Medicine
Department of Computer Science

Visualised Parallel Distributed Genetic Programming

Applied on Stock Market Prediction

Final Report

Chao Yan

Sep, 2006

Supervisor: James Jacobson

Second Marker: William J. Knottenbelt

Submitted in partial fulfilment of the requirements for

the MSc Degree in Computing Science of The University of London and for

the Diploma of Imperial College of Science, Technology and Medicine.

Contact: chaoyan@gmail.com
More information on this project is available at project webpage:
http://chaoyan.googlepages.com/geneprofit

Content

Introduction | 1

Genetic Programming (GP), as an extension of genetic algorithms, has been demonstrated to be an effective problem solving tool for tackling complex optimization and machine learning problems. In GP programs are expressed as parse trees to be executed. This form of GP has been applied successfully to a number of difficult problems like automated design, pattern recognition, robot control, classification, and many other areas.

When applied to business environment, especially capital market, GP offers not merely an alternative but a much advanced solution for market prediction against GAs. Because GP is actually evolving "strategies", not formula coefficients, to assist investment decisions. However, the process of stock market prediction requires high computational resources, e.g., large memory and search times. Thus a variety of algorithmic issues are being studied to design efficient GPs.

Distributed computing applications attempt to harness the computing resources over network, which are believed able to offer a great amount of computational power and increase the overall rate of resource utilization at large. This project tries to exploit the inherent parallelism of GP by creating an infrastructure necessary to support distributed information processing using existing Internet and hardware resources. Instead of evolving the entire population using a single processor, we present here a distributed genetic programming engine which maintains multiple independent subpopulations interacting asynchronously using a controlled coarse-grain topology. This allows not only a better exploration of the global search space, but also delays premature convergence by introducing, via migration, diversity from other demes.

1.1 Visualised Distributed Genetic Programming Engine

A general purpose distributed genetic programming engine, called MetaDGPEngine is implemented at first. Stock prediction is merely one example of applications based upon this engine later. This would allow better flexibility and encapsulation. Particularly for example, each client node is for general purpose, which does not need to be recompiled for different applications. Only server side needs to define problem space, parameters, methods, and operations. Also a fast vector 2D graphical component (MetaGUIEngine) is designed on purpose for visualising and controlling the whole structure and process. It is designed separated from MetaDGPEngine as a dispensable component but without lacking ease of use: once assign to a MetaDGPEngine, it will visualise and communicate with MetaDGPEngine automatically.

Deep studies on infrastructure and implementation of this generally purpose engine set has been done. As in this project, stock market prediction itself provides a successful example.

1.2 GeneProfit

Built upon my MetaDGPEngine, GeneProfit is a commercial standard software package that automates all necessary tasks for stock market prediction and self-training. From its graphical interface, it allows user to create different investment preference sets for any chosen stock symbol. On behind, it maintains SQL database for financial data and a virtual gene bank; it updates financial data from public sources (Yahoo™ Finance!), does all the statistics for chosen technical indicators, and trigger training process via MetaDGPEngine.

Backgrounds | 2

The original incentive of this project is to implement a comprehensive market analysis and prediction platform, with fast running speed, better prediction result and ease of use. To some extent, all that machines trying to do is no more than simulating a trader's brain. And obviously differences exist. There is something machine can not do as a natural person; however, there are also advantages of machine learning: the ability of massive calculations and data storage. Hence, I start from theoretical investment analysis methodology research based on my knowledge from a previous finance degree, trying to seek a perspective suitable for machine learning. Fortunately, technical analysis, i.e., trying to evaluate those numerical indicators has always been playing a big part of a trader's everyday job, and numerical analysis is also strength of computer.

After figuring out this intersection, we need to develop a proper solution to make use of machine's advantages. Long ago, I have worked on an implementation of Santa Fe Institute Artificial Stock Market Model based on genetic algorithm. Other than the fact that the model is 17 years old after first proposed in 1989 (Palmer et al. 1994), it also lacks in features come from restrictions of GAs. Therefore, Genetic programming is adopted, because GP is actually evolving "strategies", not formula coefficients. This is more reasonable for investment analysis.

Subsequently, performance issue emerged. It takes GPs on averages about several minutes for an AMD 64bit processor to find a perfect ant route (classic problem along with the first publication of GP by Koza) on a 20x20 map. Obviously for evaluation of two years historical data and technical indicators with full set of arithmetic and logical operators on stock market prediction, takes very much longer! Buying a super computer is an easy way out, but only for institutions not for profit-driven traders. Since PCs are getting much cheaper nowadays, distributed computing is considered as the best solution to encounter performance problem. Therefore, more researches had been done for this project on developing a suitable structure/topology of a distributed genetic programming.

2.1 Stock market investment analysis

Conventionally, there are two essential approaches of analyzing investments: *fundamental analysis* and *technical analysis*. In terms of stock market, fundamental analysis refers to investigating listed companies' financial reports, related industry's trend, general economical environment, etc. Fundamental analysis highly depends on sources and timing of information. Its accuracy can be easily biased by information asymmetry. Therefore it is usually considered as tools for long-term investment analysis.

Technical analysis focuses exclusively on the stocks' historical data, asking what does its past behavior indicate about its likely future price behavior. Technicians, chartists or market strategists, as they are variously known, believe that there are systematic statistical dependencies in stock returns - that history tends to repeat itself. They make price predictions on the basis of published data, looking for patterns and possible correlations, and applying rules on charts to assess 'trends', 'support' and 'resistance levels'. From these, they develop buy and sell signals. Due to its high sensitivity, technical analysis usually applied to short term prediction.

This project takes technical analysis as entry point, trying to exploit programming algorithm's advantage on quantitative analysis. Therefore, discussion on Efficient Market Hypothesis provides the fundamental theory that proves the effectiveness of technical analysis hence the *significance* of this research at theoretical level.

2.1.1 Efficient Market Hypothesis

In a controversial paper to the Royal Statistical Society (1953), Maurice Kendall suggested that prices of stocks and commodities seem to follow a random walk, which means that successive changes in prices are independent. Therefore, no predictable cycles should persist in a series of prices. When however such a cycle would be come apparent to investors, they immediately eliminate it by their trading. Later in 1965, Fama defined the concept of *Efficient Capital Market*. He proposed basic assumptions to this hypothesis as:

> *"a market where there are large numbers of rational profit maximizers actively competing, with each trying to predict future market values of individual securities, and where important current information is almost freely available to all participants."*

In 1970, Fama further developed his theory by distinguishing three levels of market efficiency:
1. The "*Weak form efficiency*" asserts that all past market prices and data are fully reflected in securities prices. Therefore, technical analysis does nothing to help investors make profit.
2. The "*Semi-strong form efficiency*" asserts that all publicly available information, for example financial statements, is fully reflected in securities prices. In other words, fundamental analysis is of no use.
3. The "*Strong form efficiency*" asserts that all information is fully reflected in securities prices. In other words, even insider information is of no use.

Base on above forms of market efficiency, if target stock market is weak form efficiency, then any research ground on technical analysis is meaningless. Therefore, we need to first study the market efficiency of target markets.

The efficient market hypothesis was introduced in the late 1960s. Prior to that, the prevailing view was that markets were inefficient. Inefficiency was commonly believed to exist e.g. in the United States and United Kingdom stock markets. However, earlier work by Kendall (1953) suggested that changes in UK stock market prices were random. Later work by Brealey and Dryden, and also by Cunningham found that there were no significant dependences in price changes suggesting that the UK stock market was weak-form efficient.

Further to this evidence that the UK stock market is weak form efficient, other studies of capital markets have pointed toward them being semi strong-form efficient. Studies by Firth (1976, 1979 and 1980) in the United Kingdom have compared the share prices existing after a takeover announcement with the bid offer. Firth found that the share prices were fully and instantaneously adjusted to their correct levels, thus concluding that the UK stock market was semi strong-form efficient.

A conclusion on the absence of weak form efficiency on stock market could greatly suggest opportunities to earn abnormal profits through technical analysis. In fact, this kind of conclusion is yet never certain. However a great proportion of the sample results from this research demonstrate a convincing ability to earn gains to certain degree. Therefore, we can at least be against the saying that technical analysis is of no use. And a better solution in search of an effective technical analysis strategy draws the purpose of this project.

2.1.2 Technical Analysis

Technical analysis is base on *law of supply and demand*. Technical theorists believe that market value is solely determined by interaction between supply and demand. Supply and demand are governed by numerous factors, both rational and irrational, which even include those factors that arc relied upon by the fundamentalists, as well as opinions, moods, guesses and blind necessities. The market weighs all of these factors continually and automatically. Therefore, technical analysis only needs to focus on market itself, without extra consideration of other factors influence supply and demand relationship.

The basic assumption of technical theorists is that history tends to repeat itself. In other words, past patterns of market behaviour will recur in the future and can thus be used for predictive purposes. In statistical terminology, the stock market technician relies upon the dependences of historical information (open price, close price, high price, low price and volume). Based on essential elements of price, volume and time-sensitive, investors have derived various *technical indicators*, trying to predict the future behaviour of the stock price hence maximize their profits and/or minimize risk. However, barriers exist when applying technical indicators to assist analysis.

It is difficult to choose suitable indicator or combination of indicators. Investment strategy varies greatly from short-term to long-term, so does investors' preferences. Short-term speculation values time-sensitive; while long-term investment favours low risk. We need to carefully choose best combination of indicators upon specified requirement.

Technical indicators each have its own advantages and disadvantages. A particular indicator may only work best under certain circumstances. Therefore deep knowledge and accumulative experience are required for investors to make good use of the correct set of indicators.

Technical analysis demands an ultimate rational mind. Users themselves should avoid irrational moods, guesses, etc. In addition, technical analysts are also required to look back and evaluate past performance continuously, forming the basis of persistent correctness and potential future improvement.

The above restrictions on human are exactly the advantages of computerized algorithms, especially when we applying Genetic Programming in this project. The idea of using technical analysis indicators as the variables of the Genetic Programming is because they are easier to understand than pure mathematical abstractions and because the way each indicator behaves is already known to the technical analysis enthusiast. It is the way people use the indicators that differs. This project will try to select reasonable set of popular stock market technical indicators (more details in Design chapter). Through evolution process, we expect the program can work its own way up to complexity and precision, to derive better and better strategy through mutations and crossovers of generations.

2.2 Genetic Programming

Genetic Programming (Koza et al., 1992) is a recent development in the field of *evolutionary programming* (EP) (Fogel et al., 1966) which extends classical genetic algorithms by allowing the processing of non-liner structures. This section attempts to provide some background knowledge and mechanics of Genetic Algorithms and particularly Genetic Programming.

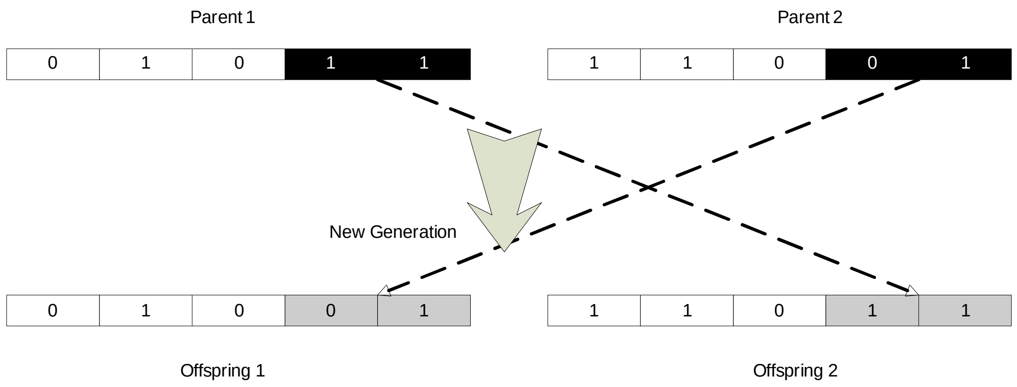
2.2.1 Genetic Algorithms in General

Genetic Algorithms (GA) (Holland, 1975; Goldberg, 1989) are stochastic search techniques working on a population of *individuals* or solutions encoded as linear gene belts, usually in the form of bit strings. Theoretically, GAs can be viewed as a search procedure that trying to find a solution subject to some boundary conditions. GA's way to accomplish this is inspired by Darwin's theory of evolution, which is based on the notion that living beings developed as a result of biological chain reactions caused by *natural selection*.

GAs start with an initial population of individuals or solutions to a problem. Each individual has its unique *chromosome*, which is a collection of genes. In traditional simple GA defined by Holland, those genetic units are mapped into binary strings. Each bit on a string corresponds to a constraint to the problem. A group of such binary strings is the population. This population evolves over time through the application of *genetic operators* which mimic those evolutionary processes found in nature, namely, survival of fittest, crossover and mutation. These biological operators do not work on the individuals as in nature, but on their chromosomal representation.

Parent chromosomes are selected in the population for reproduction. This process is called *Selection*, which is probabilistic and biased towards the best individuals. The selection procedure is vital to the successfulness of the genetic search. It propagates good solution features to the next generation. The most common form of selection method is *roulette wheel selection* (Goldberg 1989). The principle is that individuals are selected from the population with a probability that is in accordance with their relative fitness value as evaluated against the whole population. For example, an individual with a fitness value stands in the middle of the population. Then it has half of the probability to be selected. For GP, we use another popular form of selection method called tournament selection by Koza, as described later.

Selected individuals form a *mating pool*, where paired parents are randomly chosen and *Crossover* process is carried out. The crossover operator combines the characteristics of paired parents' chromosomes to create two new offspring, with the aim of having better fit than their parents. A cross point is randomly chosen along the length of the chromosomes and two chromosomes swap gene information from this point onwards, thus creating two new individuals:



Crossover operation in bit-string Genetic Algorithm

Finally, the *Mutation* operator is applied to a proportion of new generation, this introducing some random variation into the population. For a bit string genetic algorithm, individuals are randomly selected according to a mutation probability. A position on the chromosome is chosen and the value of the bit gene in this position is flipped from 0 to 1, or from 1 to 0. Mutation is considered only as a touch of finish that slightly perturbs small proportion of solutions.

Through the three genetic operators, i.e., selection, crossover and mutation processes, the population of individuals or solutions is expected to be continuously improved, as individuals in new generations will be gradually replaced by better gene belt. This sequential process will repeats itself for generations. Depending on the problem in question, the termination criteria can be either convergence, or a fixed maximum generation defined.

GAs have been demonstrated to be an effective problem solving tool for tackling complex optimization and machine learning problems, with a wide range of application, including imaging, circuit design, gas pipeline control and production scheduling (W.B. Langdon and Qureshi, 1995).

2.2.2 Mechanics of Genetic Programming

Genetic Programming extends the paradigm of bit string genetic algorithm by introducing tree structures (genotypes) of encoded programs. Therefore GP does not need to encode the problem, but generate automated programs to solve problems. This greatly expands the areas of evolutionary programming. Programs can represents anything for different problems, and GP offers a unified approach.

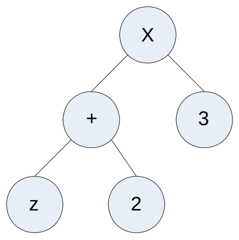
Most of the mechanics of genetic programming can be exactly the same as those in bit string genetic algorithms. This section describes those different ones.

2.2.2.1 Fitness Evaluation

The evolutionary process of genetic algorithms is driven by a fitness function that evaluates the performance of each individual in given problem environment. Therefore the fitness function provides a direction to the evolutionary process. It needs to be mentioned that, in genetic programming, the fitness value does not evaluate solution but the *ability* of finding solution for an individual in question. In this project, profit generated by individual's investment strategy (program) is evaluated on historical financial data. More details in design chapter later.

2.2.2.2 Tree Structure Encoding

Tree structure in genetic programming consists of terminals nodes and function nodes defined for a problem. Trees are recursively constructed from this set of nodes and dynamically change during the evolution process. Each particular function in the function set takes a specified number/types of arguments and should be able to accept returned value by other functions, or terminals which do not have argument. (Koza, 1995) For example, an arithmetic expression $(z+2)\times 3$ can be represented in tree structure as:



An arithmetic expression tree example

The search space in genetic programming is the space of all possible computer programs composed of terminals and functions appropriate to the problem domain. In general, *terminal set* consists of variables and/or constants (z, 2, 3 in the above example). While *function set* includes arithmetic operators ($+,-,\times,\div$, etc.), mathematical functions (Sin, Cos, Log, etc.), logical functions (AND, OR, NOT, etc.), and programming functions (IF-THEN-ELSE, etc.). Genetic programming possesses the elasticity that bit string genetic algorithms lack, therefore it can search even wide and complete problem space.

Function Type	**Function Examples**	**Data Types**	
		Inputs	**Output**
Logical	AND, OR, NOT, XOR	Boolean(s)	Boolean
Comparison	$>, <, \neq, \leqslant, \geqslant$	Real numbers	Boolean
Mathematics	$+,-,\times,\div$, POW, LOG, MAX	Real numbers	Real number
Programming	IF-THEN-ELSE, DO-WHILE	Boolean, *(void)*	*(void)*

Function Set examples defined in this project for Genetic Programming

2.2.2.3 Initial Population and Tree Creation

Similar to other genetic algorithms, initial population in genetic programming is made of randomly generated chromosomes, but in the form of programs trees. A good tree creation algorithm is important to GP. It is because that, initial population creation plus new branches of tree in crossover and mutation procedure could result at least 100,000 tree creations for a 100 population size in several generations. By far, the most common mechanism for creating trees and sub-trees in GP is the GROW and FULL methods. They are used in almost all literatures on GP.

