Genetic Programming with Memory For Financial Trading

Alexandros Agapitos[1(✉)], Anthony Brabazon[2], and Michael O'Neill[2]

[1] School of Computer Science and Informatics,
University College Dublin, Dublin, Ireland
alexandros.agapitos@ucd.ie
[2] School of Business, University College Dublin, Dublin, Ireland

Abstract. A memory-enabled program representation in strongly-typed Genetic Programming (GP) is compared against the standard representation in a number of financial time-series modelling tasks. The paper first presents a survey of GP systems that utilise memory. Thereafter, a number of simulations show that memory-enabled programs generalise better than their standard counterparts in most datasets of this problem domain.

1 Introduction

The problem of *sequence learning* is to discover the underlying function of a dynamic system, in order to be able either to produce the next step in a sequence produced by the system (sequence prediction), or correctly classify a sequence produced by the system (sequence classification). Sequence learning has numerous applications, e.g. stock market time-series prediction, protein structure prediction, speech and handwriting recognition. Genetic Programming (GP) [1] has been widely applied to sequence learning tasks. The vast majority of systems employ a sliding time-window of size W, where at each time-step t, values $\{s_{t-1}, \ldots, s_{t-W}\}$ from a sequence $\{s_i\}_{i=1}^{N}$ are used to populate an input vector of explanatory variables used to predict sequence value s_t. This input vector configuration biases towards the evolution of some sort of an *autoregressive* model.

Stateful Genetic Programming (GP) deals with the evolution of computer programs that utilise memory. Puzzlingly, very little work in sequence learning by means of stateful GP is found in the literature. The evolution of stateful sequence-processing programs is closely mirrored by the ongoing investigation of how to best evolve programs with memory or feedback loops in GP.

Any stateless program is essentially a function that maps input to output using the sliding time-window described earlier. In general, it is difficult to decide on the optimal window size. For tasks with long-term dependencies a large window is necessary. A solution might be to use a combination of several different-sized windows, but this is only applicable when the exact time-dependencies of the task are known a priori. Furthermore, fixed time-windows are inadequate when a task has variable time-dependencies. In one form or the other, stateless programs have an important drawback: only a fixed number of

G. Squillero and P. Burelli (Eds.): EvoApplications 2016, Part I, LNCS 9597, pp. 19–34, 2016.
DOI: 10.1007/978-3-319-31204-0_2

previous sequence values (the "context-length") can be taken into account in prediction/classification tasks. On the other hand, in principle, a stateful program can overcome the limited context-length by using memory that enables a dynamic program behaviour based on memory state.

We present an empirical study of time-series classification using stateful GP; the classification task is to discriminate between *go-long* or *go-short* signals in a financial trading system. The rest of the paper is structured as follows: Sect. 2 surveys previous work on the evolution of programs that use memory. Section 3 states the scope of current research. Section 4 introduces two stateful GP systems, and presents the experiment design. Section 5 analyses the results, whereas Sect. 6 concludes and discusses routes for future research.

2 Evolution of Programs with Memory

Previous work confirms that GP can automatically create programs that explicitly use memory. This section reviews previous work using a general taxonomy between *scalar memory* and *indexed memory*. In addition, a third subsection reviews previous work that investigated another form of memory via the concept of *recurrence* (i.e. implemented using time-delay feedback loops in a program).

In GP with scalar memory, the program is given access to a number of variables used as leaves in an expression-tree. In this way the same variable is accessed irrespective of program input. In GP with indexed memory, special primitives are available in the function-set which enable a parameterised access of memory elements based on an *index* value. Depending on the evaluated index, different variables can be accessed; thus different program input may trigger the access/manipulation of different variables. In summary, scalar memory allows for a "static" access to variables, whereas indexed memory allows for a "dynamic" access.

2.1 Scalar Memory

The use of memory in evolvable computer programs dates back to the work of Cramer [2] in 1985. Koza [3] (1992) used global registers that could be manipulated with storage operators. He presented an example where a single variable is used to maintain a running total during execution of a loop. Additionally, in [4], Koza used a small number of variables in a protein classification problem. Montana [5] (1994) presents two examples where GP is provided with local variables. Huelsbergen [6] (1996) used simple scalar memory in his machine-code GP system. Kirshenbaum (2000) presented work on the evolution of programs that use statically-scoped local variables [7]. Conrads et al. [8] (1998) applied linear GP with memory registers to speech time-series classification. Agapitos and Lucas [9] (2007) applied object-oriented GP to the task of evolving statistical operations on samples of data. Agapitos et al. [10] (2007) additionally evolved stateful control programs for a simulated car-racing task. They also experimented with the use of multi-objective optimisation to encourage the effective use of memory variables in car-racing controllers [11]. Poli et al. [12] (2008) introduced the concept *memory-with-memory* based on soft assignments of variables to values.

2.2 Indexed Memory

Teller (1994) introduced *indexed memory* [13]. Its implementation requires the use of an indexed array of memory cells, and the operations of `read` and `write` to enable memory access and manipulation. Using indexed memory Teller evolved programs that solve the problem of pushing blocks up against the boundaries of a world represented as a toroidal grid [14]. Jannink [15] (1994) studied the evolution of programs that use indexed memory to generate random numbers. Teller (1995) evolved programs for image classification [16] and face recognition [17]. In addition, Teller performed acoustic time-series classification [18].

Andre [19] (1994) tackled the problem of controlling an agent whose task is to collect all of the gold scattered in a five-by-five toroidal grid. Brave [20] (1996) studied a similar problem of an agent that explores the world and is required to produce a plan for reaching every arbitrary location in the world from every arbitrary starting point. Haynes and Wainwright [21] (1995) applied GP to evolve control programs for agents which have to survive in a simulated world containing mines. The agent's memory is a dynamically allocated linked-list.

