Emergent Tangled Program Graphs in Partially Observable Recursive Forecasting and ViZDoom Navigation Tasks

STEPHEN KELLY, BEACON Center for the Study of Evolution in Action, USA
ROBERT J. SMITH and MALCOLM I. HEYWOOD, Dalhousie University, Canada
WOLFGANG BANZHAF, Michigan State University, USA

Modularity represents a recurring theme in the attempt to scale evolution to the design of complex systems. However, modularity rarely forms the *central* theme of an artificial approach to evolution. In this work, we report on progress with the recently proposed Tangled Program Graph (TPG) framework in which programs are modules. The combination of the TPG representation and its variation operators enable both teams of programs and graphs of teams of programs to appear in an emergent process. The original development of TPG was limited to tasks with, for the most part, complete information. This work details two recent approaches for scaling TPG to tasks that are dominated by partially observable sources of information using different formulations of indexed memory. One formulation emphasizes the incremental construction of memory, again as an emergent process, resulting in a distributed view of state. The second formulation assumes a single global instance of memory and develops it as a communication medium, thus a single global view of state. The resulting empirical evaluation demonstrates that TPG equipped with memory is able to solve multi-task recursive time-series forecasting problems and visual navigation tasks expressed in two levels of a commercial first-person shooter environment.

CCS Concepts: • **Computing methodologies** → **Genetic programming**; *Reinforcement learning;*

Additional Key Words and Phrases: Coevolution, modularity, partial observability, time series

ACM Reference format:
Stephen Kelly, Robert J. Smith, Malcolm I. Heywood, and Wolfgang Banzhaf. 2021. Emergent Tangled Program Graphs in Partially Observable Recursive Forecasting and ViZDoom Navigation Tasks. *ACM Trans. Evol. Learn. Optim* 1, 3, Article 11 (August 2021), 41 pages.
https://doi.org/10.1145/3468857

S. K. gratefully acknowledges support through the NSERC Postdoctoral Scholarship program. R. J. S. and M. I. H. gratefully acknowledge the support of NSERC CRD program under Grant No. 499792-16 and Discovery program under Grant No. 2015-06117. W. B. gratefully acknowledges funding from the BEACON Center for the Study of Evolution in Action and from the John R. Koza endowment to Michigan State University (MSU). This material is based in part upon work supported by the National Science Foundation under Cooperative Agreement No. DBI-0939454. 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. MSU provided computational resources through the Institute for Cyber-Enabled Research. Additional support provided by ACENET, Calcul Quebec, Compute Ontario and WestGrid, and Compute Canada (www.computecanada.ca).

Authors' addresses: S. Kelly, BEACON Center for the Study of Evolution in Action, East Lansing, Michigan, USA; email: kellys27@msu.edu; R. J. Smith and M. I. Heywood, Dalhousie University, Halifax, Nova Scotia, Canada; emails: {rsmith, mheywood}@cs.dal.ca; W. Banzhaf, Michigan State University, East Lansing, Michigan, USA; email: banzhafw@cse.msu.edu.

2688-299X/2021/08-ART11 $15.00
https://doi.org/10.1145/3468857

1 INTRODUCTION

Modularity is a widespread structural principle in both natural and artificial systems. In artificial systems [Simon 2019], it is a built-in design feature that simplifies the task of designing the system, understanding the resulting mechanisms, guaranteeing the targeted functionality, and improving and maintaining a so realized system [Clark and Baldwin 2000]. In computer technology, in particular, both in hardware and software, modular approaches are prevalent. It allows easy replacement of failing parts as well as easy debugging of failing code segments [Parnas 1972] and has long been a staple in software engineering. In natural systems under evolution, modularity is a result of evolution, and thus an emergent property of the systems under consideration.

A rough general definition of modularity is offered by Callebaut [2005, p.6] when he says "...a system may be characterized as modular to the extent that each of its components operates primarily according to its own, intrinsically determined principles." The literature is full of more technical elaborations, which require that components of a module are tightly integrated but relatively independent from those of other modules. Modularity, therefore, is a matter of degree and can be seen as a gradual property of systems [Wagner and Altenberg 1996]. It is argued that they provide efficiency gains for evolvable systems.

Modularity is often associated with hierarchical organization, but modularity does not imply hierarchy. Again, Callebaut comments on one of the examples given in Simon [2019], rooms connected by corridors. While rooms connected by corridors can be considered modules, there is no notion of hierarchy involved, at least as long as we do not speak of them as being part of a building. However, this example argues for something else: Repeated patterns can often be identified as some kind of modules (in this case the "room" pattern versus the "corridor" pattern). So repetitive patterns lay the groundwork for modules, in that patterns stand out from the background (their components interact more strongly with each other than with the background). In an analysis of computer code, a similar observation can be made about code segments that are used repeatedly throughout a program. Genetic programming systems have shown a tendency to produce via evolution emergent repetitive patterns [Langdon and Banzhaf 2005, 2008]. In the present work, modules are synonymous with programs that appear in multiple places throughout a genotype.

What do we mean, then, when we speak of a hierarchy? The notion of dependency or order/level is key to understanding the concept of a hierarchy. A system that is organized in a way that its components can be ranked in some kind of order taking the form of levels (dependency, membership of different levels), is a hierarchical system. Hierarchical systems are not per se more efficient than others, they are just "ordered." But we can see that hierarchical organization ("order") in connection with modular organization ("repetitive structures") allows efficiency gains. This is the core idea of the many-to-one settings we often encounter in hierarchical modular systems.

So whether hierarchy *implies* modularity is not clear, but arguments have been made to the effect that hierarchy requires modularity [Simon 2019], at least in natural and artificial contexts with efficiency pressures. For example, in Biology, the spectacular experiments of Walter J. Gehring on *Drosophila Melanogaster* identified the so-called homeobox genes [Gehring 1992] and deciphered the function of the *pax6* gene [Gehring and Ikeo 1999] as a master gene in the formation of organs like the eye and the antenna of *D. melanogaster*. These are famous examples of the hierarchical modular design resulting from natural evolution.

In genetic programming, both modularity and hierarchy play important roles. The classical attempt to make use of modules has become known as automatically defined functions (ADFs for short) [Koza 1994], which exploit the fact that some pieces of code are expected to be reused and should therefore be separated as modules that can be repeatedly called (hierarchically ordered according to call sequence). This is not an emergent process, though, as the ADFs are defined before

a GP run starts, and a number of parameters have to be fixed (like number of arguments, etc), possibly before knowing what would be adequate in the problem setting. Encapsulation [Koza 1992] and module acquisition [Angeline 1994], however, allow the process of evolution to produce emergent modules. Results at the time were mixed, though. Koza addressed the missing potential for evolution by adding architecture-altering operations, allowing modules to be subject to more influence in evolution [Koza et al. 1999]. Modularity and hierarchy in GP have been discussed in other work [Banzhaf et al. 1999; Dostál 2013; Gerules and Janikow 2016; Woodward 2003], but the notion of emergent modularity/hierarchy, i.e., the notion that it has to evolve in the course of the GP run has not been studied sufficiently yet [O'Neill et al. 2010].

In natural systems, hierarchy often emerges in tandem with modularity able to make efficient use of the separation of tasks provided by modules. Here, the transition to a module from elements on a certain level is just repeated again and again recursively, by using lower level modules as elements for higher level modules. Simon's idea of the dominance of nearly decomposable systems [Simon 1962] argues for an increase in the efficiency of natural and artificial processes to generate complex systems, and an increase in the speed of their emergence [Simon 2002]. Simon's famous argument of "The Two Watchmakers" has been a vivid demonstration of this ability of modular and hierarchical systems. This property is central to the approach adopted in this work, where the resulting emergent organization of multiple programs into teams represents the first level of a hierarchy. However, as individuals experience more of a task, teams can either incorporate additional programs or identify conditions under which to pass execution over to other teams; resulting in the emergent organization of graphs of teams of programs.

The evolutionary advantages of modular, hierarchical systems are clear [Callebaut 2005]. The speed of their emergence in an otherwise noisy environment, the robustness of their response to internal and external perturbations and the potential for combinatorial innovation are the three main effects of such a design. While in artificial systems it is rational consideration that leads to a realization of such principles, in natural and emergent systems, it is selection that acts as the engine of emergence. Even artificial systems, like genetic and evolutionary computation algorithms can make use of this engine of emergence, as discussed in more detail in Banzhaf [2014] for the case of genetic programming.

Modularity also directly supports the scaling of phenotypes to environments that change in systematic ways [Kashtan et al. 2007; Parter et al. 2008]. For example, as an agent becomes more adept at surviving within a local environment, it has the potential to explore more, thus expanding its ability to model the environment more accurately. Addressing the new challenges does not require a total redesign of a current genotype. Instead, specific modules might need retuning and/or new modules might need to be introduced. In this way, the effectiveness of modular representations has been demonstrated under streaming data applications with shift and drift [Vahdat et al. 2015].

Tangled Program Graphs (TPG) is a framework for **genetic programming (GP)** that has leveraged *emergent* modularity in building solutions to high-dimensional (visual) reinforcement learning, e.g., Kelly and Heywood [2018b]; Kelly et al. [2019]. In TPG, a module is a program and a complete solution is composed of multiple programs interconnected within a hierarchical graph structure, or *program graph*. Two emergent properties are critical to the algorithm's success: (1) The identification of specialized, reusable modules that distinguish between context and action; and (2) The hierarchical nature of the interdependence among multiple modules within a composite solution. For example, given a problem space, or state $\vec{s}$, individual programs are free to evolve specialized behaviour in which they process only a small subset of the variables in $\vec{s}$. When multiple programs are coevolved within a graph, their specialized sensitivity to particular inputs allows the structure as a whole to develop an efficient *spatial* decomposition such that only salient variables from $\vec{s}$ are processed. Furthermore, since we are particularly interested in temporal problems

in which the input space, or environmental state, changes over time, emergent module interdependence would imply that a program graph discovers how many programs should interact to make each prediction in a sequence, in what way those programs should interact, and to what degree the subset of programs interacting should change as a function of the environmental state at each time point, $\vec{s}(t)$.

In addition, the problem environments under which our system should be capable of operating are partially observable. Thus, the state at any point in time, $\vec{s}(t)$, does not completely describe the conditions of the environment. Such environments are widely encountered in practice. For example, robots, drones or unmanned vehicles generally have limited sensor information, and are deployed in environments that change over time. These systems rely on some form of temporal memory to store salient properties of the environment such that a more complete estimation of state can be made in the future. Moreover, as the dimension of the state space increases, the computational cost of having agents make decisions based on all the state information also increases, impacting the scalability and efficiency of agents. Conversely, if the agent can establish *simpler internal models* based on lower dimensional representations, then the resulting policies will be more efficient, reducing the cost of deployment and improving scalability. In short, operation under partially observable environments will not be possible unless the agent can construct an internal model of state using memory $\vec{m}(t)$. Finally, to support operation in dynamic environments, it is critical that the memory mechanism supports dynamic access. That is, the solution must be capable of reading and writing to distinct memory locations at any moment, based on the state of the environment. Thus, just as a modular structure supports a read-only *spatial* decomposition of its world view $\vec{s}(t)$, this work establishes how modular agents are capable of a read-write *temporal* problem decomposition in $\vec{m}(t)$. As a consequence, the agent can answer questions such as how to encode information for later recall, identify what to remember, and determine when to remember what. At any point in time, the agent is free to access multiple encoded memories with distinct time delays, and integrate this information to predict the best output for the current situation. This is distinct from recursive structures that allow an agent to perform some memory tasks by accumulating/integrating experience over time through feedback, e.g., Conrads et al. [1998]; Nordin et al. [1998]. We note that a purely recursive agent does not directly support temporal decomposition, because it is limited to a fixed time delay. Such a feedback structure is equivalent to static memory access, for example, if the agent were limited to accessing the same memory register(s) at every point in time regardless of the state of the world.

In summary, the objective of this work is to extend the scope of emergent modularity in TPGs to address time-dependent problems that explicitly require the agent to make use of temporal memory. The significance of this development is twofold: (1) Temporal memory allows program graphs to integrate experience over time, which is required for operation in partially-observable environments; and (2) Temporal problem decomposition is likely beneficial in dynamic, non-stationary environments. The temporal problems we have in mind include time-series forecasting or streaming data classification tasks when the underlying process generating the data stream changes significantly over time [Agapitos et al. 2012; Heywood 2015], and visual **reinforcement learning (RL)** environments in which the physics of the world and/or task objectives change over time, for example, in video games as the player navigates complex environments without access to a global world view or map.

These applications are assumed to represent two distinct design requirements for the memory models. The non-stationary time-series prediction task is taken to require support for the very precise indexing of previously encountered events, as in the configuration of a chaotic time series in which changing a single value may completely change the properties of the task. Conversely, state information in visual RL environments are generally high-dimensional (i.e., pixels from a

frame of video) and both temporally and spatially redundant. That is to say, there is a significant amount of correlation between what pixels represent in the local (spatial) neighbourhood of the same frame and between consecutive frames. This motivates an approach to memory in which large numbers of programs (modules) need to operate collectively (i.e., multiple modules are necessary to efficiently sample different parts of high-dimensional spaces). Memory therefore represents a communication medium (for hundreds of modules) as well as a repository of internal state. Under such a setting, we formulate writing to memory as a probabilistic process such that long- and short-term memory is supported. Read operations can then be indexed to make use of these dependencies.

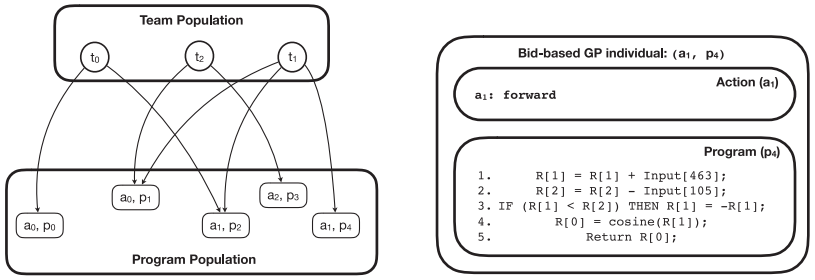
The remainder of this article is organized as follows: Section 2 provides a detailed description of how program graphs are represented, evolved, and evaluated in the TPG algorithm. This section finishes by placing the TPG approach into the wider context of other approaches for supporting modularity in GP. Sections 3 and 4 each propose unique ways to augment generic TPG with dynamic memory. Relevant background and an empirical evaluation are provided in each case. Specifically, Section 3 proposes a *structural* memory model that supports dynamic memory management through module interdependence within the program graph. An empirical evaluation in a non-stationary time-series prediction task is provided. The task is univariate (i.e., low-dimensional) but highly dynamic and sensitive to the chaotic process generating the data. Furthermore, no sliding window of previous values, or *autoregressive* state, is provided to the agent, implying that the task is only partially observable. As such, Section 3 establishes how TPG with dynamic memory exploits temporal problem decomposition to cope with a partially observable, non-stationary environment.

Section 4 introduces a *probabilistic* memory model and evaluates how TPG manages temporal *and* spacial decomposition in a dynamic, partially observable, and high-dimensional visual RL environment, VizDoom [Wydmuch et al. 2019]. That is to say, recent advances in visual reinforcement learning (a state is defined in terms of video data alone) has demonstrated that end-to-end navigational behaviours can be spontaneously discovered, albeit only after engineering a specific formulation of a deep-learning framework with multiple forms of memory [Jaderberg et al. 2019]. We demonstrate in this work for the first time the ability to evolve a policy that learns how to solve levels from the original Doom game engine. This is challenging, because a level consists of multiple rooms and corridors but is only experienced from a first-person perspective, while also requiring an agent to collect keys and associate the operation of switches with enabling additional parts of the environment. In short, an agent has to navigate a world, solve puzzles, and maintain health to complete levels.

2 TANGLED PROGRAM GRAPHS

To explicitly support the emergent discovery of modular structures, we assume a two-population symbiotic model for evolution in which teams and programs are represented by independent populations and coevolved (Figure 1). Symbiosis implies a hierarchical relation between the two populations [Heywood and Lichodzijewski 2010], with members of the team population defined in terms of individuals from the program population. The team population performs a (combinatorial) search for the relevant composition of a team under a variable length representation, i.e., no prior assumption regarding the number of programs per team. In this work, *task-specific fitness* is only ever defined at the level of the team (Section 2.2). After ranking the teams, and deleting the worst performing (Steps 1c and 1d, Algorithm 1), any programs that are not associated with a surviving team are also deleted (Step 1e).[1] Finally, variation operators are applied first to teams

[1]Implies that a variable number of programs might be deleted per generation.



(a) Symbiotic relationship (b) Bid-based GP individual

Fig. 1. Symbiotic coevolutionary relationship between Team and (Bid-based) Program Populations. 1(a) Each individual from the Team Population (t_n) defines a unique sample of individuals from the Program Population. The same bid-based GP program can appear in multiple programs, but no more than once per team. 1(b) Each bid-based GP individual is defined by a discrete terminal, *atomic* action (a_i) and a program (p_k). The same program (action) can appear in multiple bid-based GP individuals, but the combination of program and action must be unique. A Linear GP representation is assumed [Brameier and Banzhaf 2007], where `R[i]` is a reference to register *i*, `Input[i]` indicates a read only reference of index *i* of the vector of state input $\vec{s}(t)$. Generic instruction set supports arithmetic operators, cosine, (protected) logarithm, (protected) exponential, and conditional [Lichodzijewski and Heywood 2007; Wu and Banzhaf 2011].

(Steps 1(h)i to 1(h)iv) and then to programs (Step 1(h)v) as is selection at multiple levels (e.g., Wu and Banzhaf [2011]). The overall process is elitist, with the best $1 - Gap$ teams surviving, and only this subset of teams representing the parent pool. Children therefore have to be competitive with this elite to survive the next generation.