GROW begins with a set of functions to place as nodes in the tree. It *recursively* call itself, by selecting a root from the set of functions, then fills the root's arguments with random functions, then their arguments with other random functions or terminals, and so on (Luke, 2000). A pseudo-code can be written as:

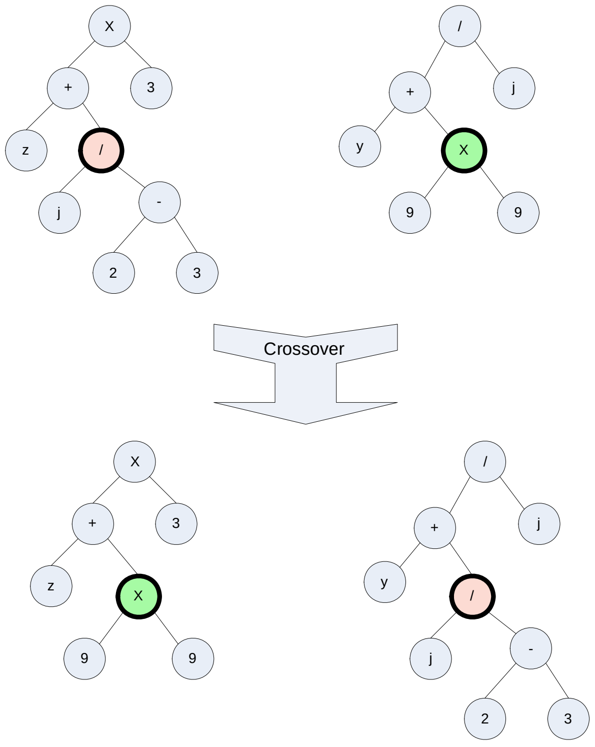
```
Given:
    Maximum depth D
    Function set F consisting of non-terminal set N and terminal set T
Do:
    New tree T = GROW(0)
```

```
GROW(depth d)
    Returns: a tree of depth <= D-d
    IF d=D, return a random terminal from T
    ELSE
        Choose a random function f from F
        IF f is a terminal, return f
        ELSE
            FOR each argument a of f
                Fill a with GROW(d+1)
            Return f with filled arguments
```

GROW method is also extensively used in generate new sub-trees at a chosen point. FULL is merely at opposite direction of GROW procedure (not from root but from terminal nodes backwards). However, GROW tree-creation algorithm has serious weakness. It offers no control over newly generated tree, in particular the tree size. Also, for this project, GROW method can not offer fine-grained control over expected probability for a particular function/technical indicator. A better algorithm for tree creation (PTC) adopted in this project will be described in design chapter.

2.2.2.4 Genetic Programming Crossover Operator

Crossover in genetic programming is far more complicated than in bit-string genetic algorithms. The complexity comes from tree manipulations. When a crossover procedure applies on two chosen parents, a random crossover point is chosen from each parent. The crossover point defines a entire sub-tree lying below the crossover point. Two new generation programs are then produced by exchanging the two sub-trees. This better is illustrated to make clear as below:

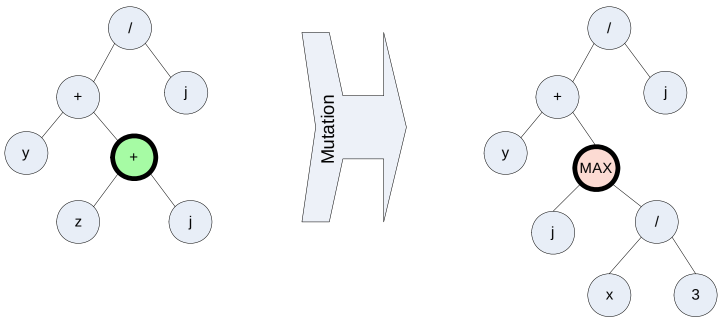


Crossover in Genetic Programming

When both crossover points are terminals, only ‘leaf’ of the tree is swapped. Therefore, the probability distribution for the selection of a crossover point favours the exchange of subtrees rather than terminals. Also in this project, different financial indicators would return different type of values (floats, percentages), and different functions would have different types on return value and arguments, for example, “+” operator returns a real number and requires two arguments of numbers, while “>=” returns a Boolean value and requires two real numbers. Therefore crossover point can not be purely randomly chosen for stock market prediction problem (and most other problems as well). The crossover procedure in genetic programming need to consider a lot to guarantee compliable and executable programs generated. This will be implemented by runtime reflection, more details in Design and Implementation chapter.

2.2.2.5 Genetic Programming Mutation Operator

Mutation is still a unary operator that alters part of a chromosome, aimed at restoring diversity in a population by applying random modifications to individual structures:



Mutation in Genetic Programming

However, in genetic programming it has reduced functionality compared to its importance in bit-string genetic algorithms. Because trees manipulated in GP are far more complicated than the linear structures of GAs. Even crossover operation applied on two identical chromosomes would give completely diversified new trees. In essence, this is due to the fact that genetic programming is evolving programs not solutions. Hence it does not need mutation to maintain diversity as important as in linear genetic algorithms for coefficients. Therefore, mutation rate is kept low by default, but still exists in my implementation. It is simply done by drop a random sub tree then grow a new sub tree from that point.

To sum up, typical evolutionary steps for genetic programming are:

```
1. Randomly create the initial population
2. Repeat:
    (a) Evaluate the fitness of each individual by execute its program.
    (b) Select parents from population using selection strategies
    (c) Generate new population using genetic operators:
         Reproduction, Crossover, Mutation
3. Until termination criteria, usually a fixed number of generations.
```

A simple genetic programming flow

Genetic programming has been applied to a broad range of problems with some success from traditional optimisation in engineering and operational research (Bäck 1997) to non-traditional areas such as data mining, composition of music and financial prediction (Kinnear 1994, Koza 1996)

2.3 Distributed Genetic Programming Parallelisms

This section provides theoretical research on parallel genetic programming algorithms. The reason and motivation for parallelism is firstly assessed. Followed by a survey of current literatures on schemes/topologies defines a parallel genetic algorithm's structure. This section looks at fundamental building blocks, more details on parallel pattern of this project can be found in design chapter.

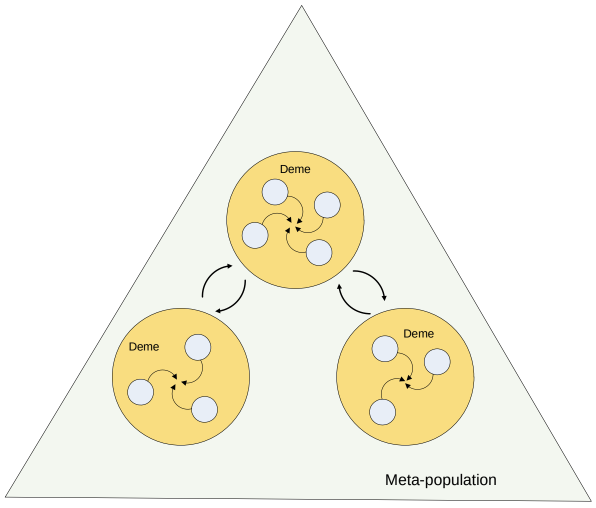
2.3.1 Motivation

The probability of success in applying genetic algorithm to a particular problem often depends on the adequacy of the size of the population in relation to the difficulty of the problem. (Harik and Miller, 1997). Since the computational burden of GAs is in accordance with size of population, more computing power is required to solve more substantial problems. Increases in computing power can be realized by either increasing the speed of a single computer or by parallelising the application. This is because those fast serial super computers are expensive and rare. Therefore, parallel implementations could be a more favourable solution.

Firstly, a parallel genetic algorithm does not lose any advantage of a linear/serial genetic algorithm. If we see the distributed network combined as a whole, or more specifically, a multiple-instruction-multiple-data (MIMD) (Flynn, 1972) computer as described later in feasibility assessment, there is no difference to the process from start to the end. However, a parallel distributed GA can search from multiple points in the search space simultaneously. This offers a *wider* search space than a serial GA. Further, since physically each paralleled GA subgroup operates on separate processors, the search speed and computation power adds up together. This offers a *deeper* precision on a given generation. Altogether, it is easy to see that parallel GAs have higher efficiency and efficacy than serial GAs.

2.3.2 Feasibility

In a previous section, we see genetic algorithms are basicly trying to mimic natural evolution. If we are adopting this process as much as in reality, then the whole process should not operate only on a single population. In nature, species tend to reproduce within subgroups or within a given circle of neighbours. An individual should have the potential to mate with partner in the entire population (*panmixia* or *meta-population*). This is where my MetaDGPEngine's 'Meta' comes from: *meta*-population. An important component of speciation theory in Biology is the *deme* concept (Mary, 1963) because this is the unit on which either selection or drift operates. Demes are random mating sub populations. A classic diagram found in almost any biology textbook on population structure looks like below:



Population structure defined in Biology

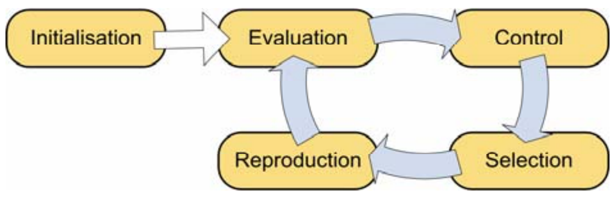
Now a distributed genetic algorithm is theoretically backed by its fundamental theory adopted from biology. But is a computerized implementation possible?

Based on Flynn's classification of computer architecture (Flynn, 1972), a *multiple-instruction-multiple-data* (MIMD) are processors may perform different instructions on different data. MIMD is the most common type of parallel computers. A MIMD multi-computer is where communication between processing elements is via message passing, as opposed to shared memory in a multiprocessor machine. The approach of multiple computers connected in a P2P model for mass computation conforms to this definition, so are all of the schemes on genetic algorithm parallelisms described in following sections.

In terms of cost, for a similar level of computation power, multiple cheap desktop processors operating as grid computing is far less expensive than a single super computer. One might argue about the communication speed, since distributed programs use network connections to transfer data, whilst super computers use high speed dedicated BUS. However, for genetic algorithm problems, once well designed, this transaction cost is very much low, because most of the time genetic algorithms are doing evaluations which involves no communication in between individuals (collusion among artificial traders is an exception not belongs to the topic of genetic algorithm).

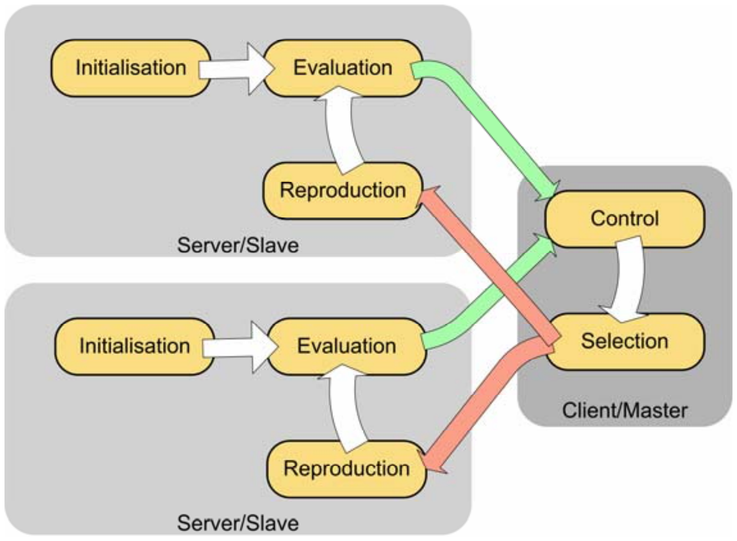
2.3.3 Task Allocation Modelling

There are many ways to *divide tasks* of the genetic algorithm process and prepare for parallelising. Firstly, let us look at an abstract model of a traditional serial GA process:



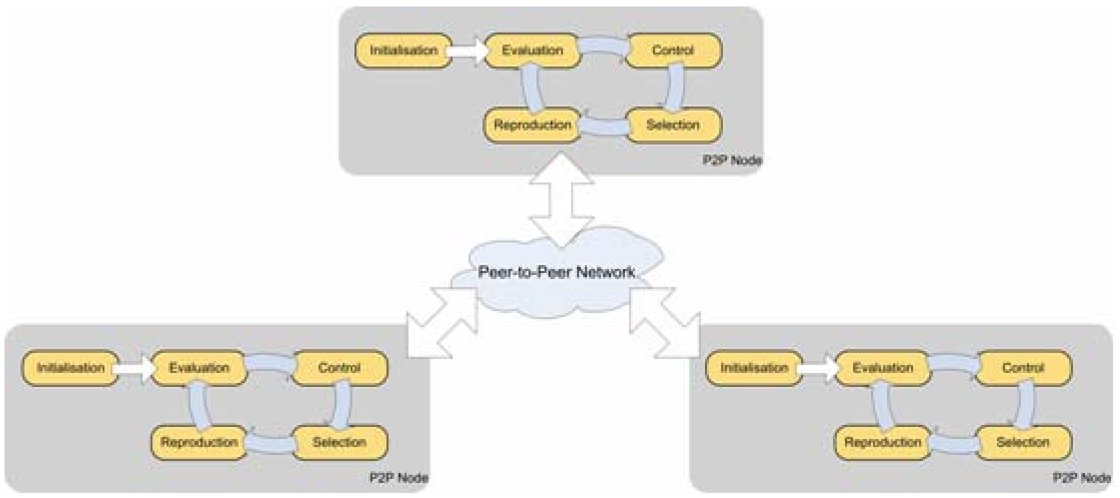
Procedures of a local machine genetic algorithm

Statistically, most time consuming calculations lies in the evaluation part, and for genetic programming, the genetic operators on tree structures. Therefore, a *client/server* sub-task model is illustrated as follow:



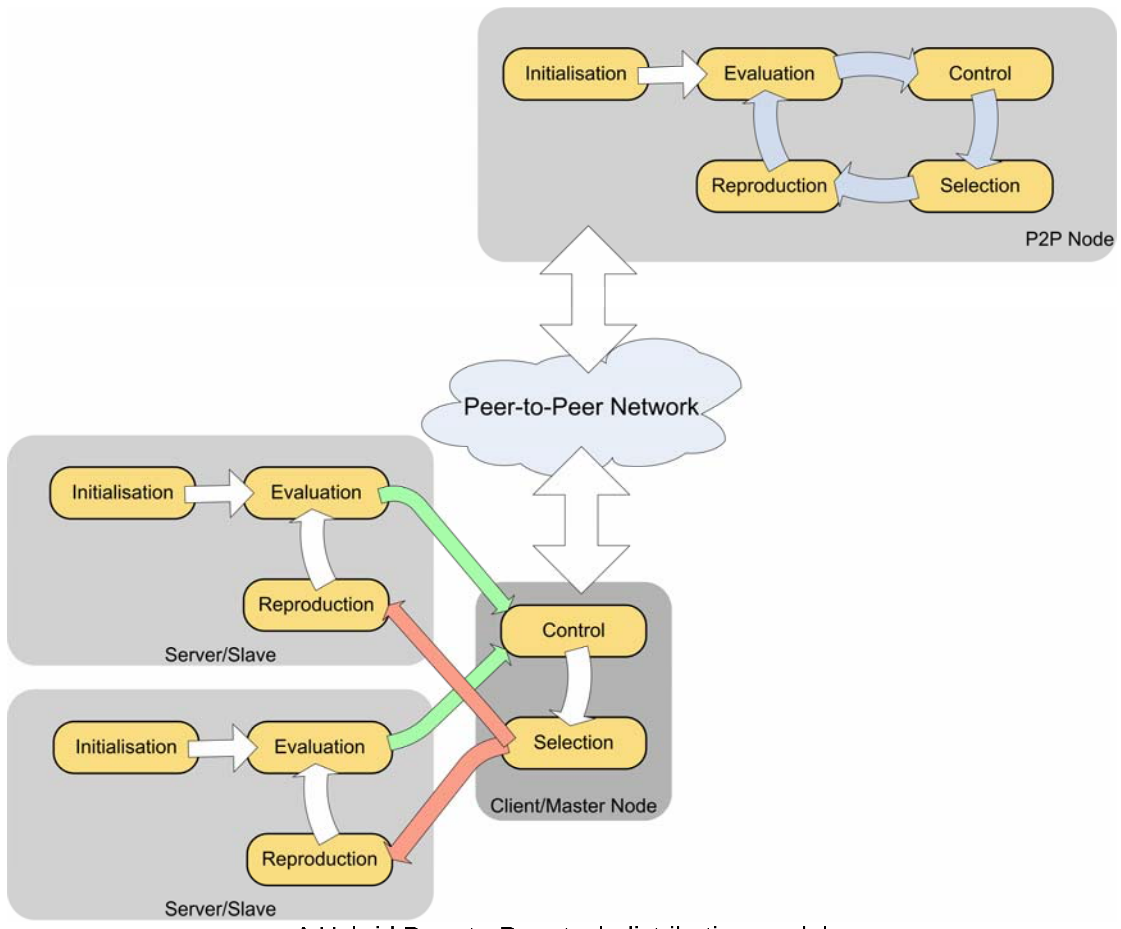
A Client/Server task division model

The client/server model expands server's ability by shifting majority parts of process to multiple clients. Further to this model, recent years see the great emergence of *peer-to-peer* network applications. In term of genetic algorithm, it also means all process is computerised on separate processors/nodes; good individuals are exchanged in between. This is more applicable for this project, since financial prediction requires extensive calculations on almost all four sub-procedures. The above client/server subdivision model is not very much effective on stock market predictions distributed genetic programming. In a peer-to-peer model, Multiple Genetic Engines may be interlinked using a peer-to-peer network to form large virtual populations. (Individuals are exchanged during Selection phase):



A Pure Peer-To-Peer task allocation model

Finally, a *hybrid* distribution combines all above models, creates large virtual populations which can be evaluated by complicated/time consuming simulations:



A Hybrid Peer-to-Peer task distribution model

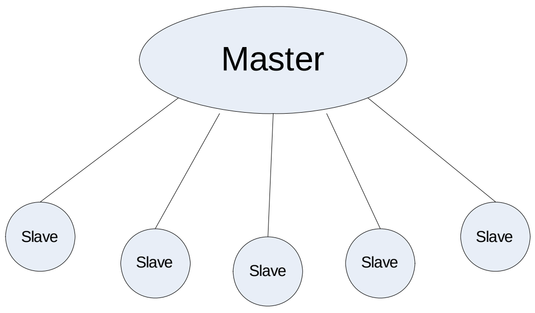
In this project, a *pure P2P task allocation* model is chosen for both efficiency and simplicity. However, assessing task allocation models is only part of the whole genetic programming parallelism in this project's background research. We still need to define a topology for distributed network targeted at genetic algorithms. There are already numberous literatures on this topic. And Dr. Enrique Alba and Dr. Jose M. Troya did a nice survey of parallel distributed genetic algorithms direct to this topic (Alba and Troya, 1999). Therefore the next section we simply make a summery of the advantages and disadvantages for each.

