A Modular Memory Framework for Time Series Prediction

Stephen Kelly, Jacob Newsted, Wolfgang Banzhaf, and Cedric Gondro
BEACON Center for the Study of Evolution in Action
Michigan State University
{kellys27,newsted1,banzhafw,gondroce}@msu.edu

ABSTRACT

Tangled Program Graphs (TPG) is a framework for genetic programming which has shown promise in challenging reinforcement learning problems with discrete action spaces. The approach has recently been extended to incorporate temporal memory mechanisms that enable operation in environments with partial-observability at multiple timescales. Here we propose a highly-modular memory structure that manages temporal properties of a task and enables operation in problems with continuous action spaces. This significantly broadens the scope of real-world applications for TPGs, from continuous-action reinforcement learning to time series forecasting. We begin by testing the new algorithm on a suite of symbolic regression benchmarks. Next, we evaluate the method in 3 challenging time series forecasting problems. Results generally match the quality of state-of-the-art solutions in both domains. In the case of time series prediction, we show that temporal memory eliminates the need to pre-specify a fixed-size sliding window of previous values, or autoregressive state, which is used by all compared methods. This is significant because it implies that no prior model for a time series is necessary, and the forecaster may adapt more easily if the properties of a series change significantly over time.

CCS CONCEPTS

• **Computing methodologies** → **Machine learning**; **Genetic programming**; **Bio-inspired approaches**;

KEYWORDS

Genetic Programming, Modularity, Memory, Time Series Prediction

ACM Reference Format:
Stephen Kelly, Jacob Newsted, Wolfgang Banzhaf, and Cedric Gondro. 2020. A Modular Memory Framework for Time Series Prediction. In *Genetic and Evolutionary Computation Conference (GECCO '20), July 8–12, 2020, Cancun, Mexico.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3377930.3390216

1 INTRODUCTION

Time Series Forecasting, or predicting future values based on previously observed values, is an important aspect of many real-world problems. For example, forecasting the demand on a server each hour, forecasting the number of passengers on public transit each day, or forecasting an EEG trace to predict changes in patient health are applications of extreme practical importance. Current methods typically rely on prior inspection of the time series and human intuition, or heuristics, in order to parameterize the prediction model [1, 29]. However, this can be problematic when the characteristics of the data stream are not well understood. Furthermore, applying a static model and parameters assumes that the underlying process generating the time series is stationary. However, many real-world forecasting environments change significantly over time. In this work, we introduce a highly modular approach to time series forecasting which builds the prediction model entirely through interaction with that environment. A temporal memory mechanism eliminates the need to hard-code any recursive structure into the model or to pre-specify a fixed window of historical values to analyze at any point in time. This represents a very general framework for time series forecasting that may have broader applications in memory-intensive tasks with non-stationary properties, e.g [6, 31].

1.1 Emergent Modularity

Tangled Program Graphs (TPG) is a Genetic Programming (GP) framework which incrementally builds computational organisms from multiple subsystems which were initially developed independently akin to compositional evolution [32]. In doing so, TPG automates two critical properties of such a system: 1) The identification of stable building blocks, or subsystems; and 2) Establishing the nature of the interaction among subsystems within a hierarchical organism, or *module interdependence*.

Relative to the first property to be automated, i.e. discovery of stable building blocks, Herbert Simon [25] suggested that the presence of stable intermediate structures speeds up evolution by providing building blocks from which increasingly complex hierarchies may be constructed. Put simply, Simon points out that if a complex system is built from structurally modular building blocks, its development is less likely to require a restart from scratch should an error be introduced during construction (see Simon's famous Watchmaker's Parable for an illustrative example of this concept). In other words, modularity helps promote stability in an evolving organism, preventing a particular genome from being a "House of Cards" [15] in which a single variation might bring it tumbling down. Ultimately, Simon's suggestion is that modular systems are more *evolvable*, that is, more capable of continuously discovering new organisms with higher fitness than their parents. This theory has been investigated widely among evolutionary biologists [21, 30, 34].

As for the second property to be automated, module interdependence, Watson et al. [32] emphasize that structural modularity (i.e. structural complexity encapsulated such that dependencies

GECCO '20, July 8–12, 2019, Cancun, Mexico

ACM ISBN 978-1-4503-7128-5/20/07…$15.00
https://doi.org/10.1145/3377930.3390216

between subsystems are weaker than dependencies within subsystems) does not imply independence of subsystems. Specifically, functional interdependence among subsystems is critical for hierarchies in which all levels of organization are meaningful. Simply accumulating multiple building blocks into an aggregate system does not capture the full potential of modularity. Module interdependence is essential for emergence because without meaningful interdependence, a hierarchy of subsystems is nothing more than the sum of its parts. Watson argues that systems with strong module interdependence are evolvable under certain conditions, namely compositional evolution.

TPG has leveraged emergent modularity to make a variety of contributions in the context of visual Reinforcement Learning (RL). In the Atari video game testbed, TPG evolved game-playing agents that match the quality of solutions from a variety of deep learning methods [13]. More importantly, TPG agents were less computationally demanding and required fewer calculations per decision than any of the other methods. This efficiency is possible because 1) the hierarchical complexity of each organism is a property that emerges through interaction with the problem environment, rather than being fixed a priori, as was the case for deep learning, e.g. [20]; and 2) subsystems within a TPG organism typically specialize on different parts of the visual input space, thus only subsets of the overall organism require execution at any given point in time.

Modularity and specialization also allow TPG to support transfer learning in challenging RL problems [12]. In this case, solutions initially evolved for simple subtasks can be reused within hierarchical organisms in order to improve learning in a more complex task. The resulting agents achieve state-of-the-art levels of play in RoboCup Half-Field Offense and surpass scores previously reported in the Ms. Pac-Man literature while employing less domain knowledge during training. Again, the highly modular organisms are shown to be significantly more efficient than state-of-the-art solutions in both domains.

Finally, modularity and specialization are also useful in dynamic environments where the distribution in sensory inputs may change drastically over time. When forced to switch randomly between multiple Atari game titles throughout evolution, TPG can evolve solutions to multiple titles simultaneously with no additional computational cost [13]. In this case, modularity was critical to avoid unlearning or "catastrophic forgetting" [16] of behaviours that are intermittently important over time.

