# Emergent Tangled Graph Representations for Atari Game Playing Agents

Stephen Kelly[1] and Malcolm I. Heywood[1]

[1]*Faculty of Computer Science, Dalhousie University, Halifax, NS. Canada*

Article originally appears at EuroGP'17 (LNCS 10196) under Springer copyright 2017
http://link.springer.com/chapter/10.1007/978-3-319-55696-3_5

**Abstract**

Organizing code into coherent programs and relating different programs to each other represents an underlying requirement for scaling genetic programming to more difficult task domains. Assuming a model in which policies are defined by teams of programs, in which team and program are represented using independent populations and coevolved, has previously been shown to support the development of variable sized teams. In this work, we generalize the approach to provide a complete framework for organizing multiple teams into arbitrarily deep/wide structures through a process of continuous evolution; hereafter the Tangled Program Graph (TPG). Benchmarking is conducted using a subset of 20 games from the Arcade Learning Environment (ALE), an Atari 2600 video game emulator. The games considered here correspond to those in which deep learning was unable to reach a threshold of play consistent with that of a human. Information provided to the learning agent is limited to that which a human would experience. That is, screen capture sensory input, Atari joystick actions, and game score. The performance of the proposed approach exceeds that of deep learning in 15 of the 20 games, with 7 of the 15 also exceeding that associated with a human level of competence. Moreover, in contrast to solutions from deep learning, solutions discovered by TPG are also very 'sparse'. Rather than assuming that *all* of the state space contributes to every decision, each action in TPG is resolved following execution of a subset of an individual's graph. This results in significantly lower computational requirements for model building than presently the case for deep learning.

## 1 Introduction

Machine learning agents applied to a reinforcement learning (RL) task attempt to maximize the reward accrued over a training episode, during which a variable number of interactions with the task environment occur. In each interaction, the agent observes the state of the environment, takes an action, and receives feedback in the form of a reward signal. It is typically only the final reward, received when an end state (or max. interactions) is encountered, that quantifies the agent's performance relative to the task objective. The agent is said to represent a *policy* for maximizing this long-term reward.

Scaling RL to real-world tasks requires a representation that is: (1) Able to cope with high-dimensional sensor data; and (2) General enough to be applied to a wide variety of tasks without extensive parameter tuning. Video games provide an interesting test domain for scalable RL. In particular, they cover a diverse range of dynamic task environments that are designed to be challenging for humans, all through a common high-dimensional visual interface, or the game screen, e.g. [1].

In this work we propose a Genetic Programming (GP) framework to address the scaling problem through emergent modularity [16]. Specifically, we adopt a graph representation for decision-making policies, in which teams of programs are capable of growing and self-organizing into complex structures through interaction with the task environment. The framework is capable of:

- Adaptively dividing the task up into distinct sets of cooperating programs (or *teams*), an emergent process as there is no knowledge regarding the correct team size/complement. Teams represent the smallest stand-alone decision-making entity. As such, a team is synonymous with a *module* in the scope of this paper.
- Establishing how to select between/recombine teams into increasingly complex decision making structures, or a *policy graph*. Throughout this work, we refer to this process as emergent modularity because multiple

independent teams are recombined with no prior knowledge regarding how many to include, which teams might work well together, or how to combine them.

Hence, the process starts off with very simple programs/teams and then adapts to develop graphs of connectivity between teams, creating policy graphs which are themselves subject to further development.

The concept is partly motivated by the intuition that (automatic) problem decomposition is an important learning skill for artificial agents, just as it is for humans. The algorithm proposed in this work, Tangled Program Graphs (TPG) is an extension of Symbiotic Bid-Based GP (SBB), a framework for automatic problem decomposition through coevolving teams of simple programs. Specifically, TPG facilitates a more open-ended approach, in which teams are incrementally organized into graphs of arbitrary topology and discovered using a single continuous cycle of evolution. Thus, more complex topologies can naturally emerge as soon as they perform better than simpler solutions.

The scope of this work is to introduce TPG and make a case for how the representation supports the development of strong yet simple policies. Empirical experiments are conducted in the Arcade Learning Environment (ALE) [1]. ALE provides a framework in which RL agents have access to hundreds of classic video games through a common sensory interface: the game screen as a high-dimensional *pixel matrix*. Moreover, actions are limited to those of the original Atari console joystick, and the ultimate feedback takes the form of game score. In short, RL is limited to the same set of experiences as a human (albeit without sound). We make a direct comparison with both Neuroevolution [5] and a recent Deep Reinforcement Learning architecture [15], and show that the TPG produces competitive agents at a fraction of the model complexity.

The remainder of this paper is organized as follows: We begin by summarizing related GP research regarding modularity in Section 2. The properties of the ALE Atari 2600 task that warrant its use as a challenging RL benchmark are then established in Section 3, as well as the specific representation assumed for the state space. Section 4 presents the framework for evolving Tangled Program Graphs, and the empirical study is performed in Section 5. Conclusions and future work are outlined in Section 6.

## 2 Background

Modular architectures are a recurring theme in GP, with early approaches such as Automatically Defined Functions [11] and Adaptive Representations through Learning [19], as well as Tag-Based Modules [21] all being motivated by the challenge of scaling GP to more complex tasks. From the perspective of modular task decomposition through teaming, previous studies have established guidelines for combining the contribution from individual team members [3] or enforcing island models [6]. Attempts have also been made to define fitness at the program as well as the team level [24, 23]. In a final development, simple bidding mechanisms have been used to guarantee that task decomposition takes place between members of a team [12, 13, 24]. This latter approach also implies that each program establishes *context* for one discrete action (i.e. context and action are independent), and there is no need to a priori define the optimal team size or relevant distribution of actions, hence the composition of a team is now an entirely evolved property.

