7.3 Genetic Programming

Ming Lu

2022-04-02

The genetic programming (GP) heuristic belongs to a class of evolutionary strategies characterized by being computer-intensive, stepwise constructors of subsets of feasible solutions, using operators that model sexual reproduction and governed by the Darwinian principle of survival of the fittest. The general structure of this heuristic is as follows: Beginning with a randomly generated set of individuals (i.e. feasible solutions for the problem), which constitutes an initial population, allow them to evolve over many successive generations under a given fitness (or performance) criterion, and by application of operators that mimic procreation in nature, such as, reproduction, crossover and mutation, until no significantly better individuals are produced. The optimal solutions are then found among the best fitted individuals of the last generation. Note that, in the context of genetic programming, the formal notation (S ,τ, F) for an optimization problem is interpreted as: S the set of individuals, τ the constraints to guarantee well-formed individuals, and F the fitness function. The individuals are programs which are encoded as trees; that is, hierarchical structures where nodes at one level may represent variables, constants or other terms, which are arguments for functions represented by nodes at a preceding level. Applying an operator means to remove some nodes or adding new ones to form new trees (or programs) of varying size and shape. Remark 7.1 From now on we use indistinctly the words “program” and “tree” to refer to individuals generated by a genetic programming heuristic, due to the natural identification between the object and its graphical representation.

Remark 7.2 An important feature about the genetic operations is that all perform local perturbations to create new programs. Therefore, in the general scheme of genetic programming outlined above, if for each generation we consecutively select the best fitted programs for procreating new offsprings (incurring in what’s known as over-selection), then we can at most guarantee that at the end of the process the best programs obtained correspond to local optimum solutions.2 Therefore we need to make some escape-moves at certain generations to aim for a global optimum. A escape-move in this context can be realized by selecting some of the worse fitted programs, and combine them among themselves or with best fitted programs, for procreating new offsprings. Genetic programming was conceived by John R. Koza (1992) as an extension of the genetic algorithm of John John Holland (1975). The main distinction between both evolutionary schemes is that in Holland’s paradigm individuals are coded as linear bit strings, usually implemented as an array of fixed length, whereas in Koza’s genetic programming the individuals are coded as non-linear structures (e.g. trees) that do not have the size limitation in the design, although of course are limited by the memory capabilities of the computer. The representation of individuals in a genetic heuristic as non-linear structures or programs is more suitable for optimization problems that deals with defining and analyzing financial decision rules, since a decision procedure is easily representable as a tree.

# 7.3.1 The Basics of Genetic Programming

Given an optimization problem that we want to solve with the genetic programming paradigm, the first step is to associate to two sets of nodes: : set of terminal nodes which represent constants (e.g. real, integer or boolean) or variables that take values from input data relevant to . : set of function nodes which represent functions relevant to . In general, the functions in the function set may include: - Arithmetic operators: ; - Relational operators: ; - Boolean operators: AND, OR, NOT; - Conditional operators: IF-THEN-ELSE; - Real functions and any domain-specific function appropriate for the problem. Terminal nodes have no descendant; function nodes have as many descendants as their number of parameters. Each node transmit to its unique predecessor its value (for terminal nodes), or the result of evaluating the function on the values transmitted by its descendants. It is customary to link nodes under this “descendant of” relation in bottom-up style, so a program is represented as a tree with terminal nodes at the bottom, function nodes at intermediate levels and a single function node (the root) at the top. The result of evaluating this tree, which gives the output of the program , is the value returned by the root after recursively evaluating every node of the tree from bottom to top. Example 7.3 Consider the following numerical equation,

with and two numerical constants, and the usual arithmetic functions, and the order relation, which can be turned into a boolean function by a common trick best illustrated by an example: true. (We are sure the reader can generalized this idea to turn any relation into a function.) This equation is a program that outputs true if the inequality holds, or false if the inequality does not holds. It can be encoded with and as the tree in Fig. 7.1. The boolean value of expression (7.4) is obtained by parsing the tree bottom-up.