1.2 Modular Memory Models

All the work outlined in Section 1.1 was conducted using an early version of TPG in which organisms were *stateless*. That is, even though agents operated in episodic, sequential decision-making environments involving hundreds or thousands of timesteps, the agents were purely reactive. They had no temporal memory mechanism to enable the integration of experience over time. This is a serious limitation in partially-observable tasks in which it is impossible to retrieve complete information about the state of the environment from a single observation. More recently, multiple models have been proposed which support temporal memory sharing among subsystems within TPG organisms, allowing agents to operate in sequential decision-making environments with partial observability at multiple time scales [11, 26, 27]. Examples from the deep learning community have also demonstrated that modularity and specialization lead to improved generalization in dynamic tasks that require temporal reasoning [3, 6].

1.3 Research Objectives

Section 1.1 established the capabilities of TPG for evolving hierarchical/modular organisms in high-dimensional (e.g. visual) RL environments with discrete action spaces. However, many real-world RL problems involve continuous action spaces, which introduce non-trivial design choices for the RL practitioner [17, 23, 24]. For example, continuous control problems cannot be solved by simply discretizing the action space due to the exponentially large number of bins over which policies would have to be learned [19]. The first research objective of this work is therefore to test the hypothesis that TPG's shared memory framework [11] can be extended to support continuous action spaces and temporal memory management *simultaneously*. This significantly broadens the scope of real-world applications for TPG, from continuous-action reinforcement learning to time series forecasting. For our initial study here, we evaluate the proposed method in well known supervised learning tasks, namely symbolic regression and time series forecasting.

TPG's success in high-dimensional RL was due in part to its capacity to adaptively decompose the input space such that individual subsystems within an organism could specialize their role relative to small subsets of the input space, or *spatial* decomposition [13]. The second objective of this work is now to examine how the modular memory mechanism allows organisms to achieve a *temporal* problem decomposition. This is significant because temporal problem decomposition is likely beneficial in dynamic, non-stationary environments. An example of this is time series forecasting or streaming data classification tasks when the underlying process generating the data stream changes significantly over time [1, 7].

The remainder of this paper is organized as follows: Section 2 provides a detailed description of our proposed algorithm. An empirical evaluation is provided in Section 3. A suite of symbolic regression benchmarks [22] are used for an initial proof-of-concept evaluation of continuous-output TPG, which is shown to generally match the solution quality of 3 modern GP frameworks. Next, we evaluate continuous-output TPG in 3 challenging time series forecasting benchmarks. Here we show that the proposed method matches the prediction accuracy of a recently proposed approach to neuroevolution based on Cartesian Genetic Programming [29]. Furthermore, temporal memory eliminates the need to pre-specify a fixed-size sliding window of previous values, or autoregressive state, which is used by all compared methods. Section 4 concludes the paper and provides an outlook to future work.

2 ALGORITHM DESCRIPTION

The algorithm investigated in this work is an extension of Tangled Program Graphs [13]. TPG was initially designed for RL tasks in which solutions map sensor inputs to a set of discrete actions. This work represents the first time the method has been used to build solutions that map inputs to a continuous (real-valued) output. This is achieved through an extension of the shared memory mechanism introduced in [11]. This section outlines the extended algorithm,

paying specific attention to two critical components: 1) How memory is shared among individual programs in a team-based model; and 2) How multiple independent teams are adaptively combined into a hierarchical organism, or *program graph*, through compositional evolution. All source code is publicly available [10].

2.1 Coevolving Independent Teams

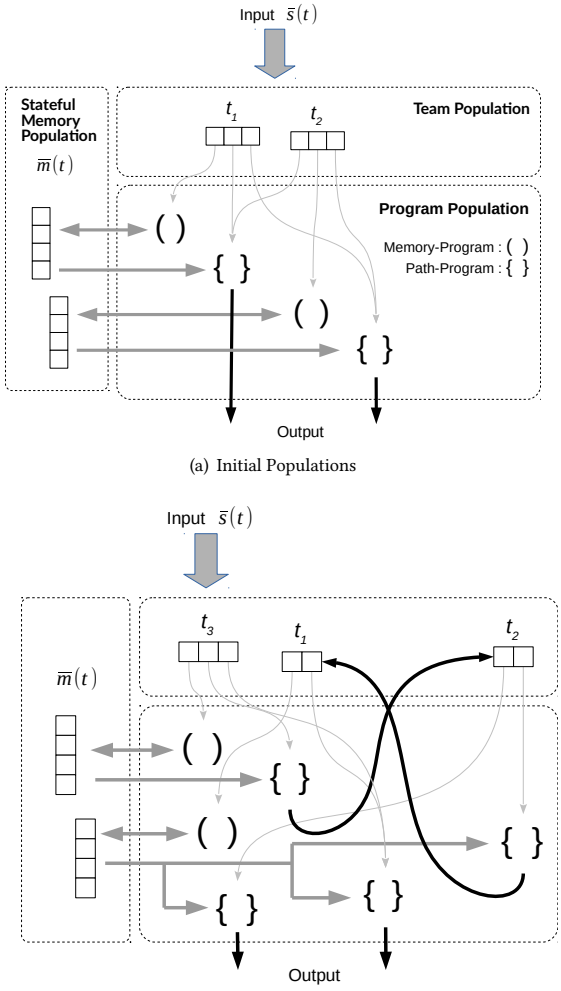
A *team of programs* is the basic representation for a complete solution in TPG. Each team defines a group of programs that *collectively* map input state at time t, $\vec{s}(t)$ to a single output value. Teams can be thought of as the root vertex in a computational graph where the *edges* are programs that process $\vec{s}(t)$ and produce output, see Figure 1(a). In this work, all programs are linear register machines [4], see Algorithm 1. For the purposes of this study, it is important to note that programs contain internal register memory that is *stateless*, that is, reset prior to each execution. Programs also have a pointer to one shared *stateful* memory bank that is only reset at the start of each evaluation. In the case of sequential decision-making or time series tasks where the program is executed multiple times per evaluation, shared stateful memory allows programs to communicate and integrate information across multiple timesteps.

Two types of program are supported within a team:

(1) **Memory-programs** manage the content of stateful memory. They read from current environmental state, $\vec{s}(t)$, and/or stateful memory, $\vec{m}(t)$, and write to $\vec{m}(t)$;
(2) **Path-programs** define potential solution outputs. They are directed graph edges that dynamically set their weight at time t as a function of $\vec{s}(t)$ and $\vec{m}(t)$. Each team maintains at least two path-programs. The team maps $\vec{s}(t)$ to a *single* output by executing all programs in order and then following the path with the largest weight. If the path-program is a leaf, then its output value is the content of its shared stateful memory register $R_s[0]$, i.e., a real value (See Algorithm 1).