Recent work has established the utility of using team building through bidding (or SBB in particular) to build *hierarchical* decision making agents over two independent phases of evolution [9, 8, 14, 4, 10]. The first phase produces a library of diverse, specialist teams. The second phase attempts to build more general policies by reusing the library. While effective in many tasks, this approach makes several assumptions that potentially impact its generality:

1. Individuals at the second phase of evolution can only define actions in terms of the teams evolved at the first (in order to place plausible limits on the number of actions).
2. The number of phases of evolution/levels in the hierarchy is guessed a priori and to date has not passed two (a law of diminishing returns per phase of evolution is typically observed).
3. The computational budget (generation or evaluation limit) used for each phase of evolution needs defining a priori.

The work herein builds on these previous studies to propose a new team-based GP representation, or a *tangled* program graph, in which teams denote vertices, and programs identify which edges are traversed. The topology of a solution, i.e. the number of teams per graph and the connectivity between teams, is now entirely the result of open ended evolution.

In the case of the task domain used for benchmarking, we assume the Arcade Learning Environment (ALE) for the Atari 2600 console [1]. High scores in the ALE are currently dominated by neural network architectures, specifically Deep Reinforcement Learning (DQN) [5] and Neuroevolution (HyperNEAT) [15]. Both methods assume an exhaustive accounting for the entire input space regardless of the agent's experience in the environment, i.e. the convolution network in Deep Learning or the Compositional Pattern Producing Network in HyperNEAT. Conversely, TPG attempts to discover a suitable input representation through interacting with the task, while simultaneously discovering an appropriate decision making policy. In this initial study, we explicitly target the subset of 20 games for which DQN did not reach the threshold of human-level play [15].

# 3 The Arcade Learning Environment

Released in 1977, the Atari 2600 has been a popular home video game console that was capable of running a large variety of games, each stored on interchangeable ROM cartridges. Hundreds of games were compatible with the console, bringing the diversity of an Arcade experience into the home through a single device. As each game is designed to be unique and challenging for human players, the Atari 2600 provides an interesting test domain for general artificial decision making agents.

The Arcade Learning Environment (ALE) provides an Atari 2600 emulator with a common reinforcement learning interface [1]. In particular, ALE allows learning agents to interact with games over discrete time steps by extracting the current game state and score from ROM, and sending joystick commands to the emulator. The Atari 2600 joystick comprises a directional paddle and single push button, which ALE translates into 18 discrete actions, or all possible combinations of direction and push button state, including 'no action'. The task is particularly challenging because any learning agent is required to operate under the same conditions as a human player, i.e. sensory input (screen), action output (joystick), and game score.

## 3.1 Screen state space representation

Agents in this work observe the game via the most general state representation, or the raw Atari 2600 screen, which is a $210 \times 160$ pixel matrix, with 128 possible colour values for each pixel, updated at 60Hz. In common with previous research, our agents interact with games at reduced frame rate, stochastically skipping $\approx 25\%$ of sequential frames, which is roughly the fastest that a human player can react. The most recent action is simply repeated in skipped frames[1]. Skipping frames in this manner implies that the environment is stochastic.

The task is considered partially observable because agents only perceive one frame at a time, while game entities typically appear intermittently (flicker) over sequential frames[2]. Hence, it is often impossible to capture the complete game state from a single frame. While various methods for hand-crafting feature sets from the raw Atari screen frames are possible, including game-specific background and object detection [1, 5], the focus of this work is learning from high-dimensional, task-independent sensory representation. However, we can reduce the dimensionality significantly by preprocessing frames based on the observation that most game entities are larger than a single pixel, and thus less resolution is required to capture important games events. Such an observation does not imply the use of image processing operators or adopting representations specifically appropriate for spatial representation, e.g. wavelets [22].

The screen quantization procedure assumes the following steps:

1. Each frame is subdivided into a $42 \times 32$ grid (Figure 1(b)), in which only 50% of the pixels in each tile are considered (implies that most state information is redundant) and each pixel assumes an 8-colour SECAM[3] encoding.

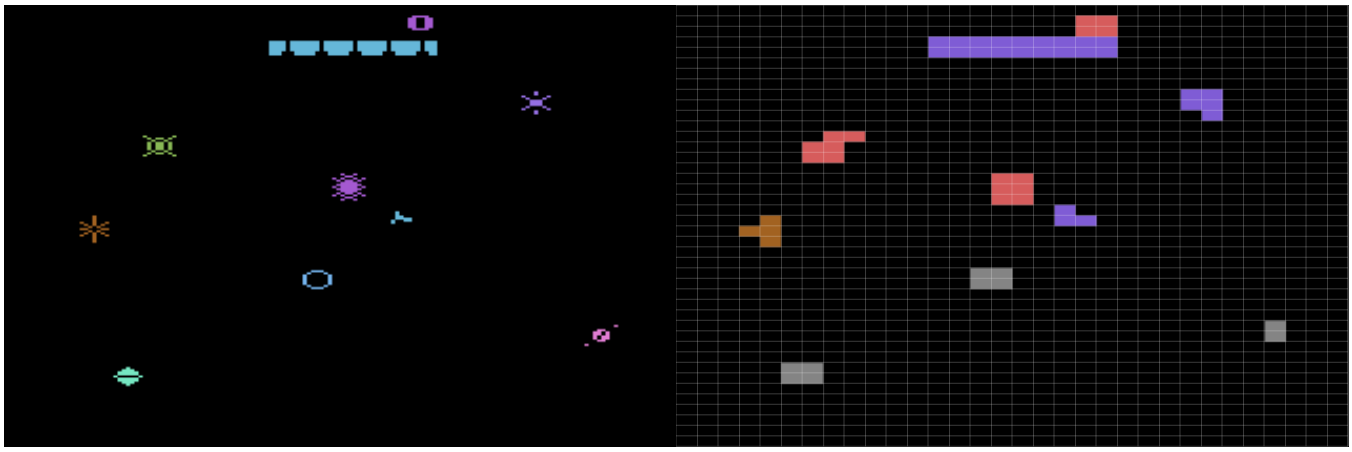
2. Each tile is described by a single byte, in which each bit encodes the presence of one of eight SECAM colours within that tile.