The output from a team is defined by assuming a bid-based approach [Lichodzijewski and Heywood 2007] in which each program defines an action and a context. Actions, a_i, are limited to a single scalar value whereas contexts are defined by a program, p_j (Figure 1(b)). At initialization all actions are terminal. For example, in an RL task actions are initially limited to the set of discrete terminal atomic actions specific to the task domain, $a_i \in \mathcal{A}$ (e.g., enumeration of joystick position into eight discrete directions). Given the current state of the environment, $\vec{s}(t)$, all the programs from the same team are executed, and the *single* program with the maximum output "wins" the right to suggest its action. Naturally, the bid-based approach makes the decomposition explicit and simplifies credit assignment, i.e., variation can be made specific to a single program–action pairing, thus credit/blame can also be attributed to specific parts of a team. Programs are therefore synonymous with modules; thus, a rather different form of modularity than 'classically' adopted within the context of genetic programming, i.e., Automatically Defined Functions [Koza 1994] or Macros [Spector and Luke 1996].

The *minimal composition* of a team is two programs with different actions, thus the same program–action pairing $\langle a_i, p_k \rangle$ can appear in multiple teams. Moreover, different programs can appear in the same team with the same action, as long as there are at least two different actions within the same team; see Figure 1(a). Variation operators modify structures at the team and then program level after first cloning the corresponding parent(s). Moreover, new programs are only introduced relative to a cloned parent program and team, implying that the original use of any program (by other teams) is unchanged.

Crossover is only defined in terms of a team, not programs (Step 1(h)i, Algorithm 1). Specifically, for two cloned parents selected with uniform probability from *NewAgents*, any program common to both parents is copied to the child. Any remaining programs are copied to the child with a fifty

ALGORITHM 1: Generic evolutionary cycle. *AgentList* is the set of teams that are retained post fitness ranking. *NewAgents* are the subset of teams that are first cloned (Step 1g) and then available for variation. Variation is also applied to members of the program population and also requires the target program to be first cloned (Step 1(h)vB).

- Initialize $Team(0)$
- Initialize $Prog(0)$

(1) For ($g = 0$; !EndOfEvolution; $g = g + 1$) ▹ Generation loop
  (a) $AgentList = \emptyset$
  (b) For all (team $\in Team(g)$) AND (team = root) ▹ Identify valid agent
    (i) agent = team;
    (ii) update(*AgentList*, team)
    (iii) For all (evaluations) ▹ Evaluation loop (Section 2.2)
      (A) Evaluate(agent)
      (B) update(agent.Fitness)
  (c) Rank(agents $\in$ *AgentList*) ▹ Select parents
  (d) Prune worst ranked *Gap* agents in *AgentList* and corresponding teams in $Team(g)$
  (e) Prune all $prog \in Prog(g)$ without a team
  (f) Select (Parents $\in$ *AgentList*)
  (g) Clone (*NewAgents*, Parents)
  (h) DO ▹ Variation
    (i) IF (rnd$< p_x$) THEN agent = Xover (*NewAgents*) ▹ Team crossover
    (ii) ELSE agent $\in$ *NewAgents*
    (iii) newRoot = DeleteProgFromTeam(root $\in$ agent, p_d) ▹ Team mutation
    (iv) newRoot = AddProgToTeam(newRoot, p_a)
    (v) IF (ModifyProgram(p_m)) THEN ▹ Program variation
      (A) Select(prog $\in$ newRoot)
      (B) Clone (newProg, prog)
      (C) ModInstr(newProg, $p_{del}, p_{add}, p_{mut}, p_{swp}$)
      (D) ModAction(newProg, p_{mn}, p_{atomic}) ▹ Action variation (Section 2.1)
    (vi) Insert(newRoot, *AgentList*) ▹ New team and progs added
  (i) WHILE count($newRoot$) $< Gap$ AND $|AgentList| < R_{size}$

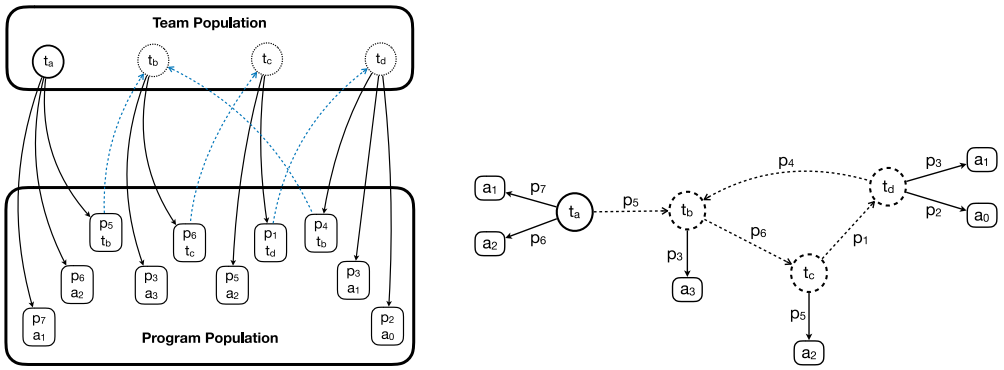
percent probability. The threshold p_x is the probability of applying crossover. Child teams can also have programs deleted or added with probabilities p_d and p_a, respectively.[2]

Variation of programs is also subject to a cloning step (Step 1(h)vB), ensuring that any other team using the same program is unaffected. Instructions can then be probabilistically deleted, added, mutated or swapped (Step 1(h)vC). Finally, the action of a program can also be modified. Depending on how this is performed, actions can remain terminal or can become pointers to other teams (detailed in Section 2.1), resulting in the emergence of complex graph structures, or Tangled Program Graphs.

2.1 Emergence of Tangled Program Graph Structures

Figure 1(a) illustrates the symbiotic relationship between team and program populations when actions are limited to the set of terminal atomic actions, $a_i \in \mathcal{A}$. This means that

[2]Subject to the above minimal team composition constraint.



(a) Symbiotic relationship (b) TPG Agent

Fig. 2. TPG representation from the perspective of 2(a) the underlying two population (symbiotic) coevolutionary model and 2(b) equivalent phenotype (ignoring the details of each program, p_k). The *root team* is identified by the bold circle whereas teams archived within the TPG graph are identified by the dashed circles. At initialization all teams are root teams; thereafter, an increasing number of teams can be archived. However, the same team can be archived by multiple root teams. Bold arcs represent programs associated with terminal atomic actions, dashed arcs represent programs defining an action in terms of another team.

co-operation/decomposition only appears at a single level, that of single teams. Conversely, as different aspects of a task are experienced, different specializations might appear across different teams. TPG attempts to discover more general policies by taking the condition under which a terminal atomic action was applied by team "i" and instead referencing team "j." If team "j" represents a better terminal action selector for the condition identified by team "i," then the new more complex policy will survive. Indeed, given state $\vec{s}(t)$ from the task environment, such a process might result in team "i" deferring to team "j," which itself defers to team "k" before a specific terminal atomic action is identified. TPG represents a continuous emergent process through which the structure associated with deferring judgement to other teams is discovered.

The key to the TPG approach is to let action mutation (Step 1(h)vD, Algorithm 1)[3] choose between selecting a terminal atomic action ($a_i \in \mathcal{A}$) or a pointer to a team in the team population ($a_i \in \mathcal{T}$). Figure 2 illustrates how a TPG agent might express a multi-team relationship in terms of the interconnection between individuals in the team and program populations, and the resulting TPG agent. The team population now has two types of teams, those that are pointed to, and those that are not. Only teams that have no incoming arcs (i.e., not pointed to) denote a *root team*. This is important, because TPG agent evaluation is only performed relative to the root teams (Section 2.2), thus only root teams represent eligible parents. Note that when cloning (Step 1g), it is therefore only the root team that is cloned and variation operators applied to.

2.2 Evaluating a TPG Agent

Assuming the TPG agent from Figure 2(b) and a current state from the environment $\vec{s}(t)$. Execution always begins relative to the TPG agent's root team (Figure 3(a)). In this example, all programs in team t_a are therefore executed and the program with the maximum `R[0]` register value returned. For example, $\arg\max(p_k \in t_a) \rightarrow p_7$. As p_7 is associated with terminal atomic action a_1, this would complete the evaluation of the TPG agent for state $\vec{s}(t)$. If, however, p_5 was the winning

[3]Threshold p_{nm} is the probability of action mutation and p_{atomic} (versus $1 - p_{atomic}$) the probability of terminal atomic action (versus team) selection.

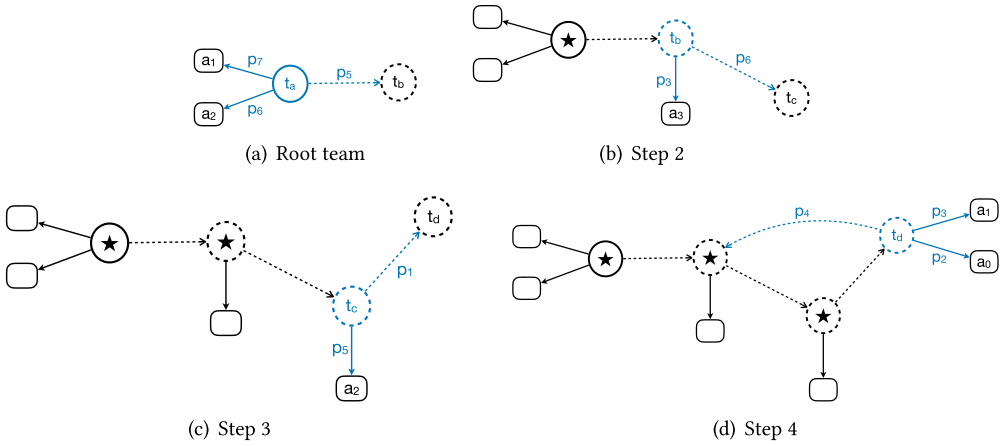


Fig. 3. Determining the action of an example TPG agent. 3(a) The *root team* is indicated by the bold circle and defines the first team to be evaluated under a new environmental state $\vec{s}(t)$. Subsequent evaluation of TPG teams is a function of action selection, with evaluation complete when a terminal atomic action is recommended. Each team evaluated is marked (the "star" in this figure). Should an action be recommended that identifies a team marked with a star, then a loop has been detected, 3(d). This forces the use of a *runner up* arc (i.e., program). Each team is guaranteed to have at least one arc terminating in an atomic action, implying that any loop can be resolved in terms of a terminal atomic action.

program from team t_a, then the corresponding action is actually another team, t_b. Before moving on to evaluate t_b, the parent team is marked (illustrated by the "star" in Figure 3(b)). The process of program execution is now repeated relative to t_b as per Figure 3(b).

Steps 2 through 4 of Figure 3 illustrate this process for a "worst-case" scenario, i.e., each team of the TPG agent are visited and at team t_d program p_4 had the maximum `R[0]` register value. At this point a loop is detected, because the action from program p_4 is a previously visited team (Figure 3(d)). Program p_4's contribution can now be ignored and the program with the next largest `R[0]` register value used to identify the path from team t_d. Thus, as long as there is at least one arc terminating in a terminal atomic action at any team, then loops can always avoided.[4] After finding a terminal atomic action for the current state, any marked teams are reset, and execution commences relative to the next state from the root node.

2.3 Discussion

The underlying motivation for TPG is to scale GP to tasks through an emergent model of modularity. However, modularity has been a reoccurring theme in GP. **Automatically Defined Functions (ADF)** [Koza 1994] or **Automatically Defined Macros (ADM)** [Spector and Luke 1996] introduce modularity through prior parameterization of what constitutes a function or macro. To do so, prior decisions need to be made to define the number of functions or macros supported and the number of arguments passed. This potentially leads to a requirement to perform runs over multiple parameterizations before adopting one that appears to work. This may lead to a limitation to the types of task that GP can potentially be applied to. These issues do not appear in TPG, because modularity is entirely emergent.

[4]Such a constraint is enforced during action mutation (Step 1(h)vD, Algorithm 1) to guarantee that any agent evaluates to an terminal atomic action.

Two examples of GP formulations that do explicitly support open ended modularity are architecture-altering operations associated with the **Genetic Programming Problem Solver (GPPS)** [Koza et al. 1999] and the "Adaptive Representations through Learning" framework of [Rosca and Ballard 1996]. The architecture-altering operations of GPPS represent a family of operations that manipulate subroutines, such as duplication, creation, and deletion. They are deployed in conjunction with a "structure preserving crossover" operation, itself based on the concept of point-typing (Section 5 in Koza et al. [1999]). Although capable of emergent discovery of modularity, the principle limitation of the approach appears to be computational.[5] For example, applying GPPS to the Two Spirals classification task required a population of 700,000 individuals executed over 500 generations, with nine different types of "max argument" definitions for appropriately limiting the properties of the modules (Section 23.2 in Koza et al. [1999]). A path for scaling GPPS from a two input classification task to RL tasks with tens of thousands of inputs is therefore not obvious.

Adaptive Representations through Learning (ARL) is based on the "bottom-up evolution hypothesis" [Rosca and Ballard 1996], which maintains that the discovery of ADF like subroutines become "stable" in a bottom-up fashion as evolution commences. The guiding principle of adaptive representations was that useful module/subroutines can only be discovered through the definition of suitable measures of performance. However, the design of subroutine-specific versus overall fitness limits the application of the approach. ARL moved beyond the constraint of supplying appropriate measures of subroutine performance by assuming that modules only be sourced from offspring with the highest fitness improvement relative to their parent [Rosca and Ballard 1996]. This presents something of a problem with regards to RL tasks with fitness plateaus or non-stationary environments. ARL also makes use of diversity measures to establish when to create subroutines and limits the number of subroutines to some *a priori* limit.

The Push language emphasized a capacity for developing modular solutions [Spector and Robinson 2002]. To do so, a specific language class is introduced to make calls to encapsulated code (e.g., the `CODE` class). A case study was then developed around the even-parity task to illustrate the ability of Push to discover solutions efficiently. However, it was also remarked that "the program is clearly re-using code, which is one of the hallmarks of modularity. On the other hand, this is probably not the sort of modularity that any human would employ." Moreover, solving each instance of the even-parity task required an independent cycle of evolution and there is no independent assessment of post training performance (all data appears during training). Conversely, Huelsbergen demonstrates that it is not so much modularity that is significant in solving *all* even-parity problems, but recursion [Huelsbergen 1998]. Later work introduced modularity through "tagging," where this was benchmarked in simple grid worlds (lawnmower and obstacle-avoidance) when used with Push [Spector et al. 2011], but appear to translate to tree structured GP [Spector et al. 2012]. We are not aware of work demonstrating the use of memory in Push.

Cartesian genetic programming (CGP) has been developed to include memory (through recursion [Turner and Miller 2017]) and modularity [Walker and Miller 2008]. As such the study conducted in Section 3 will use previous results for CGP for comparative purposes under single sequence time-series forecasting tasks before demonstrating that TPG can generalize the result to multi-sequence time-series forecasting. Modularity in CGP was again demonstrated using bit-wise tasks with no independent test of generalization (e.g., parity, adder, comparator) as well as the lawnmower and symbolic regression problems. Modularity was useful under bit-wise tasks, but not so useful under lawnmower and symbolic regression. We are not aware of attempts to

[5]See commentary in Section 6.1.2 of the Field Guide to Genetic Programming [Poli et al. 2008].

combine mechanisms for both memory and modularity, where both mechanisms will be necessary for scaling genetic programming to the tasks appearing in Sections 3 and 4.

Modularity has also been investigated within the context of Grammatical Evolution (e.g., Harper and Blair [2006] and Swafford et al. [2011]). Harper and Blair introduce "dynamically defined functions," which expand the grammar definition to support function calls with the ability to evolve the number of parameters passed. The minesweeper grid world was used to demonstrate the capability of the approach, albeit without an independent post training evaluation (no variation in mines or start condition). Swafford et al. [2011] develop an approach based on the encapsulation of sub-derivition trees in which the provision of a suitable repair operator was demonstrated to be of particular importance. Benchmarking assumed various simple grid world tasks (Santa Fe Trail, lawnmower) and symbolic regression. Again, we are not aware of attempts to scale modularity beyond these benchmarks to tasks requiring both modularity and memory mechanisms.

TPG itself represents a development from "hierarchical" **symbiotic bid-based (SBB)** GP [Kelly and Heywood 2018a]. The approach assumed two phases. In phase 1, multiple SBB populations were evolved on different versions of the ultimate task. The action set always takes the form of terminal atomic actions. Diversity measures are used to encourage the development of a range of agent policies. In phase 2, a final independent SBB population is evolved in which the action set is defined solely in terms of pointers to the teams as evolved in phase 1. Thus, in phase 2 SBB learns under what conditions to switch between some subset of the policies evolved in phase 1. Naturally, hierarchical SBB is not an emergent form of modularity, but it is agnostic to the representation assumed by GP. Conversely, both ADFs and ARL are specific to the concept of modularity through subroutines under a tree structured representation.

Multi-trees are also agnostic to the specifics of the GP representation, but rely on prior knowledge regarding the number of modules. That is to say, a multi-tree defines an ensemble of some prior number of subtrees common to all individuals in the population. Thus, in classification problems the number of ensemble members is set by the number of classes [An et al. 2013; Muni et al. 2004]. The concept of multi-trees has similarities with GP teams/ensembles [Brameier and Banzhaf 2001]. Prior knowledge is again necessary regarding the number of result producing programs/modules (per team). For example, in classification tasks this is often synonymous with the number of classes (a four-class classification problem would have four programs associated with each team) [Thomason and Soule 2007]. Furthermore, it might be necessary to define fitness at the level of program as well as team [Thomason and Soule 2007; Wu and Banzhaf 2011]. These issues also appear in the case of modularity under the Potter–DeJong formulation of co-operative coevolution [Potter and De Jong 2000] (as opposed to the variable length symbiotic model assumed by TPG and SBB). Each gene comprising a solution represents a module[6] in a fixed length genotype and the content for each solution gene is sourced from an independent population. The context for each solution gene is therefore explicit, i.e., each population only ever defines modules for a specific solution gene location. However, decisions need to be made regarding how to sample the populations and how to propagate fitness back to each module's independent population (given that fitness is only evaluated when all modules are defined). This can lead to issues about sample effectiveness in addition to requiring prior knowledge regarding the common number of modules that must appear in all solutions. Multi-trees have the same prior (regarding modules per solution), but at least do not require fitness to be defined at the level of independent modules.