Algorithm 1 Example linear register machine. Each program contains one *private stateless* register bank, $R_{private}$, and a pointer to one *shareable stateful* register bank, R_{shared}. $R_{private}$ is reset prior to each execution, while R_{shared} is reset (by an external process) at the start of each evaluation. Let R_x and R_y represent generic register banks. In the proposed instruction format, R_y can be an index to a state variable (input) or may refer to either $R_{private}$ or R_{shared}. The rules governing the target register R_x differ depending on program type. For path-programs, shared memory is read-only, thus R_x will always refer to $R_{private}$. In memory-programs, R_x may refer to either memory bank. Note that path-programs have two return values (line 7). Memory-programs have no return value as their purpose is only to manipulate the content of shared memory. A complete list of operations and instruction formats appears in Table 1.

1: $R_{private} \leftarrow 0$ ▷ reset private memory bank
2: $R_x[0] \leftarrow R_x[0] \div R_y[3]$
3: $R_x[2] \leftarrow \cos R_y[1]$
4: **if** $R_x[0] < R_y[2]$ **then**
5: $\quad R_x[0] \leftarrow -R_x[0]$
6: **if** $Path$ **then**
7: $\quad$ **return** $(R_{private}[0], R_{shared}[0])$ ▷ (weight, solution value)



(a) Initial Populations

(b) Emergence of *Program Graphs*

Figure 1: Illustration of the relationship between teams, programs, and shared memory in TPG

Note that memory-programs have read/write access to $\vec{m}(t)$, while path-programs have read-only access. This enforces a division of labour in which memory-programs manage stateful memory *content*, while path-programs define the appropriate *context* (relative to $\vec{s}(t)$ and $\vec{m}(t)$) in which their shared memory register $R_s[0]$ should be selected as the output[1]. Note that this is the critical extension that allows TPG to provide continuous-valued output. In all previous studies, path-programs defined the context for a single discrete action. Here, path-programs define the context for a single shared memory register, the content of which is determined by its associated memory-program.

Teams, programs, and shared memory registers are each stored in separate populations and coevolved. Since a team of programs represents a stand-alone solution, the team as whole is evaluated

[1] In preliminary experiments, this was found to work significantly better than allowing all programs read/write access to $\vec{m}(t)$, as is the case in [11].

Table 1: Operations and instruction formats. Path-programs encode 8 operations in a 3-bit op-code. Memory-programs use a 4-bit op-code to include 7 extra operations (highlighted). In addition, memory-programs have access to 18 constants: {−0.9, −0.8, ..., −0.1, 0.1, 0.2, ..., 0.9}, included as read-only registers at the end of their private register bank $R_{private}$ (See Algorithm 1).

Instruction	Operations
$R_x[i] \leftarrow R_x[i] \circ R_y[j]$	$\circ \in \{+, -, x, \div, x^y\}$
$R_x[i] \leftarrow \circ(R_y[j])$	$\circ \in \{\cos, \ln, \exp, \sqrt{}, \sin\}$
	$\circ \in \{\tanh, x^2, \lvert x\rvert, x^3\}$
IF $(R_x[i] \circ R_y[j])$ THEN $R_x[i] \leftarrow -R_x[i]$	$\circ \in \{<, >\}$

on the task and assigned a fitness score. Evolution is driven by a generational GA such that the worst performing teams are deleted in each generation and replaced by offspring of the surviving teams. Programs have no individual concept of fitness. After team deletion, programs that are not part of any team are also deleted. As such, selection is driven by a symbiotic relationship between programs and teams: teams will survive as long as they define a complementary group of programs, while individual programs will survive as long as they collaborate successfully within a team. The process for generating team offspring uniformly samples and clones a root team, then applies mutation-based variation operators to the cloned team, as listed in Table 2. Team variation operators may add, remove, or modify programs in the team. In short, team compliment, program length and content, and the degree of memory sharing are all adapted properties. Complete details on TPG's variation operators are available in [13] and [9].

2.2 Evolving Team Hierarchies

Evolution begins with a population of R_{size} teams, each containing at least 2 path-programs and 2 memory-programs which share stateful memory banks, Figure 1(a). Path-programs are initially all leaf nodes, immediately outputting a result. However, when a path-program is modified by variation operators, it will remain a leaf with probability p_{atomic}, and will otherwise connect to one team from the set of teams present from any previous generation, chosen with uniform probability. These connection mutations are the mechanism by which TPG supports compositional evolution, adaptively recombining multiple (previously independent) teams into variably deep/wide directed graph structures, or *program graphs*, Figure 1(b).

Execution of a program graph begins at the root team (t_3 in Figure 1(b)), where all programs in the team will execute. Graph traversal then follows the path-program with the largest weight, repeating the execution process at every team along the path until a leaf node is reached. Thus, the program graph computes one path from root to leaf for each input sample $\vec{s}(t)$, where only a subset of programs in the graph (those in teams along the path) require execution. Note that cycles may appear in the graph structure but are ignored during execution. That is, no team is visited more than once per traversal. If the edge with the largest weight leads to a team that has already been visited, the edge is simply ignored. Team variation operators are constrained such that each team maintains at least one path-program that is a leaf node, ensuring an output can always be found.

As hierarchical structures emerge, only root teams (i.e. teams with indegree of 0) define independent solutions, and only these root teams are subject to deletion, cloning, and variation. As such, program graphs incrementally grow and break apart at their root node, i.e. from the top up/down. While the team and program population sizes vary throughout evolution, the number of root teams to maintain in the population (R_{size}) is pre-specified as a learning parameter and remains constant. Thus, whenever a root team is subsumed within a program graph, it will be replaced by a new root team which is sampled with uniform probability from the team population and modified by mutation operators (Table 2). Non-root teams are protected from deletion as long as they are a component of a graph that performs well collectively. This property helps TPGs avoid catastrophic forgetting in sequential problems in which specialized team behaviours may be intermittently important over time, e.g. multi-task learning [13].