3. The decimal value for each of the tile bytes is returned, so defining a sensory state space $\vec{x}(t)$ of $42 \times 32 = 1344$ decimal features in the range of $0 - 255$, visualized in Figure 1(b) for a specific game at time step (frame) $t$.

This state representation is inspired by the Basic method defined in [1]. Note, however, that this method does not use a priori background detection or pairwise combinations of features.

[1] ALE includes a parameter *repeat_action_probability*, for which we assumed the suggested value of 0.25.

[2] Partial observability can be mitigated by averaging pixel colours across each pair of sequential frames, a preprocessing step *not* used in



(a) Atari Screen (b) Decimal Feature Grid

Figure 1: Screen quantization steps, reducing the raw Atari pixel matrix (a) to 1344 decimal sensor inputs (b).

# 4 Evolving Tangled Program Graphs

The proposed framework, Tangled Program Graphs (TPG), can be summarized by two key concepts: (1) Coevolving teams of programs, which represent single nodes of the graph, Figure 2(b); and, (2) Emergent modularity, or the process by which the graph is incrementally constructed, Figure 2(a).

## 4.1 Coevolving Teams of Programs

Evolution begins with a population of single independent teams. Each team is initialized with a stochastically chosen complement of programs over the interval $[2, \dots \omega]$. For example, Figure 2(b) represents one such candidate team consisting of 5 programs.

A linear, or register machine, representation will be assumed for programs, where linear GP provides an efficient process for skipping intron code *during* execution [2]. Each program returns a single real number, the result of executing a sequence of instructions that operate on sensor inputs or internal registers, as illustrated in Algorithm 1. Each program is also associated with *one* task-specific 'atomic' action, selected from the set of discrete actions defined by the task domain (corresponding to 18 console directions with/without the 'fire' action and a 'no action', Section 3).

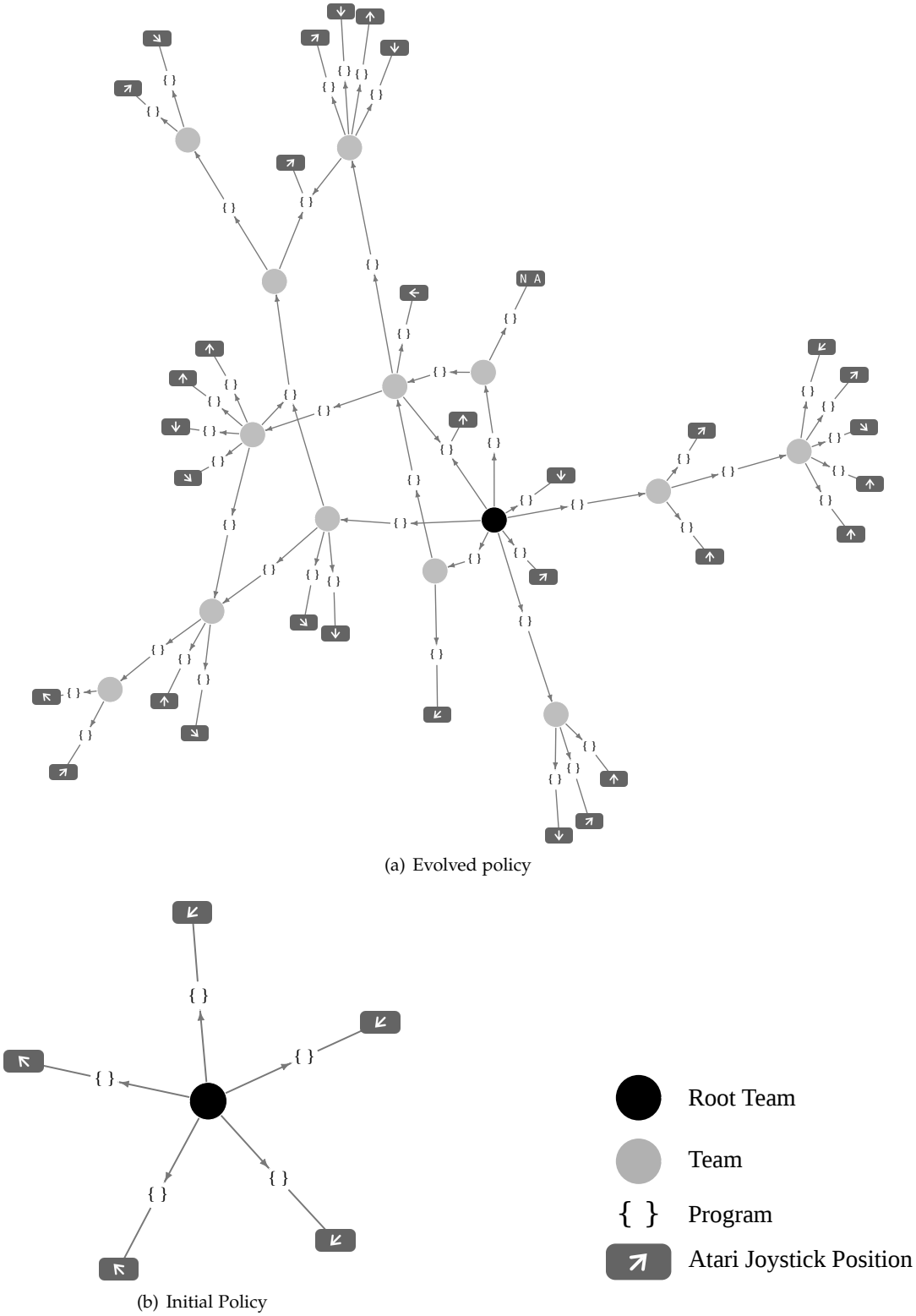
In RL tasks, all programs in a team will execute relative to the current state variables (screen) at each time step, $\vec{x}(t)$. The team then deploys the action of the program with the highest output, or the **winning bid** [12, 13, 24]. Note that the winning bid merely defines the action to deploy at time step $t$. This potentially changes the state of the task, which may or may not represent the end of the training epoch (for which a measurable reward is received). In short, teams represent the minimal decision-making entity, in which the role of each program is to define a unique context for deploying its action given the current state of the environment. Finally, each GP program has to explicitly identify which *subset* of state variables to operate on. Each Program will typically index a different subset of the state variables, leading to teams emerging that make decisions based on specific sub-regions of the state space.

