Multi-Task Learning in Atari Video Games with Emergent Tangled Program Graphs

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

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

Article originally appears at GECCO'17 under ACM copyright 2017
http://dl.acm.org/citation.cfm?doid=3071178.3071303

Abstract

The Atari 2600 video game console provides an environment for investigating the ability to build artificial agent behaviours for a variety of games using a common interface. Such a task has received attention for addressing issues such as: 1) operation directly from a high-dimensional game screen; and 2) partial observability of state. However, a general theme has been to assume a common machine learning algorithm, but completely retrain the model for each game title. Success in this respect implies that agent behaviours can be identified without hand crafting game specific attributes/actions. This work advances current state-of-the-art by evolving solutions to play multiple titles from the same run. We demonstrate that in evolving solutions to multiple game titles, agent behaviours for an individual game as well as single agents capable of playing *all* games emerge from the same evolutionary run. Moreover, the computational cost is no more than that used for building solutions for a single title. Finally, while generally matching the skill level of controllers from neuro-evolution/deep learning, the genetic programming solutions evolved here are several orders of magnitude simpler, resulting in real-time operation at a fraction of the cost.

1 Introduction

Machine learning (ML) for defining artificial agent behaviours, or *policies*, represents one of the longest standing themes regarding the application of AI to games. As such, Yannakakis and Togelius identify six general contributions [22], including the following three of particular relevance to this work: 1) *Games as AI benchmarks* recognizes that games increasingly represent an environment characterized by high dimensional spatial-temporal information. Moreover, the environments are often non-stationary and subject to partial observability. Thus, games represent a particularly rich combination of challenges for ML in general; 2) *Believable agent behaviours* implies that agent decision making should be plausible. It is increasingly unacceptable to rely on cheats or global information. Moreover, agents need to be 'proficient' in order to be believable. The more proficient/believable a behaviour, the more immersive a gaming experience potentially is; 3) *General game AI* recognizes the role of games in developing artificial general intelligence, i.e. algorithms capable of demonstrating human levels of intelligent behaviour. Indeed, a game based formulation of the Turing test has been proposed in which the play of artificial agents are compared to that of humans [7].

In particular, agent behaviours are increasingly being developed that are able to interact directly from game content as experienced by a human player, as opposed to requiring hand crafted features to be defined a priori. This potentially brings artificial behaviours closer to that of a human player, or less dependent on tricks-of-the-trade. With the above points in mind, a growing body of research has been employing a suite of Atari 2600 video game titles in which to demonstrate domain-independent reinforcement learning (e.g. [15, 1, 5, 21, 16]). In each case, the focus has been on developing game-specific agents.

In this work, we investigate the capacity to support Multi-Task Reinforcement Learning (MTRL) in the Atari environment. Thus, an agent is developed for multiple game titles simultaneously, with no prior knowledge regarding how individual games may relate to one another, if at all. The Atari 2600 environment represents an interesting test domain because each game is unique and designed to be challenging for human players. Furthermore, artificial agents experience game-play just as a human would, via the a high-dimensional game screen and Atari joystick. We propose a Genetic Programming (GP) framework to address high-dimensional MTRL

through emergent modularity [17]. Specifically, a bottom-up process is assumed in which multiple programs self-organize into collective decision-making entities, or teams, which then further develop into multi-team *policy graphs*, or Tangled Program Graphs (TPG). The TPG framework learns to play three Atari games simultaneously, producing a single control policy that matches or exceeds leading results from (game-specific) deep reinforcement learning (DQN) [15] in each game. Furthermore, unlike the representation assumed for deep learning, TPG policies start simple and adaptively complexify through interaction with task environment, resulting in agents that are exceedingly simple, operating in real-time without specialized hardware support such as GPUs.

2 Background

With regard to constructing agent behaviours for video games through machine learning, the concept of generalization has been studied primarily from three related but unique perspectives:

Domain-Independent AI is concerned with developing learning algorithms that can be applied to a variety of task domains with minimal prior knowledge and no task-dependent parameter tuning. To date, this has been the focus of studies in the Atari 2600 environment, where several approaches have demonstrated an ability to build policies for multiple games using the same learning framework [15, 1, 5, 21, 16]. In these works, a game-specific policy is developed from scratch for each game title. However, task transfer has also been employed in the Atari task [3], in which case the agent reuses experience gained in one or more *source* game titles to improve learning under a single *target* title. The Term 'General Video Game Playing' is often used when describing Domain-Independent AI. However, there is an important distinction between these terms which we acknowledge below.

General Video Game Playing is concerned with building agents that are capable of playing multiple games which they have *never seen before* [19, 12]. That is, no off-line experience / training with a specific game environment takes place prior to evaluation on that game. Thus, while learning is possible during gameplay, a prior level of general game-playing competence is required.

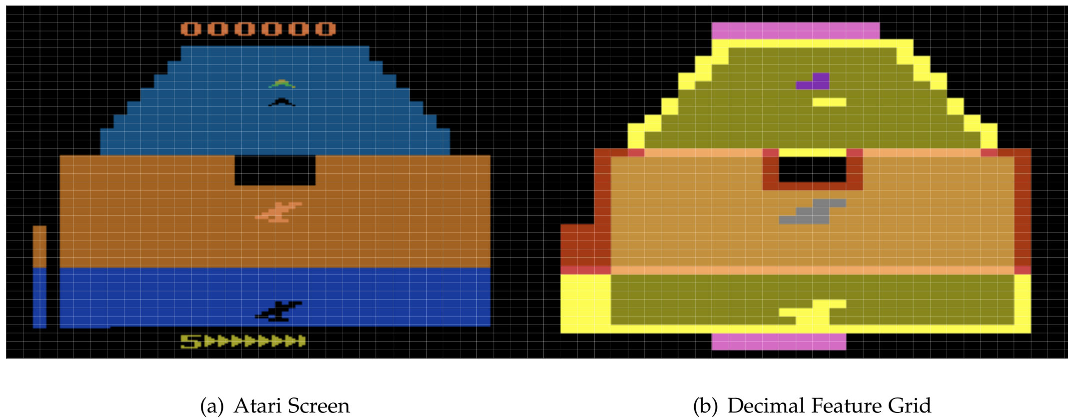
Multi-Task Learning is concerned with discovering agent behaviours for multiple tasks simultaneously [20, 18], resulting in multiple game-specific policies and/or a single policy that is capable of playing multiple game titles at a high skill level. Multi-task learning is the primary focus of this work.

