EVALUATING SIMPLE REACTIVE AGENTS IN VISUAL REINFORCEMENT LEARNING TASKS

by

Caleidgh Grace Bayer

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

at

Dalhousie University
Halifax, Nova Scotia
August 2023

This thesis is dedicated to my father. His endless support, unrelenting positivity, and unconditional love have been invaluable throughout this process. Words cannot describe my immense appreciation and gratitude. Thank you, Dad.

Table of Contents

List of Tables

List of Figures

Abstract

Visual formulations of reinforcement learning tasks are potentially challenging because (1) the state space is large and composed from pixels (so unlikely to be directly correlated with actions), (2) the underlying task might be partially observable despite the high dimensionality, and (3) rewards can be sparse, so do not necessarily discriminate between useful and not useful decisions. In this thesis we compare the classic deep Q-network (a temporal difference reinforcement learning approach) with tangled program graphs (TPG) (a genetic programming approach) under complete and partially observable visual reinforcement learning tasks from ViZDoom. We demonstrate that TPG is particularly effective at imparting structure on the partially observable task (resulting in a general policy for navigating a labyrinth), but is relatively poor at solving a fully observable (aiming) task. Conversely, DQN is very effective when presented with the complete information aiming task, but is unable to discover general solutions to the partially observable navigation task. We attribute these preferences to the different approaches TPG and DQN assume for addressing representation/feature construction versus credit assignment.

List of Abbreviations and Symbols Used

DL Deep Learning.

DQN Deep Q-Network.

FPS First-Person Shooter.

SBB Symbiotic Bid-Based GP.

TPG Tangled Program Graph.

Glossary

FPS A first-person shooter is a type of game that is viewed from a first-person perspective. The player does not see their character's body. Instead, they see their character's weapon and possibly their character's arms holding the weapon. Popular examples include Call of Duty, Overwatch, and Left 4 Dead. In constrast, a third-person shooter is a game wherein the player can see their entire character. Popular examples include World of Warcraft, and Grand Theft Auto V. "FPS" can also mean "frames per second". This meaning of the acronym will not be used in this document to avoid confusion.

GZDoom A similar source port to ZDoom, largely defined by the addition of OpenGL support..

HP "Hit points", sometimes called "Health points" are a quantification of a character's vitality. Reducing a character's hit points to zero means the character dies (and despawns).

Map In the context of video games, the "map" is the space in which entities including players and enemies exist. Typically "map" refers to the "game board", usually represented in 2D space from a top-down perspective, rather than what the player sees on the screen.

SLADE3 Software used for both creating and editing ZDoom and GZDoom levels..

Spawn In the context of video games, "spawn" is a verb that refers to the instant an entity appears on the map. This can be combined with other terms to create a noun, such as "spawn location", meaning the place on the map where something has spawned. Another related term is "despawn", which is when an entity is removed from the game.

Tic "Tic" is another term for "frame". "Tic-rate" means the same thing as frame rate.

ZDoom A source port of the 1993 game Doom. It functions as the "underlying engine" in the context of this research..

Acknowledgements

I would first like to acknowledge Dr. Malcolm Heywood for his countless hours of mentorship, as well as those in the TPG research group: Robert Smith, Alexandru Ianta, and Ryan Amaral.

On May 29, 2023, wildfires in Tantallon forced me from my home. I would like to acknowledge those that helped during that time: the two police officers that rescued my pet parrots, the relatives that took me in, the friend that took my parrots in, the insurance representative that showed me kindness, and especially the firefighters and other first responders that risked their lives. Without them, nearly all of my research data would have been lost.

On a less serious note, the following deserve and acknowledgement for their positive contribution to my wellbeing: the band AJR, red wine, smoked cheddar, seasons 1 through 5 of Game of Thrones, Twitch streamers, the following games: Deep Rock Galactic, Escape Simulator, Unsolved Case Files, Shadowrun, Advanced Dungeons and Dragons 2nd Edition, Vampire Survivors, Guild Wars 1, Stardew Valley, and most importantly, the friends that take time out of their day to play those games with me.

Chapter 1

Introduction

Video games present a high-dimensional state space, creating a challenge for visual reinforcement learning. The challenge comes from the need to create meaningful features from the thousands of pixels present at each frame. Pixels in themselves are not effective features as they do not individually capture properties that correlate with measurable outcomes, i.e. decisions leading to actions. Instead, collections of pixels need identifying that represent objects that are meaningful for the underlying application. Deep learning (DL) represents one approach for developing low-dimensional representations from a high-dimensional state space. DL achieves this for visual state spaces using a combination of a convolution operator and bottleneck architecture, thus forcing the network to find a low-dimensional embedding. It is through the bottleneck architecture that it is able to find features that are useful for success in a visual learning task. Given the ability to develop a low dimensional representation, reinforcement learning approaches to gradient decent can then be applied (e.g. Q-learning [25]) in conjunction with speedups such as 'prioritized sweeping' to accelerate the credit assignment process relative to previously visited states. In this thesis we will adopt the Deep Q-learning (DQN) framework [21], where this represents a well known approach to deep reinforcement learning.

Now, consider approaching a high-dimensional space from the perspective of genetic programming (GP). One approach might be to provide the GP approach with a set of image processing operators to manipulate the original high-dimensional representation. Strong typing can be used to relate different instructions to different data types such that only high-dimensional data has image processing operators applied to them, e.g. [26]. Conversely, once scalar operands appear, inequalities and/or arithmetic can be applied in order to develop decisions relative to the low dimensional features. The goal would be to discover a set of objects for summarizing the original image content using a lower dimensional representation.

In this research we adopt a different approach. The goal is to construct a model incrementally where different programs index different subsets of the numerous pixels given. Ideally, the combination of pixels indexed plus the feature construction within a program does is sufficient for picking out the useful information; information that contributes to solving the task at hand. Previous research under visual reinforcement learning tasks has demonstrated that such an approach, when combined with the ability to stitch multiple such programs together into a graph of programs (tangled program graphs or TPG), is effective at building reinforcement learning agents under multiple high dimensional task domains, e.g. Atari 2600 console [10, 12], ViZDoom [13, 1], Dota2 [24].

A fully observable visual reinforcement learning task is a task wherein the player has complete state information; all of the information required to solve the task, e.g. chess or checkers. No information is hidden. In this thesis, we will be working with a fully observable task from the ViZDoom suite of tasks called "Basic". In this task, the player spawns on one side of a rectangular room, and a monster spawns at a random location along the wall on the opposite side of the room. In short, the player's goal is to shoot the monster. The monster itself is stationary, fully visible at all times, and the locations where the monster may be are also visible at all times. Deep learning frameworks, in theory, should be able to learn this type of task because it can find the object (the monster) with the use of its convolution operator, i.e. the DL architecture will learn a set of features that identify the monster independent of location. Genetic programs may struggle with this sort of task because it needs to know which pixel locations to index in order to locate the monster. Ideally, a GP solution would learn to create a "letterbox", a horizontal line of pixels spanning the width of the screen, thus identifying every location the monster could possibly be. However, this means that the monster has to appear in that line of pixels with equal probability. During testing, we found the spawn locations of the monster were not following an equal, or uniform, distribution, leading to further investigation. Discussion and results of this investigation are discussed in Chapter 2.

In contrast, a partially observable task is a task wherein the player does not have complete state information. For example, the task of escaping a maze is partially observable: there is no clear path out of the maze from the starting state. The

layout of the maze, the player's current location, and the location of the exit, are all considered "hidden" information. Furthermore, given a first-person point of view, information available is ego-centric. The information the agent has is dependent upon the direction it is facing, i.e. the agent experiences state through a limited field of view relative to the structure of the environment.

In this work, we will be utilizing another ViZDoom task called "My Way Home" in order to investigate the impact of partial observability on visual reinforcement learning in a partially observable environment. In short, a player spawns at a random location in a labyrinth, consisting of several rooms connected by corridors. The player must navigate to a goal location in the maze to succeed. The labyrinth in My Way Home is not stochastic; the layout of the maze, the relationship between rooms, and the location of the goal is static. However, both our deep learning and genetic programming approaches were not provided with memory, limiting them to purely reactive behaviours (i.e. no concept of previous state). We have already seen that agents with access to some form of long term memory are able to complete both incomplete and complete information tasks [13] [1]. To what degree will these two types of agents solve this task and solve it in a structured way? What about the aforementioned fully observable task? What changes, if any, will we see? Expanding on this idea, we will also measure the impact that a biased ViZDoom random number generator has on the performance of these separate approaches to visual reinforcement learning when faced with a uniformly distributed random number generator. Specifically, this thesis poses the following hypothesis:

- Under the complete information task (Basic) we anticipate DQN performing better than TPG. DQN was previously benchmarked under the Basic task and found to be effective [27]. TPG has also been shown to be effective at the Basic task [23], but only when subject to a training 'curricula' that exposed TPG agents to a multitude of different tasks. We believe the diversity in task exposure to have a significant impact on TPG's ability to develop appropriate behaviours, as without such exposure, effective indexing schemes will not be promoted. Conversely, the convolution operation assumed by DQN effectively represents an exhaustive search for objects in the visual space irrespective of location. We anticipate biased and unbiased sampling of agent spawn points to

improve the TPG results more significantly, but likely not to a level sufficient to match that of DQN.

- Under the incomplete information task (My Way Home) we anticipate TPG performing better than DQN. The basis for this hypothesis lies in the different approaches for reward accumulation assumed by TPG and DQN. TPG attempts to maximize the average sample returns as experienced across the entire task episodes. Put another way, reward under TPG is only experienced *after* encountering a terminal condition, thus credit assignment is only performed relative to the total episodic cost/performance. Conversely, DQN is based on Q-learning (part of the temporal difference family of reinforcement learning algorithms), thus credit assignment is performed per reward. The model at the end of the episode is different from that at the start. This can result in rather more greedy credit assignment behaviours. Under the My Way Home task this might mean that DQN is very efficient at finding the most direct path to the terminal condition, when the goal condition is visible. However, when the goal state is not visible, some sort of search strategy will need to be assumed. TPG on the other hand has to work harder to develop features that recognize the terminal condition, but can construct simple navigation heuristics symbolically from small subsets of features. We again anticipate biases in the sampling of agent spawn points will have more impact on TPG performance than DQN.

This thesis develops the above evaluation in the following way: Chapter 3, the genetic programming framework being utilized is explained. Chapter 2 provides necessary background information regarding the ViZDoom framework and task scenarios. There is also discussion about the problematic random number generator in ViZDoom and the "fix" utilized in this research. In Chapter 4, experimental design and changes made to the genetic programming framework assumed are both detailed. Chapter 5 provides results, analysis, and discussion of the experiments mentioned in the previous chapter. Finally, Chapter 6 provides the conclusion.

Chapter 2

ViZDoom Background

2.1 ViZDoom Introduction

ViZDoom [27] is an artificial intelligence learning platform based on the 1993 first person shooter computer game, Doom. The ViZDoom platform allows artificial intelligence agents to interact with the Doom environment by facilitating the transfer of visual buffers (what a human player would see on a screen) from the Doom environment to an agent, and facilitating the transfer of actions (akin to buttons on a keyboard being pressed) from the agent to the Doom environment. A singular visual buffer is equivalent to a frame, also called a tic. The terms "tic rate" and "frame rate" are used interchangeably. ViZDoom comes with several tasks, or "mini-games". Each task is designed to teach the agent a certain behaviour. Examples of some of these behaviours are dodging enemy fire, aiming and shooting at enemies, and conserving ammo. The ViZDoom platform calls each iteration of a minigame an "episode". The end conditions for an episode vary from task to task. Common end conditions include:

- The player wins.
- The player dies.
- A fixed number of tics have passed (timeout).

2.1.1 Task Scenarios

ViZDoom comes with eight "standard" tasks, each meant to train agents to perform a particular skill. In this research, we will be focusing on two of these tasks: "Basic" and "My Way Home" where these are representative of complete and incomplete information tasks respectively.

Basic

The goal of Basic is to train an agent to be a fast and accurate shooter. At the start of the task, the player Spawns in the center on one side of a rectangular room, and a type of monster, known as a Cacodemon, spawns on the other side of the room, facing the player. The monster's spawn location along the y-axis is random, but it always remains at the same position on the x-axis, maintaining a constant distance from the player. The monster is unable to move. The player is able to move left and right. Figure 2.1 shows what the player sees at the start of an episode. Figure 2.2 shows the top-down map view, illustrating where the player spawns and where the Cacodemon can spawn.

The player is rewarded for shooting the monster quickly and conserving ammunition. The player starts with a reward of 0. One point is subtracted per tic and five points are subtracted per bullet fired. If the player successfully shoots the monster, 106 points are awarded. The episode times out after 300 tics. Given the travel time of the bullet, the highest possible score is 95 points. The lowest possible score is -410. While the player is given a clip containing 50 ammo, it is impossible to unload all of the ammo before the timeout occurs. Removing points for each bullet fired incentivizes accuracy, while removing a point per elapsed tic incentivizes speed.

My Way Home

My Way Home is a navigation task. The map consists of eight rooms and ten hallways. The goal of the task is to locate an armour pack as quickly as possible. The armour pack is located at the end of the rightmost hallway. Figure 2.3 shows the armour pack circled in white. The armour pack is always in the same location, however, the player spawns in at one of 17 possible spawn locations. Each spawn location has a unique identification number. The first possible location's identification number is 10, and the last location's identification number is 26. The location is randomly selected. The direction the player is facing is also randomly selected. Each room has a unique pattern on the walls, while the hallways all share the same brick pattern on the walls, floor and ceiling. The unique wall patterns should, in theory, help the player figure out where they are on the map.

Figure 2.1: Player point of view in the Basic task.

