Building Trading Systems Using Automatic Code Generation

Michael R. Bryant, Ph.D.

Disclaimer

HYPOTHETICAL OR SIMULATED PERFORMANCE RESULTS HAVE CERTAIN INHERENT LIMITATIONS. UNLIKE AN ACTUAL PERFORMANCE RECORD, SIMULATED RESULTS DO NOT REPRESENT ACTUAL TRADING. ALSO, SINCE THE TRADES HAVE NOT ACTUALLY BEEN EXECUTED, THE RESULTS MAY HAVE UNDER- OR OVER-COMPENSATED FOR THE IMPACT, IF ANY, OF CERTAIN MARKET FACTORS, SUCH AS LACK OF LIQUIDITY. SIMULATED TRADING PROGRAMS IN GENERAL ARE ALSO SUBJECT TO THE FACT THAT THEY ARE DESIGNED WITH THE BENEFIT OF HINDSIGHT. NO REPRESENTATION IS BEING MADE THAT ANY ACCOUNT WILL OR IS LIKELY TO ACHIEVE PROFITS OR LOSSES SIMILAR TO THOSE SHOWN.

Introduction

One of the biggest trends in retail trading over the past decade has been the increase in the popularity of automated trading. In this type of trading, also known as automated order execution, buy and sell signals generated by a trading system are automatically executed by a platform connected to the trader's brokerage account. This allows for hands-free trading, which enables faster execution, fewer errors, and the ability to trade shorter time frames with higher-frequency strategies.

As more and more traders have moved to automated trading, the interest in systematic trading strategies has increased. While some traders develop their own trading strategies, many traders lack the programming skills necessary to implement their ideas. Other traders lack the specific knowledge of technical trading methods or the experience required to design a viable strategy. Even for traders with the necessary skills for developing trading systems, the considerable time and effort required to develop a good strategy is often a deterrent.

A recently developed solution to this problem is the use of computer algorithms to automatically generate trading system code. The goal of this approach is to automate many of the steps in the traditional process of developing trading systems. In the traditional, manual approach to strategy development, the trader selects elements of the trading strategy based on prior experience and knowledge of technical indicators, entry and exit order types, and strategy design. Commonly, a strategy is based on a market *hypothesis*; that is, an idea of how the market works. A viable trading strategy is typically developed through a long trial-and-error process involving numerous iterations, revisions, and testing until acceptable results are achieved.

This traditional process of developing trading systems is extremely time consuming and involves systematically eliminating many ideas that simply don't work. Also, all traders have biases about how the markets work, and these biases can influence the system development process. In some cases, these biases may be helpful, but they can also limit the possible systems the trader might consider. Rather than starting with a biased view and a limited set of rules, an automatic code generator starts with a large set of rules and searches in an unbiased manner for the combinations that work while quickly eliminating those that don't.

This paper presents an overview of automatic code generation methods for building trading systems. Both simple and complex methods are discussed. A simple ad hoc method is presented that can be implemented in TradeStation's EasyLanguage scripting language to find basic price pattern-based strategies. A more complex approach based on genetic programming is also discussed.

Automatically generating trading systems is an attractive idea. However, there are several drawbacks as well. For one thing, rigorous approaches, such as those based on genetic programming, are complex and difficult to implement. Also, automatic code generation generally relies on historical simulation, which means it's an optimization process. As such, the risk of over-fitting must be addressed. These caveats are also discussed.

The Basic Approach

The basic algorithm for building trading systems using automatic code generation is depicted below in Fig. 1. It starts with a method for combining different elements of the trading strategy. These elements may include various technical indicators, such as moving averages, stochastics, and so on; different types of entry and exit orders; and logical conditions for entering and exiting the market.

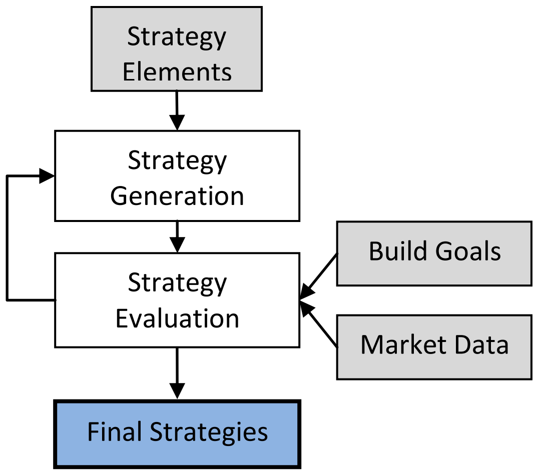


Figure 1. Basic algorithm for automated strategy building.

After the different elements are combined into a coherent strategy, it can be evaluated on the market or markets of interest. This requires market data – prices, volume, open interest, etc. – for each market. Generally speaking, you would also have a set of build goals to help rank or score each strategy. Examples of build goals include various performance measures, such as the net profit, drawdown, percentage of winners, profit factor, and so on. These could be stated as minimum requirements, such as a profit factor of at least 2.0, or as objectives to maximize, such as maximizing the net profit.

The strategy generation and evaluation steps are repeated until the termination criteria are met. The termination criteria could be as simple as creating a predetermined number of different strategies, or the process might be stopped after no further improvement in the build goals is achieved. Typically, an optimization algorithm is used to guide the strategies towards ones that meet the build goals. The final strategies are the ones with the highest rank or score based on the build goals. You could either take the single best strategy or save some number (or all) of the strategies, ranked by build goals. If there are multiple build goals, a weighted average can be used to form a single metric.

This is the most basic view of automatic system building. A more detailed description will be provided below in the section on genetic programming. This description also ignores the important problem of over-fitting, in which the strategy is fit so closely to the market data that's used during the build process that the strategy doesn't perform well in the future when applied to new data. This issue is also addressed below.

Theoretical Basis of Automatic Code Generation

As described above, building a trading system using automatic code generation is essentially an optimization problem. The combination of strategy elements that maximizes the build goals is taken as the final strategy. Some traders would object that trading systems should be constructed based on a hypothesis of market behavior or action. If you have a good hypothesis for how the markets work, a strategy can be built around that hypothesis and tested. If it works, it supports the hypothesis and justifies trading the strategy.