The approach taken in this work, Tangled Program Graphs (TPG), is an extension of the well-established Symbiotic Bid-Based (SBB) algorithm [13] for evolving teams of programs. SBB has been shown to build strong policies for a variety of reinforcement learning tasks [4, 11, 14, 8, 9], owing primarily to two key innovations: 1) Solutions are represented by a team of programs, in which each program defines a context for deploying a single action, and a bidding process determines the particular action to associate with each state observation [13]. The number and complement of programs per team, as well as the bidding behaviour of each program, are evolved properties; 2) Hierarchical decision making agents may be constructed over two independent phases of evolution, where the first phase produces a library of diverse, specialist teams and the second phase attempts to build more general policies by reusing the library. The result was a *policy tree*.

The work described here takes a more open-ended approach to organizing teams of programs into graphs as opposed to trees; hereafter Tangled Program Graphs (TPG). Specifically, each node of a policy tree is defined by a team, and each 'path' through the tree (of teams) is of equal depth. Thus, a team at the root describes a policy through a path defined in terms of teams at *all* prior levels of the tree. Unfortunately, as the tree depth increases it becomes increasingly difficult to disambiguate the relation between state and action, as new teams have to operate 'through' all previous policy levels. If teams could instead be inter-related through a graph structure, the potential to retain clarity between state and action would be retained. This is the underlying purpose of TPG.

3 Arcade learning Environment

Released in 1977, the Atari 2600 was a popular home video game console that supported hundreds of game titles. The Arcade Learning Environment (ALE) provides an Atari 2600 emulator with an interface geared towards benchmarking domain-independent AI algorithms [1]. Specifically, a game playing agent can interact with games by observing the game screen (210 x 160 pixel matrix with 128 possible colours for each pixel), and responding with 18 discrete actions defined by the Atari joystick (all possible combinations of paddle direction and fire button state, including 'no action'). Due to hardware limitations in the original console, game entities often appear intermittently over sequential frames, causing visible flicker. As such, Atari game environments are considered partially observable, since it is often impossible to observe the complete state of play from a single screen frame. In common with previous work in the ALE, agents in this study stochastically skip screen frames with probability



(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).

$p = 0.25$, with the previous action being repeated on skipped frames [1, 15]. This is a default setting in ALE, and aims to: 1) limit artificial agents to roughly the same reaction time as a human player; and 2) introduce stochasticity into the environment [6]. A single episode of play lasts a maximum of $18,000$ frames, not including skipped frames.

3.1 Image Preprocessing

Atari screen frames have a lot of redundant information. That is, visual game content is designed for maximizing entertainment, as opposed to simply conveying state, $\vec{s}(t)$, information. As such, we adopt a quantization approach to preprocessing. The following 2-step procedure is applied to every game frame:

Frame grid: The frame is subdivided into a 42×32 grid, in which only 50% of the pixels in each tile are considered and each pixel assumes an 8-colour SECAM encoding. The SECAM 8-colour encoding is provided by ALE as an alternative to the default NSTC 128-colour format. **Encoding:** Each tile is described by a single byte, in which each bit encodes the presence of one of eight SECAM colours within that tile. The final sensory state space is constructed from the decimal value of each tile byte, or $\vec{s}(t)$ of $42 \times 32 = 1,344$ decimal features in the range of $0 - 255$, visualized in Figure 1(b) for the game Zaxxon 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. That is, no feature extraction is performed as part of the preprocessing step, just a quantization of the original frame data. Each TPG program then learns to sample a potentially unique subset of inputs from $\vec{s}(t)$ for incorporating into their decision making process. The emergent properties of TPG are then required to develop the complexity of a solution, or policy graph, with programs organized into teams and teams into graphs. Thus, the specific subset of sensor inputs sampled within each TPG policy is an emergent property, discovered through interaction with the task environment alone. The implications of this *adapted* sensor efficiency on computational cost for TPG will be revisited in Section 5.3.

4 Evolving Tangled Program Graphs

4.1 Teams of Programs

A single team of programs represents the simplest stand-alone decision-making entity in the TPG framework. A linear program representation is assumed (See Algorithm 1), where such a representation facilitates skipping 'intron' code, which can potentially represent $60 - 70\%$ of program instructions [2]. Each program defines the context for one discrete action (e.g. Atari joystick position), where actions are assigned to the program at initialization and potentially modified by variation operators during evolution. In order to map a state observation to an action, each program in the team will execute relative to the current state, $\vec{s}(t)$, and return a single real valued 'bid', i.e. the content of register $R[0]$ after execution. The team then chooses the action of the program with the highest bid. If programs were not organized into teams, in which case all programs within the same population would compete for the right to suggest their action, it is very likely that degenerate individuals (programs that bid high for every state), would disrupt otherwise effective bidding strategies.