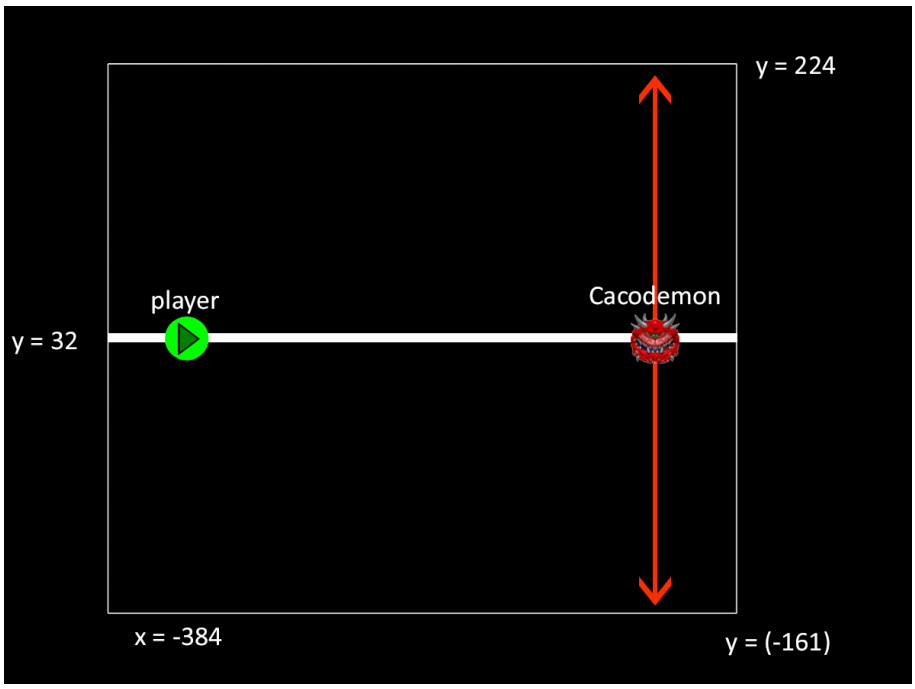


Figure 2.2: Top-down view of the Basic map, illustrating key areas on the x and y axes.

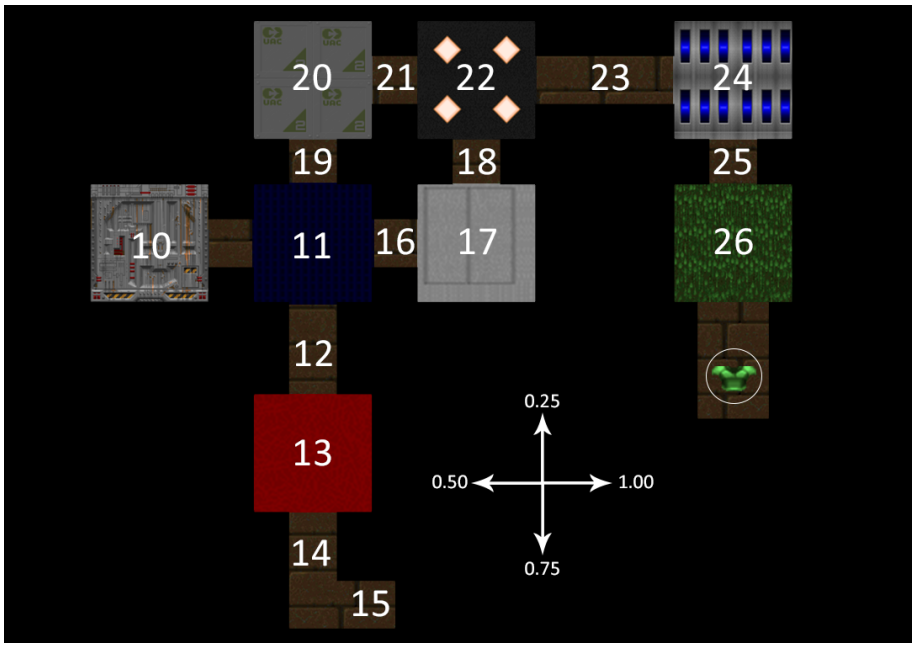


Figure 2.3: Top-down view of the My Way Home map. Each number represents one possible spawn location. A compass is shown on the bottom right.

The player begins each episode with a score of 0, whereas 0.0001 points are removed from the score for each tic that passes. This incentivizes speed. 1 point is rewarded upon finding the armour. This incentivizes finding the armour pack. The episode ends when the player finds the armour, or after 2100 tics have elapsed, whichever comes first.

2.2 ViZDoom, ZDoom, WAD Files

In order to facilitate understanding of the random number generator issue (source of a learning bias), we have to first unpack the intricacies of the ViZDoom platform itself.

2.2.1 ZDoom/GZDoom

ZDoom is a source port for Doom, meaning a platform on which users can create their own custom Doom levels. ZDoom supports modern hardware and modern operating systems. For our purposes, we can consider GZDoom and ZDoom to be equivalent. GZDoom simply provides OpenGL support.

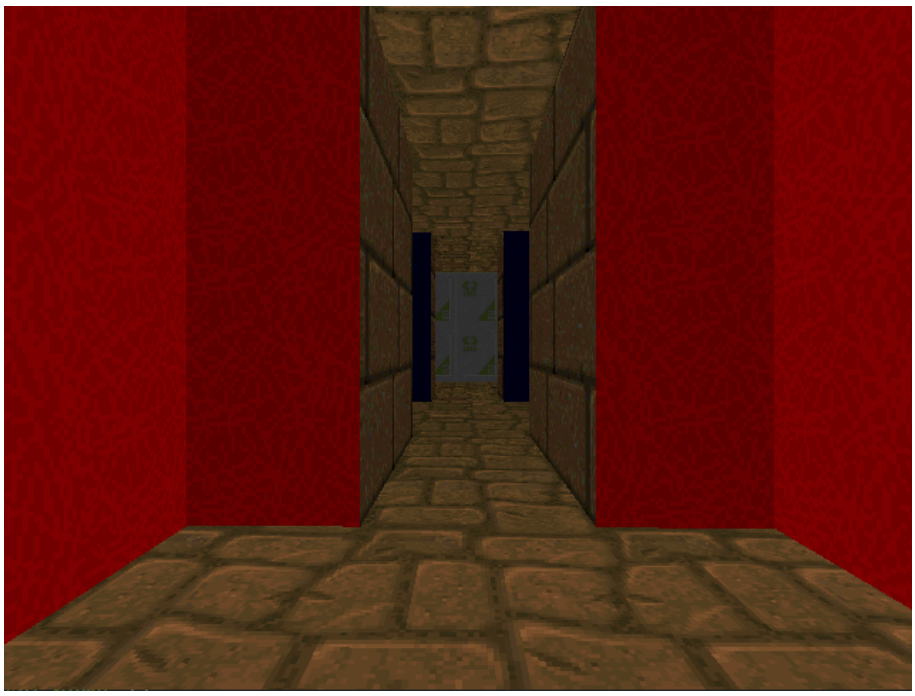


Figure 2.4: Player point of view in the My Way Home task at spawn location 14, facing 0.25.

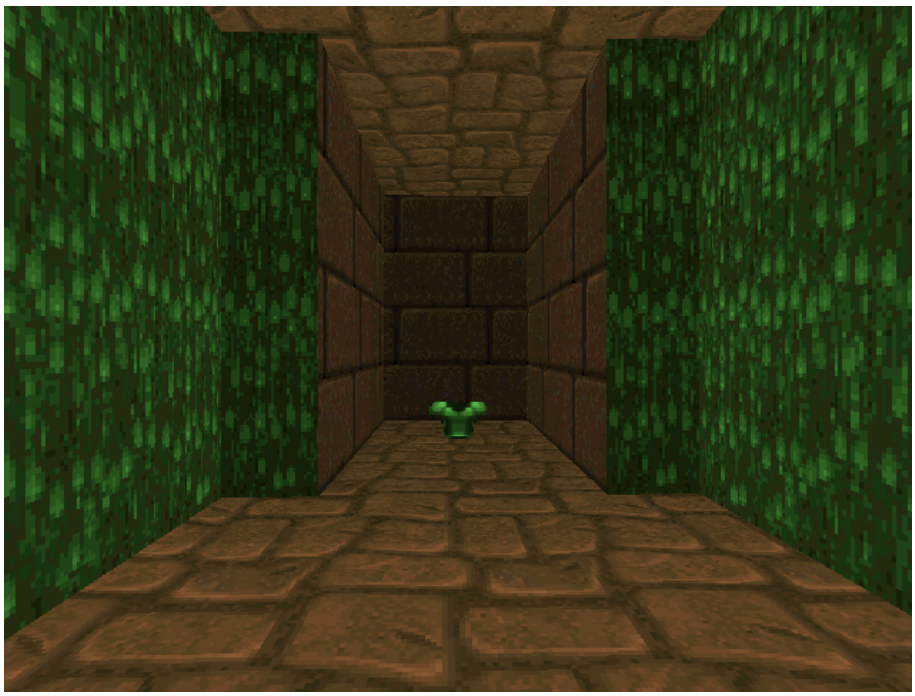


Figure 2.5: Player point of view in the My Way Home task at spawn location 27, facing 0.75.

Some definitions:

- ACS: ACS (Action Code Script) is a C-based scripting language. ZDoom provides its own custom implementation and compiler for ACS.
- Actor: Within a game environment, an actor is an entity that interacts with the environment around it. All monsters and the player character are classified as actors.

Custom games are facilitated through the creation of WAD files (or "**W**here's **A**ll the **D**ata?"). WAD files contain several components; the components relevant to this work are as follows:

- At least one Map Marker file: Map Marker files provide a visual representation of what the map will look like when rendered. Walls, floors, ceilings, enemy spawn points, player spawn points and item spawn points are all defined in this file, as well as the textures those items will be rendered with.
- At least one ACS Script file to define the behaviour and resources of entities on the map. This is typically limited to actors, but doesn't have to be. The following values must be defined within this file:
  - Monster spawn points, spawn types, and spawn timing.
  - Monster details, such as the monster's aggression towards the player, aggression towards other monsters, movement speed, hit points, weapons, and available ammo, if applicable.
  - The (x, y) spawn location of the player.
  - The spawn angle of the player (which direction they're facing).
  - The player's hit points, movement speed, weapons, and ammunition available.
  - Other variables and logic can be programmed in this file as well if desired by the programmer.

2.3 ViZDoom

As stated in section 2.1 ViZDoom (Vi-ZDoom) is a platform that allows a client, usually an agent, to communicate with the ZDoom environment. ViZDoom also provides their own WAD files for defining task scenarios, as well as their own ACS library on top of the custom ACS variant included with ZDoom. This creates some confusion when trying to figure out where given functions are defined. For example, both `random()` and `Random()` compile just fine. One accepts fixed-point values, while the other will fail gracefully.

2.4 Problematic Random Number Generator in ViZDoom

It was anecdotally noticed several times during the course of ViZDoom research that SBB and TPG agents seem to prefer one direction or another, even with the usage of action programs. This apparent bias was most noticeable in "Basic", where the agent always seemed to move to the left, even when the monster spawned on the right. In particular, the results in Figure 2.6 sparked both curiosity and concern. In order to rule out an error in the implementation of action programs, the ViZDoom WAD files were carefully analyzed using SLADE3, a well-known development environment for ZDoom. SLADE3 is open source, and allows programmers to create every component of a WAD file, including maps and scripts, and allows programmers to edit existing WAD files. It also allows programmers to run ZDoom and play through their levels to test them. The usage of SLADE3 was integral to this research, as it allowed for the unpacking, editing, and analysis of the WAD files that come with ViZDoom, without interacting with the ViZDoom client itself. It is important to note that SLADE3 operates independently of the ViZDoom environment, but rather runs on the underlying ZDoom engine.

Specifically, we wanted to test the distribution of spawn locations of the Cacodemon in "Basic", as this was the task that appeared to have a bias. While some of the ZDoom variant of ACS documentation is adequate, other portions of the documentation are very poor. As such, even retrieving the (x, y) coordinates of the monster spawn point was challenging, especially without the help of the ViZDoom client. The ZDoom flavour of ACS does not directly support file I/O. With the help of SLADE3,

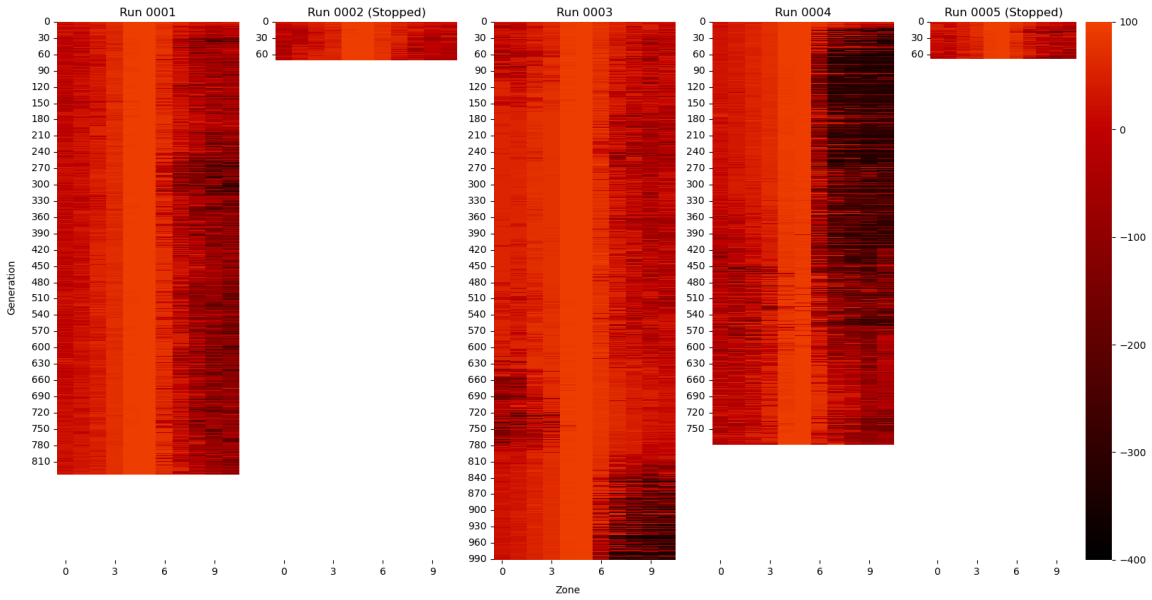


Figure 2.6: A series of heatmap depicting the best score in each zone for several experimental TPG runs. The x-axis of each heatmap represents the y-axis of the Basic map from a top-down perspective. The y-axis represents the temporal element; population evolution over time, with earlier populations at the top and later populations at the bottom. The experiment was ultimately abandoned, but the bias in scoring remains clear.