2.3.4 Topologies for Distributed Network

Many different schemes have been proposed for the parallel genetic algorithms (PGA). However, all existing models can fit into two more general classes depending on their grain of parallelism, called coarse or fine-grain PGAs (Alba and Troya, 1999). This classification relies on the computation/communication ratio. If this ratio is high we call the PGA a coarse-grain algorithm and, if low, we call it a fine-grain PGA. There are five different models of parallel genetic algorithms: *global parallelisation*, *coarse grain*, *fine grain*, *coarse and fine grain*, *coarse grain*, *global parallelisation* and *coarse grain plus coarse grain*.

2.3.4.1 Global parallelisation

Global parallel genetic algorithms are mostly designed for Client/Server task allocation model described above. Under this scheme, there is still a single population as a whole. But the time –consuming evaluation and reproduction procedure is allocated to slave workers:

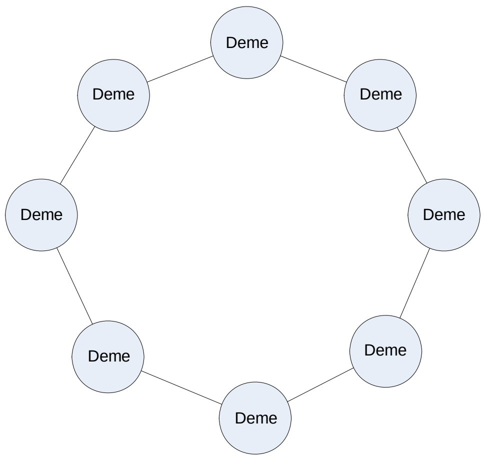


A schematic representation of a global parallel GA

Therefore, global parallelism is very suitable for evaluation extensive computations. It is easy to implement since almost all procedure is exactly the same as on local machine in serial GAs. Only evaluation and/or reproduction procedure is duplicated in slave worker processes. However, global parallelism is not ideal for a large quantity multiple processor computation because there is obviously too much job taken by Master process itself. As number of slaves adds up, master would be too busy and thereafter drag the performance of the whole distributed system.

2.3.4.2 Coarse grain parallelisation

Coarse grained parallel scheme applied to genetic algorithms is a more sophisticate idea. The whole population is divided into demes as described above. Those demes are isolated from each other, but exchange individuals occasionally with their neighbours. Coarse grain topology in distributed system architecture provides a fine model for parallel genetic algorithms since it mimic the nature's population structure the most.



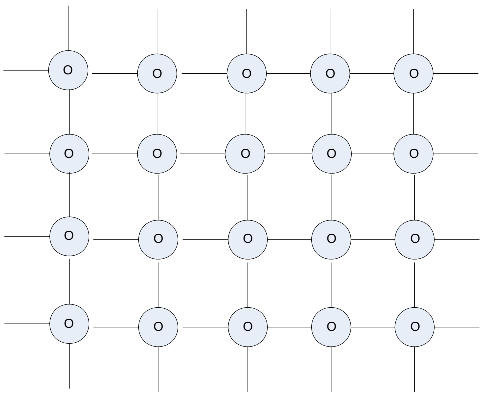
A schematic representation of a coarse grained parallel GA model

Coarse grained GAs are the most popular parallel method and many papers have been written describing innumerable aspects and details of their implementation. These are summarised in a report by Chong Sian (Sian, 1999). The first systematic study of coarse-grained parallel GA was published by Grosso in 1985. His objective was to simulate the interaction of several parallel subcomponents of a population in evolution (Grosso, 1985). His important finding was that a collection of smaller demes improves the population fitness faster than a single large panmictic population. This is also one of the observations achieved in my project.

However, the inter-communications among demes have been an ignored field among researches. This element of topology is vital, because it determines how fast a better individual spread and breeding children to other demes. If this process goes too fast, i.e., the topology has a dense connectivity, the very best individual will quickly take over large proportion of all demes. This will restrain the search space for the whole distributed network. On the other hand, if this process is too slow, i.e., the topology is sparsely connected. Each deme may have been wasting too much time computerising those individuals that are already dropped behind in terms of fitness value. In this project, a controlled coarse grain topology is designed; more reasons and details can be found at design chapter.

2.3.4.3 Fine grain parallelisation

Fine grain GAs are also known as cellular GAs. In this model, the whole population is divided into many small sub-populations. The extreme case could be one individual per sub-population. This design pattern initially was for massively parallel single-instruction-multiple-data-machine (SIMD). SIMD architecture contains a control processor that drives several task processors and their associated memories in a lock step manner. (Stone, 1975) Illustration of this model usually place individuals in a 2-Dimentional grid as below because in many SIMD implementations, the processing elements are connected using this topology:

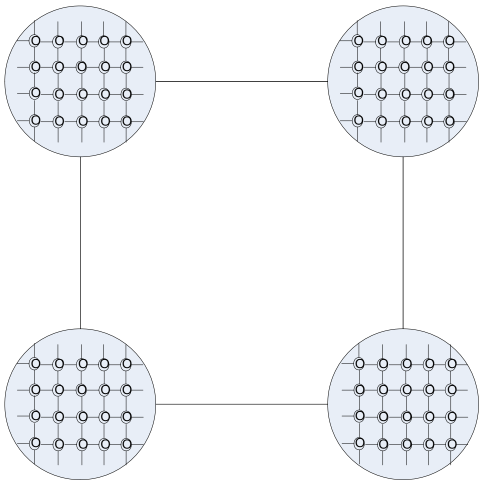

A schematic representation of a fine grained parallelism

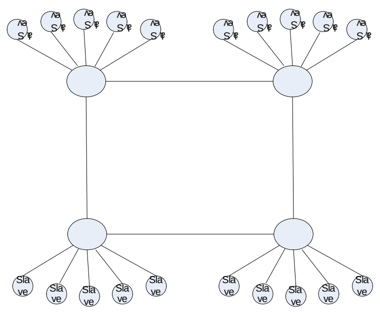
This model is more targeted at specialised computer architecture: SIMD. When a parallel computer has enough processors, performance of GAs can be dramatically improved under such scheme. Obviously we are talking about super-computers here, therefore, with the help of high bandwidth, the inter-connections between individuals or small subpopulations are the fastest.

Hence its disadvantages are apparent as well. The very first problem is that this special designed super-computer comes with a super price tag. And it is not easily available as well. Secondly, such a large inter-communication between individuals and large number of connected processors require design with extreme care. Actually it is extreme difficult. And generally speaking, the time spent is much less beneficial than spending on another topology.

2.3.4.4 Other combinations of parallelisms

There are other parallelism schemes that simply combine the models described above. They are invented for particular purposes, either for hardware or specific problem (Sian, 1999). We will not discuss these models in depth in this project.


A schematic representation for coarse and fine grain model

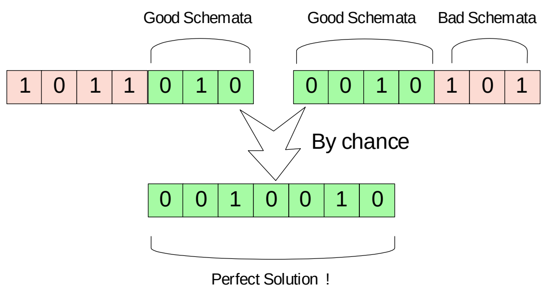


A schematic representation for coarse and global parallelisation model

2.4 The Schema Theorem

This is the final section of background research for this project. Yet we have examined the mechanism behind genetic programming; we also have studied varies models to support a parallel distributed genetic programming system. Now we need to prove in theory that a distributed genetic programming *does* work. In a survey of parallel distributed genetic algorithms conducted by Alba and Troya, they adopted a *Schema Theorem* well supports all the rationale we have described above (Alba and Troya, 1999). Hence provides the rest theoretical foundation on the significance of this project.

Alba and Troya introduced a terminology called *schemata*, which acts like sub elements that logically break down a chromosome of an individual. The search space of any genetic algorithm is nothing but large quantity of good and bad schemata. And an ideal solution is simply a combination of good schemata on all parts of chromosome. Below is my illustration of understanding on this idea. For simplicity I only drew bit-string chromosomes, but this also applies to tree structures in genetic programming.

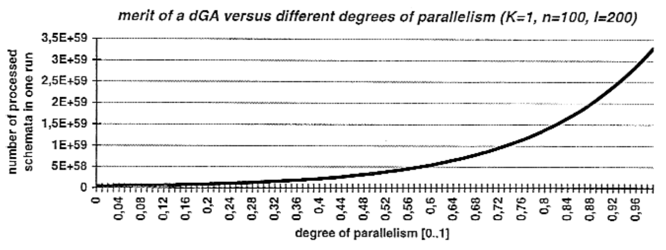


Good and bad schemata

In an initial study by Holland, he worked out a result known as the *implicit parallelism* of evolutionary algorithms. He suggests that although the genetic algorithms are working on chromosomes, the whole mechanism is actually working in a 3-Dimensional space of schemata. (Holland, 1975). Alba and Troya extend this theory. By trying to modelling the effects of genetic operators, they conclude that crossover and mutation process are actually filtering good schemata. Therefore, any problem that can be expressed as building blocks can be solved by genetic algorithms. For example, bit represents co-efficiency on bit-string; tree node represents a function or variable on a genotype as in genetic programming.

Furthermore, Alba and Troya extend their Schemata Theorem trying to *prove distributed genetic algorithms*. This was developed upon an existing measurement of merit of parallel genetic algorithms developed by Goldberg (Goldberg, 1989). Under the assumption that each individual perfectly acquires a separate and identical processor, the merit of a parallel genetic algorithm depends on the population size, string length, and degree of parallelism. Below I used their graph to illustrate this relationship (Alba and Troya, 1999):

FIGURE 11



Relationship between the merit and the degree of parallelism. Computer simulation with 100 strings of 200 bit length.

By applying the relationship between degree of parallelism and number of strings in given population, they finally conclude that the assumption is not necessary. That is to say: **a high performance distributed genetic algorithm does *not* depend on an ideally perfect parallel hardware.** (Alba and Troya, 1999)

Having every element of the rationale behind the whole project theoretically proved or supported, we now can lead to see the design issues.

Design | 3

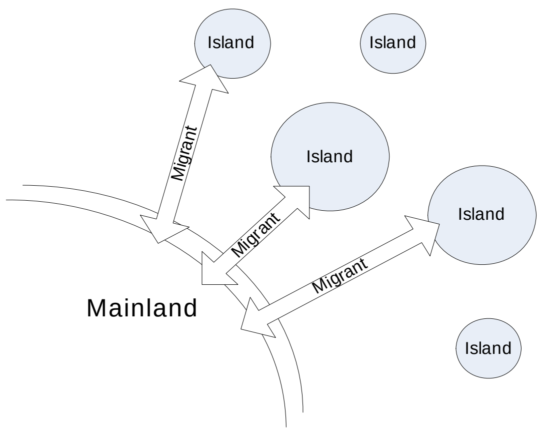
The aim of this project is to implement a visualised distributed genetic programming system applied on stock market prediction. Therefore, from a direct literal interpretation of this title, we need to design a parallel distributed genetic programming system; then a mod on financial genetic programming; finally it needs to be visualised. There are numerous literatures working good or fine on each of the sub element, however, until I am writing up this report, there is no such a comprehensive visualised system that creates and combines all these parts. In this chapter I shall talk about various important design issues involved in building up this system.

3.1 Island Model Distributed Genetic Programming

In a previous section, we have assessed many design patterns on parallel distributed genetic programming system architecture. And by comparing advantages and disadvantages, we have chosen a coarse grain parallel genetic programming model as the best candidate for our project, since it mimic the nature's population structure at the best, also very much suitable for our chosen P2P network task allocation model. However, a theoretical pattern is far from a working system's implementation. There are still lots to think about. Especially, the inter-communication between each demes within our meta-population.

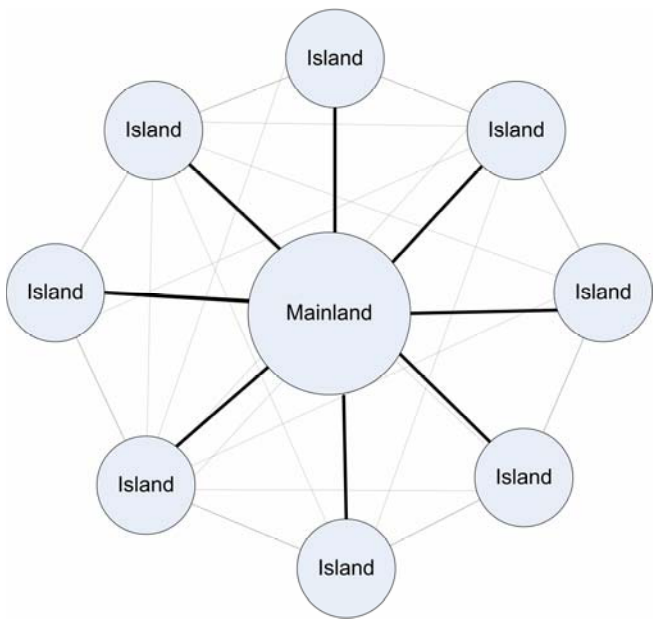
3.1.1 Deriving the Controlled Coarse Grain Parallel Genetic Programming Model

As an overseas student, I travel in between continents a lot each year. This summer I took this project's initial specification with full of question marks back home. When standing in the long queue at the very familiar passport control counter, it suddenly inspired me a working model for this distributed system. Those individuals travel in between demes (or countries, as I called it initially) can be viewed as migrants. Later after I rushed into library referencing how population structure defined in biology, I found exactly similar set of terminologies as expected. There is a mainland-island model of biological eco-system defined. And individuals migrate between islands and mainland. I illustrate it as below:



Mainland-Island model in biological eco-system

Therefore, instead of a collection of discrete peer nodes, which is ideal theoretically but lack of control in application. I modified the coarse grain parallel model into a somewhat controlled model:



A revised controlled coarse grain parallel genetic programming model

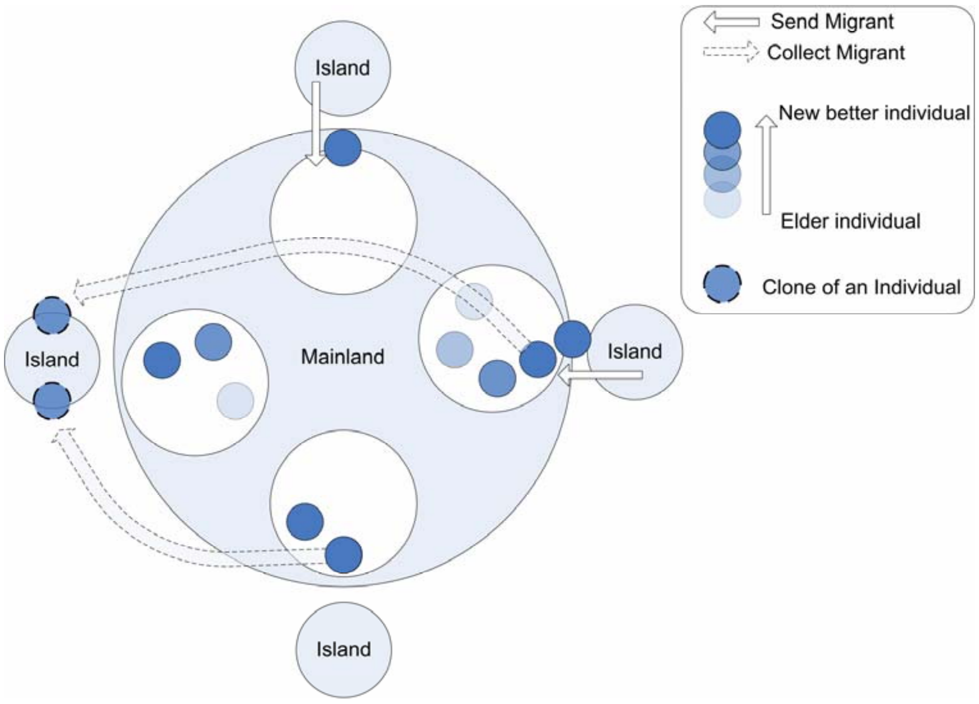
As in the graph above, individuals travel in between islands and mainland, only through thick lines. Those transparent lines only illustrate the abstract process of good migrants travel among islands, which is actually controlled and selected by mainland. This is how only good migrants are delivered (broadcasted) into other islands. Therefore mainland can control the whole population. In my system, the mainland does not take on genetic procedures; it merely does the control part. Firstly, this avoids the traffic problem in global parallelism. Controlling server can use this released computing power to achieve other task, for example those flashing animated visualisation without any delay in this project. Second, the existence of a centralised management makes the whole system easy to use and more reliable. Any failure island would not affect the whole distributed system. Each island can connect and join into this distributed network anytime but start with breeding new generation from the latest migrants. Moreover, we can use linear method to test the performance of the distributed system, for example, statistics on the performance of each processing node is collected in this project.