The evolution of Abstract Data Types is investigated in the works of Langdon [22] (1996) and Bruce [23] (1996), who independently evolved stacks, queues, priority queues and linked-lists. Nordin and Banzhaf [24] (1996) used linear GP for sound compression. Spector and Luke [25] (1996) used indexed memory to implement *culture* in a GP run. Koza [26] (1999) generalised indexed memory into an *Automatically Defined Store*. O'Neill [27] (2002) investigated the use of indexed memory in Grammatical Evolution to evolve agents for the Tartarus world, and also evolve caching algorithms. Agapitos et al. [28] (2008) evolved classifiers that use indexed memory for EEG signal classification. Agapitos et al. [29] (2011) used indexed memory to represent models of racing tracks during the evolution of car-racing controllers. Finally, Agapitos et al. [30] (2011) used GP with indexed memory to evolve decision-trees for financial time-series classification.

2.3 Recurrent GP

Another form of memory in GP may be implemented via the notion of *recurrency* in terms of time-delay feedback loops. In sequence processing, time-delay feedback loops refer to a type of program input in which program outputs issued at time $t-1$ are used as part of the input used in program execution at time t. In the case of an expression-tree, a feedback loop can be formed using an edge from a source tree-node to a destination tree-node. Most often, the source node is the root of the expression-tree (representing program output), which feeds back to the expression-tree using specially designed terminal or function nodes. Iba et al. [31] (1995) introduced special terminals which point at any non-terminal node within the tree. The value given by a terminal is the value at the indicated point in the tree during previous program execution. Sharman et al. [32] (1995) similarly used primitives to reference values returned by the root of an expression-tree or calculated at any of its constituent nodes at a previous program execution; explicit `push` functions within the expression-tree save

the value at that point in the tree by pushing it onto a stack. A feedback loop that is formed using previously-issued program outputs is presented in the work of [8], which tackled the problem of sound classification out of raw time-series. Finally, the most recent study of feedback loops in GP for sequence learning is presented in the work of [33]. The GP system is allowed access to a vector composed of a combination of past sequence values and program outputs of previous executions. Input sequences are serially processed using a single-step rolling window.

3 Scope of Research

The literature review confirms that the task of sequence classification by means of GP with memory has received very limited attention [8,18,28,30]. We hypothesise that the use of state variables in sequence learning problems is hindered when a fixed window of past sequence values is used as input. In such cases the search quickly converges to areas of the space containing autoregressive models, and ignores the use of memory. We speculate that autoregressive models form local optima in the fitness landscape and draw the attention of evolutionary search early in the process. On average, early-generation memory-enabled individuals are worse as compared against early-generation autoregressive programs, especially in cases where the number of previous sequence values form an adequate "context" for prediction/classification. Locating well-functioning stateful programs requires a more extensive search than in the case of search in spaces of stateless programs.

There is evidently a research gap between the evolution of sequence processing programs that are based on standard GP using a fixed number of past sequence value as input, and the evolution of programs with indexed memory. We believe that a method of serial processing of a sequence using a *single* sequence value as input will implicitly bias towards the use of memory. The computations that can be expressed using memory are likely to perform better than computations that merely represent variations of an autoregressive model of order one (given a single input value). The aim of this work is to test the efficiency of combining single-step sequence processing and memory-enabled GP. We will compare several stateful GP systems against standard *stateless* GP (no use of memory). The type of sequence learning problem that we address is that of financial time-series classification, which is also a novel application in the context of the proposed method.

4 Methods

4.1 GP Systems with Memory

In this study, program state is based on indexed-memory. Indexed-memory is implemented as an array of `double` variables of size M (i.e. `double[] Mem`). Operations of `read`, `write` and soft-assignment are included in the function-set to access memory.

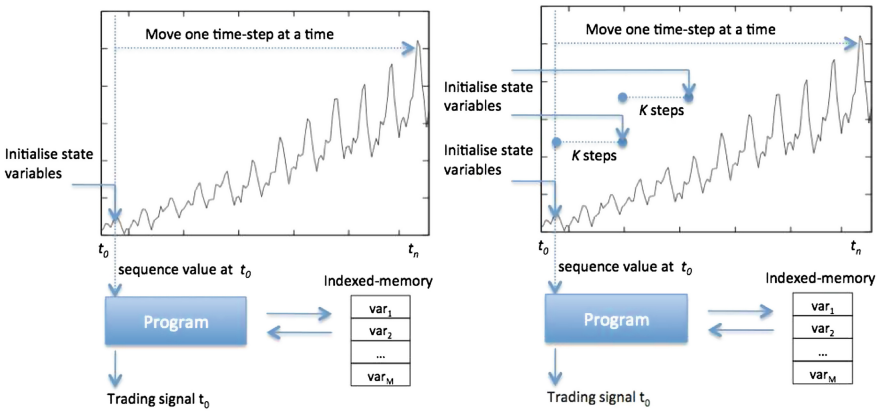

Fig. 1. Memory utilisation schemes. (a) Stateful-A. (b) Stateful-B.

Figure 1 (Stateful-A) presents the general case of a stateful sequence-processing program using a single-step rolling window. Every program in the population is allocated an individual array of `double` variables of size M. Variables are initialised to the value of zero prior to program execution. The sequence is linearly processed; at each time-step, a *single* value is fed as input to the program which outputs a value of type `double`, subsequently mapped to a trading signal. During each execution, a program is allowed to access memory. The method of initialising memory once at the beginning of sequence-processing can handle an arbitrary context-length for classification.

Figure 1 (Stateful-B) presents a different approach to the use of memory based on repeated initialisations of memory throughout sequence processing. Once a subset of sequence values of size K has been processed, memory is re-initialised to default values. The rest of the procedure is similar to that of Stateful-A. The aim is to experiment with a stateful system that can handle a context-length extending to localised parts of the entire sequence. This method requires the selection of the value for parameter K. For this purpose, one may adopt a principled approach like cross-validation. Another option is to use an adaptive value for K in different parts of the sequence. In this study we do not attempt any kind of optimisation for K, but report results using a range of values.