In summary, TPG provides the only example of genetic programming that embraces emergent modularity as the defining principle. To demonstrate the utility of this property, TPG was originally benchmarked on the ALE suite of Atari reinforcement learning tasks [Kelly and Heywood

[6]Equivalent to a program under GP or a weight / neural network in the case of neuro-evolution [Gomez et al. 2008].

2017]. Under this setting, TPG matched the performance of the deep-learning architectures, such as DQN, and surpassed the performance of HyperNEAT (a well-known neuro-evolutionary approach), while discovering solutions that were several orders of magnitude simpler. In addition, TPG was able to learn policies for playing five Atari titles simultaneously [Kelly and Heywood 2018b]. To date, the only genetic programming approaches to attempt to play titles from the ALE benchmark only did so by making use of specialized image processing instructions as opposed to composing solutions through modularity [Jia and Ebner 2017; Wilson et al. 2018].

3 BENCHMARKING OF A STRUCTURAL MEMORY MODEL IN MULTI-TASK RECURSIVE TIME-SERIES FORECASTING

A time series is a sequence of measurements or observations in time. The goal of time-series forecasting is to predict unseen future values based on previously observed values. This has applications in many important real-world problems. For example, forecasting crop yields from year to year, forecasting the demand on energy utilities, or forecasting an EEG trace to anticipate changes in patient health. Most statistical and machine learning methods rely on prior inspection of the time-series data to parameterize the prediction model using heuristics and/or human intuition [Agapitos et al. 2012; Turner and Miller 2017]. However, this assumes that enough prior data is available to estimate the characteristics of the time series. Furthermore, applying static model parameters assumes that the underlying process generating the time series is stationary. In reality, many real-world forecasting environments change significantly over time [Wagner et al. 2007].

In this section, we propose a structural memory framework that incrementally builds the prediction machine entirely through interaction with a non-stationary environment. In doing so, we eliminate the need to hard-code any recursive structure into the machine or to pre-specify a fixed-size sliding-window of historical values to analyze at any point in time. The resulting framework is exceedingly general and is likely to have broad applications in memory-intensive problem environments with non-stationary properties, e.g., Goyal et al. [2019]; Wagner et al. [2007].

In an initial study [Kelly et al. 2020], we evaluated the structural memory model described in this section in three challenging forecasting benchmarks; Sunspots [SIDC 2019], Mackey-Glass [Mackey and Glass 1977], and Laser [Hübner et al. 1989]. The Sunspots and Laser datasets are obtained from real-world recordings, while Mackey-Glass is a chaotic series generated from a parameterized equation. TPG with emergent structural memory was found to generally match the quality of state-of-the-art solutions in all three benchmark datasets. This provides a critical first step toward the significantly more challenging task investigated here, namely, *multi-task* recursive time-series forecasting. In multi-task forecasting, the goal is to build a single agent capable of forecasting multiple independent data streams. In earlier work, we establish how hierarchical team-based GP (a precursor to TPG) supports inter-task transfer learning when multiple related tasks are learned together [Kelly and Heywood 2018a]. Here, we selected three *unrelated* time-series datasets to test how a TPG agent could exploit its modular/hierarchical structure to automatically decompose a multi-faceted problem. Sections 3.4.4 and 3.4.5 provide a detailed discussion of the hierarchical task decomposition that emerges in our empirical evaluation of TPG with the structural memory model. In addition to testing the power of emergent modularity and hierarchy, this problem is a suitable testbed for dynamic memory models for the following reasons:

- **No autoregressive state.** In typical forecasting methods, the forecaster is fed individual samples in order from series $\vec{s}()$ and, given sample $\vec{s}(t)$, must predict the value of $\vec{s}(t+1)$. It is common to pack a sequence of previous values into a single *autoregressive* state representation. For example, one could define an embedding dimension D and a time delay T to predefine a sliding window of prior observations to present to the forecaster at each timestep. If

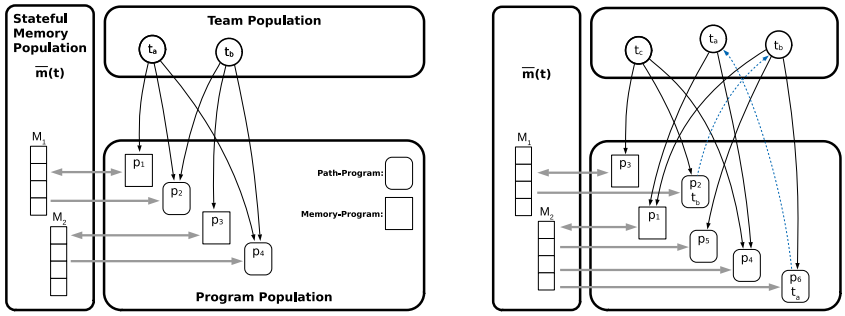
$D = 4$ and $T = 3$, then input to the forecaster at time t would be $[\vec{s}(t), \vec{s}(t-3), \vec{s}(t-6), \vec{s}(t-9)]$. Assuming the nature of a time series is stationary and known *a priori*, then D and T can be estimated such that models with no temporal memory or recursive structure can still make accurate predictions. However, in multi-task forecasting, multiple time series with unique rates of change are modeled simultaneously by a single generalized forecaster. It is infeasible to design a single autoregressive state representation that captures the salient temporal properties of all data streams. In this work, agents 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. Related studies have evolved "observation windows" in time-series forecasting, but still required human intuition to parameterize the window behaviour [Wagner et al. 2007]. By contrast, our approach is entirely emergent.

- **Non-stationary environment.** In our testbed for multi-task forecasting, agents are evaluated on multiple data streams, one at a time, in random sequence. The agent is not provided with any input marking the time at which the process generating the signal changes. As such, solutions are effectively required to discriminate signals *and* perform forecasting simultaneously. This represents a non-stationary environment that requires dynamic memory management over multiple timescales. Related work has investigated coevolutionary multi-task learning for non-stationary time-series environments [Chandra et al. 2018]. However, in that case agents operate in non-stationary environments by learning to select from a prior set of autoregressive state representations, which they refer to as *timespans*. Hence, critical components of the solution still required prior specification.
- **Recursive forecasting.** Due to the fact that agents (program graphs) 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, agents are primed with 50 samples. Thus, before predictions begin at $\vec{s}(t)$, the program graph is executed for each input sample in series $\vec{s}(t-50), \vec{s}(t-49), \ldots \vec{s}(t-1)$ and output values are ignored. When priming is complete, recursive forecasting is used to predict future values. That is, after $\vec{s}(t-1)$, samples from $\vec{s}()$ are no longer used as input. Instead, the agent's output values, or predictions, are fed back as input to predict future values. For example, the prediction for $\vec{s}(t)$ becomes the input at $t+1$, and so on. Thus, once priming is complete, the "true," or target, values are never fed into the program graph during test. As such, recursive forecasting can be characterized as *generative signal reconstruction*, and theoretically allows predictions to any time horizon [Herrera et al. 2007].

3.1 Structural Memory Model

TPG as described in Section 2 is designed for tasks in which solutions map sensor inputs to a set of discrete atomic actions. To perform time-series prediction, we will extend this framework to build program graphs capable of continuous (real-valued) output. This is achieved through a shared memory mechanism, originally introduced in Kelly and Banzhaf [2020] and Kelly et al. [2020], which serves the dual purpose role of enabling temporal memory *and* continuous output in the team-based structure of program graphs.

In this memory model, programs assume the same linear representation as shown in Figure 1(b). Internal register memory is *stateless*, that is, reset prior to each execution. To support temporal memory, we introduce a third population of shareable memory banks that are stateful, that is, only reset at the start of evaluation for each individual program graph. All programs have a pointer to one external memory bank (Figure 4). In the case of sequential decision-making or time-series tasks where the program is executed multiple times per evaluation, these shared stateful memory



(a) Initial Populations (b) Emergence of Program Graphs

Fig. 4. Illustration of the relationship between teams, programs, and modular memory in the structural memory model. Section 3.2 provides a detailed description of how a program graph maps each input state ($\vec{s}(t)$) to a continuous output.

banks allow programs to communicate and integrate information across multiple timesteps. Two types of program are now supported within a team (see Figure 4 and Algorithm 2):

(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 which parts of a program graph contribute to each prediction. They can be characterized as 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 terminal, then its output value is the content of its shared stateful memory register $M[0]$, i.e., a real value (see Algorithm 2). Otherwise, the path-program will point to another team where execution continues (this process is described in more detail in Section 3.2).

Evolution begins with a population of R_{size} teams, each containing at least two path-programs and two memory-programs, which share stateful memory banks (Figure 4). Two specialized mutation operators are introduced to support the development of shared memory. First, just as mutation operators are free to modify a program's action pointer (Step 1(h)vD, Algorithm 1), they may also change the memory pointer. Second, shared memory implies that the order of program execution within a team now potentially impacts a policy's output. As such, execution order is deterministic and defaults to the order in which programs are added to the team. However, a mutation operator is provided that may change the location of a program within the team's execution order. Shared memory also implies that a memory-program can appear more than once in the same team, since its contribution to memory management might be useful at multiple locations in the execution order. More details on parameterization are listed in Table 2.

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 path of execution through a program graph. When the path-program is terminal, its role is to define the appropriate *context*, relative to $\vec{s}(t)$ and $\vec{m}(t)$, in which its shared memory register $M[0]$ should be selected as the output.

3.2 Evaluating a TPG Agent in the Structural Memory Model

Figure 5 provides an example of how a TPG agent in the structural memory model is evaluated over two consecutive timesteps. Execution always begins at the root team (t_c). At timestep 1 (Figure 5(a))

ALGORITHM 2: Example linear register machine in the structural memory model. Each program has one internal *private stateless* register bank, R, and a mutable reference to one *shareable stateful* register bank, M. R is reset prior to each execution (line 1), while M is reset (by an external process) at the start of each policy evaluation. Each instruction will have a reference to one target register ($r1[]$) and one or two operand registers ($r2[], r3[]$). Mode bits in each instruction dictate the source for the target and operands. Mode bits are constrained such that $r3[]$ may refer to R, M, or input $\vec{s}(t)$ while $r1[]$ and $r2[]$ are restricted to R or M. Furthermore, the rules governing the target, $r1[]$, differ depending on program type. For path-programs, shared memory is read-only, thus $r1[]$ will always refer to R. In memory-programs, $r1[]$ may refer to either R or M. Note that path-programs have two return values (line 8). 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.

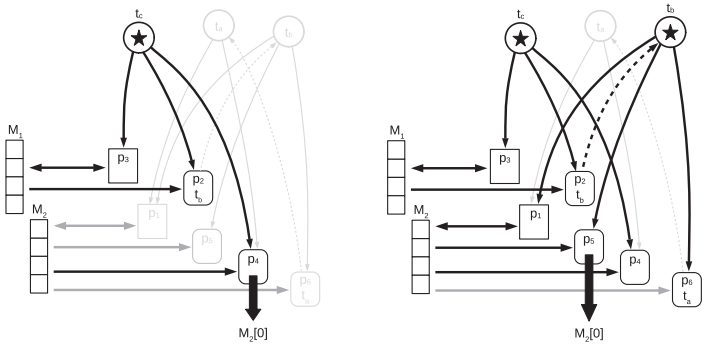
1: $R \leftarrow 0$ ▷ reset private memory bank
2: $r1[0] \leftarrow r2[0] \div r3[3]$
3: $r1[2] \leftarrow \cos r3[1]$
4: **if** $r1[0] < r3[2]$ **then**
5: $\quad r1[0] \leftarrow -r1[0]$
6: **end if**
7: **if** *Path* **then**
8: $\quad$ **return** $(R[0], M[0])$ ▷ (bid, continuous output)
9: **end if**

Table 1. Operations and Instruction Formats

Instruction	Operations
$r1[i] \leftarrow r2[i] \circ r3[j]$	$\circ \in \{+, -, x, \div, x^y\}$
$r1[i] \leftarrow \circ(r3[j])$	$\circ \in \{\cos, \ln, \exp, \sqrt{}, \sin\}$ $\circ \in \{\tanh, x^2, \lvert x\rvert, x^3\}$
IF $(r1[i] \circ r3[j])$ THEN $r1[i] \leftarrow -r1[i]$	$\circ \in \{<, >\}$

Path-programs encode eight operations in a 3-bit op-code. Memory-programs use a 4-bit op-code to include 8 extra operations (highlighted). In addition, memory programs have access to 18 constants: $\{-0.9, -0.8, \ldots, -0.1, 0.1, 0.2, \ldots, 0.9\}$, included as read-only registers at the end of their private register bank R (see Algorithm 2).

all of t_c's programs are executed in order (p_2, p_3, p_4) with $\vec{s}(t)$ as input, and the *path-program* with the highest bid (weight) defines the output. In this example p_4 had the highest bid, and since p_4 is terminal, the value stored in $M_2[0]$ is returned and execution for timestep 1 is complete. At timestep 2 (Figure 5(b)), execution again begins at t_c with $\vec{s}(t+1)$ as input. This time, p_2 has the highest bid, and since p_2 is not terminal, execution continues at t_b where programs p_1, p_5, and p_6 are executed for $\vec{s}(t+1)$. Assuming (terminal) program p_5 has the highest bid this time, the value of $M_2[0]$ is again returned. Note that in timestep 1 no memory-program with a pointer to M_2 was executed, thus the output ($M_2[0]$) could only contain a value written from a previous timestep. However, at timestep 2 p_1 is a memory-program with a pointer to M_2 and thus p_1 may have updated the value of $M_2[0]$ prior to it being returned by p_5. This example illustrates that 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



(a) Timestep 1 (b) Timestep 2

Fig. 5. Determining the output of a TPG agent in the structural memory model. Execution at each timestep begins at the root team (t_c) and continues until a terminal path-program is selected. The subset of the program graph that requires execution is dynamically selected at runtime and may differ in each timestep (highlighted in black). See Section 3.2 text for a detailed walk-through of this example.

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 et al. [2019], in which these properties where shown to improve generalization 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 can emerge from an open-ended evolutionary process.

3.3 Experimental Methodology

Our hypothesis is that emergent modularity in TPG (Section 2) combined with the temporal memory framework outlined in Section 3.1 can support multi-task time-series prediction. Specifically, multi-task learning implies that a single evolutionary run simultaneously develops: (1) Independent *specialist* solutions for each task; and (2) Multi-task *generalists* that adaptively recombine multiple previously-independent specialists into a hierarchical structure that exhibits a high level of competency in multiple tasks, as in compositional evolution [Watson and Pollack 2005]. This section outlines the components of the evolutionary framework that specifically support multi-task learning, adapted from an earlier study under multi-task reinforcement learning (Section 7 of Kelly and Heywood [2018b]).

3.3.1 Datasets. Our dataset preparation matches that in Turner and Miller [2017]. All three time-series datasets are univariate and contain 1,100 samples normalized to the interval $[0, 1]$. The fitness function measures how well solutions recursively predict the next 50 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 $t_{100}, t_{200}, \ldots t_{900}$. 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 single program graph with the best validation

score (over all generations)[7] is used to predict the next 100 samples from t_{1000} (i.e., the model is primed with samples $\vec{s}(t_{950}...t_{999})$.

3.3.2 Task switching. Each new program graph is evaluated on all three tasks as soon as it appears in the population. However, for each consecutive block of ten generations, only one time-series dataset is selected with uniform probability to be the *active* task. Fitness for a program graph in any given generation is the MSE with respect to the single active task. Thus, selective pressure is only explicitly applied for one task at a time. However, switching the active task at regular intervals throughout evolution means that an individual's long-term survival is dependent on its ability to operate in *all* tasks.

3.3.3 Elitism. There is no multi-objective component in the fitness function. However, elitism is used to ensure the population as a whole contains specialist policies for each task as well as generalist, or multi-task, policies. Specifically, the five program graphs with the best training fitness in each task are protected from deletion, regardless of which task is currently active for selection. However, this simple form of elitism does not protect multi-task policies, which may not have the highest score for any single task, but *are* able to perform relatively well on multiple tasks. A simple form of multi-task elitism identifies five elite multi-task teams in each generation using the following two-step procedure:

(1) Normalize each root team's fitness on each task relative to the rest of the current root population. Normalized score for team tm_i on task t_j, or $sc^{norm}(tm_i, t_j)$, is calculated as $(sc(tm_i, t_j) - sc_{min}(t_j))/(sc_{max}(t_j) - sc_{min}(t_j))$, where $sc(tm_i, t_j)$ is the mean score for team tm_i on task t_j and $sc_{min,max}(t_j)$ are the population-wide min and max mean scores for task t_j.
(2) Identify the multi-task elite individuals as those with the highest minimum normalized fitness over all tasks. Relative to all root teams in the current population, R, the elite multi-task team is identified as $tm_i \in R \mid \forall tm_k \in R : min(sc^{norm}(tm_i, t_{\{1..n\}}) > min(sc^{norm}(tm_k, t_{\{1..n\}})$, where $min(sc^{norm}(tm_i, t_{\{1..n\}}))$ is the minimum normalized score for team tm_i over all tasks and n denotes the number of tasks. Note that this process ranks teams by how well they perform at their weakest task and selects the five teams with the best worst-case performance.

Thus, in each generation, elitism identifies 5 champion program graphs for each task and 5 multi-task champions, for a total of 20 elite program graphs that are protected from deletion in that generation.