In summary, the hierarchical complexity and interdependency between teams in program graphs emerges entirely through interaction with the task environment. As a program graph operates, the subset of teams/programs that require execution is dynamically selected at runtime based on the current input sample and the content of stateful memory. This has two important implications: 1) Teams are free to specialize on particular aspects of the problem and may be switched in and out of the model as needed; and 2) Program graphs can dynamically select inputs and stateful memory registers that are relevant to the current state observation (i.e. inputs and memory registers indexed by programs along the active path) while ignoring inputs/memories that are not important at the current point in time. This is conceptually similar to the modular structures and *attention* mechanisms explored by Goyal at. al. [6], in which these properties were shown to improve generalization and in dynamic memory problems. However, in that case the total number of "modules" per solution required prior specification, as did the number of "active" modules at any point in time. In this work we are specifically interested in how these model characteristics emerge through compositional evolution.

3 EXPERIMENTS

This section details our experimental analysis of the extended TPG algorithm, which is abbreviated as TPG-CM to note the inclusion of (C)ontinuous output and temporal (M)emory. First, in Section 3.1 we confirm the effectiveness of continuous-outputs in TPG-CM using a suite of symbolic regression problems specifically designed to evaluate GP systems [22]. Next, in Section 3.2 we test our proposed temporal memory mechanism by evaluating TPG-CM in 3 challenging time series forecasting benchmarks. In all experiments, TPG-CM is parameterized as per Table 2.

3.1 Symbolic Regression

Our experimental methodology under symbolic regression closely follows the work in [2]. We experiment with 14 unique functions.

Table 2: Parameterization of Team and Program populations. R_{size} is the number of root teams to maintain at any point in time. R_{gap} is the proportion of root teams to delete and replace in each generation. For the team population, p_{mx} denotes a mutation operator in which: $x \in \{d, a\}$ are the probality of deleting or adding a program respectively; $x \in \{m, n, s\}$ are the probality of creating a new program, changing a path-program's action pointer (leaf or team), and changing a program's shared memory pointer respectively. ω is the max initial team size. For the program population, p_x denotes a mutation operator in which $x \in \{delete, add, mutate, swap\}$ are the probality for deleting, adding, mutating, or reordering instructions within a program. p_{atomic} is the probability that a modified action-pointer for a path-program will be atomic (leaf).

Team population			
Parameter	Value	Parameter	Value
R_{size}	360	R_{gap}	50%
p_{md}, p_{ma}	0.7	ω	10
p_{mm}	0.2	p_{mn}, p_{ms}	0.1
Program population			
Parameter	Value	Parameter	Value
Size of $R_{private}$	8	*maxProgSize*	100
Size of R_{shared}	8	p_{atomic}	0.5
p_{delete}, p_{add}	0.5	p_{mutate}, p_{swap}	1.0

Complete details on these functions and their suitability for evaluating GP system can be found in [22]. One example problem is:

$$F_1(x_1, x_2) = \frac{e^{-(x_1-1)^2}}{1.2 + (x_2 - 2.5)^2}$$

The functions used in this work have between 2 and 10 input variables. All input values are sampled randomly from the interval $[-5, 5]$. We use 1000 training samples, 10,000 validation samples, and 40,000 test samples. Training data is used to measure fitness during evolution. The fitness function used is Mean Square Error (MSE). At intervals of 10 generations, the entire population is evaluated on the validation data. Evolution proceeds until the single best individual from the validation phase matches or exceeds the best *test* score reported for any algorithm in [2], up to a wall-clock computational budget of 4 hours. The validation champion is then evaluated on the test data to produce a final test performance score. For each of the 14 problems, we perform 100 independent runs with identical input datasets. Table 3 reports test results for TPG-CM alongside 3 GP frameworks evaluated in [2].

Of the four algorithms compared in Table 3, TPG-CM achieved the best test score in 9 out of 14 symbolic regression benchmarks. EGGP_{HGT} [2] achieved the best score in 4 benchmarks, while GP [22] was the best in 1 benchmark. This evaluation represents a positive proof-of-concept for the continuous-output component in TPG-CM. A detailed comparative analysis of TPG-CM under symbolic regression is left to future work. For example, in the most general sense, the complexity of a GP system can be characterized by counting the number of program instructions that are executed per prediction by a model under test. It is possible that the model complexity of TPG-CM is significantly greater than the compared methods, however, this data was not reported for the algorithms listed in Table 3. As such, we provide a detailed discussion of the complexity of TPG-CM in the context of time series forecasting in Section 3.2.

Table 3: Results for symbolic regression problems. Values indicate median fitness (MSE) on test data. Results for $EGGP_{HGT}$, GP, and CGP are from [2]. The lowest (best) result across all algorithms is highlighted in bold.

F	EGGP_{HGT}	GP	CGP	TPG-CM
F_1	2.47E-3	5.77E-3	6.74E-3	**2.20E-3**
F_2	5.94E6	1.28E7	1.73E7	**3.81E6**
F_3	7.22E-3	1.04E-2	1.48E-2	**5.10E-3**
F_4	**2.58E13**	3.55E13	2.58E13	8.72E15
F_5	**6.90E-1**	5.13E0	7.17E0	4.12E0
F_6	4.46E0	**2.61E0**	9.28E0	9.02E1
F_7	1.51E2	4.20E2	5.76E2	**1.35E2**
F_8	2.19E-2	1.09E-1	4.49E-2	**1.74E-2**
F_9	1.57E2	1.46E2	1.71E2	**1.19E1**
F_{10}	7.69E-2	3.22E-1	1.66E-1	**4.13E-2**
F_{11}	1.59E1	3.88E1	4.96E1	**1.49E1**
F_{12}	**6.83E2**	1.25E3	7.08E2	1.06E3
F_{18}	**3.69E-1**	4.13E4	1.20E2	3.38E4
F_{21}	1.07E0	1.07E0	1.07E0	**1.06E+0**

3.2 Time Series Forecasting

Here we evaluate TPG-CM in 3 challenging time series forecasting benchmarks; Sunspots [28], Mackey-Glass [18], and Laser [8]. The Sunspots and Laser datasets are obtained from real-world recordings, while Mackey-Glass is a chaotic series generated from a parameterized equation. Our methodology and dataset preparation matches that in [29]. All 3 time series datasets are univariate and contain 1100 samples normalized to the interval $[0, 1]$.

