A COMPARISON OF TRAVERSAL STRATEGIES FOR TANGLED PROGRAM GRAPHS UNDER THE ARCADE LEARNING ENVIRONMENT

by

Alexandru Ianta

Submitted in partial fulfillment of the requirements
for the degree of Master of Computer Science

at

Dalhousie University
Halifax, Nova Scotia
April 2021

© Copyright by Alexandru Ianta, 2021

Pentru Aurel Bodîrnea

Table of Contents

List of Tables . . . . **v**

List of Figures . . . . **vii**

Abstract . . . . **xiii**

Acknowledgements . . . . **xiv**

Chapter 1 Introduction **1**

1.1 Visual Reinforcement Tasks 4

1.2 Training in TPG 5
1.2.1 The Initial Population 5
1.2.2 Evolution 8

1.3 Graph Traversal Strategies 9

1.4 Summary 13

Chapter 2 Methods **14**

2.1 Training 17

2.2 Evaluation 20

2.3 Measurement 20
2.3.1 Performance 20
2.3.2 Static Properties 21
2.3.3 Dynamic Properties 22
2.3.4 Action Distributions 22

Chapter 3 Research Environment **24**

3.1 Running Experiments 26
3.1.1 Containerization 26
3.1.2 Container Orchestration 28
3.1.3 Infrastructure-as-Code 30
3.1.4 Automatic Cloud Backups 34

3.2 Capturing Results 36
3.2.1 Metrics 37
3.2.2 Ingestion 43

3.2.3 Storage 45
3.2.4 Querying & Visualization 45

3.3 Summary 49

Chapter 4 Results 51

4.1 Training 51

4.2 Evaluation 58
4.2.1 Performance 59
4.2.2 Static Properties 60
4.2.3 Dynamic Properties 63
4.2.4 Action Distributions 63

Chapter 5 Analysis 67

5.1 Traversal Analysis 68

5.2 Team & Action Utilization 73

5.3 Learner Utilization 76

Chapter 6 Conclusion 79

6.1 Traversal Strategies 79

6.2 Summary 80

Bibliography 82

Appendix A Supplementary Data from other Generations 87

Appendix B The Performance and Future of Looking Glass 106

List of Tables

1.1 Operations found in program instructions and their function. Note that [] denote that the value in the square brackets is used as an index. For DESTINATION this is always an index into the registers of the machine. For SOURCE this can refer to an index into the registers of the machine, or into the input vector depending on the MODE of the instruction. The NEG operation, negates the value found in the register specified by DESTINATION if that value is less than the one referenced by SOURCE. 6

2.1 The 20 Atari games chosen for the Lightbeam experiment . . . 14

2.2 Lightbeam TPG Parameters 18

2.3 Lightbeam TPG Probability Parameters 19

4.1 Champions Available refers to the number of champions that achieved the respective generations of training. Balanced Champion Samples refer to the the portion of the available champions which were used to produce the results. This filtering is done because some metrics of interest are sensitive to the balance of team and learner traversal samples. For example, when considering the frequency of some action for a given game, if we compare 3 instances of team traversal with 1 instance of learner traversal (as that is what we have available) we will skew the data. So instead, in that situation, we would compare only a single instance of team traversal to the learner traversal. . . . 58

4.2 Minimum, mean, standard deviation, normalized mean, maximum and normalized maximum fitness by environment and traversal for champions after 500 generations of training. The instance column refers to the number of champions tested for each traversal type. 60

4.3 Static properties of champions after 500 generations of training. 62

4.4 Dynamic complexity measures averaged over all champions each playing 20 evaluation episodes. Depth refers to the length of the path from the root team to the returned action. 64

A.1 Minimum, mean, standard deviation, normalized mean, maximum and normalized maximum fitness by environment and traversal for champions after 300 generations of training. The instance column refers to the number of champions tested for each traversal type. 88

A.2 Static properties of champions after 300 generations of training. 89

A.3 Static properties of champions after 300 generations of training continued. 90

A.4 Dynamic properties of champions after 300 generations of training. 91

A.5 Minimum, mean, standard deviation, normalized mean, maximum and normalized maximum fitness by environment and traversal for champions after 400 generations of training. The instance column refers to the number of champions tested for each traversal type. 97

A.6 Static properties of champions after 400 generations of training. 98

A.7 Static properties of champions after 400 generations of training continued. 99

A.8 Dynamic properties of champions after 400 generations of training. 100

List of Figures

1.1 Basic Components of Tangled Program Graphs (TPG) 3

1.2 A visualization of TPG during its initialization phase. Two actions have been sampled (A B) and two learners (with their underlying programs) have been generated 'pointing to' those actions, but the first team has not yet been created. 6

1.3 The first root team produced during trainer initialization. The dashed arrows for learners 1 and 2 signify that they were the first two learners to be created, and are thus guaranteed to reference two distinct actions from the action space. 8

1.4 a) We begin execution and the root team passes the input vector along to another team. b) Bidding continues and the input vector again passes to another team. c) Bidding continues, and the input vector is passed along to the last team in this graph. d) Bidding occurs and the learner pointing back to the team in b) wins, but due to the rules of team traversal, we cannot proceed there and therefore fall back to the next highest bid pointing to action t. Figure sourced from [41]. 11

1.5 a) We begin execution and the root team passes the input vector along to another team. b) Bidding continues and the input vector again passes to another team. c) Bidding continues, and the input vector is passed along to the last team in this graph. d) Bidding occurs and the learner pointing back to the team in b) wins, we proceed back. For the given input vector we already know that the learner pointing to the team in c) will win the bid, but by the rules of learner traversal we cannot proceed there and therefore fall back to the next highest bid pointing to action t. Figure sourced from [41]. 12

3.1 Kubernetes Architecture Figure sourced from: [26] 29

3.2 Looking Glass Architecture 36

3.3 The champion for Centipede using team traversal after 500 generations of training. The root team is in teal, while other teams appear in light blue. Arrows represent learners. The teams are arranged radially where their size and distance from the root team is determined by the number of incoming learners. 40

3.5 The fitness of each root team in from a TPG population playing Venture using learner traversal at generation 500. 41

3.4 A path the champion from figure 3.3 used in making the decision to apply the 'FIRE' action to the game environment. . . . 42

3.6 The publish-subscribe ingestion pipeline in Looking Glass. The dashed arrows show the path a message containing Metric A takes through the pipeline. Note the association with Topic A in the Kafka broker and Index A in Elassandra. 44

3.7 Viewing metrics in the `tpg.lightbeam.metrics.generation` index, automatically refreshed every five minutes by Kibana. The histograms along the top show the volume of metrics coming for a given day; individual records appear underneath. . . 46

3.8 The 'Lightbeam' experiment dashboard. The top row charts, from left to right, are the RAINBOW Normalized Maximum Fitness, and the RAINBOW Normalized Average Fitness achieved by TPG, plotted against time. The bottom left graph, shows the average runtime (in hours) vs the # of generations computed averaged across instances of a particular environment. The bottom right Table shows various statistics about the paths traversed to produce actions by champion teams at 50 generation increments during training. 47

4.1 Average normalized fitness being monitored after runs begin on January 13th 2021. 52

4.2 The number of seconds a generation would take to compute for a given environment averaged over all instances. 53

4.3 The number of teams in *purple* and learners in *blue* across team traversal instances for the Bowling environment plotted against the generation when the graphs were sampled on the x-axis. Bowling-TEAM-4, the fourth entry, reaches a staggering 50,184 learners and 3,250 teams compared to the 25,000 learners and 2,500 teams of its peers at the same generation. Note, the apparent drop off in size immediately afterwards for Bowling-TEAM-4 happens because the instance was stopped to conserve computational resources. 54

4.4 The number of teams in *purple* and learners in *blue* across team traversal instances for the Asteroids environment plotted against the generation when the graphs were sampled on the x-axis. Asteroids-TEAM-4, the fourth entry, reaches 45,830 learners and 3,004 teams compared to the 20,000 learners and 2,000 teams of its peers at the same generation. The 'extra' 6th Asteroids-TEAM instance was started in an attempt to have 5 instances at generation 500 for this environment and traversal type after it became clear Asteroids-TEAM-4 was not going to finish in time. 55

4.5 The cumulative number of compute hours averaged over all instances of an environment plotted against generations. 56

4.6 Fitness curves for the Asteroids environment during training, averaged across all instances and split by traversal strategy. . . 57

4.7 Average fitness for the MsPacman environment averaged across all instances and split by traversal strategy. 57

4.8 Average fitness for the MsPacman environment for each experiment instance. The diverging purple curve is that of MsPacman-LEARNER-4. 58

4.9 Action frequencies across all environments for champions after 500 generations of training. 65

4.10 Action frequencies for champions trained on the BattleZone Atari game after 500 generations of training. Split by traversal type. 65

5.1 Hold 'LEFT' and smash 'LEFTFIRE' as much as possible. A TPG champion solution for the Asteroids Atari game. 67

5.2 First 'LEFT', the 'UP', then possibly, success! Venture-LEARNER-8 moments before its demise at the hands of the green enemy after stumbling into the first room and happening upon some treasure. 68

5.3 Line them up, then strike them down. Champions perfect the optimal shot in the Atari Bowling game after 400 generations of training. 68

5.4 Action frequency distribution across all environments for champions with team traversal after 500 generations of training plotted as a percent of the whole against the frame #. Note that many actions occurred infrequently enough to be omitted. . . 69

5.5 Action frequency distribution across all environments for champions with learner traversal after 500 generations of training plotted as a percent of the whole against the frame #. Note that many actions occurred infrequently enough to be omitted. 70

5.6 Action frequency distribution for champions trained on the BattleZone Atari game with team traversal after 500 generations of training plotted as a percent of the whole against the frame #. 71

5.7 Action frequency distribution for champions trained on the BattleZone Atari game with learner traversal after 500 generations of training plotted as a percent of the whole against the frame #. 71

5.8 **Learner traversal** vertex cover analysis for generation 500 champions. In *green*, the percentage of vertices (teams and actions) covered by all paths traced over 20 evaluation episodes for a given champion (listed on the x-axis). In *blue*, the percentage of vertices within bid range of all paths traced over 20 evaluation episodes for a given champion. Champion graphs with fewer than 50 teams were excluded as their 50% - 100% coverage is a reflection of their uniquely small graph size. Square-root y-axis scale used to highlight smaller differences in lower coverages. 74

5.9 **Team traversal** vertex cover analysis for generation 500 champions. In *green*, the percentage of vertices (teams and actions) covered by all paths traced over 20 evaluation episodes for a given champion (listed on the x-axis). In *blue* the percentage of vertices within bid range of all paths traced over 20 evaluation episodes for a given champion. Champion graphs with fewer than 50 teams were excluded as their 50% - 100% coverage is a reflection of their uniquely small graph size. Square-root y-axis scale used to highlight smaller differences in lower coverages. . 75

5.10 **Learner Traversal** edge cover analysis for generation 500 champions. In *green*, the percentage of edges (learners) covered by all paths traced over 20 evaluation episodes for a given champion (listed on the x-axis). In *blue*, the percentage of edges within bid range of all paths traced over 20 evaluation episodes for a given champion. Champion graphs with fewer than 50 learners were excluded as their 50%-100% coverage is a reflection of their uniquely small graph size. Square-root y-axis scale used to highlight smaller differences in lower coverages. 77

5.11 **Team Traversal** edge cover analysis for generation 500 champions. In *green*, the percentage of edges (learners) covered by all paths traced over 20 evaluation episodes for a given champion (listed on the x-axis). In *blue*, the percentage of edges within bid range of all paths traced over 20 evaluation episodes for a given champion. Champion graphs with fewer than 50 learners were excluded (ex: Pitfall, DoubleDunk, etc.) as their 50%-100% coverage is a reflection of their uniquely small graph size. Square-root y-axis scale used to highlight smaller differences in lower coverages. 78

A.1 Action frequencies across all environments for champions after 300 generations of training. 92

A.2 Action frequencies for champions trained on the BattleZone Atari game after 300 generations of training. Split by traversal type. 93

A.3 Action frequency distribution across all environments for champions with team traversal after 300 generations of training plotted as a percent of the whole against the frame #. Note that many actions occurred infrequently enough to be omitted. . . 94

A.4 Action frequency distribution across all environments for champions with learner traversal after 300 generations of training plotted as a percent of the whole against the frame #. Note that many actions occurred infrequently enough to be omitted. 95

A.5 Action frequency distribution for champions trained on the BattleZone Atari game with team traversal after 300 generations of training plotted as a percent of the whole against the frame #. 96

A.6 Action frequency distribution for champions trained on the BattleZone Atari game with learner traversal after 300 generations of training plotted as a percent of the whole against the frame #. 96

A.7 Action frequencies across all environments for champions after 400 generations of training. 101

A.8 Action frequencies for champions trained on the BattleZone Atari game after 400 generations of training. Split by traversal type. 102

A.9 Action frequency distribution across all environments for champions with team traversal after 400 generations of training plotted as a percent of the whole against the frame #. Note that many actions occurred infrequently enough to be omitted. . . 103

A.10 Action frequency distribution across all environments for champions with learner traversal after 400 generations of training plotted as a percent of the whole against the frame #. Note that many actions occurred infrequently enough to be omitted. 104

A.11 Action frequency distribution for champions trained on the BattleZone Atari game with learner traversal after 400 generations of training plotted as a percent of the whole against the frame #. 105

A.12 Action frequency distribution for champions trained on the BattleZone Atari game with learner traversal after 400 generations of training plotted as a percent of the whole against the frame #. 105

Abstract

Tangled program graphs provides a framework for constructing modular genetic programming solutions to visual reinforcement learning tasks. In order to guard against the development of cycles within the resulting graph, and therefore introduce the halting problem, a traversal strategy forbidding the revisiting of vertices was originally assumed. In this thesis an alternative traversal strategy wherein vertex revisits *are* allowed but edge revisits are *not* is explored. An empirical study is performed using 20 game titles from the Arcade Learning Environment in order to assess the relative impact of the different traversal strategies on the resulting agent behaviours and underlying graph characteristics. Ultimately both strategies appear to result in behaviours that are statistically very similar. The most notable differences appear in distributions of actions used to reach the same performance.

Acknowledgements

This work would not have been possible without the advice and guidance from my supervisor Dr. Malcolm Heywood. Your encouragement helped me take the plunge with an honor's thesis, and now a master's, can't wait to find out where this rabbit hole goes.

A special thank you to Dr. Mike Smit who was generously flexible on my work commitments as this thesis was being written.

I would also like to thank the TPG team in the NIMS Lab, Robert Smith, Ryan Amaral, and Caleidgh Bayer for sharing their input, TPG implementations, visualization designs and moral support throughout the process of putting this work together. It was a pleasure working with all of you!

And of course, a big thank you to my family and friends who delightfully tolerated my insanity.

Chapter 1

Introduction

Machine learning (ML) requires that three basic questions need answering: 1) how to express a candidate solution, or the representation problem; 2) how to characterize the performance of a candidate solution, or the cost function; 3) how to modify a candidate solution once its performance has been established, or the credit assignment problem. Naturally, there are many approaches to addressing these questions and corresponding trade-offs once a decision is made to answer each design question in a particular way. In this thesis we are particularly interested in the case of genetic programming [52]. Genetic programming might address the aforementioned ML design questions in the following way:

- **Representation:** an instruction set as applied to a simple register machine under a variable length representation, or linear GP [48, 44]. Thus, we assume that decisions are made a priori regarding what instructions are supported (although that need not be the case [60]). Specifically, instructions take the form of a simple register transfer language of the form $R[x] = R[x]\langle op \rangle(R[y] \cup \vec{s}(y))$. $R[x, y] \in \mathcal{R}$ are references to a finite set of `MaxReg` general purpose registers, $\mathcal{R}$. $\vec{s}(y)$ represents a reference to attribute y from the state space (inputs defined by the task). $\langle op \rangle$ are the set of two argument operations.[1] The $\cup$ operator is used to indicate that the second argument can be one of two forms, a reference to the state space (an input from the task environment) or a reference to the value of a general purpose register. Each instruction is therefore represented as an integer that is 'decoded' to a legal instruction given prior knowledge regarding the number of instructions in the instruction set, maximum number of registers (`MaxReg`) and dimension of the state space (N). In addition, rather than a single candidate solution/agent being maintained at a time, multiple candidate solutions/agents are maintained simultaneously (or the 'population').

[1]Dummy references can be used to extend this to single argument operations.

- **Cost function:** tends to reflect properties of the task. Thus, classification problems might reflect a desire to maximize detection rate on multiple classes of classification whereas function approximation might reflect a desire to minimize the mean square error. In this thesis we are interested in applying GP to episodic reinforcement learning tasks in which state is defined by pixel values sampled from a video frame (§1.1). This means that candidate solutions/agents are required to interact with a task at discrete points in time, $t = 0, 1, \cdots T$, or an episode. At each interaction the agent selects a discrete action that potentially results in a change to the task environment. Task environments in this case are defined by a video game engine (§1.1). The process repeats until either a maximum number of interactions is encountered, or the agent encounters a 'terminal state' (i.e. game won, lost, drawn). The goal of the agent is to maximize the average cumulative reward received over the episode. The cost function can also quantify other performance issues such as solution complexity and computational budget to make a decision.

- **Credit assignment:** encompasses three factors in genetic programming: 1) who in the population of candidate solutions are selected to 'reproduce' or the parents; 2) how to produce 'offspring' from the parents, and; 3) who are replaced in the original population by the offspring. In this work a 'breeder' formulation will be assumed, which is to say that the population as a whole are first evaluated in the task using the cost function. The population is ranked using the relative performance under the cost function. The bottom *Gap* percent individuals are deleted, and a copy made of *Gap* parents selected from the surviving members of the population with uniform probability. Variation operators are then applied to the cloned parents. The resulting offspring are then added to the survivors to define the new population. This defines an iteration or generation of GP. This process is elitist if the task definition is stationary and complete.

The specific formulation of GP assumed in this thesis is that of Tangled Program Graphs or TPG [49, 50, 51]. TPG explicitly composes solutions from multiple programs, thus there can never be less than two programs per agent. The underlying motivation is to decompose a task such that different programs are associated with different aspects of a task, as opposed to assuming that solutions take the form of a

single monolithic program. To facilitate this TPG assumes a representation consisting of program *and* action; hereafter the 'learner'. Such a paradigm was previously demonstrated to scale to visual reinforcement learning tasks (Section 1.1), a theme that will be continued in this thesis.

The TPG reinforcement learning framework coevolves *programs* responsible for sampling an input space into groups called *teams*. Programs are encapsulated in *learners*, and thus, a team has a set of learners. Learners, in addition to containing a program, also hold a reference to either an action or another team. In graph theoretic terms, teams and actions represent vertices and learners represent edges. A team not referenced by any learner (no incoming edges) is considered a *root team*.

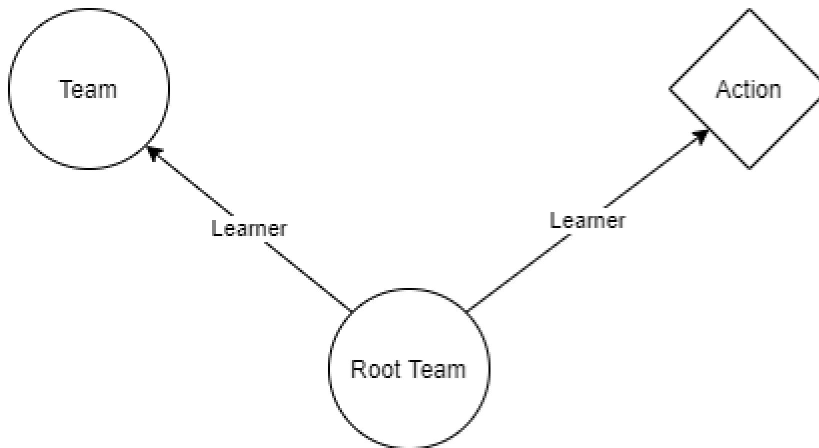

Figure 1.1: Basic Components of Tangled Program Graphs (TPG)

During training, TPG constructs a complex graph using these components (see figure 1.1). The end result is a population of root teams who use their underlying graphs to decide what action to apply to an environment given an input vector describing the environment's current state.

The decision making process works as follows: When an input vector is given to a root team, the root team passes the vector to each of its learners. A simple register machine is used to execute the learner's program. These programs are composed of `ADD`, `SUB`, `MUL`, `DIV`, and `NEG` instructions whose arguments are either fixed values, register values, or values from the input vector. After a program's execution, whatever value is left in register 0 of the machine is returned. This value represents a learner's *bid*. The bids of all learners on a team are compared, and the learner with the

highest bid earns the right to 'suggest' its reference as the action to be applied to the environment. If the learner is referencing another team instead of an action, this process repeats using the learners associated with that team, and so on, until an action is returned [49, 41].

This work describes the first attempt to collect empirical measurements at scale from the inner mechanisms of TPG during evolution and evaluation. This undertaking was motivated by a desire to determine the effect of a different graph traversal strategy for root teams to decide on actions. What follows in this chapter is a brief description of visual reinforcement tasks (1.1), TPG's evolutionary process (1.2), and a deeper look at the role of graph traversal strategies(1.3). Chapter 2 describes the methods used to test the alternate traversal, while chapter 3 dives into the software platforms and hardware infrastructure that allowed us to collect and produce the results in Chapter 4 on the basis of millions of data points. Finally, Chapters 5 & 6 provide analysis and concluding remarks.

1.1 Visual Reinforcement Tasks

In this work we use a set of visual reinforcement tasks to benchmark TPG's performance. In these kinds of tasks reinforcement learning is driven by interactions with an environment that can be visually observed. In our case these environments are a selection of games released for the Atari game console [43]. Through the use of a framework described later in chapter 2, we receive the current state of a game as a vector of pixel values. These pixels are defined as triplets whose values correspond with those of the red, green and blue (RGB) color channels [41]. For our Atari games the size of this vector is $210 \times 160 \times 3$ (height $\times$ width $\times$ RGB) [6]. Before passing this vector to TPG we first downsample it to $105 \times 80 \times 3$ using the mean function over $2 \times 2 \times 1$ blocks. Then we flatten the vector to 8400×1 by first mapping the RGB triplet to a single unique value and then flattening the result.

Our Atari game environments also have action spaces defined by the actions a player can perform on the joystick and the subset of those which are valid for the game in question. For example, if there are five valid actions in the game, the action space might look like this: {0,1,2,3,4}. Where each value corresponds with a given action like FIRE,LEFT,RIGHT,UP,DOWN.

This downsampled, flattened, input vector is passed to a root team in TPG which returns an action back to the script driving the interaction. The returned action is applied to the game state and the subsequent state as well as a reward produced by the environment (i.e. change in game score at that frame) is emitted by the game environment. This process then repeats until such time that the game reaches an end condition like winning, losing, or reaching the frame limit for this play session (the episode). The sum of the reward accrued over the course of an entire episode is the fitness achieved by that root team. In the case of our Atari games this reward corresponds to the same score a human player would receive playing the game [41]. To mitigate the effect of 'lucky' outliers, our experiments play multiple episodes during each period of evaluation, and the mean fitness a root team achieves over these episodes is used during the process of evaluating the cost function.

1.2 Training in TPG

Training in TPG is the process in which a population of root teams are evaluated on a common task, and this performance information used to rank each root team. This is then used to generate a new population of root teams based on the best performing teams of the current generation or the breeding process described above for credit assignment under GP. This section describes how the initial population is created, and how the evolutionary process works on a population to create the subsequent generation's population.

1.2.1 The Initial Population

TPG is an evolutionary reinforcement learning framework based on genetic programming. The exact mechanisms of evolution depend on the implementation of TPG used, in this section we will focus on the the python implementation (pyTPG) used in this work.[2] PyTPG uses a *trainer* object to manage the data structures and states of the algorithm. During the initialization of the trainer, we must pass it the action space of the task at hand, the trainer then samples two unique actions from that space. It will then create two learners, each of which will reference one of the sampled actions, see figure 1.2.

[2] `https://github.com/Ryan-Amaral/PyTPG`

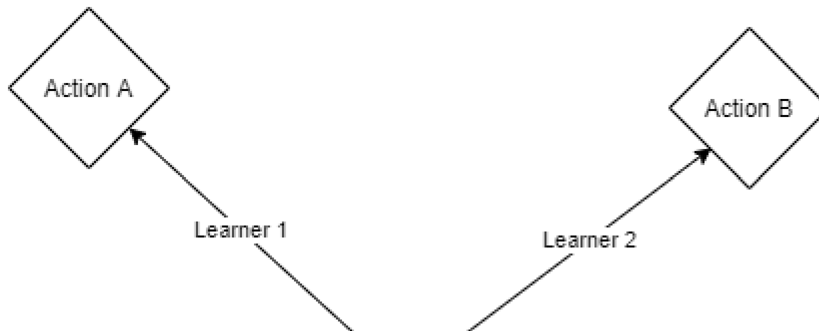

Figure 1.2: A visualization of TPG during its initialization phase. Two actions have been sampled (A B) and two learners (with their underlying programs) have been generated 'pointing to' those actions, but the first team has not yet been created.

During the process of initializing the learners we create programs for each learner by generating register machine instructions of the form ⟨MODE⟩⟨OP⟩⟨DESTINATION⟩⟨SOURCE⟩. MODE can take the value of either 0 or 1. If it is 0, the SOURCE value will be interpreted as an index into one of the registers of the machine. If it is 1, the SOURCE value will be interpreted as an index into the input vector upon which the program is run. The number of registers and the size of the input vectors are part of the parameters required to create a trainer. OP can take on a value between 0 and 4 inclusively, corresponding to the operations: ADD,SUB,MUL,DIV and NEG. ADD and SUB are binary operations, while MUL,DIV and NEG are unary operations. Table 1.1 describes the behavior of each of these operations.[3]

Operation	**Function**
ADD	[DESTINATION] + [SOURCE]
SUB	[DESTINATION] − [SOURCE]
MUL	[DESTINATION] ×2
DIV	[DESTINATION] ÷2
NEG	IF [DESTINATION] <[SOURCE] THEN [DESTINATION] ×= -1

Table 1.1: Operations found in program instructions and their function. Note that [] denote that the value in the square brackets is used as an index. For DESTINATION this is always an index into the registers of the machine. For SOURCE this can refer to an index into the registers of the machine, or into the input vector depending on the MODE of the instruction. The NEG operation, negates the value found in the register specified by DESTINATION if that value is less than the one referenced by SOURCE.

[3] The multiplier in MUL and the divisor in DIV are fixed to 2, as a result of prior work demonstrating negligible benefit from allowing them to vary [56, 53].

Thus an instruction can be generated by first randomly choosing a `MODE` between 0 and 1, an `OP` between 0 and 4, a `DESTINATION` between 0 and the number of registers in the machine minus 1, and a `SOURCE` between 0 and the number of registers in the machine minus 1 *or* between 0 and the size of the input vector minus 1, depending on what value was chosen for the `MODE`. We define an algorithm parameter *initMaxProgSize* which is given to the trainer at initialization, to determine the number of these instructions to generate for the initial learner population. Once each learner has generated a program of the desired length we create a team that references both of the learners. Because no learners point to this team, it is a root team. Finally, we continue to sample actions from the action space, create learners (and their corresponding programs) until we reach the *initMaxTeamSize*. Figure 1.3 shows a root team generated during initialization. The first two learners are identified by dashed lines, they are important because they ensure that despite the stochastic nature of the initialization process there are at least two distinct actions associated with the team. There is no rule dictating that multiple learners may not reference the same action, as can be seen in figure 1.3 through learners 2 and 3. This is because learners 2 and 3 are still unique due to differences in their underlying programs.

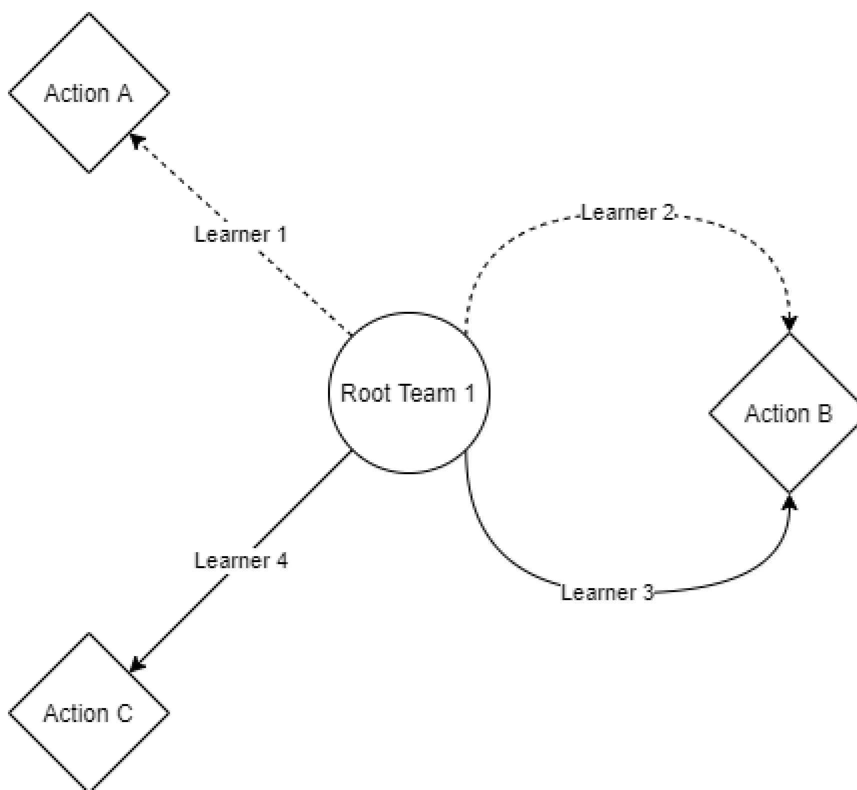

Figure 1.3: The first root team produced during trainer initialization. The dashed arrows for learners 1 and 2 signify that they were the first two learners to be created, and are thus guaranteed to reference two distinct actions from the action space.