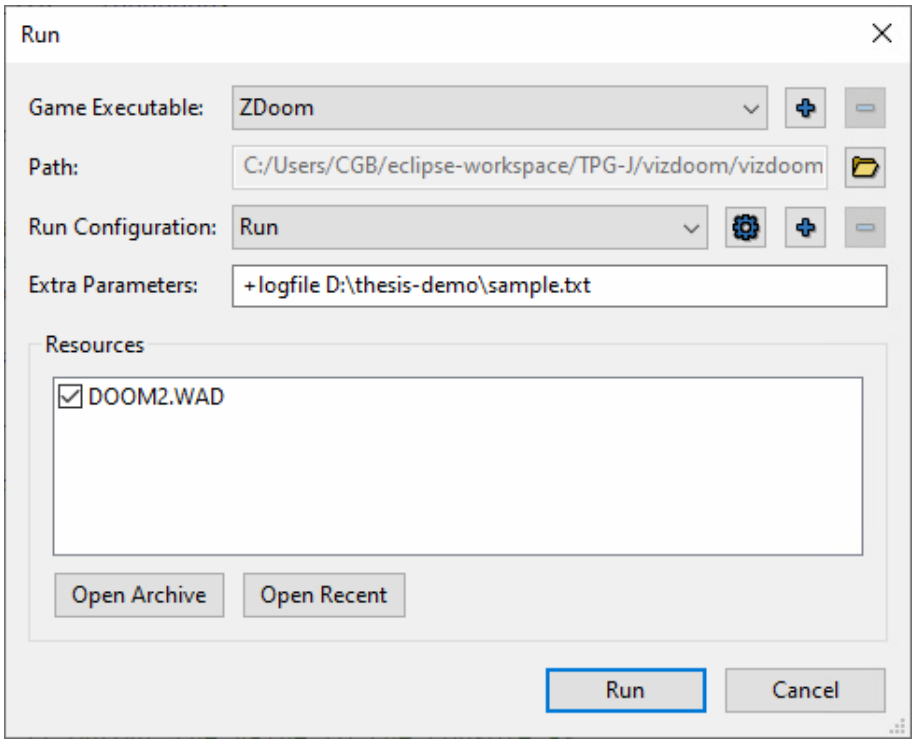


Figure 2.7: Execution parameters in SLADE3 required to store console output to a file.

through a very roundabout method, the distribution of the numbers returned by the random number generator could be tested. First, the desired value has to be logged within the ACS script to the debug console. Then, the simulation has to be manually started with a console command in the execution parameters to tell the engine to store the console output to a file. There is no documentation from SLADE3 or ZDoom regarding storing the console output to a file. It does not use the standard Linux/Windows redirection character (>). However, a useful post from an online forum provided the necessary pointers.[1] Figure 2.7 shows the execution option screen in SLADE3 with the "Extra Parameters" field populated with the console command required to store output to a file. `+logfile` is the command, `D:\thesis-demo\` is the path where we'd like to store the output file, and `sample.txt` is the name of the output file, which can be changed as desired.

The ZDoom game engine has to operate at frame rate. Thus, only a finite number of instructions can be performed per tic. It was empirically discovered that the game

[1] `https://forum.zdoom.org/viewtopic.php?f=3&t=35956`

engine's limit was 5000 calls to the random number generator per tic, then a 1 tic delay must be performed. This is reflected in the sample code provided in Listing 2.1.

```
#include "zcommon.acs"

int target_id = 10;

// Initialize the reward variable.
global int 0:reward;

// This script executes first.
script 1 OPEN
{
    // The number of trials to run.
    int trials = 10000000;

    // How many random numbers we
    // can generate in a tic.
    int per_tic = 5000;

    // Get the number of repetitions we'll
    // need in order to achieve the desired number of
    // trials.
    int repeat = trials / per_tic;

    for(int j=0; j<repeat; j++)
    {
        delay(1);

        for(int i=0; i<per_tic; i++ )
        {

            // Generate a random number between
            // -161 and 224, the possible spawn
            // loactions for the Cacodemon.
            int r = random(-161.0,224.0);

            // Output the value to the console as
            // a float.
```

```
37              log(f:r);
38
39
40              // Generate a random number using the
41              // other random function.
42              //int r = Random(-161.0, 224.0);
43              //log(f:r);
44
45          }
46      }
47      // Exit the game.
48      Exit_Normal(0);
49  }
```

Listing 2.1: The ACS code used to generate values from the random number generator so they can be analyzed.

A 10 million trial experiment was ultimately performed using `Basic.WAD` with the `SCRIPTS` file edited as shown in Listing 2.1. The resulting data was stored and analyzed with Python and Matplotlib. Figure 2.8 plots the resulting 10 million monster y-axis spawn points $(-161, 224)$ as a histogram-frequency plot. It is apparent that a uniform distribution is approximated; a far cry from what had been manually observed in the ViZDoom environment. These results imply that the random number generator provided by ZDoom is not causing monsters to be initialized with a strong left of center bias. Both the `random` and `Random` functions were tested, and both yielded similar results.

The next step was to test the random number generator again, but utilizing the ViZDoom client, rather than testing with SLADE. Knowing that ViZDoom would send a reward value, the ACS script was changed to store the y-coordinate of the monster (under the Basic task environment) as the reward instead of the original reward. This allowed for the seed counting experiment to be performed using Java. The same experiment was performed on another computer with a new install of Windows 10, the latest Python version of ViZDoom, installed using Pip, as per the instructions to ensure that all bases were covered in regards to versioning. Figure 2.9 conclusively demonstrates a severe bias towards a particular spawn point, slightly left

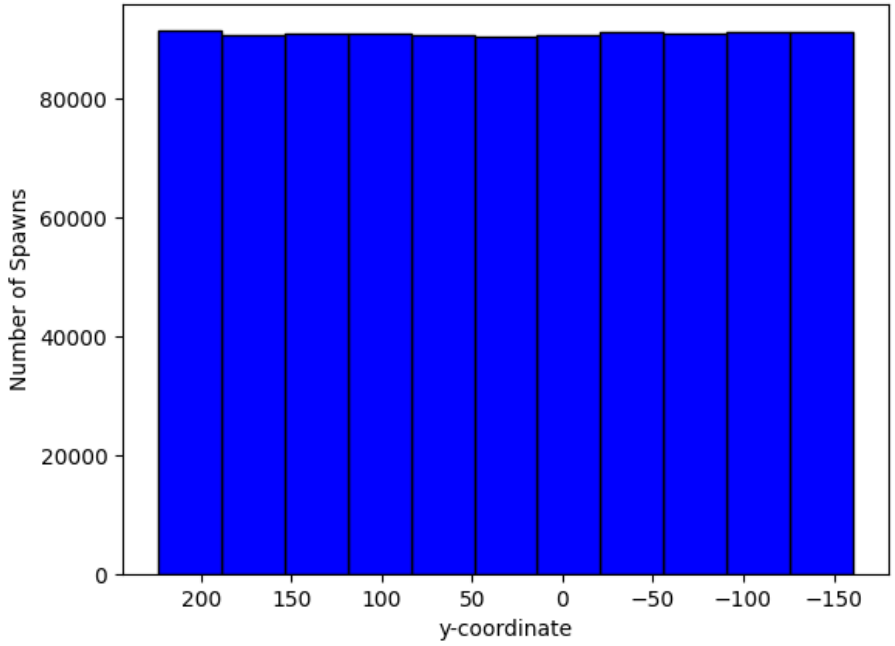


Figure 2.8: Basic spawn testing on Windows 10 with ZDoom only via SLADE 3. Histogram reflects the spawn position as sampled over 10 million calls to the spawn routine

of centre.

Looking at the 'known ViZDoom issues' on the GitHub repository, one user reported an apparent bias in ViZDoom's random number generator in 2017.[2] It is ultimately found that this is a Windows-only problem, but nothing seems to be done about it. It is noted that the issue is not reproducible in Linux. To verify that this is a Windows issue, the same experiment was executed on an Ubuntu VM, using the Python version of ViZDoom. The results were a mostly uniform distribution, which is shown in Figure 2.10.

To further verify this issue, similar testing was performed using a modified version of the My Way Home WAD file. The file was modified in a similar fashion. Figures 2.11 and 2.13 illustrate the results of performing these tests on Windows. It is evident that we have a very uneven distribution of values. Figures 2.12 and 2.14 show the results of the same test performed on Ubuntu. For transparency, these two tests used a much smaller sample size than the tests performed for Basic. The discrepancy in

[2] `https://github.com/mwydmuch/ViZDoom/issues/170`

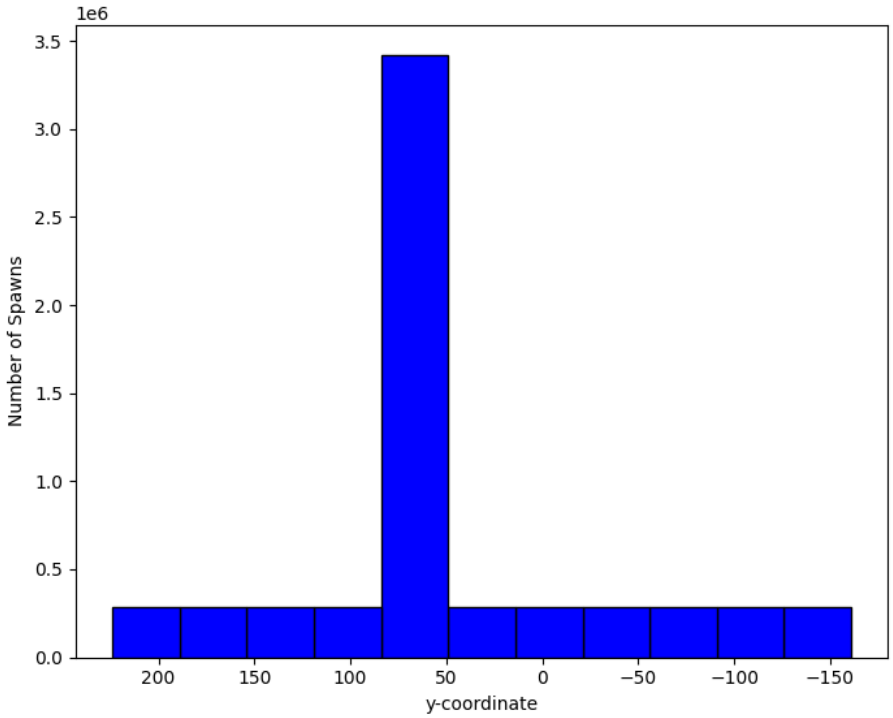


Figure 2.9: Basic spawn testing on Windows 10 using the ViZDoom client. Histogram reflects the spawn position as sampled over 10 million calls to the spawn routine. Note the y-axis scale $\times 10^6$

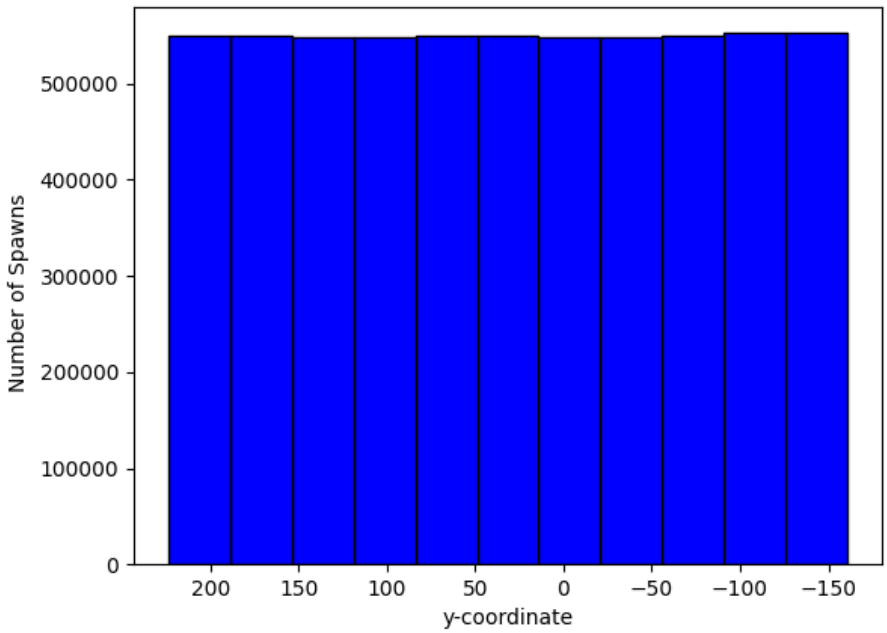


Figure 2.10: Basic spawn testing on Ubuntu VM using the ViZDoom client. Histogram reflects the spawn position as sampled over 6 million calls to the spawn routine

sample size was due the constraints on the available hardware at the time of testing.

This further solidifies the notion that the random number generator is only broken in Windows. The aforementioned GitHub post briefly brings up a theory that re-building the ViZDoom environment fixes the problem. However, building the environment is not listed as a requirement for using ViZDoom on Windows. In fact, in recent years, the Python distribution of ViZDoom has been added to 'The Python Package Index' (PyPI), which allows Windows users to install a pre-built version of ViZDoom using `pip`, a command line tool for simple installation of Python libraries. Making ViZDoom easy to install is useful, but many users may then believe that a library installed via pip is ready to use. If that is not the case for Windows users, then it needs to be part of their installation tutorial.