3.2.1 Methodology. The goal of time series forecasting is to predict (unseen) future values based on previously observed values. To do so, the model is fed individual samples in order from series $x()$ and, given sample $x(t)$, must predict the value of $x(t+1)$. Note that unlike many times series forecasting methods, we do *not* pack a sequence of previous values into a single *autoregressive* state representation to present to the prediction model. For example, it is common to define an embedding dimension D and a time delay T in order to pre-define a sliding window of prior observations to present to the model at each timestep. If $D = 4$ and $T = 3$, input to the model at time t would be $[x(t), x(t-3), x(t-6), x(t-9)]$. Assuming the nature of a time series is known a priori, then D and T can be estimated such that models with no temporal memory or recursive structure can still make accurate predictions. In this work, program graphs observe one sample at a time and therefore rely entirely on temporal memory to store previously observed values, which are retained in memory as long as necessary and selectively recalled to extrapolate future values. Due to the fact that models may only

observe one sample at a time with no autoregressive state, they must be *primed* with a series of samples before future predictions can be made. In this work, models are primed with 50 samples. Thus, before predictions begin at $x(t)$, the model is executed for each input sample in series $x(t-50), x(t-49), ..., x(t-1)$ and output values from the model are ignored. When priming is complete, recursive forecasting is used to predict future values. That is, after $x(t-1)$, samples from $x()$ are no longer used as input. Instead, the model's output values, or predictions, are fed back as input to predict future values. For example, the model's output for $x(t)$ becomes its input at $t+1$, and so on. Recursive forecasting allows predictions to any horizon. In this work, program graphs will be evaluated on forecasting up to a horizon of 100 samples.

The fitness function measures how well solutions recursively predict the next fifty samples from $t_{50}, t_{100}, ..., t_{950}$. Thus, solution fitness is the MSE over 950 predictions in total. A validation procedure measures how well each solution recursively forecasts beyond the horizon used during training. Specifically, models are used to predict the next 100 samples starting from $t_{100}, t_{200}, ...t_{900}$. MSE over the last 50 predictions from each validation set (a total of 450 predictions) is used as the validation score. To obtain a final test score, the program graph with the best validation score is used to predict the next 100 samples from t_{1000} (i.e, the model is primed with samples $x(t_{950}, ..., t_{999})$. 50 independent runs are performed in parallel. As with the symbolic regression study in Section 3.1, our first research objective is to check if TPG-CM can match state-of-the-art approaches to time series forecasting. Thus, each evolutionary run continues until either 1) the median validation MSE over all 50 runs matches or exceeds the best *test* MSE reported in [29]; or 2) the run reaches a wall-clock compute budget of 12 hours.

3.2.2 Test Results. Table 4 reports mean test scores for TPG-CM along with 3 algorithms from [29]. The compared algorithms are Autoregressive Integrated Moving Average (ARIMA), Recurrent Cartesian Genetic Programming (RCGP) and Recurrent Cartesian Genetic Programming of Artificial Neural Networks (RCGPANN). ARIMA is a standard forecasting algorithm based on statistical methods, while RCGP and RCGPANN both use GP to build prediction models with recurrent structure, making them suitable for tasks with partial observability or explicitly temporal properties such as time series forecasting. TPG-CM was able to match or exceed the performance of all compared methods in each of the 3 time series benchmarks. The single best program graph for each benchmark (max score in Table 4) was discovered by generation 9280 (Laser) , 3820 (Mackey-Glass), and 7740 (Sunspots). As in the symbolic regression study, this comparison represents a positive proof-of-concept for the continuous-output and temporal memory mechanisms of TPG-CM, and no further comparative analysis is made in this study.

Figures 2, 3, and 4 show the behaviour of TPG-CM program graphs during the test phase for each times series benchmark. Note that recursive forecasting implies that all test forecasts ("Prediction" in Figures 2(a), 3(a), and 4(a)) are produced without observing any of the 100 test points ("Target" in Figures 2(a), 3(a), and 4(a)). That is, once the model is primed (Section 3.2.1), all 100 predictions are generated entirely by feeding the model's output at t_i back to its input at t_{i+1}. This feedback signal, along with the content of stateful

Table 4: Results for time series forecasting benchmarks. Values indicate mean fitness (MSE) on test data. Results for ARIMA, RCGP, and RCGPANN are from [29]. The lowest (best) result across all algorithms is highlighted in bold.

Method	Laser mean, best	Mackey-Glass mean, best	Sunspots mean, best
ARIMA	0.0341	0.0715	0.0350
RCGP	0.0258, 0.0044	0.0645, 0.0257	1.20e+30, 0.012
RCGPANN	0.0215, 0.0164	0.0476, 0.0332	0.025, 0.0049
TPG-CM	**0.0181**, 0.0127	**0.0459, 0.0144**	**0.0182, 0.0038**

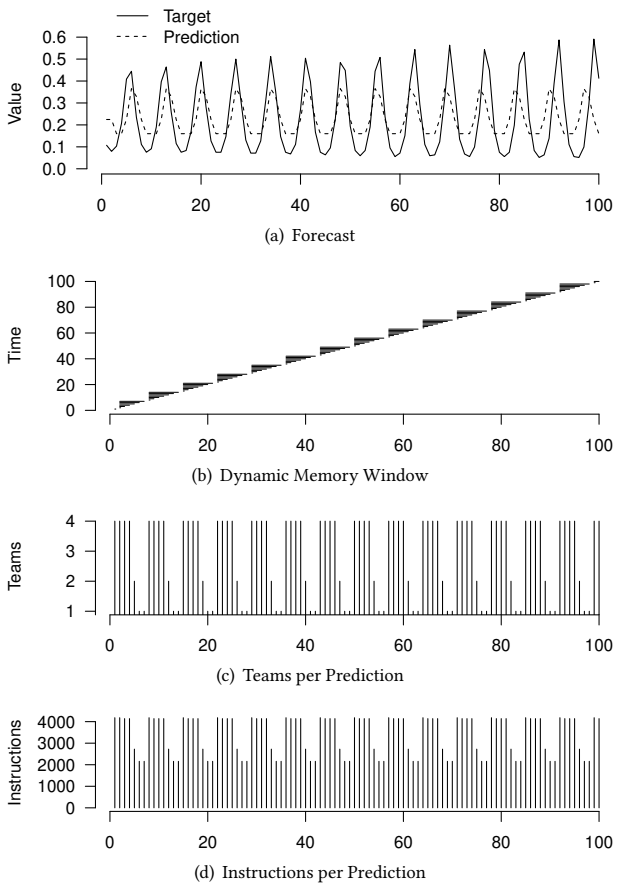


Figure 2: TPG-CM test behaviour for the Laser benchmark. X-axis in all plots indicates timesteps. See Sections 3.2.2 and 3.2.3 for details.

memory registers at t_{i+1}, is used to generate a prediction for t_{i+2}, and so on.