It is when I am writing up this report, that I found out the notion of “island model” has already been adopted on parallel genetic algorithms recent years in different ways. This further proves the feasibility of such a distributed system structure. Tanimura adopted this model to develop his Dual Individual Distributed Genetic Algorithm (Tanimura, 2000). Whitley, Rana and Heckendorn studied how to use linearly functions to test an island model distributed system (Whitley et al., 1997). Vertannen discussed the island model under parallel virtual machine (PVM).

3.1.2 Migrants Pool

A finished model for island and mainland is still not enough for implementation. How does mainland manage migrants? How is migrants selected to broadcast to other islands? In this project, a migrants pool is designed to meet this requirement. Mainland maintenance a migrants pool which is in essence a collection of queues for connected islands. Given that island only sends out new better individual in each generation, this queue is designed to be LIFO for all other islands, so that the best is always floating to the top.

For every beginning of a new generation in each island, the best individual from last generation is sent as *emigrants* to mainland. Then it tries to collect all new individuals as *immigrants* from others’ migrant pool stored at mainland. These new immigrants will be combined into island’s mating pool to produce the next generation. Below is an illustrated graph for this process, this is also the prototype for *visualisation* in the actual application. The animations implemented in project application make this process clearer.



Migrants Pool Modelling

Once an individual become a migrant, a *GUID* is assigned, which is unique in all meta-population, and even for any runs of genetic programming meta-populations. This is designed to make sure individuals can be recognised, also for storage in a Gene Bank database which will be described later. A *migration history* is attached to each individual in migrant pool to avoid duplicated migration, i.e., clones any individual only need to migrate to mainland or any island *once*.

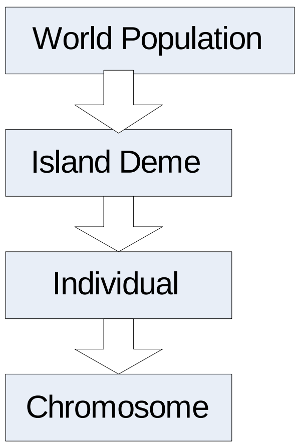
3.1.3 Genetic Programming

The procedure of genetic programming under our model is conducted in each island. In fact each island is a P2P node in a previously assessed model for task allocation. *Island* responsible for all processes involved: initialising population, evaluation, reproduction, crossover, and mutation. The parameters are collected from mainland server when island trying to connect. Basic parameters for genetic programming algorithm and default values are listed below:

Parameter Name	Default Value
Population Size	200
Terminating Generation	unlimited
Reproduction Rate	35%
Crossover Rate	60%
Mutation Rate	5%
Max Tree Depth	17
Max New Sub Tree Depth	6

Default Genetic Programming Parameters

Derived from our island model, the hierarchy of our genetic programming structure is defined as:



World hierarchy

Other algorithms and methods chosen/invented for genetic programming in this project will be described in Implementation chapter.

3.2 Financial Genetic Programming

One of the goals of this project is to achieve a precise and efficient stock market prediction, through distributed genetic programming. There are many research papers working on bit-string genetic algorithms, but less on genetic programming. In background chapter, we already proved in theory that genetic programming is more suitable for financial prediction. But the mechanism of design and implementation remains a blurred area.

3.2.1 Price VS. Return

Inherited from bit-string genetic algorithm, it is natural to build financial genetic algorithms focused on price prediction. Because the classic way to value stock price is to view the stock price as the total of a *Discounted Cash Flows*. When an investor buys a stock, he is entitled to receive all future dividends and can sell the stock in future. Therefore, the cash flows of common stock consist of dividends and a future selling price. Therefore, the price P_0 of a stock to be sold in one year at P_t after received dividend D_0 is:

$$P_0 = \frac{D_0 + P_t}{1 + r_e}$$

Using recursive substitution, the current price of the stock is:

$$\begin{aligned} P_0 &= \frac{D_1}{1+r_e} + \frac{D_2}{(1+r_e)^2} + \frac{D_3}{(1+r_e)^3} + \cdots + \frac{D_t}{(1+r_e)^t} + \cdots \\ &= \sum_{t=1}^{\infty} \frac{D_t}{(1+r_e)^t} \end{aligned}$$

Bit-string genetic algorithms see stock pays a stochastic dividend per period which is generated by a mean-reverting autoregressive *Ohrnstern-Uhlenbeck* process: (Ehretreich, 2002)

$$d_{t+1} = \bar{d} + \rho(d_t - \bar{d}) + \varepsilon_{t+1}$$

$$\text{where } \varepsilon_t \sim N(0, \sigma_\varepsilon^2)$$

Therefore the rest of the genetic algorithm's job is merely to generate this model to obtain a formula that has results best fit the trend of historical prices.

However, this is not what a trader is thinking when making investment decisions. Although from the surface traders are most interested in the price level, fundamentally they are evaluating *returns*. This is different from return in pricing model, which is:

$$R_e = \frac{P_t - P_0 + \sum D}{P_0}$$

In this project, the return is *investment gross profit*. Because trader can profit from both positive and negative price moves. If the price is expected to go up, then a trader would take *long* position, i.e. buying stock and selling later; if the price is expected to go down, then a trader sells to take a *short* position then later buys. A higher percentage return is possible if one takes long and short position on margin. And in margin trading, traders only need to pay half of the commission.

Therefore, for genetic programming, we are generating investment *strategies* based on returns rather than price valuation formula. This also has the advantages that transaction costs can be taken into consideration, which has to be ignored in bit-string genetic algorithms. A typical parameter setting for short-term financial genetic programming defined in this project looks like:

Parameter Name	Default Value	Explanation
Investment Cycle	14 days	Short-term (2 weeks) investment cycle
Required Rate of Return	4%	By default, the lowest return should equal to interest rate
Transaction Cost	5%	This sets a constraint that requires strategy generated to be rational on trading frequency

Default Financial Genetic Programming Parameters Defined in this project

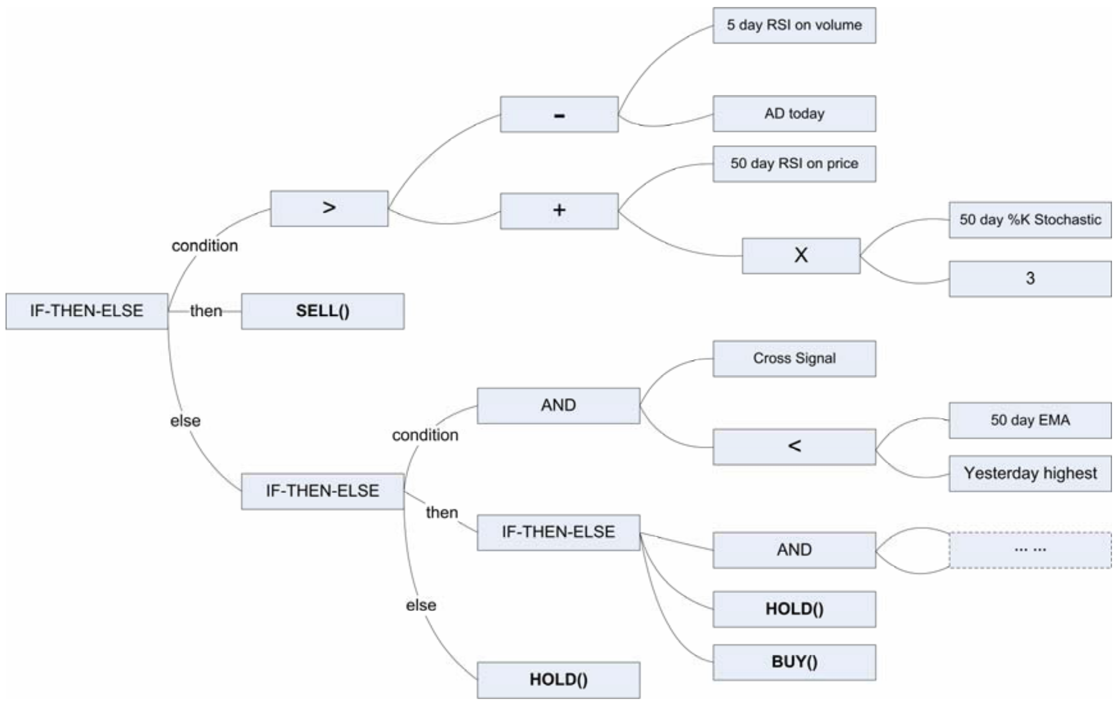
In summary, we expect our strategy programs to generate *signals* rather than price predictions, which is closer to traders' decision considerations. Signals are same to that credit rating commonly on all financial reports. In this project, **BUY**, **HOLD** and **SELL** are signals adopted.

3.2.2 Strategy Programs

We expect genetic programming to generate investment strategies for financial market prediction. How are those strategy programs defined? Just like technical traders, the strategy programs should use technical indicators as input, and by trying to find the relationships, eventually derive a program that could generate signals. An example from the result of this project, e.g., a profit making strategy programs on two years historical data for Microsoft (MSFT) obtained after 87 generations is:

```
IF ((RSIV5 - AD) > (3 * KSTO50 + RSIP50))
THEN SELL()
ELSE IF ((EMA50 < HIGH[n-1]) AND CSP())
     THEN IF (AD > (2 * KSTO10 + (RSIV5 - AD)))
          THEN HOLD()
          ELSE BUY()
     ELSE HOLD()
```

Internally this program is in tree structure representation, this is illustrated as below:



Tree structure for a sample strategy program

3.2.3 Selected Technical Indicators

The first question is why we need technical indicators, since all technical indicators are nothing more than some mathematics tools applied on raw data. The raw data for any financial market assets are merely *Open*, *Close*, *High*, *Low*, *Volume*. Our genetic programming already contains arithmetic and mathematics operators which could also obtain similar result as any indicator. Besides the fact that calculated indicators could greatly speed up our genetic algorithm process, indicators as investment measures commonly adopted by traders can also help our programs to be *human-readable*.

There is no literature yet on selecting appropriate technical indicators for machine learning. Either some are trying to include all, or some merely use moving averages on price. I did extra research into this area. There are large quantity of indicators available, some are popular, some are characteristic. In general, these indicators fall into categories, from both *mathematical* and *data type* perspectives. Therefore a sound way is to cover these area one by each: We do not need to include every indicator, but we need to keep the balance and efficiency of indicator set. In addition, most research have been focusing on price indicators, but ignore *volume*, in reality volume takes a very much important role on measuring psychological effects in stock market. Hence it also forms a big category in this project. Below is the table of indicators adopted, listed in a form crossed by characteristics and data types:

	On Price	On Price and Volume	On Volume
Moving Averages	Median Moving Average		
Stochastic	%K Stochastic		
Momentums			Momentum Percentage
Oscillators	Relative Strength Index (RSI) on price		Relative Strength Index (RSI) on volume
Accumulative Distribution (AD)		Chaikin Money Flow	

Indicators Selected by Categories

Explanations on each indicator are beyond the scope of this report. More details can be found in Appendix.

It needs to be mentioned that most indicators have length of period as a parameter. For example, there are 5 days, 10 days, and 50 days MovingAverage. This is represented in strategy programs as EMA5, EMA10, EMA50. Also, most of time the program need to compare with historical data or indicators, this is represented in strategy program as HIGH[n-2], which stands for highest day price two days ago.

Based on the flexible genetic programming engine implemented in this project, it is also interesting to mention that there is one extra indicator that returns a *boolean* value, as seen in the strategy program example in last page: CSP(). This indicator returns true when a long period indicator is crossing a shorter period indicator. This was a favourite short-term indicator mentioned by my mother, a long-term market speculator. When this is added to function set of this project's financial genetic programming system, it is interesting to observe that for a number of generations at the beginning, all islands stuck at this "best" indicator, since no other combination can beat the profit generated by this simple strategy:

```
IF CSP()
THEN BUY()
ELSE SELL()
```

The above tree output is always the first migrated individual for almost any setting. This demonstrates this project's ability of searching solution. However, this strategy is then quickly replaced by better strategy programs consist of complex grammar. This proves the working evolutionary algorithm of this project.

3.2.4 Hypothesis Investment Strategy for Evaluation

Having decided the structure of our strategy programs, we need a scheme to evaluate the performance. And we need it to return a fair *fitness value* for genetic programming procedure. Initially, works has been done trying to develop a percentage precision on prediction. But how do we measure the accuracy of a strategy program? Again I realised that this is for price prediction. We need to have different tool for return/profit measurement. It is also required to be comparable for different assets/stocks.

A *hypothesis investment strategy* based on investment cycle is defined for evaluation of financial genetic programming. Initially every individual has a zero holding (fitness value equals 0). Whenever there is a BUY or SELL signal placed at a particular day of historical data, we *buy* or *short sell* one unit of currency (e.g. 1 sterling) at that stocks current price. Within the period of *investment cycle*, for example 14 days from the trade is made, if return rate (profit margin) at any day beyond the *required return* rate (4% by default), then we *sell* or *buy back* this asset. If at the end of investment cycle from the day trade was made, the return rate on any day below the required return, we force to sell or buy back this asset. This is implemented by attaching an account book for each individual during evaluation procedure.

This hypothesis investment strategy for evaluation test implements following features:

1. It is based on profit returns.
2. By standardising one currency unit per trade, it provides a comparable fitness value for different individuals in one population and populations for different settings of test.
3. It put investment cycle and required rate of return into practice, and give them meanings in financial genetic programming.
4. Fitness value is not capped as it should not be in financial prediction: there does not exist any set of 100% accurate trading activities in any context.

3.3 Visualisation

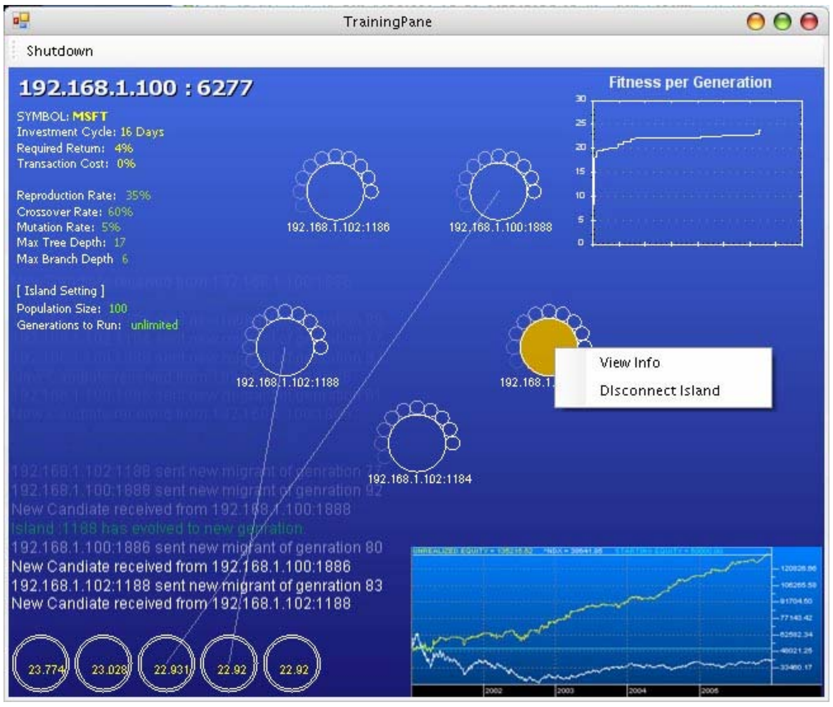
Visualisation adds value to distribute genetic programming. Firstly, visualisation assists performance study of genetic programming. Adjustments doesn't need to be made after waiting for a long time to gather data and put them into statistical package finally get the same set of graphs. A real time rendering engine provides an efficient and effective tool.

Secondly, a distributed network need a system to control and manage, there is nothing else better than a visualised platform. Especially for financial users, they do not want to or do not have enough time to understand academic models behind distributed genetic programming. A visualised animation explains everything straightforward.

The visualisation of this project needs to implement following features:
1. Each elements need to be represented: Island, Individual migrants, Current best individuals.
2. A rolling message pane is needed to inform user what is going on in real time.
3. Statistical graphics need to be presented and updated in real time.

For a common design in graphcis area, we need a *script system* and a fast *graphic rendering engine* to accomplish these, which are exactly the same as technologies used in game development. My MetaGraphicEngine implemented all above and beyond, it can be easier expanded. Also interfaces are defined internally in between MetaDGPEngine and this graphical engine. It is very much easy to add more information or statistical graph to our visualised interface. The graphical engine needs to run in separate thread so that it does not affect the performance of MetaDGPEngine.

Below is a typical screenshot of the running application's training pane, screenshots can not demonstrate transparent vector animations implemented by my 2D graphic rendering engine.