Initial individuals. The initial trees conforming the first population are produce by randomly selecting elements from the terminal or function sets, , observing the following recursive rules: (1) Begin by selecting at random an element of to be the root of the tree. (3) For an already defined node of the tree, if its label is a function , then define as many descendants nodes as arguments that takes and of the appropriate type from ; if the label is a terminal element (from ), then this node has no descendants. (3) Repeat rule 2 until a predefined depth is reached or until all nodes at a same level are terminal with the proviso that all nodes at the maximum specified depth must be terminal. (The depth of a tree is the longest path from the root to a terminal node.)

There are two variants to this construction which Koza called Full and Grow. In the Full method terminal elements can only be selected for nodes at the last level, which is located at the given depth. Thus all trees are fully branching trees. In the Grow method terminals can be selected at any level, and hence trees has branches of varying lengths (bounded by the predefined length). A sensible generative method is a mix of Full and Grow applied at varying depths ranging from 2 to the maximum established depth. For example, build an equal number of trees with the Full and the Grow method for each depth , up to maximum depth. This last strategy is what Koza calls the “ramped half-and-half” method.

Closure property. The provisos in rule (2) for constructing trees refers to a closure restriction stated by Koza in his genetic programming paradigm, where it is required that all functions (and by extension all programs) be well defined and closed for any combination of the appropriate number of arguments that are inputted to it. This guarantees that we only build valid programs, and hence reduces the search space by avoiding to work with malformed programs, which will eventually be discarded by the natural selection process anyway. Fitness. The fitness of a program is a measure that, from the genetic point of view, should reflect its probabilities of reproduction; that is, its expected surviving time and number of successive descendants. As regards the underlying optimization problem, the fitness of a program should reflect some rate of approximation of its -value to the -value of the currently best solution, or a predefined target solution. This is known as raw fitness. Selection. Programs are selected according to their fitness value. But, by Remark 7.2, we must allow some not-so-well fitted programs to be selected also, to ensure some degree of variety and have some hope of reaching a global optimum. We must avoid over-selecting programs. There are various selection strategies proposed in the literature that overcomes the over-selection and guarantees some variety in the population up to certain degree. One effective method is to sort programs from best to worst, with respect to some normalization of their raw fitness that would equally distribute them, and apply a random selection with a probability distribution function skewed towards best fitted programs; so that most of the time the best are chosen but sometimes a not-so-good program is also chosen. This is known as ranking selection, a technique attributed to Baker (1985). Note that this probabilistic selection is in fact a sufficient condition to have almost surely convergence to a global optimum (i.e. global convergence with probability 1). Operations. The three most frequently used genetic operators are: reproduction, crossover and mutation. The first step in applying any of the operations is to select programs, which should be done following some strategy ensuring variety, and as outlined above.

Reproduction: it consist on selecting a program and transfer a copy of it, without any modifications, to the next generation. Crossover: it consists on randomly selecting two programs (the parents) to generate two new individuals (the offsprings) by a process mimicking sexual recombination.5 From a copy of each parent randomly select a node (the crossover point), and then exchange the subtrees hanging from the chosen nodes, to obtain two new trees with pieces of material from both parents. After the operation we apply some survival criteria (based on fitness) to remove one parent and one child. This is important for keeping the population size bounded. Figure 7.3 shows a full picture of a crossover operation. Mutation: it consists on choosing at random some node in the tree, remove this node together with the subtree below it, and then randomly generate a new subtree in its place. Mutation is a unary operator, for it applies to a single program. The purpose of mutation is to restore diversity in a population that may be converging too rapidly, and hence, very likely to a local optimum.6 However, in practice, when crossover is applied hardly two trees that exchange parts will produce a child equal to one of them. Thus, in general, the algorithm should perform few mutations and rely more on crossover and reproduction. There are other less popular operators, which include permutation, editing, encapsulating, and destroying, but we shall not deal with these and refer the reader to (Koza 1992) for a full description and extensive analysis. Stopping criterion. A genetic programming algorithm can go on forever, and hence we must force it to stop. The stopping criterion is usually set as a predetermined number of generations or conditioned to the verification of some problem-specific success test, as for example, that the programs do not improve their fitness through some generations. Setting up the control parameters. Depending on the problem we wish to apply the GP scheme, one should adjust the parameters that controls the algorithm accordingly. The most important parameters to calibrate are: the size of the populations, the number of generations, the number of iterations, the frequency of applying each genetic operator, and the size of the programs. Koza (1992, Chap. 5) provides a set of default values that have been widely validated through the extensive practice of genetic programming. However, other authors refrain from giving a set of exact values and rather provide some general guidelines (Reeves and Rowe, 2003, Chap. 2). Our advise is to take Koza’s generic parameter values as a starter, and proceed to adjust them through several trials considering the particular features of the problem and computational resources (e.g., more often than not, one is forced to reduce the size of the population from the recommended 500 individuals, due to machine memory limitations). For the financial application of GP presented in the next section, we present our own table of parameters values obtained following the previously stated advise.