Algorithm 1 Example program in which execution is sequential. Programs may include two-argument instructions of the form $R[x] \leftarrow R[x] \circ R[y]$ in which $\circ \in +, -, x, \div$; single-argument instructions of the form $R[x] \leftarrow \circ(R[y])$ in which $\circ \in cos, ln, exp$; and a conditional statement of the the form IF $(R[x] < R[y])$ THEN $R[x] \leftarrow -R[x]$. $R[x]$ is a reference to an internal register, while $R[y]$ 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.

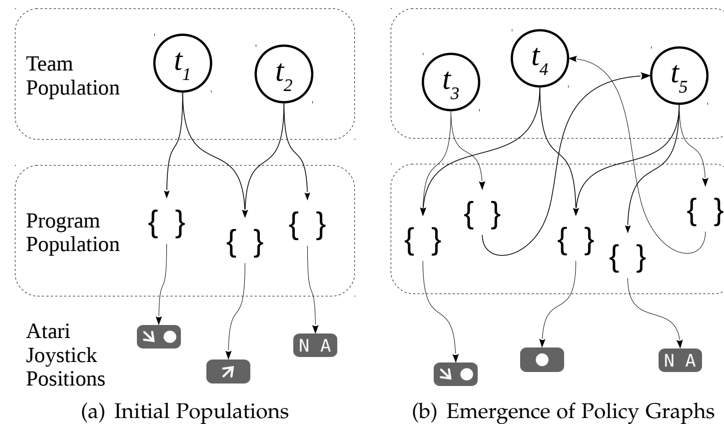
1. $R[0] \leftarrow R[0] - R[3]$
2. $R[1] \leftarrow R[0] \div R[7]$
3. $R[1] \leftarrow Log(R[1])$
4. IF $(R[0] < R[1])$ THEN $R[0] \leftarrow -R[0]$
5. RETURN $R[0]$

Adaptively building teams of programs is addressed here through the use of a symbiotic relation between a team population and a program population, or Symbiotic Bid-Based GP (SBB) [13]. Each individual of the team population represents an index to some subset of the program population (See Figure 2(a)). Team individuals therefore assume a variable length representation in which each individual is stochastically initialized with $[2, ..., \omega]$ pointers to programs from the program population. Furthermore, the team initialization process ensures there are at least two different actions indexed by the complement of programs within the same team. The same program may appear in multiple teams, but must appear in at least one team to survive between consecutive generations.

Performance (i.e. fitness) is only expressed at the level of teams, and takes the form of the task dependent objective (e.g. Atari game score). After evaluating the performance of all teams, the worst $PopGap$ teams are deleted from the team population. Moreover, any program that fails to be indexed by any team (following team deletion) must have been associated with the worst performing teams, hence is also deleted. This avoids the need to make arbitrary decisions regarding the definition of fitness at the program level. New teams are introduced by sampling, cloning, and modifying $PopGap$ surviving teams. Naturally, if there is a performance benefit in smaller/larger teams and/or different program complements, this will be reflected in the surviving team–program complements [13], i.e. team–program complexity is a developmental trait.

4.2 Policy Graphs

Programs are initialized with atomic actions defined by the task environment (i.e. Joystick positions, Figure 2(a)). In order to enable the evolution of hierarchically organized code under a completely open ended process of evolution (i.e. emergent modularity [17]), program variation operators are allowed to introduce actions that index other teams within the team population. When a program's action is modified, it has an equal probability



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

Figure 2: Illustration of the relation between team and program populations at initialization (a) and during evolution as arbitrarily deep/wide *tangled program graphs* emerge (b).

of referencing either an atomic action or another team. Thus, variation operators have the ability to *incrementally* construct multi-team *policy graphs* (Figure 2(b)). Each vertex in the graph is a team, while each team member, or program, represents one outgoing edge leading either to another team or an atomic action. Naturally, decision-making in a policy graph begins at the root team (e.g. t_3 in Figure 2(b)), where each program in the team will produce one bid relative to the current state observation. Graph traversal then follows the program / edge with the largest bid, repeating the bidding process for the *same* state, $\vec{s}(t)$, at every team / vertex along the path until an atomic action is reached. Thus, in sequential decision making tasks, the policy graph computes *one* path from root to action at every time step, where only a subset of programs in the graph (i.e those in teams along the path) require execution. Algorithm 2 details the process for evaluating the TPG policy graph, which is repeated at every time step until an end-of-game state is encountered and fitness for the policy graph can be expressed.

Algorithm 2 Selecting an action through traversal of a policy graph. P is the current program population. tm_i is the current team (initially a root node). $\vec{s}$ is the state observation. V is the set of teams visited throughout *this* traversal (initially empty). First, all programs in tm_i are executed relative to the current state $\vec{s}$ (Lines 4,5). The algorithm then considers each program in order of bid (highest to lowest, Line 7). If the program has an atomic action, the action is returned (Line 8). Otherwise, if the program's action points to a team that has not yet been visited, the procedure is called recursively on that team. Thus, while a policy graph may contain cycles, they are not followed during traversal. In order to ensure an atomic is always found, team variation operators are constrained such that each team maintains at least one program that has an atomic action.

```
1:  procedure SelectAction(tm_i, s⃗, V)
2:      A = {x ∈ P : x has atomic action}
3:      V = V ∪ tm_i                                    ▷ add tm_i to visited teams
4:      for all p_i ∈ tm_i do
5:          bid(p_i) = exec(p_i, s⃗)                     ▷ run prog. on s⃗ and save result
6:      tm'_i = sort(tm_i)                              ▷ sort progs by bid, highest to lowest
7:      for all p_i ∈ tm'_i do
8:          if p_i ∈ A then return action(p_i)          ▷ atomic reached
9:          else if action(p_i) ∉ V then
10:             return SelectAction(action(p_i), s⃗, V)  ▷ edge
```

As multi-team policy graphs emerge, an increasingly tangled graph of connectivity develops between the team and program populations. The number of unique solutions, or policy graphs, at any given generation is equal to the number of root nodes (i.e. teams that are not referenced as any program's action, for example t_3 in Figure 2(b)) in the team population. Only these root teams are candidates to have their fitness evaluated, and are subject to modification by the variation operators.