4.2 Feature Extraction

Technical Analysis (TA) indicators are used as filters to pre-process a *raw sequence* $\{s_i\}_{i=1}^{N}$ into a *feature sequence* $\{f_i\}_{i=1}^{K}$, with K < N. These indicators are [34,35]: (a) **simple moving average** (MA), (b) **trade break out** (TBO), (c) **filter** (FIL), (d) **volatility** (VOL), and **momentum** (MOM). TA indicators are parameterised with lag periods of certain time-steps. We used lag

Table 1. Strongly-typed language.

FUNCTION SET		
	Argument(s) type	**Return type**
Mathematical		
+, -, *	double, double	double
/ (protected)	double, double	double
log (protected)	double	double
sqrt (protected)	double	double
sin	double	double
Boolean logic		
and, or, nand, nor, xor	boolean, boolean	boolean
Memory access		
read	double	double
write, softAdd, softMul	double, double	double
Predicates		
$\geq$, $<$	double, double	boolean
Conditional		
IF-Then-Else	boolean, double, double	double
TERMINAL SET		
	Value	**Type**
Constants	$\{-10.0, -9.0, -8.0, \ldots, 8.0, 9.0, 10.0\}$	double
Input features	MA(5), MA(10), ..., MA(195), MA(200)	double
	TBO(5), TBO(10), ..., TBO(195), TBO(200)	double
	FIL(5), FIL(10), ..., FIL(195), FIL(200)	double
	MOM(5), MOM(10), ..., MOM(195), MOM(200)	double
	VOL(5), VOL(10), ..., VOL(195), VOL(200)	double

periods ranging from 5 to 200 time-steps, with a step of 5 time-steps. This generates 40 feature sequences using each indicator. Given 5 indicators, we generate 200 feature sequences in total. In order to synchronise the feature sequences, all technical indicators are invoked starting on sample s_{200} of the raw sequence. Features populate an input vector $X \in \mathbb{R}^{200}$. That is, at each time-step 200 feature-values are extracted, one value from each feature sequence respectively.

4.3 Program Language

Each program is given access to an array of variables `Mem`. Standard `read(i)` and `write(i, value)` primitives are included in the function-set. They operate as in [14]. `read` has one argument, which is evaluated to a memory index; it returns the value of memory at that index. `write` has two arguments; the first is evaluated to a memory index, while the second is evaluated to a value. `write` stores the value to the memory location and returns the value that has just overwritten (i.e. the value prior to the update).

In addition, we use soft-assignment operators; these are `softAdd(i, value)` and `softMul(i, value)`. The former performs `Mem[i]=Mem[i]*value`, whereas the latter performs `Mem[i]=Mem[i]*value`. Both operators have two arguments; the first evaluates to a memory index, and the second evaluates to a soft update value. Both operators return the value stored in `Mem[i]` prior to the update. In order overcome the potential problem of `IndexOufOfBounds` exceptions in all four memory access operations above, we apply `i modulo M` prior to accessing memory. Here, i is the evaluated index in an operation, and M is the memory size. M is set to 10 in the experiments.

The GP system evolves programs with real-valued output. Table 1 presents the strongly-typed function and terminal sets.

4.4 Fitness Function

The fitness function (to be maximised) is the **Sharpe ratio**. For a benchmark trading strategy R_b with a constant risk-free return, the Sharpe ratio of trading strategy R_a is defined as $\mathbb{E}[R_a]/\sqrt{var[R_a]}$, where $\mathbb{E}[R_a]$ is the average daily return (ADR) and $\sqrt{var[R_a]}$ is the standard deviation of daily returns using R_a over a trading period T. In order to calculate $\mathbb{E}[R_a]$ and $var[R_a]$ we need to first generate the sequence of daily returns. A sequence of daily returns $\{r_t\}_{t=1}^{N-1}$ is produced out of a sequence of closing prices $\{s_t\}_{t=1}^{N}$ using $r_t = (s_t - s_{t-1})/s_{t-1}$, where s_t and s_{t-1} are the sequence values at time t and $t-1$ respectively. Positive program output is interpreted as *go-long* trading signal ($b = 1$), whereas negative program output is interpreted as *go-short* trading signal ($b = -1$). Let r_t be the daily return at time t, and let b_{t-1} be the trading signal generated by the program at time $t-1$. Then $d_t = b_{t-1}r_t$ is the realised return at time t.

$\mathbb{E}[R_a]$ and $var[R_a]$ given a program R_a that produces a series of trading signals $\{b_t\}_{t=1}^{T}$ over a sequence of daily returns $\{r_t\}_{t=1}^{T}$ are calculated as:

$$\mathbb{E}[R_a] = \frac{1}{T}\sum_{t=2}^{T} b_{t-1}r_t \tag{1}$$

$$var[R_a] = \frac{1}{T-2}\sum_{t=2}^{T}(b_{t-1}r_t - \mathbb{E}[R_a])^2 \tag{2}$$

4.5 Financial Time-Series Datasets

The datasets used are the foreign exchange rate of EUR/USD, S&P500 and Nikkei225 daily indices for the period of 01/01/1990 to 31/03/2010. The first 2,500 trading days are used for training, whereas the remaining 2,729 are equally divided into two sets of 1,364 days respectively. The *validation-set* is used for model selection, whereas the *test-set* is used to assess the generalisation performance. N best-of-generation programs are gathered during N generations of an evolutionary run for purposes of model selection. The output from a run is the best-of-generation program with the highest ADR on the validation-set.