In fact, the approach described here is not fundamentally different than that. Each candidate strategy constructed during the build process, as depicted in Fig. 1, is essentially a hypothesis that is either supported or refuted by the evaluation. If out-of-sample testing is used, the final strategies can be further supported or refuted by the out-of-sample results.

Another way to view automatic code generation is as a problem of *statistical inference*. The price data can be thought of as a combination of “signal” and “noise”. The signal is the tradable part of the data, and the noise is everything else. In this context, the strategy building process is a nonlinear curve-fitting problem where the objective is finding strategies that fit the signal while ignoring the noise and avoiding over-fitting. At the same time, market data is often non-stationary: the statistical properties change over time. A successful strategy is therefore one that fits the stationary elements of the market signal with adequate degrees-of-freedom to avoid over-fitting. Although discussed in more detail below, out-of-sample testing is generally used to verify that the strategies are not over-fit to the market.

Pattern System Code Generator for TradeStation

This section describes an ad hoc approach to automatic code generation in which a trading system for TradeStation automatically generates other, pattern-based trading systems for TradeStation. The *AutoSystemGen* system searches for a set of trading rules, along with the associated parameter values, that meet a specified set of performance requirements.

Depending on the performance requirements, it might find several or even dozens of trading systems that meet the requirements. It then writes the EasyLanguage code for each system to a file. For illustrative purposes, the rules for the generated systems are restricted to price patterns. In principle, this technique could be expanded to automatically generate systems drawing from a wide variety of entry and exit techniques applicable to almost any market.

Price Pattern Rules

While almost any type of indicator or trading logic could be included in the trading system generator described here, to keep things fairly simple, the rules of the generated systems will be restricted to price patterns. Each entry rule of a generated trading system will have the following form:

```
P1[N1]  Ineq P2[N2]
```

where P1 and P2 are prices (open, high, low, or close), N1 and N2 are the number of bars to look back (e.g., Close[2] is the close two bars ago), and Ineq is an inequality operator, either <= or >=. Examples of rules include the following:

```
Close <= Close[2]
Low[2] <= High[10]
High[3] >= Close[4]
```

and so on. P1, P2, N1, N2, and Ineq are all variables to be determined by the system generation process.

N1 and N2 will be restricted to the range 0 – 20. Also, the number of rules, NRules, will be a variable with values ranging from one to 10. A trade entry will be triggered if all the rules are true. In that case, the entry will be taken at the open of the next bar. The trade direction will be set beforehand, so that the system will be generating systems that are either all long or all short trades. To obtain trading logic for both long and short trades, the system can be run twice, once for long trades and the second time for short trades.

Trades will be exited at the market after a fixed number of bars, NX, which will range from one to 20.

Finding the Rules

The key to this process is finding candidate trading systems. A system can consist of between one and 10 rules of the form shown above. Trades are entered at market if all the rules are true, and trades are exited a certain number of bars later. If this were coded as a traditional TradeStation system, with a maximum of 10 rules, there would be 52 inputs. This would make for a cumbersome strategy.

Instead, a different approach will be used. At each step of the optimization, the values for each variable (P1, P2, N1, N2, Ineq, NRules, and NX) will be chosen randomly. A different set of values of P1, P2, N1, N2, and Ineq will be selected for each rule, for a total of NRules sets of values.

Each step of the optimization will generate a different trading system as the variables are randomly selected. If the performance results of the system meet the requirements entered by the user, the generated system will be written to a file in EasyLanguage code.

Putting it All Together

The code for the AutoSystemGen system and its related functions is available at Breakout Futures (http://www.breakoutfutures.com/) on the Free Downloads page.

The first input to the strategy is called OptStep. To run the system, OptStep should be optimized in TradeStation by varying it from 1 to some large number, such as 10,000, in steps of 1. This will cause AutoSystemGen to generate, for example, 10,000 different trading systems. The ones that meet the specified performance criteria are written to the file shown as an input to the WriteSystem function (e.g., C:\AutoSysGen-Output1.txt). The performance criteria are specified via the system inputs (reqNetProfit, reqMaxDD, etc.).

Most of the hard work is performed by the functions that the system calls. The function GetPatVars randomly selects the values for the variables that determine the trading rules. To determine whether or not a trade entry will occur on the next bar, the price pattern rules are evaluated by the function EvalPattern. Finally, if the system meets the performance criteria, the corresponding EasyLanguage code is generated and written out to a text file by the function WriteSystem.

Example

As an example, consider the 30-year treasury bond futures market (symbol @US.P in TradeStation 8). AutoSystemGen was optimized over the past 20 years of T-bond prices with the OptStep input incremented from 1 to 10000. This means the system evaluated 10,000 different trading systems. The optimization was run twice, once for long trades and once for short trades. The following performance requirements were used: net profit of at least $30,000, worst-case drawdown no more than $7500, at least 200 trades, percent profitable of at least 50%, and profit factor of at least 1.2. On a dual core computer running Vista, it took approximately 10 minutes to run each optimization (10,000 systems per optimization).