3.3.4 Parameterization. Compared to the single-task time-series prediction with TPG described in Kelly et al. [2020], the most significant parameter change under multi-task learning (TPG-MT) is the number of root teams to maintain in the population, R_{size}, which we reduce to 180 (from 360). With fewer root teams to evaluate in each generation, more generations (and more task switching cycles) are possible within a fixed computational budget. Table 2 provides a complete list of learning parameters. All evolutionary runs are performed on a shared cluster with a wall-clock computational limit of 4 h per job. We perform 20 independent runs. At intervals of 100 generations, the entire population is validated as described in Section 3.3.1. Test policies are identified from the validation process using the steps outlined in Section 3.3.3 (i.e., 1 specialist for each task and 1 multi-task generalist).

[7]Experiments are run on a shared cluster with wall-clock computational limit of 4 h per job. Each run was allowed to proceed for as many generations as possible within that time.

Table 2. Parameterization of Team and Program Populations

Team population			
Parameter	Value	Parameter	Value
R_{size}	180	R_{gap}	50% of *Roots*
p_{md}, p_{ma}	0.7	ω	10
p_{mm}	0.2	p_{mn}, p_{ms}	0.1
p_{mo}	0.5	p_x	0
Program population			
Parameter	Value	Parameter	Value
Size of reg. bank R	8	*maxProgSize*	100
Size of reg. bank M	8	p_{atomic}	0.5
$p_{mDelete}, p_{mAdd}$	0.5	$p_{mMutate}, p_{mSwap}$	1.0

For the team population, p_{mx} denotes a mutation operator in which: $x \in \{d, a\}$ are the prob. of deleting or adding a program, respectively; $x \in \{m, n, s\}$ are the prob. of creating a new program, changing a path-program's action pointer (leaf or team), and changing a program's shared memory pointer, respectively. p_{mo} is the prob. of changing a program's position in the team execution order. ω is the max initial team size. No team crossover is used here, i.e., $p_x = 0$. For the program population, p_{mx} denotes a mutation operator in which $x \in \{Delete, Add, Mutate, Swap\}$ are the prob. 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).

3.4 Results

3.4.1 Comparison Algorithms. The structural memory model is benchmarked in multi-task recursive forecasting by comparing its prediction accuracy with 7 alternative recursive forecasting algorithms over the 3 datasets outlined in Section 3.3.1. Turner and Miller [2017] established the performance of common statistical and machine learning methods such as **Autoregressive Integrated Moving Average (ARIMA)** and a **Multi-layer Perceptron (MLP)**, more complex methods such as **Recursive Cartesian Genetic Programming (RCGP)**, and a very recent approach to neural architecture search, or **Recursive Cartesian Genetic Programming of Artificial Neural Networks (RCGPANN)**. In addition, we include results for **Nonlinear Autoregressive Neural Network (NARNET)** and **Long Short-term Memory (LSTM)** from the Matlab Deep-learning Toolbox.[8] Finally, TPG-CM [Kelly et al. 2020], our evaluation of TPG's structural memory model in single-task time-series forecasting, is also included. These represent a breadth of modern time-series forecasting algorithms.

The evolutionary methods (RCGP, RCGPANN, and TPG) all perform a search over the structure of potential prediction machines, adapting the model complexity through interaction with each time series. Conversely, the more standard neural network machine learning methods (MLP, NARNET, and LSTM) require that structural properties of the network (i.e., number of hidden layers and number of nodes in each layer) be specified prior to training. Furthermore, the error back propagation algorithms used to train these networks require that correct input-output pairs be known for each prediction, thus the networks cannot be trained directly under recursive forecasting in which only a portion of the actual time series is available as input (50 samples, or one-half of each training slice; see Section 3.3.1). Turner and Miller [2017] propose a training methodology to address these issues, which is adopted here. First, we perform a simple architecture search

[8] https://www.mathworks.com/products/deep-learning.html.

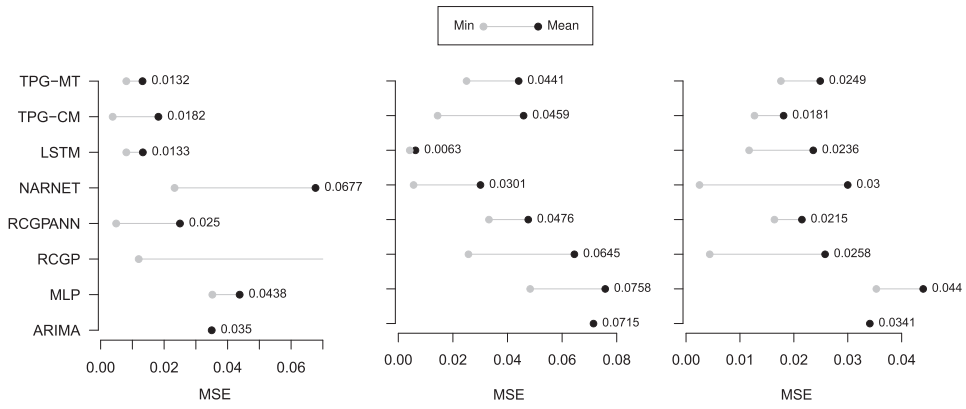
over networks with 1 and 2 layers and 5, 10, 20, and 50 neurons in each layer. Thus, 8 unique network architectures are trained in parallel for 1000 epochs using one-step-ahead prediction on the training set (the first 1,000 samples in each dataset). After each training epoch, the networks are evaluated on the validation set using recursive forecasting. That is, the network is evaluated on how well it recursively predicts the next 100 samples from $t = 100, t = 200, \ldots, t = 900$ (when given the previous 50 samples as input for priming). The single network configuration with the best validation score after *any* epoch is then returned as the final trained network. This process filters out any networks that may have overfit during training.

Finally, two important distinctions between TPG and the compared methods should be emphasized. First, all methods except TPG and LSTM employ some type of pre-specified sliding-window (autoregressive) state representation (see Section 3). In contrast, TPG and LSTM observe only 1 input at a time, and thus rely entirely on memory to encode temporal properties of the time series. Second, all methods except TPG-MT learn a solution for each time series independently, while TPG-MT builds single-task solutions for all three time-series *and* multi-task solutions simultaneously from a single evolutionary run.

3.4.2 Single-task Test Performance. Figure 6 provides test MSE for champion **single-task program graphs in our multi-task experiment (TPG-MT)**, along with seven alternative recursive time-series prediction algorithms outlined in Section 3.4.1. Given that the experimental methodologies differ significantly, Figure 6 is not meant to imply a definitive ranking. The goal of this comparison is to confirm that single-task solutions discovered by TPG-MT are generally competitive with state-of-the-art approaches to recursive time-series forecasting. Indeed, TPG-MT produces program graphs for the Sunspots datasets that improve on the mean MSE of all compared approaches, while the specialist TPG-MT solution for the Laser and Mackey-Glass datasets rank fourth and third, respectively, of eight compared algorithms. Kelly et al. [2020] provide addition behavioural analysis of TPG-CM (i.e., single-task TPG). Having established that TPG-MT can still produce competitive single-task solutions, the next section explores the test performance of *multi-task* program graphs capable of predicting more than one time series.

3.4.3 Generalist Test Performance. Figure 7 reports the test performance for the best multi-task program graph discovered for every combination of the three time-series tasks considered in this work (i.e., four unique program graphs). The MSE for each multi-task generalist is compared to the mean MSE of each single-task approach listed in Figure 6. For example, the best three-task solution from TPG-MT, Figure 7(a), achieves an MSE that is better than the single-task mean of all compared approaches on Laser, all but LSTM on Sunspots, and ranks in the middle of all comparison algorithms on Mackey-Glass. The best two-task solutions are shown in Figures 7(b), 7(c), and 7(d). The two-task program graphs consistently rank in the top three of eight comparison (single-task) algorithms. If multiple related tasks are learned together, then experience gained in one task may improve learning in the another, i.e., transfer learning [Kelly and Heywood 2018a]. Conversely, this study specifically identifies three *unrelated* time-series tasks that should be challenging to learn simultaneously. As such, the multi-task results presented in Figure 7 are promising even when they do not quite match the scores achieved by single-task solutions.

To confirm the importance of hierarchical models in the multi-task recursive forecasting task, we also include results for a version of TPG parameterized with $p_{atomic} = 1.0$ (SBB-MT in Figure 7). In this case, TPG's ability to construct program graphs is disabled, and all evolved prediction machines will take the form of a single team of programs. This increases the forecasting error in every combination of tasks. However, hierarchical model building provides the greatest benefit under the three-task scenario, Figure 7(a). In the next section, we investigate how multi-task program graphs



(a) Sunspots (b) Mackey-Glass (c) Laser

Fig. 6. Test results for single-task recursive time-series forecasting. Points indicate mean and min MSE on test data over 50 independent runs for each method. TPG-MT builds specialized solutions for all three time-series tasks simultaneously from a single set of 50 runs. All other methods learn each task independently from scratch (i.e., 50 runs are performed for each task). Results for ARIMA, MLP, RCGP, and RCGPANN are from [Turner and Miller 2017]. Results for TPG-CM are from [Kelly et al. 2020]. Note that RCGP performed so poorly on the Sunspots time series that its mean MSE is off the chart.

evolve and how emergent modularity/hierarchy in temporal memory structures specifically supports adaptation in this dynamic environment with multiple disparate timescales.

3.4.4 Training and Development. Figure 8 plots developmental properties of the TPG-MT run that produced the best three-task generalist (Figure 7(a)). Fitness for the elite specialist in each task is shown in Figure 8(a). While the most significant improvement takes place between generations 0 and 2,000, progress continues in all tasks throughout the duration of the run. Figure 8(b) plots the development of hierarchical complexity for these specialist policies as measured by the number of teams appearing within each program graph. Note that changes in fitness are typically correlated with transitions in hierarchical complexity. Program graphs often benefit from subsuming additional teams, but the correlation between changes in hierarchical complexity and fitness can also be neutral or negative. In either case, adaptively recombining multiple independent teams into hierarchical structures, or compositional evolution, clearly plays a significant role in model building with TPGs. Developmental data for the single multi-task generalist appears in Figures 8(c) and 8(d). Note that *generalist* implies that all three fitness curves in Figure 8(c), as well as the complexity curve in Figure 8(d), are derived from the same program graph. Compositional evolution appears to also play a critical role in the development of generalists. For example, many large and small changes in hierarchical complexity between generations 0 and 1,000 result in significant fitness fluctuations, but overall improvement in all tasks. The hierarchy seems to stabilize at generation $\approx$ 900. From there on, smaller hierarchical transitions result in relative fitness trade offs throughout evolution. Interestingly, the champion generalist policy graph at generation 5,000 contains $\approx$ 33 teams (Figure 8(d)), indicating that it is no more complex than any of the specialist program graphs (Figure 8(b)).

3.4.5 Solution Analysis. Figures 9 and 10 show example test behaviours for specialist and generalist program graphs produced from a single evolutionary run. That is, the data for each benchmark in Figure 9 is the result of an independent agent, while the data in Figure 10 is the result of one agent capable of recursively forecasting two unrelated time series. Note that recursive forecasting

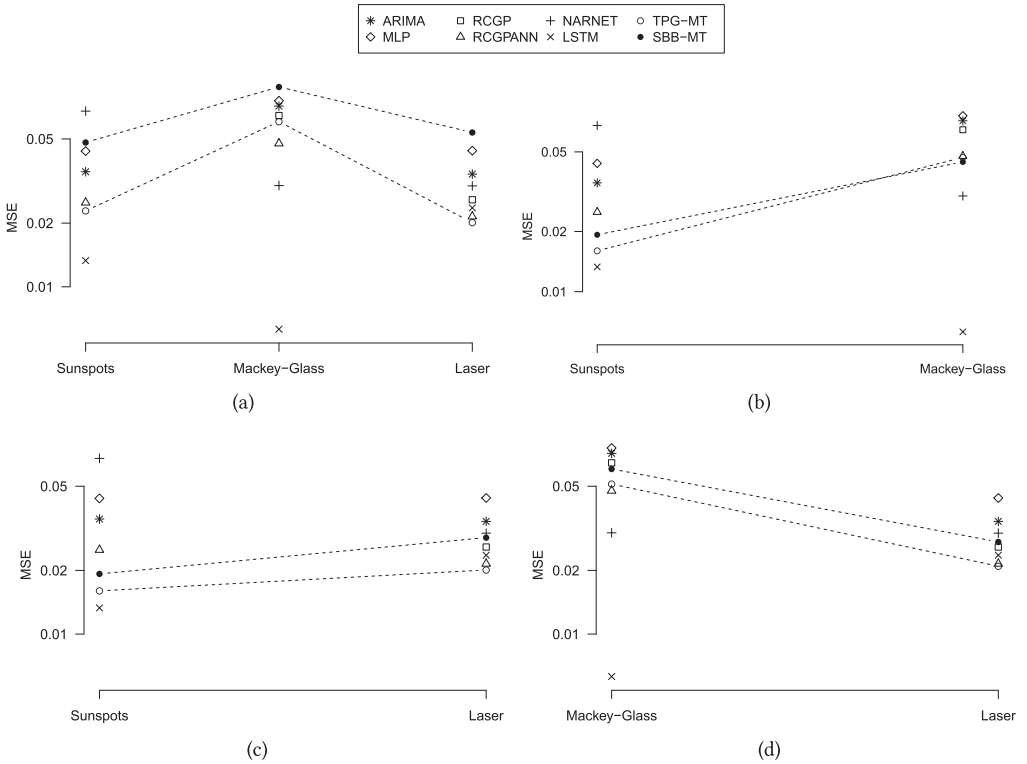
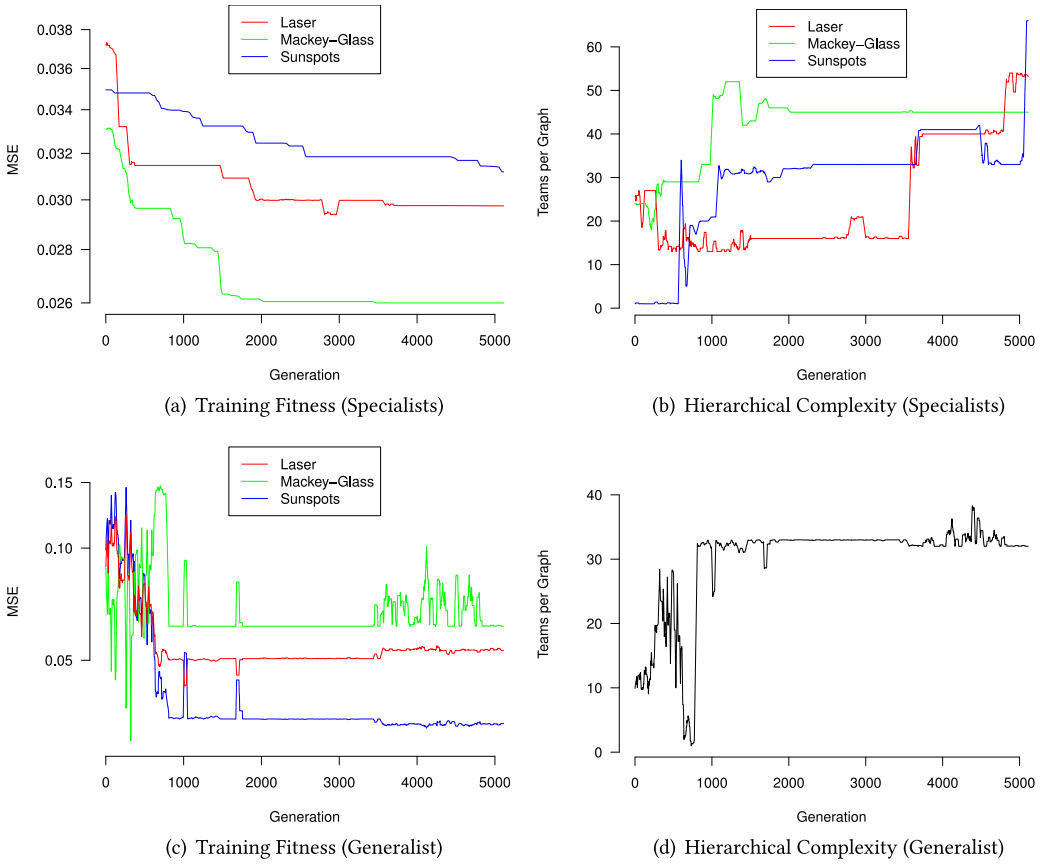


Fig. 7. TPG-MT multi-task test performance. MSE for ARIMA, MLP, RCGP, RCGPANN, NARNET, and LSTM are the mean over 50 *single-task* runs in each time-series task. That is, each method is trained from scratch for each task. (Also listed in Figure 6). Results for TPG-MT are for the single best *multi-task* program graph for each combination of tasks. SBB-MT is TPG parameterized with $p_{atomic} = 1.0$. In this case, TPG's ability to construct team hierarchies is disabled and all evolved prediction machines will take the form of a single team of programs. For TPG-MT and SBB-MT, all 4 multi-task program graphs were evolved simultaneously from a single set of 50 runs. The dotted line connecting points indicates that the MSE on each time series is achieved by the same (multi-task) program graph. Note that RCGP preformed so poorly on the Sunspots time series that its MSE is off the chart.

means that all test forecasts are produced without observing any of the 100 test points. That is, once the model is primed (Section 3.3), 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 memory registers at t_{i+1}, is the only information available to generate a prediction for t_{i+2}. As such, the multi-task forecaster in Figure 10 effectively uses the priming stage to discriminate between two independent signals *and* accumulate enough information in memory to trigger a recursive forecast.