# 7.3.2 Finding Profitable Trading Rules with Genetic Programming

A widely studied application of genetic programming in finance is the automatic generation of technical trading rules for a given financial market and analysis of their profitability. The classic work on this subject is by Allen and Karjalainen (1999), who applied a GP heuristic to generate trading rules from Technical Analysis adapted to the daily prices of the S&P 500 index from 1928 to 1995. Considering transaction costs, these researchers found that the rules produced by their algorithm failed to make consistent excess returns over a buy-and-hold strategy in out-of-sample test periods. Following the work of Allen and Karjalainen (from now on AK), other researchers have tried similar GP heuristics to produce technical trading rules for different financial instruments. Neely et al. (1997) produced trading rules for foreign exchange markets, and reported positive excess returns when trading with six exchange rates over the period 1981–1995; Wang (2000) applied GP to generate rules to trade in S&P 500 futures markets alone and to trade in spot and futures markets simultaneously, finding that his GP trading rules did not beat buy-and-hold in both cases; Roger (2002) investigated GP generated rules to trade with particular stocks (as opposed to an index as did AK) from the Paris Stock Exchange, and found that in 9 out of 10 experiments the trading rules outperformed buy-and-hold in out-of-sample test periods, and after deducting a 0.25% per one-way transaction. There are many others experiences with GP generation of technical trading rules with varying conclusions.7 Those resulting in negative excess returns over buy-andhold (as AK or Wang), can be interpreted as being in line with market efficiency. On the other hand, those experiments resulting in GP generated rules that consistently outperform buy-and-hold, should be taken first as an indication of some deficiency in the model, before being considered as a contest to the efficiency of markets hypothesis. For example, profitable GP generated rules could have been obtained because the penalty for trading (represented in the transaction costs) has been set too low; or due to over-fitting the data, which occurs when programs that are trained in a given time period of the data, produce other programs by means of some of the genetic operators, and these new programs are trained in the same time period (and with same data) as their parents, whereby knowledge about the data is transmitted through generations. Another possible cause of flaw in the model is the data snooping bias from the selection of specific time periods for training or testing; e.g., periods where data seems more (or less) volatile could influence the results, and by selecting these periods the researcher could be unwillingly forcing the conclusions. These observations motivate the following provisos that should be attached to the definition of a GP algorithm for finding profitable technical trading rules: (1) Transaction costs should be part of the fitness function, and trials be made with different values, within realistically reasonable quantities. (2) Consider separate time periods for generating the trading rules and testing their profitability. As a matter of fact, in the process of generating programs in the GP heuristic, there should be a time period for training and a different time period for validation (or selecting). In general, the time series data should be split into training, validation and test periods, in order to construct trading rules that generalize beyond the training data. (3) The data snooping bias can be avoided by applying the rolling forward experimental set up. This consist on defining two or more partially overlapping sequences of training–validation–test time periods, as in the example shown in Table 7.1, and make a trial of the GP algorithm on each of these sequences.