The systems generated by this process are shown below. These are the systems written to the file AutoSysGen-Output1.txt by the WriteSystem function. The first ones are the long-only systems, followed by a short-only system (the only one that met the performance criteria).

```
System 2332, @US.P, 9/17/2007 12:23:00, Long Trades
 Net Profit = 53562.50, Max DD = -7381.25, Num Trades = 250, Percent
Wins = 56.80, Prof factor = 1.631
{ System code starts here... }
Var: EntNext (false);
```

```
EntNext = Open[2] >= Low[16] and
    Low[9] >= Low[3] and
    Close[14] <= Low[6] and
    High[1] >= Low[3];
If EntNext then
   Buy next bar at market;
If BarsSinceEntry = 2 then
   Sell next bar at market;
{ End system code }

System 5771, @US.P, 9/17/2007 12:27:00, Long Trades
 Net Profit = 42145.00, Max DD = -5733.75, Num Trades = 207, Percent
Wins = 57.00, Prof factor = 1.631
{ System code starts here... }
Var: EntNext (false);
EntNext = High[7] >= Low[19] and
    Close[20] >= Close[5] and
    High[18] >= Low[2] and
    High[2] <= Open[6];
If EntNext then
   Buy next bar at market;
If BarsSinceEntry = 2 then
   Sell next bar at market;
{ End system code }

System 7622, @US.P, 9/17/2007 12:29:00, Long Trades
 Net Profit = 59348.75, Max DD = -7222.50, Num Trades = 208, Percent
Wins = 60.58, Prof factor = 1.924
{ System code starts here... }
Var: EntNext (false);
EntNext = Low[2] <= High[9] and
    Open[11] >= Open[18] and
    Close[17] <= Low[18];
If EntNext then
   Buy next bar at market;
If BarsSinceEntry = 3 then
   Sell next bar at market;
{ End system code }

System 7718, @US.P, 9/17/2007 12:29:00, Long Trades
 Net Profit = 35526.25, Max DD = -6936.25, Num Trades = 292, Percent
Wins = 56.85, Prof factor = 1.418
{ System code starts here... }
Var: EntNext (false);
EntNext = Close[3] >= High[19] and
    High[8] >= Low[9] and
    High[6] <= Open[10] and
    Low[16] <= High[3];
If EntNext then
   Buy next bar at market;
If BarsSinceEntry = 1 then
   Sell next bar at market;
{ End system code }
```

```
System 6160, @US.P, 9/17/2007 12:42:00, Short Trades
 Net Profit = 31277.50, Max DD = -6846.25, Num Trades = 369, Percent
Wins = 51.76, Prof factor = 1.297
{ System code starts here... }
Var: EntNext (false);
EntNext = High[9] >= Low[6] and
    Close[15] >= High[8] and
    High[7] <= Low[20] and
    High[6] >= High[7];
If EntNext then
   Sell short next bar at market;
If BarsSinceEntry = 1 then
   Buy to cover next bar at market;
{ End system code }
```

The listing for each system includes the system number (corresponding to the OptStep input), market symbol, current date, and whether the system is long-only or short-only. The next line contains a few summary performance statistics to help in evaluating each system. Finally, the system code is shown. To evaluate the systems in TradeStation, the code between the two comment lines ({ …}) can be copied and pasted into a strategy in TradeStation, then run in the chart window.

For example, the first system shown above (#2332) was copied to TradeStation and saved as a strategy. When inserted into the @US.P chart, the following equity curve was obtained:

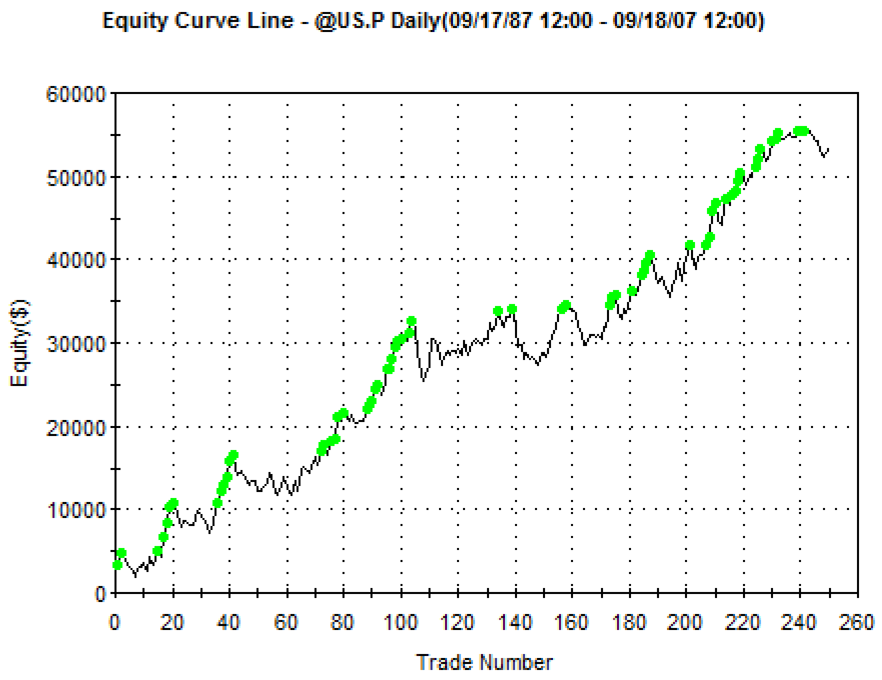


Figure 2. Long-only system for T-bonds, last 20 years, with $15 per trade deducted for trading costs, generated by system AutoSystemGen.

The last system in the output file is for a short-only system (#6160). When saved in TradeStation as a strategy and applied to the same T-bond chart, the following equity curve was produced:

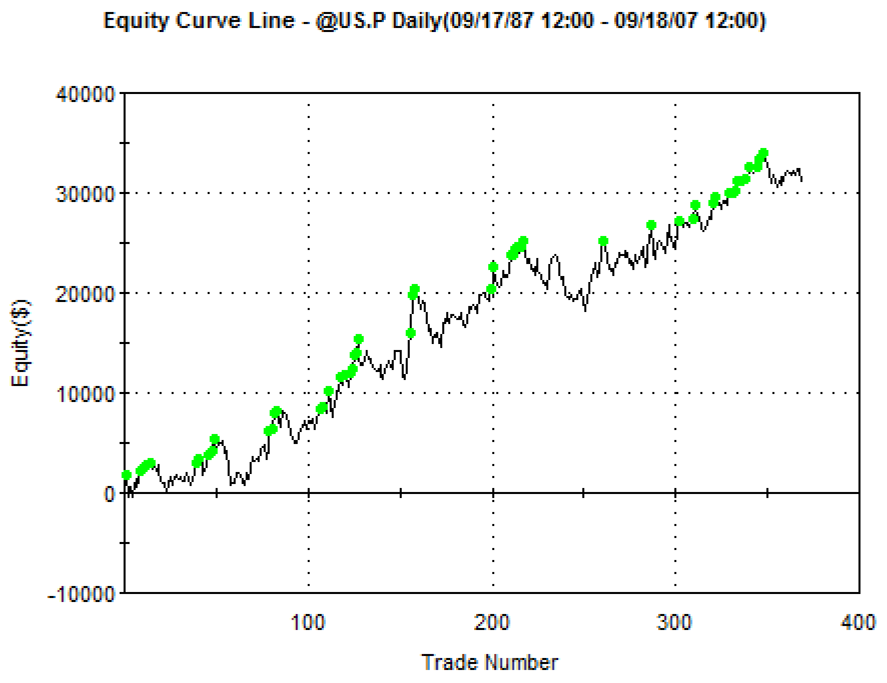


Figure 3. Short-only system for T-bonds, last 20 years, with $15 per trade deducted for trading costs, generated by system AutoSystemGen.

With a small amount of additional effort, the two systems could be combined into a single system that generates both long and short trades in the same system.

It's notable that the random selection of strategy elements is as effective as it is. There's no optimization per se in this approach. Each strategy is generated randomly and independently of all others. By contrast, if an optimization method were used, the results from one step in the strategy generation process would be used to guide the next step, and the results would generally converge towards the build goals over successive steps. Even lacking this, the random generation process is still fairly effective.

Even though an optimization algorithm is not used to generate the strategies, there's still a risk of over-fitting. Because the final strategies are selected from a large number of candidate strategies (10,000 in this example), it's possible that the results could be due to random chance. To test for this, the final strategies should be evaluated on data not used during the strategy generation. This is called *out-of-sample testing*. If the out-of-sample results are not good, the strategy is suspect.

Genetic Programming for Automatic Code Generation

The ad hoc approach described in the previous section is simple but has two limitations: (1) the randomly generated strategies don't converge towards the build goals, and (2) the template of the pattern system is difficult to generalize to more complex strategies. This suggests a more sophisticated approach is needed.

A method for automatic code generation that addresses both these concerns is called *genetic programming* (GP),[1] which belongs to a class of techniques called evolutionary algorithms. Evolutionary algorithms and GP in particular were developed by researchers in artificial intelligence based on the biological concepts of reproduction and evolution. A GP algorithm "evolves" a population of trading strategies from an initial population of randomly generated members. Members of the population compete against each other based on their "fitness." The fitter members are selected as "parents" to produce a new member of the population, which replaces a weaker (less fit) member.

Two parents are combined using a technique called crossover, which mimics genetic crossover in biological reproduction. In crossover, part of one parent's genome is combined with part of the other parent's genome to produce the child genome. For trading system generation, genomes can represent the trading rules and order logic of the strategy.

Other members of the population are produced via mutation, is which one member of the population is selected to be modified by randomly changing parts of its genome. Typically, a majority (e.g., 90%) of new members of the population are produced via crossover, with the remaining members produced via mutation.

Over successive generations of reproduction, the overall fitness of the population tends to increase. The process is stopped after some number of generations or when the fitness stops increasing. The solution is generally taken as the fittest member of the resulting population.

The initial GP population might have as few as 50 members or as many as 1000 or more. A typical build process might progress over anywhere from 10 to 100 generations or more. The number of strategies constructed and evaluated during the build process is equal to the size of the population multiplied by the number of generations.

In the context of building trading strategies, GP enables the synthesis of strategies given only a high level set of performance goals. The GP process does the rest. This approach has several significant benefits, including:

- Reduces the need for knowledge of technical indicators and strategy design. The GP algorithm selects the individual trading rules, indicators, and other elements of the strategy for you.
- The rule construction process allows for considerable complexity, including nonlinear trading rules.
- The GP process eliminates the most labor intensive and tedious elements of the traditional strategy development process; namely, coming up with a new trading idea, programming it, verifying the code, testing the strategy, modifying the code, and repeating. This is all done automatically in GP.
- The GP process is unbiased. Whereas most traders have developed biases for or against specific indicators and/or trading logic, GP is guided only by what works.
- By incorporating proper trading rule semantics, the GP process can be designed to produce logically correct trading rules and error-free code.
- The GP process often produces results that are not only unique but non-obvious. In many cases, these *hidden gems* would be nearly impossible to find any other way.
- By automating the build process, the time required to develop a viable strategy can be reduced from weeks or months to a matter of minutes in some cases, depending on the length of the input price data file and other build settings.

Genetic programming has been successfully used in a variety of fields, including signal and image processing, process control, bioinformatics, data modeling, programming code generation, computer games, and economic modeling; see, for example Poli et al.[2] An overview of using GP in finance is provided by Chen.[3] Colin[4] was one of the first to explain how to use GP for optimizing combinations of rules for a trading strategy.

Various academic studies have demonstrated the benefits of GP in trading. For example, Karjalainen[5] found that price pattern trading rules evolved using GP for S&P 500 futures provided an advantage over buy-and-hold returns in out-of-sample testing. Similarly, Potvin et al.[6] found that rules generated through a GP process for individual stocks outperformed buy-and-hold in out-of-sample testing during falling and sideways markets. Kaucic[7] combined a genetic algorithm with other learning methods to generate simple trading rules for the S&P 500 index and found positive results compared to buy-and-hold on out-of-sample testing.

References

1. J. Koza. Genetic Programming. The MIT Press, Cambridge, MA. 1992.

2. R. Poli, W. B. Langdon, and N. F. McPhee. A field guide to genetic programming. Published via http://lulu.com and freely available at http://www.gp-field-guide.org.uk, 2008. (With contributions from J. R. Koza).

3. Shu-Heng Chen (Editor). Genetic Algorithms and Genetic Programming in Computational Finance. Kluwer Academic Publishers, Norwell, MA. 2002.
4. A. Colin. Genetic algorithms for financial modeling, Trading on the Edge. 1994, Pages 165-168. John Wiley & Sons, Inc. New York.