This whole process repeats to create as many root teams as defined in the *teamPopSize* parameter, thus creating the population for generation 0.

1.2.2 Evolution

In pyTPG, evolution is a two step process. First we select the members of the root team population which are to be retained, and purge the rest of them (as per above 'breeder' model of credit assignment). Then we generate new root teams by cloning from the selected root teams and mutating their clones. New root teams are generated in this manner until we again reach our desired *teamPopSize*.

The selection process works by ranking root teams by the fitness they've achieved during the last round of episodes. A *gap* parameter represents the number of teams to drop after this ranking takes place. For example, with a gap of 0.5, half of the root teams would be purged during this step. The only exception to this rank and purge procedure, occurs if the algorithm parameter *elitist* is set to true. In that case, the

root team that has achieved the highest fitness thus far is always preserved, even if it performed poorly compared to its peers in the last round.

During the generation process, random root teams that survived selection are chosen to be cloned. Their clones then under go iterations of mutation in which, learners may be added, removed, or themselves mutated. Multiple things can occur when a learner is mutated. There is a chance that the underlying program will be mutated by adding, deleting, or swapping its instructions. There is also a chance that a learner will mutate its reference, this can happen in one of two ways: either a new action from the action space is sampled and referenced, or a random team (not necessarily a root team) is chosen from the global population as the new reference of the learner.

A process we call *rampant mutation* dictates how many iterations of mutation occur on these clones. *Rampancy parameters* allow us to configure this process. Rampancy parameters take the form: `<INTERVAL>,<MIN>,<MAX>` where `INTERVAL` specifies the number of generations between rampancy, `MIN` specifies the minimum number of mutation iterations, and `MAX` specifies the maximum number of mutation iterations. For example the rampancy parameters `5,3,10` indicate that every 5th generation will be subject to rampant mutation, where a random number of mutation iterations between 3 and 10 will be performed. On generations where rampancy does not occur, only one iteration of mutation takes place, thus the parameters `1,1,1` would be analogous to no rampancy at all. For this work we used the parameters `1,5,5` resulting in every generation being subject to 5 iterations of mutation to produce the subsequent generation's population. Rampant mutation was developed by a colleague in the NIMS lab, Robert Smith, while working on ways to increase genetic diversity throughout training in TPG.

1.3 Graph Traversal Strategies

In section 1 we showed how a root team in TPG, given an input vector, decides on what action to apply to the environment. Because teams, learners and actions can be thought of in graph theoretic terms as vertices and edges, this decision making process amounts to a graph traversal, beginning at the root team, and descending through learners until an action is found. When considering a single root team, it is

tempting to think of these graphs as trees. However during evolution it is possible for a learner to mutate in such a way that its new reference points to a team which has a learner pointing to the team whose learner is currently mutating, thus introducing the possibility of entering a cycle which would result in no action being returned.

Two mechanisms exist within TPG that aim to prevent this from happening. The first happens during evolution, where we ensure that all teams have at least one learner pointing to an action. The second happens at execution time, that is when an action is being decided upon as a function of an input vector, here we keep a list of teams that we have already visited on our traversal path, and we forbid following a learner whose action returns us to a previously visited team. This is the default behavior of TPG and will henceforth be referred to as **team traversal**, figure 1.4 demonstrates team traversal in action.

We now arrive at the motivation for this thesis. In this thesis the question posed is to ask what happens if we allow TPG to revisit previously visited teams during execution, and instead prevent cycles from occurring by stipulating that no *learner* should be visited twice. This approach will be referred to as **learner traversal**, and is shown in action by figure 1.5.

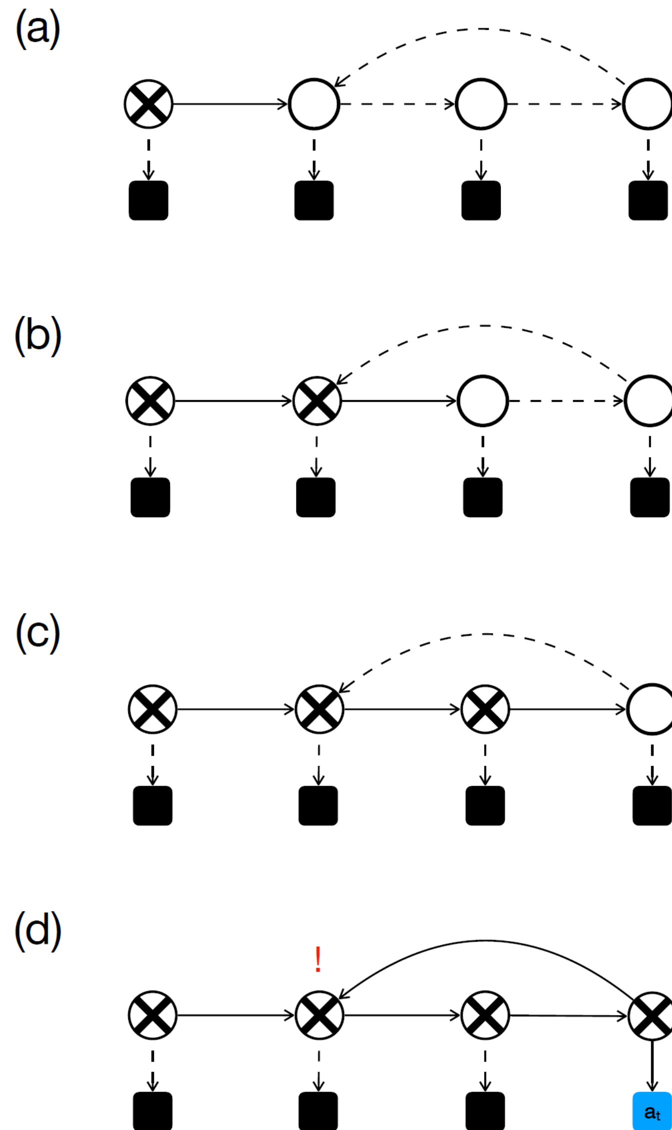

Figure 1.4: a) We begin execution and the root team passes the input vector along to another team. b) Bidding continues and the input vector again passes to another team. c) Bidding continues, and the input vector is passed along to the last team in this graph. d) Bidding occurs and the learner pointing back to the team in b) wins, but due to the rules of team traversal, we cannot proceed there and therefore fall back to the next highest bid pointing to action t. Figure sourced from [41].

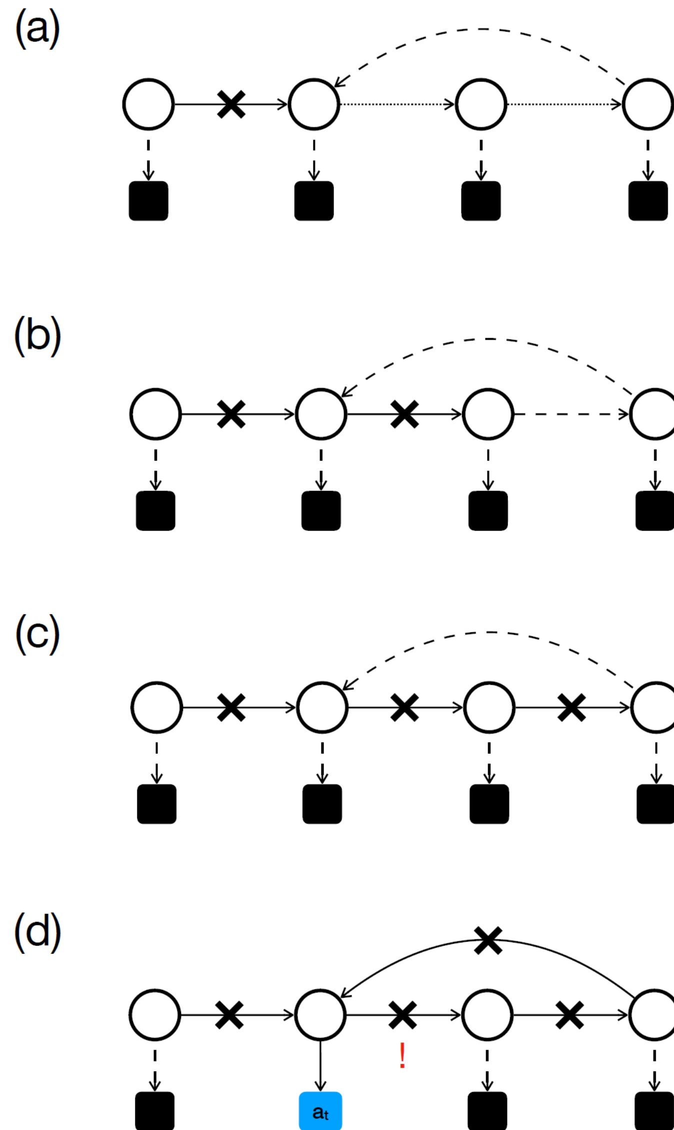

Figure 1.5: a) We begin execution and the root team passes the input vector along to another team. b) Bidding continues and the input vector again passes to another team. c) Bidding continues, and the input vector is passed along to the last team in this graph. d) Bidding occurs and the learner pointing back to the team in b) wins, we proceed back. For the given input vector we already know that the learner pointing to the team in c) will win the bid, but by the rules of learner traversal we cannot proceed there and therefore fall back to the next highest bid pointing to action t. Figure sourced from [41].

It is unclear what kind of effect this change in graph traversal strategy would have. Perhaps this new freedom to re-visit teams could manifest in smaller overall graph sizes as more information could be encoded in fewer vertices. Performance gains could arise from the fact that revisits do not incur additional computational cost

through program execution. The revisited team has by definition already performed its bidding round for the current input vector, so we can simply store and reuse the bids to determine the next learner without having to execute any new programs.

1.4 Summary

TPG is a reinforcement learning framework based on genetic programming and the co-evolution of teams of programs that sample the input space, and learners who stitch these teams together to create a tangled program graph. In this graph a population of root teams exist each of which we will use to play a variety of Atari games, observing what kind of effect different graph traversal strategies have on the global graph, the fitness of the root teams, performance and other characteristics of TPG.

In order to develop this topic, we first introduce the OpenAI gym distribution for the Atari reinforcement learning tasks and establish how TPG interfaces to the game engine (§2). Also addressed are the criterion for performance evaluation and metrics employed for describing/distinguishing between static and dynamic properties of TPG solutions. Section 3 summarizes how we addressed the management of large amounts of information generated by the resulting performance evaluation. Which is to say that over 20 game titles for two different marking schemes and five initializations per title, terabytes of performance/save data were collected. Developing a framework to scale to the management and recovery of the runs as well as supporting the post training analysis and organization of the resulting data was a significant undertaking. Section 4 details the results of the empirical evaluation. To do so, both training and test performance is considered, as are static and dynamic properties of the TPG graphs under the two marking schemes. Performance relative to two deep learning frameworks is also included in order to provide further context to these results. Section 5 summarizes findings from the empirical study and overall conclusions are drawn in Section 6.

Chapter 2

Methods

The OpenAI Gym is a python toolkit for reinforcement learning research that provides, amongst other things, dozens of Atari games under a standard API wrapper which AI agents can interact with[45, 16]. The base TPG implementation used for this work was developed by a fellow lab member, Ryan Amaral[42]. It is a python implementation dubbed 'pyTPG'. The primary experiment, henceforth, 'Lightbeam', would consist of testing pyTPG's performance playing a selection of 20 Atari games, (see Table 2.1) that straddle the performance from two benchmark deep learning algorithms: RAINBOW and DQN [47, 54]. For example, DQN represents the original demonstration of visual RL using deep learning in which DQN performed significantly better than 'human play' on 3 of the 20 titles and worse on 11 of the 20 titles [54]. Rainbow represents a deep learning framework that incorporated six different optimizations developed since DQN and returned the best benchmarking performance against 6 other deep learning approaches [47], was the best algorithm on 6 of the 20 game titles from Table 2.1. In short, the 20 titles span a combination of games that deep learning still find challenging.[1]

Environment			
Asteroids-v0	FishingDerby-v0	Kangaroo-v0	Robotank-v0
BattleZone-v0	Freeway-v0	Krull-v0	Skiing-v0
Bowling-v0	Frostbite-v0	MsPacman-v0	TimePilot-v0
Centipede-v0	Gravitar-v0	Pitfall-v0	Tutankham-v0
DoubleDunk-v0	IceHockey-v0	PrivateEye-v0	Venture-v0

Table 2.1: The 20 Atari games chosen for the Lightbeam experiment

.

PyTPG provides a mechanism for modifying aspects of the internal algorithm through a configuration system that leverages python's ability to swap functions at

[1] Montezuma Revenge is often considered difficult on account of not rewarding intermediate puzzle solving steps. However the 'Pitfall' title also has this property and is included in the 20 titles used in the benchmarking conducted in this thesis.

run-time. To use this system, one determines which functions of the algorithm must be changed to implement the desired customization, then one implements the customized versions of these functions in a separate python script. Finally, new command line parameter(s) are added to allow switching between the customized functions and the default implementations provided by pyTPG. When the configuration system is passed the command line arguments, it determines which underlying functions should be used during execution.

We use this mechanism to implement learner traversals in pyTPG. When a game state is returned by an Atari Gym environment, it is passed to a root team through its `act(self,state,visited,...)` function which returns the action to be applied. This 'act' function passes the game state to all learners attached to the root team so they may return bids determining which of the learners earns the right to 'suggest' its action. The default implementation of this function (see Listing 2.1) implements team traversal.[2]

Listing 2.1: Act function in Team.py implementing team traversal. Parts of this function have been omitted or truncated for brevity.

```
def act(self, state, visited):

    # Throw an exception on revisit
    if str(self.id) in visited:
        raise(Exception("Already visited team!"))

    # Add this team's id to the list of visited ids
    visited.append(str(self.id))

    '''
    Valid learners are ones which:
        * Are action atomic
        * Whose team we have not yet visited
    '''
```

[2]Both traversal code samples make use of *lambdas*. These are anonymous function that are defined inline following the lambda keyword.

```
    valid_learners = [lrnr for lrnr in self.learners
            if lrnr.isActionAtomic() or
            str(lrnr.getActionTeam().id) not in visited]

    # Find the learner with the highest bid
    top_learner = max(valid_learners,
        key=lambda lrnr: lrnr.bid(state))

    # Return the action suggested by that learner
    return top_learner.getAction(state, visited=visited)
```

Note how the function keeps track of which teams have already been visited through a list dubbed 'visited'. If the `top_learner` returns another team as its action, this list is passed to that team's `act` function. When using learner traversals we would like this function to instead keep track of learner ids in the visited list, and only collect bids from learners that have yet to be visited. `act_learnerTrav` in Listing 2.2 implements these changes.

Listing 2.2: `act_learnerTrav` replaces the act function when `--traversal learner` is passed as a command line parameter.

```
def act_learnerTrav(self, state, visited):

    '''
    Valid learners are ones which:
        * Are action atomic
        * We have not yet visited
    '''
    valid_learners = [lrnr for lrnr in self.learners
                if lrnr.isActionAtomic() or
                str(lrnr.id) not in visited]

    # Find the learner with the highest bid
    top_learner = max(valid_learners,
        key=lambda lrnr: lrnr.bid(state))
```

```
# Add the top_learner's id to the visited list
visited.append(str(top_learner.id))

# Return the action suggested by that learner
return top_learner.getAction(state, visited=visited)
```

2.1 Training

500 generations of training were done with team traversal for each of the 20 environments in Table 2.1; the same was done with learner traversal. The number 500 was chosen in part through a result of informal test trials on earlier versions of pyTPG which showed fitness plateauing around the 500 mark. These tests however were performed before the addition of rampant mutation. To mitigate the influence of outliers, 5 instances of each environment-traversal combination were run for a total of (2 traversal types $\times$ 20 games $\times$ 5 instances) 200 runs performing a cumulative 100,000 generations of training. 'Lightbeam' refers to the final set of these experiment instances, distinguishing them from several previous attempts at running this same experiment which failed as a result of the complexities in operating the Looking Glass platform for the first time, integrating with ACENET, or human error. This distinction is necessary as partial data from past attempts, as well as data from unrelated experiments using Looking Glass, would also exist on the platform and would have to be excluded from our analysis.

Aside from the environment, traversal type and instance identifier, the rest of the algorithm parameters remained fixed to the values used by Stephen Kelly in earlier work [50, 49] (see Table 2.2 and 2.3).

Parameter	Value	Description
Episodes	5	The number of times a root team plays the game after evolution. The average fitness achieved across these playthroughs are used to evaluate the root team.
End Generation	500	The number of generations at which training should stop.
Run Key	lightbeam	The experiment to which this run belongs.
Frames	10000	The maximum number of frames in a single playthrough of a game. If the game doesn't reach an end state by this number of frames it is terminated.
Initial Team Pop.	360	The number of root teams in generation 0.
Initial Max Team Size	5	The maximum number of learners a root team at generation 0 will have.
Initial Max Program Size	128	The maximum number of instructions a program at generation 0 will have.
Gap	0.5	The percentage of root teams to eliminate during selection at evolution-time, expressed as a decimal.
Input Size	8400	The size of the game state vector passed to TPG after downsampling and flattening.
Register Count	8	The number of registers in the state machine on which programs are run.
Eliteist	true	If true, ensures the highest scoring root team is never eliminated during selection.
Rampancy Params.	1,5,5	Dictates that every generation, during evolution, the mutation step should be applied 5 times.

Table 2.2: Lightbeam TPG Parameters

Parameter	Value	Description
Learner Delete Prob.	0.7	The probability that learners are deleted from a team during mutation.
Learner Add Prob.	0.7	The probability that learners are added to a team during mutation.
Learner Mutate Prob.	0.3	The probability that a learner is mutated during mutation.
Program Mutate Prob.	0.66	The probability that a program is mutated during mutation.
Action Mutate Prob.	0.33	The probability that an action associated with a learner is mutated.
Atomic Action Prob.	0.5	The probability that an action mutates to an atomic action.
Instruction Mutate Prob.	1.0	The probability that an instruction is mutated by flipping a bit.
Instruction Add Prob.	0.5	The probability that an instruction is added to a program.
Instruction Delete Prob.	0.5	The probability that an instruction is deleted from a program.
Instruction Swap Prob.	1.0	The probability than an instruction is swapped from a program.

Table 2.3: Lightbeam TPG Probability Parameters

The bulk of these experiments were run on the 'Cedar' compute cluster provided by the Atlantic Computational Excellence Network (ACEnet) program. The ACENET program is a collaboration between academic institutions in Atlantic Canada which aims to support computationally intensive research[2]. Cedar itself is a compute cluster that makes thousands of Intel cores, terabytes of RAM and storage space, as well as hundreds of NVIDIA GPUs available to researchers [8]. A script was written encoding the above parameters while provisioning compute nodes with 24 cores and 8GB of RAM for each experiment instance. The maximum amount of time a single job is allowed to take on Cedar is 28 days. Thus, a mechanism for saving and restarting an experiment was implemented (further discussed in section 3.1.4). The job priority of these experiments varied over time as a function of resources used. Users that have used fewer resources had higher priority [21, 8]. Given that we had 200 experiments to run, we found our jobs queuing less often as our experiments progressed. To supplement our computational needs jobs were also started on the NIMS cluster (our local hardware resources, further described in Chapter 3) though using only 5 cores

per job there.

2.2 Evaluation

Once we trained a champion for 500 generations, it would be run through an evaluation phase consisting of 20 playthroughs (episodes) of the game it was trained on. To evaluate the generalized performance of of a champion, in contrast to the playthroughs done in training, a random number of frames between 1 and 30 were skipped at the start of the game. This aims to control for champions who merely 'memorized' one series of actions that maximize fitness.

This phase is significantly less computationally intensive compared to training and as such was run on a single workstation.

2.3 Measurement

To capture the effects different graph traversals might have, data was gathered about as many characteristics of the algorithm as possible within time, computational and creative constraints. This was deemed necessary as the traversal change is subtle and it is unclear where its impacts may manifest. Measured characteristics fall within the following categories: performance, static properties, dynamic properties, and action distributions. All the statistics measured are with respect to their environment and traversal type unless otherwise stated. The majority of these statistics are computed from the evaluation phase, however some, in particular those with respect the computed generations (such as fitness curves) necessarily come from the training phase.

2.3.1 Performance

Performance data primarily encompasses the measurement of fitness and related statistics.

- Minimum Fitness
- Mean Fitness
- Maximum Fitness

- RAINBOW Normalized Maximum Fitness
- RAINBOW Normalized Mean Fitness

Normalizing performance, f_{score}, relative to the results from Rainbow provides a relative measure of performance and enables the display of results across different game titles using a common performance plot. The normalization is adopted from that in the original DQN benchmarking [54], but with the Rainbow final performance as the normalizing factor. Thus, the normalization has the form,

$$f_{score}(i) = 100 \times \frac{TPG(i) - rnd(i)}{RB(i) - rnd(i)} \tag{2.1}$$

where $TPG(i)$ is the average TPG agent score on game title i, $RB(i)$ is the equivalent performance for Rainbow, and $rnd(i)$ the average score for a random policy on title i (see [54, 46] for random agent performance).

2.3.2 Static Properties

These are properties of the champions at a given generation which do not vary between episodes. They are primarily comprised of measurements of the champion's graph.

- Champion Graphs
- Champion graph diameter with respect to root [3]
- # of Learners
- # of Teams
- # of Instructions
- # of Learners per Team
- # of Instruction per Team

[3] Diameter with respect to root refers to the 'longest shortest path' to any vertex from the vertex representing the root team [17].

The objective of collecting these properties is to characterize the overall complexity of the TPG graphs. However, they do not reflect the complexity of a root team's decision making. Which is to say that in order to suggest an action on any given state, TPG execution begins at the root team's vertex. All learners are evaluated, resulting in the selection of one action. Such an action might be atomic or a reference to another team in the root team's graph. In the former case this would be the end of the TPG graph evaluation. In the latter the learners associated with the next team are all evaluated, with the process repeating until an atomic action is encountered. This implies that there is a considerable difference between the 'static' complexity of a TPG graph (i.e. the graph containing all root teams, everything) and what is evaluated in order to map from a root team to an atomic action or the 'dynamic' complexity.

2.3.3 Dynamic Properties

These are properties of the champions at a given generation which vary between episodes, they are collected by recording individual paths taken by champions through their graphs from their root team to the atomic action they return.

- Maximum Path Depth
- Mean Path Depth
- Minimum Path Depth
- Average Instructions per Action
- Average Execution Time

As noted above these metrics are designed to characterize how much of a root team's graph is actually 'visited' in order to make a decision.

2.3.4 Action Distributions

Action distributions refer to the frequencies of actions during play. Frames here refers to an index into the number of frames played during an episode, it functions as a proxy for time. The raw number of appearances is the most basic measurement, while the

frequency over frames data shows us if the different traversals use different actions at different points in the game.

- Action Frequency
- Action Frequency over Frames

The action distribution metrics will be used to imply whether there is any significant difference in the overall 'behaviour' between team and learner marking schemes. Thus, for example, it might transpire that the two marking schemes result in agents that have the same quality of play (as defined by game score), but do so using different action distributions.

Chapter 3

Research Environment

A major challenge in exploring the effect of adaptations on TPG are the logistics of running experiments at scale. To produce sound data we would like to run multiple instances of the algorithm both with and without an adaptation, holding other parameters the same.

For popular machine learning algorithms there exist tools and frameworks that help one develop experiments, producing desired metrics and figures [32][20]. However TPG has been implemented from scratch in the NIMS (Network Information Management and Security) lab, and therefore lacks the feature support of these popular frameworks.

The lab has produced several implementations over the years, however two implementations in particular get the most use. TPG-J a java implementation, and pyTPG a python implementation. These implementations are not interchangeable, each differ in some aspects from each other and the original TPG paper by Stephen Kelly (implemented in C++). This work leveraged pyTPG as the OpenAI Gym toolkit integrates best with python and provides a test bed of dozens of Atari games to train and evaluate TPG on.

Initial experimentation involved forking the pyTPG git repository, patching in the necessary adaptions to support learner traversals and then adding in code which would measure the metrics of interest and output them to a file. With over 23 algorithm parameters and many more auxiliary parameters dictating the number of threads to use, paths to result files, etc. starting even a few experiments was error prone. Once results were produced, the task of collating them together for comparison was not trivial.

To run a simple experiment, say testing an adaptation on three different Atari games, one would require six instances. One per game, with the adaptation, and one per game without the adaptation. While the pyTPG implementation does offer

some parallelization, computing several hundred or thousands of generations could still take a few weeks. Thus, to perform this experiment on a reasonable timescale one would like to start six instances of the algorithm on six different machines. This meant installing pyTPG's dependencies on six different machines, running the modified implementation, and later coming back to each of the six machines to retrieve the results.

It became evident that it would not be possible to produce correct results at scale in this manner, much less analyze them on the timeline of a master's thesis. The issues described above can be summarized into the following broad categories:

Execution Challenges:

1. Installing pyTPG dependencies was time consuming, and had to be done on as many machines as one wanted to use in parallel.

2. Each instance in an experiment had to be started manually.

3. Starting a TPG instance was error prone due to the number of parameters.

4. If an instance was interrupted it would have to be restarted from scratch.

Analysis Challenges:

1. Metrics produced by each instance had to be retrieved manually.

2. Visualizing metrics involved the creation of experiment specific scripts that parsed result files.

3. Analyzing metrics was time consuming and error prone.

The following sections describe the systems and infrastructure provisioned to address these challenges and perform the experiment central to this thesis. However, these systems were designed to be as flexible and extensible as possible, and could therefore be leveraged with minimal effort by other researchers working on custom machine learning algorithms to create a research environment conducive to producing robust results at scale.

3.1 Running Experiments

The NIMS lab owns around 10 geographically co-located servers with varying amounts of CPU, RAM, and Disk resources. This is our primary deployment environment for tooling, testing and light experimental computation. In addition, a separately provisioned virtual machine (VM) hosted on the Dalhousie Open Stack cloud environment acts as a control plane (see section 3.1.2). Together these make up what will be referred to as the NIMS Cluster.

The bulk of the experimental computation for this work was executed on the Cedar cluster provided by the Atlantic Computational Excellence Network (ACENET) under the Compute Canada program.

Four technologies are leveraged together to address the execution challenges described. Containerization, container orchestration, infrastructure-as-code, and automatic cloud backups.

3.1.1 Containerization

Containerization is the process of creating discrete run-time environments setup with the bare necessities required to execute an application (containers). They package up code and dependencies so that applications can be deployed quickly and reliably from one computing environment to another [35].

The Docker container engine was the base of our containerization effort. In the process of containerizing an application, one produces an image, which is then run in a container. In Docker, images are built using a 'Dockerfile' which describes the execution environment that the application requires to run. Listing 3.1 is an excerpt from the Dockerfile used to create the pyTPG docker image used to run an experiment instance. The command `docker build . -t nimslab/tpg-v2:lightbeam` tells docker to build an image using the Dockerfile found in the current directory and tag it with the name `nimslab/tpg-v2:lightbeam`.

Listing 3.1: Excerpt of PyTPG Dockerfile

```
FROM python:3.6


#Update & get sudo & cmake
```

```
#Need sudo for llvm install
#Need cmake for gym['atari'] install
RUN apt-get update &&
    apt-get -y install sudo &&
    apt-get -y install cmake

#Install llvm as pre-req for llvmlite python module
RUN apt-get update && sudo apt-get -y install llvm

RUN python -m pip install --upgrade pip

RUN pip install numpy
RUN pip install llvmlite
... # Other dependencies truncated for brevity
RUN pip install numba
RUN pip install gym['atari']

#Copy NIMS Python library files into image
COPY . .

ENTRYPOINT [ "python3" ,"./nims/experiments/atari.py"]
```

This Dockerfile sits in the working directory of the NIMS Python Library which contains a collection of utilities and the `atari.py` script which initializes pyTPG and runs it on the Atari Gym environment.

To simplify the creation of complex application environments Dockefiles allow one to build upon other images. Popular frameworks, tools, libraries, languages, etc. provide base images from which to build on Docker Hub [13]. In the excerpt we begin with `FROM python:3.6` to specify that our image should build upon the the python image tagged with version 3.6. The python image is itself built upon a Debian image [31].

The next portions of listing 3.1 execute commands that install the system dependencies required by some of the python dependencies used by PyTPG and then the

python dependencies themselves.

The `COPY . .` command is used to copy the files from the working directory into the docker image. Importantly this includes the `atari.py` script that actually runs an experiment instance.

Finally `ENTRYPOINT [ "python3","./nims/experiments/atari.py"]` tells Docker what command to run when a container is created using the `nimslab/tpg-v2:lightbeam` image.

The final step is to run `docker push nimslab/tpg-v2:lightbeam`, which uploads the image to the NIMS lab docker hub account. This allows us to pull our image onto other machines, sparing us the trouble of building the image on every machine we would like to run pyTPG on.

At this point, we have addressed the first execution challenge. Running pyTPG is now trivial on any machine with docker installed. We simply execute:

```
docker run -it nimslab/tpg-v2:lightbeam <params>
```

and Docker will automatically pull the image from docker hub and start a container in which the `atari.py` script is executed using the parameters passed in the `docker run` command.

3.1.2 Container Orchestration

Container orchestration refers to the process of automating the deployment, management, scaling and networking of containers [36].

The challenge that orchestration allows us to address is that, even with containerization, we still have to manually logon to every server we would like to run pyTPG on and execute the `docker run` command. With orchestration we can configure all our available servers into a cluster and let an orchestration engine distribute the execution of containers in the most efficient way possible given the compute resources available on each cluster member, henceforth referred to as a node(s).

The NIMS cluster is a Kubernetes (K8) cluster. In kubernetes, a special subset of node(s), called 'control-planes' act as the command and control nodes for the cluster.