Since no autoregressive state is provided, each program graph must define a mechanism for encoding previous observations within stateful memory registers, and recalling or overwriting these memories as required. Essentially, each program graph defines an embedding dimension that is adapted to the characteristics of the particular dataset(s) observed during training. The prediction at each timestep 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. Thus, the time point at which stateful memory registers are overwritten or left to accumulate is selected based on the



(a) Training Fitness (Specialists)

(b) Hierarchical Complexity (Specialists)

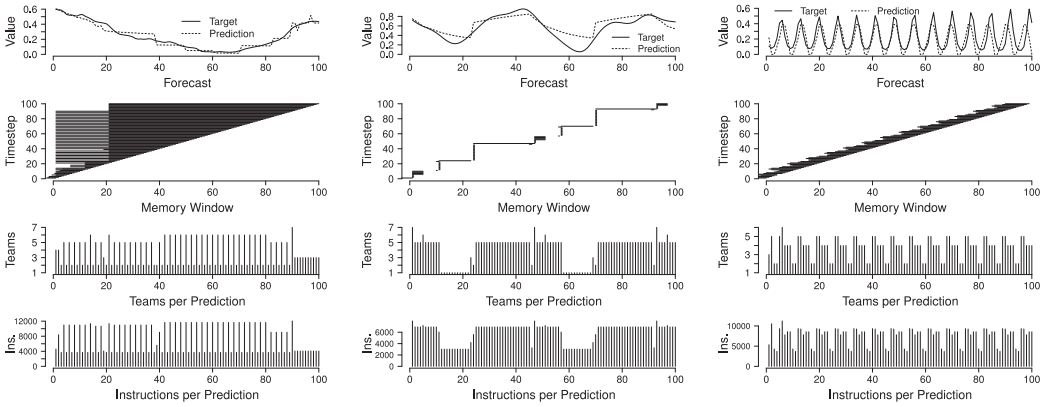
(c) Training Fitness (Generalist)

(d) Hierarchical Complexity (Generalist)

Fig. 8. TPG-MT training fitness and hierarchical development. Panels (a) and (b) represent data from the best single-task *specialist* solutions in each task. Panels (c) and (d) show data for the single best multi-task *generalist*.

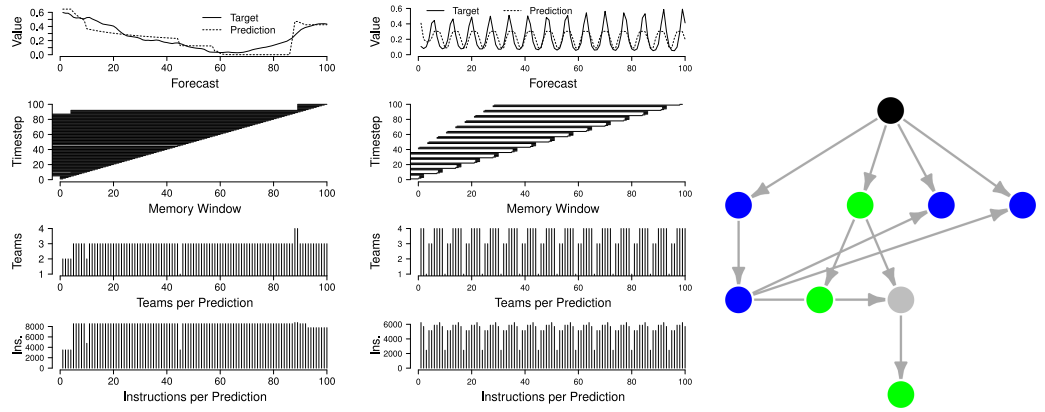
current input as well as the content of stateful memory. As the execution path changes over time, the solution's embedding dimension also becomes dynamic. In particular, the "age" distribution of memories accessed at any point in time effectively defines a dynamic temporal window. For example, in Figures 9 and 10, "Memory Window" depicts the agent's memory timespan 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 agents exhibit a unique pattern of dynamic memory access for each time series, and how the general rate of change in the target data is reflected in the agent's memory access. By contrast, our approach is entirely emergent.

Figures 9 and 10 also show how the computational complexity of program graph execution is a dynamic property. Interestingly, even though the champion program graphs may subsume up to 60 teams per graph (see Figures 8(b) and 8(d)), typically only one to nine teams are visited per timestep under test, and the rate of path switching again correlates with the target data and dynamic memory window. Naturally, each team executes a unique subset of programs, each with a variable length list of instructions. Dynamic runtime complexity improves the efficiency of agent



(a) Sunspots (b) Laser (c) Laser

Fig. 9. Example test behaviour for three specialist program graphs produced from a single TPG-MT run. X-axis in all plots indicates timesteps. Program intron removal is performed prior to execution, thus instruction counts are for effective instructions only.



(a) Sunspots (b) Laser (c) Program Graph

Fig. 10. Example test behaviour for a program graph capable of recursive prediction in the Sunspots and Laser datasets (see Figure 7(c)). X-axis in panels (a) and (b) indicates timesteps. Instruction counts are post intron removal. Panel (c) shows the program graph associated with these results. Each node represents one team of programs. Green and blue nodes are used only for Sunspots and Laser, respectively, i.e., specialist teams. Black and gray nodes are used for both datasets, i.e., generalist teams.

deployment when averaged over many timesteps. This is especially significant as complex (temporal) problems call for increasingly complex agents, as will be investigated in Section 4.

Figure 10(c) shows the multi-task program graph capable of recursively forecasting the Sunspots *and* Laser time series with an MSE that is comparable to all single-task methods (Test data for this graph is reported in Figures 10(a), 10(b), and 7(c)). Each node in the graph represents one team of programs. In total, the graph uses only nine teams during test for the Sunspots and Laser benchmarks, and thus the structure and runtime complexity of this multi-task graph is no more than any of the specialist agents in Figure 9. Furthermore, a task decomposition within the graph is apparent, as it contains specialized teams that are used only for Sunspots (green) or Laser (blue), as well as generalist teams that contribute to forecasting in both environments (black, gray). The

generality of this solution comes at the cost of some prediction accuracy, as its MSE on each task is worse than that of the specialist solutions. However, there is also a significant advantage for multi-task behaviours in real-world environments. In health care, for example, multiple clinical prediction tasks with continuously shifting conditions are routinely performed in parallel, and having a generalized prediction machine that can handle multiple tasks while also modeling correlations between tasks distributed in time is highly advantageous [Harutyunyan et al. 2019].

4 BENCHMARKING OF A PROBABILISTIC MEMORY MODEL

Section 3 demonstrates that the TPG can be extended to support an incremental model of indexed memory. In this section a completely different approach is taken in which indexed memory is available from the outset, but there is only one instance of indexed memory across the entire population of TPG agents. This means that indexed memory represents a "communication medium," so explicitly establishing a common view of internal state as experienced by all TPG agents. In addition, write operations are probabilistic, with the content of the writing program's registers being distributed across the common memory to simulate the properties of long- and short-term memory. Such a model of memory has previously been shown to be capable of supporting the emergent discovery of navigation behaviours under "deathmatches" in ViZDoom tournaments [Smith and Heywood 2019a] and Dota 2 one-on-one competitions [Smith and Heywood 2019b, 2020]. The ViZDoom environment results in agents experiencing state from a high-dimensional first-person three-dimensional perspective and will be returned to in this work, i.e., multiple partially observable properties exist.

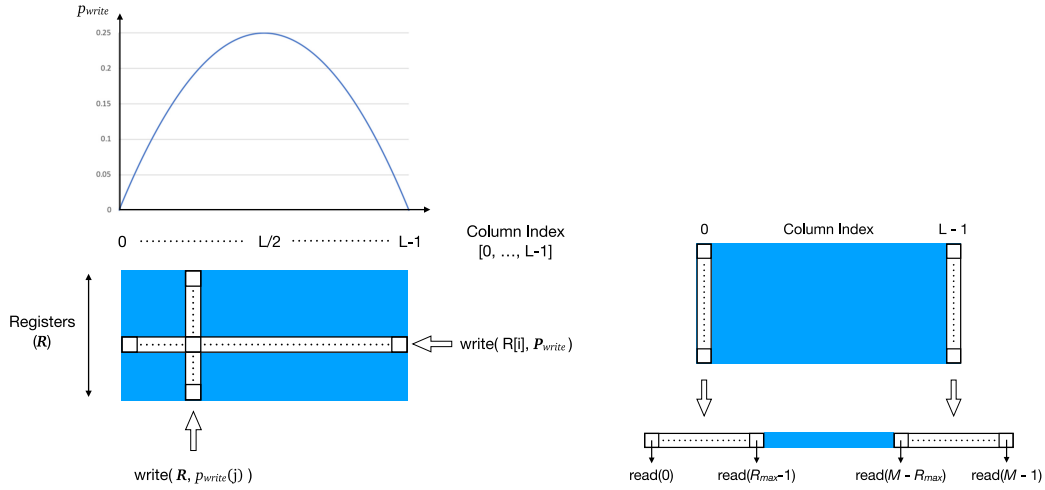
The model of memory developed here is motivated by an underlying desire to operate in high-dimensional state spaces. In particular $\vec{s}$ represents a $N = 320 \times 240 = 76,800$ dimension pixel space (Section 4.2), or 75 times larger than that appearing in typical image classification benchmarks such as CIFAR.[9] However, pixel spaces from video input are also highly redundant. With pixels in the same spatial and temporal locality describing similar content. Thus, unlike the recursive time-series forecasting tasks of Section 3 (dimension $N = 1$) where the chaotic nature of the underlying process results in a lot of sensitivity to specific values of input state, the visual RL task is considered less sensitive to the value of particular input pixels. However, to capture state across such a large state space, we cannot assume that single programs index all the state space (a given under Section 3). Instead, we provide TPG with one instance of indexed memory, $\vec{m}$, that all programs may read from/ write to. In effect forcing a common view of (internal) state where the dimension of indexed memory is very much lower than the state space ($|\vec{m}| << D$).

In the following, Section 4.1 develops the proposed probabilistic model of indexed memory. Properties of the ViZDoom benchmarking task are outlined (Section 4.2) after which two ViZDoom tasks are characterized: Take Cover (Section 4.3) and Pathfinding (Section 4.4). The former is used to conduct an ablation study (Section 4.6) and the latter represents a large scale challenge for memory (Section 4.7). Section 4.5 defines the common TPG parameterization assumed throughout.

4.1 Probabilistic Indexed Memory Model

Each TPG program is capable of stateful operation, thus under a linear GP representation (Figure 1(b)) register values are not reset after execution. Let $\langle p_i(t) : \mathcal{R} \rangle$ denote the value of each of the R_{max} registers associated with program i at interaction t with the environment. Thus, in a non-Markovian task, $\langle p_i(t) : \mathcal{R} \rangle$ potentially carries useful information between instances of program execution. That said, register state has two basic limitations: (1) any instruction (from the same program) can modify $\mathcal{R}$, making it more difficult to retain long term properties (high likelihood of

[9] https://www.cs.toronto.edu/~kriz/cifar.html.



(a) Write (b) Read

Fig. 11. Probabilistic Indexed Memory Model. Indexed memory is organized as $R_{max} \times L$ locations, where R_{max} is the number of registers a program may manipulate (see Figure 1(b)). 11(a) Write operations: `write`($\mathcal{R}$) are probabilistic (Algorithm 3). The process can be visualized as distributing the value of a specific register `R[i]` across L columns of indexed memory according to probability distribution P_{write}. Equivalently, the $\mathcal{R}$ register values are written to column j with probability p_{write}. 11(b) Read operations: `R[i] = Mem[j]` assume a "column major" format in which the L sets of R_{max} memory locations are concatenated into a single vector of indexed memory.

disruption), and (2) $\langle p_i(t) : \mathcal{R} \rangle$ is specific to each program, thus it is not possible to pass "memory" between the multiple programs that make up TPG individuals.

Indexed memory is synonymous with the manner in which a CPU accesses "global" memory, i.e., with read/write instructions specifying an address. Previous research has considered the role of indexed memory in the evolution of basic data structures [Langdon 1998; Teller 1994] and reviewed memory models assumed for GP and neural networks [Smith and Heywood 2020]. Indexed memory was associated with the evolution of "mental models" for navigation within a 4×4 grid world [Brave 1996]. A two phase evolutionary cycle was necessary, with cycle 1 only writing content and cycle 2 rewarded for reading back the relevant content. Such a two cycle process assumes that the agent is able to navigate the environment in the first cycle. However, for the tasks addressed in this work, memory is a pre-requisite for navigation. Indexed memory models have also been proposed for use with robot controllers, e.g., Andersson et al. [1999]. In this case, a performance measure was needed to define what memory content was actually saved as well as the criterion for replacement. Moreover, each write operation wrote the entire state space to memory. Neither would be feasible in the task environments considered here.

A theme developed further by our approach is the concept of retaining a single "instance" of indexed memory [Spector and Luke 1996]. This implies that the state of indexed memory, $\vec{m}(t)$, is *never* reset. Thus, each TPG agent inherits the indexed memory state as left from the previous agent's interaction. Moreover, indexed memory is never reset between generations. Our motivation is to ensure that all agents evolve a common/shared concept for what constitutes useful memory content, and where to find it. Thus, only at the evaluation of the first TPG agent at the first generation does $\vec{m}(0) = \text{NULL}$.

Indexed memory, $\vec{m}$, is formulated to provide a probabilistic write operation and indexed read, Figure 11. We assume that the data written to memory takes the form of the vector of register state, $\mathcal{R}$, from program $p_i(t)$ at the point where a write instruction is executed, or `write(`$\mathcal{R}$`)`.[10] Thus, register values are assumed to be a suitable low dimensional encoding for defining what the state of the agent is, as opposed to $\vec{s}(t)$.

A **write operation** also has to define where in memory to write data. A probabilistic model is assumed (Algorithm 3) in which the vector $\mathcal{R}$ is distributed across L columns of indexed memory, Figure 11(a). Thus, from the perspective of a write operation, $\vec{m}$ is organized as a $R_{max} \times L$ matrix. The probability of the write operation is defined by a probability distribution, P_{write}:

$$P_{write}\left(\frac{L}{2} \pm c\right) = \alpha - (\beta \times \mathrm{c})^2, \tag{1}$$

where α is the maximum probability (Figure 11(a)), β is a scaling factor used to ensure that P_{write} extends over all the entire "width" of memory (i.e., a function of the value for L) such that $P_{write} > 0$ (Figure 11(a)), and $0 \leq c < \frac{L}{2}$ is the memory column index. Equation (4.1), therefore, implies that columns in the region of $\frac{L}{2}$ are written to with a higher probability than columns in the regions about 0 or $L-1$ (Algorithm 3). This defines short- and-long term memory within $\vec{m}$, respectively.

ALGORITHM 3: Write function for Indexed Memory $\vec{m}$. Function called by a write instruction of the form: `write(`$\mathcal{R}$`)` where $\mathcal{R}$ is the vector of register content (`R[0]`, ..., `R[`R_{max}`-1]`) of the program when the write instruction is called. Step 2 identifies the mid point of memory $\vec{m}$, effectively dividing memory into upper and lower memory banks. Step 3 sets up the indexing for each bank such that the likelihood of performing a write decreases as a function of the distribution defined in Step 4. Step 5 defines the inner loop in terms of the set of registers, `R[i]`$\in \mathcal{R}$, that can source data for a write. Step 6 tests for a write to the upper memory bank and Step 9 repeats the process for the lower bank.

```
1:  function WRITE(R)
2:      mid = L/2
3:      for offset := 0 < mid do
4:          p_write = α − (β × offset)^2          ▷ Define P_write (Figure 11(a))
5:          for j := 0 < R_max do
6:              if rnd[0, 1) ≤ p_write then       ▷ Upper Memory Bank
7:                  m⃗[mid + offset][j] = R[j]
8:              end if
9:              if rnd[0, 1) ≤ p_write then       ▷ Lower Memory Bank
10:                 m⃗[mid − offset][j] = R[j]
11:             end if
12:         end for
13:     end for
14: end function
```

Read operations assume a memory model in which $\vec{m}$ is perceived as a single array of consecutively indexed memory locations, Figure 11(b), i.e., a "column major" format. A read instruction therefore specifies an index to read from and a target register, or `R[i] = Mem[j]` where $0 \leq j < R_{max} \times L$. In effect, we anticipate that over time read references to indexed memory will

[10] We do not preclude the same program performing multiple write (or read) operations.

learn to distinguish between short- and long-term memory locations, whereas write references will learn what (register) states are actually useful to record.

In summary, TPG will incrementally stitch together teams into a tangled program graph (Section 2). To support indexed memory, programs are provided with read and write instructions (Figure 11). The definition of the write instruction distributes register state, $\mathcal{R}$, across indexed memory, $\vec{m}$, to support long- and short-term retention of state (Algorithm 3). Finally, by maintaining a single instance of indexed memory, $\vec{m}$, we encourage initially independent TPG agents to share a common "view" of indexed memory content. In doing so, we reduce the disruption potentially resulting from different agents having incompatible indexed memory content, thus offspring are more likely to be compatible.

4.2 ViZDoom Environment

ViZDoom[11] represents an efficient multi-platform game engine [Kempka et al. 2016; Wydmuch et al. 2019] that presents state from a "semi-realistic" three-dimensional first-person perspective. Features that make the ViZDoom environment relevant to this research include:

- Visual state, $\vec{s}(t)$, can be defined over a range of pixel vector resolutions, from $160 \times 120 = 19{,}200$ pixels to $1{,}920 \times 1{,}080 = 2{,}073{,}600$ pixels. Resolution has an impact on the complexity of the state space.
- ViZDoom describes a three-dimensional world in which the learning agent is limited to state described by a first-person perspective alone. Hence, the same object can have multiple perspectives, and the perception of location is subject to the orientation of the agent. This makes object recognition much more difficult than in two-dimensional environments and state information is generally incomplete.
- There are many different types of object to interact with, some of which might be explicitly hostile, some beneficial, and others neutral. In addition, depending on the level design, the agent might be required to solve puzzles (e.g., enabling doors by first operating levers or switches in completely different rooms).
- ViZDoom is parameterized with eight source task scenarios[12] as well as "target" tasks such as Deathmatches. The source tasks can be used individually, or as part of a curricula to develop agents with enough capability to address target tasks. Moreover, the original commercial (or custom) level "maps" can be used as the basis for further types of assessment, such as puzzle solving and complex navigation tasks.
- Light-weight game engine that does not need GPU support for operation [Kempka et al. 2016]. This means that the game can be played at 100's of frames per second on a CPU.