Unlike the previous hierarchical version of SBB [10, 8, 9], TPG maintains only one population of programs and one population of teams. Team development is driven by a breeder model of evolution such that a fixed fraction of the least desirable teams (*PopGap*) are deleted at each generation and replaced by the offspring of surviving teams.[4] Team offspring are created by cloning the team along with all its programs, and then applying mutation-based variation operators to the cloned team and programs, as parameterized in Table 1. Thus, evolution is driven by 'group-level' selection in which the team is judged as a whole rather than by the performance of individual components. As such, programs have no individual fitness. However, at each generation, orphaned programs

this work.

[3] ALE provides SECAM as an alternative encoding to the default NSTC format.

[4] Individuals in the team population merely index a subset of programs from the program population under a variable length representation. A valid team conforms to the constraint that it must index a minimum of 2 programs and have at least two different actions.




(a) Evolved policy

(b) Initial Policy

Figure 2: TPG Policies. Decision making in each time step (frame) begins at the root team and follows the edge with the winning program bid (output) until an atomic action (Atari Joystick Position) is reached. The initial population contains only single-team policies (b). Multi-team graphs emerge as evolution progresses (a). The policies pictured are from a game that does not use the fire button. Thus, only directional joystick positions and 'No Action' appear.

**Algorithm 1** Example program in which execution is sequential. Programs may include two-argument instructions of the form $R[i] \leftarrow R[x] \circ R[y]$ in which $\circ \in +, -, x, \div$; single-argument instructions of the form $R[i] \leftarrow \circ(R[y])$ in which $\circ \in cos, ln, exp$; and a conditional statement of the the form IF $(R[i] < R[x])$ THEN $R[i] \leftarrow -R[i]$. $R[i]$ is a reference to an internal register, while $R[x]$ may reference internal registers or state variables (sensor inputs). Determining *which* of the available sensor inputs are actually used in the program, as well as the number of instructions and their operations, are both emergent properties of the evolutionary process.

1. $R[0] \leftarrow R[0] - R[3]$
2. $R[1] \leftarrow R[0] \div R[7]$
3. $R[1] \leftarrow Log(R[1])$
4. IF $(R[0] < R[1])$ THEN $R[0] \leftarrow -R[0]$
5. RETURN $R[0]$

– those that are no longer a member of any team – are deleted, i.e. they were only associated with the worst performing teams.

## 4.2 Emergent Modularity

All programs are initialized with exclusively atomic actions, thus only single-team policies exist in the initial population, Figure 2(b). As such, all initial teams represent 'graph' root nodes. In order to support code reuse and emergent modularity, search operators will occasionally mutate a program's action. The modified action has an equal probability of referencing either an atomic action or another team. Thus, search operators have the ability to *incrementally* construct multi-team graphs, Figure 2(a). Naturally, decision making in multi-team graphs begins at the root node (team) and follows *one* path through the network until an atomic action is selected. Cycles may exist in the graph, but they are never followed during execution. That is, a team is never visited twice per decision. The single visit constraint is enforced by testing whether the action of the program with highest output (winning bid established during team evaluation, Section 4.1) corresponds to a previously visited team. If so, the next highest bid is selected, and the validation step repeats. Each team maintains at least one program with an atomic action, hence guaranteeing cycles never appear.

The number of unique policies in the population at any given generation is equal to the number of root teams, i.e. teams that are not referenced as any program's action. This number fluctuates, as root teams are sometimes 'subsumed' by another graph. For example, variation operators may mutate a program's action to point to the root team of an existing policy graph, in which case there would be one less policy in the population. Conversely, a graph may be separated through the reverse process, resulting in a new root team / policy. Only root teams are subject to modification by the variation operators. Internal nodes (teams and individual programs) are essentially cached blocks of code, which may appear in more than one policy graph and at more than one location in the same graph. However, a team is never simultaneously a root *and* an internal node in another policy graph. In short, graphs of multiple teams emerge through a continuous process of development. Most importantly, as programs composing a team typically index different subsets of the state space (i.e., the screen), the resulting policy graph will incrementally index more of the state space *and* prioritize the types of decisions made in different regions.

## 4.3 Diversity Maintenance

Searching for *good* behaviours as well as *different kinds* of behaviours has two key benefits: 1) diversity helps prevent premature (team) convergence; and 2) when developing a library of reusable code, a diverse population represents a versatile toolbox for subsequent reuse [10]. In this work, diversity is maintained by ensuring that each program's bidding behaviour is unique w.r.t the rest of the program population. To achieve this, a global archive of the most recent 50 state observations is maintained at all times, where each observation is simply a vector of integers representing a single quantized game frame (See Section 3.1), as experienced by some member of the team population. When a new program is created or an existing program is modified, its *profile*[5] of bids over the archive is required to be unique relative to the rest of the program population [13]. Such a definition is task-independent.