5. Risto Karjalainen. Evolving technical trading rules for S&P 500 futures, Advanced Trading Rules, 2002, Pages 345-366. Elsevier Science, Oxford, UK.

6. Jean-Yves Potvin, Patrick Soriano, Maxime Vallee. Generating trading rules on the stock markets with genetic programming. Computers & Operations Research, Volume 31, Issue 7, June 2004, Pages 1033-1047.

7. Massimiliano Kaucic. Investment using evolutionary learning methods and technical rules. European Journal of Operational Research, Volume 207, Issue 3, 16 December 2010, Pages 1717-1727.

A Build Algorithm Using Genetic Programming

Expanding on the build algorithm presented previously (see Fig. 1), a more detailed algorithm is illustrated below in Fig. 4 based on genetic programming. The gray-shaded boxes represent the input data, which includes the price data for the market(s) of interest, the indicators and order types in the so-called *build set*, and the options and performance criteria (build goals) selected by the user.

The algorithm starts with the Strategy Generation step. An initial population of trading strategies is randomly developed from the available technical indicators and rule types in the build set. Any options that the user has selected, such as exiting all positions at end-of-day, are applied at this point. Each strategy is then evaluated over the price data for the market(s) of interest, and a fitness value is assigned based on a weighted average of the build goals specified by the user. For example, you might select net profit and drawdown as the two performance metrics and weight each one equally. The fitness would then be the average of the (normalized) net profit and drawdown.

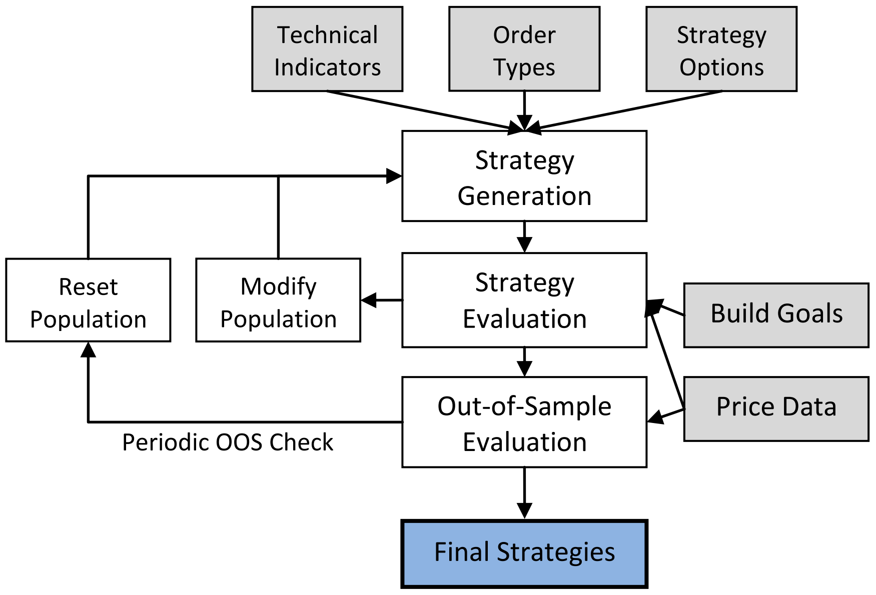


Figure 4. Build algorithm for automatic code generation with genetic programming.

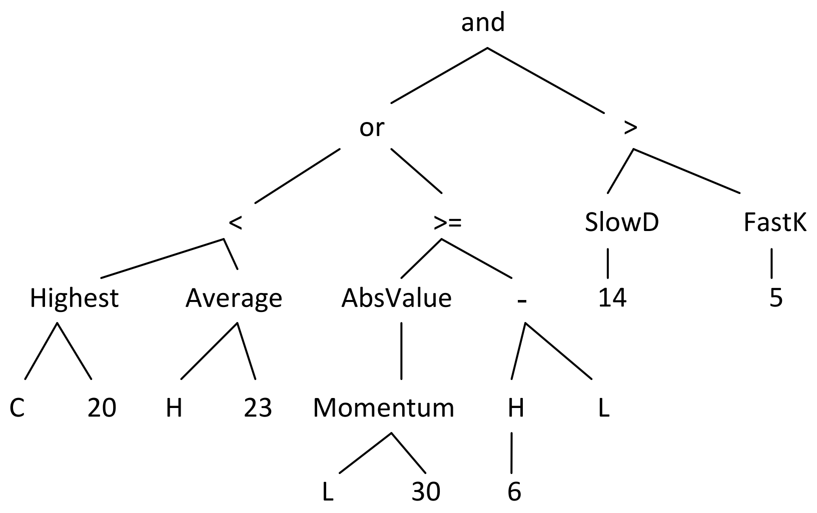
To generate new members of the population, members of the current population are selected at random, and the fitter ones are chosen as parents for crossover and mutation. A less fit member is selected at random to be replaced by the new member. The process is repeated until as many new members have been created as there are members in the current population. This step represents one generation.

Out-of-sample results are computed on a segment of the data not used to calculate the fitness. An optional check can be made periodically to make sure the out-of-sample results are positive. If the results are not above a threshold chosen by the user, the process can be reset, causing the population to be re-initialized and the generation count to be reset to zero. After the specified number of generations has been successfully completed, the top strategies are taken as the ones with the highest fitness.

Entry Conditions

The GP process can be used to evolve two essential strategy elements simultaneously: entry conditions and orders for entry and exit. The entry conditions are typically represented as tree structures, as shown below in Fig. 5.

The tree structure enables the generation of entry conditions with considerable complexity. Each node in the tree has between zero and three inputs, each of which leads to further branching. The tree is constructed recursively starting at the top with a logical operator (and, or, >, <, etc.) and proceeding to technical indicator functions, prices, and, finally, constants, such as indicator lengths. Each branch is terminated with a node that has no inputs.



((Highest(C, 20) < Average(H, 23)) or (AbsValue(Momentum(L, 30)) >= H[6] – L)) and (SlowD(14) > FastK(5))

Figure 5. Entry condition example, showing tree structure and corresponding EasyLanguage code.