Given this background, I set out to correct this issue myself. I developed a sufficiently simple, but seemingly effective fix for the Basic scenario. Given the width of the Cacodemon, there were effectively 11 slots on the map where it could spawn. I created 11 different WAD files, each allowing the Cacodemon to spawn in only one slot. Then, I used the random number generator in Java to randomly select one WAD

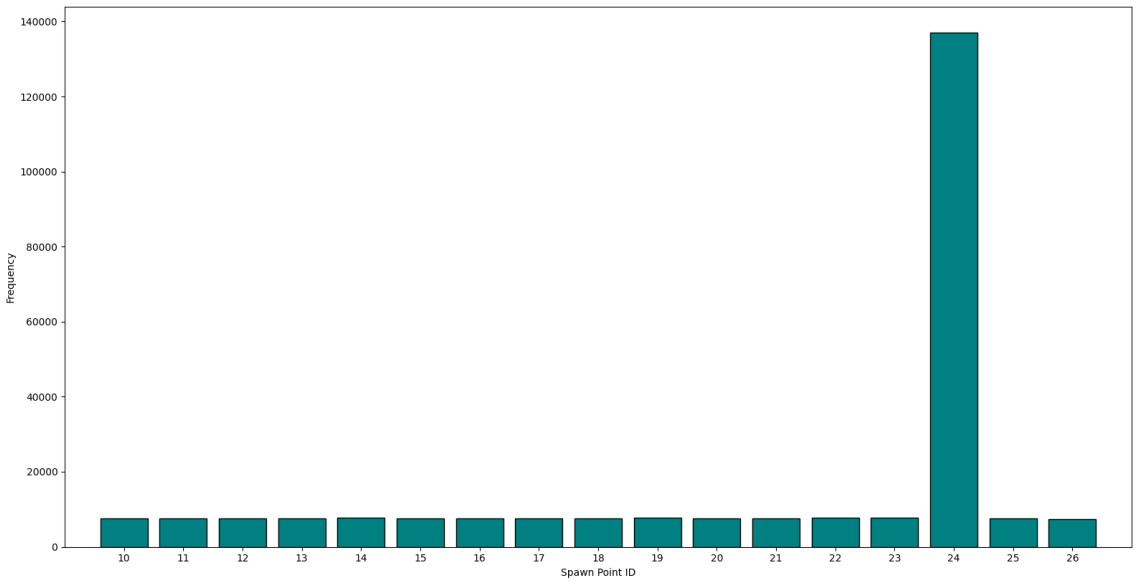


Figure 2.11: Histogram depicting the frequency of room spawns with ViZDoom on Windows for the My Way Home task.

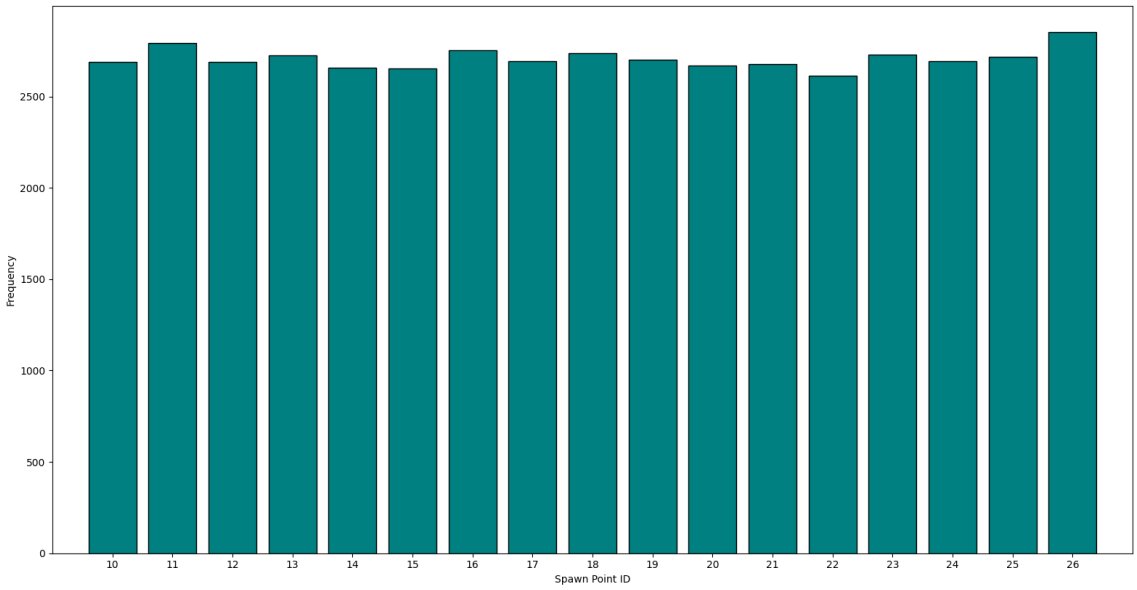


Figure 2.12: Histogram depicting the frequency of room spawns with ViZDoom on an Ubuntu VM for the My Way Home task.

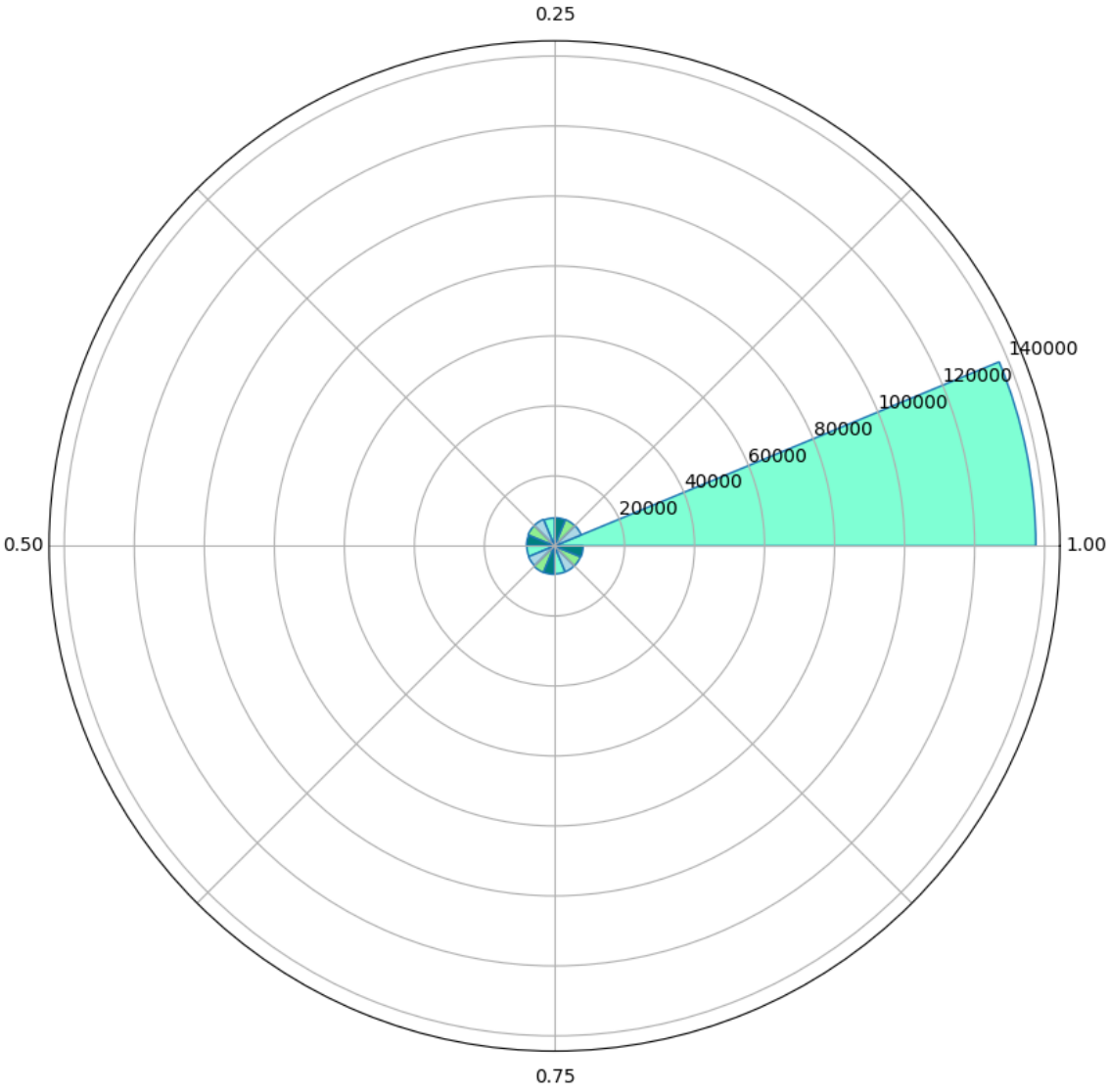


Figure 2.13: Polar histogram depicting the direction the players spawns facing in the My Way Home Task, using Windows and ViZDoom.

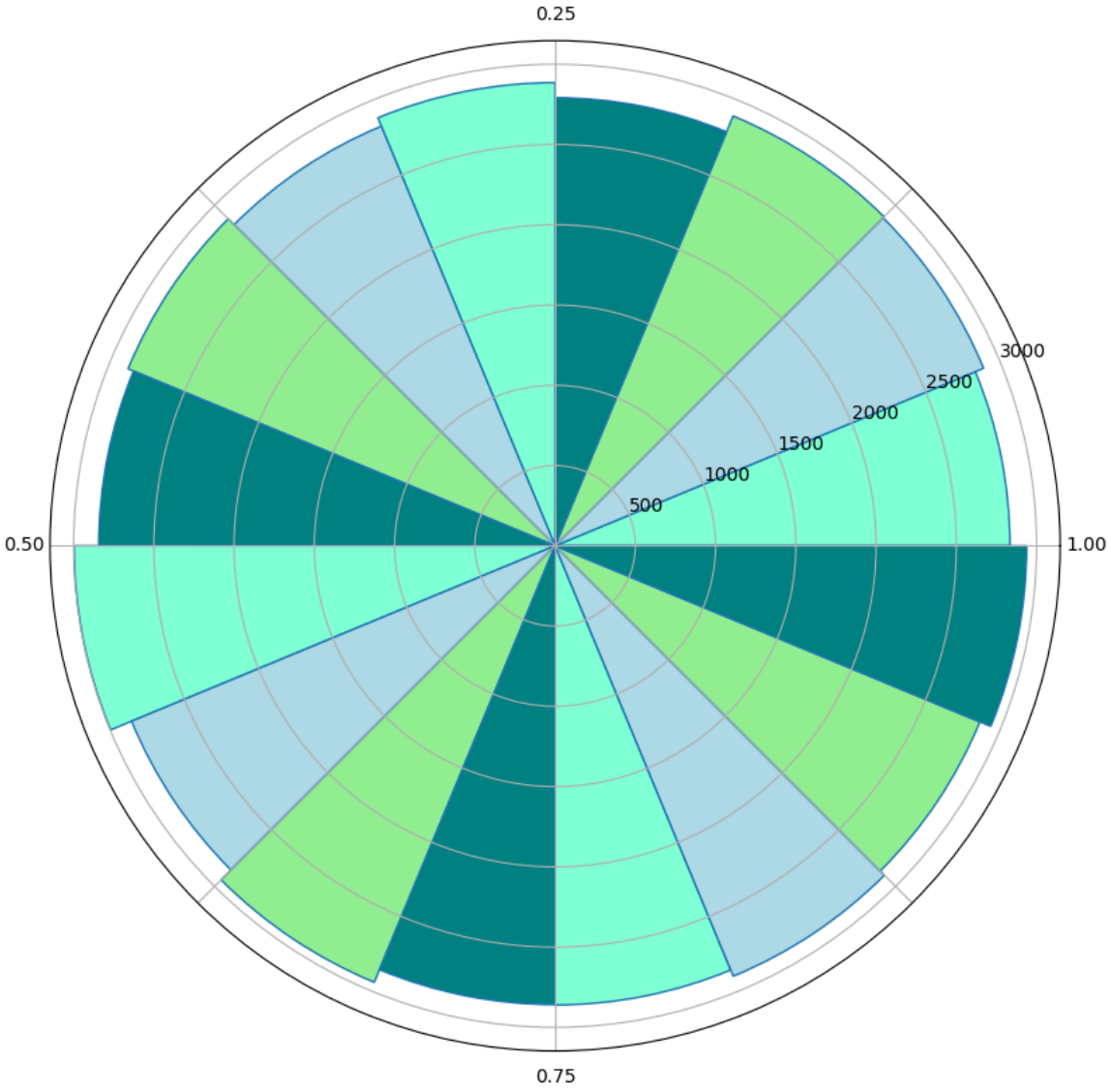


Figure 2.14: Polar histogram depicting the direction the players spawns facing in the My Way Home Task, using an Ubuntu VM and ViZDoom.

file to show to an agent per episode. This technique seemed to work, but was not extensible to tasks like My Way Home, where the players spawns at a random angle between 0 and 1. This technique wouldn't work on tasks like Defend the Center, either, where infinite monsters spawn at random points.