In this work, we perform two studies using the ViZDoom environment:

- An ablation study using the **Take Cover** source task. The purpose of the ablation study is to illustrate the contribution from different components of the TPG framework. Specifically, TPG will be assessed with and without hierarchical modularity and with and without probabilistic indexed memory.
- A **Level Pathfinding** task in which the underlying goal is to navigate a level as fast as possible. The particular interest to this work is learning to navigate a multi-room labyrinth while simultaneously addressing interactions with hostile agents, stochastic spawn states and simple puzzle solving (e.g., key to door association, switch to door activation). The

[11] https://github.com/mwydmuch/ViZDoom.
[12] https://github.com/mwydmuch/ViZDoom/tree/master/scenarios.

ability to navigate was previously shown to appear spontaneously under Deathmatches when memory was provided [Smith and Heywood 2019a].

In the following, we begin by summarizing the properties of each task and define their respective performance functions (Sections 4.3 and 4.4). Parameterization of TPG is discussed in Section 4.5, where this is common to both studies. Results for the ablation and pathfinding studies then appear in Sections 4.6 and 4.7, respectively.

4.3 Take Cover Task

Take Cover represents a source task environment defined in the ViZDoom game engine [Kempka et al. 2016]. The task scenario is designed to teach an agent to avoid projectiles, thus the agent is rewarded for behaviours that maximize its lifespan. The world takes the form of a rectangular room (so there is no place to "hide") with the agent spawned in the centre of the longer wall. Opponents are spawned from random locations on the opposite wall and launch "fire balls" at the agent. The agent may only move left or right. The longer the agent lives, the more opponents appear. The reward function is just the count of the number of game frames that the agent survives. The agent perceives state from a first-person perspective, through sequential frames of video from the game engine, or a state space of dimension (N) of $320 \times 240 = 76{,}800$ pixels. The initial RGB pixel format is concatenated into a single 24-bit number [Smith and Heywood 2018].[13] This task is challenging due to the high dimensionality of the state space (which pixels should be indexed to make a decision), the need to estimate the direction from which projectiles appear, and the rate of change of such projectiles.

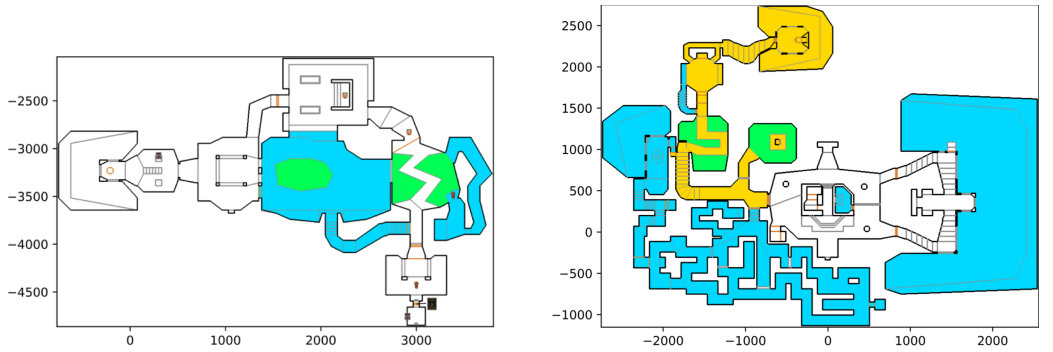
4.4 Pathfinding Task

The Level Pathfinding environment consists of multiple rooms of different size/configuration connected together by corridors and doors (Figure 12). There are also various objects distributed across the environment, some are potentially beneficial[14] and some may or may not be useful, e.g., barrels of acid. Object locations are fixed, but the spawn points for TPG agents are selected randomly and opponent Bots roam stochastically. Moreover, to succeed in this task, it is necessary to find 'hidden' items, such as a key to gain access to new parts of the environment. The comparatively rare nature of key collecting and using it to open doors makes the task particularly challenging, even for agents with memory.

For the purpose of this study, two map levels are assumed, Figure 12. State, $\vec{s}(t)$, at any point in time represents a first-person perspective, thus agents are unaware of the global knowledge as captured by the maps. As per the Take Cover task, state is defined by the content of a 320×240 frame buffer, or a dimension of $N = 76{,}800$. A set of eight discrete atomic actions ($\mathcal{A}$) are assumed: Forward, Backward, Turn Left/Right, Strafe Left/Right, Interact, and Shoot.

4.4.1 Curriculum and Performance Function. Attempting to evolve TPG agents directly against the Pathfinding task is not possible "tabula rasa," i.e., the performance objective of Pathfinding is not sufficient to provide a useful search gradient. Such scenarios therefore require some form of "curricula" to be constructed that represent simpler tasks. Success against a set of simpler "source" tasks can then be used to achieve a level of ability before exposing the TPG agent to the "target" task. Many mechanisms have been proposed for achieving this, including incremental evolution [Gomez and Miikkulainen 1997], layered learning [Stone 2000], task decomposition [Whiteson et al. 2005], or task transfer [Kelly and Heywood 2018a; Taylor and Stone 2009].

[13] Each of the original R-G-B channels is encoded in 8-bits and then concatenated into one 24-bit number.
[14] Medical kits, two types of armour, and several different forms of weaponry.



(a) Level 1 map (b) Level 2 map

Fig. 12. Topology of environment for pathfinding task. Up to three zones are identified (white, blue, yellow). Agents are initialized in the white zone. Green regions within a zone identify an acid bath, i.e., negative for agent health, (a) corresponds to "E1M1: Hanger" level from Doom. Level exit is at co-ordinate (2,900, −4,700) and (b) corresponds to the 'E1M2: Nuclear Plant' level from Doom. The key to exit this zone is at co-ordinate (1,000, 300). The door to exit the white zone is at co-ordinate (−750, 400), but can only be opened with the key. The yellow zone contains the level exit at co-ordinate (−350, 2,300) and has two acid baths (green). Blue zones (secret areas) typically have extra requirements for access and may detract from finding the shortest path to the level exit.

In this work, a stochastic framework is assumed in which a "bag," $\mathcal{B}$, is defined, consisting of the eight source tasks[15] from ViZDoom.[16] A source task is selected stochastically (without replacement) from $\mathcal{B}$ and fitness expressed in terms of performance over the last three source tasks experienced by the agent [Smith and Heywood 2018]. Once *Max. Bag* iterations through the entire set of $\mathcal{B}$ source tasks has been performed, the target Pathfinding task is introduced and fitness thereafter only reflects performance on the target task.

The target task takes the form of finding an efficient solution for navigating the level 2 map[17] (Figure 12(b)). There are three zones, of which only the white and yellow zones are important for finding the shortest path through the level. The learning agent is spawned anywhere (at any orientation) in the start zone and has to find a key before it can open a door to the finish zone. While in either the start or finish zones a learning agent might gain access to the "blue zones," which will only make the task of getting to the exit door (in the finish zone) more difficult. Moreover, the agent does not have access to the map depicted by Figure 12(b) and the various zones are not identified by colour. Instead, the TPG-agent is only able to perceive game state through the first-person perspective, thus has to use memory to facilitate navigation of the task.

Successfully finding the key represents a critical, but rare event, likewise, associating the key with opening a specific door. In short, success in the Pathfinding task is only possible if the agent is able to switch between multiple objectives that might well have changing priority through the course of the navigation task, e.g., surviving might imply avoiding/ shooting opponent Bots, collecting health packs/ shields, avoiding acid baths as well as attempting to navigate different regions of the level. With this in mind, fitness rewards five specific properties/objectives, of which four ($r_{key}, r_{door}, r_{exit}, r_{switch}$) can only be satisfied sequentially, as per Algorithm 4. Having satisfied

[15]Basic, Deadly Corridor, Defend the Centre, Defend the Line, Health Gathering, My Way Home, Predict Position, Take Cover.

[16]https://github.com/mwydmuch/ViZDoom/tree/master/scenarios.

[17]The "E1M2–Nuclear Plant" from the original 1993 release of Doom as designed by John Romero.

Table 3. Relation between Opcodes and Operands

Opcode	Instruction
$\langle op_0 \rangle \in \{<\}$	IF $R[x]\langle op_0 \rangle R[y]$ THEN $R[x] = -R[x]$
$\langle op_1 \rangle \in \{cosine, ln, exp\}$	$R[x] = \langle op_1 \rangle (R[y])$
$\langle op_2 \rangle \in \{+, -, \div, \times\}$	$R[x] = R[x]\langle op_2 \rangle R[y]$

Register-Register instructions index registers alone, or $x, y \in \{0, \ldots, R_{max} - 1\}$. A Register-Input reference uses a different range for the y operand: Register-Input $y \in \{0, \ldots, N - 1\}$. N is the number of pixels in the input. *ln* and *exp* are (protected) natural logarithm and exponential operators (e.g., absolute value of the operand is assumed) and NaN is trapped in the case of division.

Table 4. TPG Parameterization Assumed for Pathfinding Tasks

Parameter	Value	Parameter	Value
Team population size	600	Max Team Size	12
Gap (%)	50	Max Instructions per Program	1,024
p_x	1.0	Max. Bags (Navigation)	50 (25,000)
p_d	0.7	Max. Gen (Take Cover)	5,000
p_a	0.7	p_{del}	0.5
p_{nm}	0.2	p_{add}	0.5
p_{atomic}	0.5	p_{swp} and p_{mut}	1.0

p denote probabilities of applying different variation operators. Max. Bags is the number of iterations through the ten source tasks of the training curricula (Section 4.4) before performing Max. Navigation generations on the navigation task.

the four sequential components, then r_{time} ranks the quality of a qualifying solution. Each of the four sequential objectives has a specific goal criterion, which contributes r_{max} when satisfied.

Note also that solving r_{switch} might be considered comparatively easy, whereas r_{key} and r_{door} might be considered "deceptive." That is to say, r_{key} and r_{door} only reward minimizing the Euclidean distance (which may lead to dead ends) and ignores other factors such as agent health or location of opponent Bots. Likewise, r_{exit} ignores the need to "enable" doors by first throwing a switch and ignores anything to do with survival.

4.5 Parameterization

Programs are expressed in an imperative programming language, or linear GP [Brameier and Banzhaf 2007], with the specific form of instructions defined by Table 3. A mode bit selects between one of two addressing modes: Register-Register and Register-Input. Each program has $R_{max} = 8$ registers that are unique to each program. Read and Write instructions are defined independently (Section 4.1). Write instructions assume a probabilistic model (Equation (4.1)) thus are parameterized by the number of registers per program, $R_{max} = 8$ and number of columns, L (Figure 11(a)). We assume $L = 100$, thus $|\vec{m}| = 800$. A maximum probability of writing to memory of $\alpha = 0.25$ implies that $\beta = 0.01$ defines the lowest probability at $L = 0$ or 99 (i.e., short-term memory is 25 times more likely to be written to than long term memory). A Read instruction assumes the form `R[i] = Mem[j]` (Figure 11(b)) with $0 \leq j < |\vec{m}|$. Table 4 summarizes the TPG parameters assumed for this work.

The only differences between the Take Cover and Pathfinding task parameterizations are that the Take Cover task does not need to switch between multiple source tasks during training ("bags" in Table 4) before encountering the target task. Thus, for Take Cover there is only ever

ALGORITHM 4: Overall fitness is the combination of 5 rewards ($r_{time}, r_{key}, r_{door}, r_{exit}, r_{switch}$). There are two parameters: $\tau (= 10^{-5})$ a minimum (e.g., for penalizing suicidal behaviour) and $r_{max} (= 1{,}000)$ representing maximum reward. Function "timeReward" receives argument $t_{episode}$, i.e., the time the agent has spent within the level in seconds of simulated time. Function "distReward" receives four arguments: cartesian (x, y) co-ordinate of the TPG agent; (x, y) co-ordinate of the next objective; fitness value for the previous objective; and a boolean (`True, False`) value indicating whether the agent satisfied the present objective or not. "Euclid" returns the scalar Euclidian distance between two cartesian co-ordinates: *current*(x, y) and *target*(x, y)

function OVERALLFITNESS
  r_{time} = TIMEREWARD($t_{episode}$)
  r_{key} = DISTREWARD(player(x, y), redKey(x, y), r_{time}, redKeyPickedUp)
  r_{door} = DISTREWARD(player(x, y), redDoor(x, y), r_{key}, redDoorOpened)
  r_{exit} = DISTREWARD(player(x, y), exitDoor(x, y), r_{door}, exitDoorOpened)
  r_{switch} = DISTREWARD(player(x, y), lastSwitch(x, y), r_{exit}, lastSwitchActive)
  return $r_{time} + r_{key} + r_{door} + r_{exit} + r_{switch}$
end function

function TIMEREWARD(t)
  `reward` = $r_{max} \div t$
  if `reward` $< \tau$ **then**
    return 0
  else
    return `reward`
  end if
end function

function DISTREWARD(*current*(x, y), *target*(x, y), *lastReward*, *condition*)
  if *lastReward* ≤ 0 **then**
    return 0
  else if *condition* == `True` **then**
    return r_{max}
  end if
  `dist` = Euclid(*current*(x, y), *target*(x, y))
  `reward` = $r_{max} \div \max($`dist`$, 1.0)$
  if `reward` $< \tau$ **then**
    return 0
  else
    return `reward`
  end if
end function

one task, not a curriculum of multiple tasks, hence the number of task exposures is expressed as generations rather than "bags." The Take Cover task also applied the mutation operations five times per child, where this was observed to accelerate evolution irrespective of the TPG formulation.[18]

[18]Pathfinding study performed before this was discovered.

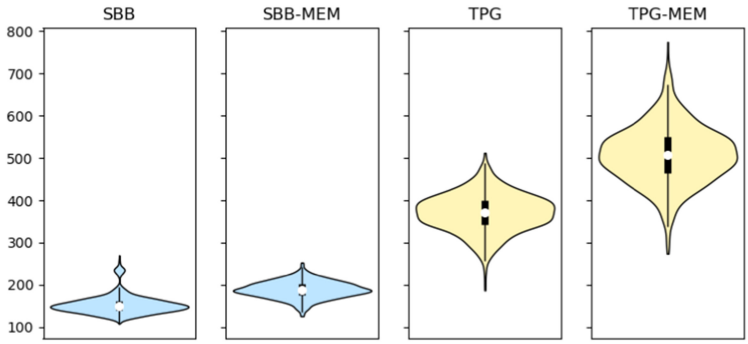


Fig. 13. Ablation study test performance on Take Cover task. Violin captures the distribution of all test runs from each ablation. Inner box plot represents the 25th, 50th, and 75th quartile.

4.6 Results under Take Cover task

The Take Cover task is sufficiently concise yet challenging to enable an ablation study to be performed in which the following scenarios are considered:

- SBB: represents the case of TPG limited to evolving single teams. Modularity is still present, but the capacity to evolve hierarchical relationships is disabled. This represents the case of symbiotic bid-based GP, a formulation that was previously demonstrated to be capable of performing significantly better than GP without modularity under classification [Lichodzijewski and Heywood 2010] and non-stationary streaming data classification tasks [Vahdat et al. 2015].
- SBB-MEM: adds the probabilistic indexed memory model to SBB.
- TPG: represents the case of TPG without the probabilistic indexed memory model (Section 2).
- TPG-MEM: adds the probabilistic indexed memory model to TPG 4.1.

Ten independent runs are performed for each ablation and the champion agent then exposed to 100 test trials in which it is spawned at different points on the wall. Figure 13 summarizes the resulting test performance (agent lifespan) where larger values are better. The distributions are sufficiently normal for a Student's t-test to be applied with significance at the 99% confidence interval supporting a ranking of: TPG-MEM > TPG > SBB-MEM > SBB. In short, going from no memory to memory provides a 23% (SBB) to 36% (TPG) improvement, whereas supporting hierarchical modular relationships provides a 140% (no memory) to 168% (with memory) improvement.

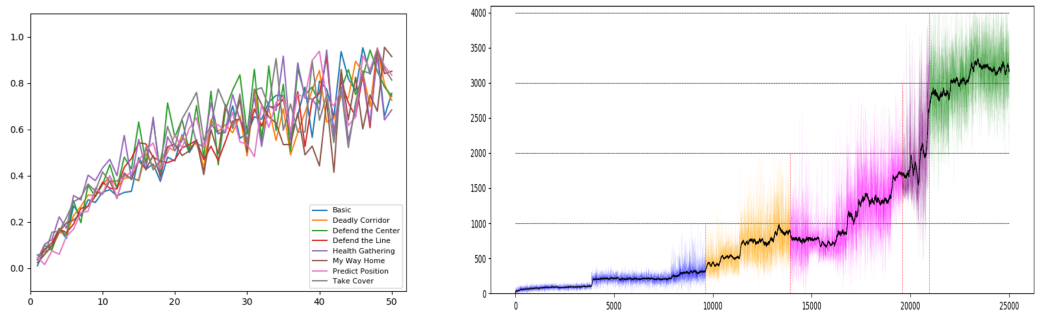
The relative complexity of the respective champion solution can also be characterized, Table 5. In short, adding memory to SBB did not result in significant increases to the complexity of SBB solution statistics. Similarly, TPG complexity with and without memory was also for the most part very similar. The one exception being with regards to the average instructions executed per decision (last row). This was 47% higher under TPG without memory. Naturally, TPG was always more complex than the corresponding SBB scenario, although as indicated by the figures for the best case champion, these could also be simple.