The crossover operator of the GP process replaces a subtree in one parent with a subtree from the other parent. For example, the subtree on the right of Fig. 5, starting with ">" (i.e., SlowD(14) > FastK(5)), might be replaced with a different subtree from another member of the population. Mutation changes individual nodes in the tree. For example, the "Average" node might be replaced with "Lowest" so that the subtree Average(H, 23) becomes Lowest(H, 23).

An entry condition can be evolved separately for long and short trades or one entry condition can be logically reversed for the other side of the market. Each entry condition is a logical statement; it evaluates to either true or false. A value of true means the entry condition is satisfied for that market side (long or short), which is necessary for the entry order to be placed.

In order to generate meaningful entry conditions, both *syntactic* and *semantic* rules need to be applied when building the conditions. Syntactic rules ensure that each node containing a function satisfies the input requirements for the function. For example, the Momentum function requires a price as the first input and a length as the second input. Semantic rules ensure that comparisons between different nodes are meaningful. For example, it makes sense to compare the Highest(C, 20) to a moving average since both functions return a price. However, it would not be meaningful to compare the closing price to the time of day or to compare a stochastic, which has a value between 0 and 100, to a moving average of price. The semantic rules enforce these requirements.

Order Types

The key to evolving entry and exit orders using genetic programming is representing the different types of orders in a generalized fashion. For example, stop and limit entry prices can be represented as follows:

```
EntryPrice = PriceValue +/- Fr * PriceDiff
```

where PriceValue is anything that represents a price, such as the O, H, L, C, Highest(price, N), Average(price, N), and so on; Fr is a constant multiplier; and PriceDiff is anything that evaluates to the difference between two prices, such as the average true range (ATR), the difference between two moving averages, etc.

Using this formula, the following would be valid long stop entry prices:

```
EntryPrice = Average(C, 10) + 3.5 * AbsValue(C[5] - H[14])

EntryPrice = H + 2 * AbsValue(Average(C, 20) - Lowest(H, 15))
```

These could also be short limit entries since short limit entries are also above the market and therefore use a “+” sign to add the price difference to the price value. Target and protective stop exits can be constructed in much the same way as stop and limit entry orders.

Applying crossover and mutation to trading orders of this type involves replacing parts of the orders and/ or randomly selecting new parameter values to create new orders.

Trading Strategy Structure

While genetic programming is capable of generating trading strategies with considerable variety, it’s necessary to start with a generalized structure for the strategies to follow. The strategy structure shown below in pseudo-code provides a framework for building strategies based on entry conditions and order types like those discussed above:

```
Inputs: N1, N2, N3, …

LongEntryCondition = …

ShortEntryCondition = …

If position is flat and LongEntryCondition is true then
      Long entry order…
      Initialize long exit orders as necessary…

If position is flat and ShortEntryCondition is true then
      Short entry order…
      Initialize short exit orders as necessary…

If position is long then
      Long exit order 1…
      Long exit order 2…
      …

If position is short then
      Short exit order 1…
      Short exit order 2…
      …

[Optional end-of-day exit]
```

The strategies start with the list of inputs. An input is provided for any indicator parameter, price pattern look-back length, and any parameters required by the entry and exit orders, such as the look-back length for the ATR.

The LongEntryCondition and ShortEntryCondition variables are the true/false entry conditions evolved by the genetic programming process, such as shown in Fig. 5. A long entry order is placed if the long entry condition is true, provided the position is currently flat (out of the market). Likewise, a short entry order is placed if the short entry condition is true, provided the position is currently flat.

Only one type of entry order is allowed for each side of the market (long/short), although they can be different for each side. When an entry order is placed, one or more variables for the exit orders may be initialized within the entry order code block.

The statements for the exit orders follow the entry orders. One or more exit orders may be used. It's helpful to ensure that each strategy has an exit-at-a-loss and an exit-at-a-profit. This prevents trades from remaining open indefinitely.

An optional end-of-day exit can be used to ensure intraday strategies exit at the day's close.

Example

To illustrate using genetic programming for automatic code generation in strategy building, the program Adaptrade Builder (http://www.adaptrade.com/Builder/) was run on daily bars of a stock index futures market for a small population and a limited number of generations. The performance metrics chosen to guide the process were the net profit, number of trades, correlation coefficient, statistical significance, and the return/drawdown ratio. Specific targets were set for the number of trades and the return/drawdown ratio. The other selected metrics were maximized. The fitness function was a weighted average of terms for each metric.

The population size was set to 100, and all members of the population were saved. The in-sample/out-of-sample division of data was set to 80% in-sample and 20% out-of-sample (OOS), with the OOS period following the in-sample period. The build process was run over a total of 10 generations. To illustrate how the results evolved during the build, the OOS net profit was recorded after the initial population was generated and after five and 10 generations. Fig. 6 demonstrates that the number of members of the population with OOS net profits of at least $1,000 increased after five and 10 generations.

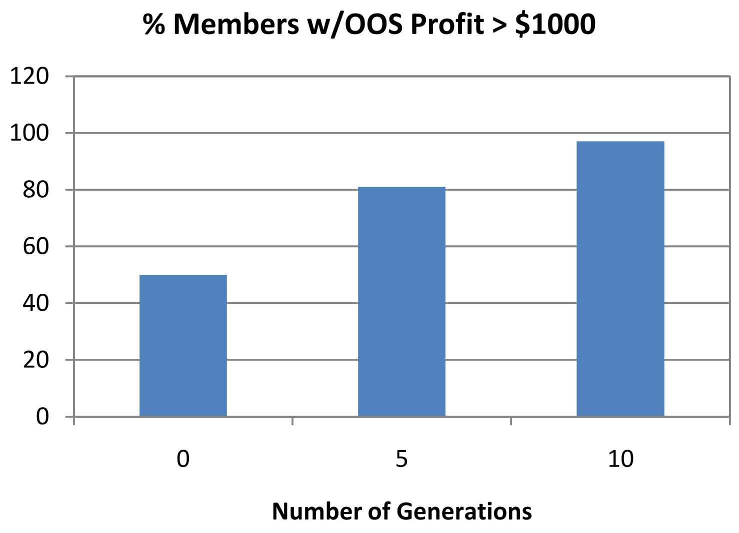


Figure 6. Percentage of population members with out-of-sample net profit greater than $1,000.