Since program graphs are not provided with an autoregressive state, each program graph must define a mechanism for encoding previous observations within stateful memory registers, and recalling or resetting/overwriting these memories as required. Essentially, each program graph defines an embedding dimension

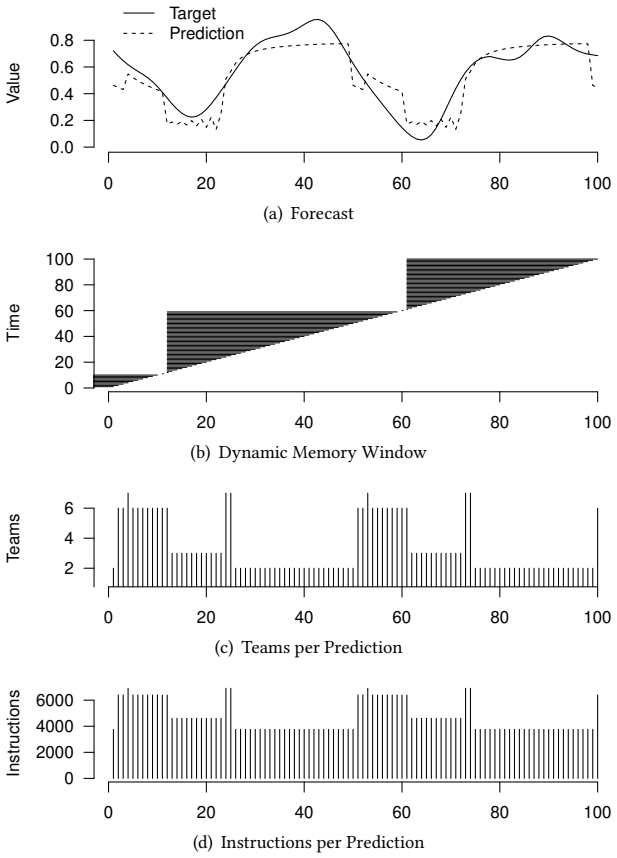


Figure 3: TPG-CM test behaviour for the Mackey-Glass benchmark. X-axis in all plots indicates timesteps. See Sections 3.2.2 and 3.2.3 for details.

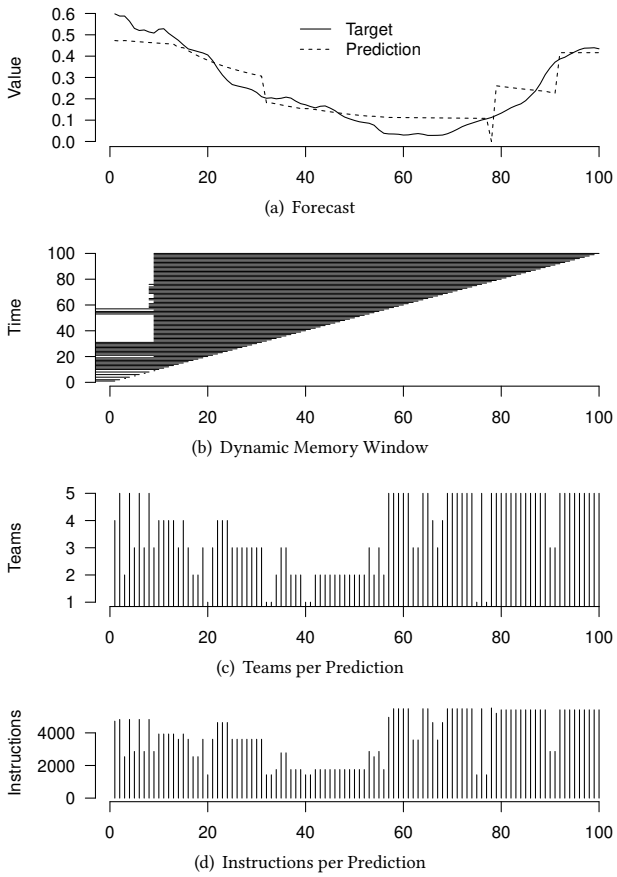


Figure 4: TPG-CM test behaviour for the Sunspots benchmark. X-axis in all plots indicates timesteps. See Sections 3.2.2 and 3.2.3 for details.

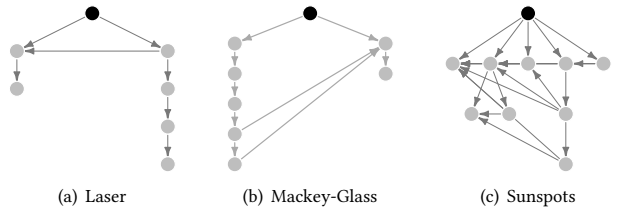


Figure 5: Champion program graphs evolved for each time series benchmark. Each node represents one team of programs. See Section 2 for details.

that is adapted to the characteristics of the particular time series observed during training. Recall from Section 2 that each execution requires traversing one path through the program graph, where each team along the path will read/write to a unique set of stateful memory registers. As the active path changes over time, the solution's embedding dimension also becomes dynamic. In particular, the "age" of memories accessed at any point in time effectively defines a memory window that fluctuates in width over time. The time point at which stateful memory registers are reset or left to accumulate is selected based on the current input as well as the content of stateful memory. Figures 2(b), 3(b), and 4(b) depict the width of these dynamic memory windows at each timestep during test. The memory windows for time t_1 to t_{100} are stacked vertically along the Y-axis. Each horizontal line depicts the window width from the newest memory accessed at time t (right-hand-side) to the oldest memory accessed at time t (left-hand-side). Notice how the TPG-CM champion for each time series exhibits a unique pattern of dynamic memory access. For example, the program graph for the Laser benchmark accumulates and overwrites memory registers at a frequency that clearly reflects the frequency of the target data (compare Figure 2(b) and 2(a)). A similar correlation between the frequency of dynamic memory access and the frequency of target data can be seen for the other benchmarks. Related studies have evolved "observation windows" in time series forecasting, but still required human intuition in order to parameterize the window behaviour [31]. By contrast, our approach is entirely emergent.