2.4.1 Correcting the Monster seeding under Windows 10

In order to facilitate the transfer and usage of a value from the ViZDoom client to the ZDoom engine, both the WAD file and the client-side code have to be modified. The WAD file must have a variable in which to hold the value, a subroutine in which the value is set, and then that value has to be used. Modifications to basic.WAD can be found in Algorithm 2.2. Scripts and functions that were unchanged have been replaced with '...' for readability while maintaining reproducibility.

```
#include "zcommon.acs"

int target_id = 10;

global int 0:reward;
global int 1:USER1;
global int 2:USER2;

script 1 OPEN
{
    reward = 0;
    USER1 = 0;
    USER2 = ACS_ExecuteWithResult(5, 0, 0, 0);

    SpawnTarget();

}

int c =0;

script 2 ENTER
{
   ...
}
```

```
25
26 script 3 (void)
27 {
28     ...
29 }
30
31 script 4 (void)
32 {
33     ...
34 }
35
36 function void SpawnTarget(void)
37 {
38     Spawn("Cacodemon", 0.0, USER2,0.0,target_id,128);
39
40     USER1 = GetActorY(target_id);
41
42     ...
43 }
44
45 script 5 (void)
46 {
47     SetResultValue(GetCVar("monsterspawn") + 0.01 - 0.01);
48 }
```

Listing 2.2: Modifications made to basic.WAD to allow intake of custom values from ViZDoom client.

The following changes were made, reflected in Listing 2.2:

- The addition of two global variables on lines 6 and 7, `USER1` and `USER2`. The names of these variables **must** be as shown. `USER1` is where we will store the spawn location of the monster after it spawns. This allows us to verify that our value was in fact used from the ViZDoom side. `USER2` is where our custom value will be stored.

- The addition of a script, starting on line 45, that takes the custom value from the client and stores it in `USER2`.

- Then, we have to call this script at the beginning of level execution. This happens in the opening script, on line 13.
- Finally, we have to force the Cacodemon to spawn where we tell it to, on line 38. Then, on line 40, we get its y-position and set `USER1` to that value. This allows us to verify that the Cacodemon spawned where we told it to from the client-side.

Next, we will discuss the code required on the VizDoom side, in both Python and Java, in the interest of reproducibility.

```
1  # Set the minimum and maximum values for our
2  # random number generator.
3  min = -161
4  max = 224
5
6  # Generate random double value from -161 to 224
7  # with 5 decimal places.
8  rand_spawn = round(random.uniform(min, max), 5)
9
10 # Multiply by 65536 to convert to ZDoom floating point.
11 rand_spawn = rand_spawn * 65536;
12
13 # Create a string to hold our console command.
14 cmd = "set monsterspawn " + str(rand_spawn)
15
16 # Send the console command to the game.
17 game.send_game_command(cmd)
18
19 # Start a new episode.
20 game.new_episode()
```

Listing 2.3: Python ViZDoom client-side code for setting a custom monster spawn location in Basic

```
1  // Set the minimum and maximum values for our
2  // random number generator.
3  int min = -161;
4  int max = 224;
```

```
5
6  // Create a new instance of a random number generator.
7  Random r = new Random();
8
9  // Generate our random number.
10 float randSpawn = min + r.nextFloat() * (max - min);
11
12 // Multiply by 65536 to convert to ZDoom floating point.
13 randSpawn = randSpawn * 65536;
14
15 // Create a string to hold our console command.
16 String cmd = "set monsterspawn " + String.format("%.5f", a);
17
18 // Send the console command to the game.
19 game.sendGameCommand(cmd);
20
21 // Start a new episode.
22 game.newEpisode();
```

Listing 2.4: Java ViZDoom client-side code for setting a custom monster spawn location in Basic

The "key" to sending a value from the VizDoom client to the ZDoom engine lies in the `send_game_command` function. The comments in Listings 2.3 and 2.4 describe what each line of code is doing. In short, we use Python's (Java's) random number generator to obtain a value that is much closer to truly random. Then, we build a console command as a string. The command begins with the `set` command. The `set` command takes two arguments. The first argument is an arbitrary variable name. This variable name has to be the same as the string found on line 47 of Algorithm 2.2. As per good coding practices, descriptive variable names should be used where possible.

This function is poorly explained in the ViZDoom documentation found on their GitHub page. Figure 2.15 shows a screen shot from the ViZDoom GitHub documentation at the time of writing. There is no description of syntax, no list of accepted commands, no inputs or outputs. The ZDoom documentation linked to by the ViZDoom documentation *is* more comprehensive, but at no point does it provide

Figure 2.15: A screenshot of the VizDoom documentation provided for the send_game_command at the time of writing

a complete list of valid console commands. At the very bottom of the CVAR page, there is a brief mention of the `set` command, with no example usage provided. That mention links to yet another page, that provides a list of console commands specific to game customization. These console commands were not included on the ZDoom page about console commands.

2.4.2 ViZDoom Environment Summary

The ViZDoom environment represents a very widely employed game engine for evaluating visual reinforcement learning algorithms under a first person perspective. On the positive side, there are a wide range of opportunities for designing custom environments, thus facilitating the development of experiments designed to test specific properties of learning agents. On the other hand, we encountered biases in spawn point generation that introduced corresponding biases into the behaviours of our learning agents. We were, however, able to correct these limitations using some of the tools provided for customizing the game engine.

This thesis will use two 'standard' task scenarios (Basic and My Way Home) from the ViZDoom game engine as examples of complete and incomplete information reinforcement learning tasks described under visual (first person) state. Previous research using these task scenarios appears not to have recognized the biased nature of entities reliant on a random number generator, such as spawn points, in part because (1) independent testing of post training performance was not performed and (2) the visual reinforcement learning agents appear to assume some form of deep

reinforcement learning. As will later become apparent, this is a both a strength and a weakness.

Chapter 3

Tangled Program Graphs and Deep Q-Networks

3.1 Genetic Programming

Genetic programming is a subtype of evolutionary algorithms [16]. Genetic programs are built using assembly language.

In the book, A Field Guide to Genetic Programming, genetic programming is defined as: "Genetic programming (GP) is an evolutionary computation (EC) technique that automatically solves problems without requiring the user to know or specify the form or structure of the solution in advance." [22], note that "evolutionary computation techniques" are also known as evolutionary algorithms. In this context, a "program" is a series of assembly instructions. A main idea of genetic programming is realized through the genetic programming cycle. The genetic programming cycle can be viewed as a series of steps; Initialization, Evaluation, Selection, and Reproduction. Figure 3.1 provides a visual flowchart of these steps.

First, during the initialization step, a group of randomly generated programs is created. This group is called a population, and each program is called an "individual". Unlike most machine learning approaches, this means that multiple candidate solutions are maintained. A competition therefore takes place between the individuals of the population for 'survival'. As will become apparent, the concept of credit assignment therefore acts both on individuals and the population as a whole.

Next, each program in the population is evaluated, using a fitness function. A fitness function is a mathematical equation that quantifies how well an individual performed at a task. For example, the fitness function for a team in a game of soccer is, generally, how many times the ball went into the opponent's net. The result of the fitness function is known as a score. The score is saved with the individual.

Then, we decide which individuals to keep, and which individuals to delete from the population, based on their score.

Following selection, we create new individuals, through reproduction. Two types

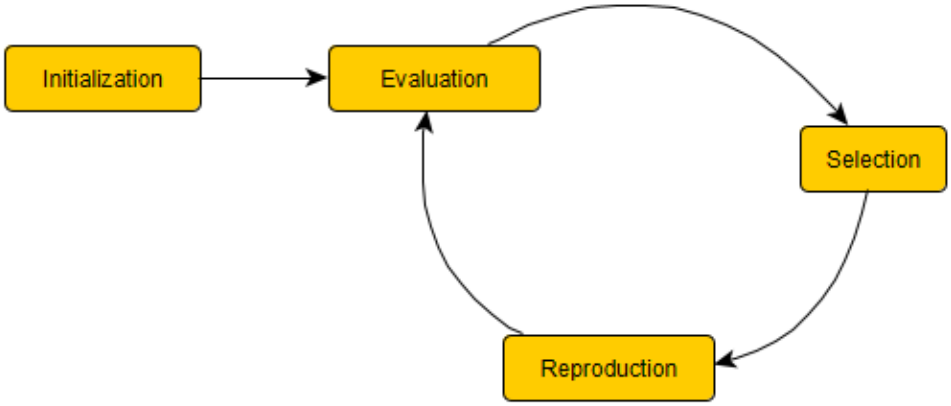


Figure 3.1: An illustration of the genetic programming cycle.

of reproduction often seen are crossover and mutation. In crossover, two parent individuals are selected, and parts of each parent program are randomly combined into a new, child program. In mutation, one parent program is selected, then duplicated to create a child program. Parts of the child program are then chosen at random, and those chosen parts are changed at random.

Finally, the cycle begins again by evaluating our new population of programs.

3.2 Team-Based GP

Team-Based GP is a variant on the standard GP structure. First proposed in 1995, Team-Based GP was based on the idea of a "team" of programs; grouping of individuals into a team [7]. With the introduction of teams of programs came two questions:

- How do we fairly distribute a score across all members of a team (credit assignment)?
- Similarly, how do we select programs for use in reproduction, given that their score may not be representative of that program's performance?

To solve the problem of credit assignment, it was proposed that the term "individual" should refer to a team, rather than a singular program, and that teams would be treated as individuals within the population. Scores were assigned to a team, instead of being assigned to a single program. This removed the problem of figuring out how to distribute a score across all team members [7].

There are two different ways to implement teams; a heterogeneous team and a homogeneous team. In a heterogeneous team, each member of the team has its own program. In a homogeneous team, every member of the team has the same program, as in, a copy of the same program is made for each team member. It was found that, in a game environment where the player had simultaneous control over **several** player characters, homogeneous teams outperformed two sets of heterogeneous teams, where each set had a different reproduction technique applied [7]. However, it was later noted that, in regards to homogeneous teams, "...there is nothing to be gained from the combination of the outputs of completely identical programs" [3]. Much of the following work focused on the evolution of heterogeneous teams, despite the fact that heterogeneous teams did not perform as well [7].

The next notable advancement in this area was the idea of deriving a single output from a team of programs for use in classification and regression tasks. Some combination techniques proposed included averaging each of a team member's outputs and averaging the outputs but with a fitness-based weight value for each team member. However, it was found that another combination technique, titled "Winner Take All" (WTA) performed best on both classification and regression tasks with binary output (0 and 1). In WTA, a singular program is chosen to provide an output for the entire team, based on the program's confidence in its prediction. Because of the binary nature of the possible outputs, it was hypothesized that the closer a program's output value was to either 0 or 1, the more confident the program was in its prediction [3]. WTA proved to be the best in an experiment where it was compared against eight other combination techniques, with a training classification error of 0.02% and a testing classification error of just 0.33% [3].

3.2.1 Symbiotic Bid-Based GP

Another incremental improvement to the Team-Based GP framework, which was later titled "Bid-Based GP" [18], was the conceptualization of "actions". Actions were needed in order to perform multi-class classification tasks (as opposed to a binary classification task or a regression problem). Bid-Based GP implemented both heterogeneous teams, the WTA combination technique, and actions. In order to properly define actions, more descriptive terminology was defined with the invention of Bid-Based GP:

- Learner: A member of a team.
- Bidding: the act of executing a learner's program.
- Bid: the value produced by executing a learner's program.

Actions came about because learners needed a way to translate their bid to a non-numeric value. In the case of a multi-class classification task, that non-numeric value is class label. At initialization, each learner is assigned a class label as their action. During evaluation, each learner executes their bid program and produces a bid. The learner with the highest bid is then allowed to suggest its assigned action. This action can only be changed through mutation, during the reproduction phase. Bid-Based GP was tested against three datasets. One dataset had three class labels, the second had two class labels, and the third had five class labels. Bid-Based GP had an accuracy rate of over 90% across all three datasets [18]. Bid-Based GP was further improved upon into a framework called Symbiotic Bid-Based GP (SBB). SBB has been utilized in several works, against game-playing tasks such as RoboCup Soccer and Rubik's cube solving, as well as other classification tasks [8] [9] [17] [18] [19] [5] [20] [6].

3.2.2 Tangled Program Graphs

Tangled Program Graphs was created from the addition of a significant feature to the Bid-Based GP/SBB framework. A significant addition was made to the set of suggestable actions; instead of an atomic action, learners are now able to point at another team. This feature resulted in a framework that was fundamentally different enough to warrant a new name; Tangled Program Graphs (TPG) [11].

When a learner wins a bid, instead of returning an (atomic) action, some learners are able to refer to another team in the population, akin to how a family physician will refer a patient to a specialist physician when needed. When called upon by a learner, the newly appointed team executes the bidding process again. If the resulting action is a reference to yet another team, the process repeats, although cycles are prevented in the code implementation. This structure naturally lends itself to a graphical visual representation of teams and learners. Teams are represented by vertices and learners are represented by edges.

Further terminology was established. A "root team" is defined as a team with an in-degree of zero. Root teams are now considered individuals within a population, as opposed to a "regular" team. This is the team where the bidding process begins.

Figure 3.2 shows a visualization of three individuals in a Bid-Based GP context, while Figure 3.3 shows a visualization of a single individual in a TPG context, where learners can refer to other teams as their action.

TPG was first tested against the Atari Learning Environment [2], a suite of software used to train AIs on Atari video games using reinforcement learning. Their goal of this experiment was to create multi-task agents; agents that were capable of playing more than one game. TPG was tested against Centipede, Frostbite, Ms. Pac-Man, Asteroids, Battle Zone and Zaxxon. The authors theorized that, because learners could point to another team, "specialist" teams would emerge, teams that were highly skilled at one task, or one scenario within a task, as well as multi-purpose, "generalist" teams that are able to recognize when another team would be better suited to solving the current problem.