4.6 Run Parameters

All GP systems are generational and elitist (1 % of population size). They use tournament selection with a tournament size of 4. The population size is set to 1,000 individuals, and evolution proceeds for 50 generations. Ramped-half-and-half tree creation with a maximum depth of 5 is used to initialise a run. Throughout evolution, expression-trees are allowed to grow up to a maximum depth of 10. The evolutionary search is performed using subtree mutation, point mutation and subtree crossover. In point mutation an inner- or leaf-node is

replaced random-uniformly from the set of available function and terminal nodes respectively, under type and arity constraints. The probability of a single node being mutated in point mutation is $1/size$, where $size$ is the number of nodes in an expression-tree. A probability governs the application between mutation and crossover, set to 0.7 in favour of mutation. When mutation is selected, subtree mutation is applied with a probability 0.6 relative to point mutation.

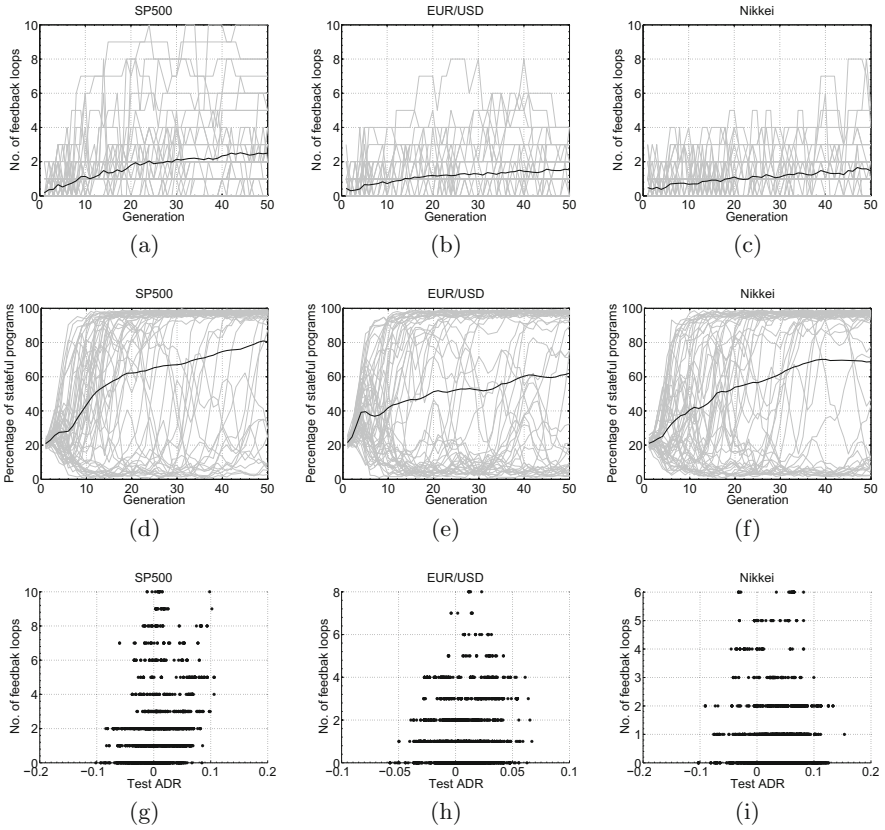

Fig. 2. First row shows the evolution of feedback loops in the best-of-generation individuals (50 runs, mean in bold). Second row shows the evolution of proportion of stateful programs in the population (50 runs, mean in bold). Third row presents scatter plots of test ADR versus number of feedback loops is best-of-generation individuals of 50 runs (2,500 points in total). All graphs are based on simulations using stateful-A.

4.7 Experiment Design

We will compare the generalisation performance of the stateful GP systems described in Sect. 4.1, and also that of a standard GP system that uses no memory (stateless). The stateless system does not include memory access primitives

(Table 1) in its function set; otherwise, function and terminal sets are similar to those of the stateful systems. In addition, we experiment with different time-steps K for memory re-initialisation in Stateful-B system; those of 20, 40, 60, 80, and 100 time-steps.

5 Results

5.1 Generalisation Performance

We performed 50 independent evolutionary runs for each of the three datasets using both stateful and standard GP (stateless) systems. Figure 3 shows the distribution of generalisation performance (measured as ADR over the test-set period) for the best-of-run selected programs in 50 runs. We performed Mann-Whitney U tests (included the Bonferroni correction to adjust for multiple comparisons) to test the significance on the difference of median test ADR between different systems. We set the level of significance at $p \leq 0.05$.

For the case of S&P500 (Fig. 3(a)), a statistically significant difference ($p = 0.046$) was found between stateless and stateful-B-60, with the latter outperforming the former. Overall, median generalisation performance of stateful systems is never worse than that of stateless GP, however not always statistically significant. Also, no statistically significant differences were found between the performance of stateful systems. Nonetheless, we observe that the median performance of stateful-B systems that utilise large time-gaps (K of 60, 80,100) in-between memory re-initialisations performed better than stateful-A.

For the case of EUR/USD (Fig. 3(b)), stateful-B-100 yielded a slightly higher median performance than stateless GP, however the difference is not statistically significant. The differences in the median performance of both stateless and stateful-B-100 against stateful-B-40 and stateful-B-60 are statistically significant with $p < 0.03$ and $p < 0.04$ respectively; stateless and stateful-B-100 perform better. Also, in all but the case against stateful-B-100, the stateless system yields higher median generalisation performance that the rest of stateful systems (although no statistical significance in difference).

For the case of Nikkei (Fig. 3(c)) results suggested that three stateful systems generalised significantly better than stateless GP. Statistically significant differences were found between stateless vs. stateful-B-40 ($p = 0.006$), stateless vs. stateful-B-60 ($p = 0.03$), and stateless vs. stateful-B-100 ($p = 0.009$). No statistically significant differences were found between stateful systems. However, stateful-B-100 that allows for a large time-gap in-between memory re-initialisations (K set to 100) yields a higher median test-set ADR than stateful-A.