The GP algorithm by AK (and Wang’s too) work with these provisos. AK tested their rules with one-way transaction costs of 0.1, 0.25 and 0.5%, to find that only with the rather low fee of 0.1% a few trading rules gave positive excess returns. They also applied different periods of time for training and validation, and used the rolling forward method. We have conducted our own experiment using GP to find rules for trading in the Spanish Stock Market index (IBEX) and the Dow Jones Industrial (DJI) index (Llorente-Lopez and Arratia 2012). We followed the AK methodology, and based on our own experience we shall provide a general description of the experimental set up, that hopefully can guide the reader into conducting her own GP experiment for this financial application. We first observe that the problem of finding profitable trading rules from Technical Analysis has the following global parameters: • the essential features of the financial time series that is used as information for trading in the given financial asset; for example, daily or monthly price history, volume history, returns, or the time series of any other quantifiable fact of the asset; • the trading strategy that is carry out. These parameters condition and determine the building blocks to use for constructing trading rules and, by extension, the genetic programming scheme to learn these trading rules. Indeed, we argue that as trading rules are in general functions that take as input financial data (i.e., financial time series) and output a sequence of signals that trigger some trading action, it is the trading strategy that determines the nature of these signals. For example, if the strategy is to always buy and sell the same fixed quantity of the asset, then the signals emitted by the trading rules should be either “buy” or “sell”, that is a binary valued output. But for a more sophisticated strategy that buys and sells varying proportions of the asset, then the trading rules should be functions that output signals from a possibly infinite set of real values (say, in [0, 1]) to quantify different degrees of buy or sell. Therefore, for the sake of precision, when applying a genetic programming scheme to learn technical trading rules one must indicate the underlying financial asset and the trading strategy, and be conscious that results are relative to these parameters. In this section a genetic programming solution is given for: Finding technical trading rules for common stocks, using as information the daily price history and applying the simple, yet common, trading strategy of buying and selling the same fixed proportion of the stock the day following the trading signal

Thus, all trading rules give binary valued signals (false for sell or true for buy), and the trading strategy consist on specifying the position to take the following day in the market (long or out), for a given trading rule signal and current position. More specifically: if we are currently long in the market, and the trading rule signals “sell”, then we sell and move out of the market; if the current position is out, and the trading rule signals “buy”, then we buy and move in to the market; in the other two cases, the current position is preserved. Each time we trade an equal amount of stock (e.g., one unit). This trading strategy is fully describe by Table 7.2.

The general experimental set up. Following provisos (2) and (3) for the experimental set up of a GP algorithm for finding rules for trading with certain financial instrument, we must train the rules in a period different from the validation (where the best rules are selected), and then test their profitability in another period. To avoid the data snooping bias we applied the rolling forward method, which consists on repeating the experiment with different overlapping sequences of training-valuation-test periods. The number of these sequences and the length of each period conforming each sequence is determined by the amount of data available. In the case of DJI and IBEX we have plenty of data, and study these time series from 1995 to 2011 . This allow us to define five time sequences, each divided in a training period of 4 years long, a validation period of 2 years, and a testing period of 2 years. Beginning in 1995 we can set the sequences overlapping for 6 years (see back Table 7.1, where the first 3 sequences are presented). For each data sequence, we made 50 runs of the GP algorithm (the parameter “number of iterations” is set to 50). This give us 50 “best rules” to be tested for profitability. In each run, the three data periods are used in the following manner: In each generation of rules, the best fitted rule against the training period is selected and its fitness is evaluated against the validation period. If this rule has better validation fitness than the previous candidate to “best rule”, then this is the new “best rule”. The process is repeated until the stopping criterion is met. The resulting “best rule” is tested for profitability against the test period. We denote the result of applying the fitness function fit to a rule in the training period by , and in the valuation period by . Individuals are trading rules. Since each trading rule is a boolean function that takes as input a given price history, and returns true to signal a “buy”, or false to signal a “sell”, it can be easily represented by the parsing tree of a boolean expression. Take, for example, a triple moving average crossover rule for periods of 9,20 and 50 market days (cf. Sect. 6.1.3). This rule can be expressed by the following boolean function

We use to denote the average of the price history considered over the recent past days, and up to current time ; i.e., . Figure presents a tree representation of the triple moving average crossover rule , 50)-periods.

Formally, the trading rules are built from the following set of terminals and functions of different types and varying number of parameters:

Terminals: constants of type real, integer and boolean. Reals are fixed to a finite set of values around 1 to represent a set of normalize prices; integers range from 0 to 252 , to represent working days up to a year; boolean constants have value true or false. Variables: price that takes as its value the stock price of current day (a real number).