3.3 Deep Q-Network

The Deep Q-Network (DQN) appeared in 2015 and represented a milestone in the application of reinforcement learning to tasks described using high-dimensional (visual) state spaces. Prior to DQN each application would need appropriate task specific features to be designed by a human. DQN made use of a convolution neural network bottleneck architecture to develop an application specific encoding of the original state space without the intervention of a human. Competitive performance across a suite of 49 Atari 2600 game titles was then demonstrated, where competitive implied

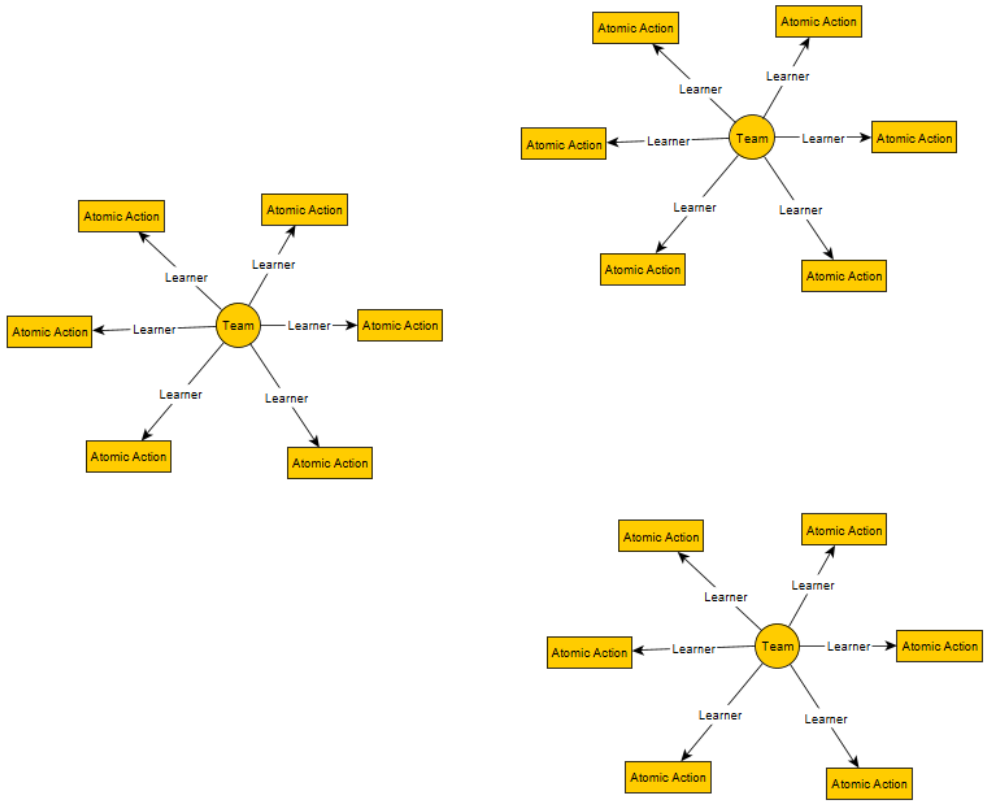


Figure 3.2: Three example individuals in a Bid-Based GP framework.

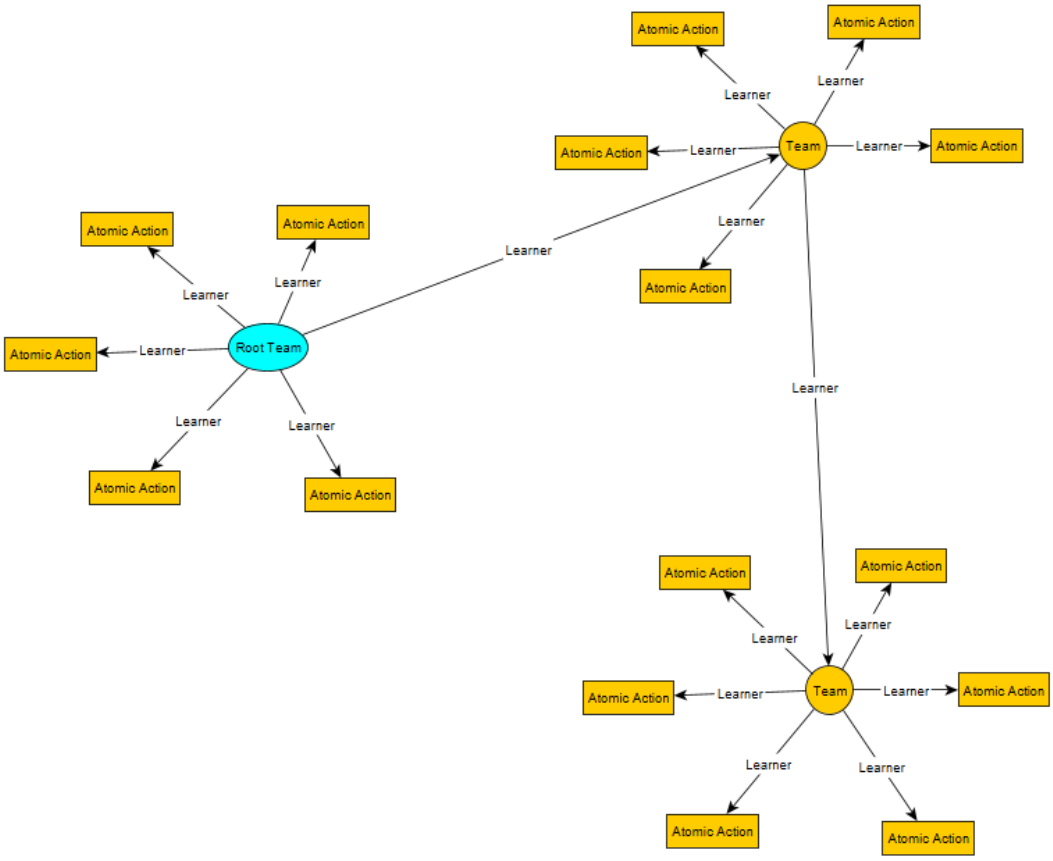


Figure 3.3: One example individual in the TPG framework.

that in 50 percent of the game titles performance of a human player could be reached (or bettered). Previous research with classical reinforcement learning methods was unable to reach human levels of performance unless game specific features were first provided [2].

DQN assumes a Q-learning formulation for a temporal difference error, or

$$Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma \max_{a}' Q(s',a') - Q(s,a)] \tag{3.1}$$

where r is the reward received on the transition between sequential states (a to a'); $Q(s,a)$ is the value of action a under state s.

The neural network has to provide a Q-value for each action at each state using its value function properties. However, as recognized by the original authors, several important caveats preclude a direct implementation of equation 3.1. Specifically, the correlation between state and therefore Q-values will introduce instabilities into gradient based parameter optimization.[1] In order to address this the authors introduced several innovations, summarized as follows.

- frame skip: agent only experiences every k-th frame with the action choice repeated over the k frames. The assumption being that there will be little if any meaningful state change between the intervening frames.

- frame staking: Objects typically move in games whereas the decision making agent is purely relative. In order to provide the agent with movement information, 4 consecutive frames are superimposed to produce a single 'stacked' frame. It is this that DQN receives as the input.

- experience replay: Rather than perform temporal difference on the sequence of states directly (which are correlated), the learning agent chooses an action for the current state and then performs weight updating relative to a history of state (s', s) and corresponding (a', r, a) outcomes. DQN is parameterized with

[1] Correlated state and action in this case is synonymous with performing supervised learning on a classification dataset that has not been stratified. Credit assignment might be performed relative to a single class before encountering another. This potentially results in learning agents unable to label anything other than a single class.

a history of a million previous states from which only 32 are sampled for Q-learning. It is unlikely that the 32 samples are correlated given the length of the history.

- weight caches: Different parameterizations of the neural network weights are employed for $Q(s, a)$ versus $Q(s', a')$. Specifically, $Q(s, a)$ represents the incrementally updated values whereas $Q(s', a')$ is retained for C sequential batches of updates. This represents another mechanism by which the DQN authors attempt to improve the stability of the weight updating process.

In summary, DQN represent the first of a sequence of deep reinforcement learning frameworks designed to make use of the convolutional neural bottleneck representation to address the feature construction challenge in high-dimensional state spaces. We adopt it in this work as an example of a gradient approach to credit assignment that is also provided as part of the ViZDoom game engine.

3.4 Summary

TPG will be assumed in this thesis as the one of two learning agents for evaluation under visual reinforcement learning tasks. Unlike most neural approaches to reinforcement learning tasks, TPG experiences tasks episodically. This means that TPG agents only undergo credit assignment after the episode's terminal condition is encountered. Agents are then ranked relative to other agents from the same population. TPG agents experience state by developing suitable indexing schemes for identifying which pixels to explicitly sample state from. Moreover, Team composition is an emergent process in which the topology of individual teams is explicitly unique. In short, model building is very much a process of organizing programs to identify contexts for either referencing other teams or choosing a particular action (from a task specific set of actions). Conversely, neural networks typically assume a fixed topology (designed by a human) and concentrate on optimizing a set of real-valued weights using some form of gradient decent.

DQN on the other hand, uses a convolutional neural network with a bottle neck architecture in order to develop an encoding of the original high-dimensional visual state space. When combined with a fully connected hidden layer and an output layer

with as many outputs as actions, then the Q-learning temporal difference method can be applied. To do so, the DQN authors introduce several other innovations in order to address the issue of correlated observations that lead to instabilities in weight optimization.

Both TPG and DQN represent reactive agents, which is to say, they have no capacity for learning how to represent the past.[2] They also approach state representation and credit assignment very differently. For these reasons we assume these as representative examples of very different approaches to (visual) reinforcement learning tasks.

[2]DQN uses a history of past states for weight updating, but cannot represent past states post training.

Chapter 4

Empirical Methodology

4.1 Methods for Maximizing TPG Performance

In this thesis, we adopted several optimizations for TPG: Rampant mutation, Action programs, SmallTPG, Phasic and Score maintenance. The underlying motivation being to reduce the complexity of TPG solutions as much as possible. These optimizations where previously evaluated under ViZDoom and simulated robot locomotion tasks [1].

4.1.1 Rampant Mutation

Rampant mutation has been shown previously to simplify TPG solutions while maintaining the same level of performance [1]. When using rampant mutation, child teams are mutated *mutationCount* times as opposed to the typical single mutation. This increases greater variability and therefore greater genetic diversity (the degree of neutrality in the representation space might of course limit the effectiveness of single and multiple mutations). Through increasing genetic diversity, we anticipate being able to decrease the likelihood of one familial line dominating the entire population. This contributes positively to the development of multiple TPG species, thus decreasing execution time. Naturally, the concept of diversity pertains to the representations as opposed to guaranteeing behavioural differences.

4.1.2 Action Programs

Canonical TPG assumed that actions were discrete and assigned randomly at learner initialization. However, there is no reason why action 'selection' cannot also be performed by a program. Thus, a learner is now defined by a bid program, action program, and optionally a pointer to another team. An "action program" is similar in structure to a bid program. It is a series of assembly instructions that have access

to a set of program specific registers. We replace singular atomic actions with action programs. The learner's action is instead the values in the first A action program registers, where A is the number of actions. This allows real-valued actions, (such as moving a certain distance) as well as multiple actions simultaneously, which is supported in ViZDoom. This means that a single learner is able to produce more than one action, whereas previously, we needed at least one learner for every possible atomic action. Action programs were previously shown to contribute significantly to the simplification of TPG solutions [1].

Operationally, learner evaluation commences as described in Section 3.2.1. Thus, for state $\vec{s}_t$ a winning learner is first identified following competition between the 'bid programs' from each learner in the same team. Only if the bid program 'wins' does evaluation move on to the corresponding learner's action program. Such a program is executed under the same state, resulting in values appearing in the action program's registers. For single action selection, the register with the maximum value identifies a 'winning' action. Under multi-action selection, a vector of actions is returned (values in the first A registers), resulting in the simultaneous parameterization of multiple actions.

4.1.3 SmallTPG

The "default" maximum number of learners per team is 12, with a minimum of 2.[1] When using SmallTPG, we limit teams to a maximum of four learners. Champion teams were anecdotally observed utilizing only a small fraction of their available graph structures. This is taken to imply that teams fill out with 'hitchhikers' that never win a round of bidding, hence we can save computation time by reducing the maximum number of learners. This strategy was shown to reduce complexity without sacrificing performance [1].

4.1.4 Phasic

Phasic implies that the learning process is divided into two phases. During the first phase, learners are prohibited from evolving a pointer to another team as their action.

[1]Less than 2 learners per team results in a degenerate team, i.e. only defines one action/action program for any state.

When combined with SmallTPG, the entire solution is limited to four learners. Applying this constraint on the learning process will force learners to first evolve effective single team behaviours before pointing at another team. Ideally, after transitioning to the second phase, during which team pointers are enabled, implies that the only teams that a learner could possibly link up with are teams that consist solely of other highly skilled learners. In Table 4.1 "phaseFlip" dictates the number of generations before which we transition to the second phase of learning.

4.1.5 Score maintenance

In previous implementations of TPG, individuals that survived to the next generation were able to keep the scores they earned during their birth generation, and were judged on those scores thereafter. By doing so, computation time was reduced, since a portion of the population did not need to execute more episodes. However, when a task is partially observable or non-stationary this might imply that some individuals simply "got lucky". Because these individuals had high training scores, they stayed in the population and continued to propagate their lackluster genes. During testing, these individuals performed poorly when exposed to more scenarios. For this research, we removed the speedup and forced individuals to repeatedly compete for their right to reproduce, with the goal of weeding out lucky individuals while maintaining truly skilled individuals that are able to demonstrate their skill repeatedly.