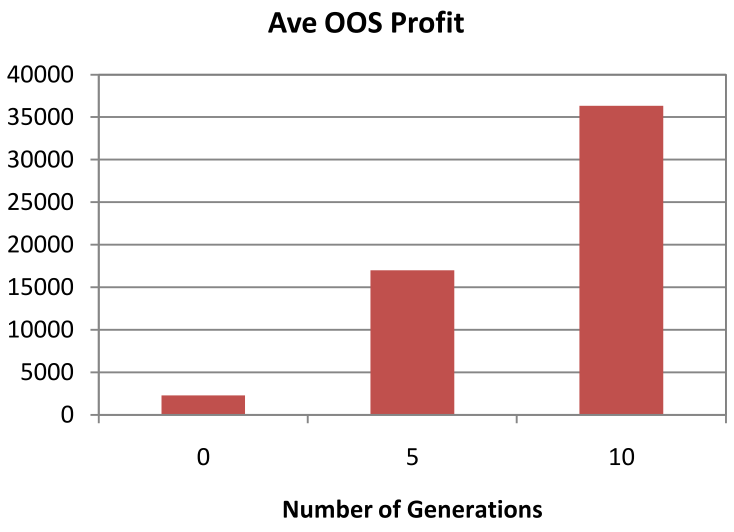


Figure 7. Average out-of-sample net profit of population members.

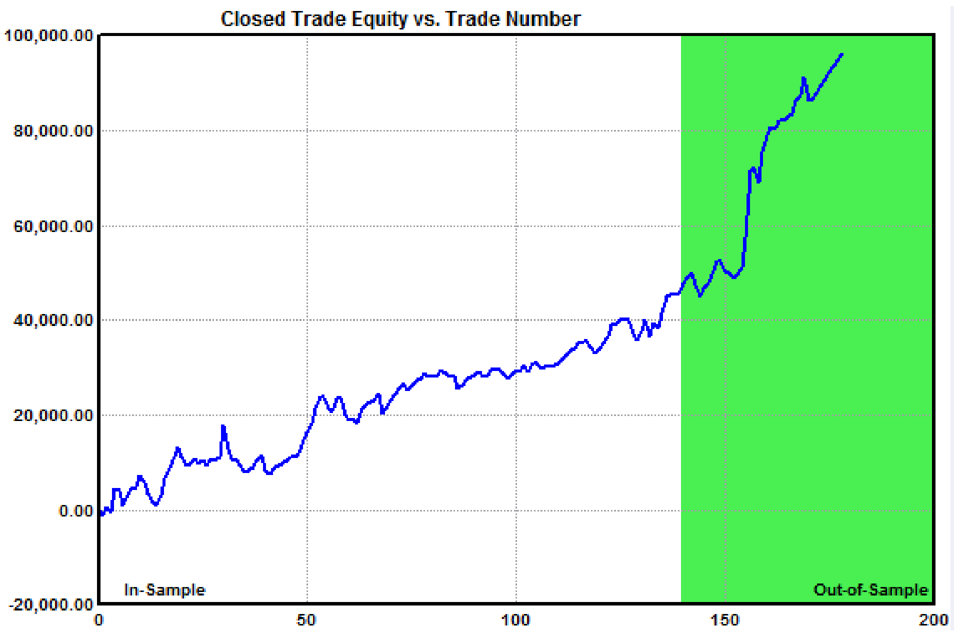


Figure 8. Closed trade equity curve after 10 generations.

Similarly, the average OOS net profit of the population increased after five and 10 generations, as shown in Fig. 7. Note that these results are for the OOS net profit. By definition, the out-of-sample data is not used in the build, so OOS results are unbiased; they don't benefit from hindsight. This implies that the GP process not only tends to improve the in-sample results over successive generations, which is a direct effect of the GP algorithm, but the OOS results also tend to improve as the strategies are evolved. This indicates a high quality build.

The equity curve for one of the top strategies is shown above in Fig. 8 after 10 generations, with the out-of-sample equity curve shown in green.

Finally, the EasyLanguage (TradeStation) code for the corresponding strategy is listed below.

```
{
 EasyLanguage Strategy Code for TradeStation
 Population member: 46

 Created by: Adaptrade Builder version 1.1.0.0
 Created:    10/19/2010 2:19:52 PM

 TradeStation code for TS 6 or newer

 Price File:  C:\TestData.txt
 Build Dates:
}

{ Strategy inputs }
Inputs: NL1 (74),
        NL2 (20),
        NL3 (85),
        NBarEnL1 (59),
        NATREnL (84),
        EntFrL (3.8189),
        NATRTargL (57),
        TargFrL (1.6168),
        NBarExL (100),
        NBarEnS1 (40),
        NBarEnS2 (49),
        NBarEnS3 (7),
        EntFrS (0.6971),
        NBarExS (6),
        NATRTrailS (33),
        ATRFrTrailS (1.4126),
        TrailPctS (50.0000);

{ Variables for average true range for entry and exit orders }
Var:    ATREnL (0),
        ATRTargL (0),
        ATRTrailS (0);

{ Variables for money management and/or trailing stop exit orders }
Var:    SStop (0),
        NewSStop (0),
        STrailOn (false);

{ Variables for entry conditions }
Var:    EntCondL (false),
        EntCondS (false);

{ Average true range }
ATREnL = AvgTrueRange(NATREnL);
ATRTargL = AvgTrueRange(NATRTargL);
ATRTrailS = AvgTrueRange(NATRTrailS);

{ Entry conditions }
EntCondL = (Highest(Volume, NL1) >= Lowest(Volume, NL2)) or (Volume < Average(Volume,
NL3));

EntCondS = true;

{ Entry orders }
If MarketPosition = 0 and EntCondL then begin
   Buy next bar at XAverage(L, NBarEnL1) + EntFrL * ATREnL stop;
end;

If MarketPosition = 0 and EntCondS then begin
   Sell short next bar at Highest(H, NBarEnS1) - EntFrS * AbsValue(Lowest(L,
NBarEnS2) - Lowest(H, NBarEnS3)) stop;
```

```
   STrailOn = false;
   SStop = Power(10, 10);
end;

{ Exit orders, long trades }
If MarketPosition > 0 then begin

   If BarsSinceEntry >= NBarExL then
      Sell next bar at market;

   Sell next bar at EntryPrice + TargFrL * ATRTargL limit;
end;

{ Exit orders, short trades }
If MarketPosition < 0 then begin

   If EntryPrice - C > ATRFrTrailS * ATRTrailS then
      STrailOn = true;

   If STrailOn then begin
      NewSStop = EntryPrice - TrailPctS * (EntryPrice - C)/100.;
      SStop = MinList(SStop, NewSStop);
   end;

   If BarsSinceEntry >= NBarExS then
      Buy to cover next bar at market;

   If STrailOn then
      Buy to cover next bar at SStop stop;
end;
```