5.2 Model Selection

The second column of Fig. 3 shows the distributions of generation numbers for model selection (generation number in which the best-of-run validation ADR occurs) in 50 runs for the three problems. We also plotted the evolution of best-of-generation test ADR in Fig. 4 in order to contrast it against the generations of

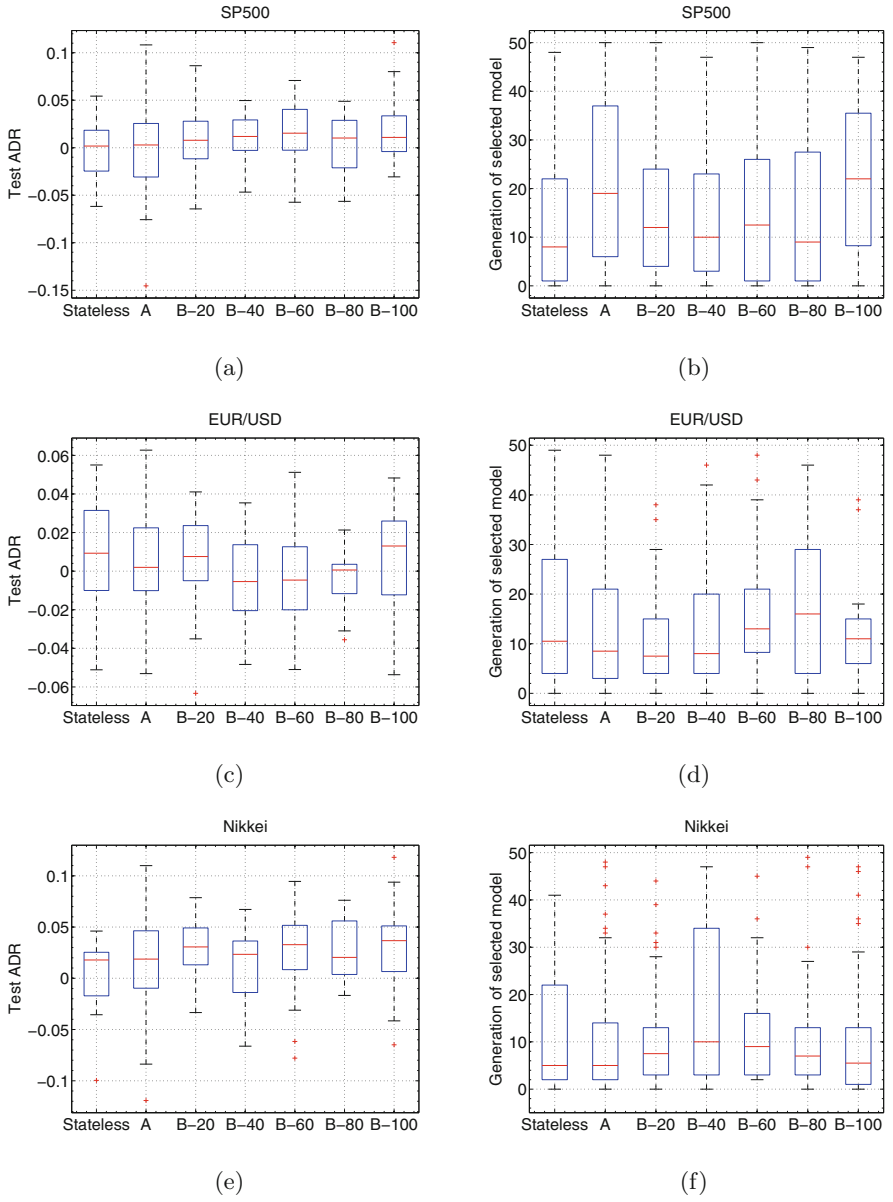

Fig. 3. First column shows the distributions of the test-set ADR (measure of generalisation) for different systems. Second column shows the distributions of generation numbers in which the validation-set ADR was the highest, and therefore model selection was performed.