4.2 DQN Parameterization

DQN was utilized as a "black box" in this research. The example provided on the ViZDoom Git repository was used as a base and modified as needed to serve the correct task, as well as storing data in a familiar fashion. Parameters utilized are exactly as they appear in the example. [2]

4.3 Experimental Design

As previously described, a total of four learning experiments are assumed:

[2] `https://github.com/Farama-Foundation/ViZDoom/blob/master/examples/python/learning_pytorch.py`

Parameter	**Value**	**Description**
attempts	5	The number of episodes an individual agent participated in each generation.
iterations	1000	The number of generations to perform.
mutationCount	5	The number of times to run the mutation method for each individual being mutated.
mutateEvery	-1	(not used) Only mutate individuals every X generations.
actionMode	AP	Action programs (AP) or atomic actions (AA).
teamPopSize	120	The number of individuals in the initial population.
teamGap	50%	The percentage of individuals that will be kept during selection.
probLearnerDelete	70%	The probability that a learner will be deleted during reproduction.
probLearnerAdd	70%	The probability that a learner will be added to a team structure during reproduction.
maximumProgramSize	128	The maximum number of instructions allowed in a *bid program*.
maximumActionProgramSize	128	The maximum number of instructions allowed in an *action program*.
numberofActionRegisters	7	The number of registers available to an agent to utilize during the execution of its action program.
maximumTeamSize	4	The maximum number of learners per team. (Known as small TPG)
minimumLearnersPerTeam	2	The minimum number of learners per team.
phaseFlip	500	The number of generations to perform before "flipping" to the second phase.

Table 4.1: A description of the parameters required by our TPG implementation.

		Testing	
		Biased	Uniform
Train	Biased	B, B	B, U
	Uniform	U, B	U, U

Table 4.2: Common training and Test Treatments for Basic and My Way Home as used with both TPG and DQN learning agents

1. TPG and DQN trained on Basic (My Way Home) utilizing the original random number generator.

2. TPG and DQN trained on Basic (My Way Home) utilizing an external, uniform random number generator.

Following training, we assume eight testing scenarios:

1. TPG and DQN trained on Basic (My Way Home) utilizing the original random number generator, tested on Basic (My Way Home) utilizing the original random number generator.

2. TPG and DQN trained on Basic (My Way Home) utilizing the original random number generator, tested on Basic (My Way Home) utilizing an external, uniform random number generator.

3. TPG and DQN trained on Basic (My Way Home) utilizing an external, uniform random number generator, tested on Basic (My Way Home) utilizing the original random number generator.

4. TPG and DQN trained on Basic (My Way Home) utilizing an external, uniform random number generator, tested on Basic (My Way Home) utilizing an external, uniform random number generator.

This set of scenarios is more succinctly represented in Table 4.2.

4.3.1 TPG and Java

While TPG has a Python implementation, the Java version is the most familiar implementation. This means that the Java version of ViZDoom is assumed for TPG

	Operating System	CPU	GPU
Computer 1	Windows 10	Intel i7-3770	NVIDIA GeForce GTX 550 Ti
Computer 2	Windows 10	Intel i7-4790K	NVIDIA GeForce GTX 960
Computer 3	Windows 10	Intel i7-2600	NVIDIA GeForce GTX 1070

Table 4.3: A summary of the specifications of the computers utilized in this research.

experiments. In the interest of transparency, the Java version of ViZDoom was retired and has not been updated since 2020 according to the commit history on the official ViZDoom Git repository. However, this research began in 2020 and as such, the Java version continued to be utilized for consistency.

TPG instances were initialized with a population of 120 root teams. Each instance (of which there were four) lasted for 1,000 generations. Each team was given 5 episodes of their respective task.

4.3.2 DQN and Python

The Python version of ViZDoom is currently supported and kept up to date.

4.3.3 Hardware and Software Specifications

A total of three computers were utilized for this research. Table 4.3 provides OS, CPU and GPU specifications. Computer 2 was only utilized to gather data about the ViZDoom random number generator (Section 2.4). Computer 2 was wiped clean and loaded with a fresh installation of Windows 10 prior to the aforementioned experiments. ViZDoom was installed via pip as per the instructions on the ViZDoom Git repository. Comparison data obtained from a Linux operating system was obtained via an Ubuntu virtual machine on Computer 3.

For the learning and testing experiments, Computer 2 was used for TPG with Java. Computer 3 was used for DQN with Python. Corresponding versions of ViZDoom were installed.

Chapter 5

Results

5.1 Measure the impact of spawn biases in ViZDoom

Learning agents in general are sensitive to biases in the environment [15], or "garbage in, garbage out". Moreover, learning agents will only experience a finite set of scenarios during training. If those training scenarios are not representative of the underlying task, then there will be a disconnect between performance of the agent under training versus that observed under "test", i.e. the agent fails to generalize to solve the task as perceived by the user [4].

Our hypothesis is that irrespective of the type of visual reinforcement learning agent, the ability to solve the underlying objective of the task (generalization) will be heavily biased by the selection of spawn points. However, as demonstrated in Section 2.4, selecting Windows versus Ubuntu as the operating system for performing experiments with ViZDoom will completely change the behaviour of the spawn point. Thus, a seemingly innocuous and unrelated decision regarding operating system will have a profound impact on the quality of the resulting learning agents.

In order to investigate this hypothesis we assume two of the task environments provided in the ViZDoom game engine: Basic and My Way Home. As previously established (§2.1) we will use two tasks to investigate the impact of spawning biases in ViZDoom:

- **Basic:** represents a simple scenario in which the environment consists of a single rectangular room. The agent is spawned in the centre of one the two longer walls. A single monster, the "Cacodemon" is spawned at a "random" location on the opposite wall. The monster is stationary. The player can strafe left, strafe right, or shoot. The goal of the task is shoot the monster as quickly and as accurately as possible over an episode of 300 frames. An agent scores +106 for shooting the monster, -5 for each bullet used and, -1 for each elapsed

frame.

- **My Way Home:** requires the agent to navigate a multi-room (8 rooms connected by several corridors, total of 17 possible spawn locations) scenario to locate a green armour pack. The armour is always located in the same room, but the agent is spawned in a "randomly" chosen spawn point with a "random" orientation. The agent actions are turn left/ right and move forward. The total episode length is 2,100 whereas the reward is +1 for finding the armour pack and -0.0001 for each elapsed time frame.

In both cases we will assume agents that are purely reactive. This will make the task of navigation particularly difficult. We naturally anticipate agents to be better at navigating to the armour when they are initialized in rooms closer to the armour. Further, we would expect a significant learning bias to appear if the agent does not experience enough diversity in training cases, i.e. due to the broken random number generator under ViZDoom installations performed on the Windows operating system.

A total of two "treatments" will be performed to reflect the original (biased) random number generator versus fixed (uniform) random number generator as applied during training versus test in Basic and My Way Home (Table 4.2). Moreover, we consider two entirely unrelated learning agents: tangled program graphs (TPG) and Deep Q-learning (DQN). As previously introduced, TPG is an emergent approach to evolving graphs of teams of programs (§3.2.2). Conversely, DQN represents the first example of (gradient based) reinforcement learning that was based on a deep convolutional neural network architecture [21]. DQN demonstrated for the first time that visual reinforcement learning tasks could be solved using video state information alone. Since this initial result, DQN has been applied to many other visual reinforcement learning tasks, including Basic in ViZDoom [27].

Naturally, TPG and DQN have no commonality with regards to representation, credit assignment or cost function. The only common factor is state and actions that the agent may assume under each task. As a consequence, any common properties in the behaviours of the agents to solve under the different treatments must be due to the configuration of the task, i.e. stochastic properties of the spawned monster (under "Basic") and initialization of the agent (under "My Way Home"). This leads

Trained on	Tested on	Average	High	Low	Median
Biased	Biased	71.751	95	-400	95
Biased	Uniform	41.23	95	-405	66
Uniform	Biased	78.561	95	-320	95
Uniform	Uniform	61.756	95	-340	71

Table 5.1: DQN Basic Testing - 1000 Episodes.

Trained on	Tested on	Average	High	Low	Median
Biased	Biased	-42.3793	95.0	-410.0	94.0
Biased	Uniform	-165.5253	95.0	-410.0	-76.0
Uniform	Biased	-85.44717	95.0	-410.0	52.0
Uniform	Uniform	-132.415	95.0	-410.0	-39.0

Table 5.2: TPG Basic Testing - 100 Episodes, top 0.001% of teams

to the assessment of each type of learning agent under training and test conditions that reflect both treatments, or a total of four experiments, Table 4.2.

In order to select TPG individuals to test, every team, from every generation, from every run was sorted into a list by their average training score. Then, the top 0.001% (between 40 and 50) of agents were used for testing. This process was repeated for teams trained on the biased and uniform versions of My Way Home. TPG agents were only tested on 100 episodes of each version of My Way Home, while each of the two DQN agents were tested on 1000 episodes. This decision was made in the interest of time; testing approximately 100 TPG individuals on 1000 episodes per individual would simply take too long.

DQN trained on the biased Basic scenario was able to perform on the uniform scenario, on average, almost equally as well a human player. The human player average was 65.65 with a mean of 68. DQN trained on the biased scenario had an

Trained on	Tested on	Average	High	Low	Median
Biased	Biased	-0.1759584	0.9396	-0.21	-0.21
Biased	Uniform	-0.0840731	0.9951	-0.21	-0.21
Uniform	Biased	-0.1720361	0.9351	-0.21	-0.21
Uniform	Uniform	-0.094625	0.9944	-0.21	-0.21

Table 5.3: DQN My Way Home Testing - 1000 Episodes.

Trained on	Tested on	Average	High	Low	Median
Biased	Biased	0.579446357	0.9971	-0.21	0.9227
Biased	Uniform	0.157993	0.9957	-0.21	-0.21
Uniform	Biased	0.183844	0.9977	-0.21	-0.21
Uniform	Uniform	0.56109	0.9967	-0.21	0.86729

Table 5.4: TPG My Way Home Testing - 100 Episodes, top 0.001% of teams.

Task	Average	Median	Max	Min
Basic	65.65	68	79	52
My Way Home	0.969085	0.97185	0.9895	0.9454

Table 5.5: A summary of scores obtained by a human player. The human player received 20 episodes of each task, utilizing the uniform random number generator. The human player was familiar with both tasks, but is not a skillful player of first person shooter games. The key mapping was also changed from the default provided with ViZDoom to conform with modern first person shooter standards; forward, left, backwards, and right movement were mapped to WASD respectively, rather than the arrow keys. Available actions were not changed (for instance, forward (W) and backwards (S) were disabled when the human player was being Tested against the Basic task).

average score of 41.23 and a median score of 66. TPG trained on the biased and uniform scenarios was not able to perform as well as a human player. However, in My Way Home, DQN did not perform as well as a human player.

5.2 Basic Results and Discussion

Given the width of the Cacodemon, there are effectively 11 "zones" in which it can spawn. Each agent was tested on 100 episodes in each zone. The values in Figure 5.1 represent the mean of the 100 episodes. The left side of the figure represents the left side of the map (which is why the Y-axis values appear reversed). It is evident that TPG trained on the biased version of Basic is unable to score well on the right side of the map. Interestingly, DQN trained on the biased version of Basic is able to score highly on the right side of the map. This suggests that DQN has learned to truly aim, while TPG has evolved a strategy that works, but, only in the scenario for which it has been trained.

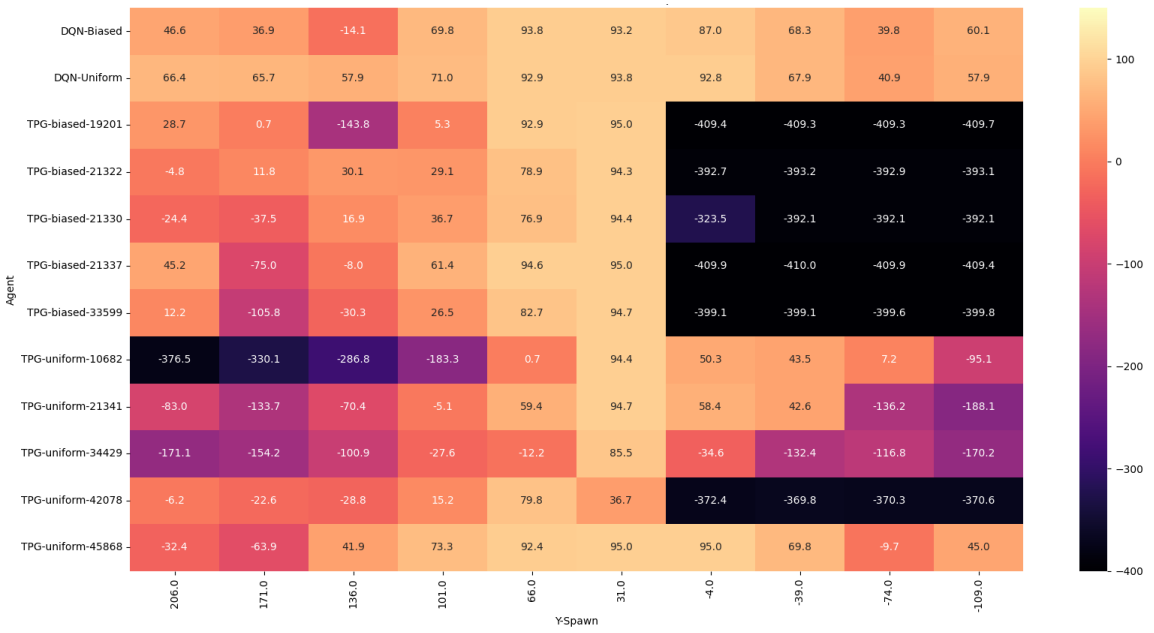
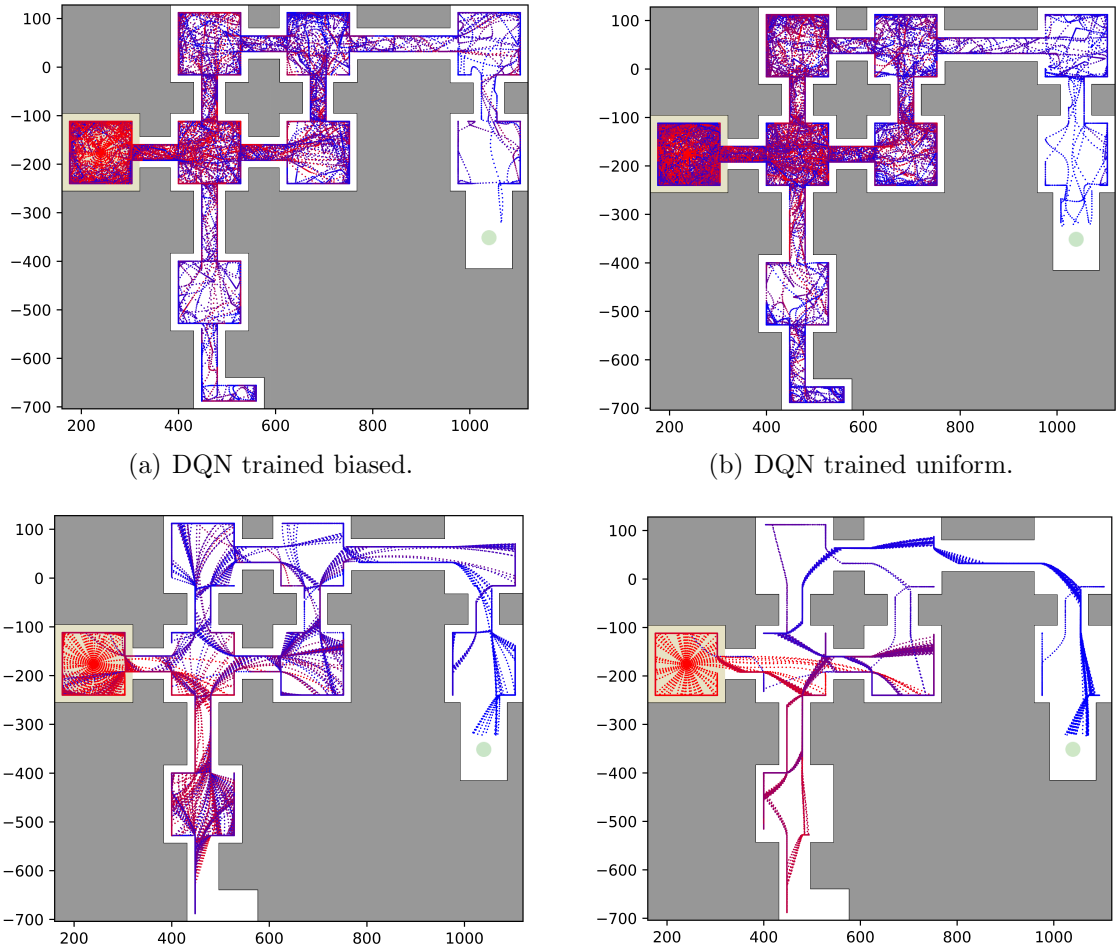