In each generation, *PopGap* of the root teams are deleted and replaced by offspring of the surviving roots. The process for generating team offspring uniformly samples and clones a root team, then applies mutation-based variation operators to the cloned team which remove, add, and mutate some of its programs (Parameterized in Table 1). The team generation process introduces the same number of root nodes as were deleted. However, the total number of root nodes after generating offspring fluctuates, as root teams (along with the lower policy graph) are sometimes 'subsumed' by a new team. Conversely, graphs can be separated, for example through program action mutation, resulting in new root nodes / policies. This implies that after initialization, both team and program population size varies, and a balance is reached between the fraction of the team population that become 'archived' as internal nodes (i.e. a library of reusable code) and the fraction that remain as root nodes[1].

In summary, the critical idea behind TPG is that SBB [13] is extended to allow policy graphs to emerge, defining the inter-relation between teams. As programs composing a team typically index different subsets of the state space (i.e., the screen), the resulting policy graph will incrementally adapt, indexing more or less of the state space *and* defining the types of decisions made in different regions.

[1] The fraction of root/internal nodes could be influenced by tuning the probability of program-action variation operators, i.e. atomic action versus team pointer. Each occurs with equal probability in this study. Thus, the root/internal node ratio is entirely the result of environmental interaction and selective pressure. While it is possible that all root nodes could be subsumed during evolution, it was not empirically observed.

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. ω is the max initial team size. For the program population, p_x denotes a mutation operator in which $x \in \{delete, add, mutate, swap\}$ are the prob. 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	ω	5
p_{mm}	0.2	p_{mn}	0.1
Program population			
Parameter	Value	Parameter	Value
$numRegisters$	8	$maxProgSize$	96
p_{delete}, p_{add}	0.5	p_{mutate}, p_{swap}	1.0

5 Empirical Experiment

The capability of TPG will be investigated under the particularly challenging task of building multi-task agents for Atari 2600 video games. Previous studies have established the ability of TPG to build game-specific controllers for 20 unique games [10]. Conversely, the goal of this work is to produce both game-specific *and* multi-task policies from the *same* evolutionary run. For this initial multi-task experiment, we define two groups of games to be learned simultaneously, each containing 3 games for which (game-specific) TPG policies matched or exceeded test scores from DQN.

Other than previous success with learning each game individually, the two groupings are not based on any intuition regarding multi-task compatibility. Game group A includes Centipede, Frostbite, and Ms. Pac-Man, three games with no obvious commonalities. Centipede is a vertically oriented shooting game, Frostbite is an adventure game in which the player must build an igloo by jumping on blocks of ice that float across the screen horizontally, and Ms. Pac-Man is a maze task in which the player must navigate a series of mazes to collect pellets. Game group B contains three games that may, at first glance, appear similar to one another. Specifically, group B contains Asteroids, Battle Zone, and Zaxxon, all shooting games in which the player gains points by aiming and firing a gun at on-screen targets. However, their similarities are relatively superficial, as each game title defines its own graphical environment, colour scheme, physics, objective(s), and scoring scheme. Furthermore, joystick actions are not necessarily correlated between game titles. For example, in Asteroids the 'down' action causes the spaceship avatar to enter hyperspace, disappearing and reappearing at a random screen location. In Battle Zone, the 'down' action causes the first-person tank avatar to reverse, and foreground targets shrink, appearing to retreat into the distance. In Zaxxon, a third-person plane-flying / shooting game, the 'down' action is interpreted as 'pull-up', causing the plane to move vertically up the screen.

Having identified two *diverse* groups of games titles, the next section outlines the experimental setup designed to test if both game-specific and multi-task policies, capable of playing all games in a given group, can emerge from a single evolutionary run.

5.1 Experimental Setup

Five runs of TPG are conducted for 200 (300) generations in game group A (B). All runs use the same parameterization, Table 1. In order to support the development of multi-task policies, environment switching is introduced such that the population is exposed to different game titles over the course of evolution. Thus, for each consecutive block of 10 generations, one game title is selected with uniform probability to be the *active* title. The same state representation is used for all game titles (Section 3.1)[2]. Each policy graph is evaluated in 5 episodes per generation under the active title, up to a lifetime maximum of 5 evaluations under each game title. Multiple evaluations in each game are required in order to mitigate the effects of stochasticity in the environment. Thus, each policy stores a historical record of up to 15 evaluations (5 in each of the 3 titles). However, a policy's fitness in any given generation is its average score over 5 episodes in the *active title only*. Thus, selective pressure is only explicitly applied relative to a single game title. However, stochastically switching the active title at regular intervals throughout

[2]Note that the state representation is simply a quantized screen capture. It does not include any state variables that explicitly identify high-level game information such as the active game title or the location of the agent's avatar

evolution implies that a policy's long-term survival is dependent on a level of competence in *all* games. Finally, to facilitate the development of multi-task policies, the single best policy graph for each game title (i.e. that with the highest mean score over 5 games) is protected from deletion, a simple form of elitism that ensures the population as a whole never entirely 'forgets' any game.