192.168.1.100 : 6277
SYMBOL: MSFT
Investment Cycle: 16 Days
Required Return: 4%
Transaction Cost: 0%

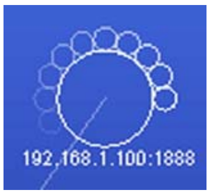
Reproduction Rate: 35%
Crossover Rate: 60%
Mutation Rate: 5%
Max Tree Depth: 17
Max Branch Depth 6

[Island Setting]
Population Size: 100
Generations to Run: unlimited

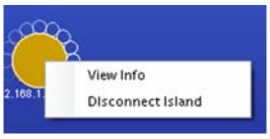
```
192.168.1.102:1188 sent new migrant of genration 77
192.168.1.100:1888 sent new migrant of genration 92
New Candiate received from 192.168.1.100:1888
Island :1188 has evolved to new genration
192.168.1.100:1886 sent new migrant of genration 80
New Candiate received from 192.168.1.100:1886
192.168.1.102:1188 sent new migrant of genration 83
New Candiate received from 192.168.1.102:1188
```

Screenshot of the running application's training pane

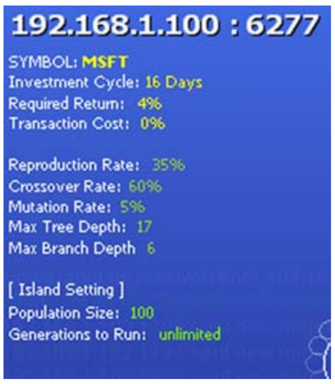
An island is represented in circle form:



The small circles around it are the migrants it has sent to the mainland, elder individual will gradually fade out. Below the bigger circle is the island IP address and port number.



All elements are mouse sensitive in my graphic engine, in this example a pop-up menu will be shown when user right-click on an island. Administrator can force to disconnect an island if required.

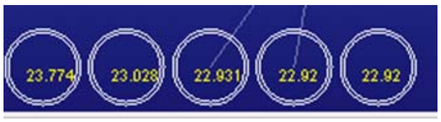

192.168.1.100 : 6277

SYMBOL: MSFT
Investment Cycle: 16 Days
Required Return: 4%
Transaction Cost: 0%

Reproduction Rate: 35%
Crossover Rate: 60%
Mutation Rate: 5%
Max Tree Depth: 17
Max Branch Depth 6

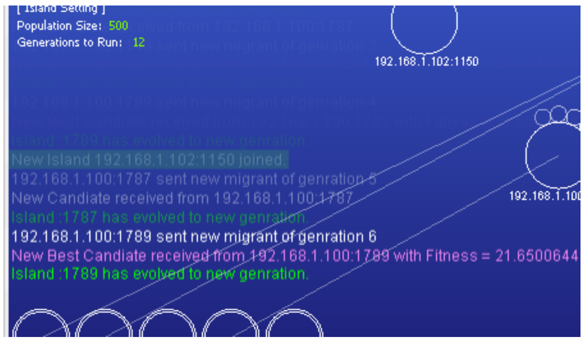
[Island Setting]
Population Size: 100
Generations to Run: unlimited

This area shows the parameter and status of genetic programming. In this example, the server address is 192.168.1.100 with port 6277. We are training on Microsoft's stock. And all others are the parameters described in sections before.



Candidates Pane

Current best individuals in mainland are shown at the bottom with its fitness value rounded to the nearest 0.01. The number of *candidates* to store in GeneBank is set by user in main interface. Three most recent received candidates are connected by a straight line to its originating island.



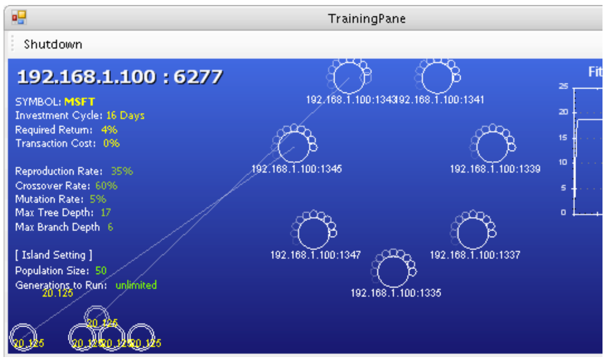
```
[ Island Setting ]
Population Size:  500
Generations to Run:  12
```

```
Island :1789 has evolved to new genration.
New Island 192.168.1.102:1150 joined.
192.168.1.100:1787 sent new migrant of genration 5
New Candiate received from 192.168.1.100:1787
Island :1787 has evolved to new genration.
192.168.1.100:1789 sent new migrant of genration 6
New Best Candiate received from 192.168.1.100:1789 with Fitness = 21.6500644
Island :1789 has evolved to new genration.
```

RollerTextBox

A rolling message pane is shown at the left. New information gradually roll from bottom, elder messages will fade out gradually. This component actually is capable of many things. For example, user can set each message's background and foreground colours; from how many lines the message start to fade out; the speed of rolling up, etc… This *RollerTextBox* component is employed many places throughout this software package, providing a favourable visual effect.

And the size of any element is in respect with ratios auto-adjusted, so that form can be resized flexibly without problem.

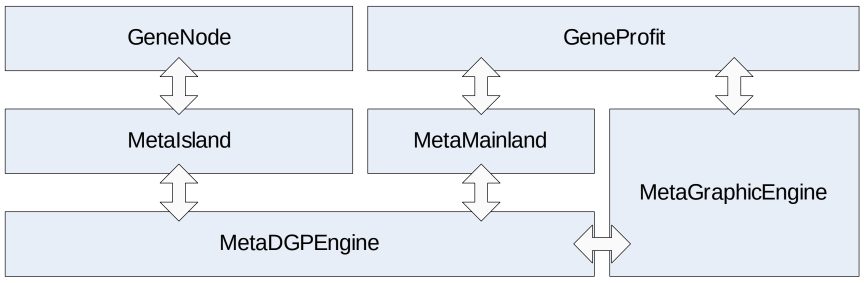


```
192.168.1.100 : 6277
SYMBOL: MSFT
Investment Cycle: 16 Days
Required Return:  4%
Transaction Cost:  0%

Reproduction Rate:  35%
Crossover Rate: 60%
Mutation Rate:  5%
Max Tree Depth:  17
Max Branch Depth  6

[ Island Setting ]
Population Size:  50
Generations to Run:  unlimited
```

Example of a resized very small form

3.4 Application Structure

A comprehensive application structure needs to be designed to support such a rich content software package. At the beginning of development of this project, I have decided that a generic distributed genetic programming (DGP) engine needs to be implemented first and bug free. Then a graphical engine and modules for financial genetic programming are added thereafter. This provides good milestones to schedule the development of this project. In terms of final product, we also have a generic DGP engine and a 2D vector graphic engine reusable for future projects.



Application Structure Outlined

(Final project adds up to 62,052 lines of programming codes. Therefore class diagram is taken out from this report, since it can not fit into any page. Please refer to Visual Studio.Net™ Class Explorer to browse the classes implemented in this project's source code.)

3.4.1 Meta Distributed Genetic Programming Engine (MetaDGPEngine)

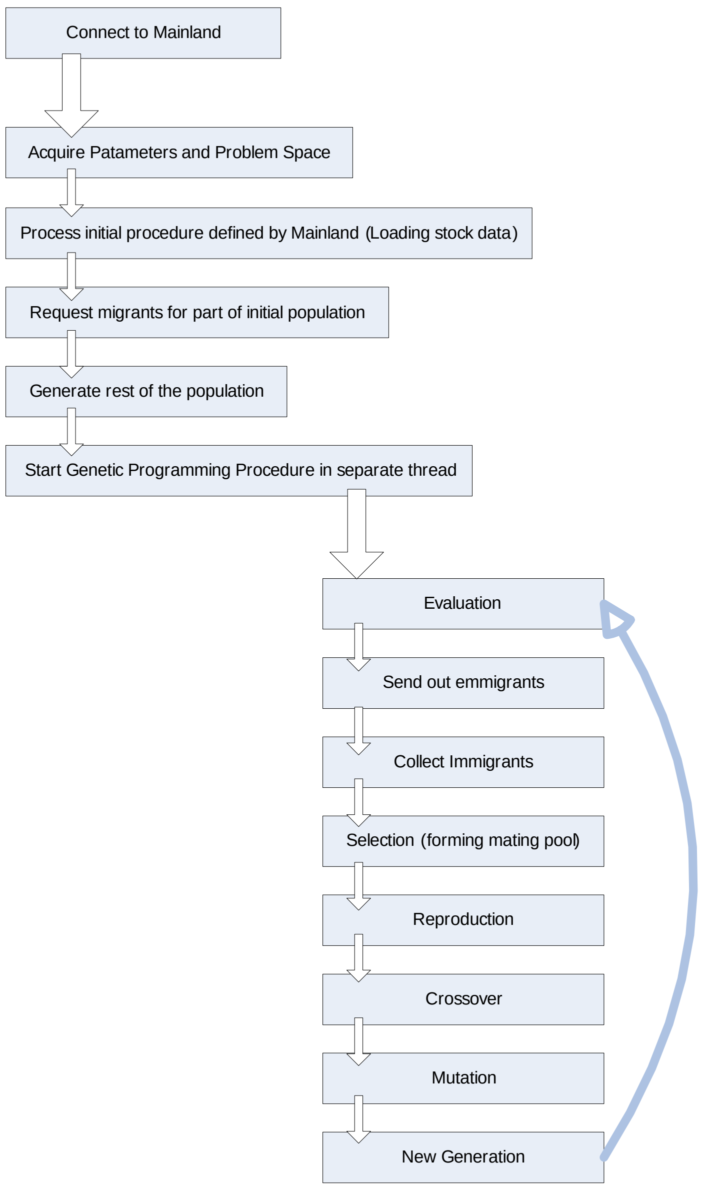
MetaDGPEngine is a generic DGP engine that implements all related models described yet. Here theoretical models are finally coded.

In terms of genetic programming, MetaDGPEngine's tree genotype accepts all user defined types, and has internal syntax check to guarantee generated programs can be compiled.

Controlled coarse grained parallel distributed network is implemented by TCP socket. The reason is that I design this software package to be used in intranet, rather than P2P over internet. Although migrants fit more into UDP packet, a continuous connection between islands and mainland is favourable for control and management. More detail in implementation chapter.

3.4.1.1 MetaIsland

MetaIsland is the implementation of islands in Island Model. Ideally it does not need to worry about the problem space, this information should be provided by mainland once connected. In my implementation, the MetaIsland requires completely *zero configuration.* (Except server address since we do not want to enable host discovery) Its only job is to repeat genetic programming procedure. The basic control flow of MetaIsland is illustrated as:


Connect to Mainland
Acquire Patameters and Problem Space
Process initial procedure defined by Mainland (Loading stock data)
Request migrants for part of initial population
Generate rest of the population
Start Genetic Programming Procedure in separate thread
Evaluation
Send out emmigrants
Collect Immigrants
Selection (forming mating pool)
Reproduction
Crossover
Mutation
New Generation

Sequential Diagram of MetaIsland

3.4.1.2 MetaMainland

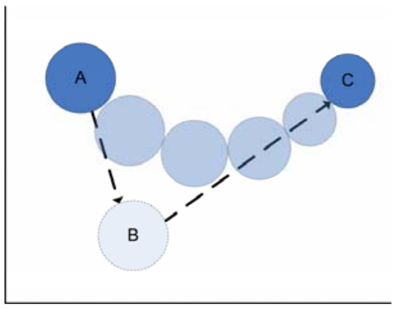
MetaMainland is essentially a TCP server. Once started, it listens to server port for incoming messages and process messages. Features of the TCPSocketServer implemented in this project are described in implementation chapter. The server needs to have a good load balance, scalability, and vulnerability. MetaMainland has a basic authentication to ensure connections are from MetaIsland clients. Incoming messages are mixed by commands and datagram. I have designed a command system similar to FTP for intercommunication between island and mainland. Datagram may contain parameters, binary programs, and serialised individuals.

3.4.1.3 Meta Graphics Engine

MetaGraphicEngine is a 2D vector graphic engine. With the aim of encapsulation and ease of use, it has a similar script commands as in game development, to automatically generate animations in between any two key frame. As a freelance designer, I borrowed this idea from my familiar designing package Macromedia Flash™. Each action of an element takes arguments of *new element properties*, and a length of *animation interval*. For example, series orders for an element at position A like:

```
MoveTo(pos(B), 200ms);
MoveTo(pos(C), 200ms);
ResizeTo(size(C), 200ms);
```

A 200ms animation of frames will be automatically generated with actual track path of transparent circles shown below:

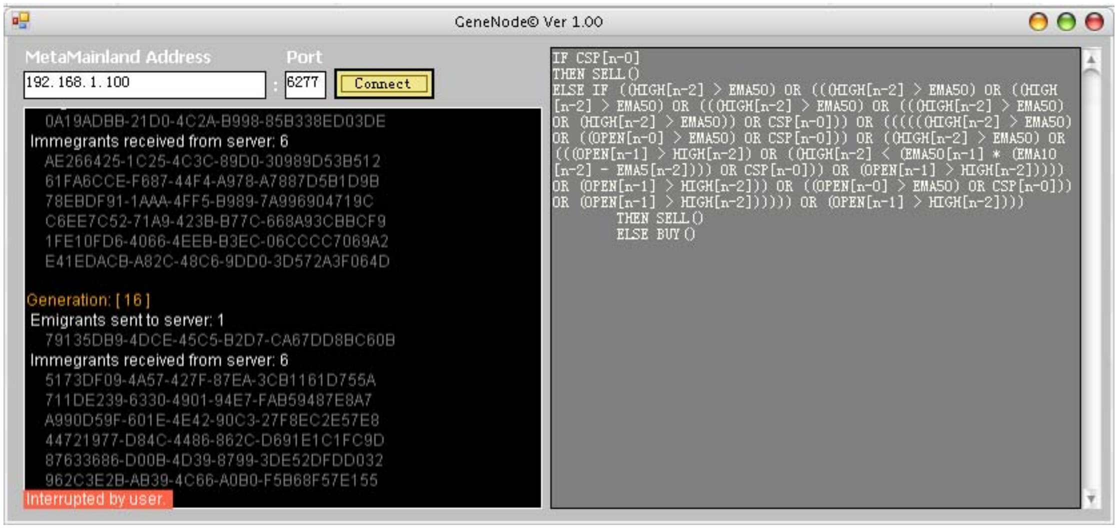


Automated animation

Frames and sprites (the way element is called in graphic engine) are internal information maintained by graphic engine. Calling party only need to order script command as above and animated frames will be generated and attached to sprite. Besides MoveTo and ResizeTo, it also support transparency animation command FadeTo. Therefore, the rendered picture is easily brought alive with series of command.

3.4.2 GeneNode

GeneNode is a GUI interface of MetaIsland that works as a peer node in our parallel distributed network, an example screenshot looks like:



```
192. 168. 1. 100 : 6277
```

```
   0A19ADBB-21D0-4C2A-B998-85B338ED03DE
 Immegrants received from server: 6
   AE266425-1C25-4C3C-89D0-30989D53B512
   61FA6CCE-F687-44F4-A978-A7887D5B1D9B
   78EBDF91-1AAA-4FF5-B989-7A996904719C
   C6EE7C52-71A9-423B-B77C-668A93CBBCF9
   1FE10FD6-4066-4EEB-B3EC-06CCCC7069A2
   E41EDACB-A82C-48C6-9DD0-3D572A3F064D

Generation: [16]
 Emigrants sent to server: 1
   79135DB9-4DCE-45C5-B2D7-CA67DD8BC60B
 Immegrants received from server: 6
   5173DF09-4A57-427F-87EA-3CB1161D755A
   711DE239-6330-4901-94E7-FAB59487E8A7
   A990D59F-601E-4E42-90C3-27F8EC2E57E8
   44721977-D84C-4486-862C-D691E1C1FC9D
   87633686-D00B-4D39-8799-3DE52DFDD032
   962C3E2B-AB39-4C66-A0B0-F5B68F57E155
Interrupted by user.
```

```
IF CSP[n-0]
THEN SELL()
ELSE IF ((HIGH[n-2] > EMA50) OR (((HIGH[n-2] > EMA50) OR ((HIGH
[n-2] > EMA50) OR (((HIGH[n-2] > EMA50) OR (((HIGH[n-2] > EMA50)
OR (HIGH[n-2] > EMA50)) OR CSP[n-0])) OR (((((HIGH[n-2] > EMA50)
OR ((OPEN[n-0] > EMA50) OR CSP[n-0])) OR ((HIGH[n-2] > EMA50) OR
(((OPEN[n-1] > HIGH[n-2]) OR ((HIGH[n-2] < (EMA50[n-1] * (EMA10
[n-2] - EMA5[n-2]))) OR CSP[n-0])) OR (OPEN[n-1] > HIGH[n-2]))))
OR (OPEN[n-1] > HIGH[n-2])) OR ((OPEN[n-0] > EMA50) OR CSP[n-0]))
OR (OPEN[n-1] > HIGH[n-2]))))) OR (OPEN[n-1] > HIGH[n-2])))
        THEN SELL()
        ELSE BUY()
```

Screenshot of GeneNode program

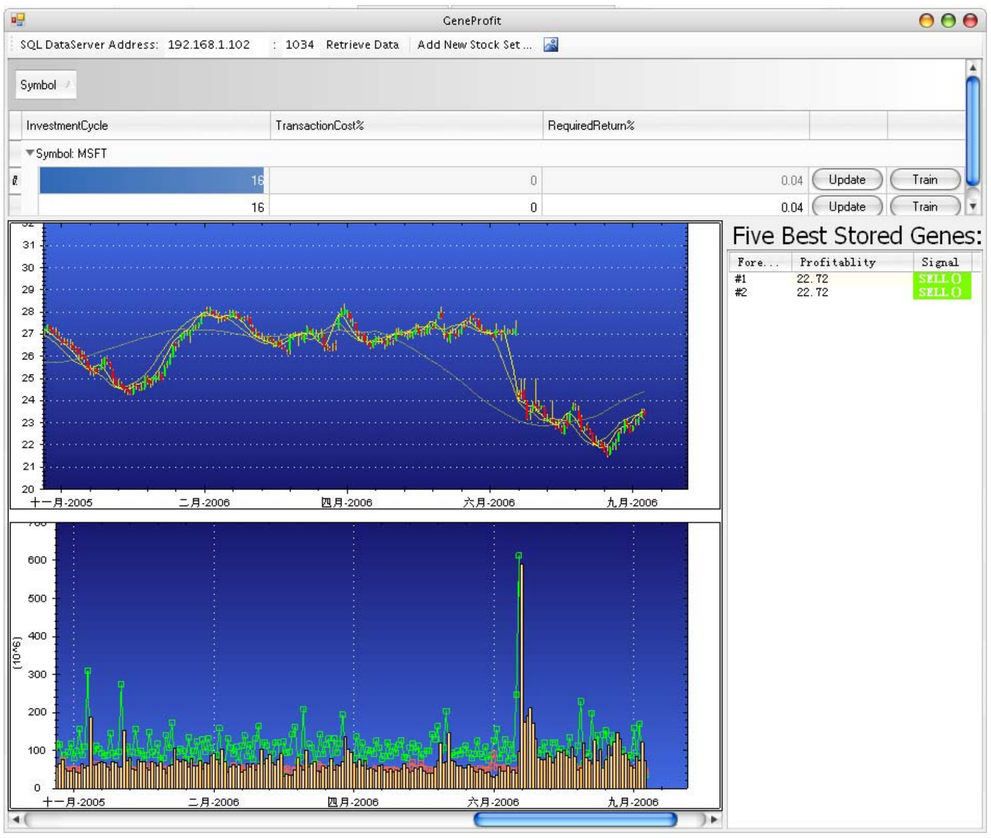
GeneNode has a very simple interface inherited from the simplified design of MetaIsland, although the mechanism behind involved a great amount of logics implemented on design models. The only input is server address and port. The text box on the left shows information as rolling text messages. We can see it just finished generation 16, one emigrant were sent to mainland, and 6 immigrants were received from other islands. Those strings are unique IDs of individual generated by system.