In conclusion, support for hierarchical modularity appears to be important when scaling to high-dimensional tasks as it provides a mechanism for contextually organizing teams and programs (e.g., only 4 of 14 teams visited or 28 of 92 programs executed per decision under TPG-MEM). Memory provides the opportunity to co-ordinate state information globally across their multiple

Table 5. Solution Complexities under Take Cover Task

Configuration	SBB	SBB-MEM	TPG	TPG-MEM
Num. Teams	1 (1)	1 (1)	15.4 (11)	14.4 (10)
Num. Programs	4.8 (4)	6 (9)	101.2 (52)	92.7 (35)
Programs/Team	4.8 (4)	6 (9)	6.8 (4.73)	6.58 (3.5)
Instructions/Program	716.6 (464.75)	698.9 (828)	626.2 (742)	598.8 (862.5)
Instructions/Team	3,571 (1,859)	4,086 (7,452)	4,115 (3,507)	3,932 (3,018)
Median Teams visited/Decision	1 (1)	1 (1)	3.9 (3)	4 (2)
Avg. Programs exe./Decision	4 (4)	6 (9)	30.7 (18)	28 (7)
Avg. Instr./Decision	3,571 (1,859)	4,086 (7,452)	19,647 (11,030)	13,294 (3,283)

Each "value" represents the average over all 10 solutions, parenthesis represents the value for the single best champion in each ablation. "Configuration" defines the ablation. Rows 2 to 6 represent static properties. Rows 7 to 9 capture dynamic properties.



(a) Source Task Fitness

(b) Navigation Task Fitness

Fig. 14. Training fitness of best TPG-agent on (a) 10 source tasks and (b) E1M2 navigation task. Colours in panel (b) represent different parts of the navigation task being satisfied. Bold back line represents average performance.

independent modules, thus providing the basis for retaining state information beyond any single frame. Both properties appear to be beneficial both independently and collectively.

4.7 Results under Pathfinding Task

4.7.1 Training. Training is initially performed against the bag of eight source tasks or training curricula for 50 iterations of bag content. Figure 14(a) summarizes progress against these source tasks. Specifically, the performance of the best overall agent is plotted, with bag fitness normalized by the best single TPG-agent on that bag. That is to say, the performance of the best specialist agent is used to normalize the performance of the TPG-agent with best average performance across all 8 source tasks. The underlying curve improves, however, the trajectory for performance of specific source tasks varies as the TPG-agent with best average performance switches and/or improvements on some subset of the source tasks improves at the expense of a single source task.

Figure 14(b) illustrates the development of the champion TPG-agent relative to the Pathfinding fitness function (Algorithm 4) on the E1M2 map (Figure 12(b)). The red key is first successfully picked up w.r.t. stochastic spawn points anywhere in the white zone (Figure 12(b)) at $\approx$ 9,600 generation. The key is first successfully used to open the door to enter the yellow zone at $\approx$ 13,900 generation. The exit door is first discovered at $\approx$ 19,580 generation and the exit switch at around generation 20,900. The last $\approx$ 4,000 generations are used to improve on the path time.

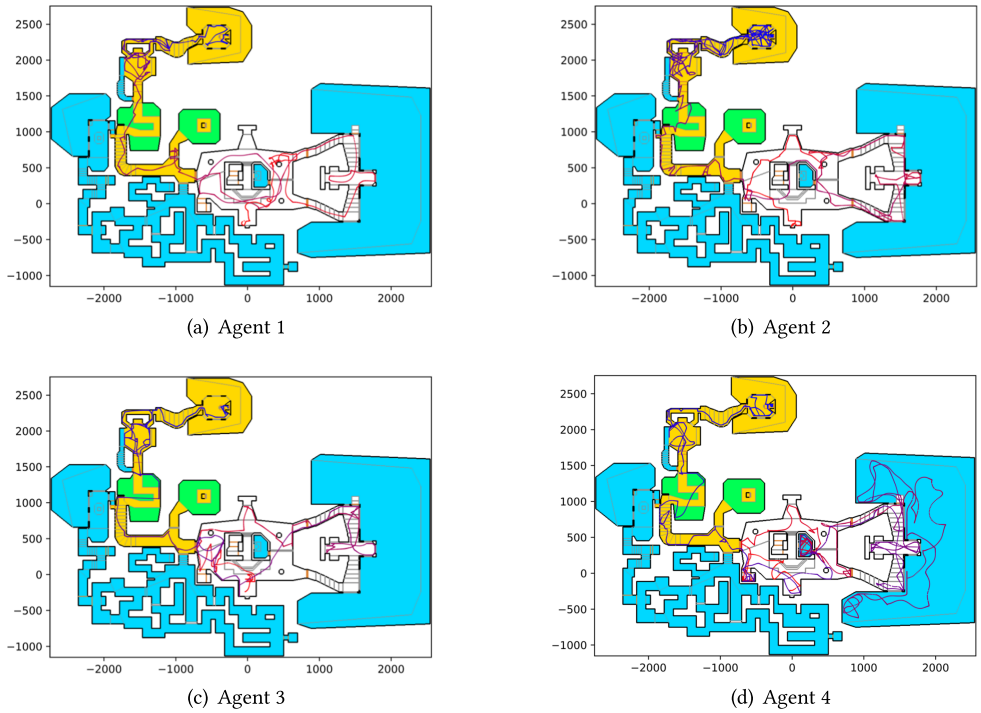
Table 6. Performance Properties of Top 20 TPG Agents under Level 2 E1M2 Map

Training Agent Rank	Avg. Time (min)	Avg. Health Items	Avg. # Ammo. Items	Avg. # Armour Items	Avg. # Enemies Killed	Avg. # Barrels Destr.	Final Health	Final Armour	Succ. Ratio
1	3.04	10	8	6	9	3	80	161	100
3	4.34	10	8	5	5	3	81	157	100
13	4.46	7	6	5	4	5	67	113	100
2	5.81	7	9	5	6	4	80	155	100
7	5.87	10	2	1	2	5	77	151	100
11	7.42	6	3	2	2	5	74	133	100
5	7.66	10	7	3	3	5	83	153	100
4	8.11	14	11	7	5	5	147	158	100
6	5.61	8	8	4	4	5	75	164	95.2
9	7.91	8	5	3	9	5	74	91	95.2
19	5.15	4	2	3	3	4	79	87	90.9
10	8.59	6	6	6	6	5	74	145	90.9
15	5.31	9	2	4	4	4	68	136	87.0
8	6.625	14	6	4	4	5	76	152	87.0
17	4.88	7	2	3	8	4	69	97	83.3
16	6.97	6	3	4	4	5	70	61	83.3
14	7.92	7	4	4	4	5	71	71	83.3
12	8.23	8	6	4	4	4	75	130	83.3
18	8.61	7	4	3	3	4	66	81	83.3
20	5.39	5	4	3	1	3	67	107	74.1
mean	6.395	8.15	5.3	3.95	4.5	4.4	77.65	123.25	91.8
median	6.25	7.5	5.5	4.0	4.0	5.0	74.5	133.0	93.1
StdDev	1.62	2.56	2.57	1.4	2.13	0.74	16.7	31.9	8.2
SD/Mean	25.4%	31.5%	48.5%	35.3%	47.4%	16.7%	21.5%	25.9%	8.35%

All columns reflect the average performance as evaluated across 20 successful navigations of the level. Better Times are lowest. # Health/Ammo./Armour/Enemies Killed/Barrels Destroyed all reflect counts of items that could be maximized as part of a strategy. Final Health/ Armour are the average final values of these properties once the agent completes the level. Larger values are better. Success Ratio is the ratio of successful trials (always 20) to unsuccessful trials. An unsuccessful trial would be one in which the agent's health reached zero before exiting the level.

4.7.2 Test on previously seen map. Performance of the top 20 agents from the population with respect to the previously encountered level 2 map is summarized in Table 6, where "rank" are identified in terms of training fitness (Algorithm 4). Test conditions take the form of selecting a spawn point (and orientation) from zone 1 (Figure 12(b)) with uniform probability, and measuring the simulated time to successfully reach the level exit. The process is repeated until 20 successful navigations are returned, i.e., some attempts might fail, due to agent health decreasing to zero before reaching the exit. The performance of the top 20 agents is grouped into sets reflecting the level of navigation success. Thus, agents with a 100% success ratio solved all 20 initializations immediately, an agent with a success ratio of 87% needed to experience 23 initializations to return 20 successful level navigations. In the case of each evaluation, memory content is initialized to the value at the end of training.

In all cases, the fitness function employed during training is the same. However, some agents might adopt a navigation strategy that collects health packs and armour, resulting in a longer path, but one that, say, makes use of health packs to correct actions associated with the path taken. This appears to be the case of the agent with the slowest navigation time at the 100% success ratio



(a) Agent 1 (b) Agent 2

(c) Agent 3 (d) Agent 4

Fig. 15. Example of paths taken by top four ranked TPG agents in E1M2 Pathfinding task. Start position is at co-ordinate (0, −300). Key is located at co-ordinate (1,150, 400). The door that can only be opened with the key is at co-ordinate (−700, 375). Level exit is at co-ordinate (−150, 2,300). Path colour transitions from red to blue indicate the progression of time.

(agent 4). Conversely, the fastest agent has a much lower final health than agent 4 but also removes the highest number of opponent bots (9), while also collecting health, ammunition and armour items.

Some interesting "specialist" agents appear. Agent 7 has a success ratio of 100%, but achieves this while for the most part ignoring armour entirely. All agents at some level "engage" with acid barrels, in some cases shooting them. It is difficult to tell whether this is because some of the opponent agents are also the same colour (green), or if this is part of a navigation strategy (the location of barrels does not change).

Figure 15 illustrates the specific path used to navigate by agents 1 through 4 for the same spawn point. Agent 1 was the top ranked agent under both training and test. The agent ignores the "hidden" zones and is comparatively direct in the path adopted. Indeed, when it comes to the acid bath in zone 2, the agent chooses to cross it at the shortest point. Agent 2 wanders around the latter half of zone 2, even jumping in the acid bath twice, but does immediately collect the health pack after exiting the bath for the last time. Agent 3 has a particularly direct approach to zone 2, but spends much more time in zone 1, returning the second ranked navigation time. Agent 4 returns the longest of the navigation times for the agents with 100% success rate. Some of this appears to be due to a detour into and out of one of the hidden zone attached to zone 1, and a considerable amount of time also seems to be spent in the room at co-ordinate (−500, 0). For comparison the "Par time" for E1M2 is 1 min, 15 s, where this reflects the time of the level designer (John Romero), plus 30 s,[19] whereas the best time of agent 1 was 2.8 min (168 s).

[19] https://doom.fandom.com/wiki/Par_time.

Table 7. Structural Properties of Top 20 TPG Agents under Pathfinding Task

Statistic	# Teams	# Programs	# Instructions	Prog./Team	Instr./Prog.	Instr./Team
Mean	156.65	1,302	880,946.6	8.33	676	5,626
Median	158.0	1,306.5	874,773.5	8.42	674.9	5,610.6
Std Dev.	7.1%	13.2%	16.6%	12.1%	10.5%	15.3%

Table 8. Summary of Number of Agent outcomes in the Previously (Unseen) Level 1 Pathfinding Task

Level 1	Success		Death		Time Out	
Parameterization	all	agent 1	all	agent 1	all	agent 1
Initial	41 (10.25%)	4 (20%)	201 (50.25%)	7 (35%)	158 (89.75%)	9 (45%)
Modified	73 (18.25%)	11 (55%)	143 (37.75%)	7 (35%)	184 (46%)	2 (10%)

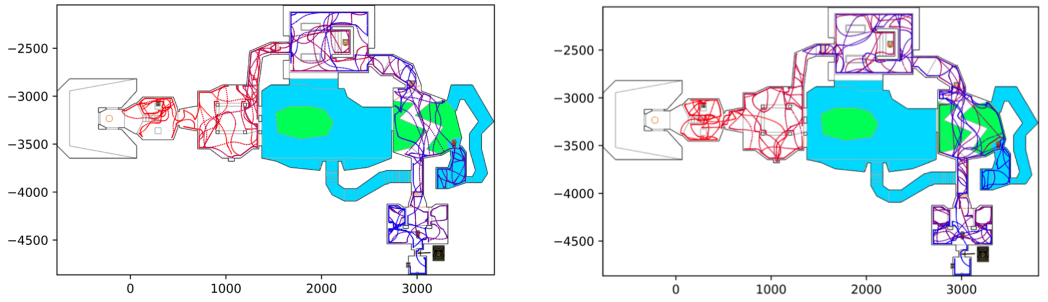
"all" represents a count over all 20 agents, each initialized 20 times (total of 400 tests). "agent 1" is the contribution of agent 1 alone (best performing agent under level 2). "Initial" is the parameterization of level 1 as is. "Modified" re-parameterizes the level to use the same wall/floor texture/colour and switch function as experienced by the agent under training conditions.

Table 7 provides a summary of the size of the top 20 agents. The first three columns represent total number of teams/programs/instructions per agent, whereas the last three columns represent averages at a node or arc level. Note, however, that these are both "static" performance metrics. To make a decision (map from state-to-action) only a fraction of the TPG agent's graph need be evaluated. Supplementary material provides a video of Agent 1 traversing E1M2 and the corresponding path navigation superimposed on the global map. Relative to the static characterization of TPG complexity (Table 7), there were on average 24 teams visited per decision (of a median of 158), during which 208 of the 1,300 programs were (on average) evaluated.

4.7.3 Test on previously unseen map. Performance is now tested using the previously never experienced level 1 map (Figure 12(a)). The level 1 map represents a completely different configuration and parameterization than experienced from level 2. Parameters include the layout of objects (barrels, health packs, opponent Bots, armour) as well as the types of switches used to activate doors and the texture/colour given to the floors and wall. Given that no prior experience existed under this parameterization, the level 1 map was modified to provide the same wall/floor texture/colour as experienced by the agents during training. Likewise, the switches were also updated to ensure continuity of function (e.g., door enabling versus exit door enabling). Note that the *layout* of the level is *unchanged* by either modification.

Table 8 summarizes the overall performance evaluation in terms of the number of times that agents either successfully navigated to the level exit versus died (e.g., acid bath or hits from an opponent Bot) versus a session time out (max. of 40 simulated minutes). The most successful agent is again "Agent 1." Figure 16 provides two illustrative summaries of the path taken by this agent. In both cases the agent systematically performs a sweep of a room before "deciding" where to go next. What differentiates between the length of the episode is that in Figure 16(a) agent 1 found the switch to open the level exit door immediately, whereas in Figure 16(b) the same agent failed to recognize the switch. This ultimately resulted in the agent backtracking several rooms before finally returning to the exit room, where it found the switch, operated it and successfully completed the level.

Table 9 summarizes statistics from 20 *successful* trials on the previously unseen E1M1 level by agent 1 (success ratio of 40%). Given the much longer duration to navigate the level, the lower final Armour and Health is anticipated. However, there are also less items to accumulate/interact with



(a) 22 minute path (b) 33 minute path

Fig. 16. Agent 1 navigating the previously unseen E1M1 level. Agent is stochastically spawned in left most room. Level exit is always in room co-ordinate (3,000, −4,900) (a) Illustrates the systematic room-by-room search that Agent 1 conducts. (b) Illustrates a search that took longer due to the agent doubling back and repeating some of the search.

Table 9. Performance of Agent 1 on Previously Unseen E1M1 Level 1 Map

Metric	Avg. Time (min)	Avg. Health Items	Avg. # Ammo. Items	Avg. # Armour Items	Avg. # Enemies Killed	Avg. # Barrels Destr.	Final Health	Final Armour
mean	26	11	3	10	3	3	26	4
median	26	11	3	10	4	3	25	4
StdDev	4.88	2.21	0.99	1.63	0.75	1.4	16.25	2.23
SD/Mean	19.0%	21.1%	31.4%	16.6%	22.2%	50%	62.5%	50.8%

Results collected over 20 successful runs. Success ratio was 40%.

in this level than level 2. That said, the average number of Armour Items accumulated per trial is significantly higher in level 1 than in 2.

5 CONCLUSION

TPG provides the basis for discovering complex systems through emergent modularity, or a divide-and-conquer approach to model building. Thus, modules are synonymous with programs and are adaptively coevolved to form the highly-modular hierarchical decomposition of a task. TPG was originally demonstrated within the context of visual reinforcement learning tasks using the ALE suite of benchmarks [Kelly and Heywood 2017, 2018b]. In this work, two approaches are presented in which the original TPG framework (Section 2) is generalized to support: (1) real-valued actions under partially observable multi-task recursive time-series forecasting tasks or TPG-MT (Section 3), and (2) visual reinforcement learning tasks requiring the development of multiple skills over long temporal horizons using a probabilistic indexed memory model (Section 4). The approaches are distinct and complementary.

TPG-MT introduces multiple types of program/module, those that may read and write to their own bank of indexed temporal memory, and those that can only read from memory, but define a real valued action (both can read from external state). The TPG graph emerges as before (care of action mutation). However, the structure of the TPG graph now also helps to establish the order in which the state of local banks of memory are read from/written to. We demonstrate for the first time that such an architecture is sufficient to learn to generate *multiple* non-stationary time-series sequences without assuming an autoregressive representation of state. An ablation study

demonstrates that only when the hierarchical form of modularity central to TPG is provided can these behaviours emerge.

Conversely, the probabilistic indexed memory model assumes that any program may perform a read or write to indexed memory. Moreover, the probabilistic definition of write operation results in direct support for short and long term memory.[20] However, there is only ever one instance of memory shared across all modules. Modules should therefore learn to be "respectful" in their write operations. As a result, memory acts as a global communication medium between all modules, otherwise modules themselves would only be aware of their own state. The resulting empirical evaluation demonstrates for the first time that TPG agents have the capacity to develop strategies for solving a multi-faceted navigation problem that is high dimensional (>76,000), partially observable, non-stationary and requires the recall of events from long term memory (key-to-door and switch-to-door association as well as navigation). Moreover, the policy is sufficiently general to support navigation under a previously unseen task (E1M1 from E1M2). An additional ablation study demonstrates that: (1) TPG's hierarchical form of modularity outperforms modularity as defined by teams alone (no hierarchical relationships between modules), and; (2) probabilistic memory provides significant performance improvements over agents without memory.