5.2 Test Results

During training, the team and program populations are saved every 10 generations, or once for each block of 10 generations associated with the current active game title. Post training, the cached populations are reloaded and all policies are tested in 30 episodes under each title as per established test conditions [15, 10]. This process produces a set of test results from the entire population at intervals of 10 generations throughout evolution, from which we can observe the single best scores achieved in each title, as well as the best multi-task policy score, i.e. the single policy able to play all 3 games at the highest level of competence. A multi-objective performance metric is used to identify the best multi-task policy from each test set post policy development.[3]

Pareto non-dominated policies, $\vec{P}$, are identified relative to each game objective (mean test score in each game). A policy p_i is said to dominate another policy p_j if p_i is better than p_j in at least one objective and no worse in all others. From the set of non-dominated policies, $p_k \in \vec{P}$, **choose a single multi-task champion**, p_k^*, as the policy with the largest product over mean test scores from each game title, or $p_k^* = \max_{p_k \in P}(f1(p_k) \times f2(p_k) \times f3(p_k))$, where $f1(p_k)$, $f2(p_k)$, and $f3(p_k)$ are the mean scores for policy p_k in each of the 3 game titles.

Figure 3(a) plots incremental test results over all generations for each game group, where the Left-Hand-Side (LHS) plot reports the best score achieved in each title by any policy, and the Right-Hand-Side (RHS) plot reports the best multi-task policy scores. Scores are normalized relative to DQN's score (from [15]) in the same game (100%) and random play (0%). Normalized score is calculated as 100 $\times$ (TPG score - random play score)/(DQN score - random play score)[4]. The random play score refers to the mean score achieved by an agent that simply selects random actions at each time step (frame) [15].

The Group A experiment produces one game-specific policy for each game that ultimately exceeds the level of DQN, LHS of Figure 3(a). Surprisingly, in the case of Frostbite and Centipede (LHS of Figure 3(a)), TPG began with initial policies (i.e. generation 1, prior to any learning) that exceeded the level of DQN. In Centipede, this initial policy was degenerate, selecting the 'up-right-fire' action in every frame, but nonetheless accumulating a score of 12,890. While completely un-interesting, the strategy managed to exceed the test score of DQN (8,390) [15] *and* the reported test score for a human professional video game tester (11,963) [15]. From this starting point, the single best policy in Centipede improves throughout evolution to become more responsive and interesting; maintaining the strategy of shooting from the far right while gaining the ability to avoid threats through vertical and horizontal movement.

Likewise, the single best policy in Frostbite begins at a high level of play relative to DQN, and significantly improves throughout evolution. Interestingly, the final champion policy in Frostbite defined a relatively long-term strategy, which involved building an igloo by jumping on horizontally-moving ice bergs for an extended period of play ($\approx$ 1000 frames). When the igloo is complete, the agent promptly navigates to the entrance and enters the igloo (a trajectory consuming $\approx$ 200 frames), advancing to the next level. A long-term strategy also emerges for Ms. Pac-Man, in which the agent navigates directly to a power pill and eats it. Ghosts, which are normally threats to Ms. Pac-Man, become temporarily edible after a power pill is eaten. Thus, after eating the pill, the Ms. Pac-Man agent moves throughout the same section of maze, gaining points by eating ghosts.

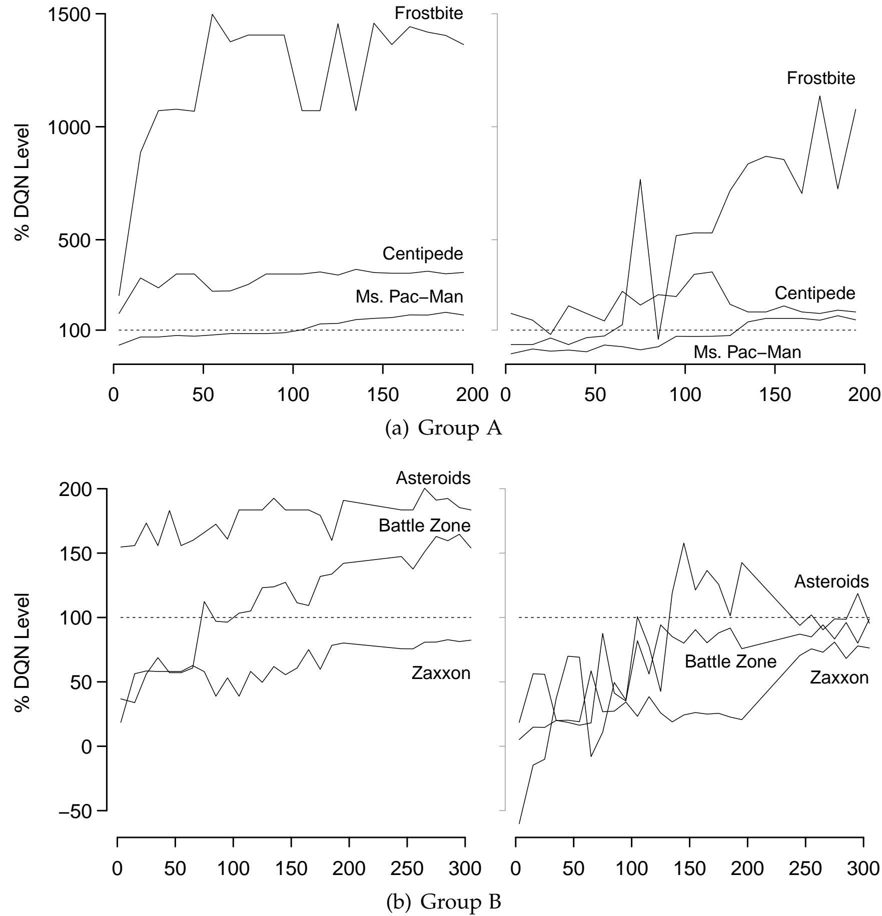
From the group B experiment, the game-specific Asteroids policy begins at a high level of play relative to DQN, while game-specific champions in Battle Zone and Zaxxon slowly emerge over the course of evolution (LHS of Figure 3(b)).