The text box on the right updates the current best individual's program from last evaluation procedure.

3.4.3 GeneProfit

GeneProfit is a control platform for financial market prediction. It also instantiates MetaMainland. The first screen showed to user is the stock management interface, where user can add/remove setting for stock predictions, either different sets for same stock or multiple stocks. Unlike other scientific programs, the interface of GeneProfit is kept very much simple. The vast logics of parallel distributed genetic programming are hidden from end user.

GeneProfit update information from Yahoo!™ Finance at background, historical data and statistical data calculated upon are then uploaded to Database. User can view the result of best stored strategy programs in Genebank for today's stock ticker. User can also choose to train a particular setting, which will lead to train pane that actually instantiates MetaMainland and MetaGraphicEngine.


GeneProfit

SQL DataServer Address: 192.168.1.102 : 1034 Retrieve Data Add New Stock Set ...

Symbol

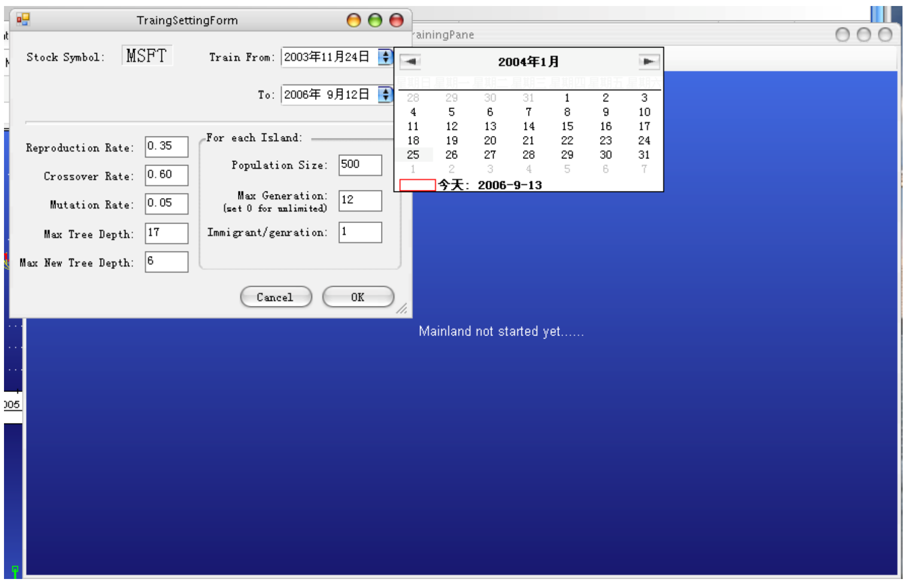
InvestmentCycle	TransactionCost%	RequiredReturn%		
Symbol: MSFT				
16	0	0.04	Update	Train
16	0	0.04	Update	Train

Five Best Stored Genes:

Fore...	Profitablity	Signal
#1	22.72	SELL()
#2	22.72	SELL()

Screenshot for GeneProfit

When user chooses to “Train” a strategy setting, the training setting window brought up:


TraingSettingForm

Stock Symbol: MSFT Train From: 2003年11月24日

To: 2006年 9月12日

Reproduction Rate: 0.35
Crossover Rate: 0.60
Mutation Rate: 0.05
Max Tree Depth: 17
Max New Tree Depth: 6

For each Island:
Population Size: 500
Max Generation: (set 0 for unlimited) 12
Immigrant/genration: 1

Cancel OK

2004年1月

28	29	30	31	1	2	3
4	5	6	7	8	9	10
11	12	13	14	15	16	17
18	19	20	21	22	23	24
25	26	27	28	29	30	31
1	2	3	4	5	6	7

今天: 2006-9-13

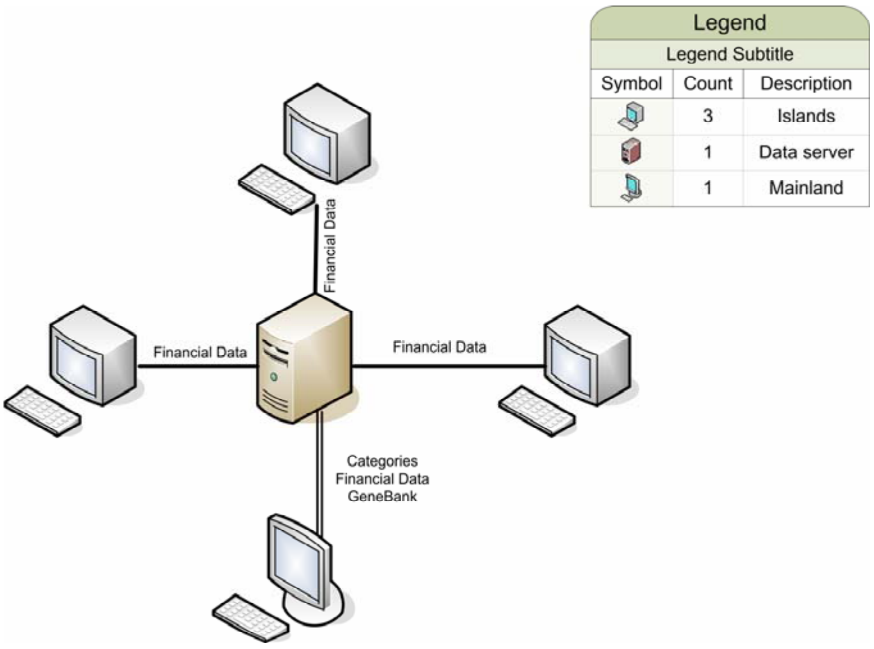
ainingPane

Mainland not started yet......

User can adjust basic parameter for genetic programming in this window, and start the training. Training can be applied on customized period of historical data. In this example, although Yahoo!™ Finance stores data for MSFT from year 1971, we choose to train individuals from year 2003 to now. Calendar format is customized by user’s operation system in this screenshot.

3.4.4 SQL Database

Conventionally, genetic programming implementations simply use text data files as input/output. This is a quick and easy solution. However, for a distributed cluster, and especially scale of this project, which involved large quantity of historical data and historical trained programs, local text files are the worst choice. But a solution is actually quite simple. All we need is a database system. Most modern database system supports *distributed* computing, and offers good performance on load balancing and access speed.



Data Server

There are three tables in our design of database:

1. **Category**: stores the settings for investment preferences.
2. **GeneBank**: stores the individuals from each training process.
3. **FinancialData**: one table per stock symbol. The historical data and technical statistics, only this table is accessible to islands in financial genetic programming.

4 | Implementation

This chapter describes areas on implementation of this project, as a supplementary to most lacking details from previous chapters.

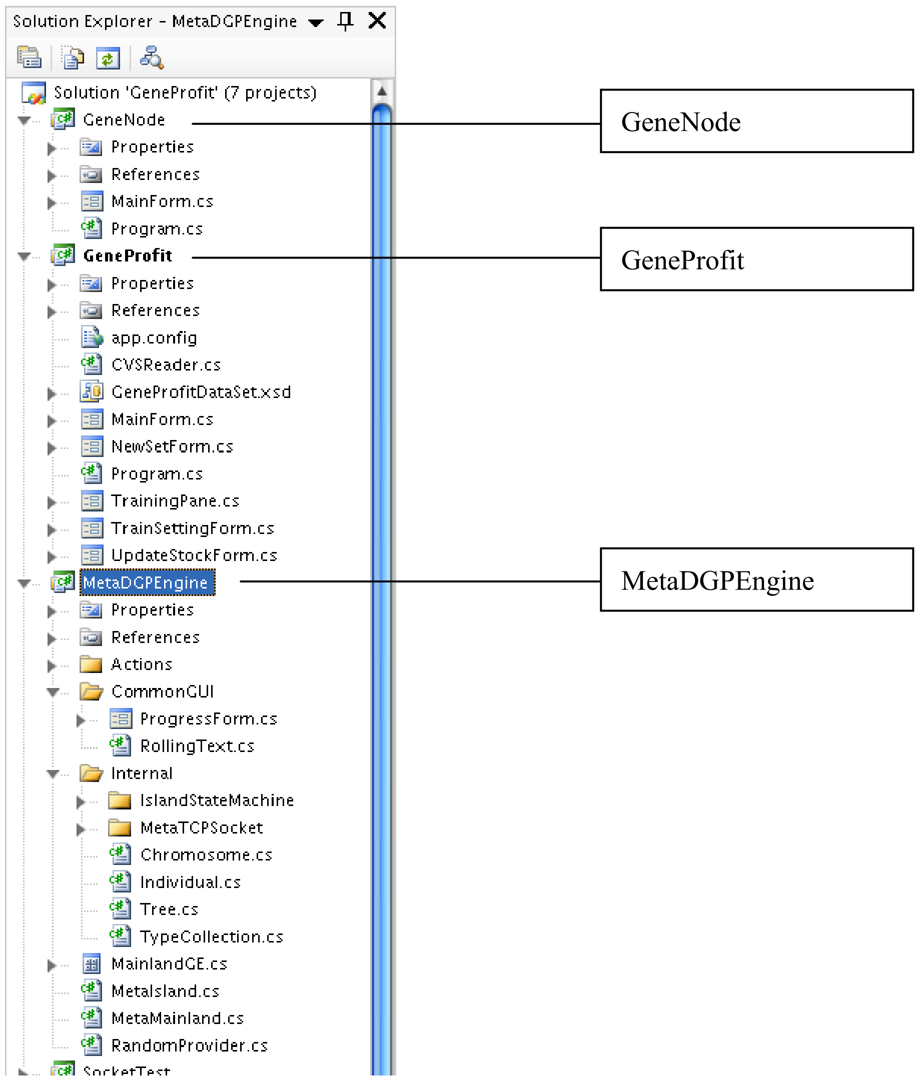
4.1 Programming Language and Development Environment

This project is chosen to be implemented by *C# 2.0*. A direct reason is that it is a good chance to learn this fast growingly popular programming language in software development industry. Secondly, dotNet framework offers an enormous but still very easy to use class library. Especially in socket programming, multi-threading, events and delegates, and GDI+ etc., these advantages are directly targeting at most elements of this project. Thirdly, the *Visual Studio.Net* developing environment is a very convenient and efficient IDE.

The SQL database system is chosen to be taken by *Microsoft SQL Server Express 2005*. Performance is needless to say for the scale of database used in this project. Firstly, it offers an industry leading graphical management tools. Building and configuring the whole database for this project is within a few clicks. Secondly, Visual Studio.Net IDE also fully support SQL Server. The coding for connections to SQL Server is also greatly simplified.

4.2 Project Structure

It is better to describe this section by a screenshot a project's solution explorer in Visual Studio.Net:



Figure# Where have 62,052 lines of codes gone?

4.3 Genetic Programming

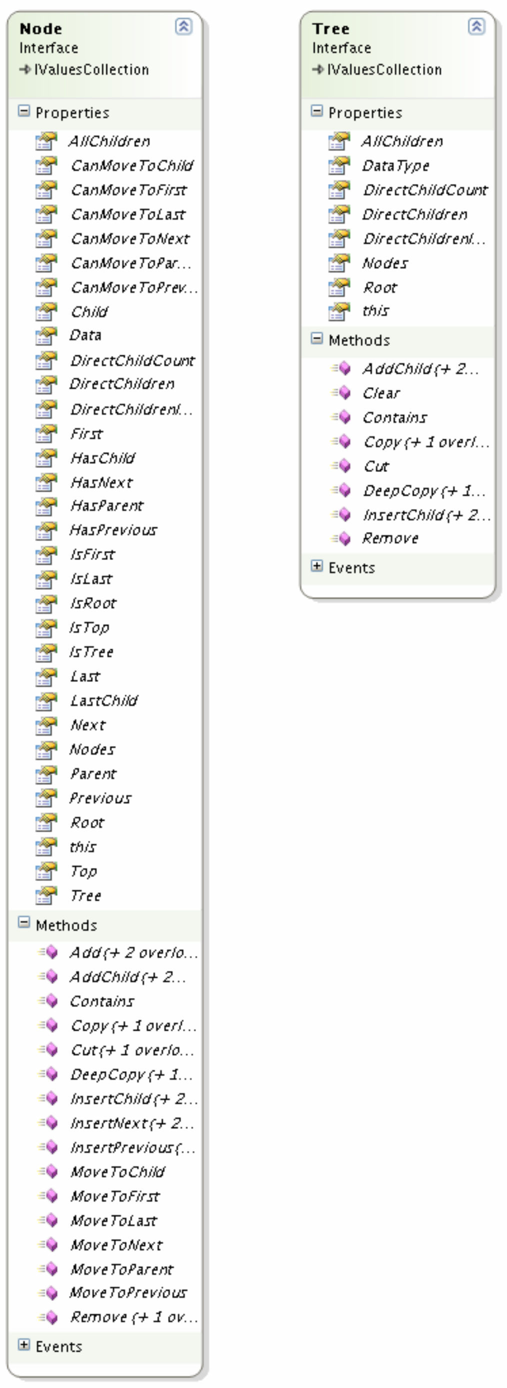
4.3.1 Generic Tree

C# 2.0 provides many new features on generic types like List, HashTable, Dcitionary, etc. But there is no *tree* type. In addition, even there exists one in C#'s library, we could not use it directly. Our genetic programming procedure requires extensive copy and cut operations on branches of trees. But the generic types provided in C# are actually by references, i.e., pointers. Therefore, a customized tree structure needs to be implemented for our project. And it is challenging to fulfil these features:

1. It needs to support copy, cut, and paste of sub-trees.
2. Tree nodes need to store object references as content.

3. It should be able to quickly search for given object.
4. It offers functionality to be serialised to XML strings, and also decode from a given XML string.
5. It should use little memory and free up memory used quickly, since genetic operators development large quantities of new sub-trees quickly.