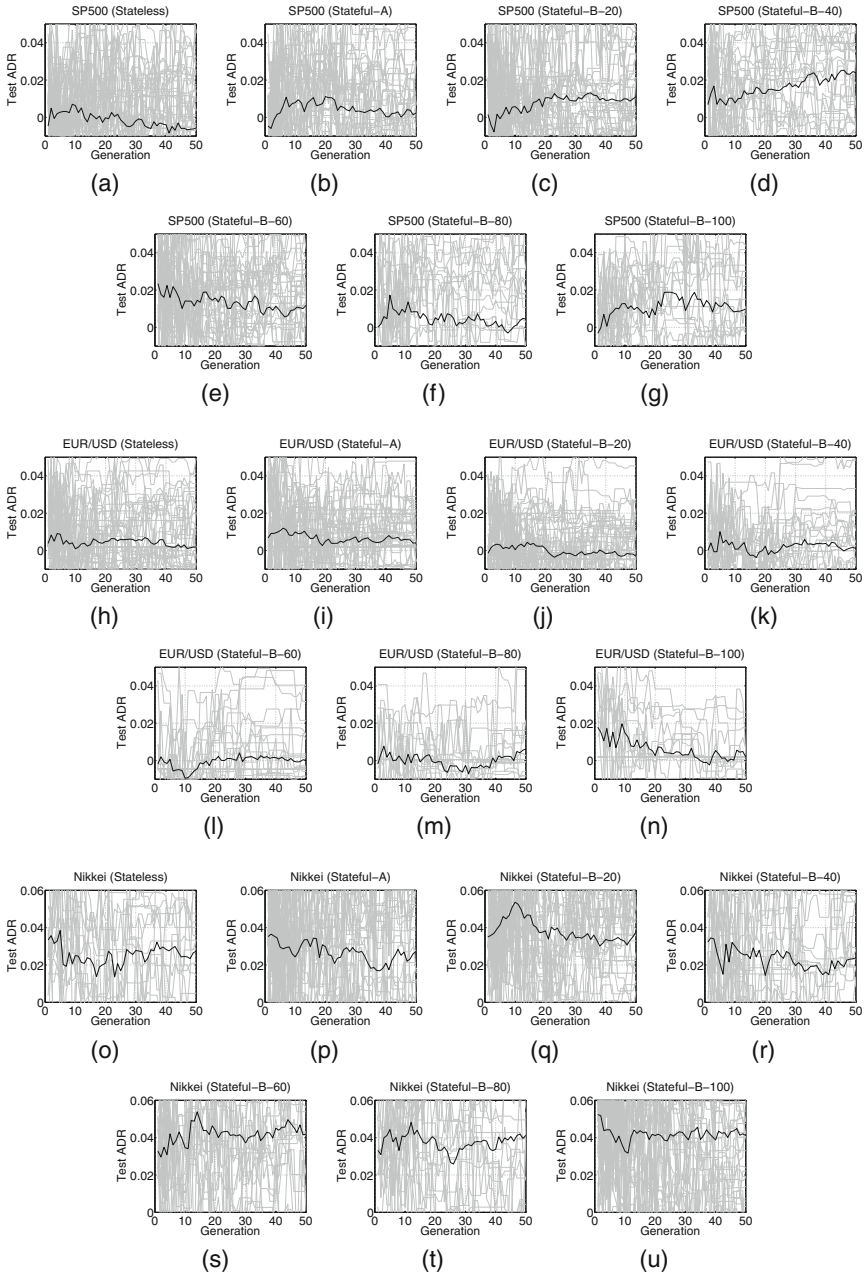

Fig. 4. Evolution of best-of-generation test-set ADR. Each graph plots 50 independent evolutionary runs, and the average is shown in bold. Figure 4(a) to Fig. 4(g) for S&P500. Figure 4(h) to Fig. 4(n) for EUR/USD. Figure 4(o) to Fig. 4(u) for Nikkei.

model selection in Fig. 3. This way we can assess the ability of the validation-set estimate to track the generalisation performance of an evolved solution.

For the case of S&P500 we observe that, on average, generalisation in stateless GP (Fig. 4(a)) improves approximately up to generation 10; loss of generalisation is seen after that point. The median generation number for model selection in stateless (Fig. 3(b)) is similarly found at approximately generation 10. In addition we observe that model selection is effective in all stateful-A, stateful-B-60 and stateful-B-80. It is biasing the selection of the model towards the beginning of the evolutionary runs (median generation of approximately 10). Indeed, in all three systems, the generalisation curves (Fig. 4) show that loss of generalisation is realised early in the evolutionary run. There are however two cases, those of stateful-B-20 and stateful-B-40 in which the generalisation performance improves on average past generation 30 in Fig. 4(c) and Fig. 4(d) respectively. In those runs, the median generation number for model selection is approximately generation 10. The validation-set ADR estimate is therefore hindering the discovery of better-generalising programs in this cases.

For the case of EUR/USD, Fig. 4 suggests that, on average, generalisation improves at the very early stages of runs, and then worsens for the systems of stateless, stateful-A, stateful-B-40, stateful-B-80. The median generation number for model selection is approximately 10 in Fig. 3(d) for the aforementioned systems. The curves for stateful-B-20 in Fig. 4(j) show that, on average, the generalisation performance improves up to approximately generation 20, however model selection is realised much earlier in the majority of runs (Fig. 3(d)).

Finally, for the case of Nikkei, in the systems of stateless, stateful-A, stateful-B-40, generalisation as a function of generation number attains a global maximum at the initial random generation (Fig. 4(p) and (r)). The median generation number in Fig. 3(f) is well under 10 generations for stateful-A. In the case of stateful-B-40 the median generation number is set to 10 generations and the upper-bound of the interquartile range is set to approximately generation 33. This results in significant loss of generalisation at that point. In the case of stateless, Fig. 4(o) shows that a drop in generalisation is realised at approximately generation 6 after which generalisation is shown to improve up to the final generation. The median generation number in Fig. 3(f) for the stateless system is bellow generation 5. Finally, model selection is effective for stateful-B-20, stateful-B-60 and stateful-B-80; it successfully captures the global maximum of test-set ADR at approximately after 10 generations.

5.3 Memory Utilisation

The analysis in this section is based on the simulations using stateful-A. The first row of Fig. 2 shows the evolution of the *number of feedback loops* in best-of-generation individuals. The number of feedback loops refer to the count of `write`, `softAdd` and `softMul` primitives that are found in an expression-tree. We use the term *feedback loop* instead of *memory access* to emphasise the fact that memory has been written to before it is accessed. We observe that on average

the best-of-generation individuals contain at least one feedback loop, and that in the case of S&P500 the average feedback loops is set to 2.

The second row of Fig. 2 shows the evolution of the percentage of stateful programs in the population. A stateful program is one that contains at least one of the memory manipulation primitives `write`, `softAdd` or `softMul`. First we note that the initial proportion of stateful individuals is consistently set to approximately 20 % in all problems. Additionally we observe that, on average, in all three problems the percentage increases with increasing generation number, indicating that memory-based computations are fitter than stateless ones. Nonetheless, it is shown that in all problems there exist roughly two categories of runs. Those in which the population becomes quickly dominated by stateful programs, and those in which stateful programs are completely eliminated at the early stages of evolution. One explanation for this is due to the stochastic effects of selection in the early stages of evolution. It seems that in this problem domain both kinds of stateful and stateless programs have similar fitness in early stages of a run, thus selection is unable to discriminate between them. Subsequent sampling errors focus the search around either stateful of stateless programs, and evolution is unable to escape these local maxima.