In addition to discovering a single champion policy for each game title, a *single multi-task policy* also emerges from both group A and group B experiments. The RHS of Figure 3 reports game scores for the best multi-task policy from each test set, as identified by the multi-objective performance metric outlined in Section 5.1. Note that the multi-objective metric is only applied post-training to identify the best multi-task policy. While no single policy is initially capable of playing all three games at a level equivalent to DQN, a competent multi-task policy emerges by generation $\approx$ 125 (Group A) and $\approx$ 250 (Group B).

Table 2 reports the final test scores for the best game-specific TPG policy in each game title (TPG-GS) and the champion multi-task policy (TPG-MT). Note that in group A, all test scores reported for TPG exceed the level of

[3]Fitness at each generation is only relative to the current (single) game title.

[4]Normalizing scores makes it possible to plot TPG's progress relative to all games together regardless of the scoring scheme in different games, and facilitates making a direct comparison with DQN. All TPG and DQN scores were greater than random play, thus normalization never produces a negative result.



(a) Group A

(b) Group B

Figure 3: Post-training incremental test results. Once for each consecutive block of 10 generations (x-axis), all policies are re-loaded from checkpoint files and tested in 30 episodes under each game title for a maximum of 18,000 frames each. The LHS plot reports the single-best mean test score achieved in each game by *any* policy (i.e. the scores reported for each game at any point are from 3 different policies), while the RHS plot reports mean scores for the best multi-task policy (see procedure in §5.1 (i.e. the scores reported for each game are from the same policy). All scores are normalized relative to DQN's score in the same game (100%) and a random agent (0%). TPG scores are from the single best of 5 independent runs. DQN scores are from [15].

play achieved by DQN [15] in the same games ($p < 0.05$ from two-tailed test). In group B, game-specific TPG policies in Asteroids and Battle Zone exceed the level of DQN, while the remaining TPG scores simply match scores from DQN in each game (i.e. there is no statistical difference, $p > 0.05$ from two-tailed test).

Table 2: Test results with game switching. The final population of TPG polices in each run are tested in 30 episodes for each game title. TPG-GS reports the best mean test score for each game achieved by any policy (i.e. each score is from a different *specialist* policy graph). TPG-MT reports the best mean score for each game from a single *generalist*, multi-task policy graph. DQN reports test scores from [15], i.e. for policies trained and tested on a single game title only.

Game	TPG-GS	TPG-MT	DQN
Centipede	30689(±734)	14683(±1809)	8309(±5237)
Frostbite	3836(±243)	3092(±416)	328.3(±250.5)
Ms. Pac-Man	4127(±230)	3988 (±598)	2311(±525)
Asteroids	2389(±1044)	1431.4(±409)	1629(±542)
Battle Zone	40933.4(±9892)	23700(±7566)	26300 (±7725)
Zaxxon	4483.7(±1677)	4363.5(±1780)	4977(±1235)

Finally, in order to confirm the significance of game switching, we re-evaluate TPG policy graphs, as developed and reported in a separate study [10], over all three games from group A and B, with the resulting test scores shown in Table 3. Each policy in Table 3 was exposed to a single game titled during evolution (i.e. no game switching takes place). In this case, no policy is able to achieve significant scores in any game other than the title it experienced during training. Thus, within either of the groups considered in this study, proficiency in any single game title does not directly transfer to any other game within the same group.

Table 3: Test results without game switching. The champion (game-specific) TPG policy for each game from [10] is re-tested in 30 episodes under each game title and mean scores are reported. Column headings indicate which game the policy was *trained* for, while rows indicate the active game under test. (i.e. Each policy was trained for *one* game and has never seen the other two before).

	Centipede	Frostbite	Ms. Pac-Man
Centipede	35264(±7886)	2191(±1464)	160(±0)
Frostbite	0(±0)	5358(±2962)	150(±21)
Ms. Pac-Man	60(±0)	210 (±0)	7920(±932)
	Asteroids	Battle Zone	Zaxxon
Asteroids	2673(±625)	500(±900)	300(±180)
Battle Zone	407.3(±192)	40033.4(±5798)	0(±0)
Zaxxon	135.7(±29)	2900(±2203)	6050(±1867)

5.3 Simplicity Through Emergent Modularity

As discussed in Section 4, the simplest stand-alone decision-making entity in TPG is a single team of programs, where all policies are initialized as a single-team of between 2 and ω programs. Throughout evolution, search operators may incrementally combine teams to form policy graphs. By compartmentalizing decision-making over multiple independent modules (teams), and incrementally combining modules into policy graphs, two critical benefits are achieved:

Solution complexity: The number and complement of programs per team and teams per policy graph, is an emergent, open-ended property driven by interaction with the task environment. That is, policies are initialized in their simplest form and only complexify when/if simpler solutions are outperformed (see [10] for empirical evidence of this property).

State space selectivity: Each program indexes a small proportion of the state space. As the the number of teams and programs in each policy graph increases, the policy will index more of the state space *and* optimize the decisions made in each region. However, recall form section 4.2 that a single decision, or mapping from state observation to atomic action, requires traversing a single path from root node to atomic action. As such, while the decision-making capacity of the policy graph expands through complexification, the *cost* of making each decision, as measured by the number of programs which require execution, remains relatively low.

Figure 4 quantifies these properties by examining, for the champion multi-task policy throughout evolution from Group A (RHS of Figure 3(a)), the number of teams per policy vs. teams visited per decision (Figure 4(a)), the proportion of input space covered by the policy as a whole vs. the proportion indexed per decision (Figure 4(b)), and the average number of instructions required for each decision (Figure 4(c)). Table 4 summarizes this data for the champion multi-task policy from each game group. Thus, the ultimate run-time efficiency of policy graphs is a factor of how many instructions are executed to make each decision. Interestingly, even for an evolved multi-task policy graph (i.e post-training), the number of instructions executed depends on the game in play, for example, ranging from 1064 in Ms. Pac-Man to 588 in Centipede for the Group A multi-task champion, Table 4. For perspective, DQN performs millions of weight computations for each decision and defines the architecture a priori, using the same level of complexity for each game.