3.2.3 Model Complexity. Figure 5 shows the champion program graphs for each time series benchmark. Each node in the graph

represents one team of programs. Every execution of the program graph begins at the root node and follows one path, which may terminate at any node. Furthermore, each team executes a unique subset of programs, each with a variable length list of instructions. Since the path of execution is dynamically selected, the computational complexity of program graph execution is also a dynamic property. Subplots (c) and (d) in Figures 2, 3, and 4 show the runtime complexity for the champion program graph from each time series benchmark. For example, Figure 2(c) indicates that the champion program graph for the Laser benchmark switches between executing 4, 2, and 1 team per prediction at a frequency that correlates with the target data and dynamic memory window in Figures 2(a) and 2(b). The rate of path switching is also evident in the number of instructions executed. For example, the predictions that involve 4 teams result in the execution of roughly 4000 instructions (See Figure 2(d)). This can be correlated with the program graph for the Laser benchmark, Figure 5(a), which has two paths that would result in executing 4 teams. Note that there is a slight difference in the number of instructions executed for some of the 4-team graph traversals during test (Compare Figures 2(c) and 2(d)). This implies that both 4-team paths are used during test.

Dynamic runtime complexity improves the efficiency of model deployment when averaged over many timesteps. This is especially significant as complex (temporal) problems call for increasingly complex models. Note, for example, that the maximum runtime complexity of these program graphs is significantly greater than the complexity of program graphs evolved in visual RL benchmarks, which typically required fewer than 2000 instructions per execution, eg. [13, 14]. However, the policies evolved for visual RL had no temporal memory capability. The shared memory mechanism proposed in this work (Section 2) introduces memory-programs, which come with added computational cost. In particular, an effective instruction in a path-program is any instruction that effects the final value in the edge weight output register, or $R_p[0]$ (See Algorithm 1). However, in memory-programs, instructions that effect the final value of any shared register position in R_s are effective. Thus, memory-programs typically have a larger proportion of effective code. As a result, memory sharing incurs a significant computational cost relative to programs without shared memory, since fewer ineffective instructions, or introns, can be removed prior to program execution.

4 CONCLUSIONS AND FUTURE WORK

TPG has been extended to support continuous output and a modular temporal memory mechanism. We validate the new algorithm, TPG-CM, in a suite of symbolic regression and time series forecasting benchmarks, which represent a broad class of problems for which previous versions of TPG were not applicable. We show that TPG-CM can match the solution quality of current state-of-the-art methods in both domains. In the case of time series forecasting, we emphasize that no prior models of a data stream are necessary since solutions are adapted entirely through environmental interaction.

Future work will provide a more detailed comparative analysis of TPG-CM with state-of-the-art methods in regression and time series prediction benchmarks, and address real-world applications of function regression with large input spaces (e.g. genomic prediction [33]). We are also interested to know how the dynamic properties of TPG-CM will behave in explicitly non-stationary time series environments [1, 31] and dynamic memory tasks in which the input distribution changes significantly from training to test environments [6]. Another open question is how TPG-CM will operate in continuous-action RL (e.g. [24]). Furthermore, it might be possible to support discrete actions and continuous actions simultaneously, as is required in challenging RL benchmarks such as RoboCup Soccer [5]. Finally, given that compositional evolution of program graphs has previously shown promise in multi-task reinforcement learning scenarios [13], the added temporal memory mechanism in TPG-CM might provide further benefit under multi-task time series prediction, where the goal is to build a single model capable of forecasting multiple independent data streams [35]. In short, this work significantly broadens the scope of our existing methods and opens a breadth of future research opportunities.

ACKNOWLEDGEMENTS

S.K. gratefully acknowledges support through the NSERC Postdoctoral Scholarship program. This material is based in part upon work supported by the National Science Foundation under Cooperative Agreement No. DBI-0939454 to the BEACON Center for Evolution in Action at Michigan State University. W.B. acknowledges support from the John R. Koza Endowment fund for part of this work. Michigan State University provided computational resources through the Institute for Cyber-Enabled Research. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

[1] Alexandros Agapitos, Michael O'Neill, and Anthony Brabazon. 2012. Genetic Programming for the Induction of Seasonal Forecasts: A Study on Weather Derivatives. In *Financial Decision Making Using Computational Intelligence*, Michael Doumpos, Constantin Zopounidis, and Panos M. Pardalos (Eds.). Springer US, Boston, MA, 159–188.

[2] Timothy Atkinson, Detlef Plump, and Susan Stepney. 2019. Evolving Graphs with Horizontal Gene Transfer. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO '19)*. Association for Computing Machinery, New York, NY, USA, 968–976.

[3] Andrea Banino, Adria Puigdomenech Badia, Raphael Koster, Martin J. Chadwick, Vinicius Zambaldi, Demis Hassabis, Caswell Barry, Matthew Botvinick, Dharshan Kumaran, and Charles Blundell. 2020. MEMO: A Deep Network for Flexible Combination of Episodic Memories. (2020). arXiv:cs.LG/2001.10913

[4] Markus Brameier and Wolfgang Banzhaf. 2007. *Linear Genetic Programming*. Springer.

[5] Haotian Fu, Hongyao Tang, Jianye Hao, Zihan Lei, Yingfeng Chen, and Changjie Fan. 2019. Deep Multi-Agent Reinforcement Learning with Discrete-Continuous Hybrid Action Spaces. (2019). arXiv:cs.LG/1903.04959

[6] Anirudh Goyal, Alex Lamb, Jordan Hoffmann, Shagun Sodhani, Sergey Levine, Yoshua Bengio, and Bernhard Schölkopf. 2019. Recurrent Independent Mechanisms. (2019). arXiv:cs.LG/1909.10893

[7] Malcolm I. Heywood. 2015. Evolutionary model building under streaming data for classification tasks: opportunities and challenges. *Genetic Programming and Evolvable Machines* 16, 3 (2015), 283–326.

[8] U. Hübner, N. B. Abraham, and C. O. Weiss. 1989. Dimensions and entropies of chaotic intensity pulsations in a single-mode far-infrared NH_3 laser. *Phys. Rev. A* 40 (1989), 6354–6365.

[9] Stephen Kelly. 2018. *Scaling Genetic Programming to Challenging Reinforcement Tasks through Emergent Modularity*. Ph.D. Dissertation. Faculty of Computer Science, Dalhousie University.

[10] Stephen Kelly. 2020. *TPG Source Code*. http://stephenkelly.ca/?q=research.