[5] A vector of 50 double-precision values, or the program's output when executed relative to each unique state stored in the archive.

Table 1: Parameterization of Team and Program populations. 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; $x \in \{m, n\}$ are the prob. of creating a new program or changing the program action respectively. $\omega$ is the max. initial team size. For the program population, $p_x$ denotes a mutation operator in which $x \in \{delete, add, mutate, swap\}$ are the probabilities for deleting, adding, mutating, or reordering instructions within a program.

| Team population | | | |
|---|---|---|---|
| Parameter | Value | Parameter | Value |
| $PopSize$ | 360 | $PopGap$ | 50% of *Root* Teams |
| $p_{md}, p_{ma}$ | 0.7 | $\omega$ | 5 |
| $p_{mm}$ | 0.2 | $p_{mn}$ | 0.1 |
| Program population | | | |
| $numRegisters$ | 8 | $maxProgSize$ | 96 |
| $p_{delete}, p_{add}$ | 0.5 | $p_{mutate}, p_{swap}$ | 1.0 |

# 5 Empirical Experiments

The goal of this research is to establish the baseline capability of TPG over a diverse selection of Atari 2600 video games [1]. These games are particularly interesting because they are known to be challenging for both humans and learning algorithms. For this initial study, we concentrate on the 20 games in which Deep Reinforcement Learning, the algorithm with the most high scores to date, failed to reach human-level play [15].

## 5.1 Experimental Setup

We conducted 5 independent runs of TPG in each game, for as many generations as possible given our resource time constraint[6]. The same parameterization for TPG was used for all games, Table 1. The only information provided to the agents is the number of atomic actions available for each game, the raw pixel screen matrix at each frame (time step) during play, and the final game score. Each episode continues until the simulator returns a 'game over' signal or a maximum of 18,000 frames is reached. Policy graphs are evaluated in 5 episodes per generation, up to a maximum of 10 episodes per lifetime. This allows weak policies to be identified and replaced after only 5 evaluations in one generation, while promising policies are verified with an additional 5 evaluations. Team fitness is simply the mean game score over all evaluations for that team. The single champion from each run was identified as the individual with the highest training reward, or mean score over at least 5 evaluations. As per established test conditions [15], champions are evaluated in 30 test games for a maximum of 5 minutes of play (max. 18,000 frames) in each game.

## 5.2 Results

Test results for TPG, along with game scores for a human professional video game tester (from [15]) and two comparison algorithms, are reported in Table 2. To the best of our knowledge, Deep Reinforcement Learning (DQN) [15] and HyperNeat (NEAT) [5] are the only two algorithms, working from the 'pixel' state representation, that previously held the highest (machine-learning) score in at least one of the games considered. It is apparent that TPG is competitive with both methods, achieving a new highest score in 14 of the 20 games. In 7 of these, TPG also outperforms the human professional video game tester.

## 5.3 Solution Analysis

### 5.3.1 Model Complexity

An evolved TPG policy is essentially a directed graph in which vertices are teams and edges are programs (see Figure 2(a)). The run-time efficiency of TPG policies is a factor of how many instructions are executed in order to make a decision in any single time step. Recall from Section 4 that each decision requires following a *single* path from the root team to atomic action. The column 'Ins' in Table 2 reports the average number of instructions executed along this path in each time step over the 30 test games.

[6] All experiments were conducted on a shared cluster with a maximum run-time of 2 weeks. The nature of some games allowed for $> 1000$ generations, while others limited evolution to the order of a few hundred.

Two observations regarding the run-time complexity of evolved policy graphs appear: 1) There is significant diversity across different game titles, thus policy complexity scales based on the requirements of each environment. 2) The overall complexity level, requiring between 116 (Double Dunk) and 1036 (Ms. Pac-Man) instructions on average, is significantly less than both comparison algorithms. For example, both DQN and Hyper-NEAT employ neural network architectures consisting of $> 800,000$ weights, all of which are computed for *every* decision (See Appendix A in [5] and 'Model Architecture' under Methods in [15]). The simplicity of TPG's solutions is made possible by the modular nature of the architecture, which allows for adaptive, environment-driven complexification.

Table 2: Results for TPG along with the top learning algorithms in the ALE literature which use a raw pixel state representation. Also reported for TPG is the number of teams in each champion policy (Tms), the average number of instructions required for the policy to make a decision in each time step (Ins), and the proportion of the input space covered by the policy (%IP). Scores in bold indicate the highest score for a learning algorithm for that game, while a gray cell indicates the score was also better than the human professional video game tester (Hum).