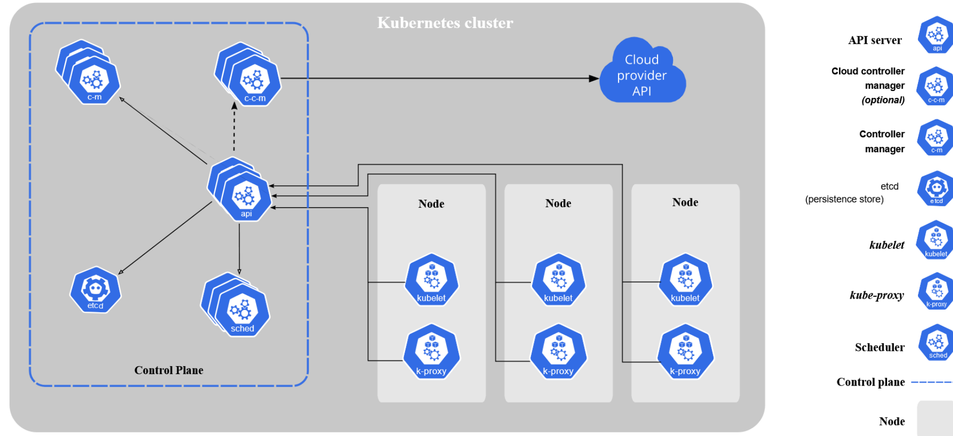

Figure 3.1: Kubernetes Architecture Figure sourced from: [26]

To improve the reliability of the NIMS cluster, the control plane in the cluster is not actually co-located with the rest of the NIMS nodes, and instead is hosted as a VM on Dalhousie's Open Stack cloud. The role of the VM is strictly that of orchestration, and no other workloads are run on it.

Kubernetes allows one to deploy resources on a cluster of nodes. Two common resource types relevant to our work are Deployments and Jobs. Both types ultimately manage 'pods', which are the smallest deployable units of computing on Kubernetes. Pods are collections of containers that share storage and network resources and importantly, are all co-located and co-scheduled on the same node[29].

Deployments are resources which describe a desired state and through the use of a controller maintain that state on the cluster[12]. When we want to host something like a TPG visualization tool we would use a deployment to specify that a container, with the visualization tool's docker image, should always be running on the cluster.

Jobs are similar to deployments, however, they manage pods who are expected to do something and then *terminate*. They can be configured to automatically restart crashed containers up to a certain number of retries, or to keep queuing containers until a certain number complete execution successfully[22]. Training TPG on an environment using an experiment image like the one discussed in section 3.1.1 is an example of how job resources can be used on a k8 cluster.

With our NIMS cluster we are now able to deploy containerized versions of TPG

across our hardware resources in an efficient, automated manner, addressing the second execution challenge.

3.1.3 Infrastructure-as-Code

Infrastructure-as-Code (IaC) is the management of IT infrastructure in a descriptive model, using the same versioning as source code. The same way source code traditionally produces a binary, IaC reliably produces computing environments [37].

Modern cloud providers like Azure, Google Cloud Platform and Amazon Web Services provide some form of IaC solution [7, 37, 18]. However, the NIMS cluster is on premises at Dalhousie, and operated by our lab, so we leveraged a provider agnostic IaC solution called Terraform.

Terraform allows us to create configuration files written in terraform language. These files describe a desired infrastructure state[34] and have a '.tf' file extension. They are similar to resource configuration files that kubernetes uses natively but terraform's configuration language is much more powerful than static configuration files. We will leverage this to generate the command-line parameters for sets of related experiments in an human-error free fashion.

Terraform modules are packages of resources that are used together and can easily be reused in other modules. They consist of `main.tf`, `variables.tf`, `outputs.tf` files. The core syntax of the language as given from the terraform documentation is shown in listing 3.2

Listing 3.2: Terrafrom Language Syntax [34]

```
<BLOCK TYPE> "<BLOCK LABEL>" "<BLOCK LABEL>" {
  # Block body
  <IDENTIFIER> = <EXPRESSION> # Argument
}
```

Once one defines resources in this manner one can use terraform to deploy, update or destroy infrastructure in a reliable, repeaTable, reusable way. Terraform achieves this by manipulating the state of infrastructure resources through a *provider*. In our case the provider is the kubernetes NIMS cluster. If one issues the `terraform plan` command in the working directory of a terraform module, the terraform engine will

parse the configuration files in the module and query the current state of infrastructure resources through the provider. It will then compare the state described in the configuration files against the current state and determine what changes need to take place to bring the current state to the desired state. Finally the `plan` command will output the list of changes terraform intends to execute. Executing the `terraform apply` command actually performs the planned changes.

Below are key excerpts from the resource block describing a pyTPG experiment job as it appears in its `main.tf` file.

Listing 3.3: A pyTPG experiment deployed as a kubernetes job as expressed in terraform language

```
resource "kubernetes_job" "lightbeam_job"{
    count = length(local.tasks)
    metadata{
        name =
        "lightbeam-
        ${lower(local.tasks[count.index][0])}-
        ${lower(local.tasks[count.index][1])}-
        ${lower(local.tasks[count.index][2])}"
    }
    spec {
      template{
        spec{
          container {
           name = "lightbeam-container-${count.index}"
           image_pull_policy = "Always"
           image = "nimslab/tpg-v2:lightbeam"
           command = [
             "python3",
             "/app/nims/experiments/atari.py",
             "--environments", local.tasks[count.index][0],
             "--instance", local.tasks[count.index][2],
```

```
                "--episodes", "5",
                "--end_generation", "500",
                "--run_key", "lightbeam",
                "--frames", "18000",
                "--threads", "5",
                "--initial_team_population", "360",
                "--initial_max_team_size", "5",
                "--initial_max_program_size","128",
                "--gap", "0.5",
                "--input_size", "8400",
                "--register_count", "8",
                "--learner_delete_probability", "0.7",
                "--learner_add_probability", "0.7",
                "--learner_mutate_probability", "0.3",
                "--program_mutate_probability", "0.66",
                "--action_mutate_probability", "0.33",
                "--atomic_action_probability", "0.5",
                "--instruction_mutate_probability", "1.0",
                "--instruction_add_probability", "0.5",
                "--instruction_delete_probability", "0.5",
                "--instruction_swap_probability", "1.0",
                "--elitist",
                "--rampancy_parameters", "1,5,5",
                "--max-no-ops", "0",
                "--checkpoint", "1",
                "--trainer-checkpoint", "25",
                "--path-trace-checkpoint", "50",
                "--traversal", local.tasks[count.index][1]
                ]
}}}}}
```

Note how parameters that stay fixed for all experiments in a set are defined in the `command` identifier. Text inside ${} are evaluated as language expressions. In

listing 3.3 these expressions make references to a `local` block, the contents of this block are shown in listing 3.4 below.

Listing 3.4: Local variables in the main.tf file

```
# Compute the cartesian product of the experiment variables.
tasks = tolist(
    setproduct(
        var.games,
        var.traversal,
        var.instances
    )
)
```

The `games`, `traversal`, `instances` are defined in the `variables.tf` file described in listing 3.5

Listing 3.5: Global variables in the variables.tf file

```
variable "traversal"{
    type = set(string)
    default = ["team", "learner"]
}
variable "games" {
  type = set(string)
  default = [
  "Robotank-v0",
  "Skiing-v0",
  "TimePilot-v0",
  "Tutankham-v0",
  "Venture-v0"
  ]
}
variable "instances" {
  type = set(string)
  default = ["1","2","3","4","5"]
```

```
}
```

Putting it all together, when one applies these configurations to the NIMS cluster, the cartesian product of traversals, games, and instances are computed. Using the default values shown in the listings above that will result in a list of $5 \times 5 \times 2$ (or 50) records that end up stored in the `local.tasks` variable. These records will look something like `Skiing-v0-learner-1, Skiing-v0-learner-2,....` The length of this list is used to determine the number of kubernetes jobs to create `count = length(local.tasks)`. Finally the values for the command-line parameters `--environments`, `--instance`, and `--traversal` are populated from the records in the list for each of the 50 jobs.

In this fashion we address execution challenge three, because terraform is responsible for determining the number of experiments and generating the command-line values. We have eliminated the possibility of human-error when starting experiments and trivialized the task of launching hundreds of experiments efficiently on our cluster.

3.1.4 Automatic Cloud Backups

The last execution challenge deals with restarting runs that were interrupted. PyTPG allows us to encode the current state of the algorithm to a binary format using 'pickle', an object serialization module in python [28]. To leverage this feature, we bind the command line parameters, and other run-time information from the instance to the primary object being serialized. Thus, when we load the object not only is the evolutionary work on the graph preserved, but we also have all the information we need to resume the run (like the last computed generation for instance).

There are two other key issues at play.

1. For ease of use we would like the command that starts a new instance to be the same one that loads an existing instance if one exists.

2. Just because a run was interrupted on one machine, does not mean that it will be restarted on the same machine.

We address both problems using a Microsoft Office 365 Business subscription which allows us 1TB of cloud storage on Onedrive[9]. The Microsoft Graph API

allows us to communicate with our Onedrive storage space programmatically [39]. Finally, use the simple encoding shown in listing 3.6 to uniquely identify an experiment instance.

Listing 3.6: Encodings used to identify instances through the automated cloud backup and restart processes. The 'run digest' is designed to uniquely identify the combinations of parameters we are interested in testing in an experiment set. The 'run key' gives a name to the experiment set, in this case 'lightbeam'. These together combine with the last computed generation to create the name of the zip file containing all the instance data required for a restart.

```
#Lightbeam Run Digest
<environment>-<traversal_type>-<instance>
Asteroids-v0-TEAM-1

<run_key>_<run_digest>
lightbeam_Asteroids-v0-TEAM-1

#Lightbeam Run Instance backup
<run_key>_<run_digest>_<generation>.zip
lightbeam_Asteroids-v0-TEAM-1_125.zip
```

These pieces are put together in the logic of our start script. First, the command line parameters are read and used to determine the `run_key` and `run_digest` for the instance. Then we check the local file system *and* Onedrive for a backup zip file containing the `run_key` and `run_digest` in its name. If one is found on both the local file system and Onedrive, we pick the one with with the highest `generation` value to restart from. If this back up is being sourced from Onedrive, it is downloaded at this stage. We unpack the backup files and load in the saved state generated by 'pickle'. Finally we resume evolution from the last computed generation.

One of our command-line parameters `--trainer-checkpoint` is used to specify at which interval of generations a backup is emitted to Onedrive. From the example in listing 3.3 this interval is 25 generations.

Our instances now regularly emit backups, and are capable of restarting on any internet connected machine by fetching the related backup from Onedrive. Best of

all, because the same command is used to start an instance and to restart it, none of our terraform configurations or other launch scripts need to be adapted. We can simply run the same ones again, and experiments that have already made progress will resume from where they left off +/-25 generations.

3.2 Capturing Results

Once an experiment is launched it produces a steady stream of data. This data must be captured, collected, stored and made available for analysis. With the infrastructure described in the previous section one can easily launch tens or even hundreds of experiments at once, but without an automated system to collect the data produced by these experiments, it can become prohibitively time consuming to analyze results. These analysis challenges were tackled using Looking Glass, a custom data analysis platform.

The entire Looking Glass system is deployed through a terraform module onto the NIMS cluster. In this section we will examine the kind of data that we gathered during the training and testing phases of our experiments, and the configuration of the systems in Looking Glass bring this data together for analysis. There are three main aspects of the Looking Glass platform: Ingestion, Storage, and Query & Visualization. Figure 3.2 provides a high-level schematic of Looking Glass.

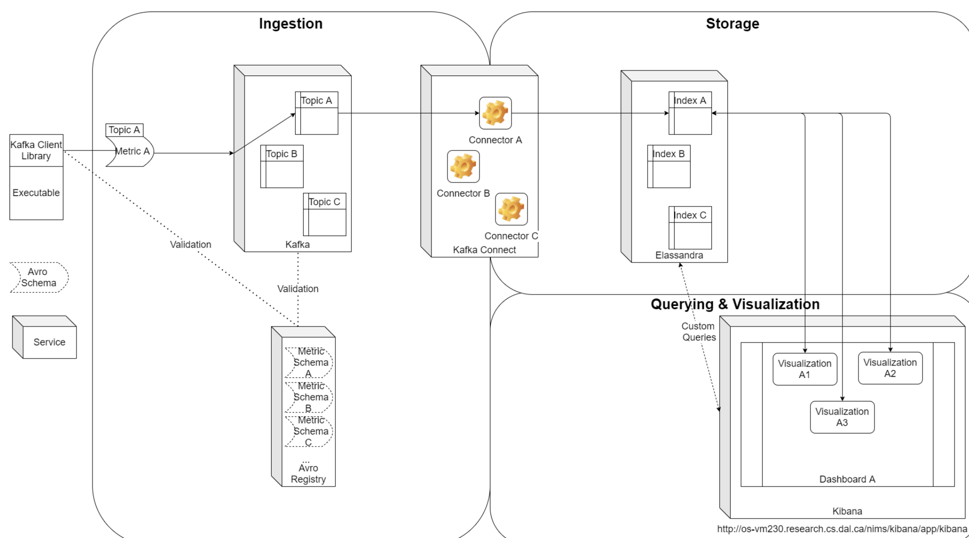

Figure 3.2: Looking Glass Architecture

This breakdown of responsibilities within Looking Glass exists to create as flexible a platform as possible. Ingestion is separated from storage so that once a TPG implementation has the capability to emit metrics, it will not have to be adapted if the storage solution is changed, or if there is some desire to send that same data to other services downstream. Similarly, querying and visualization is encapsulated away so that the raw data is accessible to any client that requires it. If one is not satisfied with the querying and visualization capabilities that come with Looking Glass, one can easily write their own solution and fetch the raw data via a web API.

The following popular open source software powers Looking Glass: Ingestion is handled by Kafka and Avro, storage is provided by Elassandra (a packaged configuration of Elasticsearch with Cassandra [14]), and Querying & Visualization is provided by Kibana. Kafka is an open-source distributed event streaming platform[5], Avro is a data serialization system[3], Elassandra is a distributed No-SQL database[14], and Kibana is a web app that allows users to interact with Elassandra by creating visualizations and running queries[25].

3.2.1 Metrics

Before one can analyze anything, one must decide what to analyze. In this work, metrics refer to groups of properties whose values are of interest for analysis. We used five key metrics to capture data from the training and test phases of our experiments. Each metric is emitted at a particular point of execution and described in its entirety by an Avro schema.

Avro is a data serialization system developed and maintained by the Apache foundation. Using an avro schema we can encode our metrics in a compact and fast binary format suiTable for transmission. Avro supports dynamic typing which allows us to mark fields as optional, or with multiple types[3]. This allows the reuse of schemas between implementations of TPG where appropriate.

Ultimately, these metrics will land in Cassandra, a wide-column store capable of backing the dynamically typed values described in our schemas[38, 4]. The data there will then be queried by Elasticsearch. We will go over this in more detail later, but from a metrics design perspective it is important to note that performing SQL style joins using Elasticsearch is expensive[23]. Instead the 'elastic' way is to

denormalize the data ensuring all fields of interest for a query exist within a metric. Here denormalizing means that the same fields will appear in multiple metrics, rather than organizing commonly used fields into a record and using foreign keys to reference them in other records. While this does incur a higher storage requirement, disk space is often far more available than computing power[40]. Listing 3.7 shows a small sample of the avro schema describig the generation metrics. Fields like `run_id`, `run_key`, and `run_digest` will appear in all metrics as a result of the aforementioned denormalization.

Listing 3.7: A small sample of the Generation Metric avro schema

```
{
    "namespace":"lookingglass.tpg",
    "type":"record",
    "name":"GenerationMetric",
    "fields":[
        {"name":"run_id","type":"string"},
        {"name":"run_key", "type":"string"},
        {"name":"run_digest", "type":"string"},
        {"name":"date", "type": ["string","null"]},
        {"name":"generation", "type":"int"}
    ]
}
```

Generation Metric

The generation metric is emitted once per generation of training, after all root teams have had played the required number of games (episodes) in the environment. 157 fields appear within this metric; for brevity they will be discussed at a high level. The purpose of this metric is primarily to feed our fitness curve visualizations, as such they contain sufficient information to uniquely identify the experiment, the generation they represent, and the fitness achieved across the root team population.

In addition we also collect information about the types of instructions the programs in the population have, and a broad range of information regarding the parameterization of TPG used to produce the metrics. This includes the actual algorithm parameters but also things like the programming language and version of the implementation.

Graph Metric

The graph metric is emitted every 25 generations of training. In testing this metric is emitted once per champion. While containing only 34 fields, it actually is our largest metric in terms of individual record size, as it contains the full population graph created by TPG in training, and full champion graph in testing. Our 'Inside the box' visualization tool notably uses these metrics to render the graphs for visual inspection and analysis, see figure 3.3.

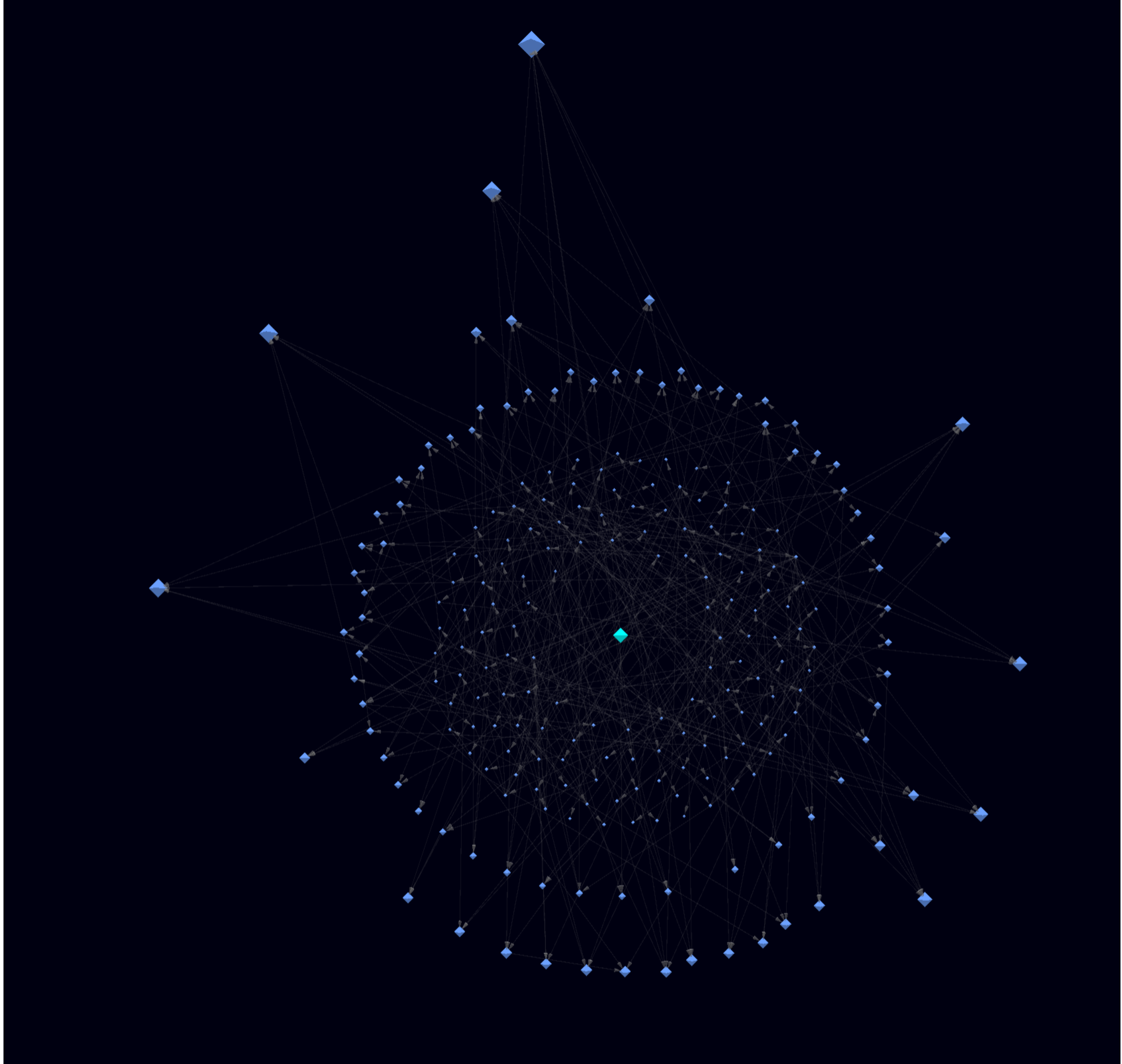

Figure 3.3: The champion for Centipede using team traversal after 500 generations of training. The root team is in teal, while other teams appear in light blue. Arrows represent learners. The teams are arranged radially where their size and distance from the root team is determined by the number of incoming learners.

Path Metric

Closely related to the graph metric, the path metric is emitted both in training and testing. In training, after a generation completes, the champion root team is pulled out and set to play one additional round on the environment. During this special round, we capture the path(s) the champion traverses through its graph to produce every action it applies to the game environment. These can amount to thousands of records per round as such, during training, this is only done every 25 generations.

However this 'path tracing' is done for all 20 test episodes the final champions are put through during the testing phase. Our 'Inside the box' visualization tool allows us to render what such a path looks like, see 3.4.

Team Metric

The team metric is emitted at the end of every generation of training. It contains the fitness of every root team in the population. In addition, each team metric also reports the number of teams and learners, in the root team it represents. This information can be used to generate graphs like the one in 3.5.

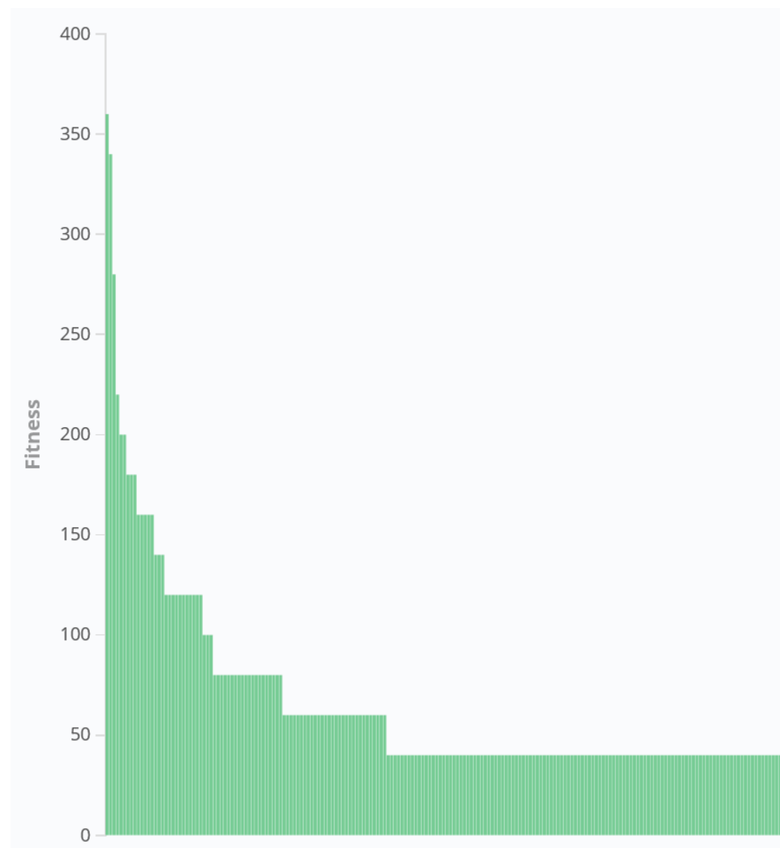

Figure 3.5: The fitness of each root team in from a TPG population playing Venture using learner traversal at generation 500.

Episode Metric

The episode metric contains 96 fields measuring the performance of champions during the test phase. One episode metric is emitted for every game (episode) played. Here

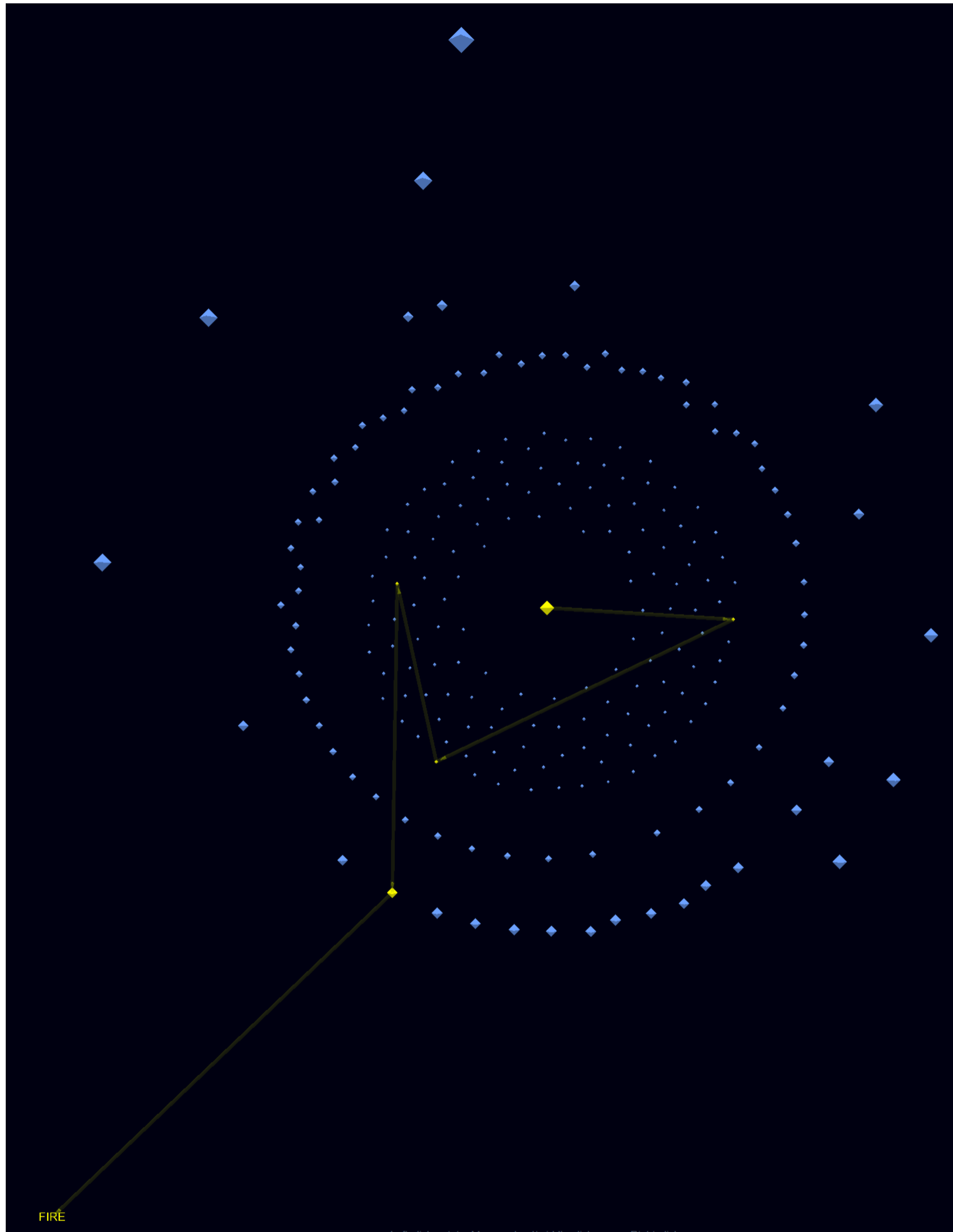

Figure 3.4: A path the champion from figure 3.3 used in making the decision to apply the 'FIRE' action to the game environment.

we capture the id of the graph metric associated with the champion, as well as static graph statistics like the number of teams, learners, and the diameter of the graph relative to the root node. We also record dynamic statistics that vary depending on the paths traversed while playing the game during the episode. These include things like the mean number of executed instructions per team by type of instruction (add, sub, mult, div, neg). These metrics also include the fitness statistics used to assess the performance a champion during the evaluation phase.

3.2.2 Ingestion

Ingestion is handled by Apache Kafka, an open-source distributed event streaming platform popular in the enterprise space [5]. We use Kafka to bring to life a publish-subscribe messaging model, in which execuTables of interest like implementations of TPG, are *producers* of the previously described metrics.

These metrics will be sent to a Kafka broker, who manages events by splitting them into *topics*. Topics can be thought of as addresses, and in our case we create a topic for each kind of metric we would like to emit[55].

On the other side of the equation we use Kafka Connect, a tool to reliably stream data between Kafka and Elassandra . This tool runs as a web service and allows us to dynamically create *connectors* which act as *consumers* for our topics [24]. Figure 3.6 shows the end to end ingestion flow in Looking Glass.

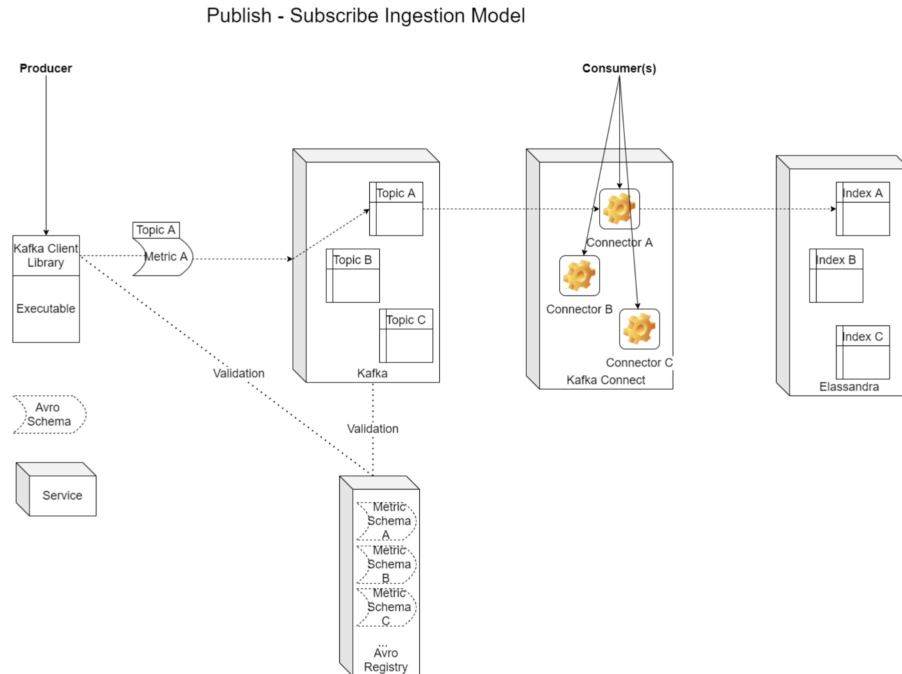

Figure 3.6: The publish-subscribe ingestion pipeline in Looking Glass. The dashed arrows show the path a message containing Metric A takes through the pipeline. Note the association with Topic A in the Kafka broker and Index A in Elassandra.