[11] Stephen Kelly and Wolfgang Banzhaf. 2020. Temporal Memory Sharing in Visual Reinforcement Learning. In *Genetic Programming Theory and Practice XVII*, Wolfgang Banzhaf, Lee Spector, and Leigh Sheneman (Eds.). Springer International Publishing, Cham, 101–119.

[12] Stephen Kelly and Malcolm I. Heywood. 2018. Discovering Agent Behaviors Through Code Reuse: Examples From Half-Field Offense and Ms. Pac-Man. *IEEE Transactions on Games* 10, 2 (June 2018), 195–208.

[13] Stephen Kelly and Malcolm I. Heywood. 2018. Emergent Solutions to High-Dimensional Multitask Reinforcement Learning. *Evolutionary Computation* 26, 3 (2018), 347–380.

[14] Stephen Kelly, Robert J. Smith, and Malcolm I. Heywood. 2019. Emergent Policy Discovery for Visual Reinforcement Learning Through Tangled Program Graphs: A Tutorial. In *Genetic Programming Theory and Practice XVI*, Wolfgang Banzhaf, Lee Spector, and Leigh Sheneman (Eds.). Springer International Publishing, Cham, 37–57.

[15] John F. C. Kingman. 1978. A simple model for the balance between selection and mutation. *Journal of Applied Probability* 15, 1 (1978), 1–12.

[16] James Kirkpatrick, Razvan Pascanu, Neil Rabinowitz, Joel Veness, Guillaume Desjardins, Andrei A. Rusu, Kieran Milan, John Quan, Tiago Ramalho, Agnieszka Grabska-Barwinska, Demis Hassabis, Claudia Clopath, Dharshan Kumaran, and Raia Hadsell. 2017. Overcoming catastrophic forgetting in neural networks. *Proceedings of the National Academy of Sciences* 114, 13 (2017), 3521–3526.

[17] Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. 2015. Continuous control with deep reinforcement learning. (2015). arXiv:cs.LG/1509.02971

[18] Michael C. Mackey and Leon Glass. 1977. Oscillation and chaos in physiological control systems. *Science* 197, 4300 (1977), 287–289.

[19] Luke Metz, Julian Ibarz, Navdeep Jaitly, and James Davidson. 2017. Discrete Sequential Prediction of Continuous Actions for Deep RL. (2017). arXiv:cs.LG/1705.05035

[20] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. 2015. Human-level control through deep reinforcement learning. *Nature* 518, 7540 (2015), 529–533.

[21] Aurora M. Nedelcu and Richard E. Michod. 2002. Evolvability, modularity, and individuality during the transition to multicellularity in volvocalean green algae. In *In Modularity in development and evolution*, Wagner G. Schlosser G. (Ed.). Chicago Press, 470–489.

[22] Miguel Nicolau, Alexandros Agapitos, Michael O'Neill, and Anthony Brabazon. 2015. Guidelines for defining benchmark problems in Genetic Programming. In *2015 IEEE Congress on Evolutionary Computation (CEC)*. 1152–1159.

[23] Richard J. Preen and Larry Bull. 2013. Dynamical Genetic Programming in Xcsf. *Evolutionary Computation* 21, 3 (Sept. 2013), 361–387.

[24] Benjamin Recht. 2019. A Tour of Reinforcement Learning: The View from Continuous Control. *Annual Review of Control, Robotics, and Autonomous Systems* 2, 1 (2019), 253–279.

[25] Herbert A. Simon. 1962. The Architecture of Complexity. *Proceedings of the American Philosophical Society* 106 (01 1962), 467–482.

[26] Robert J. Smith and Malcolm I. Heywood. 2019. Evolving Dota 2 Shadow Fiend Bots Using Genetic Programming with External Memory. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO '19)*. Association for Computing Machinery, New York, NY, USA, 179–187.

[27] Robert J. Smith and Malcolm I. Heywood. 2019. A Model of External Memory for Navigation in Partially Observable Visual Reinforcement Learning Tasks. In *Genetic Programming*, Lukas Sekanina, Ting Hu, Nuno Lourenço, Hendrik Richter, and Pablo García-Sánchez (Eds.). Springer International Publishing, Cham, 162–177.

[28] SIDC Team. 2020 (accessed December, 2019). *World Data Center for the production, preservation and dissemination of the international sunspot number*. http://sidc.be/silso/home.

[29] Andrew James Turner and Julian Francis Miller. 2017. Recurrent Cartesian Genetic Programming of Artificial Neural Networks. *Genetic Programming and Evolvable Machines* 18, 2 (June 2017), 185–212.

[30] Günter P. Wagner and Lee Altenberg. 1996. Perspective: Complex Adaptations and the Evolution of Evolvability. *Evolution* 50, 3 (1996), 967–976.

[31] Neal Wagner, Zbigniew Michalewicz, Moutaz Khouja, and Rob R. McGregor. 2007. Time Series Forecasting for Dynamic Environments: The DyFor Genetic Program Model. *IEEE Transactions on Evolutionary Computation* 11, 4 (Aug 2007), 433–452.

[32] Richard A. Watson and Jordan B. Pollack. 2005. Modular interdependency in complex dynamical systems. *Artificial Life* 11, 4 (2005), 445–457.

[33] Ian Whalen, Wolfgang Banzhaf, Hawlader Abdullah, and Cedric Gondro. 2020. Evolving SNP Panels for Genomic Prediction. In *Evolution in Action: Past, Present and Future - A Festschrift in Honor of Erik D. Goodman*, Wolfgang Banzhaf, Betty H.C. Cheng, Kalyanmoy Deb, Kay E. Holekamp, Richard E. Lenski, Charles Ofria, Robert T. Pennock, William F. Punch, and Danielle J. Whittaker (Eds.). Springer, Cham, Switzerland, 465–485.

[34] Andrew S. Yang. 2001. Modularity, evolvability, and adaptive radiations: a comparison of the hemi- and holometabolous insects. *Evolution and Development* 3, 2 (2001), 59–72.

[35] Maoxin Yang, Qinghua Hu, and Yun Wang. 2019. Multi-task Learning Method for Hierarchical Time Series Forecasting. In *Artificial Neural Networks and Machine Learning – ICANN 2019: Text and Time Series*, Igor V. Tetko, Věra Kůrková, Pavel Karpov, and Fabian Theis (Eds.). Springer International Publishing, Cham, 474–485.