IF-THEN-ELSE, another boolean function, but of three boolean parameters; if the boolean valued of the first parameter is true then return the value of second parameter, otherwise return value of third parameter. , are real type functions of two real parameters. Finally, the domain-specific real functions with one parameter of type integer: and , interpreted respectively as moving average, maximum and minimum of the series of prices over the past days, and the price lagged by days from the current day.

Now, in order to comply with the closure property of any genetic program and at the same time with our intended binary valued trading signals, we impose the following restrictions in the construction of trading rules: - only allow boolean type functions as the root of the tree (this ensures that the output of the rule is true or false); - every function node must have as many descendants as the number of its parameters, and each descendant is a terminal or function node of a type matching the type of the corresponding parameter in the function.

Initial population. We apply the ramped half-and-half method described in the general initialization procedure in Sect. 7.3.1. Each trading rule is limited to at most 80 nodes (terminals and functions), which sets a bound to the depth of the associated tree around 7. The size of each population is limited to 200 . Fitness. The fitness function measures excess return over a buy-and-hold strategy, taking into account transaction costs. To define it, we shall assume that the cost of a one-way transaction is always a fixed positive fraction (or percentage) of the current price. Recall from Problem that the return for a single trade, say to buy at date and sell at date , considering transaction costs at the rate of , is given by

The log return (with transaction costs) is

Let be the number of trading days, and be the number of trades made by the rule within the trading period . This is a sequence of pairs of “buy” followed by “sell” signals (or an empty sequence in case the rule never signals to go in the market).

Now in order to compute the total continuously compounded return for the trading rule generating this sequence of pairs of signals, we can mark and save explicitly the entry dates and the exit dates, and at the end of the period only sum over all corresponding terms given by Eq. (7.5). Alternatively, for a less intensive use of computer memory, we can compute, on each day , the daily return and check if the rule is signaling to be in or out, in which case multiply by 1 or 0 respectively. We take this latter approach, which is formalize as follows. Consider the indicator function:

Then the continuously compounded return for the trading rule throughout the trading period is

and the return for the buy-and-hold strategy (buy the first day, sell the last day) is

The excess return (or fitness) for the trading rule is given by

Selection. In selecting programs (potential trading rules) we avoid over-selection by using a rank-based selection. This is done as follows. The current population of programs is sorted, based on their raw fitness value (fit ), from best fitted (which has rank 1) to worst fitted (which has rank ). Let (resp. ) be the fitness value of the best (resp. worst) fitted program. Then the rank-based fitness value for the program of rank is

Note that and . Using as an adjusted fitness, only for the purposes of selection, programs which have close raw fitness are now separated and have similar chance of being randomly selected, while programs with very high fitness are bound for the value, reducing the chance of being the most selected. Genetic operations. We apply crossover with high probability. For that the parameter crossover is set to , and define the variable that sets the number of programs that should be obtained by crossover from the current population. Crossover is done as explained in the general scheme: two trading rules are recombine by joining their tree representations at some compatible node, and two offsprings are thus generated. An example is shown in Fig. 7.3. To keep the size of the population bounded by , we eliminate one parent and one offspring applying the following survival criterion: the best fitted offspring replaces one of the parents, randomly selected using a probability distribution skewed towards the worst fitted parent. In this way we keep the best fitted offspring, plus one of the parents that is not always the best fitted.

Mutation is applied to the surviving offspring from the crossover operation (following a general advise given by Koza (1992)). This is to guarantee certain degree of diversity, in case parents are too similar; although this is a rare event, and hence this should be done with low probability. We set the probability of mutation . We remark that the algorithm by AK does crossover always; i.e. the authors have set . Their crossover produces only one offspring, which replaces one of the parents in the manner we have described. Also AK does not perform any mutation. Stopping criterion. This is set to a limited number of generations (in our experiments we used 100), or if the best rule does not improves for a number of generations (we used 50 ).

Table summarizes the values used for the parameters of our GP algorithm for finding trading rules. A note on implementation. The GP heuristic makes an intensive use of memory space, so it is important to use a compact representation of individuals with fast access to the information. The article by Keith and Martin (1994) studies various ways of implementing the nodes and storing the trees (programs) generated by a GP, efficiently with respect to memory space and execution time. Although the analysis they do is geared for a C++ platform, it can be adapted to other programming language