In our project, there is a complicated Tree structure implemented covering all above features in namespace *MetaDGPEngine.Internal.Tree*



Node
Interface
→ IValuesCollection

Properties
- *AllChildren*
- *CanMoveToChild*
- *CanMoveToFirst*
- *CanMoveToLast*
- *CanMoveToNext*
- *CanMoveToPar…*
- *CanMoveToPrev…*
- *Child*
- *Data*
- *DirectChildCount*
- *DirectChildren*
- *DirectChildrenI…*
- *First*
- *HasChild*
- *HasNext*
- *HasParent*
- *HasPrevious*
- *IsFirst*
- *IsLast*
- *IsRoot*
- *IsTop*
- *IsTree*
- *Last*
- *LastChild*
- *Next*
- *Nodes*
- *Parent*
- *Previous*
- *Root*
- *this*
- *Top*
- *Tree*

Methods
- *Add(+ 2 overlo…*
- *AddChild(+ 2…*
- *Contains*
- *Copy(+ 1 overl…*
- *Cut(+ 1 overlo…*
- *DeepCopy(+ 1…*
- *InsertChild(+ 2…*
- *InsertNext(+ 2…*
- *InsertPrevious(…*
- *MoveToChild*
- *MoveToFirst*
- *MoveToLast*
- *MoveToNext*
- *MoveToParent*
- *MoveToPrevious*
- *Remove(+ 1 ov…*

Events

Tree
Interface
→ IValuesCollection

Properties
- *AllChildren*
- *DataType*
- *DirectChildCount*
- *DirectChildren*
- *DirectChildrenI…*
- *Nodes*
- *Root*
- *this*

Methods
- *AddChild(+ 2…*
- *Clear*
- *Contains*
- *Copy(+ 1 overl…*
- *Cut*
- *DeepCopy(+ 1…*
- *InsertChild(+ 2…*
- *Remove*

Events

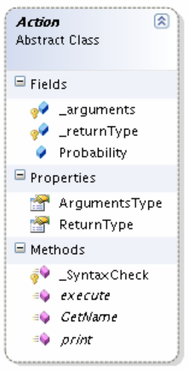
4.3.2 Tree Creation Algorithm

Instead of commonly accepted GROW and FULL algorithms for tree-creation in genetic programming. This project adopts a newer and better Strongly-Typed Probabilistic Tree Creation (STPTC) (Luke, 2000). PTC is a modified version of GROW by employing probability of appearance of functions and variables in tree. Also tree size is well controlled in this algorithm. For financial genetic programming, we don't want the strategy programs generated to be too long. This algorithm is modified to be strong typed according to the defined generic function nature in this project (next section on Action). The pseudocode of this algorithm is: (Luke, 2000)

```
Given:
    maximum depth bound D
    disjoint nonterminal subsets $N_y$ of nonterminal set N for each $y \in Y$
    disjoint terminal subsets $T_y$ of terminal set T for each $y \in Y$
    computed nonterminal-choice-probalilities py for each $y \in Y$
    For each $T_y$ and $N_y$
        Probabilities $q_{n,y}$ and $q_{t,y}$ for each $t_y \in T_y$ and $n_y \in N_y$
    Return type for the tree $y_r \in Y$
Do:
    New tree T = STPTC(0, $y_r$)

STPTC(depth d, return type $y \in Y$)
    Returns: a tree of depth <= D-d and of type y
    IF d=D, return a random terminal from $T_y$ (by $q_{t,y}$ possibility)
    ELSE IF, with probability $p_y$ a nonterminal must be picked,
        Choose a non terminal $n_y$ from Ny (by $q_{n,y}$ possibilities)
        For each argument a of $n_y$ of argument type $y_a$
            Fill a with STPTC(d+1, $y_a$)
        Return the completed nonterminal $n_y$ with filled arguments
        Else return a terminal from $T_y$(by $q_{t,y}$ probabilities)
```

4.3.3 Action instead of Function/Variable

In this project, functions and variables in genetic programming terminology are abstracted as *Actions*. Because Variables is simple a function that doesn't require arguments. For every action, it has a return type, and a collection of types of each argument. This will be the criteria for tree creation and form the foundation for the whole syntax of generated programs.



4.3.4 Binary Tournament Selection Method

The algorithm for selection in genetic programming in this project is Binary Tournament Selection (Koza, 1995). Each time two individuals are randomly selected, the one with higher fitness value will be selected into mating pool. This mimics the process of natural competition. And also guarantees that a super fit individual is not overly cloned in new generation. Hence loose the convergence of process of genetic programming.

4.3.5 Crossover and Mutation in Chromosome

The way how crossover and mutation works in this project is exactly as illustrated in background chapter. Crossover and mutation is implemented as public method in Chromosome class rather than individual, because Individual in MetaDGPEngine can incorporate more than one chromosome. This was due to that in some circumstances, for instance another design of forecasters in financial market prediction, an individual could have short-term, mid-term and long-term chromosomes as a fully capable forecaster.

There is a specially *PickNonterminalRate* by default set to 90% so that genetic operators tend to apply on inner trunk of trees.

4.3.6 Mersenne Twister Random Generator

An important element of Genetic Programming implementation is randomicity, which is fundamental to various aspects such as search space and selection process. However, in terms of implementation, most researches lose sight of the random generator provided by programming language. Although .Net framework is still new technology, the random generator it included in library implements a nearly 40 years old fluffy algorithm! This problem is similar to Java as well. Hence the system's random generator set its own bias for our Genetic Programming implementation. Due to the enormous frequency that random number is used, this problem is never small to neglect. Therefore, in this project, I have adopted Mersenne Twister as our random generation, after researches and comparisons over many existing algorithms on efficiency and quality.

"The Mersenne twister is a pseudorandom number generator developed in 1997 by Makoto Matsumoto and Takuji Nishimura. It can provide fast generation of very high quality pseudorandom numbers, having been designed specifically to rectify many of the flaws found in older algorithms." (Wikipedia)

This algorithm was designed to have a colossal period of 219937 − 1 (the creators of the algorithm proved this property). This period explains the origin of the name: it is a Mersenne prime, and some of the guarantees of the algorithm depend on internal use of Mersenne primes.

In practice, there is little reason to use larger ones, as most applications do not require 2^19937 unique combinations. For example, to guarantee a completely random deck of cards only requires 52! combinations - approximately 2^225. Second, it has a very high order of dimensional equidistribution. By default, there is negligible serial correlation between successive values in the output sequence.

It is faster than all but the most statistically unsound generators. Quality is also guaranteed since it passes numerous tests for statistical randomness, including the stringent Diehard tests.

Pseudocode:

```
// Create a length 624 array to store the state of the generator
 var int[0..623] MT
 var int y
 // Initialise the generator from a seed
 function initialiseGenerator ( 32-bit int seed ) {
    MT[0] := seed
    for i from 1 to 623 { // loop over each other element
       MT[i] := last_32bits_of((69069 * MT[i-1]) + 1)
    }
 }

 // Generate an array of 624 untempered numbers
 function generateNumbers() {
    for i from 0 to 622 {
       y := 32nd_bit_of(MT[i]) + last_31bits_of(MT[i+1])
       if y even {
          MT[i] := MT[(i + 397) % 624] bitwise_xor (right_shift_by_1_bit(y))
       } else if y odd {
          MT[i] := MT[(i + 397) % 624] bitwise_xor (right_shift_by_1_bit(y)) bitwise_xor
(2567483615)
       }
    }
    y := 32nd_bit_of(MT[623]) + last_31bits_of(MT[0])
    if y even {
       MT[623] := MT[396] bitwise_xor (right_shift_by_1_bit(y))
    } else if y odd {
       MT[623] := MT[396] bitwise_xor (right_shift_by_1_bit(y)) bitwise_xor (2567483615)
    }
 }

 // Extract a tempered pseudorandom number based on the i-th value
 function extractNumber(int i) {
    y := MT[i]
    y := y bitwise_xor (right_shift_by_11_bits(y))
    y := y bitwise_xor (left_shift_by_7_bits(y) bitwise_and (2636928640))
    y := y bitwise_xor (left_shift_by_15_bits(y) bitwise_and (4022730752))
    y := y bitwise_xor (right_shift_by_18_bits(y))
    return y
  }
```

4.2 Data Structures and Reflection Programming

4.2.1 Environment Options

Environment Options contains parameters for many areas, like Genetic Programming parameters, Distributed Network parameters, and financial genetic programming parameters. For implementation of a generic DGP engine, these are all combined into a single *hashtable*, called EnvironmentOptions. This hashtable is shard by both mainland and island, so it serves and a collection of general variables across the distributed network. A detailed Environment

Options set for this project is listed below:

Parameter Name	Default Value
Population Size	200
Terminating Generation	unlimited
Reproduction Rate	35%
Crossover Rate	60%
Mutation Rate	5%
Max Tree Depth	17
Max New Sub Tree Depth	6
Data Server Address	192.1.168.102,1034
Mainland Address	192.1.168.100:6277
Stock Symbol	-
Date From	-
Date To	-
Investment Cycle	14 days
Required Rate of Return	4%
Transaction Cost	5%

Default Environment Options in this project

4.2.2 Tree and Hashtable serialisation

In order to transfer content of individuals and environment options, a serialisation method is needed. The most common choice is XML serialisation. Although C# provides a full mechanism for XML serialisation, it is very much buggy and has lots of restrictions. Therefore XML serialisation needs to be implemented by hand. In this project, the main data structures need serialisations are trees and hashtables.

For trees, actions contained in nodes are serialised by a method call to get the name of the action. The tree type has an internal adapter that temporarily stores this string tree and then uses C#'s system call to serialise.

However for hashtables, it is a bit more complicated. Since objects contained in hashtable are of different types. Therefore, for each element of hashtable, the key name is serialized to string, while the value is interpreted into two fields, one is type, and the other is the value in string form. For example:

```
<EnvironmentOptions>
  <CrossoverRate>
    <Type>System.Double</Type>
    <Value>0.65000000000</Value>
  </CrossoverRate >
  ...
```

More details in type instantiation can be found in next section talking about runtime reflection programming.

4.2.3 Late Binding via .Net Reflection

As described above, MetaDGPEngine is designed for generic purpose, not merely targeted at stock market prediction. It might be a lot easier for a single machine Genetic Programming implementation, but under a P2P environment things become much more complicated. We want our GeneNode to be a “thin” client, i.e. a “click-and-run” program without any configuration other than server address. This would greatly reduce the cost of maintenance, since all the modification and configuration for different problem or different parameters in same problem are only on server side. This is natural for Genetic Programming operators (selection, reproduction, crossover and mutation), because they do not care about the actual meaning of programs evolved. Problem comes from Evaluation process, where we need to access problem-specified data and functionalities. In a prototype version of this project, .Net *Remoting* was introduced. Those problem-specified data and methods are supported by remote calling server. However, soon we found out that Evaluation is the most time-consuming process, so this would completely slow down the whole performance and generate large overhead of network transactions, which eliminated any advantages on distributed computation. Therefore, we need a way to collection types and methods from server side and instantiate locally during runtime. Without reflection, this would be never made possible.

Reflection is an activity in computation that allows an object to have information about the structure and reason about its own computation. Reflective programming technique has been growing popular recent years. .Net’s industry leading reflection support is another reason C# is chosen. Many of the services available in .Net and exposed via c# depend on the presence of metadata. Reflection examines existing types via their metadata and is done using a rich set of types in the *System.Reflection* namespace. It is possible to dynamically create new types at runtime via the classes in the *System.Reflection.Emit* namespace, and/or extend the metadata for existing types with custom attributes. At runtime, these elements are all contained within an *AppDomain*.

4.2.3.1 Generic Type Converter

Parameters and variables are received from server as serialized hashtable, for example an integer describing MaxGenerations could be serialized as:

```
…
<MaxGenerations>
    <Type>System.Int32</Type>
    <Value>500</Value>
</MaxGenerations>
…
```

Upon received required variables, GeneNode need to create (instantiate) this variable according to its type, and then load the data by converting string to this particular type. We use Reflection to check if target type has a TypeConverter interface. A lot of non-basic types, including Unit, implement this because TypeConverter is also the way the designer in Visual Studio is able to use these types. If it doesn't have a TypeConverter, try falling back to Convert.

```
Type propertyType = Type.GetType(node["Type"].InnerText);
string value = node["Value"].InnerText;

object convertedValue = null;
object[] attributes = propertyType.GetCustomAttributes(typeof(System.ComponentModel.TypeConverterAttribute), false);
foreach (System.ComponentModel.TypeConverterAttribute converterAttribute in attributes)
{
    System.ComponentModel.TypeConverter converter =
(System.ComponentModel.TypeConverter)Activator.CreateInstance(Type.GetType(converterAttribute.ConverterTypeName));
    if (converter.CanConvertFrom(value.GetType()))
    {
        convertedValue = converter.ConvertFrom(value);
        break;
    }
}
if (convertedValue == null)
{
    try
    {
        convertedValue = Convert.ChangeType(value, propertyType);
    }
    catch (InvalidCastException)
    {
        Console.WriteLine("Can't convert");
    }
}
```

4.2.3.2 Load DLL Assemblies and Invoke at Runtime

Actions are loaded by each GeneNode during runtime, for stock market, they could be, for example, BUY(), HOLD(), SELL(), KSTO5[n-2], etc. Codes are compiled into a DLL file and downloaded by GeneNode upon start. GeneNode does not know how many actions are there and what types of parameters those actions require to execute. With Refelction, this is made easy:

1. Instantiate types from DLL modules, load assemblies into AppDomain

    Object o = Activator.**CreateInstance**(“Actions.dll”, sType);

2. Retrieve common method and examines its parameters

    MethodInfo mi = aType.**GetMethod**(“execute”);
    foreach (ParameterInfo pi in mi.**GetParameters**())
    {
        … …
    }

3. Invoke method with parameters

    mi.**invoke**(o, {para1, para2, para3 … });

4.3 Controlled Coarse Grain Distributed Processing

In design chapter we have listed the rationale of choosing a TCP protocol as the fundamental network communication layer for our distributed network. The parties communicating via TCP socket can live on different machines on the network, but they can also live on the same machine, or even within the same process. C# has a sound class in System.Net library called *Socket* that does most of the jobs.

However, it is still far from a finished encapsulated helper class to be used in this project. We need to implement a Server side and a Client side socket class for our mainland-island TCP communication. And further for a network service that is scalable and fault tolerance, we need to replace the threadpool provided in .Net library.

4.3.1 MetaTCPSocket Components

MetaTCPSocket namespace contains sealed classes trying to provide an all-in-one solution to the need of fast and reliable TCP client/server communications. MetaTCPClient implements the socket logics for client side, MetaTCPServer does the server side. MetaTCPClient is a very straightforward wrapper of Socket class with little added features like string form datagram. This is because that a client only needs to connect to one server, and clients usually expect to be blocked while waiting for information coming from server side. However, for a server side component, careful designs need to be done to ensure a bunch of features. Socket programming has always been mentioned as disaster since in network environment, nothing is impossible; and impossible come from no where.

4.3.1.1 Scalability

A server side socket component needs to be implemented in the way of multi-threading. Therefore communication requests from different clients can all be served. We do not want to lose any incoming request, so each request need to be served in its own thread. Therefore, we need a mechanism to management this collection of threads.

4.3.1.2 Load Balance

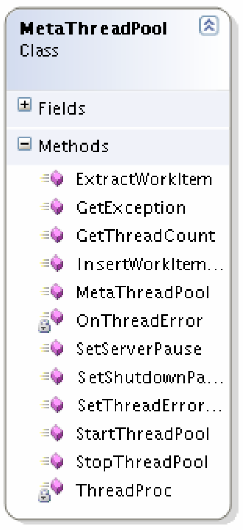
Since this parallel distributed genetic programming system is designed mainly for time-consuming computation tasks. Most of the time clients should be doing its own calculations. Therefore the network traffic is not very much significant. Therefore, server side socket merely adds a small delay to avoid DoS attack. Provided that our system is runned in intranet, there is little chance to be attacked in this form, so this feature is only added to prevent certain kind of faulty client.

4.3.1.3 Fault Tolerance

Server needs to provide a high degree of fault tolerance and this feature needs to be transparent from user. The most significant problem is thread-safe. All communication requests are processed in separate thread. We need to check if the connection is already bad, so we can disconnect it and recycle the resources occupied.

4.3.2 MetaThreadPool

All above features demand an effective mechanism to manage a collection of threads. C# has a *System.Threading.ThreadPool* class provides a pool of threads that can be used to post work items, process asynchronous I/O, wait on behalf of other threads, and process timers. However, after some research of documentation, I found that this thread pool is actually implemented in program's *heap*. This is definitely not favourable to be used in a socket communication component. Network communication has mysterious possibilities, especially when applied to large scale communications. A threadpool placed at the general memory could lead to fatal error. It was observable that in early buggy implementation of MetaSocket component, any error directly lock the "Frameworkserver.exe" which is the process of .Net Framework Dynamic Library. Therefore, a MetaThreadPool specially designed on purpose, trying to do similar job as in system library, but offers locally controlled threads management.



4.3.3 Command Set

A simple command set similar to FTP protocol is designed for communication between Mainland and Island.

Command String	Meaning
CI	**C**reate **I**sland. Server sends out environment options and initial migrants for connecting island. Inside mainland, migrants pool is prepared for this island, as well as other collections In MetaGraphicEngine, animation is created for new island and positions for other islands.
NG	**N**ew **G**eneration. Mainland receives emigrants from island one by each, until ACK command. If any new individual is better than any current one in Candidates pool, add new individual into candidates pool and knock out the excess one. Then emigrants are selected from other islands' migrant pool and sended to island. An ACK command from server side ends this process. Mainland trigger events. MetaGraphicEngine collect statistical information during this generation of evolution.
DI	**D**isconnect **I**sland. Mainland releases resources occupied by this islands. MetaGraphicEngine generate animation for this event.
ACK	**Ack**nowledgement. Used in context with other commands.

Command Set for Mainland Island Communication

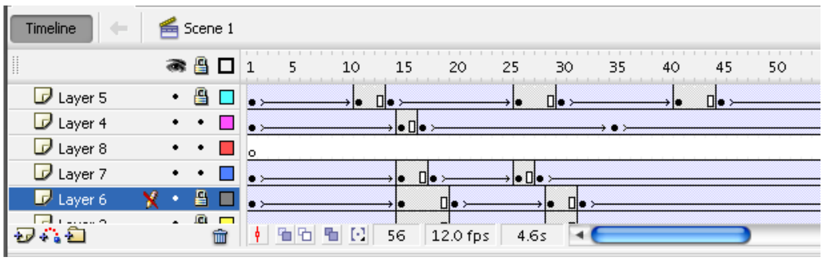
Mainland has a corresponding event system to broadcast these actions. The most usual usage is by MetaGraphicEngine, but this can be used in other ways as user wish. All users need to do is to listen to the OnChange event from MetaMainland and process the following messages, similar to windows event system:

Event	Param 1	Param 2	Meaning
ME_NEW_ISLAND	IslandInfo		New island successfuly joined
ME_NEW_GENERATION	Islandinfo		Island signals new generation.
ME_NEWMIGRANT_ISLAND	Individual	IslandInfo	Island send in new individual
ME_RMV_ISLAND	IslandInfo		Island quit
ME_NEW_CANDIDATE	Individual	IslandInfo	New candidate updated from certain island
ME_RMV_CANDIDATE	Individual		Old candidate is replace by better individual

Events Table of Mainland

4.4 2D Vector Graphic Engine

Continue to the explanation in design chapter in this area, my graphic engine is a system that maintains a collection of sprites. Each sprite has its own properties and drawing method. And most importantly, its own animation *frames stack*. This idea is borrowed from Macromedia's famous animation design package Flash. Therefore it is better to place a screenshot of work area in flash that best describe what is going on in animation frames stack of our 2D vector graphic engine.