Relative to prior work involving genetic programming and modularity (Section 2.3), we make the observation that the earlier works have for the most part relied on toy tasks that had no independent assessment for post-training generalization.[21] As such these results have not carried the role of modularity beyond the confines of the specialist GP literature. Additionally, as the ablation studies make clear, the combination of modularity and memory provides a joined mechanism by which genetic programming may continue to scale. In taking up tasks such as non-stationary time series, ALE, VizDoom or controllers for benchmarks in robotics, we hope to motivate other researchers to adopt tasks that can increase the scope of genetic programming benchmarking.

The general flexibility of TPGs, their low computational footprint (CPU rather than GPU) [Desnos et al. 2021], and the availability of multiple code bases[22] provide an opportunity for future research to be conducted in multiple directions. Streaming data applications might encompass forecasting for any number of prediction/detection problems. Research into multi-task and curriculum learning will continue to scale for the development of TPG solutions to reinforcement learning applications. Likewise, the development of appropriate code bases that make use of parallel execution will further decrease application development time. A longer term objective encompasses the combination of multiple forms of memory within one TPG framework. This is not as straightforward as it might at first appear, because memory introduces feedback loops into (internal) state. The results of doing so can have unpredictable effects on agent operation, a feature that plays right to the strength of an evolutionary algorithm like the one discussed here.

REFERENCES

Alexandros Agapitos, Michael O'Neill, and Anthony Brabazon. 2012. *Genetic Programming for the Induction of Seasonal Forecasts: A Study on Weather Derivatives*. Springer, Boston, MA, 159–188.

Jinungn An, Jonghyun Lee, and Changwook Ahn. 2013. An efficient GP approach to recognizing cognitive tasks from fNIRS neural signals. *Sci. China Info. Sci.* 56 (2013), 1–7.

Björn Andersson, Per Svensson, Peter Nordin, and Mats G. Nordahl. 1999. Reactive and memory-based genetic programming for robot control. In *Proceedings of the European Conference on Genetic Programming (LNCS, Vol. 1598)*. Springer, 161–172.

Peter J. Angeline. 1994. Genetic programming and emergent intelligence. In *Advances in Genetic Programming*, K. Kinnear (Ed.). Vol. 1. MIT Press, 75–98.

[20] Future work could consider different mechanisms by which short and long-term memory is supported.

[21] Tasks such as the lawnmower or minesweeper, bit manipulation problems such as parity or addition.

[22] https://web.cs.dal.ca/~mheywood/Code/index.html.

Wolfgang Banzhaf. 2014. Genetic programming and emergence. *Genetic Program. Evolv. Mach.* 15 (2014), 63–73.

Wolfgang Banzhaf, Dirk Banscherus, and Peter Dittrich. 1999. Hierarchical genetic programming using local modules. Retrieved from http://interjournal.org/manuscript_abstract.php?29953.

Markus Brameier and Wolfgang Banzhaf. 2001. Evolving teams of predictors with linear genetic programming. *Genetic Program. Evolv. Mach.* 2, 4 (2001), 381–407.

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

Scott Brave. 1996. The evolution of memory and mental models using genetic programming. In *Proceedings of the Annual Conference on Genetic Programming*. Morgan Kaufmann.

Werner Callebaut. 2005. The ubiquity of modularity. In *Modularity: Understanding the Development and Evolution of Natural Complex Systems*, Werner Callebaut and Diego Rasskin-Gutman (Eds.). MIT Press, Cambridge, MA.

Rohitash Chandra, Yew-Soon Ong, and Chi-Keong Goh. 2018. Co-evolutionary multi-task learning for dynamic time series prediction. *Appl. Soft Comput.* 70 (2018), 576–589.

Kim B. Clark and Carliss Y. Baldwin. 2000. *Design Rules. Vol. 1: The Power of Modularity*. MIT Press, Cambridge, MA.

Markus Conrads, Peter Nordin, and Wolfgang Banzhaf. 1998. Speech sound discrimination with genetic programming. In *Genetic Programming*, Wolfgang Banzhaf, Riccardo Poli, Marc Schoenauer, and Terence C. Fogarty (Eds.). Springer, Berlin, 113–129.

Karol Desnos, Nicolas Sourbier, Pierre-Yves Raumer, Olivier Gesny, and Maxime Pelcat. 2021. Gegelati: Lightweight artificial intelligence through generic and evolvable tangled program graphs. In *Proceedings of the Workshop on Design and Architectures for Signal and Image Processing*. ACM, 35–43.

Martin Dostál. 2013. Modularity in genetic programming. In *Handbook of Optimization*. Springer, 365–393.

Walter J. Gehring. 1992. The homeobox in perspective. *Trends Biochem. Sci.* 17 (1992), 277–280.

Walter J. Gehring and Kazuho Ikeo. 1999. Pax 6: Mastering eye morphogenesis and eye evolution. *Trends Genetics* 15 (1999), 371–377.

George Gerules and Cezary Janikow. 2016. A survey of modularity in genetic programming. In *Proceedings of the IEEE Congress on Evolutionary Computation*. IEEE Press, 5034–5043.

Faustino Gomez, Jürgen Schmidhuber, and Risto Miikkulainen. 2008. Accelerated neural evolution through cooperatively coevolved synapses. *J. Mach. Learn. Res.* 9 (2008), 937–965.

Faustino J. Gomez and Risto Miikkulainen. 1997. Incremental evolution of complex general behavior. *Adapt. Behav.* 5, 3–4 (1997), 317–342.

Anirudh Goyal, Alex Lamb, Jordan Hoffmann, Shagun Sodhani, Sergey Levine, Yoshua Bengio, and Bernhard Schölkopf. 2019. Recurrent independent mechanisms. Retrieved from https://abs/1909.10893.

Robin Harper and Alan Blair. 2006. Dynamically defined functions in grammatical evolution. In *Proceedings of the IEEE International Conference on Evolutionary Computation*. IEEE Press, 2638–2645.

Hrayr Harutyunyan, Hrant Khachatrian, David C. Kale, Greg Ver Steeg, and Aram Galstyan. 2019. Multitask learning and benchmarking with clinical time series data. *Sci. Data* 6, 1 (Dec. 2019), 96.

Luis J. Herrera, Hector Pomares, Ignacio Rojas, Alberto Guillen, Alberto Prieto, and Olga Valenzuela. 2007. Recursive prediction for long term time series forecasting using advanced models. *Neurocomputing* 70, 16 (2007), 2870–2880.

Malcolm I. Heywood. 2015. Evolutionary model building under streaming data for classification tasks: Opportunities and challenges. *Genetic Program. Evolv. Mach.* 16, 3 (2015), 283–326.

Malcolm I. Heywood and Peter Lichodzijewski. 2010. Symbiogenesis as a mechanism for building complex adaptive systems: A review. In *Proceedings of Applications of Evolutionary Computation (LNCS, Vol. 6024)*. Springer, 51–60.

Uwe Hübner, Nimmi B. Abraham, and Carlos 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.

Lorenz Huelsbergen. 1998. Finding general solutions to the parity problem by evolving machine-language representations. In *Proceedings of the Annual Conference on Genetic Programming*. Morgan Kaufmann, 158–166.

Max Jaderberg, Wojciech M. Czarnecki, Iain Dunning, Luke Marris, Guy Lever, Antonio García Castañeda, Charles Beattie, Neil C. Rabinowitz, Ari S. Morcos, Avraham Ruderman, Nicolas Sonnerat, Tim Green, Louise Deason, Joel Z. Leibo, David Silver, Demis Hassabis, Koray Kavukcuoglu, and Thore Graepel. 2019. Human-level performance in 3D multiplayer games with population-based reinforcement learning. *Science* 364 (2019), 859–865.

Baozhu Jia and Marc Ebner. 2017. Evolving game state features from raw pixels. In *Proceedings of the European Conference on Genetic Programming (LNCS, Vol. 10196)*. Springer, 52–63.

Nadav Kashtan, Elad Noor, and Uri Alon. 2007. Varying environments can speed up evolution. *Proc. Natl. Acad. Sci. U.S.A.* 104, 34 (2007), 13711–13716.

Stephen Kelly and Wolfgang Banzhaf. 2020. Temporal memory sharing in visual reinforcement learning. In *Genetic Programming Theory and Practice*, Wolfgang Banzhaf, Erik D. Goodman, Leigh Sheneman, Leonardo Trujillo, and Bill Worzel (Eds.). Vol. XVII. Springer, 101–120.

Stephen Kelly and Malcolm I. Heywood. 2017. Emergent tangled graph representations for atari game playing agents. In *Proceedings of the European Conference on Genetic Programming (LNCS, Vol. 10196)*. Springer, 64–79.

Stephen Kelly and Malcolm I. Heywood. 2018a. Discovering agent behaviors through code reuse: Examples from half-field offense and Ms. Pac-Man. *IEEE Trans. Games* 10, 2 (2018), 195–208.

Stephen Kelly and Malcolm I. Heywood. 2018b. Emergent solutions to high-dimensional multitask reinforcement learning. *Evolution. Comput.* 26, 3 (2018), 378–380.

Stephen Kelly, Jacob Newsted, Wolfgang Banzhaf, and Cedric Gondro. 2020. A modular memory framework for time series prediction. In *Proceedings of the Genetic and Evolutionary Computation Conference*. ACM Press, 949–957.

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*, Wolfgang Banzhaf, Lee Spector, and Leigh Sheneman (Eds.), Vol. XVI. Springer, Cham, 37–57.

Michal Kempka, Marek Wydmuch, Grzegorz Runc, Jakub Toczek, and Wojciech Jaskowski. 2016. ViZDoom: A doom-based AI research platform for visual reinforcement learning. In *Proceedings of the IEEE Conference on Computational Intelligence and Games*. IEEE Press, 1–8.

John R. Koza. 1992. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press.

John R. Koza. 1994. *Genetic Programming II: Automatic Discovery of Reusable Programms*. MIT Press.

John R. Koza, David Andre, Martin A. Keane, and Forrest H. Bennett III. 1999. *Genetic Programming III: Darwinian Invention and Problem Solving*. Vol. 3. Morgan Kaufmann.

William B. Langdon. 1998. *Genetic Programming and Data Structures*. Kluwer Academic.

William B. Langdon and Wolfgang Banzhaf. 2005. Repeated sequences in linear genetic programming genomes. *Complex Syst.* 15 (2005), 285–306.

William B. Langdon and Wolfgang Banzhaf. 2008. Repeated patterns in genetic programming. *Natural Comput.* 7, 4 (2008), 589–613.

Peter Lichodzijewski and Malcolm I. Heywood. 2007. Pareto-coevolutionary genetic programming for problem decomposition in multi-class classification. In *Proceedings of the Genetic and Evolutionary Computation Conference*. ACM Press, 464–471.

Peter Lichodzijewski and Malcolm I. Heywood. 2010. Symbiosis, complexification and simplicity under GP. In *Proceedings of the Genetic and Evolutionary Computation Conference*. ACM Press, 853–860.

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

Durga Prasad Muni, Nikhil R. Pal, and Jyotirmay Das. 2004. A novel approach to design classifiers using genetic programming. *IEEE Trans. Evolution. Comput.* 8, 2 (2004), 183–196.

Peter Nordin, Wolfgang Banzhaf, and Markus Brameier. 1998. Evolution of a world model for a miniature robot using genetic programming. *Robot. Auton. Syst.* 25 (1998), 105–116.

Michael O'Neill, Leonardo Vanneschi, Steven Gustafson, and Wolfgang Banzhaf. 2010. Open issues in genetic programming. *Genetic Program. Evolv. Mach.* 11, 3–4 (2010), 339–363.

David L. Parnas. 1972. On the criteria to be used in decomposing systems into modules. In *Pioneers and Their Contributions to Software Engineering*. Springer, 479–498.

Merav Parter, Nadav Kashtan, and Uri Alon. 2008. Facilitated variation: How evolution learns from past environments to generalize to new environments. *PLoS Comput. Biol.* 4, 11 (2008), e1000206.

Riccardo Poli, William B. Langdon, and Nicholas F. McPhee. 2008. *A Field Guide to Genetic Programming*. Lulu Press.

Mitchell A. Potter and Kenneth A. De Jong. 2000. Cooperative coevolution: An architecture for evolving coadapted subcomponents. *Evolution. Comput.* 8, 1 (2000), 1–29.

Justinian P. Rosca and Dana H. Ballard. 1996. Discovery of subroutines in genetic programming. In *Advances in Genetic Programming II*, Peter J. Angeline and Kenneth E. Kinnear (Eds.). MIT Press, Chapter 9, 177–201.

SIDC. 2020. World Data Center for the production, preservation, and dissemination of the international sunspot number. Retrieved from http://sidc.be/silso/home.

Herbert A. Simon. 1962. The architecture of complexity. *Proc. Amer. Philos. Soc.* 106 (1962), 467–482.

Herbert A. Simon. 2002. Near decomposability and the speed of evolution. *Industr. Corp. Change* 11 (2002), 587–599.

Herbert A. Simon. 2019. *The Sciences of the Artificial*. MIT Press, re-issued 3rd ed., Cambridge, MA.

Robert J. Smith and Malcolm I. Heywood. 2018. Scaling tangled program graphs to visual reinforcement learning in ViZDoom. In *Proceedings of the European Conference on Genetic Programming (LNCS, Vol. 10781)*. Springer, 135–150.

Robert J. Smith and Malcolm I. Heywood. 2019b. Evolving Dota 2 Shadow Fiend bots using genetic programming with external memory. In *Proceedings of the Genetic and Evolutionary Computation Conference*. ACM Press, 179–187.

Robert J. Smith and Malcolm I. Heywood. 2019a. A model of external memory for navigation in partially observable visual reinforcement learning tasks. In *Proceedings of the European Conference on Genetic Programming (LNCS, Vol. 11451)*. Springer, 162–177.

Robert J. Smith and Malcolm I. Heywood. 2020. Evolving a Dota 2 Hero Bot with a probabilistic shared memory model. In *Genetic Programming Theory and Practice XVII*, Wolfgang Banzhaf, Eric Goodman, Leigh Sheneman, Leonardo Trujillo, and Bill Worzel (Eds.). Springer International Publishing, 345–366.

Lee Spector, Kyle I. Harrington, and Thomas Helmuth. 2012. Tag-based modularity in tree-based genetic programming. In *Proceedings of the Genetic and Evolutionary Computation Conference*. ACM Press, 815–822.

Lee Spector and Sean Luke. 1996. Cultural transmission of information in genetic programming. In *Proceedings of the Annual Conference on Genetic Programming*. Morgan Kaufmann, 209–214.

Lee Spector, Brian Martin, Kyle I. Harrington, and Thomas Helmuth. 2011. Tag-based modules in genetic programming. In *Proceedings of the Genetic and Evolutionary Computation Conference*. ACM Press, 1419–1426.

Lee Spector and Alan J. Robinson. 2002. Genetic programming and autoconstructive evolution with the push programming language. *Genetic Programming and Evolvable Machines* 3, 1 (2002), 7–40.

Peter Stone. 2000. *Layered Learning in Multiagent Systems*. MIT Press.

John M. Swafford, Erik Hemberg, Michael O'Neill, Miguel Nicolau, and Anthony Brabazon. 2011. A non-destructive grammar modification approach to modularity in grammatical evolution. In *Proceedings of the Genetic and Evolutionary Computation Conference*. ACM Press, 1411–1418.

Matthew E. Taylor and Peter Stone. 2009. Transfer learning for reinforcement learning domains: A survey. *J. Mach. Learn. Res.* 10, 1 (2009), 1633–1685.

Astro Teller. 1994. The evolution of mental models. In *Advances in Genetic Programming*, K. E. Kinnear (Ed.). MIT Press, 199–220.

Russell Thomason and Terence Soule. 2007. Novel ways of improving cooperation and performance in ensemble classifiers. In *Proceedings of the Genetic and Evolutionary Computation Conference*. ACM Press, 1708–1715.

Andrew J. Turner and Julian F. Miller. 2017. Recurrent cartesian genetic programming of artificial neural networks. *Genetic Program. Evolv. Mach.* 18, 2 (June 2017), 185–212.

Ali Vahdat, Jillian Morgan, Andrew R. McIntyre, Malcolm I. Heywood, and A. Nur Zincir-Heywood. 2015. Evolving GP classifiers for streaming data tasks with concept change and label budgets: A benchmarking study. In *Handbook of Genetic Programming Applications*, Amir H. Gandomi, Amir H. Alavi, and Conor Ryan (Eds.). Springer, 451–480.

Günter P. Wagner and Lee Altenberg. 1996. Complex adaptations and the evolution of evolvability. *Evolution* 50 (1996), 921–931.

Neal Wagner, Zbigniew Michalewicz, Moutaz Khouja, and Rob R. McGregor. 2007. Time series forecasting for dynamic environments: The DyFor genetic program model. *IEEE Trans. Evolution. Comput.* 11, 4 (Aug. 2007), 433–452.

James A. Walker and Julian F. Miller. 2008. The automatic acquisition, evolution and reuse of modules in cartesian genetic programming. *IEEE Trans. Evolution. Comput.* 12, 4 (2008), 397–417.

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

Shimon Whiteson, Nate Kohl, Risto Miikkulainen, and Peter Stone. 2005. Evolving keepaway soccer players through task decomposition. *Mach. Learn.* 59, 1 (2005), 5–30.

Dennis G. Wilson, S. Cussat-Blanc, H. Luga, and J. F. Miller. 2018. Evolving simple programs for playing Atari games. In *Proceedings of the Genetic and Evolutionary Computation Conference*. ACM Press, 229–236.

John R. Woodward. 2003. Modularity in genetic programming. In *Proceedings of the European Conference on Genetic Programming (Lecture Notes in Computer Science, Vol. 2610)*. Springer, 254–263.

Shelly X. Wu and Wolfgang Banzhaf. 2011. Rethinking multilevel selection in genetic programming. In *Proceedings of the Genetic and Evolutionary Computation Conference*. ACM Press, 1403–1410.

Marek Wydmuch, Michal Kempka, and Wojciech Jaskowski. 2019. ViZDoom competitions: Playing doom from pixels. *IEEE Trans. Games* 11, 3 (2019), 248–259.

Received May 2020; revised March 2021; accepted May 2021