| Game | DQN | HNEAT | Hum | TPG | Tms | Ins | %IP |
|---|---|---|---|---|---|---|---|
| Alien | 3069(±1093) | 1586 | 6875 | **3382.7**(±1364) | 46 | 455 | 56 |
| Amidar | **739.5**(±3024) | 184.4 | 1676 | 398.4(±91) | 63 | 812 | 69 |
| Asterix | **6012**(±1744) | 2340 | 8503 | 2400(±505) | 42 | 414 | 51 |
| Asteroids | 1629(±542) | 1694 | 13157 | **3050.7**(±947) | 13 | 346 | 23 |
| BankHeist | 429.7(±650) | 214 | 734.4 | **1051**(±56) | 58 | 572 | 65 |
| BattleZone | 26300(±7725) | 36200 | 37800 | **47233.4**(±11924) | 4 | 123 | 11 |
| Bowling | 42.4(±88) | 135.8 | 154.8 | **223.7**(±1) | 56 | 585 | 57 |
| Centipede | 8309(±5237) | 25275.2 | 11963 | **34731.7**(±12333) | 28 | 516 | 39 |
| C.Command | 6687(±2916) | 3960 | 9882 | **7010**(±2861) | 51 | 280 | 58 |
| DoubleDunk | -18.1(±2.6) | 2 | -15.5 | 2(±0) | 4 | 116 | 6 |
| Frostbite | 328.3(±250.5) | 2260 | 4335 | **8144.4**(±1213) | 21 | 382 | 28 |
| Gravitar | 306.7(±223.9) | 370 | 2672 | **786.7**(±503) | 13 | 496 | 36 |
| M'sRevenge | 0 | 0 | 4367 | 0(±0) | 18 | 55 | 28 |
| Ms.Pac-Man | 2311(±525) | 3408 | 15693 | **5156**(±1089) | 111 | 1036 | 83 |
| PrivateEye | 1788(±5473) | 10747.4 | 69571 | **15028.3**(±24) | 59 | 938 | 60 |
| RiverRaid | **8316**(±1049) | 2616 | 13513 | 3884.7(±566) | 67 | 660 | 64 |
| Seaquest | **5286**(±1310) | 716 | 20182 | 1368(±443) | 22 | 392 | 37 |
| Venture | 380(±238.6) | NA | 1188 | **576.7**(±192) | 3 | 165 | 7 |
| WizardOfWor | 3393(±2019) | 3360 | 4757 | **5196.7**(±2550) | 17 | 247 | 31 |
| Zaxxon | 4977(±1235) | 3000 | 9173 | **6233.4**(±1018) | 20 | 424 | 33 |

#### 5.3.2 Emergent Modularity

All policies in TPG are initialized as a single team of programs (See Figure 2(b)), which represents the simplest possible decision-making entity in TPG. The rate of growth into more complex multi-team graph structures is driven by interaction with the task environment, i.e. the development of complex policies is only possible if simpler policies are out performed.

Figure 3 depicts the development of modularity for TPG polices throughout evolution in 4 games where TPG ultimately achieved the best score of any learning algorithm. Clearly, different game environments result in different levels of complexity in the champion policies. Ms. Pac-Man is known to be a challenging game [18, 20] and, perhaps not surprisingly, benefits form relatively complex structures. On the other hand, TPG managed to reach a high level of play in Asteroids with simple policies containing 7 teams. The trajectory denoted by 'Random' in Figure 3 refers to a run in which policies were assigned random fitness values. The lack of development confirms that complex policies emerge by selective pressure rather than drift or other potential biases.

#### 5.3.3 Adapted Visual Field

Each Atari game presents a unique graphical environment in which important game events occur in different areas of the screen and at different resolutions. Part of the challenge with high-dimensional visual input data is determining what information is relevant to the task. TPG begins with single teams, thus minimal screen coverage,