Until recently, most applications of genetic programming to trading strategy generation have been academic studies based on limited rule sets, overly simple entry and exit logic, and custom-written code, making the results unsuitable for most traders. At the same time, most available software that implements GP for market trading has either been targeted to professional traders and priced accordingly or is very complicated to set up and use. Adaptrade Builder was designed to make GP simple to use for any trader, individual or professional, who has a basic understanding of strategy trading and the TradeStation platform. More information on Builder can be found at www.Adaptrade.com.

Over-fitting

Building trading systems via automatic code generation is a type of optimization. Most systematic traders are probably familiar with parameter optimization, in which the inputs to a strategy are optimized. Unlike parameter optimization, automatic code generation optimizes the strategy's trading logic. Nonetheless, the risk of over-optimization, or "over-fitting", is also a concern for automatic code generation, just as it is for parameter optimization.

Typically, optimization is performed over one segment of data, called the optimization or in-sample segment, and tested on different data, called the test or out-of-sample segment. Over-fitting refers to the problem of optimizing a strategy so that it fits the in-sample segment well but doesn't work well on any other data, including the out-of-sample data.

Poor out-of-sample performance is usually caused by one of several factors. One important factor is the so-called *number of degrees-of-freedom* in the in-sample segment. The number of degrees-of-freedom, which is equal to the number of trades minus the number of rules and conditions of the strategy, determines how tightly the strategy fits the data. Provided inputs are added for each parameter in the strategy, the number of strategy inputs can be used as a proxy for the number of rules and conditions. For example, if a strategy has 100 trades and 10 inputs, it has 90 degrees-of-freedom. The more degrees-of-freedom, the less likely it is that the strategy will be over-fit to the market and the more likely it is that it will have good out-of-sample performance.

The number of degrees-of-freedom can be increased during the build process by including the number of trades and/or the number of strategy inputs as build goals. Assuming the fitness metric is a weighted average of the build goals, all other things being equal, increasing the weighting for the number of trades will result in strategies with more trades and therefore more degrees-of-freedom. Likewise, increasing the weighting for the (negative) number of inputs will result in strategies with fewer inputs, which will also increase the number of degrees-of-freedom.

Another option is to include the *statistical significance* as a build goal. The statistical significance can be calculated by applying the Student's t test to the average trade. This will measure the probability that the average trade is greater than zero. The t test is based on the number of degrees-of-freedom but is a more complete measure of whether a strategy is over-fit than the number of degrees-of-freedom alone. One way, then, to improve out-of-sample performance is to include the significance in the fitness function, which will tend to generate strategies that have a high statistical significance.

Another important factor affecting out-of-sample performance is the variety of market conditions in the in-sample segment. Generally speaking, it's better to optimize over data that includes a wide variety of market conditions, such as up trending and down trending markets, periods of consolidation, high and low volatility, etc. The more variety in the in-sample segment, the more likely it is that the strategy will perform well on other data, including out-of-sample data and in real-time trading. While the future never exactly duplicates the past, provided the future (or out-of-sample data) is similar enough to at least part of the in-sample segment, the strategy should perform well on new data.

The value of optimizing over a variety of market conditions presumes that good performance is achieved over each part of the in-sample segment. One way to measure this is with the correlation coefficient of the equity curve, which measures how closely the equity curve approximates a straight line. If the equity curve is a straight line, it implies that the performance is uniform over all segments of the data. Obviously, this is desirable if the goal is to achieve good performance over as many different types of market conditions as possible. The correlation coefficient for the strategies generated via automatic code generation can be increased by including the correlation coefficient as a build goal and weighting it as part of the fitness function.

Unfortunately, there will be cases where even with a high significance, a correlation coefficient close to 1, and a wide variety of market conditions in the in-sample segment, the out-of-sample performance will be poor. This can happen for several reasons. First, even a simple strategy with few parameters can in some cases fit the *noise* rather than the *signal*. By definition, noise is any part of the market data that does not contribute to profitable trading signals. Secondly, the market dynamics on which the strategy logic is based (i.e., the signal) may have changed in the out-of-sample segment enough to negatively impact performance. This is sometimes due to a fundamental change in the market, such as the switch from floor-based to electronic trading. However, more subtle changes, often related to the trading patterns of market participants, are also possible, particularly for shorter-term trading.

If this appears to be the problem, the solution may be as simple as rebuilding the strategy with new trading logic. Using a tool such as Adaptrade Builder makes this much easier than if a manual approach to trading strategy development were used. Another possible solution is to include the most recent data in the optimization segment and test it out-of-sample by tracking the performance in real-time. In most cases, a strategy that has a large number of trades, a high significance value and good performance on the in-sample segment will continue to perform well for some period of time post-optimization.

For additional information, please visit www.Adaptrade.com.

Index