Finally, the last row of Fig. 2 presents scatter plots of test ADR versus the number of feedback loops in all 2,500 best-of-generation individuals accumulated during 50 evolutionary runs of 50 generations each. Our aim is to qualify the relationship between generalisation performance and memory utilisation. Figure 2(g) shows that in the case of S&P500 a number of feedback loops greater than 3 is mainly seen with positive test-set ADR. The Pearson correlation coefficient (PCC) of ADR vs. feedback loops is 0.25, suggesting a weak linear relationship between the two. In the case of EUR/USD (Fig. 2(h)) we see that a number of feedback loops greater than 4 is primarily associated with a positive test-set ADR. In this case PCC is 0.11 suggesting a very weak linear relationship between feedback loops and generalisation. Finally, Fig. 2(i) shows that positive test-set ADR is most often obtained from programs with a number of feedback loops greater than 3. The PCC is this case is -0.03, and suggests no relationship between feedback loops and generalisation.

6 Conclusion

Stateful GP for sequence processing using a single-step rolling-window approach was empirically confirmed to generalise similarly well with standard GP in one problem and outperform it in two other problems. Memory re-initialisation during sequence processing was shown to be advantageous and performed better than the case of no re-initialisation. In addition, it was shown that the frequency in which memory is re-initialised exerts an effect in generalisation performance; the less often the better. In practice, principled approaches like cross-validation should be employed for choosing the optimal frequency. Moreover, an interesting relationship was discovered between the number of memory accesses and generalisation performance; a positive out-of-sample ADR was obtained in the majority

of programs that made use of more than 3 feedback loops in their execution for all three problems.

Validation-based model selection performs well in the majority of cases, however, there are cases where it cannot accurately estimate the generalisation performance. We believe that there is an interplay between the modelling technique used and the sampling bias that can be unintentionally introduced when choosing the hold-out dataset at random. The issue of sampling bias becomes particularly important in the case of sequence datasets in which the samples are not independently distributed but often exhibit time-dependencies. Furthermore, theory dictates that reducing the data sample-size during training can exert a negative impact on the generalisation performance of a learning system. Future work should focus on analytical methods for model selection. Analytical methods usually require the optimism of training error to be expressed as a function of the training sample-size and model complexity. Defining complexity measures for stateful programs is a promising topic for future work.

Finally, results suggested that there is an inherent difficulty in focusing the search in parts of the space populated by stateful individuals. Nearly-random stateless programs tend to perform equally well (if not better) as stateful ones in the early stages of evolution. Sampling errors in selection may eliminate stateful programs altogether from the population. Specifically, it was shown that there are many cases in which the search quickly concentrated solely around stateless programs. We strongly believe that if stateful programs are to be evolved, the exploration of "stateful" areas of the search space needs to be emphasised early in the run, giving more chances of sampling nearby areas of initially-unfit stateful programs. We are currently experimenting with a new type of dynamic multi-objective fitness function that introduces a bias towards stateful individuals (i.e. rewards the presence of memory-based primitives in an expression-tree) in the early stages of evolution, then gradually reduces this bias focussing solely on Sharpe ratio.

Acknowledgement. This publication has emanated from research conducted with the financial support of Science Foundation Ireland under Grant Number 08/SRC/FM1389.

References

1. Langdon, W.B., Poli, R.: Foundations of Genetic Programming. Springer-Verlag, Heidelberg (2002)
2. Cramer, N.L.: A representation for the adaptive generation of simple sequential programs. In: Grefenstette, J.J. (ed.), Proceedings of an International Conference on Genetic Algorithms and the Applications, Carnegie-Mellon University, pp. 183–187. Pittsburgh, PA, USA, 24–26 July 1985
3. Koza, J.: Genetic Programming: on the programming of computers by means of natural selection. MIT Press, Cambridge, MA (1992)
4. Koza, J.: Genetic Programming II: automatic discovery of reusable programs. MIT Press, Cambridge, MA (1994)