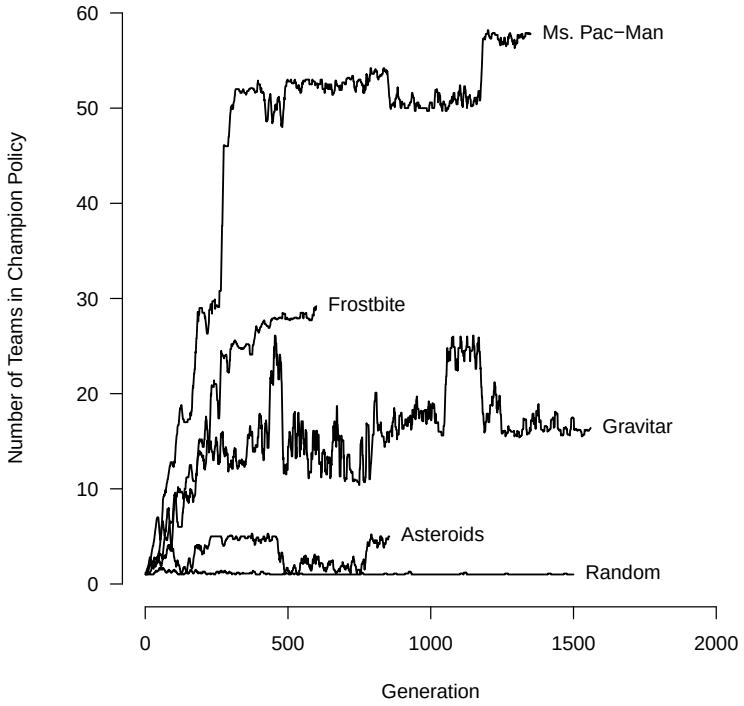


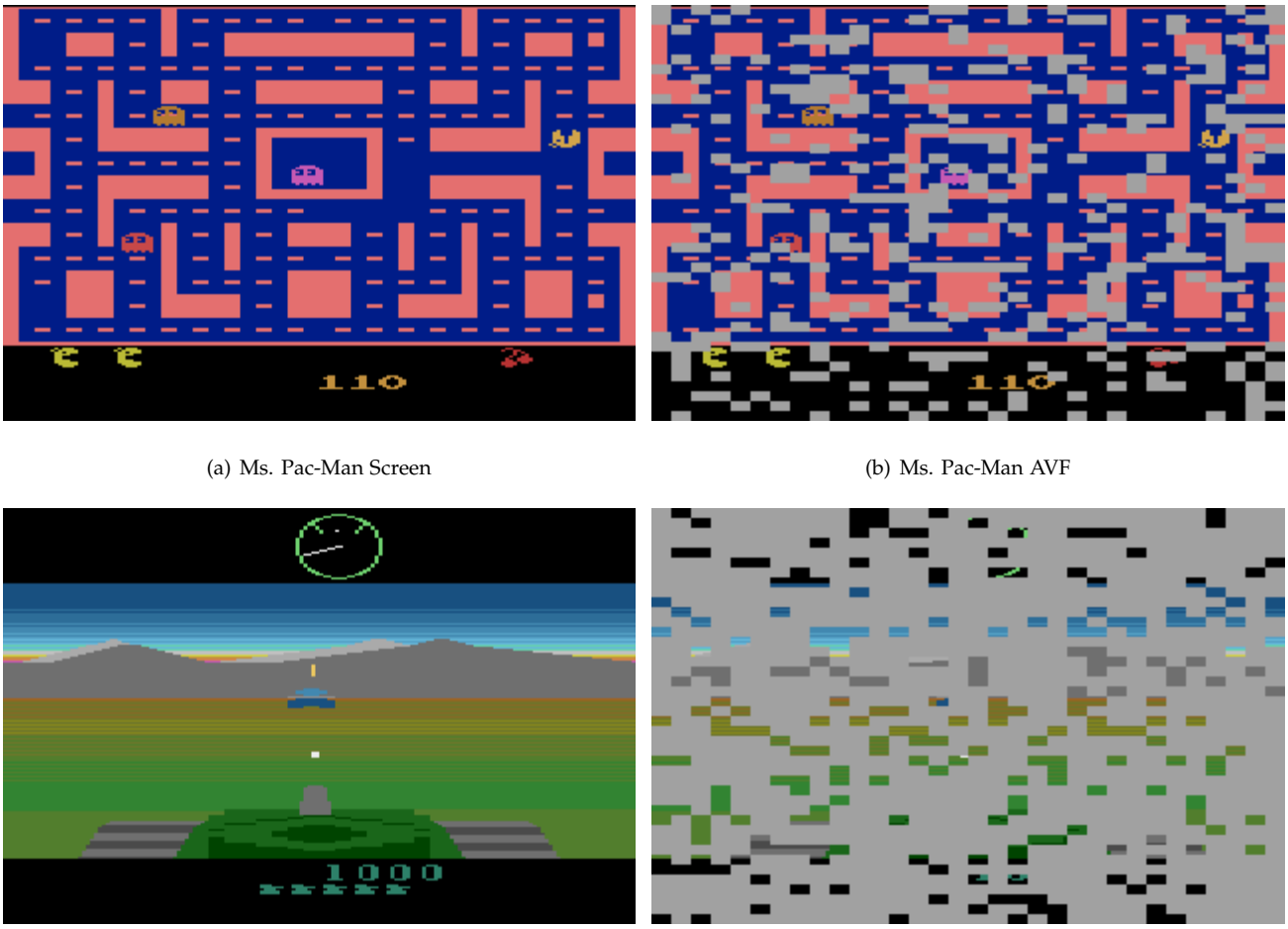

Figure 3: Number of teams per champion TPG policy throughout evolution in 4 games where TPG ultimately achieved the best score of any learning algorithm. 'Random' is the control experiment, referring to a run in which policies were assigned random fitness values. Each line depicts the median over 5 runs. The diverse nature of each game implies that the cost of evaluating policies varies. Thus, given the same time constraint, a different number of generations are possible in each game environment.

and incrementally explores more of the screen through complexification. By doing so, the utility of policies with complex screen coverage is continually checked against simpler alternatives, only persisting when they prove useful. Indeed, in 8 of the 14 games for which TPG achieved the highest score for a learning algorithm, it did so while indexing less that 50% of the screen, minimizing the number of instructions required per decision. In contrast, neural network representations as applied to the ALE task define sub-fields that are fully connected to the input space (e.g. through a convolution applied to all locations of the state space). In short, the architecture is predefined as opposed to developed through interaction with the environment.

For example, Figure 4 shows the Adapted Visual Field (AVF) of champion TPG agents in Ms. Pac-Man and Battle Zone. In the case of Ms. Pac-Man, the game defines a 2-dimensional maze environment that the player navigates in order to collect pills, where the pills are evenly distributed throughout the maze. Relatively high resolution is required in order to distinguish objects such as the agent's avatar and pills from the maze walls, and near-complete screen coverage is required to locate all the active pills and guide the avatar to/from any maze location. On the other hand, Battle Zone is a first person shooter game in which the agent can swivel left or right to position targets at centre-screen before shooting. While even screen coverage is helpful in locating targets and determining the direction to swivel, targets are large and thus low-resolution coverage is sufficient. Interestingly, Battle Zone includes a high-resolution global radar view in the top centre of the screen, which the champion policy's low-resolution AVF did not make efficient use of. Nonetheless, the bare-bones policy was able to out-perform the human video game tester without this advantage.

## 6 Conclusion and Future Work

A Tangled Program Graph (TPG) representation is proposed for discovering deep combinations of programs that collectively define policies in high-dimensional reinforcement learning tasks. Benchmarking is conducted under the subset of 20 games from the Atari 2600 ALE challenge for which the Deep Reinforcement Learning (DQN) framework did not reach the threshold for human-level play. In terms of score ranking for learning algorithms, TPG returns the best score in 14 games, DQN in 4 games, with HyperNeat and TPG tied in 1 game. Moreover, in 7 of the 15 games for which TPG provides a better strategy than DQN, TPG is also better than the threshold for




(a) Ms. Pac-Man Screen

(b) Ms. Pac-Man AVF

(c) Battle Zone Screen

(d) Battle Zone AVF

Figure 4: Adapted Visual Field (AVF) of champion TPG policies in Ms.Pac-Man and Battle Zone, two games in which TPG achieved the highest score of any learning algorithm. The champion Battle Zone policy also scored higher than the human professional video game tester. Grey regions indicate areas of the screen not indexed by the policy.

human-level play.

Key to TPG is support for emergent modularity. That is to say, the ability to identify decisions local to different sub-regions of the state–action space and then organize such decisions hierarchically. This makes for a very efficient decision making process that completely decouples the overall complexity of a candidate solution (total number of programs) from the number of programs actually executed to make a decision (or size of the graph versus the proportion of the graph traversed to make a decision).