The Avro Registry, another component provided by Confluent, works together with the Confluent Python Kafka client library [10] and the broker to resolve the schemas of the messages to be sent between them. The registry is a powerful tool which we under-utilize at this point in time. It allows for fine-grained control over schema updates. One can introduce changes to a schema and the registry will start building a history for it. This history can then be used by downstream consumers to become aware of the changes and adapt accordingly. Through this mechanism, consumers expecting the old schema may attempt to translate between versions minimizing the impact a change has on the overall pipeline[33]. This is a great place for further development to improve the overall robustness of the Looking Glass system.

On a final note, Kafka is meant to handle large quantities of small messages [55]. Our graph metrics, after some evolution would pass the default 1MB message size limit imposed by Kafka. This was reconfigured to allow messages up to 25MB in order to accommodate our needs.

3.2.3 Storage

Our kafka connectors insert data into Elasticsearch, a distributed, RESTful search and analytics engine that allows us to work with the mountains of data our experiments produce[15]. In Looking Glass, Elasticsearch is provided by Elassandra, which is an open-source packaging of Elasticsearch with Cassandra [14]. As briefly alluded to earlier, Cassandra is a distributed wide-column store [4]. This makes it the ideal candidate to persist the flexible metrics defined in our avro schemas, as wide column stores allow you to store values of different types on different rows of the same namespace (Cassandra's versions of Tables)[38]. In addition Cassandra is highly scalable exhibiting linear increases in throughput when additional nodes work together in a cluster [59].

During the development of this work, power failures and other various technical problems would result in down time for some of the servers in the NIMS cluster. To improve the reliability of the system, Looking Glass initializes three Elassandra nodes in a cluster configuration for the storage solution. This cluster was then manually configured to use a replication factor of at least two for all metric indices. In this way, the failure of any single node would not affect the integrity our results. In addition, queries made against the cluster would have their computational cost shared between cluster nodes, allowing for near real-time analysis of vast amounts of data.

3.2.4 Querying & Visualization

Kibana was our primary analysis tool for the experiments described in this thesis. Kibana is a free open-source user interface that allows you to visualize data from Elasticsearch [25]. The metrics described previously would all make their way into a corresponding Elasticsearch index. For the 'lightbeam' experiment, the following were the primary indexes of interest:

- `tpg.lightbeam.metrics.path`
- `tpg.lightbeam.metrics.generation`
- `tpg.lightbeam.metrics.graph`
- `tpg.lightbeam.metrics.test.episodes`

- `tpg.lightbeam.metrics.team`

Kibana allows us to view these metrics as they come in through its 'Discover' view, see Figure 3.7.

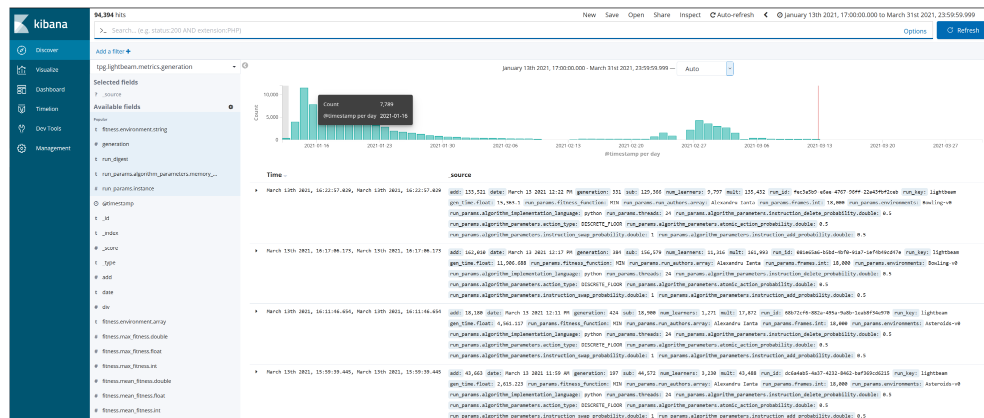

Figure 3.7: Viewing metrics in the `tpg.lightbeam.metrics.generation` index, automatically refreshed every five minutes by Kibana. The histograms along the top show the volume of metrics coming for a given day; individual records appear underneath.

Amongst other things, Kibana also allows us to create live visualizations of the data. These visualizations can then be grouped together to form dashboards that provide us with a birds eye view of our experimental results, see Figure 3.8.

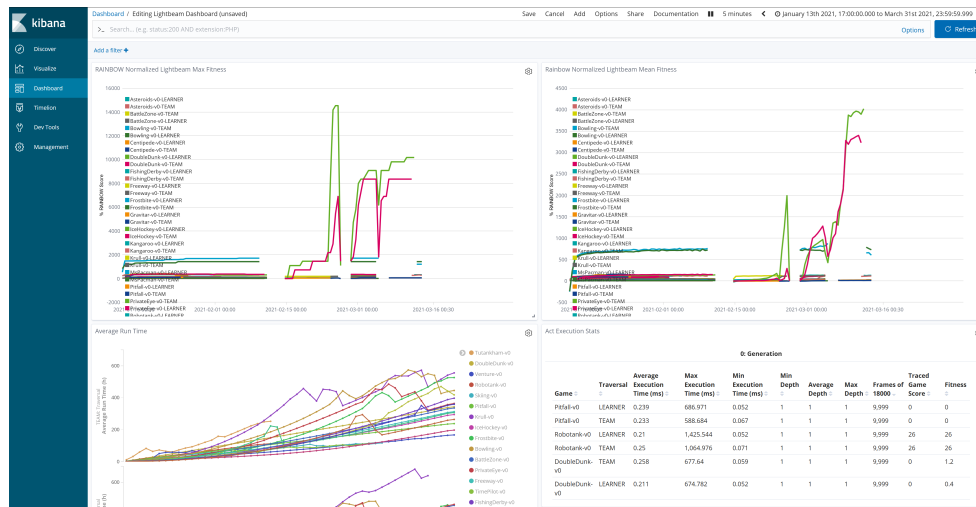

Figure 3.8: The 'Lightbeam' experiment dashboard. The top row charts, from left to right, are the RAINBOW Normalized Maximum Fitness, and the RAINBOW Normalized Average Fitness achieved by TPG, plotted against time. The bottom left graph, shows the average runtime (in hours) vs the # of generations computed averaged across instances of a particular environment. The bottom right Table shows various statistics about the paths traversed to produce actions by champion teams at 50 generation increments during training.

It is easy to filter or sort the data behind any visualization by any valid field described in our metric's schema (environment, traversal type, fitness score, generation, etc.). Many of the figures found in the results section of this work were produced using Kibana visualizations. Kibana also allows users to easily download the raw data backing the visualization in .CSV format for further analysis in other tools, or for insertion into papers[19].

Like many other aspects of the Looking Glass platform, Kibana is a powerful tool, currently under-utilized. Kibana offers a rich set of features, backed by Elasticsearch that are worth diving into through future work. For example, one can leverage a machine learning pack to automatically analyze metrics as the come in and provide anomaly detection that can help focus analysis work[27].

It is important to note however, that Kibana is not the only point of access for our experimental results. Kibana works as a client for the Elasticsearch web API. Should we want to use a different visualization tool, or perform analysis not possible through Kibana directly, we can simply use Elasticsearch's API directly. This API allows us to query, filter, and sort our data programmatically so we may retrieve what we are

interested in and do with it whatever we please. Figures 3.3 and 3.4 are produced using Three.js visualizations [11] in a Vue webapp[30], served by Node.js[1]. This app, dubbed 'inside-the-box', sends an Elasticsearch query via HTTP to retrieve the data it needs to produce its visualizations.

Listing 3.8: The query used by 'inside-the-box' to retrieve the data shown in Listing 3.9. The variable `graph_id` is populated through a user interface.

```
"query":{
    "match":{
        "graph_id.keyword": graph_id
    }
}
```

Listing 3.9: The response provided by Elasticsearch's Search API for the query in Listing 3.8. The `graph_data` and `visualization_links.string` fields have been truncated for brevity. The shards section refers to the 3 Elassandra nodes deployed to provide reliability and improve query throughput.


```
{
"took": 20,
"timed_out": false,
"_shards": {
    "total": 3,
    "successful": 3,
    "skipped": 0,
    "failed": 0
},
"hits": {
    "total": 1,
    "max_score": 7.0387836,
    "hits": [{
      "_index": "tpg.lightbeam.metrics.graph",
      "_type": "_doc",
```

```
      "_id": "H0bd99b56-c9c5-40d7-b3fd-5ca5584654df",
      "_score": 7.0387836,
      "_source": {
        "generation": 500,
        "num_learners": 1004,
        "run_id": "9c549e2e-6ac6-459d-8b30-d64c325ec096",
        "run_key": "lightbeam",
        "edges": 1218,
        "graph_data": "{\"nodes\": ,..."
        "graph_id": "8554d3da-8db3-4d07-a784-ae008cc8e3a1",
        "num_actions": 18,
        "run_digest": "Centipede-v0-TEAM-2",
        "environment": "Centipede-v0",
        "@timestamp": "2021-02-01T21:15:06.676Z",
        "num_teams": 225,
        "visualization_links": {
          "string": ".../nims/tpg3d/#/.../8554d3da..."
        },
        "traversal_type": "TEAM"
        }
    }
]}}
```


3.3 Summary

Docker allows us to package our algorithm in a porTable, easy to execute way. Kubernetes then allows us to orchestrate the execution of docker containers across our available compute resources in an efficient, automated manner. Terraform removes the opportunity for human error by computing the combinations of starting parameters needed to explore the the parameter space we are interested in, and seamlessly passes the resulting jobs to Kubernetes. Finally, should an error, outage, or other disruption occur, our integration with Onedrive allows us to easily resume experiments

on any machine or compute node available to us. Together these solve the execution challenges posed at the start of this chapter.

With Looking Glass deployed on the NIMS cluster, we more than address all of the analysis challenges we set out to tackle at the beginning of this chapter. Through the ingestion pipeline, TPG implementations emit their metrics over the internet to Looking Glass and save us from the trouble of having to retrieve CSV files from remote servers. Kibana provides powerful tools for visualizing and analyzing the metrics sent to Looking Glass right out of the box. Should that not be enough, Elasticsearch exposes all of our data through its powerful query APIs which can easily be used by any external program to retrieve relevant portions of our data sets and perform additional analysis.

Best of all, Looking Glass is easily deployable on any kubernetes cluster because all the key configurations connecting these various tools are built into a terraform module available on Github[1]. This allows other researchers to save themselves the setup work we have already done, and bring a powerful research environment online for their own needs as easily as `terraform apply`.

[1] https://github.com/aianta/Looking-Glass

Chapter 4

Results

From January to March 2021, 100,024 generations were computed, 3,214 champion graphs were saved, 36 million root teams were captured during training, and 62 million paths were traced during evaluation phases at various generations. During this time, 137 of the planned 200 experiment instances reached the 500 target generations of training. These will be the primary focus of our investigation, though evaluation data for generations 300 and 400 showing similar trends to those discussed here can be found in Appendix A.

Throughout this section we will refer to specific experiment instances by their run digests (introduced in section 3.1.4). The digest takes the form *environment-traversal-instance*, so instance 3 of the krull environment using learner traversal would be Krull-LEARNER-3.

4.1 Training

The training phase began on January 13th 2021 when the 200 experiments were queued on ACENET. Over the next three months training would continue towards producing the 500 generation champions for the evaluation phase. Figure 4.1 shows the average fitness curves normalized to final RAINBOW score (Eqn. (2.1) as they were monitored through Kibana. Note that these normalized charts were plotted against time (not generations) and were primarily used to gauge the status of the instances. Gaps in the curves of figure 4.1 happen due to a lack of instances reporting generation results at those periods in time. These gaps occurred primarily due to increases in the time required to compute a single generation, but less commonly also due to errors arising from hardware constraints (lack of disk space or memory), or computational time limits being reached (maximum of 28 days on ACENET).

Due to these 'outages' subsets of the 200 experiments would be restarted leveraging the mechanisms described in chapter 3. While these systems streamlined the restart

process, generations would be lost during a restart as backups could not be feasibly taken for all instances at every generation. Since ACENET prioritizes users underutilizing resources, each restart would increase the time spent in queue by a job.

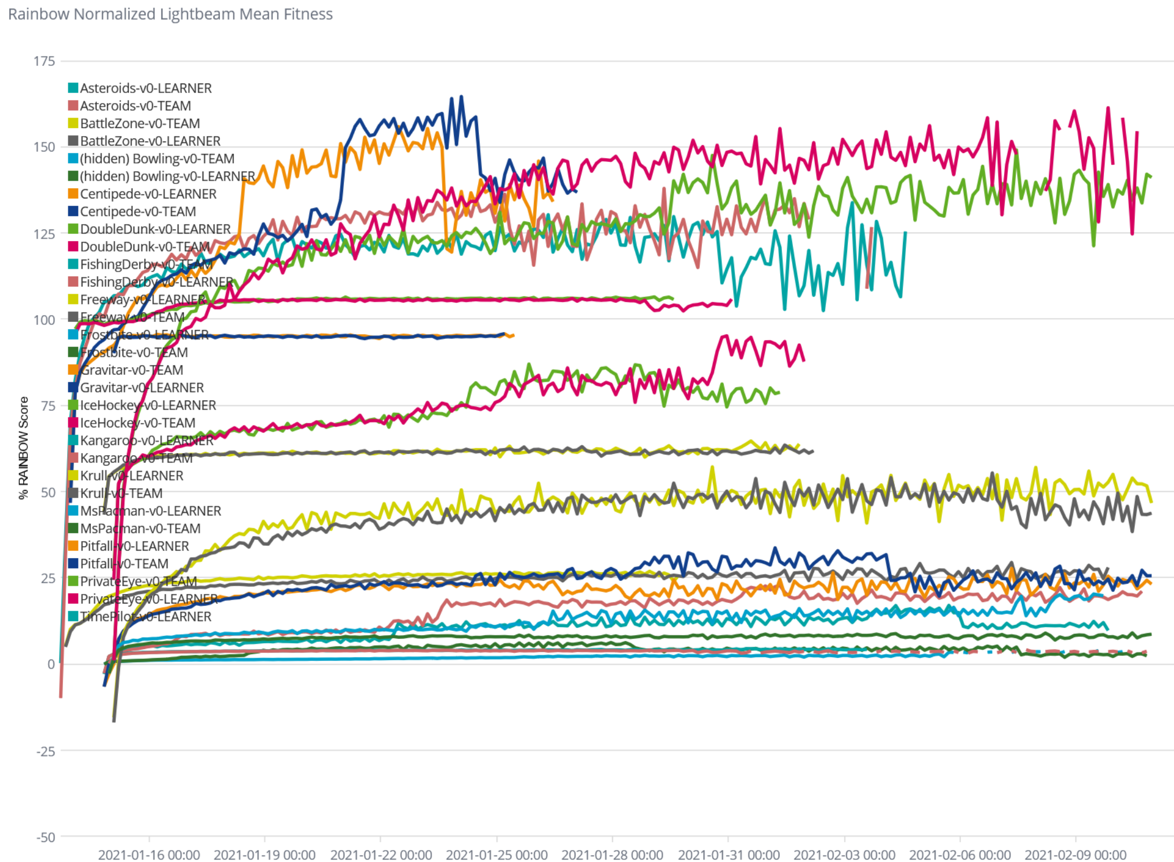

Figure 4.1: Average normalized fitness being monitored after runs begin on January 13th 2021.

Many environments across both traversal strategies experienced dramatic increases in compute time per generation as shown in figure 4.2. Games like Krull and Tutankham in particular topped 10,000 seconds or 2.7 hours of computation per generation. This discrepancy was likely driven by the fact that many other game's episodes did not play their full 10,000 frames where as they did.

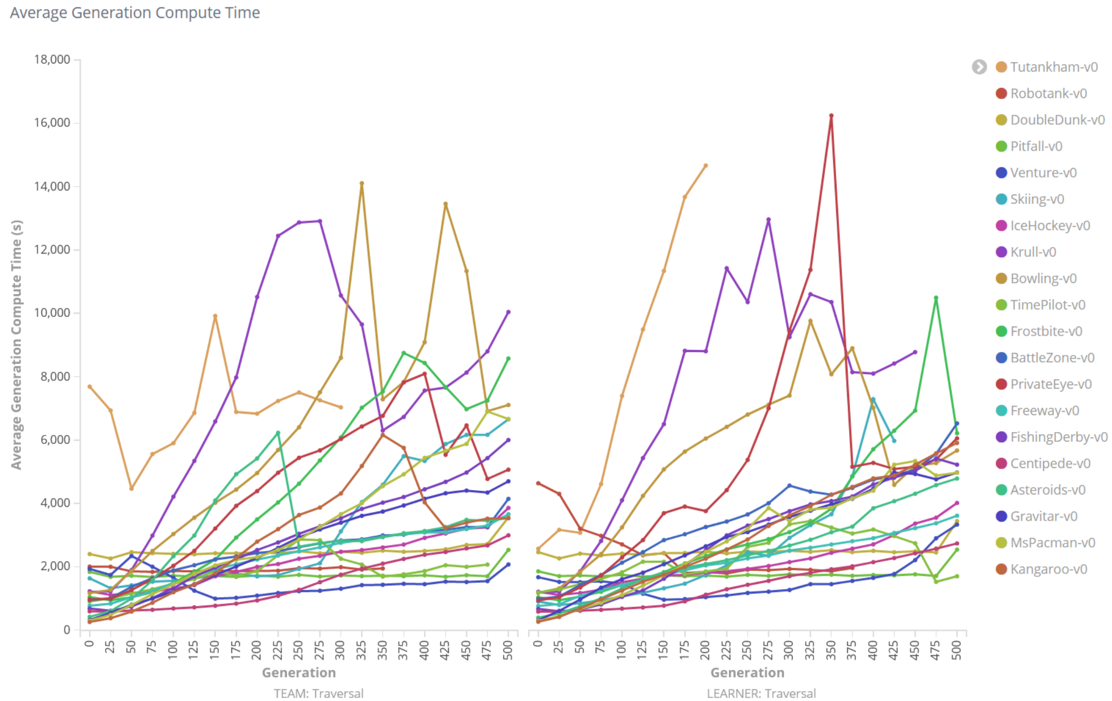

Figure 4.2: The number of seconds a generation would take to compute for a given environment averaged over all instances.

Interestingly, in other cases these compute time trends were not always shared across all instances of a particular environment. Notably, Asteroids-TEAM-4 and Bowling-TEAM-4 reached 10,311s/gen and 9,468s/gen respectively, while their peers averaged 2,000s/gen and 5,000s/gen respectively. This divergence may be partially explained by the explosive growth in the underlying graph for those instances, figures 4.3 and 4.4 show the number of teams and learners split on TEAM traversal instances of Bowling and Asteroids respectively. Note that any apparent drop offs in graph size shown in figures 4.3 and 4.4 like generations 300+ for Asteroids-TEAM-4, 350+ for Asteroids-TEAM-6, 400+ for Bowling-TEAM-4, etc. are due to the experiment instances not reaching those generations and hence no data being available. Particularly slow experiment instances like Asteroids-TEAM-4 and Bowling-TEAM-4 were ultimately cancelled to conserve computational resources and improve the queue times for other instances including instances which were to replace them.

Nevertheless all instances were given hundreds of hours of compute time producing ample data points for further analysis. Figure 4.5 shows the average cumulative number of hours required by an environment to reach a given generation on the

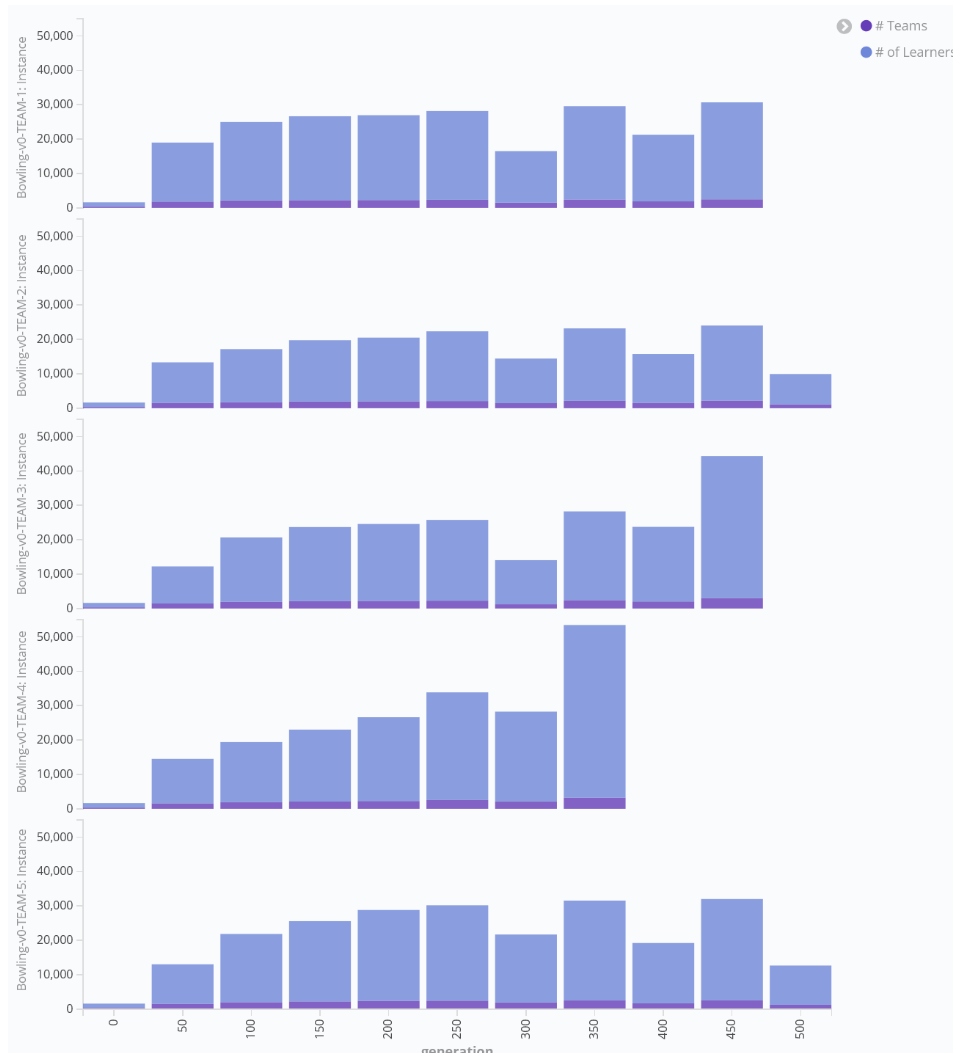

Figure 4.3: The number of teams in *purple* and learners in *blue* across team traversal instances for the Bowling environment plotted against the generation when the graphs were sampled on the x-axis. Bowling-TEAM-4, the fourth entry, reaches a staggering 50,184 learners and 3,250 teams compared to the 25,000 learners and 2,500 teams of its peers at the same generation. Note, the apparent drop off in size immediately afterwards for Bowling-TEAM-4 happens because the instance was stopped to conserve computational resources.

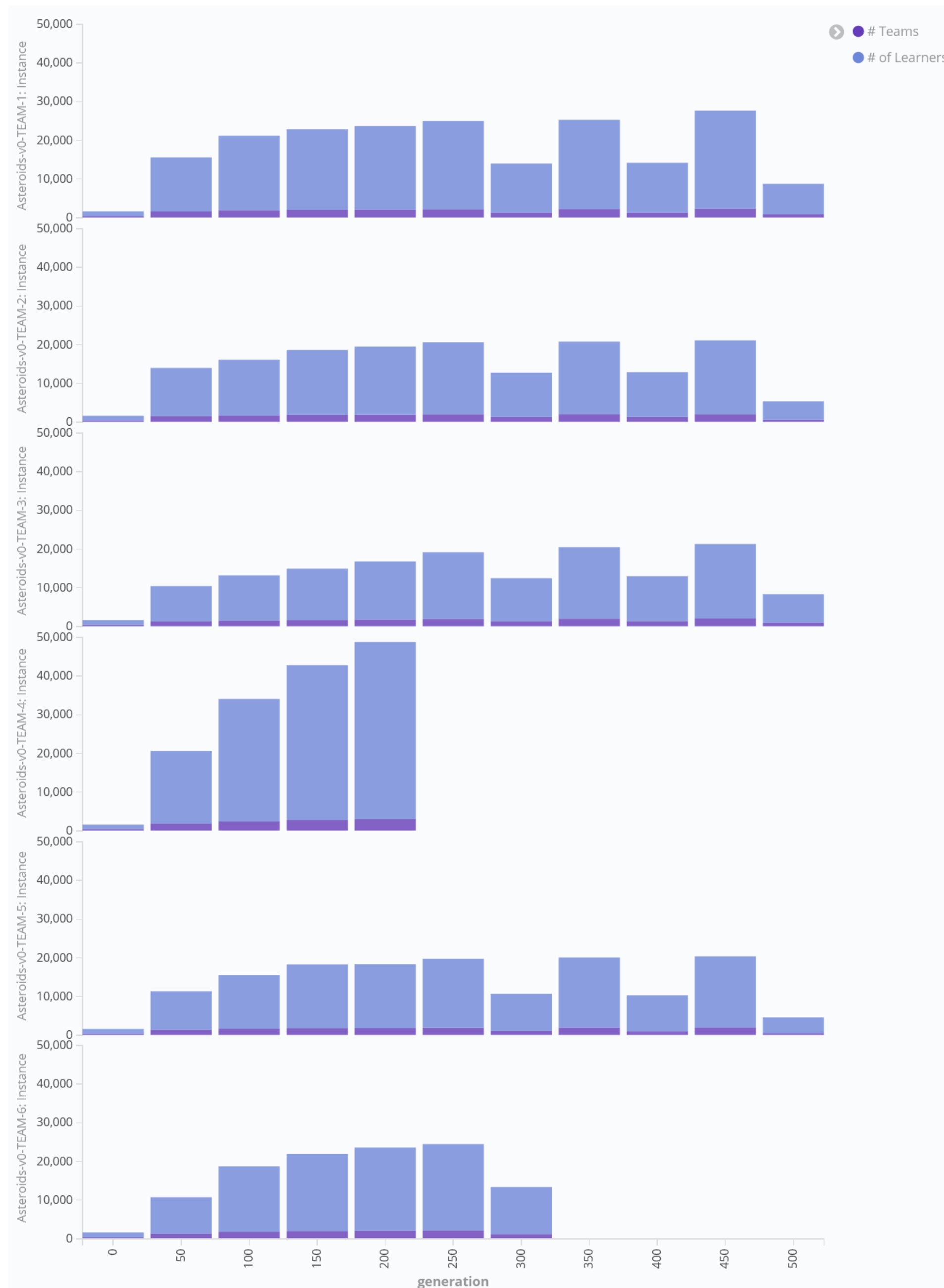

Figure 4.4: The number of teams in *purple* and learners in *blue* across team traversal instances for the Asteroids environment plotted against the generation when the graphs were sampled on the x-axis. Asteroids-TEAM-4, the fourth entry, reaches 45,830 learners and 3,004 teams compared to the 20,000 learners and 2,000 teams of its peers at the same generation. The 'extra' 6th Asteroids-TEAM instance was started in an attempt to have 5 instances at generation 500 for this environment and traversal type after it became clear Asteroids-TEAM-4 was not going to finish in time.

x-axis.

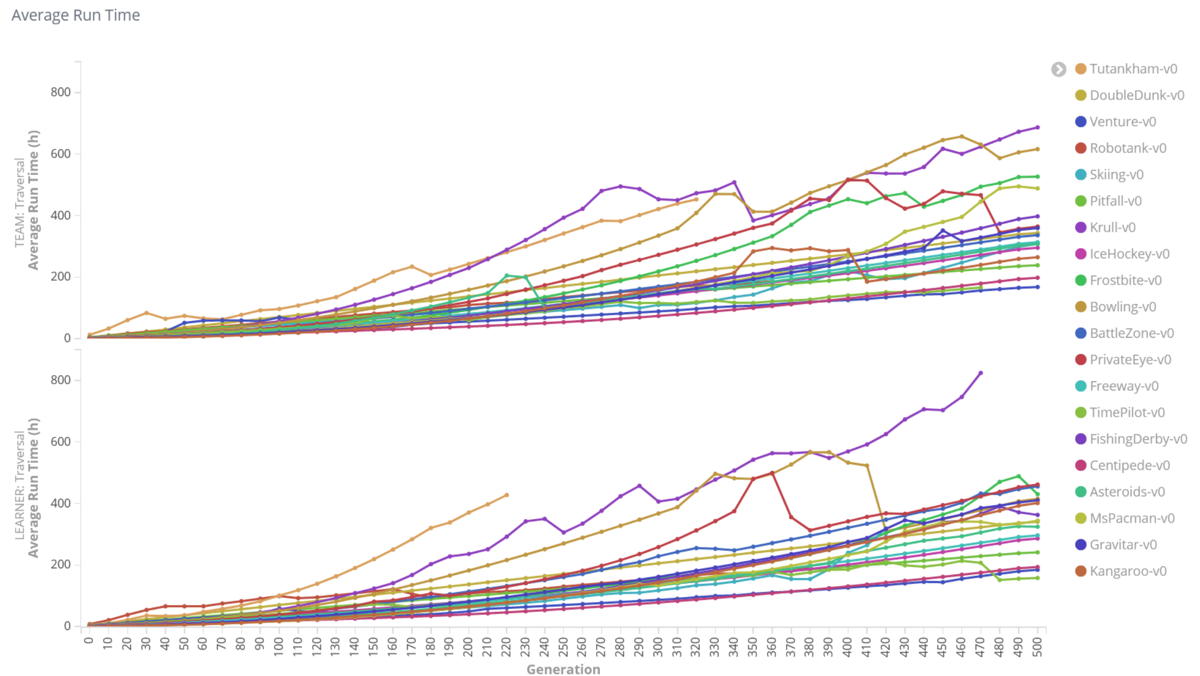

Figure 4.5: The cumulative number of compute hours averaged over all instances of an environment plotted against generations.