5. Montana, D.J.: Strongly typed genetic programming. BBN Technical Report #7866, Bolt Beranek and Newman Inc, 10 Moulton Street, Cambridge, MA 02138, USA, March 1994
6. Huelsbergen, L.: Toward simulated evolution of machine language iteration. In: Koza, J.R., Goldberg, D.E., Fogel, D.B., Riolo, R.L., eds.: Genetic Programming 1996: Proceedings of the First Annual Conference, Stanford University, CA, USA, pp. 315–320. MIT Press, 28–31 July 1996
7. Kirshenbaum, E.: Genetic programming with statically scoped local variables. Technical Report HPL-2000-106, Hewlett Packard Laboratories, Palo Alto, 11 August 2000
8. Conrads, M., Nordin, P., Banzhaf, W.: Speech sound discrimination with genetic programming. In: Banzhaf, W., Poli, R., Schoenauer, M., Fogarty, T.C. (eds.) EuroGP 1998. LNCS, vol. 1391, p. 113. Springer, Heidelberg (1998)
9. Agapitos, A., Lucas, S.: Evolving a statistics class using object oriented evolutionary programming. In: Ebner, M., O'Neill, M., Ekárt, A., Vanneschi, L., Esparcia-Alcázar, A.I. (eds.) EuroGP 2007. LNCS, vol. 4445, pp. 291–300. Springer, Heidelberg (2007)
10. Agapitos, A., Togelius, J., Lucas, S.M.: Evolving controllers for simulated car racing using object oriented genetic programming. In: Proceedings of the Genetic and Evolutionay Computation Conference (2007)
11. Agapitos, A., Togelius, J., Lucas, S.M.: Multiobjective techniques for the use of state in genetic programming applied to simulated car racing. In: Proceedings of IEEE CEC, pp. 1562–1569 (2007)
12. Poli, R., McPhee, N.F., Citi, L., Crane, E.: Memory with memory in genetic programming. J. Artif. Evol. Appl. **2009**, 429–433 (2009)
13. Teller, A.: Turing completeness in the language of genetic programming with indexed memory. In: Proceedings of the IEEE World Congress on Computational Intelligence. vol. 1, Orlando, Florida, USA, pp. 136–141. IEEE Press (27–29 June 1994) (1994)
14. Teller, A.: The evolution of mental models. In: Kinnear Jr., K.E. (ed.) Advances in Genetic Programming. MIT Press, Cambridge (1994)
15. Jannink, J.: Cracking and co-evolving randomizers. In: Kinnear Jr., K.E. (ed.) Advances in Genetic Programming, pp. 425–443. MIT Press, Cambridge (1994)
16. Teller, A., Veloso, M.: A controlled experiment: Evolution for learning difficult image classification. In: Pinto-Ferreira, C., Mamede, N.J. (eds.) Progress in Artificial Intelligence. LNCS, vol. 990, pp. 165–176. Springer, Heidelberg (1995)
17. Teller, A., Veloso, M.: Algorithm evolution for face recognition: What makes a picture difficult. In: International Conference on Evolutionary Computation, Perth, Australia, pp. 608–613. IEEE Press, 1–3 December 1995
18. Teller, A., Veloso, M.: Program evolution for data mining. Int. J. Expert Syst. **8**(3), 216–236 (1995)
19. Andre, D.: Evolution of mapmaking ability: Strategies for the evolution of learning, planning, and memory using genetic programming. In: Proceedings of the IEEE WCCI. vol. 1, pp. 250–255. Florida, USA (27–29 June 1994) (1994)
20. Brave, S.: The evolution of memory and mental models using genetic programming. In Koza, J.R., Goldberg, D.E., Fogel, D.B., Riolo, R.L., eds.: Genetic Programming 1996: Proceedings of the First Annual Conference, Stanford University, CA, USA, MIT Press (28–31 July 1996) 261–266

21. Haynes, T.D., Wainwright, R.L.: A simulation of adaptive agents in hostile environment. In: George, K.M., Carroll, J.H., Deaton, E., Oppenheim, D., Hightower, J. (eds.) Proceedings of the 1995 ACM Symposium on Applied Computing, pp. 318–323. USA, ACM Press, Nashville (1995)
22. Langdon, W.B.: Genetic Programming and Data Structures: Genetic Programming + Data Structures = Automatic Programming! vol. 1 of Genetic Programming. Kluwer, Boston, 24 April 1998
23. Bruce, W.S.: Automatic generation of object-oriented programs using genetic programming. In: GP 1996: Proceedings of the 1st Annual Conference
24. Nordin, P., Banzhaf, W.: Programmatic compression of images and sound. In: Koza, J.R., Goldberg, D.E., Fogel, D.B., Riolo, R.L. (eds.): Genetic Programming 1996: Proceedings of the First Annual Conference, Stanford University, CA, USA, pp. 345–350. MIT Press, 28–31 July 1996
25. Spector, L., Luke, S.: Cultural transmission of information in genetic programming. In: Koza, J.R., Goldberg, D.E., Fogel, D.B., Riolo, R.L. (eds.): Genetic Programming 1996: Proceedings of the First Annual Conference, Stanford University, CA, USA, pp. 209–214. MIT Press, 28–31 July 1996
26. Koza, J.R., Andre, D., Bennett III., F.H., Keane, M.: Genetic Programming 3: Darwinian Invention and Problem Solving. Morgan Kaufman
27. O'Neill, M., Ryan, C.: Investigations into memory in grammatical evolution. In: GECCO 2002, (ed.), pp. 141–144 (2002)
28. Agapitos, A., Dyson, M., Lucas, S.M., Sepulveda, F.: Learning to recognise mental activities: genetic programming of stateful classifiers for brain-computer interfacing. In: GECCO 2008: Proceedings of the 10th annual conference on Genetic and evolutionary computation (2008)
29. Agapitos, A., O'Neill, M., Brabazon, A., Theodoridis, T.: Learning environment models in car racing using stateful genetic programming. In: Proceedings of the IEEE Conference on Computational Intelligence and Games, Seoul, South Korea, pp. 219–226. IEEE (31 August - 3 September 2011) (2011)
30. Agapitos, A., O'Neill, M., Brabazon, A.: Stateful program representations for evolving technical trading rules. In: GECCO 2011: Proceedings of the 13th Annual Conference Companion on Genetic and Evolutionary Computation (2011)
31. Iba, H., de Garis, H., Sato, T.: Temporal data processing using genetic programming. In: Eshelman, L. (ed.): Genetic Algorithms: Proceedings of the Sixth International Conference (ICGA95), pp. 279–286, 15–19 July 1995
32. Sharman, K.C., Esparcia Alcazar, A.I., Li, Y.: Evolving signal processing algorithms by genetic programming. In: Zalzala, A.M.S. (ed.): First International Conference on Genetic Algorithms in Engineering Systems: Innovations and Applications, GALESIA. vol. 414, pp. 473–480. Sheffield, UK, IEE, 12–14 September 1995
33. Alfaro-Cid, E., Sharman, K., Esparcia-Alcazar, A.I.: Genetic programming and serial processing for time series classification. Evol. Comput. **22**(2), 265–285 (2014)
34. Brabazon, A., O'Neill, M.: Biologically Inspired Algorithms for Financial Modelling. Natural Computing Series. Springer, Heidelberg (2006)
35. Tsang, E.P.K., Li, J., Markose, S., Er, H., Salhi, A., Lori, G.: EDDIE in financial decision making. J. Manage. Econ. **20**, 101–112 (2000)