Such an approach is much more efficient than the representations assumed to date for deep learning, in which specialized (GPU) hardware support is necessary. Specifically, DQN assumes a fixed neural topology from the outset (i.e., hidden layer contains $> 800,000$ weights and this cost is independent of game title) and a specific association with the state space (a computationally costly deep learning correlation step). Moreover, DQN assumes a correlation filter to discover encoded representations of game state. This introduces millions of calculations per frame whereas TPG merely subsamples the original frame information. The capacity of TPG to make decisions efficiently while not compromising on the quality of the resulting policies might also open up additional application areas to GP (e.g. real-time interpretation of video for obstacle avoidance in autonomous cars).

Having established the baseline capability of TPG, future work will investigate the utility of explicit diversity maintenance through multi-objective fitness regularization. In particular, switching between *multiple* diversity measures (in combination with fitness-based selection) has been shown to impact the development of modularity [7, 17]. Future work is also likely to expand the set of test games, further validating the generality of the approach.

**Acknowledgments.** S. Kelly gratefully acknowledges support from the Nova Scotia Graduate Scholarship program. M. Heywood gratefully acknowledges support from the NSERC Discovery program. All runs were completed on cloud computing infrastructure provided by ACENET, the regional computing consortium for universities in Atlantic Canada. The TPG code base is not in any way parallel, but in adopting ACENET the five independent runs for each of the 20 games were conducted in parallel.

## References

[1] Marc G. Bellemare, Yavar Naddaf, Joel Veness, and Michael Bowling. The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 2012.

[2] M. Brameier and W. Banzhaf. *Linear Genetic Programming*. Springer, 1st edition, 2007.

[3] Markus Brameier and Wolfgang Banzhaf. Evolving teams of predictors with linear genetic programming. *Genetic Programming and Evolvable Machines*, 2(4):381–407, December 2001.

[4] J. A. Doucette, P. Lichodzijewski, and M. I. Heywood. Hierarchical task decomposition through symbiosis in reinforcement learning. In *Proceedings of the ACM Genetic and Evolutionary Computation Conference*, pages 97–104, 2012.

[5] Matthew Hausknecht, Joel Lehman, Risto Miikkulainen, and Peter Stone. A neuroevolution approach to general Atari game playing. *IEEE Transactions on Computational Intelligence and AI in Games*, 6(4):355–366, 2014.

[6] K. Imamura, T. Soule, R. B. Heckendorn, and J. A. Foster. Behavioural diversity and probabilistically optimal GP ensemble. *Genetic Programming and Evolvable Machines*, 4(3):235–254, 2003.

[7] N. Kashtan, E. Noor, and U. Alon. Varying environments can speed up evolution. *Proceedings of the National Academy of Sciences*, 104(34):13711–13716, 2007.

[8] S. Kelly and M. I. Heywood. Genotypic versus behavioural diversity for teams of programs under the 4-v-3 keepaway soccer task. In *Proceedings of the AAAI Conference on Artificial Intelligence*, pages 3110–3111, 2014.

[9] S. Kelly and M. I. Heywood. On diversity, teaming, and hierarchical policies: Observations from the keepaway soccer task. In *European Conference on Genetic Programming*, volume 8599 of *LNCS*, pages 75–86. Springer, 2014.

[10] S. Kelly, P. Lichodzijewski, and M. I. Heywood. On run time libraries and hierarchical symbiosis. In *IEEE Congress on Evolutionary Computation*, pages 3245–3252, 2012.

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

[12] P. Lichodzijewski and M. I. Heywood. Managing team-based problem solving with symbiotic bid-based genetic programming. In *Proceedings of the ACM Genetic and Evolutionary Computation Conference*, pages 863–870, 2008.

[13] P. Lichodzijewski and M. I. Heywood. Symbiosis, complexification and simplicity under GP. In *Proceedings of the ACM Genetic and Evolutionary Computation Conference*, pages 853–860, 2010.

[14] P. Lichodzijewski and M. I. Heywood. The Rubik cube and GP temporal sequence learning: an initial study. In *Genetic Programming Theory and Practice VIII*, chapter 3, pages 35–54. Springer, 2011.

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

[16] Stefano Nolfi. Using emergent modularity to develop control systems for mobile robots. *Adaptive behavior*, 5(3-4):343–363, 1997.

[17] M. Parter, N. Kashtan, and U. Alon. Facilitated variation: How evolution learns from past environments to generalize to new environments. *PLoS Computational Biology*, 4(11):e1000206, 2008.

[18] T. Pepels and M. H. M. Winands. Enhancements for monte-carlo tree search in ms pac-man. In *IEEE Symposium on Computational Intelligence in Games*, pages 265–272, 2012.

[19] Justinian Rosca. Towards automatic discovery of building blocks in genetic programming. In *Working Notes for the AAAI Symposium on Genetic Programming*, pages 78–85. AAAI, 10–12 1995.

[20] Jacob Schrum and Risto Miikkulainen. Discovering multimodal behavior in ms. pac-man through evolution of modular neural networks. *IEEE Transactions on Computational Intelligence and AI in Games*, 8(1):67–81, 2016.

[21] Lee Spector, Brian Martin, Kyle Harrington, and Thomas Helmuth. Tag-based modules in genetic programming. In *Proceedings of the 13th Annual Conference on Genetic and Evolutionary Computation*, pages 1419–1426. ACM, 2011.

[22] S. Steenkiste, J. Koutník, K. Driessens, and J. Schmidhuber. A wavelet-based encoding for neuroevolution. In *Proceedings of the ACM Genetic and Evolutionary Computation Conference*, pages 517–524, 2016.

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

[24] Shelly Xiaonan Wu and Wolfgang Banzhaf. Rethinking multilevel selection in genetic programming. In *Proceedings of the ACM Genetic and Evolutionary Computation Conference*, pages 1403–1410, 2011.