Typical TPG fitness curves were observed for the majority of environments with maximum fitness rising rapidly in the first 100-200 generations before succumbing to a trend of periodic step-like increases into generation 500 (see figure 4.6). While one traversal strategy appeared to achieve higher maximum fitness than the other in some environments, no clear trend established itself as the reverse would occur in other environments.

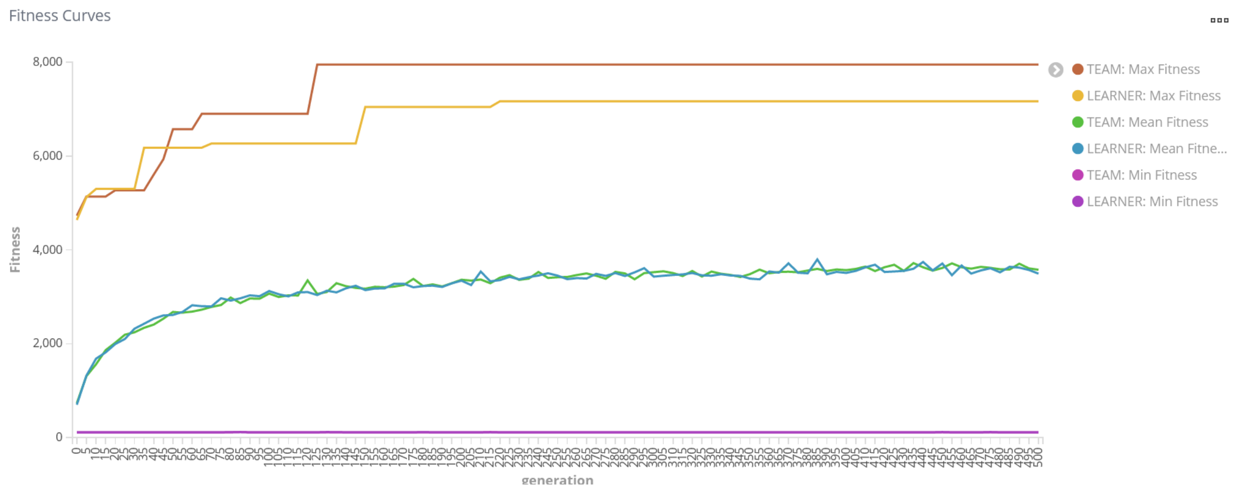

Figure 4.6: Fitness curves for the Asteroids environment during training, averaged across all instances and split by traversal strategy.

Intriguing outliers did appear through the training phase. For example the MsPacman instances seemed to show markedly better performance under learner traversal than team traversal as shown by the discrepancy in average fitness seen in figure 4.7.

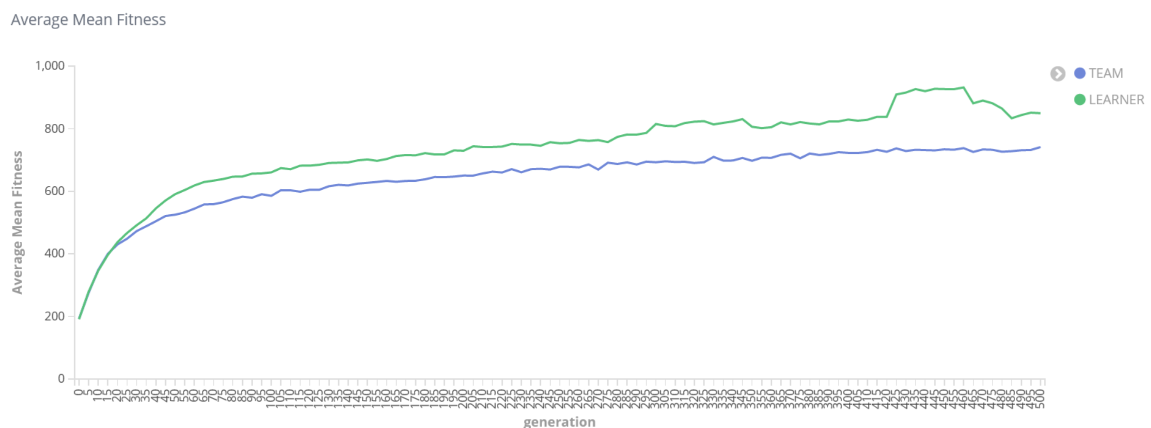

Figure 4.7: Average fitness for the MsPacman environment averaged across all instances and split by traversal strategy.

Further investigation by splitting average fitness on run digests instead of traversals in Kibana (see figure 4.8) found this phenomenon to be driven by one learner instance (MsPacman-LEARNER-4) in particular who seemed to have 'seen something' other instances failed to discover about the game. One would expect this performance not to be reproducible. Nevertheless, one cannot help but wonder if

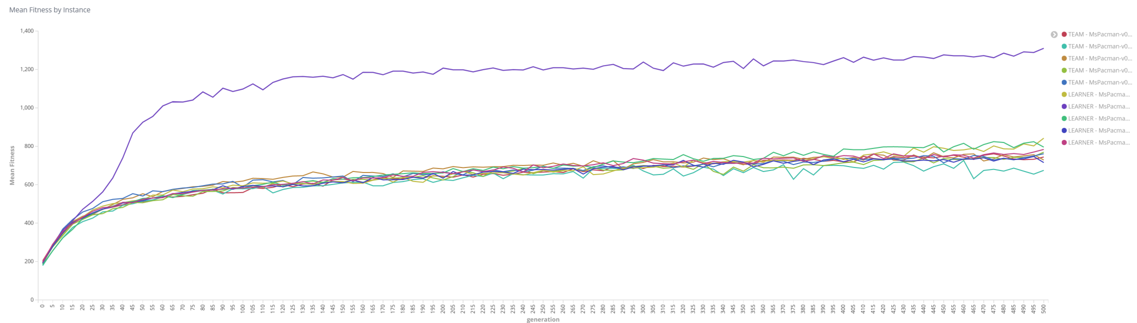

Figure 4.8: Average fitness for the MsPacman environment for each experiment instance. The diverging purple curve is that of MsPacman-LEARNER-4.

a deeper dive into the tangled program graphs of outliers such as these could yield interesting insights about the particular strategies that set these champions apart.

4.2 Evaluation

The 130 balanced sample of champions that completed 500 generations of training were each put through 20 episodes to evaluate their performance, static and dynamic properties. Tables 4.2 and 4.3 are derived from the resulting 2,600 episode records. Balanced samples here mean that an equal number of champions were tested for team and learner traversals. So if 3 team traversal and 5 learner traversal champions were available for an environment at 500 generations, only 6 of the 8 champions would be tested (3 team and 3 learner), so that the action distribution metrics would not be skewed.

Generation	Champions Available	Balanced Champion Samples
300	178	172
400	154	148
500	137	130

Table 4.1: Champions Available refers to the number of champions that achieved the respective generations of training. Balanced Champion Samples refer to the the portion of the available champions which were used to produce the results. This filtering is done because some metrics of interest are sensitive to the balance of team and learner traversal samples. For example, when considering the frequency of some action for a given game, if we compare 3 instances of team traversal with 1 instance of learner traversal (as that is what we have available) we will skew the data. So instead, in that situation, we would compare only a single instance of team traversal to the learner traversal.

4.2.1 Performance

A two tailed, pairwise, t-test on the 2,600 episode fitness values gave 0.352717521, failing to establish significant difference between the two traversals. For Table 4.2 the *Max* value refers to the highest achieved fitness across the 20 episodes, the *Avg.* refers to the mean fitness across the 20 episodes, and similarly the *Min* refers to the minimum across the 20 episodes. The number of instances refers to the number of champions tested for each traversal type, so for the Asteroids-v0 environment these results came from testing 4 champions trained using learner traversal and 4 champions trained using team traversal.

The normalized values are in comparison to RAINBOW, as given by equation 2.1 where 100% refers to matching the score achieved by RAINBOW and anything less (more) than 100 implies that the score from Rainbow is proportionally better (worse) than TPG. One thing to keep in mind when considering the comparison to RAINBOW is that TPG lacks certain advantages that RAINBOW has in this particular set of tasks. One such advantage in RAINBOW's favor is the practice of 'frame-stacking' where multiple frames from the game are superimposed onto a single input vector for RAINBOW to process. This results in 'smears' on objects in the game that are in motion, giving RAINBOW additional information about movement that TPG lacks[41]. Thus TPG's comparable and in some cases better performance on certain titles like Bowling or Venture are particularly remarkable.

Env.	Trav.	Fitness					Inst.
		Max	N. Max	Mean	N. Mean	Min	
Asteroids	L	8010	365.7%	3345.63 ±1361.5	131.7%	1180	4
	T	8610	395.8%	3258.9 ±1492.1	127.4%	1030	
BattleZone	T	67000	108.4%	18325 ±10068.24	26.8%	2000	4
	L	43000	68.1%	17400 ±8678.42	25.2%	1000	
Bowling	T	119	1389.9%	86.925 ±12.44	925.0%	68	2
	L	109	1244.9%	82.6 ±12.40	862.3%	63	
Centipede	T	29127	444.9%	6234.4 ±4231.87	68.2%	1870	5
	L	19280	282.9%	5573.59 ±3206.53	57.3%	1461	
DoubleD.	T	2	112.6%	-1.54 ±1.01	93.2%	-2	5
	L	2	112.6%	-4.22 ±6.78	78.6%	-24	
FishingD.	L	-31	49.3%	-72.44 ±18.44	15.7%	-99	5
	T	-41	41.2%	-81.75 ±14.38	8.1%	-99	
Freeway	L	28	82.4%	22.9 ±1.80	67.4%	18	5
	T	28	82.4%	22.51 ±1.79	66.2%	19	
Frostbite	T	2580	26.4%	710.625 ±647.49	6.8%	90	4
	L	260	2.0%	174.75 ±33.35	1.2%	80	
Gravitar	T	2300	170.7%	485 ±334.33	25.0%	0	5
	L	2000	146.6%	412.5 ±337.96	19.2%	0	
IceHockey	L	5	131.7%	-0.36 ±2.70	88.1%	-9	5
	T	8	156.1%	-1.32 ±4.51	80.3%	-13	
Kangaroo	L	1200	7.9%	685 ±260.34	4.3%	0	4
	T	1200	7.9%	680 ±237.91	4.3%	200	
MsPacman	L	3250	58.0%	848.8 ±562.58	10.7%	80	5
	T	1770	28.8%	522.5 ±327.55	4.2%	130	
Pitfall	L	0	99.9%	0 ±0	99.9%	0	5
	T	0	99.9%	0 ±0	99.9%	0	
PrivateEye	L	15000	355.8%	5159.6 ±5332.69	122.0%	-1000	2
	T	4100	96.8%	2919.6 ±2083.64	68.8%	-1000	
Venture	L	800	14545.5%	89 ±164.86	1618.2%	0	5
	T	600	10909.1%	45 ±112.58	818.2%	0	

Table 4.2: Minimum, mean, standard deviation, normalized mean, maximum and normalized maximum fitness by environment and traversal for champions after 500 generations of training. The instance column refers to the number of champions tested for each traversal type.

4.2.2 Static Properties

In Table 4.3 columns: *# Learners*, *# Teams*, *# Learners/Team*, *# Inst./Team*, and *Diameter w.r.t. Root* are all means computed over the champions for that given environment and traversal type. Diameter with respect to root refers to the 'longest shortest path' to any vertex from the vertex representing the root team [17]. No

significant difference between traversals is given by a two-tailed, pairwise t-test for any column.

The graph diameter values shown correlate with the size of the respective graph. Large diameters like those reported on BattleZone (up to 20) occur in graphs with 5000 learners and 775 teams, whereas on the other extreme we have environments like Pitfall with only a single team and a diameter of 1. This makes intuitive sense as one can expect longer shortest-paths in larger complex graphs.

Env.	Trav.	# Learners	# Teams	# Learners/Team	# Inst./Team	# of Inst.	Diameter w.r.t Root
Asteroids	L	3667.5	542.25	6.66	65.83	242987.75	13.5
	T	2004.25	364.5	6.252	65.974	132773	13.5
BattleZone	T	5211.5	817.5	7.866	63.045	328690.5	20
	L	4679.5	749.75	7.632	61.525	285345	18.25
Bowling	T	5391	840.5	8.069	64.287	345743	18.5
	L	6420.5	861.5	8.233	66.124	422960.5	19.5
Centipede	T	2741.4	463.6	6.388	65.761	175936.2	13.6
	L	3396.2	557.4	6.888	63.365	213528.6	15.8
DoubleDunk	T	404	82.6	5.337	64.036	26448.4	8
	L	886.8	173.2	5.463	62.977	56374.8	9.8
FishingDerby	L	6525.4	919.4	8.49	67.534	433939.6	20.8
	T	6188	879	8.321	67.13	407725.6	20.2
Freeway	L	3343.2	535.6	6.75	65.299	218128.8	14.8
	T	3727.4	644.6	7.212	65.525	243660.8	17.4
Frostbite	T	4909	726	7.421	63.57	307231	19.75
	L	4477	718	7.326	65.199	291820.75	17
Gravitar	T	7139.4	958	8.892	68.45	498740.6	19.2
	L	5403.6	739.2	8.088	68.317	371276.4	16.2
IceHockey	L	4128	681.6	7.315	64.916	268427.2	16.8
	T	2757.2	503	6.48	63.586	172627.6	13.4
Kangaroo	L	4897.25	754.75	7.869	66.719	326181.5	17.25
	T	1708.75	328.75	6.156	65.959	111044.75	11.5
MsPacman	L	4077	589.6	7.274	65.76	275643.2	13.4
	T	5004.8	750.2	8.003	69.022	342444.2	18.2
Pitfall	L	4	1	4	59.4	245.4	1
	T	3.6	1	3.6	71.173	253.2	1
PrivateEye	L	6697	967	8.769	67.877	453342	21
	T	4050.5	705	7.253	65.018	259686.5	17.5
Venture	L	2672.2	416.2	6.684	63.528	168756	15
	T	884.8	178.2	5.427	64.418	57555.8	11.4

Table 4.3: Static properties of champions after 500 generations of training.

4.2.3 Dynamic Properties

Table 4.4 shows the maximum, mean, and minimum depth or path length recorded, the number of program instructions executed, and the time taken, in milliseconds, to resolve an action for the input vector for champions of an environment averaged over the 20 evaluation episodes. A pairwise two-tailed t-test gives values greater than 0.4 for all these properties, failing to establish a significant difference between the traversal strategies.

4.2.4 Action Distributions

Differences in action distributions were observed between learner and team traversals. Figure 4.9 shows the frequency of actions across all environments for champions after 500 generations of training. Learner traversal champions made more use of the 'FIRE','UP','NOOP','UPLEFTFIRE','LEFTFIRE', 'RIGHTFIRE', and 'UPLEFT' actions and less use of the 'UPFIRE', 'DOWNRIGHT', 'DOWN', 'DOWNLEFTFIRE', 'DOWNFIRE', 'UPRIGHTFIRE', 'LEFT', 'DOWNRIGHTFIRE','UPRIGHT', 'DOWNLEFT', and 'RIGHT' actions compared to their team traversal counterparts overall.

Environment	Traversal	Depth			Inst. / Action	Ex. Time (ms)
		Max	Mean	Min		
Asteroids	LEARNER	3	2.515	2	1888.302	0.286
	TEAM	2	2	2	1699.149	0.249
BattleZone	LEARNER	8	4.05	2	4142.286	0.974
	TEAM	6	4.022	3	3767.347	0.872
Bowling	TEAM	6	4.011	3	3360.65	0.865
	LEARNER	4	2.991	2	2549.583	0.587
Centipede	TEAM	5	2.789	2	2119.811	0.488
	LEARNER	4	2.697	2	2034.961	0.512
DoubleDunk	LEARNER	5	4.115	2	2569.771	0.595
	TEAM	5	3.451	2	1778.154	0.412
FishingDerby	LEARNER	8	4.389	2	3597.166	0.875
	TEAM	7	4.222	2	3796.17	0.928
Freeway	TEAM	7	3.572	2	3108.098	0.815
	LEARNER	5	3.281	2	1823.94	0.486
Frostbite	LEARNER	7	4.889	2	4392.095	1.081
	TEAM	7	4.517	1	2291.34	0.685
Gravitar	LEARNER	5	2.874	2	2948.947	0.662
	TEAM	5	2.703	2	2733.82	0.688
IceHockey	TEAM	7	4.589	2	3102.739	0.832
	LEARNER	6	4.266	3	3425.736	0.839
Kangaroo	TEAM	7	3.389	1	2525.632	0.647
	LEARNER	5	3.274	2	3000.996	0.705
MsPacman	TEAM	7	3.369	2	2772.332	0.686
	LEARNER	6	4.014	1	3220.894	0.819
Pitfall	LEARNER	1	1	1	250.324	0.093
	TEAM	1	1	1	256.323	0.077
PrivateEye	TEAM	9	5.033	3	2663.387	0.713
	LEARNER	8	6.042	2	6675.161	1.648
Venture	TEAM	8	4.579	3	2738.373	0.589
	LEARNER	7	3.82	1	2843.982	0.665

Table 4.4: Dynamic complexity measures averaged over all champions each playing 20 evaluation episodes. Depth refers to the length of the path from the root team to the returned action.

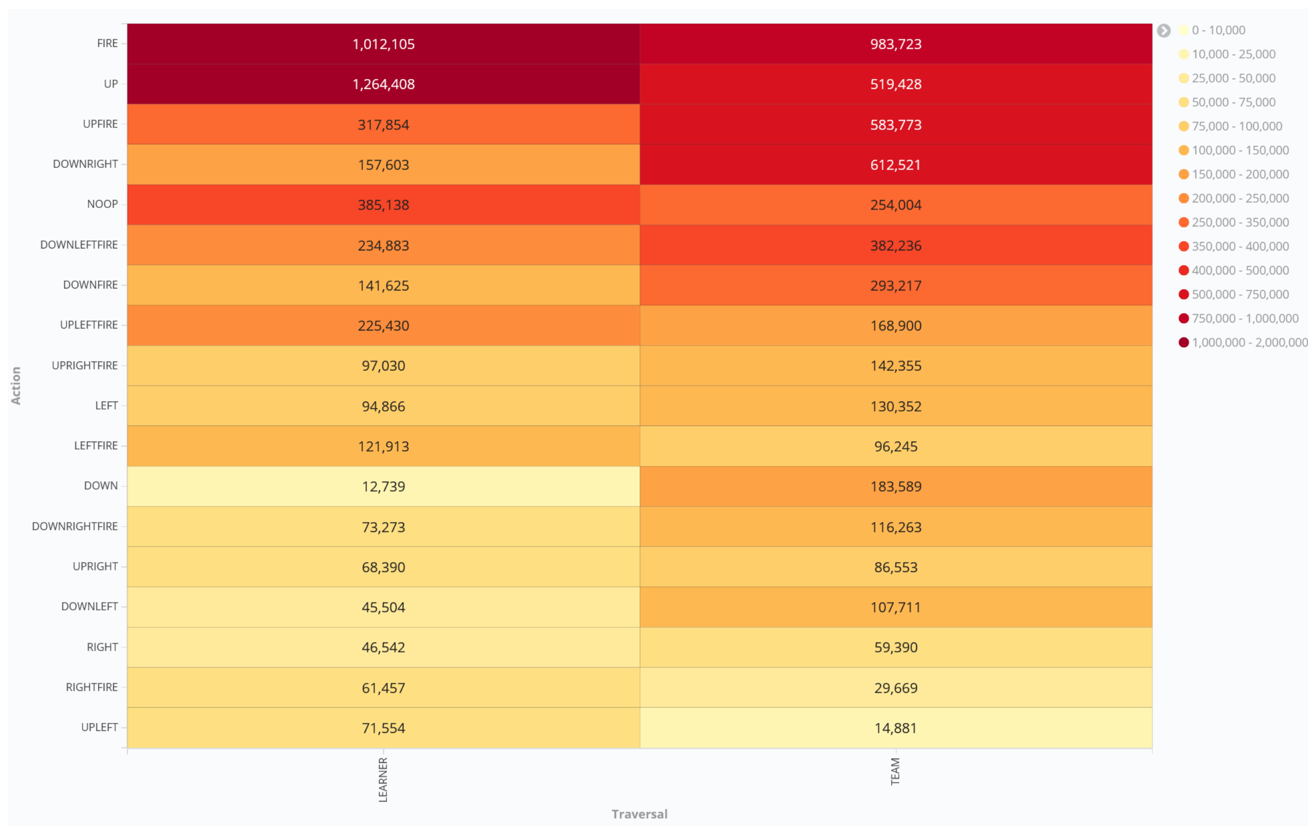

Figure 4.9: Action frequencies across all environments for champions after 500 generations of training.

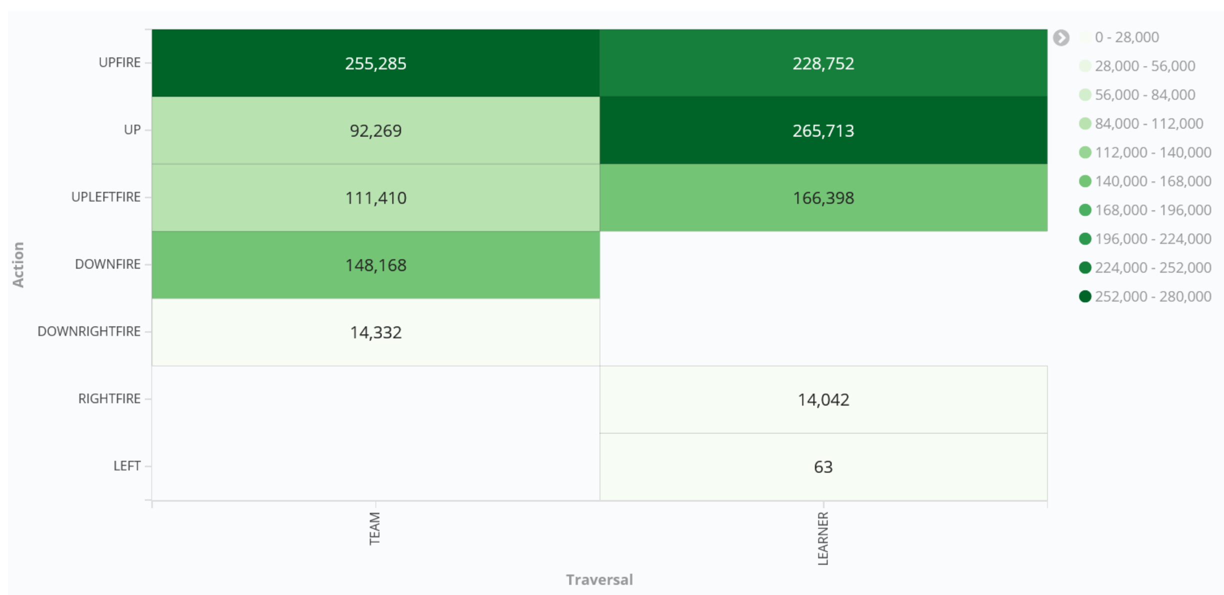

Figure 4.10: Action frequencies for champions trained on the BattleZone Atari game after 500 generations of training. Split by traversal type.

Two-tailed, pairwise t-tests performed on the action distribution Tables for individual environments failed to established statistical significance between the traversal strategies for any game.

Chapter 5

Analysis

Before diving into traversal comparisons, it can be constructive to view some of the most interesting solutions that champions found in action. For the Asteroids environment the winning strategy amongst the champions was to sit in place in the middle of the screen while rotating and firing as fast as possible. Figure 5.3 shows a champion after 500 generations of training in the middle of doing just that.

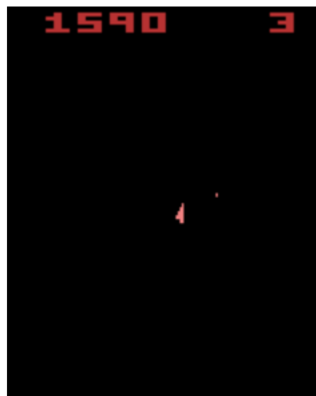

Figure 5.1: Hold 'LEFT' and smash 'LEFTFIRE' as much as possible. A TPG champion solution for the Asteroids Atari game.

Venture, a game where TPG appears to sometimes perform 100 times better than RAINBOW, has a less elegant solution. Venture-LEARNER-8 is the grand champion after 500 generations, and it happens to have discovered that if one goes up and to the left when the game starts one can stumble into one of the first 'rooms' in Venture. In this first room, if one is lucky, based on enemy movement an continuing with the up and left strategy from before it is possible to collect some treasure. Many times though Venture-LEARNER-8 misses the first room entirely. Conversely, continuing to hug the left wall may also lead it to a second room with a diamond in the middle surrounded by some traps. It does not appear to make anywhere near as much progress in this room than in the first with it's up left strategy, but nevertheless, in comparison to its peers who merely dodge enemies guarding the rooms by moving all

the way to the right of the screen, Venture-LEARNER-8 earns its champion belt.

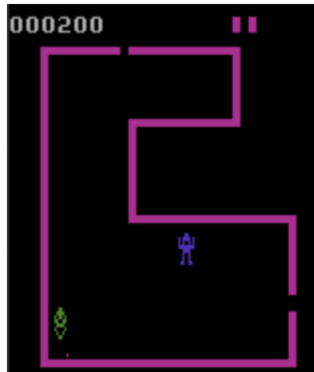

Figure 5.2: First 'LEFT', the 'UP', then possibly, success! Venture-LEARNER-8 moments before its demise at the hands of the green enemy after stumbling into the first room and happening upon some treasure.

A strategy more in line with one a human player may employ appears in the Bowling environment, where champions have figured out the correct button presses to align the ball to just slightly above the middle pin. This often results in a cascade of pins toppling, and in some cases even a strike!

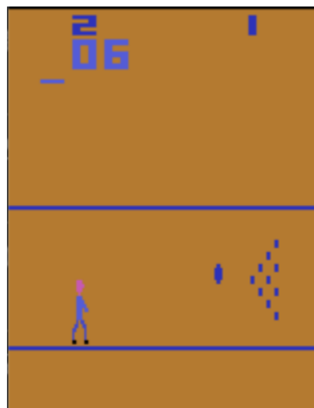

Figure 5.3: Line them up, then strike them down. Champions perfect the optimal shot in the Atari Bowling game after 400 generations of training.

5.1 Traversal Analysis

The most promising looking metrics to support a difference between team and learner traversal strategies came from the results gathered on action distributions. Because different games have different game ending conditions, there is significant variability

in the number of frames played for a given game. For example, a champion playing MsPacman may get caught by a ghost and only play a few thousand frames while in Bowling, a no-op agent could just idle around before the 10,000 frame limit ends the game. Therefore the sample sizes for action frequencies vary between games and the action distribution differences observed in figure 4.9 are not necessarily conclusive of a difference between the traversals.

Because the environment specific t-tests showed no significant differences between the traversals it is possible that the observed differences come from the discovery of slightly different strategies to maximize fitness in the environment, rather than particular preferences corresponding to traversal strategy.

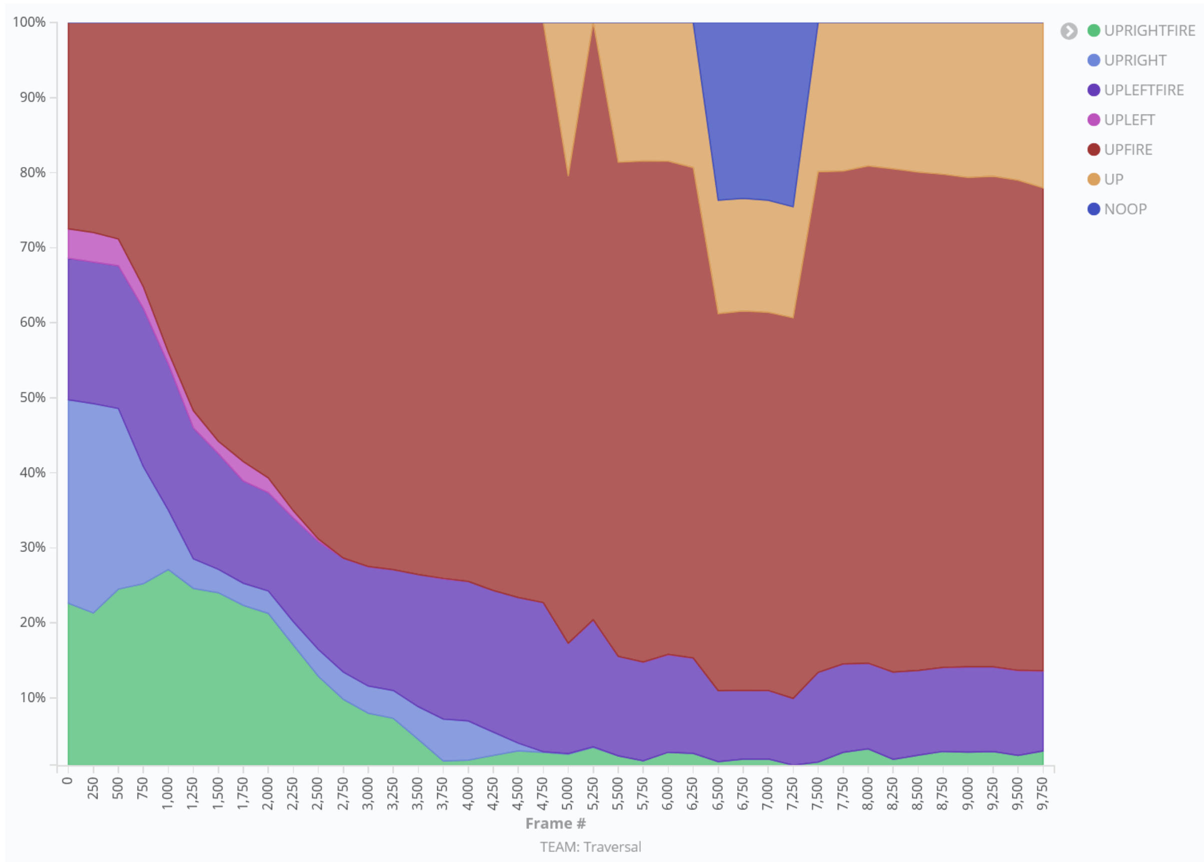

Figure 5.4: Action frequency distribution across all environments for champions with team traversal after 500 generations of training plotted as a percent of the whole against the frame #. Note that many actions occurred infrequently enough to be omitted.