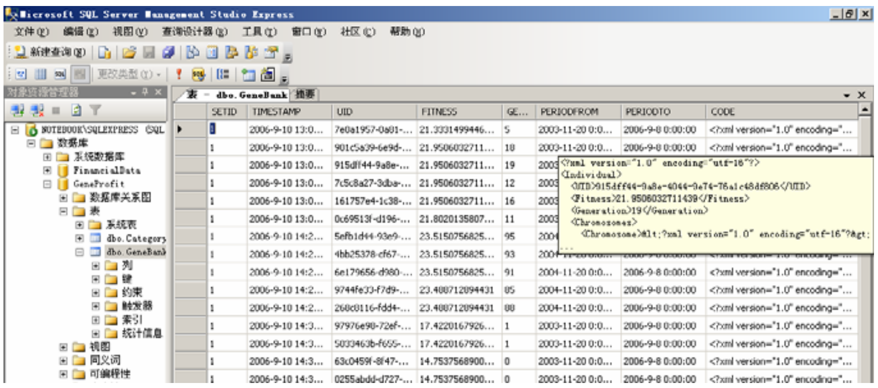
A screenshot of typical flash design work area

Our animation frames stack is worked the same way as above. Various layers of animation frames add up together generate the final effect of animation. Intermediate frames are automatically calculated, based on properties of sprite (position, size, transparency, colour, etc.)

Drawing of sprites is done by calling .Net's *GDI+* library. Almost all PC computers available on market support Windows's GDI+ hardware acceleration, therefore although our graphic engine has complicated animations rendered runtime; performance of MetaMainland is not affected.

4.5 SQL Server Express™ Database Engine

As today's computer applications continue to grow in complexity and in the amount of information they must store and manage, a stable and efficient database engine is a key ingredient for the overall success of any project. Microsoft SQL Server 2005 Express Edition (SQL Server Express) is a lightweight and "free" database engine.



SETID	TIMESTAMP	UID	FITNESS	GE...	PERIODFROM	PERIODTO	CODE
1	2006-9-10 13:0...	7e0a1957-0a01-...	21.3331499446...	5	2003-11-20 0:0...	2006-9-8 0:00:00	<?xml version="1.0" encoding="...
1	2006-9-10 13:0...	901c5a39-6e9d-...	21.9506032711...	18	2003-11-20 0:0...	2006-9-8 0:00:00	<?xml version="1.0" encoding="...
1	2006-9-10 13:0...	915dff44-9a8e-...	21.9506032711...	19	2003		
1	2006-9-10 13:0...	7c5c8a27-3dba-...	21.9506032711...	12	2003		
1	2006-9-10 13:0...	161757e4-1c38-...	21.9506032711...	16	2003		
1	2006-9-10 13:0...	0c69513f-d196-...	21.8020135807...	11	2003		
1	2006-9-10 14:2...	5efb1d44-93e9-...	23.5150756825...	95	2004		
1	2006-9-10 14:2...	4bb25378-cf67-...	23.5150756825...	93	2004		
1	2006-9-10 14:2...	6e179656-d980-...	23.5150756825...	91	2004-11-20 0:0...	2006-9-8 0:00:00	<?xml version="1.0" encoding="...
1	2006-9-10 14:2...	9744fe33-f7d9-...	23.400712094431	85	2004-11-20 0:0...	2006-9-8 0:00:00	<?xml version="1.0" encoding="...
1	2006-9-10 14:2...	260c0116-fdd4-...	23.400712094431	88	2004-11-20 0:0...	2006-9-8 0:00:00	<?xml version="1.0" encoding="...
1	2006-9-10 14:3...	97976e90-72ef-...	17.4220167926...	1	2003-11-20 0:0...	2006-9-8 0:00:00	<?xml version="1.0" encoding="...
1	2006-9-10 14:3...	5033463b-f655-...	17.4220167926...	1	2003-11-20 0:0...	2006-9-8 0:00:00	<?xml version="1.0" encoding="...
1	2006-9-10 14:3...	63c0459f-8f47-...	14.7537568900...	0	2003-11-20 0:0...	2006-9-8 0:00:00	<?xml version="1.0" encoding="...
1	2006-9-10 14:3...	0255abdd-d727-...	14.7537568900...	0	2003-11-20 0:0...	2006-9-8 0:00:00	<?xml version="1.0" encoding="...

```
<?xml version="1.0" encoding="utf-16"?>
<Individual>
  <UID>915dff44-9a8e-4044-9e74-76a1c48df806</UID>
  <Fitness>21.9506032711439</Fitness>
  <Generation>19</Generation>
  <Chromosomes>
    <Chromosome>&lt;?xml version="1.0" encoding="utf-16"?&gt;
...
```

Screen shot of SQL Server Management Studio maintaining GeneBank database

SQL database engine is accessed by provided .Net component within namespace

System.Data.SQL. A connection is established when data access is needed, then and SQLAdapter object is created with SQL query statement. In Visual Studio.Net, most the the SQL query string can be written with the assistance of wizards.

4.5.1 Category Table Definition

Field Name	Data Type
SETID	Int (Primary Key)
SYMBOL	Nchar(10)
CYCLE	Int
TC	Float
RR	Float

4.5.2 GeneBank Table Definition

Field Name	Data Type
SETID	Int (foreign Key)
TIMESTAMP	Datetime
UID	uniqueidentifier
FITNESS	Float
GENERATION	Int
PERIODFROM	Datetime
PERIODTO	Datetime
CODE	text

5 | Conclusion

5.1 Achievements

In this project, a working comprehensive software package is implemented that accomplished three main categories of challenges: *parallel distributed Genetic Programming engine*, *financial stock market prediction*, and a *visualised distributed network interface*. Many researches have been done to support rationales behind each category. I have learned a lot from each topic as well as developed own models where necessary. This project meets and beyond its initial specifications in many ways. *Firstly*, a generic Distributed Genetic Programming engine is developed, which can be applied to other problems rather than stock market for this project. *Secondly*, a financial genetic programming with various improvements to current research in this area has been achieved. This project suggests a new way to measure fitness value for financial activity is derived, and has its own approach on choosing technical indicators. *Thirdly*, a favourable graphic engine with frames animation is developed, which can be taken out as a separate component to be re-used in future project in other areas. *Finally*, this project presents a complete solution brings all these features together – GeneProfit.

5.2 Future Improvements

Due to the scale of this project, this project can be further improved from wide perspectives.

1. The distributed genetic programming engine can be evolved to a P2P system over internet. This requires a more reliable server, or may be clusters, to serve computing power from computers all over the world. Also, server side needs to have an improved security system.

2. The stock market prediction genetic programming system may be extended to other financial assets' prediction. Or in another direction, the system could be redesigned to suggest hourly investment signals.

3. The end user interface can be further developed to be more human-friendly. For example, GeneProfit could automate task that update financial data everyday and train gene strategy accordingly. GeneProfit may be extended to be able to suggest portfolio combinations depending on forecast results. Or rather, GeneProfit can incorporate other means of financial

prediction algorithm, for example, *news sensitive prediction* to overcome the disadvantage of purely based on technical analysis.

4. The visualised system can be greatly improved as well. Currently it does a great job to visualise and do basic management of distributed system. We could improve it as a visualised genetic programming solver, which allows user to create new problems visually and let the distributed genetic programming engine to solve customized problems.

References

Alba E. and Troya J.M., *A Survey of Parallel Distributed Genetic Algorithms*, John Wiley & Sons, Inc. Vol 4, No. 4, 1999

D.E. Goldberg, *Sizing populations for serial and parallel genetic algorithms*. Proceeding of the 3rd ICGA. Morgan Kaufmann, 1989

Ehrentreich, A *Corrected Version of the Santa Fe Institure Artificial Stock Market Model,* University of Halle-Wittenberg, 2002

Flynn, M.J. Some computer organizations and their effectiveness, IEEE Transactions on Coputers 21, 9, 948-960, 1972

Iba H. and Sasaki T., *Using Genetic Programming to Predict Financial Data*, the University of Tokyo, 1999

Jensen, M., *Some anomalous evidence regarding market efficiency*, Journal of Financial Economics, 6, 95-101, 1978

J.H. Holland, *Adaptation in natural and artificial systems*, University of Michigan Press, Ann Arbor, 1975

Kaboudan, M.A., *Genetic Programming Prediction of Stock Prices*, Computational Economics, 16: 207-236, 2000

K.E. Kinnear, Jr., *Advances in Genetic Programming*. MIT Press, 1994

Koza, J.R., *Genetic Programming: on the programming of computers by means of natural selection*. MIT Press, 1992

Koza, J., Goldberg, D.Fogel, D. & Riolo, R. (ed.), *Procedings of the Frst Annual Coference on Genetic Programming*, MIT Press, 1996

LeBaron, B., *Building the Santa Fe Artificial Stock market*. Working paper, http://people.brandeis.edu/blebaron/wps/sfisum.pdf. 2002

Li and Tsang, *Improving Techincal Analyusis Predictions: An Application of Genetic Programming*, Florida AI Research Symposium, USA, 1999

Luke, S., *Two Fast Tree-Creation Algorithms for Genetic Programming*, IEEE Transactions on Evolutionary Computation, 2000

Montana, D.J., *Strongly Typed Genetic Programming*, Evolutionary Computation, 3(2):199-230. Cambridge, MA: MIT Press, 1995

M. Matsumoto and T. Nishimura, *Mersenne twister: A 623-dimensionally equidistributed uniform pseudorandom number generator*, ACM Trans. on Modeling and Computer Simulations, 1998.

Mayr, N., *Animal Species and Evolution*, University Press, 1963

Miller, B.L. & Goldberg, D.E., *Genetic Alorithms, tournament selection, and the effects of Noise*, IlliGAL Report No. 95006, 1995

Pictet O.V., *Parallel Genetic Programming and its application to trading model induction*, CUI, University of Geneva, 1997

Sian, C. F., *A java based distributed approach to Genetic Programming on the Internet*, University of Birmingham, 1999

Schoder, D. and Fishchbach, K. Peer-to-Peer prospects, Commun, ACM 46, 2, 27-29, Feb. 2003

Tan K.C., Wang M.L. and Peng W., *A P2P Genetic Algorithm Environment for the Internet*, ACM 0001-0782, 2005

Potvin, Soriano and Vallee, *Generation trading rules on the stock markets with genetic programming*, Computers & Operations Research 31 (2004) 1003-1047, 2004

Tanese, R. *Distributed genetic algorithms.* In Schaffer, J., editor, Proceedings of the Third International Conference on Genetic Algorithms, pages 434--439, 1989

Appendix Technical Indicators

Median Moving Average

Moving Averages in there various forms are used to smooth data so that the underlying trend is more discernible. Since a moving average's aim is to recognize a trending market from historical prices the sensitivity of the measures will depend on the number of historical values used. For relatively few values used the moving average may itself oscillate rapidly and give many force signals to the start of trending markets. However, the more values used within the evaluation of the moving average the further into a trending cycle before it is detected and conversely when the trend finishes or changes direction the indicator will take correspondingly longer the reflect this.

> "Trends always go further than rational people expect, or even imagine. Most investors don't have the stomach for extended rallies or declines. The philosophy of not having a predetermined profit objective allows us to continue with a trend for its full duration and then some. We try very hard to avoid the pitfalls of liquidating a trade too early, even at the cost of giving back large profits..."
>
> John W.Henry Technical Trading Indicators Chapter 3

The median measure of centrality can be used with the classical moving average is order to include another filter to the historical time series. The median price is given by:

$$Meidan = \frac{High + Low}{2}$$

where High is the highest value traded on the previous day and Low is the lowest traded price. Then the x-day Median Moving Average is defined by:

$$\text{x-day Median Moving Average} = \frac{\sum_{n=0}^{x-1} m(n)}{x}$$

where m(0) is the previous days median price, p(1) is the median price on the day before and so on.

Accumulation/Distribution Indicators

These indicators measure to what degree on net as asset is being accumulated (i.e. brought) or distributed (i.e. sold) by the market as a whole.

The accumulation/distribution indicator illustrates the degree to which an asset is being accumulated or distributed by the market over a given number of periods. The indicator uses the closing price's proximity to the high or low to determine if accumulation or distribution is taking place in the market. This proximity measure is then multiplied by the volume in order to give more weight to moves with correspondingly higher volume.

Application and Trade Signal Generation

A divergence between the price action and the Accumulation/Distribution indicator can signal that a trend is nearing completion, a trends continuation and imminent breakouts from trading ranges. The actual value of this indicator is of no significance, what is significant is its change in value relative to the previous periods which can warn of a possible break-out during a trading range (falling/rising indicator), the continuation of a trend (higher highs in uptrend, or lower lows in downtrend) or a change/completion of a trend (divergence between the price action and the direction of the indicator).

Chaikin Money Flow (CMF)

Chaikin Money Flow (CMF) is a volume weighted average of Accumulation/Distribution over the specified period, which is usually taken to be 21 days. The CMF offers a volume weighted indicator on the following two principles:
ˆ The nearer the close is to the high the more accumulation is taking place.
ˆ The nearer the close is to the low the more distribution is taking place.

Interpretation and Trade Signal Generation

A sell signal is generated in positive overbought territory when higher highs diverge into a lower high and the indicator continues to decrease. Conversely, a buy signal in generated in negative oversold territory when lower lows diverge into a high low and the indicator continues to increase.

The CMF indicator can be used as a confirmation signal after a breakout of a trading range. When a market breaks higher then the breakout is confirmed if the CMF moves into positive territory and continues to get stronger. Conversely, if the market down after a trading range then the breakout if confirmed if the CMF move into negative territory and continues to weaken.

The CMF indicator is evaluated for the following steps:

1. Evaluate the Volume Weighted Accumulation/Distribution over each of the days within the period considered for the calculation. The Volume Weighted Accumulation/Distribution on each day is given by:

$$\frac{(Close - Low) - (high - Close)}{High - Low} * Volume$$

2. Sum the Volume Weighted Accumulation/Distribution over the period and the divide the result by the sum of the volume over the period.

Oscillators

Within this section we deal with Oscillators such as the money flow index, stochastics, momentum and rate of change (ROC) indicators. Oscillators are generally used to identify short term price reversal points as opposed to longer term trending dynamics.

Momentum and Rate of Change (ROC) Indicators

The Momentum and Rate of Change (ROC) will get similar numerical values and can be used together or interchangeable within a system. The momentum indicator as the name suggests is the velocity with which the price is rising or failing, and hence will reflect how aggressively the asset is being purchased or sold. Whereas the ROC indicator approximately represents percentage change of the asset over the considered period. Interpretation Extended values and/or turning points of the momentum are good indicators of oversold or overbought conditions (respectively).

Stochastics

The Stochastics Oscillator compares the closing price with the price over a given period. The rational is that if the closing price is near to the highest traded price over a given number of previous sessions then the stock is bullish. Similarly, if the closing price in near lowest traded price over a given number of previous sessions then the stock is bearish.

Evaluation the fast %K Stochastic

The (fast) Stochastic %K depends on the following variable:
1. Periods - the number of previous time periods used over which the closing price is compared. Now

the formulae for the Stochastic %K is:

$$\left(\frac{LastClose - LowestLowOverPeriod}{HighestHighOverPeriod - LowestLowOverPeriod} \right) \times 100$$

where the ”lowest low” (respec. ”highest high”) is the highest (respec. lowest) close of the asset over the period under consideration. Since the ”Last close”, will lie between the hghest high and lowest low this indicator will lie between 0 and 100.

RSI

The Relative Strength Index (RSI) is one of the most popular overbought/oversold (OB/OS) indicators. The RSI was developed in 1978 by Welles Wilder.

The RSI is basically an internal strength index which is adjusted on a daily basis by the amount by which the market rose or fell. A high RSI occurs when the market has been rallying sharply and a low RSI occurs when the market has been selling off sharply. The RSI values range from zero to 100.

One characteristic of the RSI is that it moves slower when it reaches very overbought or oversold conditions, and then snaps back very quickly when the market enters even a mild correction. This brings the RSI back to more neutral levels and indicates that the price trend may be able to resume.

The general formula for the RSI is:
RSI = 100 - 100/(1 + RS)

where RS = (Avg of n-day up closes)/(Avg of n-day down closes)

RSIV
The Volume RSI Index (VRSI) is based on the same concepts as the Relative Strength Index (RSI), one of the most popular overbought/oversold (OB/OS) indicators.

Unlike the RSI, the VRSI uses volume in place of price. The RSI definition and sample will provide a better understanding of this indicator.

VR(N) = SMA(MAX(VOL-REF(VOL,1),0),N,1)/SMA(ABS(VOL-REF(VOL,1)),N,1)*100

OBV

On Balance Volume (OBV) relates price to volume, and tries to capture the buying and selling pressure in the market. It assumes that when a security closes up for the day, the number of shares transacted represent buying power. Conversely, the amount of volume on a down day represents selling power.

Therefore, if the price ends up for the day on 10,000 shares traded, OBV's value will increase by 10,000. Should the price decrease on 25,000 shares, OBV's value will decrease by 25,000.