Table 4: Complexity of champion multi-task policy graphs from each game group. The cost of making each decision is relative to the average number of teams visited per decision (Teams), average number of instructions executed per decision (Instructions), and proportion of state space indexed per decision (%state).

Game	Teams	Instructions	%State
Centipede	3	588	17
Frostbite	4	844	24
Ms. Pac-Man	5	1064	29
Asteroids	5	999	27
Battle Zone	6	1211	31
Zaxxon	6	1244	32

5.4 Modular Task Decomposition

Problem decomposition takes place at two levels in TPG: 1) Program-level, in which individual programs within a team each define a unique context for deploying a single action; and 2) Team-level, in which individual teams within a policy graph each define a unique program complement, and therefore represent a unique mapping from state observation to action. Moreover, since each program typically indexes only a small portion of the state space, the resulting mapping will be sensitive to a specific region of the state space. This section examines how modularity at the *team* level supports the development of multi-task policies.

As TPG policy graphs develop, they will subsume an increasing number of stand-alone decision-making modules (teams) into a hierarchical decision-making structure, or policy graph. The degree to which individual teams specialize relative to each objective experienced during evolution, i.e. the 3 game titles, can be characterized by looking at which teams contribute to decision making at least once during test, relative to each game title.

Figure 5 shows a multi-task TPG policy graph from generation 175 of the group A experiment. The Venn diagram indicates which teams are visited at least once while playing each game, over all test episodes. Naturally, the root team contributes to every decision (black circle in the graph, center of Venn diagram). 7 teams contribute to playing both Ms. Pac-Man and Frostbite, while the rest of the teams specialize for a specific game title. In short, both generalist and specialist teams appear within the same policy and *collectively* define a policy capable of playing multiple game titles.

6 Conclusion

The simultaneous evolution of agent behaviours for multiple game titles is demonstrated under the Atari 2600 gaming environment. To do so, a new approach is proposed for providing emergent modularity in GP, or Tangled Program Graphs (TPG). The resulting TPG solutions match or exceed current state-of-the-art from deep learning (trained on single game titles), while not requiring any more training resource than necessary for developing agent's under a single game title. Moreover, TPG solutions are particularly elegant, thus supporting real-time operation without specialized hardware.

Acknowledgements

This research was supported by the NSERC Discovery and NSGS programs.