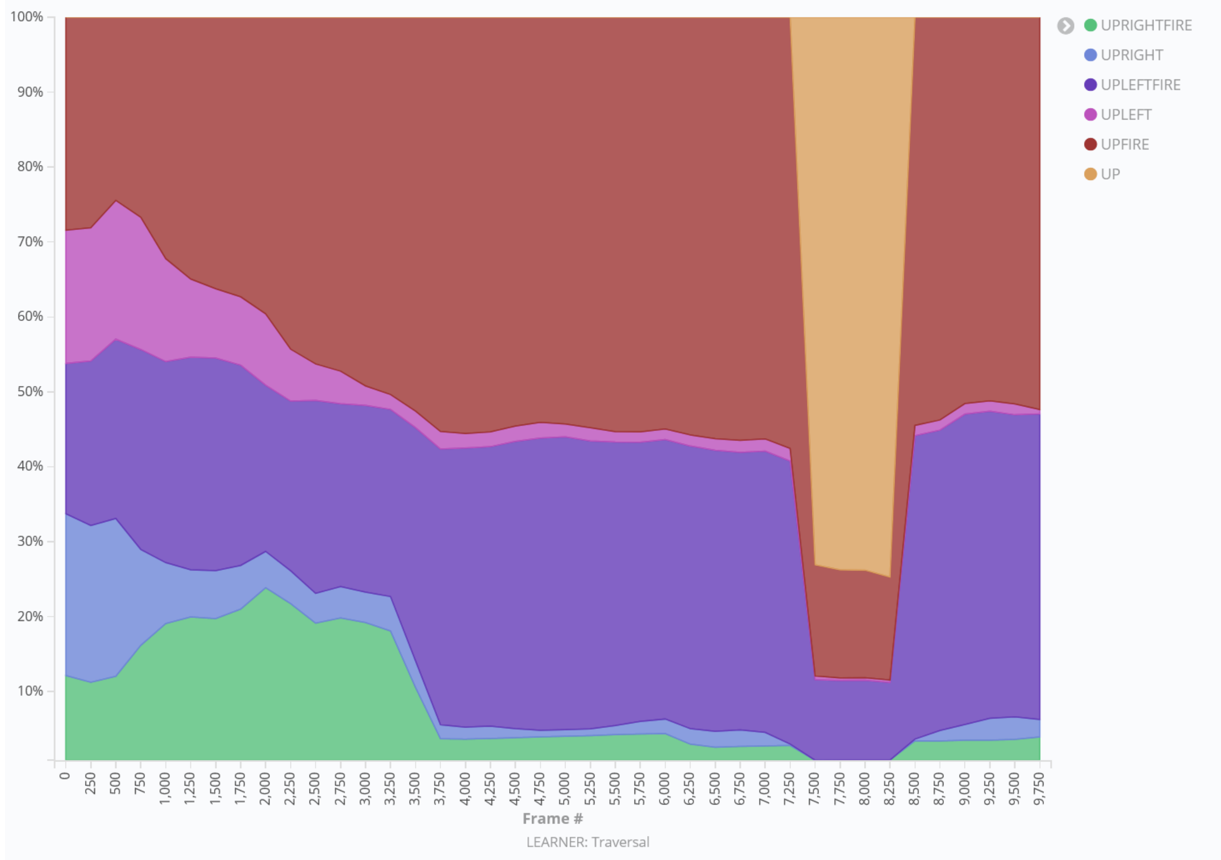

Figure 5.5: Action frequency distribution across all environments for champions with learner traversal after 500 generations of training plotted as a percent of the whole against the frame #. Note that many actions occurred infrequently enough to be omitted.

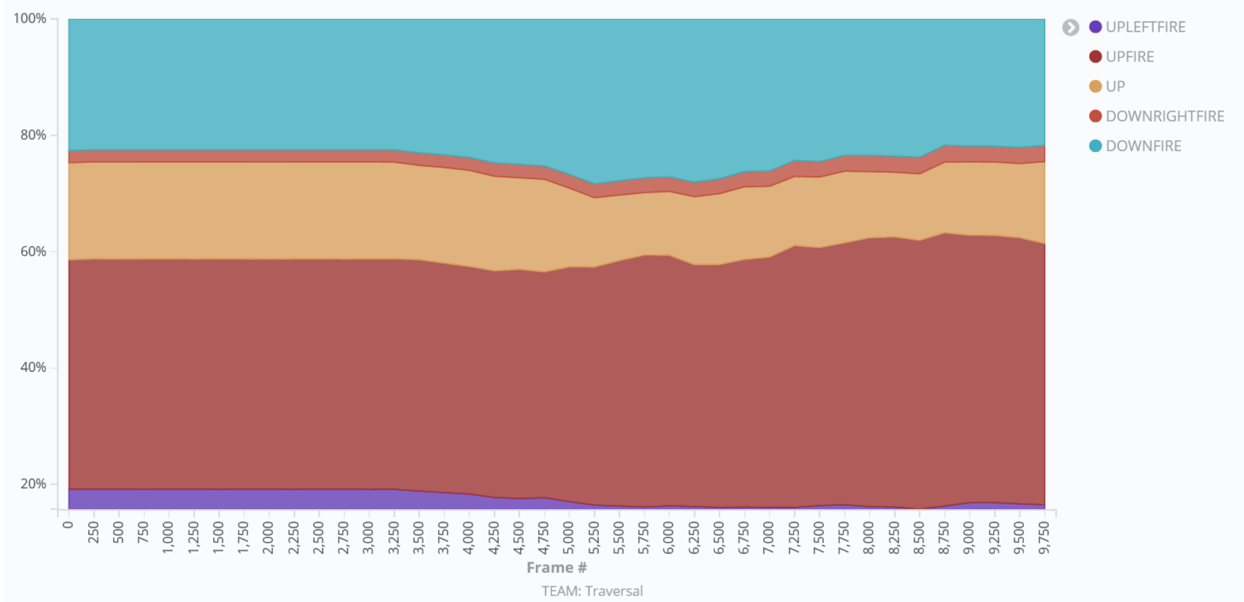

Figure 5.6: Action frequency distribution for champions trained on the BattleZone Atari game with team traversal after 500 generations of training plotted as a percent of the whole against the frame #.

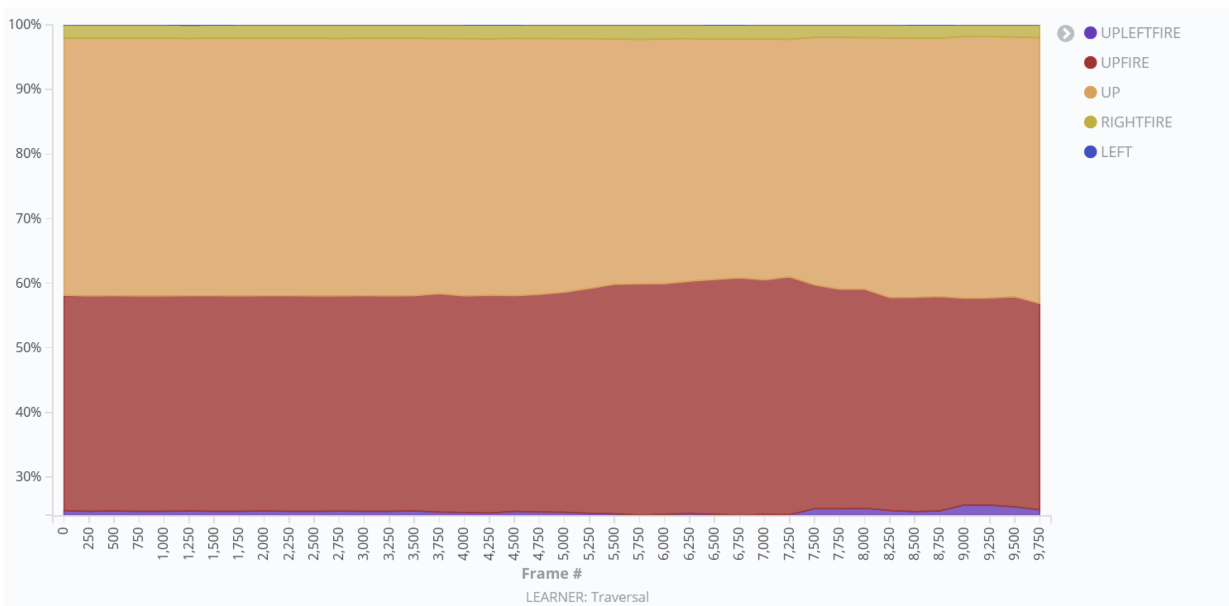

Figure 5.7: Action frequency distribution for champions trained on the BattleZone Atari game with learner traversal after 500 generations of training plotted as a percent of the whole against the frame #.

This conclusion is further supported by figures 5.4 and 5.5 which show the action

distributions as percentages of a whole plotted against the frames in which they appear. While there are differences between the traversals, they share major trends:

- 20% 'UPLEFTFIRE' 'mountain' starting from frames 0 and descending into frame 3,000.
- Heavy usage of 'UPFIRE' throughout, reaching peak usage around frame 3,500.
- A dramatic increase in 'UP' usage around frames 6,500-7,000.
- Diminishing usage of 'UPLEFT' and 'UPRIGHT' largely giving way to the primary usage of 'UPLEFTFIRE'.

When comparing performance, static and dynamic properties between the two traversals (using t-tests) no statistically significant differences were found. This prompted a more in-depth look at the path trace data we collected during the evaluation phases. Were learner traversal champions revisiting teams in their graphs with seemingly no distinguishable effect?

We sought to answer this question by specifically searching for the team revisits we hypothesized learner traversals would allow. To do this a java program was written to comb through the path traces gathered during the evaluation phases and look for paths where a team was revisited. In addition, because our path trace data also contained information on which learners bid at each stage in the path, the program would also look for 'revisit sites', that is, points in the path where, had a different learner won the bid, a team revisit would have occurred. Finally, the program would take note of which teams, actions and learners were visited across all episodes for a given champion. This data would be combined with the data collected about the respective champion's graph and used to determine how much of the champion's graph was actually utilized during the evaluation phase.

The analysis described was performed on both team and learner traversal champions. While one naturally would not expect to find team revisits in the team traversal paths, we did want to determine if the difference in traversal strategy amounted to a difference in graph utilization (path coverage of the graph) during evaluation. Thus, with this in mind, 2,214,761 learner traversal paths and 2,510,913 team traversal paths for a total of 4,725,674 paths were analyzed for the 500 generation champions.

No team revisits, nor team revisit sites were found, a result that potentially implies that new variation operators should be introduced that explicitly attempt to introduce looping structures (as opposed to the current case in which looping structures are 'assumed to appear'.

5.2 Team & Action Utilization

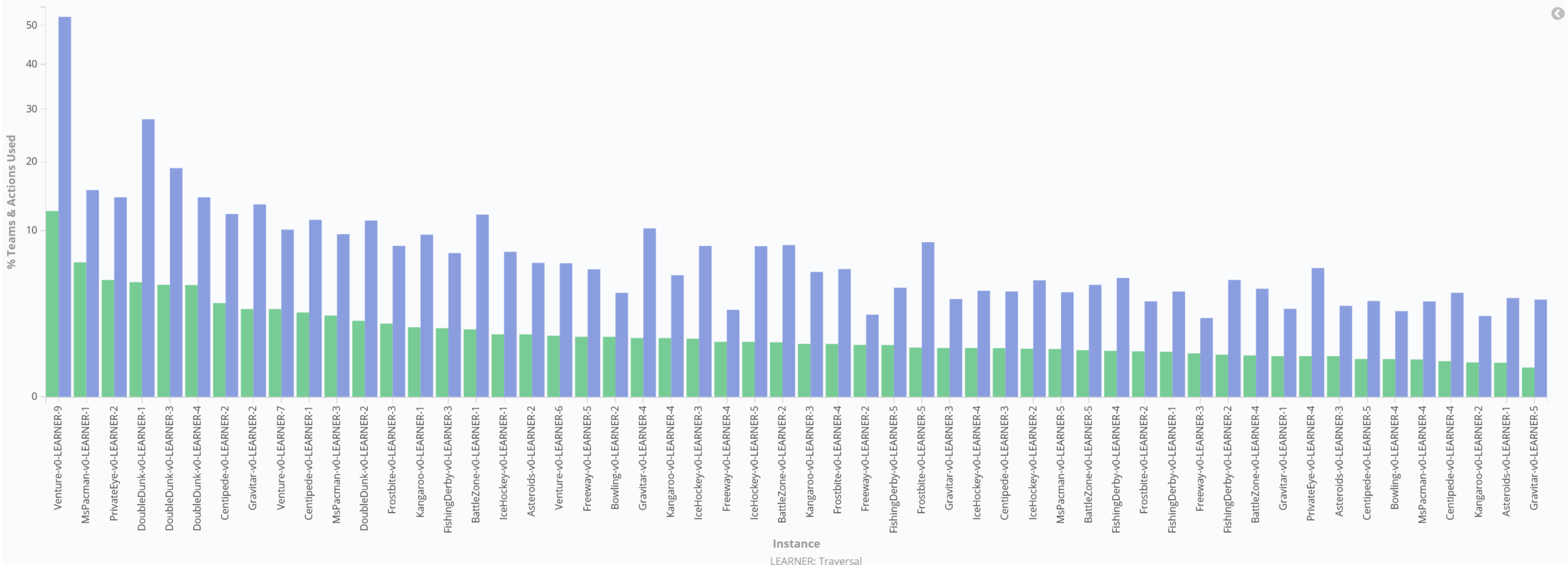

Figure 5.8: **Learner traversal** vertex cover analysis for generation 500 champions. In *green*, the percentage of vertices (teams and actions) covered by all paths traced over 20 evaluation episodes for a given champion (listed on the x-axis). In *blue*, the percentage of vertices within bid range of all paths traced over 20 evaluation episodes for a given champion. Champion graphs with fewer than 50 teams were excluded as their 50% - 100% coverage is a reflection of their uniquely small graph size. Square-root y-axis scale used to highlight smaller differences in lower coverages.

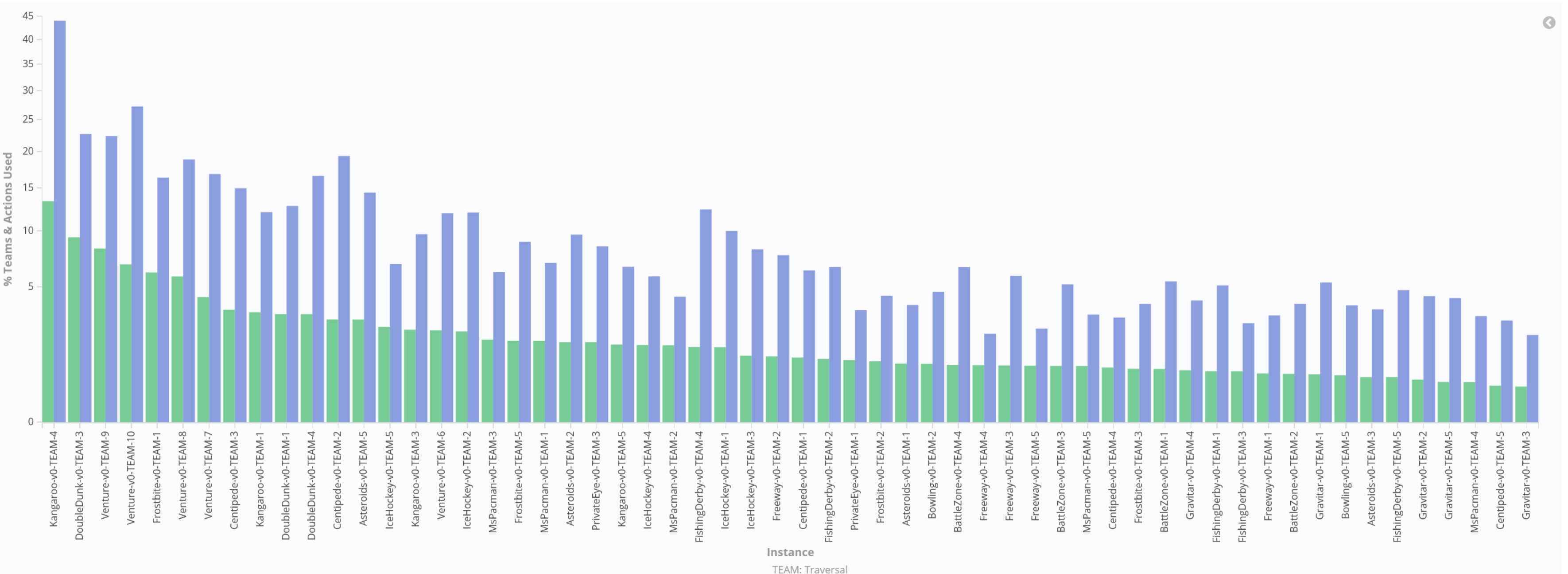

Figure 5.9: **Team traversal** vertex cover analysis for generation 500 champions. In *green*, the percentage of vertices (teams and actions) covered by all paths traced over 20 evaluation episodes for a given champion (listed on the x-axis). In *blue* the percentage of vertices within bid range of all paths traced over 20 evaluation episodes for a given champion. Champion graphs with fewer than 50 teams were excluded as their 50% - 100% coverage is a reflection of their uniquely small graph size. Square-root y-axis scale used to highlight smaller differences in lower coverages.

Figures 5.8 and 5.9 show the percentage of vertices covered by all paths during the 20 episode evaluation runs for individual champions with learner traversal and team traversal respectively. Champion graphs with fewer than 50 teams are excluded because their high vertex covers are more so a reflection of their graph size than the algorithm's utilization. The maximum graph utilization in vertx terms is 13.33% and 12.5% achieved by Kangaroo-TEAM-4 and Venture-LEARNER-9 respectively. Relaxing our utilization definition to include teams and actions which were contenders at the bidding stage of graph traversal (bid-range) we have maximum utilization of 44% and 52.237% for team and learner traversals respectively achieved by the same champions. Comparing vertex covers between traversals with two-tailed pairwise t-tests gives 0.449 for our strict definition, and 0.624 for our relaxed definition, showing no significant difference between the traversal strategies.

5.3 Learner Utilization

Figures 5.10 and 5.11 show the percentage of edges covered by all paths during the 20 episode evaluation runs for individual champions with learner traversal and team traversal respectively. Champion graphs with fewer than 50 learners are excluded because their high edge covers are more so a reflection of their graph size than the algorithm's utilization. The maximum graph utilization in edge terms is 5.556% and 3.15% achieved by DoubleDunk-TEAM-2 and Venture-LEARNER-10 respectively. Relaxing our utilization definition to include learners which were contenders at the bidding stage of graph traversal (bid-range) we have a maximum utilization of 23.333% and 19.048% achieved by DoubleDunk-TEAM-2 and Venture-LEARNER-9 respectively. Comparing edge covers between traversals with two-tailed pairwise t-tests gives 0.672 for our strict definition, and 0.437 for our relaxed definition, showing no significant difference between the traversal strategies.

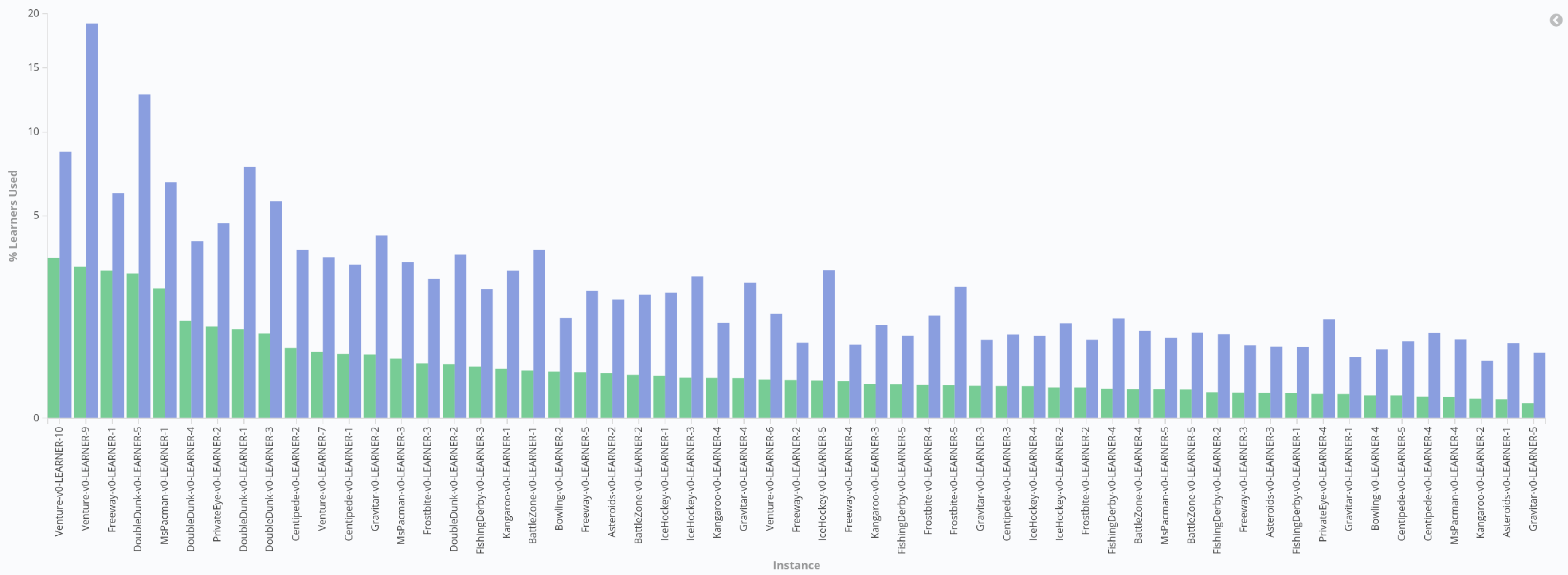

Figure 5.10: **Learner Traversal** edge cover analysis for generation 500 champions. In *green*, the percentage of edges (learners) covered by all paths traced over 20 evaluation episodes for a given champion (listed on the x-axis). In *blue*, the percentage of edges within bid range of all paths traced over 20 evaluation episodes for a given champion. Champion graphs with fewer than 50 learners were excluded as their 50%-100% coverage is a reflection of their uniquely small graph size. Square-root y-axis scale used to highlight smaller differences in lower coverages.

Figure 5.11: **Team Traversal** edge cover analysis for generation 500 champions. In *green*, the percentage of edges (learners) covered by all paths traced over 20 evaluation episodes for a given champion (listed on the x-axis). In *blue*, the percentage of edges within bid range of all paths traced over 20 evaluation episodes for a given champion. Champion graphs with fewer than 50 learners were excluded (ex: Pitfall, DoubleDunk, etc.) as their 50%-100% coverage is a reflection of their uniquely small graph size. Square-root y-axis scale used to highlight smaller differences in lower coverages.

Chapter 6

Conclusion

6.1 Traversal Strategies

No significant difference could be asserted between team and learner traversal strategies in terms of performance, static graph properties, dynamic graph properties, action distributions, or graph coverage either in terms of vertices or edges. Furthermore, the hypothesized revisiting of teams theoretically possible under the learner traversal strategy was not observed. One possible explanation for this result is that no structural cycles exist in the underlying graphs. Their existence was assumed from previous work done on TPG as well as the nature of learner mutation in pyTPG, which allows learners to mutate into pointing to any other team aside from its parent team and any team it was already pointing to. Further investigation could first attempt to confirm the existence of structural cycles from the saved champion graphs. Failure to find these structural cycles could indicate that the evolutionary process implemented in pyTPG indirectly prevents cycles from occurring. Another factor might be due to the relatively high number of root teams replaced per generation (50%) while the number of generations is relatively low (500). Which is to say, there might be a need for a relatively low turn-over in TPG agents for individuals to be around long enough to generate cyclic structures.

Our own analysis revealed that across all traversal strategies and environments less than 15% of vertices and 25% of edges appear in the paths used by the champions to produce actions during play. Even then, most champions utilized under 5% of both vertices and edges. This might suggest that a subset of programs quickly dominate bidding in their teams. If this is the case it merits further investigation into whether this happens because the suggested actions are indeed that proficient at returning fitness maximizing actions (in which case one may look to prune unused learners that survive simply by association) or if the behavior is an unwanted side-effect of TPG's design. In the latter case perhaps the selection mechanism is insufficient to remove

programs with high bids that produce sub-optimal results.

Nevertheless, it would be interesting to prune all unused learners, teams and actions from champions then re-evaluate them to assert that performance remains the same. If so, this technique could be used to 'compress' champions for practical application. Additionally, such low graph utilization could justify the development of more complex mutation and evolution mechanisms that can attempt to make better use of the evolved graph structures. For example, drawing additional inspiration from the hierarchical evolutionary process of Symbiotic Bid-Based GP previously studied by Robert Smith [57, 58], a two-phase evolutionary process where graph structure mutations are performed after the traditional TPG mutations.

Other findings originating from our observations during training indicate that there may be some cases in which the pyTPG implementation's builds significantly larger graphs than usual as was the case with Asteroids-TEAM-4 and Bowling-TEAM-4. This peculiar behavior may be worth further investigation to determine if it is rooted in a bug or in the algorithm's design.

Additionally, we made an attempt to explore how action frequencies vary over time by plotting them against the frame index. It is possible there may yet be some difference between traversals if one considered the unique action sequences throughout an episode of play. For example, one could compare metrics like the longest common action sequences between champions.

Finally, this work serves as a template for the kinds of metrics that would be of interest in future traversal strategy development. Graph utilization can be included from the start in future experiments, and with the quantity of data we were able to capture using Looking Glass one can start thinking about metrics like the percentage of teams or learners shared between root teams in a given population.

6.2 Summary

The contribution of this work is twofold, from a technical perspective it describes the operation of a set of mature industry leading technologies that can be used to facilitate reinforcement learning research at scale. This work can be reused and expanded upon by other researchers. From a scientific perspective this work contributes the first fine grained glance into the inner workings of TPG backed by millions of data points. It

produced evidence of scarce graph usage during execution in TPG trained champions, and a surprising lack of cycles observed during the learner traversal strategy. The contents of TPG's 'black box' has never been more accessible.

In conclusion, it was found that pyTPG builds sparse, possibly tree-like, champion graphs under both team and learner traversal strategies, resulting in no significant difference between the traversals. In particular no instance of the hypothesized team revisit under learner traversal was identified. This result motivates further investigation of the evolutionary process in pyTPG enabled by the tools developed for this work. In particular with respect to implementing variation operators designed to augment graph utilization and investigating the prevalence of hitchhikers (graph structures that do not contribute to performance but persist by their proximity to high-performing structures).

Bibliography

[1] About node.js, March 2021. https://nodejs.org/en/about/.

[2] Accelerate discovery, March 2021. https://www.ace-net.ca/.

[3] Apache avro™ 1.10.1 documentation, March 2021. https://avro.apache.org/docs/current/.

[4] Apache cassandra, March 2021. https://cassandra.apache.org/.

[5] Apache kafka, March 2021. https://kafka.apache.org/.

[6] Asteroids-v0, March 2021. https://gym.openai.com/envs/Asteroids-v0/.

[7] Aws cloudformation, March 2021. https://aws.amazon.com/cloudformation/.

[8] Cedar, March 2021. https://docs.computecanada.ca/wiki/Cedar.

[9] Compare onedrive cloud storage pricing and plans, March 2021. https://www.microsoft.com/en-ca/microsoft-365/onedrive/compare-onedrive-plans.

[10] The confluent_kafka api, March 2021. https://docs.confluent.io/platform/current/clients/confluent-kafka-python/html/index.html.

[11] Creating a scene, March 2021. https://threejs.org/docs/index.html#manual/en/introduction/Creating-a-scene.

[12] Deployments, March 2021. https://kubernetes.io/docs/concepts/workloads/controllers/deployment/.

[13] Dockerhub explore, March 2021. https://hub.docker.com/search?q=&type=image.

[14] Elassandra, March 2021. https://www.elassandra.io/.

[15] Elasticsearch features, March 2021. https://www.elastic.co/elasticsearch/features.

[16] Getting started with gym, March 2021. https://gym.openai.com/docs/.

[17] Graph diameter, March 2021. https://mathworld.wolfram.com/GraphDiameter.html.

[18] Infrastructure as code, March 2021. https://cloud.google.com/solutions/infrastructure-as-code.

[19] Inspecting the data, March 2021. https://www.elastic.co/guide/en/kibana/6.8/tutorial-inspect.html.

[20] Introduction to tensorflow, February 2021. https://www.tensorflow.org/learn.

[21] Job scheduling policies, March 2021. https://docs.computecanada.ca/wiki/Job_scheduling_policies.

[22] Jobs, March 2021. https://kubernetes.io/docs/concepts/workloads/controllers/job/.

[23] Joining queries, March 2021. https://www.elastic.co/guide/en/elasticsearch/reference/current/joining-queries.html.

[24] Kafka connect, March 2021. https://docs.confluent.io/platform/current/connect/index.html.