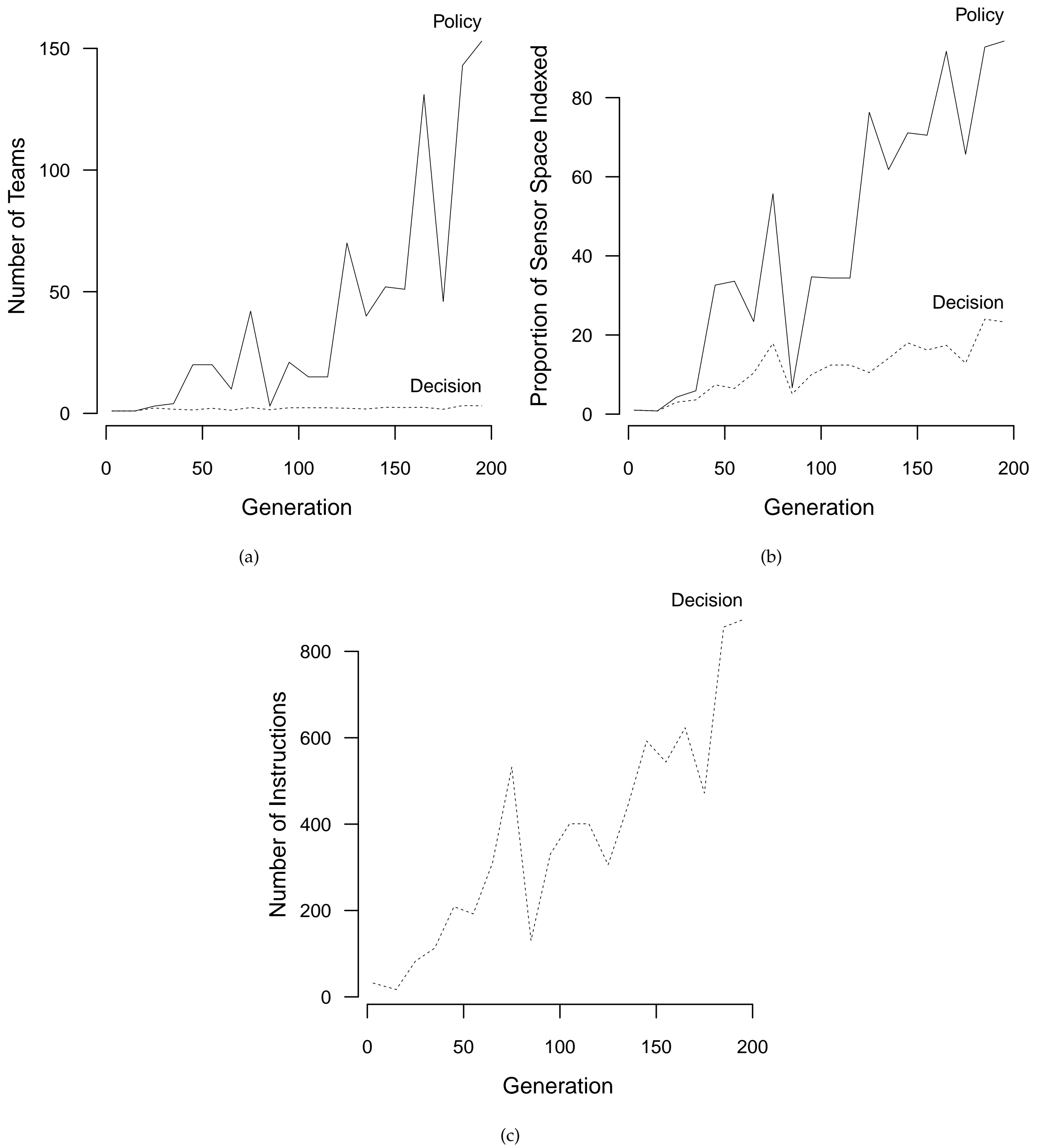


Figure 4: Complexity of champion multi-task policies identified from incremental tests (RHS of Group A, Figure 3). Reported are the number of teams per policy vs. teams visited per decision (a), the proportion of input space covered by the policy as a whole vs. the proportion indexed per decision (b), and the average number of instructions executed for each decision (c).

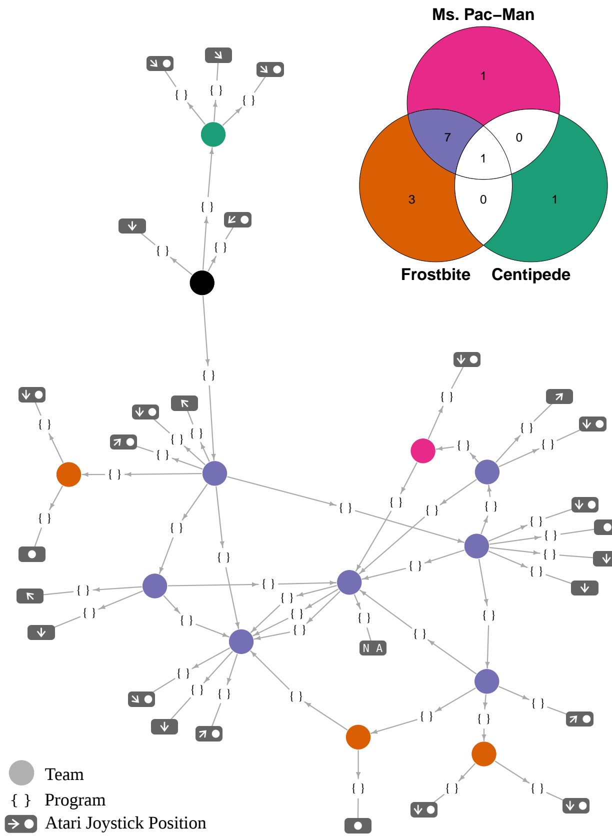


Figure 5: Champion multi-task TPG policy graph from the group A experiment. Decision making in a policy graph begins at the root node and follows *one* path through the graph until an atomic action (joystick position) is reached (See Algorithm 2). Venn diagram indicates which teams are visited while playing each game, over all test episodes. Note that only graph nodes (teams and programs) that contributed to decision-making during test are shown.

References

[1] M. G. Bellemare, Y. Naddaf, J. Veness, and M. 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] A. Braylan, M. Hollenbeck, E. Meyerson, and R. Miikkulainen. Reuse of neural modules for general video game playing. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, AAAI'16, pages 353–359. AAAI Press, 2016.

[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] M. Hausknecht, J. Lehman, R. Miikkulainen, and P. Stone. A neuroevolution approach to general Atari game playing. *IEEE Transactions on Computational Intelligence and AI in Games*, 6(4):355–366, 2014.

[6] M. Hausknecht and P. Stone. The impact of determinism on learning atari 2600 games. In *Workshop at the AAAI Conference on Artificial Intelligence*, 2015.

[7] P. Hingston. A Turing Test for computer game bots. *IEEE Transactions on Computational Intelligence and AI in Games*, 1(3):169–177, 2009.

[8] 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.

[9] S. Kelly and M. I. Heywood. Knowledge transfer from keepaway soccer to half-field offense through program symbiosis: Building simple programs for a complex task. In *Proceedings of the ACM Genetic and Evolutionary Computation Conference*, pages 1143–1150, 2015.

[10] S. Kelly and M. I. Heywood. Emergent tangled graph representations for atari game playing agents. In *European Conference on Genetic Programming*, volume 10196 of *LNCS*, pages 64–79, 2017.

[11] 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.

[12] J. Levine, C. B. Congdon, M. Ebner, G. Kendall, S. M. Lucas, R. Miikkulainen, T. Schaul, and T. Thompson. General Video Game Playing. In S. M. Lucas, M. Mateas, M. Preuss, P. Spronck, and J. Togelius, editors, *Artificial and Computational Intelligence in Games*, volume 6, pages 77–83. 2013.

[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] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[16] Y. Naddaf. *Game-Independent AI Agents for Playing Atari 2600 Console Games*. Masters thesis, University of Alberta, 2010.

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

[18] E. Parisotto, J. L. Ba, and R. Salakhutdinov. Actor-mimic: Deep multitask and transfer reinforcement learning. In *International Conference on Learning Representations*, 2016.

[19] D. Perez-Liebana, S. Samothrakis, J. Togelius, T. Schaul, and S. M. Lucas. General video game AI: Competition, challenges and opportunities. In *Thirtieth AAAI Conference on Artificial Intelligence*, 2016.

[20] Matthew E. Taylor and Peter Stone. Transfer learning for reinforcement learning domains: A survey. *Journal of Machine Learning Research*, 10(1):1633–1685, 2009.

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

[22] G. N. Yannakakis and J. Togelius. A panorama of artificial and computational intelligence in games. *IEEE Transactions on Computational Intelligence and AI in Games*, 7(4):317–335, 2015.