[25] Kibana, March 2021. https://www.elastic.co/kibana.

[26] Kubernetes components, February 2021. https://kubernetes.io/docs/concepts/overview/components/.

[27] Machine learning, March 2021. https://www.elastic.co/guide/en/kibana/current/xpack-ml.html.

[28] pickle — python object serialization, March 2021. https://docs.python.org/3/library/pickle.html.

[29] Pods, March 2021. https://kubernetes.io/docs/concepts/workloads/pods/.

[30] The progressive javascript framework, March 2021. https://vuejs.org/.

[31] python - docker official images, February 2021. https://hub.docker.com/_/python.

[32] Pytorch, February 2021. https://pytorch.org/.

[33] Schema registry overview, March 2021. https://docs.confluent.io/platform/current/schema-registry/index.html.

[34] Terraform language documentation, March 2021. https://www.terraform.io/docs/language/index.html.

[35] What is a container?, February 2021. https://www.docker.com/resources/what-container.

[36] What is container orchestration?, February 2021. https://www.redhat.com/en/topics/containers/what-is-container-orchestration.

[37] What is infrastructure as code?, February 2021. https://docs.microsoft.com/en-us/azure/devops/learn/what-is-infrastructure-as-code.

[38] Wide column stores, March 2021. https://db-engines.com/en/article/Wide+Column+Stores.

[39] Working with files in microsoft graph, March 2021. https://docs.microsoft.com/en-us/graph/api/resources/onedrive?view=graph-rest-1.0.

[40] Shawn Adams. Can i do sql-style joins in elasticsearch?, April 2020. https://rockset.com/blog/can-i-do-sql-style-joins-in-elasticsearch/.

[41] Caleidgh Bayer Robert Smith Malcolm Heywood Alexandru Ianta, Ryan Amaral. On the impact of tangled program graph marking schemes under the atari reinforcement learning benchmark. In *GECCO '21*, July 2021.

[42] Ryan Amaral. tpg-python, March 2021. https://github.com/Ryan-Amaral/PyTPG.

[43] 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*, 47:253–279, 2013.

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

[45] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. *CoRR*, abs/1606.01540, 2016.

[46] Matthew J. 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.

[47] Matteo Hessel, Joseph Modayil, Hado van Hasselt, Tom Schaul, Georg Ostrovski, Will Dabney, Dan Horgan, Bilal Piot, Mohammad Gheshlaghi Azar, and David Silver. Rainbow: Combining improvements in deep reinforcement learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, pages 3215–3222, 2018.

[48] Lorenz Huelsbergen. Toward simulated evolution of machine-language iteration. In *Proceedings of the First Annual Conference on Genetic Programming*, pages 315–320. MIT Press, 1996.

[49] 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.

[50] Stephen Kelly and Malcolm I. Heywood. Emergent solutions to high-dimensional multitask reinforcement learning. *Evolutionary Computation*, 26(3):347–380, 2018.

[51] Stephen Kelly, Robert J. Smith, and Malcolm I. Heywood. Emergent policy discovery for visual reinforcement learning through tangled program graphs: A tutorial. In Wolfgang Banzhaf, Lee Spector, and Leigh Sheneman, editors, *Genetic Programming Theory and Practice XVI*, Genetic and Evolutionary Computation, pages 37–57, 2018.

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

[53] Alexander Loginov and Malcolm I. Heywood. On evolving multi-agent FX traders. In Anna Isabel Esparcia-Alcázar and Antonio Miguel Mora, editors, *Applications of Evolutionary Computation - 17th European Conference, EvoApplications 2014, Granada, Spain, April 23-25, 2014, Revised Selected Papers*, volume 8602 of *Lecture Notes in Computer Science*, pages 203–214. Springer, 2014.

[54] 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.

[55] Gwen Shapira Neha Narkhede and Todd Palino. *Kafka: The Definitive Guide: Real-Time Data and Stream Processing at Scale*. O'Reilly, 2017.

[56] Miguel Nicolau and James McDermott. Genetic programming symbolic regression: What is the prior on the prediction? In Wolfgang Banzhaf, Erik D. Goodman, Leigh Sheneman, Leonardo Trujillo, and Bill Worzel, editors, *Genetic Programming Theory and Practice XVII [GPTP 2019, Michigan State University, East Lansing, Michigan, USA, May 16-19, 2019]*, pages 201–225. Springer, 2019.

[57] Robert J. Smith and Malcolm I. Heywood. Coevolving deep hierarchies of programs to solve complex tasks. In Peter A. N. Bosman, editor, *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO 2017, Berlin, Germany, July 15-19, 2017*, pages 1009–1016. ACM, 2017.

[58] Robert J. Smith, Stephen Kelly, and Malcolm I. Heywood. Discovering rubik's cube subgroups using coevolutionary GP: A five twist experiment. In Tobias Friedrich, Frank Neumann, and Andrew M. Sutton, editors, *Proceedings of the 2016 on Genetic and Evolutionary Computation Conference, Denver, CO, USA, July 20 - 24, 2016*, pages 789–796. ACM, 2016.

[59] Hans-Arno Jacobsen SSergio Gomez-Villamor Victor Muntes-Mulero and Serge Mankovskii Tilmann Rabl, Mohammad Sadoghi. Solving big data challenges for enterprise application performance management. In *Proceedings of the VLDB Endowment*, volume 5, pages 1724–1735, August 2012.

[60] Garnett Carl Wilson and Malcolm I. Heywood. Introducing probabilistic adaptive mapping developmental genetic programming with redundant mappings. *Genet. Program. Evolvable Mach.*, 8(2):187–220, 2007.

Appendix A

Supplementary Data from other Generations

For reference, the figures and Tables for performance, static, dynamic, and action distributions are provided for champions after 300 and 400 generations of training respectively.

Env.	Trav.	Fitness Max	N. Max	Avg.	N. Avg.	Min	Inst.
Asteroids	L	8090	369.7%	3396.2 ±1408.93	134.3%	730	5
	T	8060	368.2%	3276.6 ±1485.71	128.3%	980	
BattleZone	L	45000	71.5%	17180 ±8295.03	24.8%	1000	5
	T	45000	71.5%	15640 ±8999.46	22.3%	1000	
Bowling	T	153	1882.6%	89.62 ±19.86	964.1%	60	5
	L	126	1491.3%	86.11 ±15.06	913.2%	49	
Centipede	L	26251	397.6%	7040.18 ±4818.58	81.5%	2216	5
	T	29127	444.9%	6375.23 ±3994.05	70.5%	1584	
DoubleD.	T	2	112.6%	-1.26 ±1.15	94.8%	-2	5
	L	2	112.6%	-2.02 ±2.82	90.6%	-16	
FishingD.	L	-37	44.5%	-76.72 ±18.43	12.2%	-99	5
	T	-25	54.2%	-81.97 ±14.92	7.9%	-99	
Freeway	L	28	82.4%	22.9 ±1.78	67.4%	19	5
	T	25	73.5%	22.33 ±1.38	65.7%	19	
Frostbite	T	2890	29.7%	553.6 ±574.53	5.1%	70	5
	L	710	6.8%	216.3 ±140.16	1.6%	50	
Gravitar	L	2200	162.6%	456 ±328.65	22.7%	0	5
	T	2100	154.6%	306 ±324.67	10.7%	0	
IceHockey	L	5	131.7%	-0.54 ±3.13	86.7%	-9	5
	T	7	148.0%	-1.41 ±4.35	79.6%	-14	
Kangaroo	L	1400	9.2%	752 ±246.77	4.8%	200	5
	T	1200	7.9%	696 ±230.61	4.4%	200	
Krull	L	11001	131.6%	6975.967 ±2784.86	75.3%	1262	3
	T	11954	145.0%	6526.933 ±2955.78	69.0%	90	
MsPacman	L	3380	60.6%	784.375 ±566.46	9.4%	210	4
	T	1860	30.6%	498.4 ±326.30	3.8%	50	
Pitfall	L	0	99.9%	0 ±0	99.9%	0	5
	T	0	99.9%	0 ±0	99.9%	0	
PrivateEye	L	15100	358.2%	4034.425 ±5440.88	95.3%	-1000	4
	T	14666	347.8%	3109.188 ±3975.51	73.3%	-1000	
Robotank	T	20	30.8%	12.633 ±3.52	18.4%	6	3
	L	20	30.8%	12.083 ±2.97	17.5%	4	
Skiing	T	-6872	216.2%	-8312.317 ±923.31	188.7%	-9010	3
	L	-7372	206.6%	-8815.8 ±420.17	179.1%	-9008	
TimePilot	T	12200	92.2%	4091.25 ±1878.44	5.6%	400	4
	L	8600	53.8%	3966.25 ±2129.25	4.3%	500	
Venture	L	500	9090.9%	53 ±126.85	963.6%	0	5
	T	400	7272.7%	27 ±81.06	490.9%	0	

Table A.1: Minimum, mean, standard deviation, normalized mean, maximum and normalized maximum fitness by environment and traversal for champions after 300 generations of training. The instance column refers to the number of champions tested for each traversal type.

Env.	Trav.	# Learners	# Teams	# Learners/Team	# Inst./Team	# of Inst.	Diameter w.r.t Root
Asteroids	L	2016.2	359.4	6.226	67.412	136363.8	12.2
	T	2560.8	455.6	6.732	66.671	170009.6	15.8
BattleZone	L	3411.8	564.8	6.885	63.623	215278.2	16.2
	T	2027	396.8	6.045	63.934	124653.8	13.4
Bowling	T	6555.4	871.4	8.337	63.648	418982.4	18.4
	L	7993.8	1031.2	9.119	66.985	536703.4	20.8
Centipede	L	2227.6	379.2	5.902	65.772	142752.8	12.2
	T	1613.4	329.4	5.838	65.596	105514.4	12.2
DoubleDunk	T	404	82.6	5.337	64.036	26448.4	8
	L	902.6	176	5.464	62.881	57238.8	10
FishingDerby	L	4500.2	645.8	7.373	66.925	298691.8	16.8
	T	2697	467.4	6.196	73.413	180180.6	13.6
Freeway	L	1859.8	332.4	5.443	64.718	122036.2	11.4
	T	3217.8	564.8	6.86	65.05	209151.4	15.6
Frostbite	T	5610.8	821.8	7.805	64.044	358994.4	19.8
	L	1942.8	373	5.61	60.577	128335.4	12
Gravitar	L	5603.4	795.4	8.303	68.793	387484.4	18.4
	T	4035.8	647	7.557	67.956	277803.4	15.8
IceHockey	L	1570.2	285.8	5.714	65.955	99122.2	11
	T	1538	279.2	5.658	64.354	93669.2	10
Kangaroo	L	2869.6	499.4	6.671	67.422	193092.4	13.6
	T	1726.4	335	6.109	66.067	112818.6	12.8
Krull	L	9481	1200.667	9.521	67.467	631377.333	26
	T	7855	919.333	8.53	67.425	491249.333	19.333
MsPacman	L	2702.75	391.25	6.408	64.859	178819.5	10.25
	T	5767.8	846.6	8.439	69.032	395500.4	19

Table A.2: Static properties of champions after 300 generations of training.

Env.	Trav.	# Learners	# Teams	# Learners/Team	# Inst./Team	# of Inst.	Diameter w.r.t Root
Pitfall	L	4	1	4	59.4	245.4	1
	T	3.6	1	3.6	71.173	253.2	1
PrivateEye	L	5599	840.5	8.232	67.556	375307.75	19
	T	5671.5	843.5	8.018	65.155	362273.5	19.25
Robotank	T	291	61	5.144	64.645	19158.333	7
	L	184.667	40	4.67	65.53	11752	4
Skiing	T	2430	474.333	6.26	60.049	144864.333	14
	L	3516.333	629.333	6.845	62.043	218726.333	17
TimePilot	T	1648.75	327.75	6.025	67.421	112043.25	11.75
	L	2511.25	419.25	6.457	66.42	165722.75	12.25
Venture	L	900.4	167	5.475	63.352	59398.6	11.2
	T	649.8	131.8	5.195	63.846	42095.6	10.2

Table A.3: Static properties of champions after 300 generations of training continued.

Env.	Trav.	Depth			Inst. / Action	Ex. Time (ms)
		Max	Mean	Min		
Asteroids	T	7	2.766	2	2051.242	0.524
	L	4	2.998	2	2012.029	0.461
BattleZone	L	8	4.067	2	3695.388	0.926
	T	7	4.041	2	2922.362	0.794
Bowling	L	6	3.451	2	2751.092	0.656
	T	6	3.529	1	3359.787	0.801
Centipede	T	5	2.901	2	2390.726	0.574
	L	4	2.841	1	1635.122	0.454
DoubleDunk	L	5	4.296	3	2761.579	0.668
	T	5	3.334	2	1732.589	0.446
FishingDerby	L	9	4.309	3	3002.384	0.75
	T	7	4.299	1	3162.088	0.846
Freeway	T	5	3.789	2	2978.142	0.835
	L	4	2.681	1	1256.678	0.356
Frostbite	T	9	4.897	3	3041.437	0.921
	L	6	2.811	1	1898.099	0.509
Gravitar	L	4	3.258	2	2745.167	0.769
	T	4	2.772	1	2727.392	0.812
IceHockey	T	7	3.942	2	2112.466	0.648
	L	6	3.771	2	2840.19	0.759
Kangaroo	T	7	3.283	1	2776.562	0.803
	L	5	3.308	2	2787.663	0.796
Krull	T	7	3.839	1	3437.794	0.794
	L	5	3.515	2	3435.996	0.928
MsPacman	T	7	3.174	1	3349.425	0.581
	L	4	2.807	1	2158.585	0.622
Pitfall	L	1	1	1	245.403	0.106
	T	1	1	1	253.201	0.081
PrivateEye	L	8	4.624	3	4487.406	1.13
	T	7	4.937	3	3753.153	0.992
Robotank	L	3	2.33	2	1242.289	0.324
	T	3	2.329	2	1296.341	0.318
Skiing	L	7	3.854	2	2082.44	0.491
	T	6	3.935	2	2316.676	0.558
TimePilot	T	7	2.268	1	2480.945	0.562
	L	5	3.696	2	2711.833	0.706
Venture	T	8	4.697	2	2170.429	0.469
	L	7	3.552	2	1732.077	0.459

Table A.4: Dynamic properties of champions after 300 generations of training.

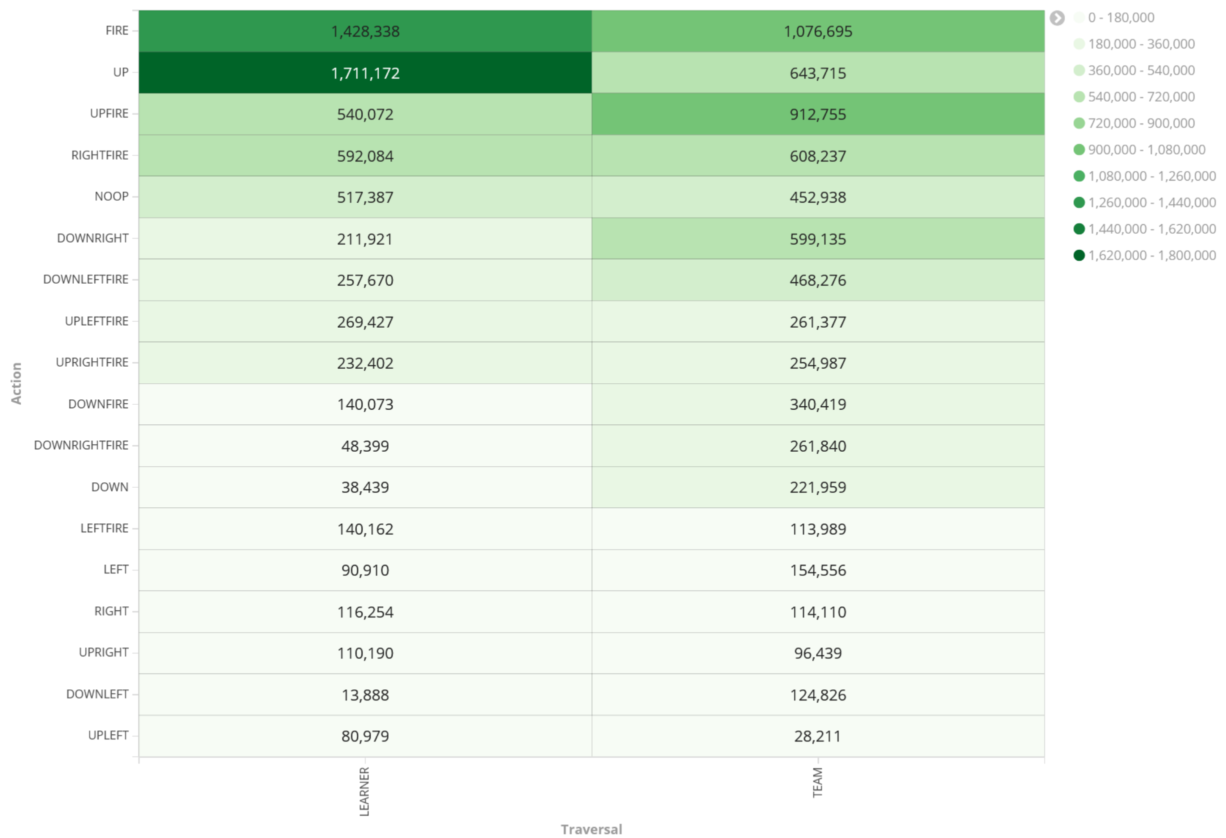

Figure A.1: Action frequencies across all environments for champions after 300 generations of training.

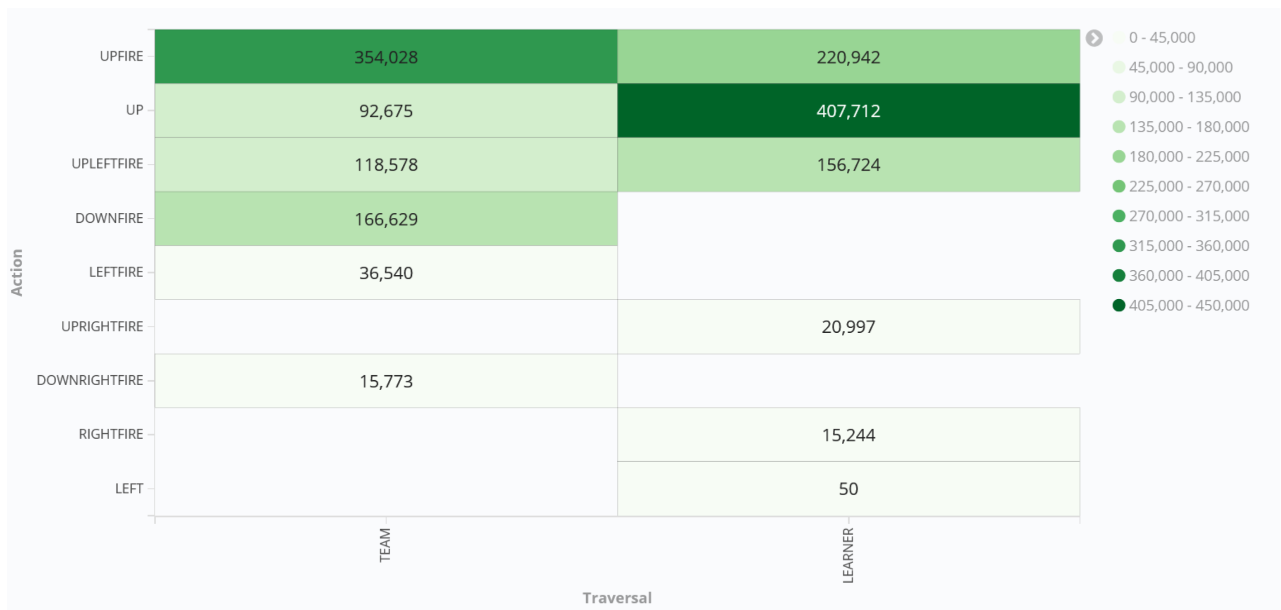

Figure A.2: Action frequencies for champions trained on the BattleZone Atari game after 300 generations of training. Split by traversal type.

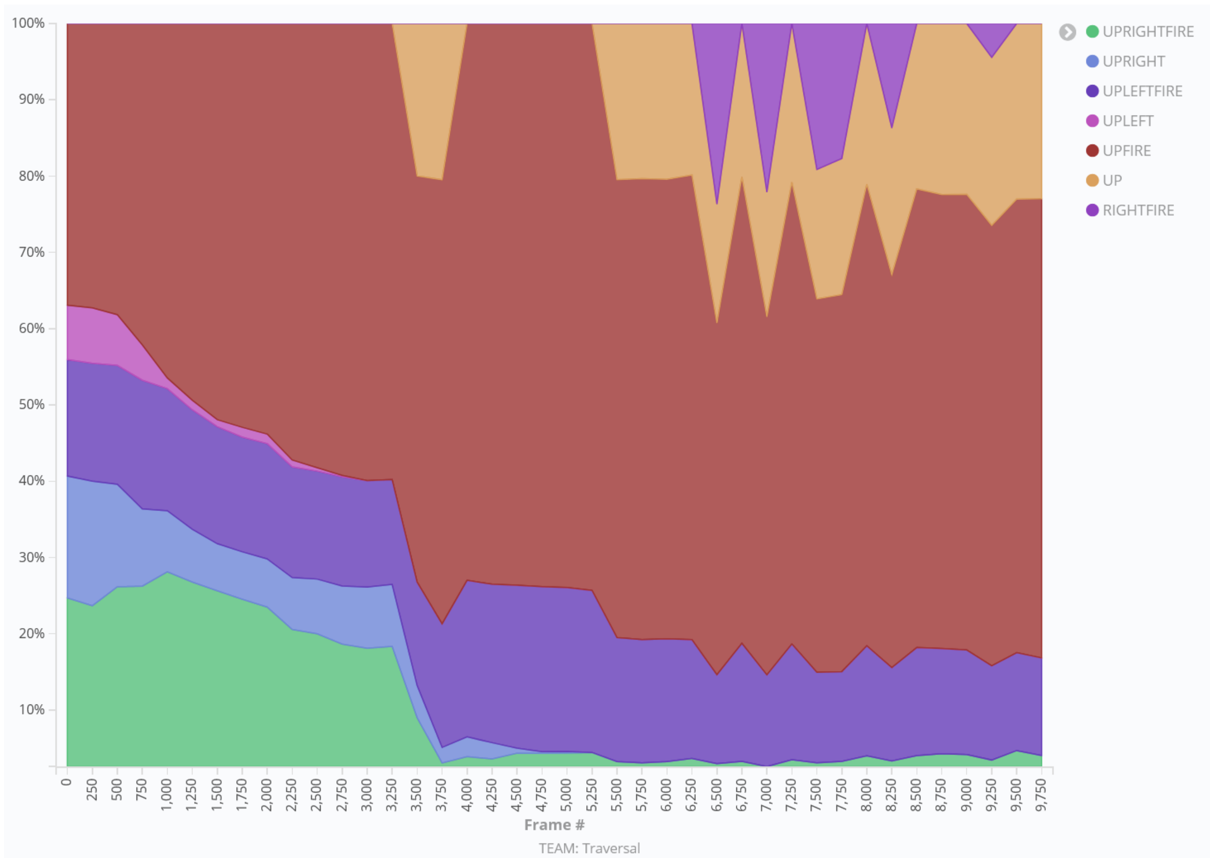

Figure A.3: Action frequency distribution across all environments for champions with team traversal after 300 generations of training plotted as a percent of the whole against the frame #. Note that many actions occurred infrequently enough to be omitted.

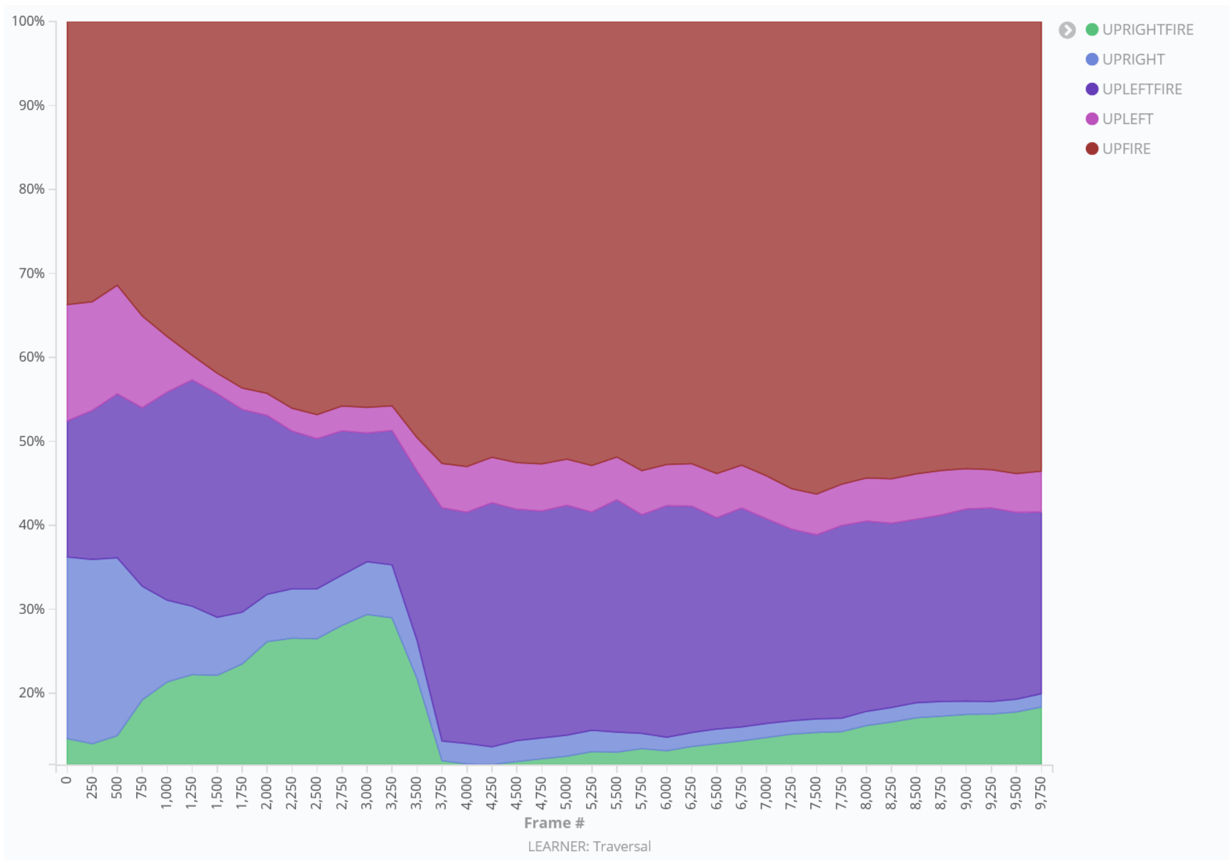

Figure A.4: Action frequency distribution across all environments for champions with learner traversal after 300 generations of training plotted as a percent of the whole against the frame #. Note that many actions occurred infrequently enough to be omitted.

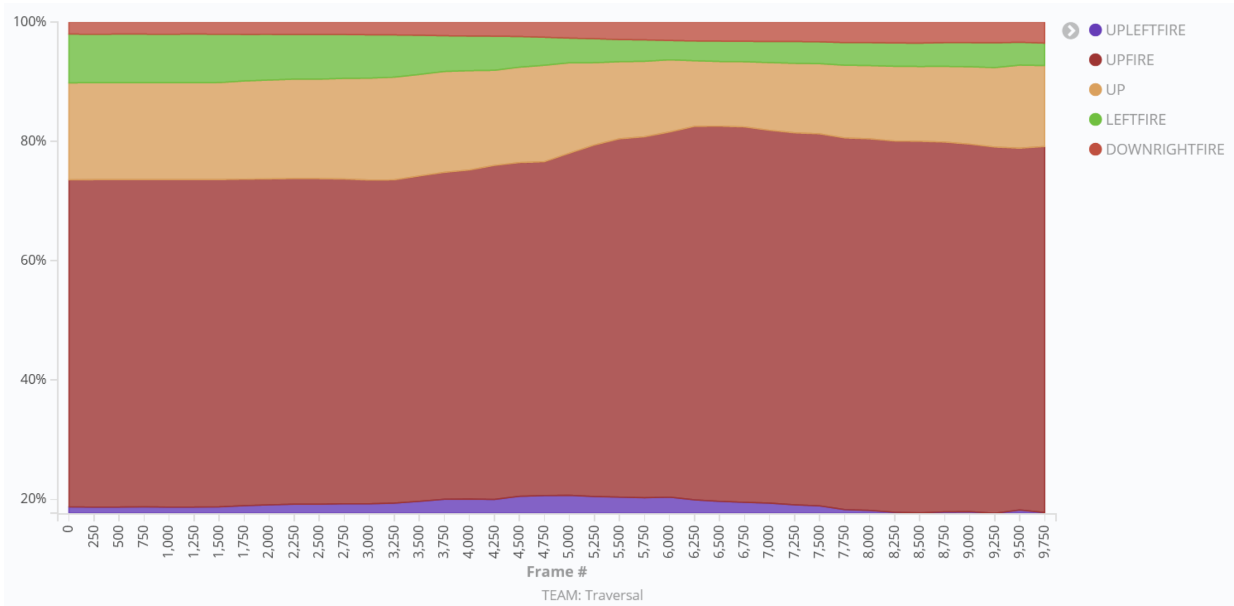

Figure A.5: Action frequency distribution for champions trained on the BattleZone Atari game with team traversal after 300 generations of training plotted as a percent of the whole against the frame #.

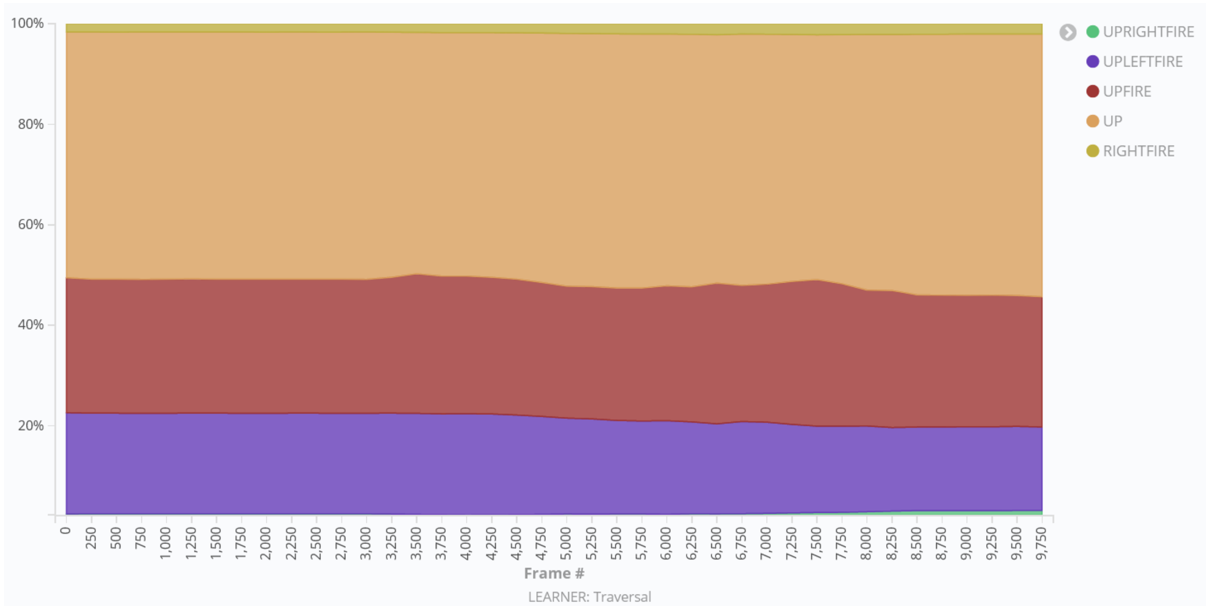

Figure A.6: Action frequency distribution for champions trained on the BattleZone Atari game with learner traversal after 300 generations of training plotted as a percent of the whole against the frame #.

Env.	Trav.	Fitness: Max	N. Max	Mean	N. Mean	Min	Inst.
Asteroids	T	12100	570.8%	3221.875 ±1788.61	125.5%	930	4
	L	7910	360.7%	3077.875 ±1365.70	118.3%	580	
BattleZone	T	35000	54.7%	18150 ±8121.11	26.5%	2000	4
	L	49000	78.2%	16637.5 ±8756.77	23.9%	2000	
Bowling	L	125	1476.8%	86.213 ±16.43	914.7%	56	4
	T	139	1679.7%	75.025 ±38.60	752.5%	0	
Centipede	T	18630	272.2%	6332.84 ±3595.63	69.8%	1746	5
	L	24677	371.7%	5729.24 ±3601.30	59.9%	2216	
DoubleD.	T	2	112.6%	-1.5 ±1.03	93.4%	-2	5
	L	2	112.6%	-2.46 ±2.86	88.2%	-12	
FishingD.	L	-27	52.6%	-75.41 ±18.80	13.2%	-99	5
	T	-35	46.1%	-81.07 ±13.62	8.6%	-99	
Freeway	L	27	79.4%	22.65 ±1.73	66.6%	18	5
	T	27	79.4%	22.33 ±1.51	65.7%	19	
Frostbite	T	2660	27.2%	528.9 ±530.13	4.9%	60	5
	L	720	6.9%	223.7 ±165.00	1.7%	90	
Gravitar	T	1150	78.4%	422 ±266.58	20.0%	0	5
	L	2000	146.6%	404.5 ±301.33	18.6%	0	
IceHockey	L	5	131.7%	-0.69 ±3.17	85.4%	-12	5
	T	7	148.0%	-1.32 ±4.75	80.3%	-14	
Kangaroo	L	1400	9.2%	780 ±234.09	5.0%	200	5
	T	1400	9.2%	700 ±204.93	4.4%	200	
Krull	T	11379	136.9%	6443.725 ±3076.98	67.8%	930	2
	L	9351	108.5%	5140.075 ±2374.14	49.6%	1602	
MsPacman	L	2240	38.1%	802.9 ±492.89	9.8%	210	5
	T	2050	34.4%	589.2 ±430.54	5.6%	80	
Pitfall	L	0	99.9%	0 ±0	99.9%	0	5
	T	0	99.9%	0 ±0	99.9%	0	
PrivateEye	L	14556	345.2%	3863.5 ±4469.67	91.2%	-1000	2
	T	4478	105.8%	1866.2 ±2455.56	43.7%	-1000	
Skiing	T	-8882	177.8%	-8938.4 ±38.79	176.7%	-9010	1
	L	-7390	206.3%	-9081.15 ±1578.17	174.0%	-13537	
TimePilot	T	8300	50.6%	4195 ±2067.96	6.7%	400	2
	L	11800	88.0%	4020 ±2324.77	4.8%	700	
Venture	L	700	12727.3%	47 ±126.85	854.5%	0	5
	T	300	5454.5%	19 ±61.14	345.5%	0	

Table A.5: Minimum, mean, standard deviation, normalized mean, maximum and normalized maximum fitness by environment and traversal for champions after 400 generations of training. The instance column refers to the number of champions tested for each traversal type.

Env.	Trav.	# Learners	# Teams	# Learners/Team	# Inst./Team	# of Inst.	Diameter w.r.t Root
Asteroids	T	2712.25	471.5	6.727	66.061	180548	15.25
	L	2512	445	6.564	66.408	167344.25	13.5
BattleZone	T	3503	599.75	6.907	62.771	213430.5	16.75
	L	3989.25	661.5	7.24	61.515	242858.75	16.5
Bowling	L	7543	982.75	8.884	66.877	506618.25	20.25
	T	8791	1130.5	9.388	63.16	550212	22.5
Centipede	T	2741.4	463.6	6.388	65.761	175936.2	13.6
	L	3210.4	517	6.411	65.079	201548.2	14.6
DoubleDunk	T	404	82.6	5.337	64.036	26448.4	8
	L	902.6	176	5.464	62.881	57238.8	10
FishingDerby	L	5154	729.2	7.723	67.226	342682	17.6
	T	2578.6	463	6.143	73.194	172339.4	13
Freeway	L	3343.2	535.6	6.75	65.299	218128.8	14.8
	T	3743.8	649.4	7.231	65.42	244108.4	16.6
Frostbite	T	6355.2	896	8.127	63.949	404028	20.8
	L	3821.8	631.6	7.004	65.145	248301	15.8
Gravitar	T	4170.6	665.2	7.691	67.796	284765.4	16.4
	L	5148	765.6	8.136	68.623	355691	18
IceHockey	L	2222.2	384	6.039	65.423	140663.8	11.6
	T	2484.8	457.4	6.305	63.388	155488	13.4
Kangaroo	L	5343.6	807.2	8.066	67.674	363594.2	18
	T	4699.6	594.2	7.431	66.291	315791	15.8
Krull	T	5577.5	804	8.119	65.395	348949.5	18
	L	4762	756.5	7.875	68.873	331592.5	17

Table A.6: Static properties of champions after 400 generations of training.

Env.	Trav.	# Learners	# Teams	# Learners/Team	# Inst./Team	# of Inst.	Diameter w.r.t Root
MsPacman	L	3982.8	574.2	7.16	65.684	269177.2	13.2
	T	5767.8	846.6	8.439	69.032	395500.4	19
Pitfall	L	4	1	4	59.4	245.4	1
	T	3.6	1	3.6	71.173	253.2	1
PrivateEye	L	6908	963	8.671	68.003	468852	20.5
	T	7805.5	1048	8.969	64.161	489204.5	20.5
Skiing	T	2281	455	6.187	58.296	131294	13
	L	3156	632	6.44	61.474	196013	17
TimePilot	T	1611.5	330	5.951	67.386	109799	12
	L	1257	261	5.821	66.055	82227	9.5
Venture	L	1400.4	255.6	5.876	63.409	91563	12.2
	T	705	142.8	5.258	64.166	45530.6	10.6

Table A.7: Static properties of champions after 400 generations of training continued.

Env.	Trav.	Depth Max	Depth Mean	Depth Min.	Inst. / Action	Ex. Time (ms)
Asteroids	T	7	2.742	2	2304.732	0.605
	L	4	3.14	2	2235.894	0.537
BattleZone	L	8	4.179	2	4060.726	1.042
	T	7	5.216	4	4411.206	1.184
Bowling	L	6	3.38	2	2463.069	0.642
	T	6	3.721	2	3258.217	0.929
Centipede	T	5	2.733	2	2052.346	0.518
	L	4	2.657	2	1963.538	0.502
DoubleDunk	L	5	4.298	3	2722.207	0.695
	T	5	3.533	2	1796.941	0.439
FishingDerby	L	9	4.126	2	3001.052	0.739
	T	7	4.271	1	3509.018	0.985
Freeway	L	5	3.279	2	1822.857	0.568
	T	5	3.493	2	3211.293	0.953
Frostbite	T	9	5.598	3	3526.873	1.223
	L	6	3.641	2	2304.993	0.781
Gravitar	L	4	3.025	1	2607.088	0.681
	T	4	2.69	2	2375.059	0.633
IceHockey	T	7	4.333	2	2725.122	0.738
	L	6	3.734	1	3227.704	0.752
Kangaroo	T	7	3.647	1	2792.086	0.76
	L	5	3.134	2	2752.819	0.662
Krull	L	7	3.602	2	2525.618	0.576
	T	5	2.93	2	2503.396	0.511
MsPacman	T	7	3.148	1	3349.882	0.783
	L	6	4.162	1	2996.454	0.814
Pitfall	L	1	1	1	245.394	0.092
	T	1	1	1	253.182	0.071
PrivateEye	L	7	5.5	3	5217.377	1.209
	T	5	3.217	2	2003.513	0.515
Skiing	L	4	4	4	2252.333	0.674
	T	3	3	3	2422	0.647
TimePilot	T	7	3.029	2	3084.492	0.699
	L	5	4.189	2	2477.993	0.584
Venture	L	9	3.165	2	1950.828	0.431
	T	8	5.189	3	2558.535	0.552

Table A.8: Dynamic properties of champions after 400 generations of training.

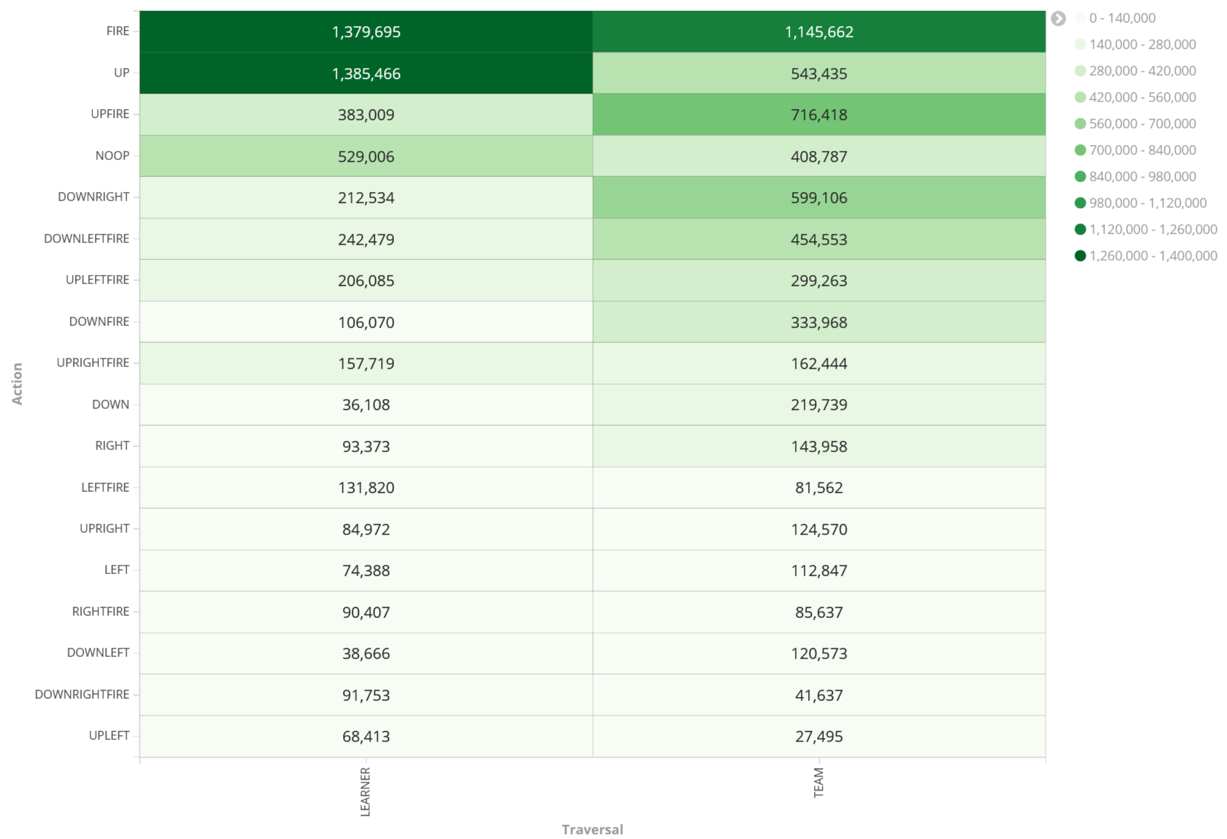

Figure A.7: Action frequencies across all environments for champions after 400 generations of training.

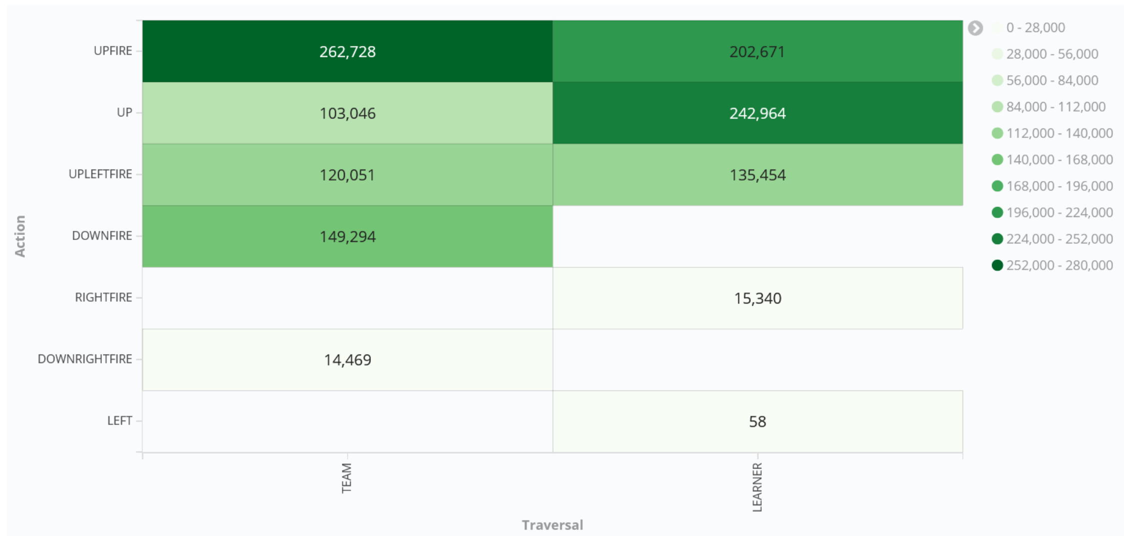

Figure A.8: Action frequencies for champions trained on the BattleZone Atari game after 400 generations of training. Split by traversal type.

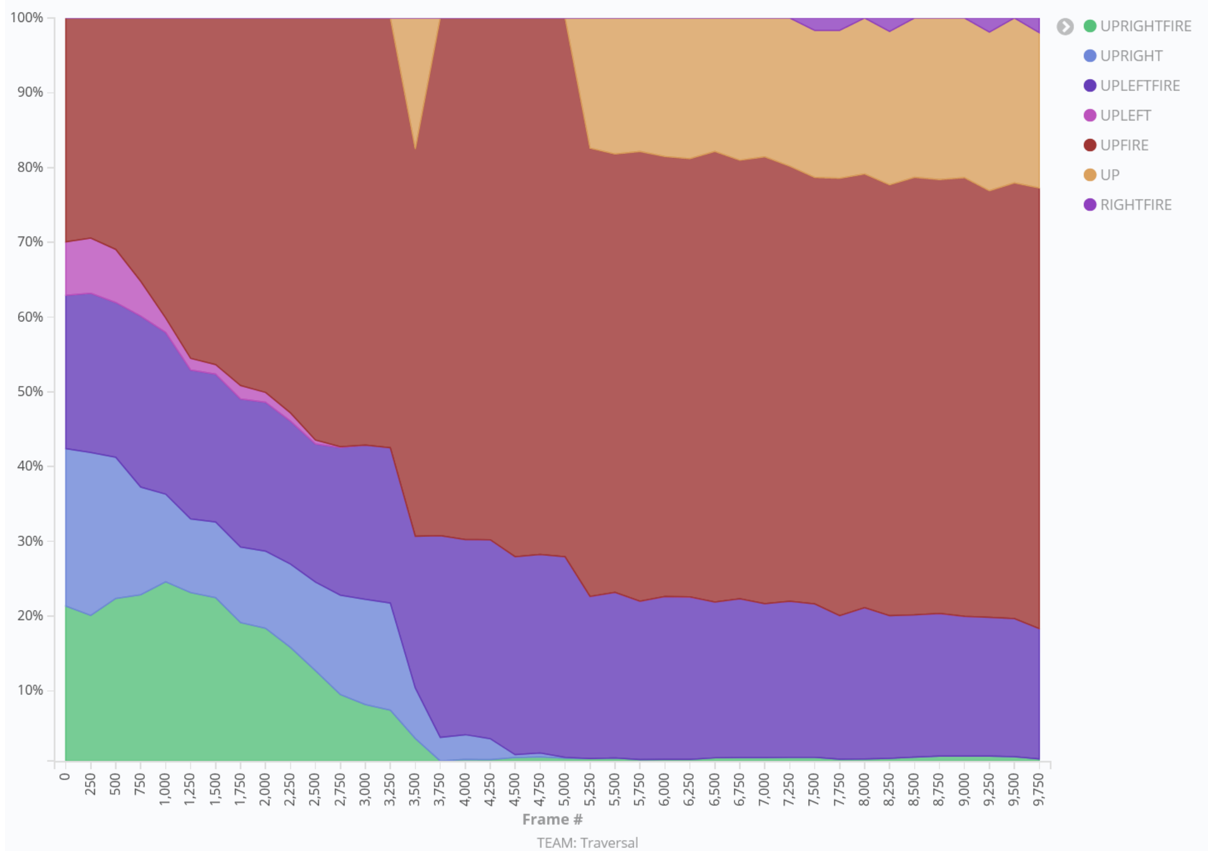

Figure A.9: Action frequency distribution across all environments for champions with team traversal after 400 generations of training plotted as a percent of the whole against the frame #. Note that many actions occurred infrequently enough to be omitted.

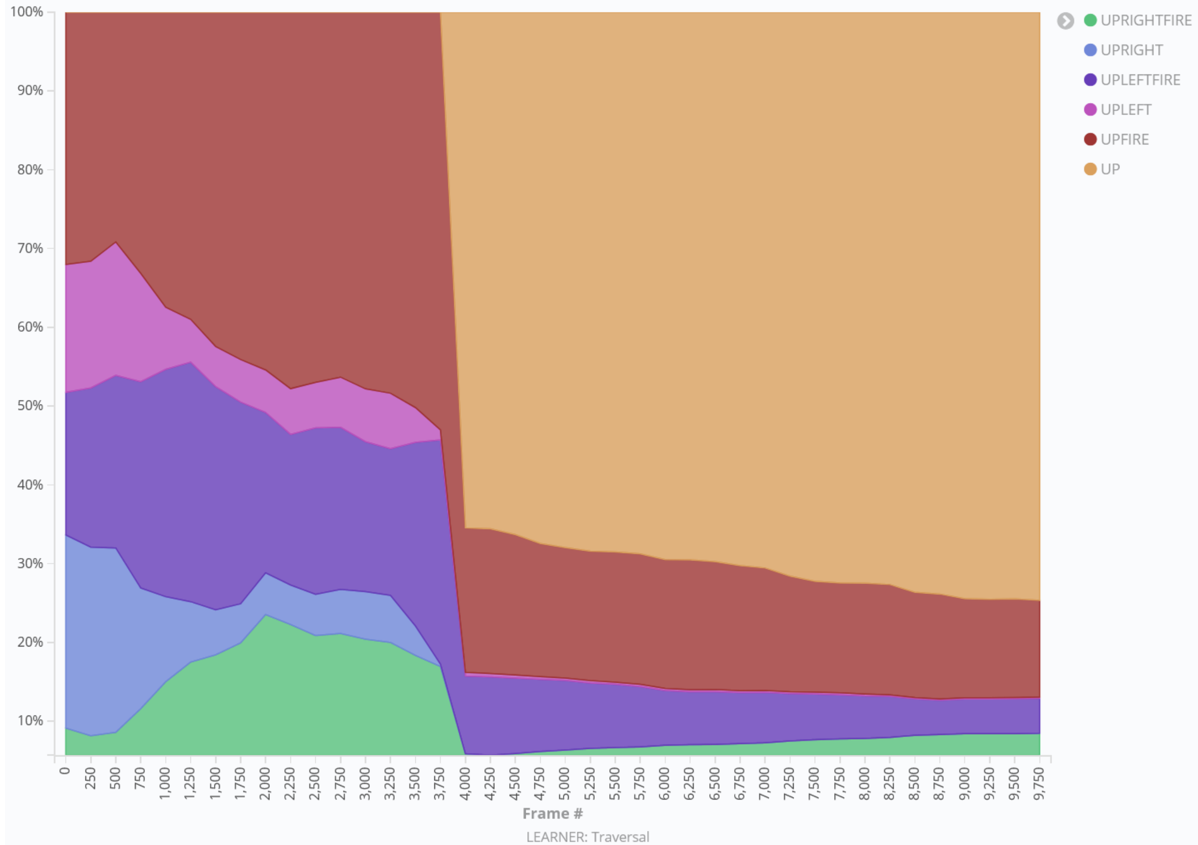

Figure A.10: Action frequency distribution across all environments for champions with learner traversal after 400 generations of training plotted as a percent of the whole against the frame #. Note that many actions occurred infrequently enough to be omitted.

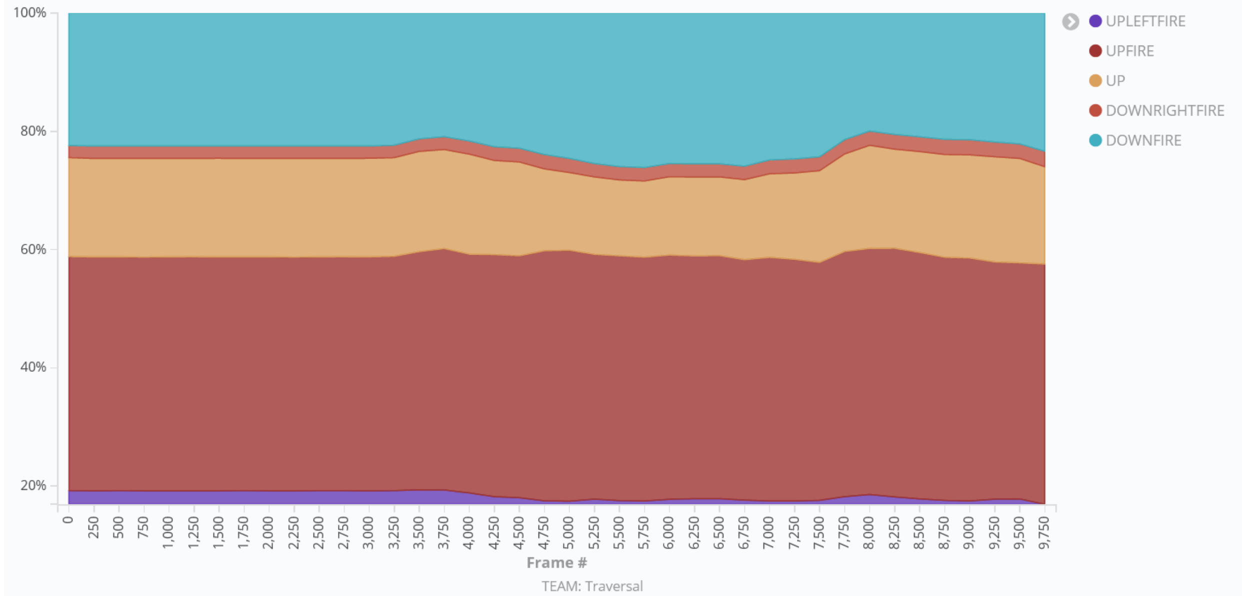

Figure A.11: Action frequency distribution for champions trained on the BattleZone Atari game with learner traversal after 400 generations of training plotted as a percent of the whole against the frame #.

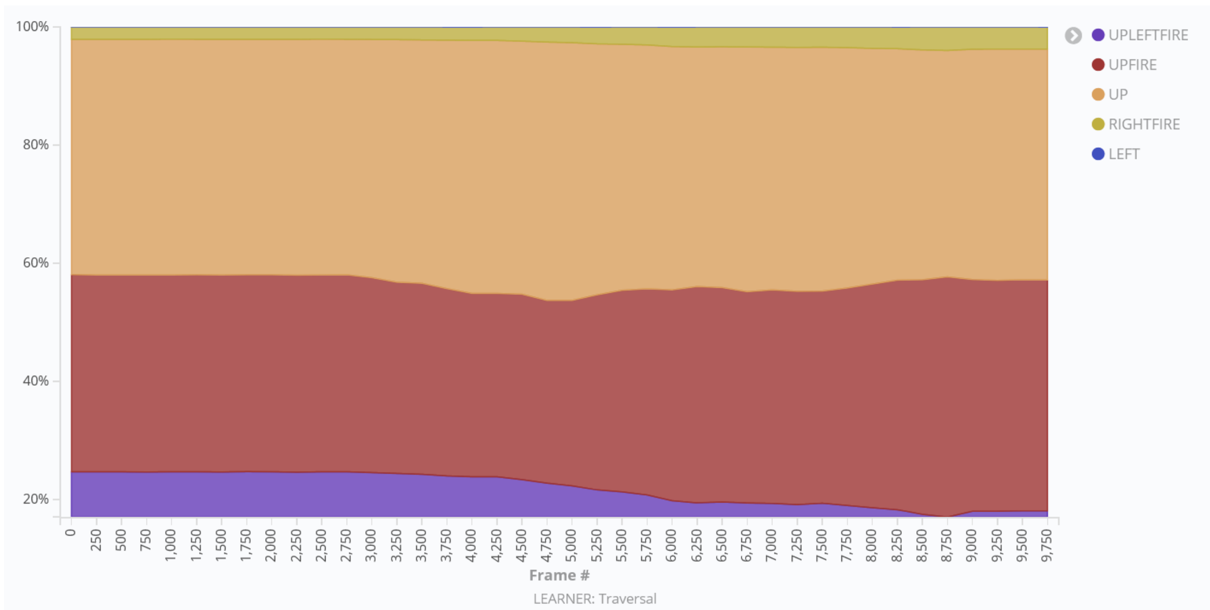

Figure A.12: Action frequency distribution for champions trained on the BattleZone Atari game with learner traversal after 400 generations of training plotted as a percent of the whole against the frame #.

Appendix B

The Performance and Future of Looking Glass

The Looking Glass platform performed remarkably in allowing us to perform experiments on this scale. Because of the the platform we were able to take a near 'capture everything' approach in terms of data collection, which gave us flexibility during analysis to pivot and take a different perspective on our results (in terms of graph coverage) without having to develop a new set of experiments to capture the required data saving months of computational time.

The ability to monitor experiments in near real-time allowed us to restart failed runs faster. The redundancy provided by the underlying Elassandra cluster, gave peace of mind against the possibility of hardware failure. During analysis, Kibana became a sort of creative workbench for exploring the data collected all in one spot. Where in the past the time between coming up with an interesting relationship to search for and seeing the relevant data visualized could have been hours or even days, with Kibana it was often minutes.

There is ample room for improvement however, it is clear that Looking Glass and the research pipeline as a whole lack polish and of course documentation. When other colleagues started making use of the platform, bugs in the code creating automatic cloud backups sent saves every generation for their experiments resulting in tens of thousands of zip files sitting in a single OneDrive folder. This resulted in Automatic Cloud Backups ultimately failing, thankfully, when the bulk of the experiments were already complete.

A non-exhaustive list of future work might include:

- Terraform configuration files used to deploy Looking Glass have our lab specific values hard-coded into them and must be refactored for general usage.
- Parts of Kibana and many other services are exposed through plaintext HTTP, where they should be using HTTPS or not be exposed at all for security reasons.

- Standard libraries should be developed for interacting with the platform from various programming languages, and documentation provided for their use.
- Kafka should be deployed as a cluster, similar to Elassandra to allow scaling based on metric throughput needs.
- The one elastic search index per metric schema approach to storage does not seem scaleable.
- Long term storage and archival of data needs to be implemented for the platform to be come a one-stop shop for all experiments.
- An independent metrics, monitoring and reporting system should be implemented for the Looking Glass components themselves to facilitate troubleshooting and resource management.
- Documentation and a set of test Kafka topics, connectors, elastic search indexes and Kibana dashboards/visualizations should be developed to give researchers a way to familiarize themselves with the platform and test the integration with their experiments.

In any case, Looking Glass has clearly demonstrated its ability to ingest, store, and allow the querying of millions of data points. Using this platform future work could attempt to trace the evolutionary process at a higher resolution. For example, population graphs can be stored before and after a round of mutation and used to compute the graph differences between them. The evolution of a single team and its learners mapped out across several generations and contrasted between different mutation strategies. In essence, with the tools developed in this work one will be able to guide further development of TPG and other custom RL algorithms.