Figure 5.1: This figure is a heatmap that relates the scores of each individual to the location of the monster ("Y-spawn" denotes the Y-coordinate of where the monster spawned, see Figure 2.2). Lighter colours indicate a higher score for that region. A topdown view is depicted where the player spawns towards the bottom of the image and the monster spawns along the top. TPG individuals are differentiated with an identification number. Simply put, this number denotes the age of the team, as in, Team 19201 was the 19,201$^{\text{st}}$ team created during that particular training execution.



(a) DQN trained biased.

(b) DQN trained uniform.

(c) TPG trained biased.

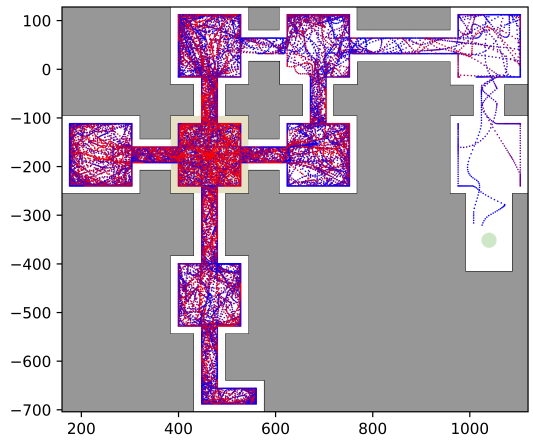
(d) TPG trained uniform.

Figure 5.2: Spawn point 10: The paths taken from spawn point 10 by (a) DQN trained biased (b) DQN trained uniform (c) TPG trained biased (d) TPG trained uniform.

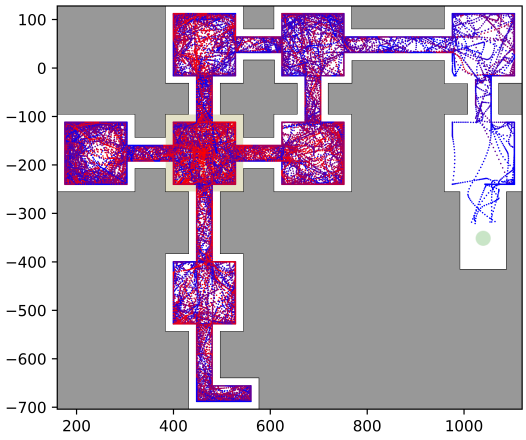
5.3 My Way Home Results and Discussion

To better understand and decompose the effect of the random number generator on training and testing in My Way Home, individuals were tested at each of the spawn points.

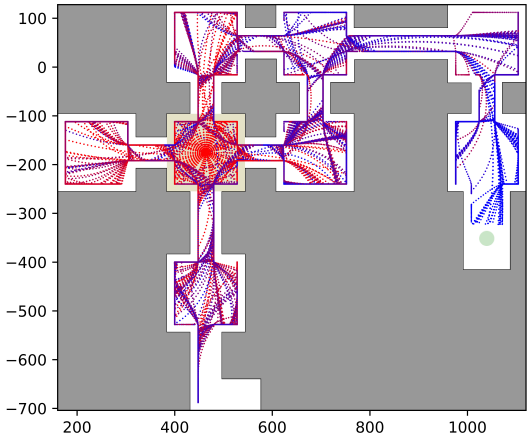
Each agent was given 100 episodes in each of the 17 spawn points on the My Way Home map. At the start of each episode, the agent was spawned facing a uniformly distributed random angle. These agents include the five best performers from TPG trained on the biased scenario, the five best performers from TPG trained on the uniform scenario, a DQN agent trained on the biased scenario, and a DQN agent



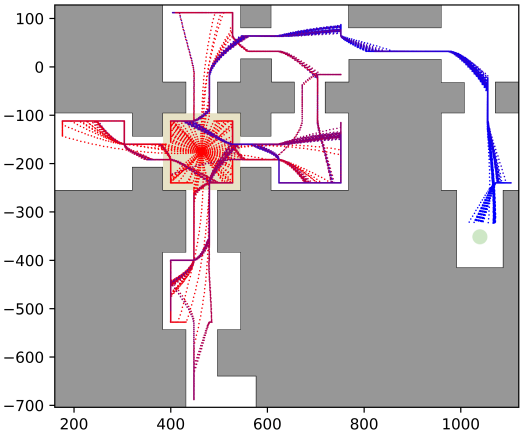
(a) DQN trained biased.



(b) DQN trained uniform.

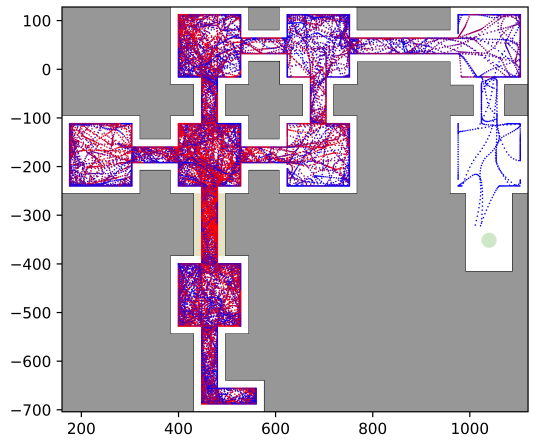


(c) TPG trained biased.

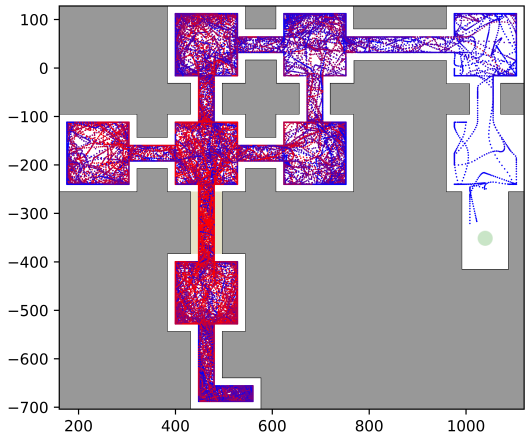


(d) TPG trained uniform.

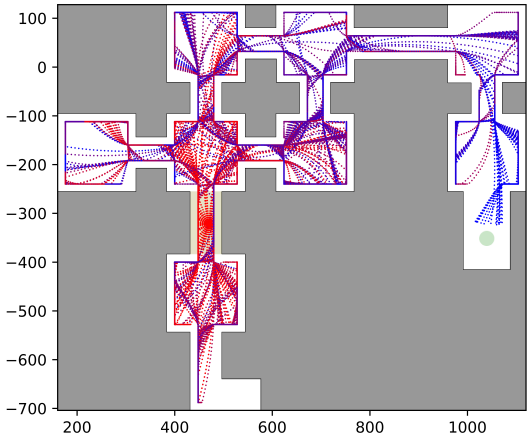
Figure 5.3: Spawn point 11: The paths taken from spawn point 11 by (a) DQN trained biased (b) DQN trained uniform (c) TPG trained biased (d) TPG trained uniform.



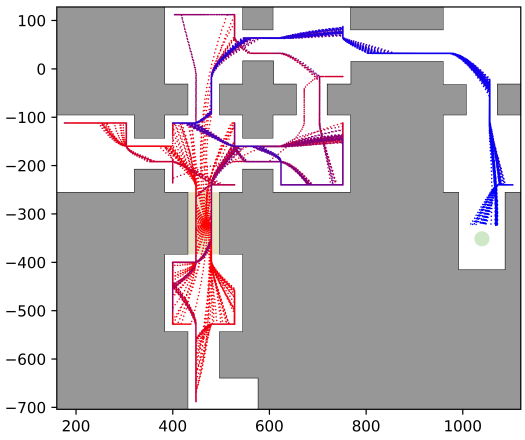
(a) DQN trained biased.



(b) DQN trained uniform.

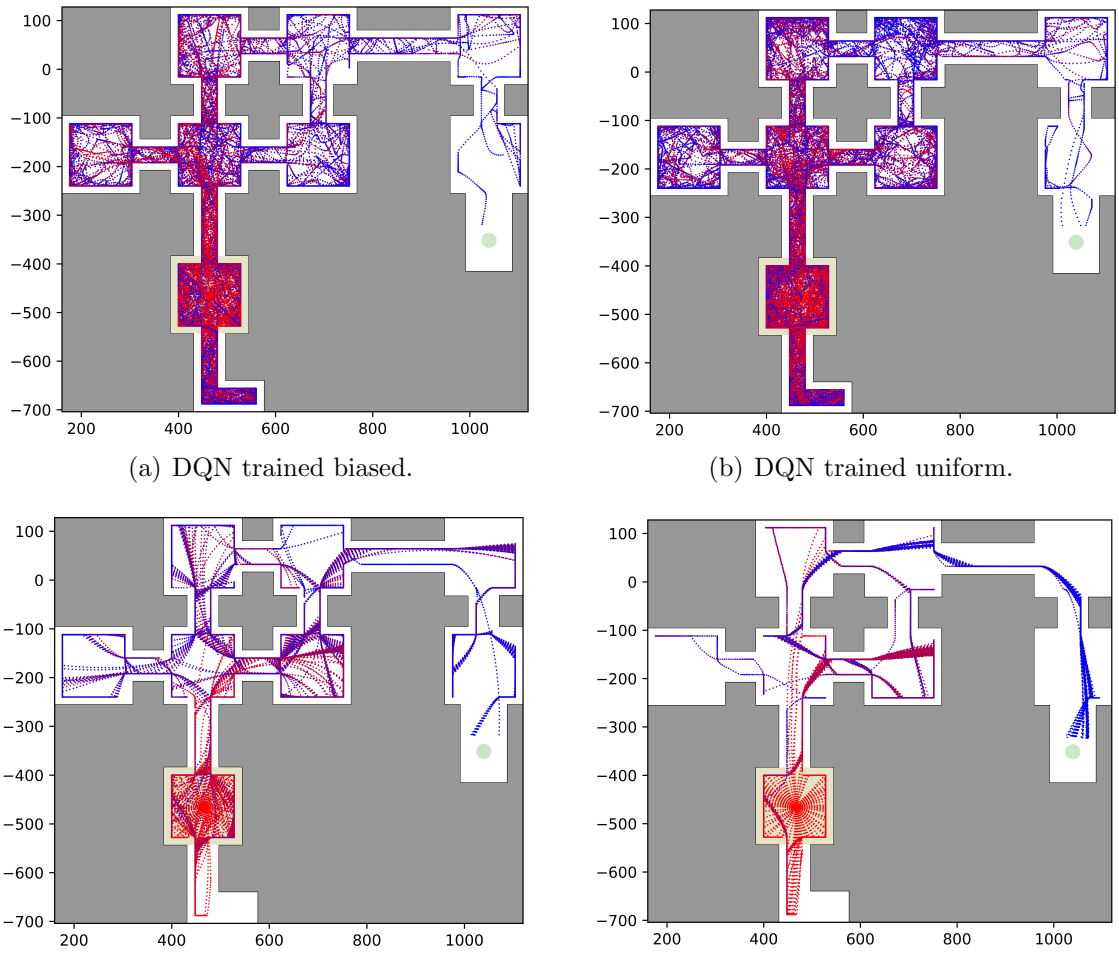


(c) TPG trained biased.



(d) TPG trained uniform.

Figure 5.4: Spawn point 12: The paths taken from spawn point 12 by (a) DQN trained biased (b) DQN trained uniform (c) TPG trained biased (d) TPG trained uniform.



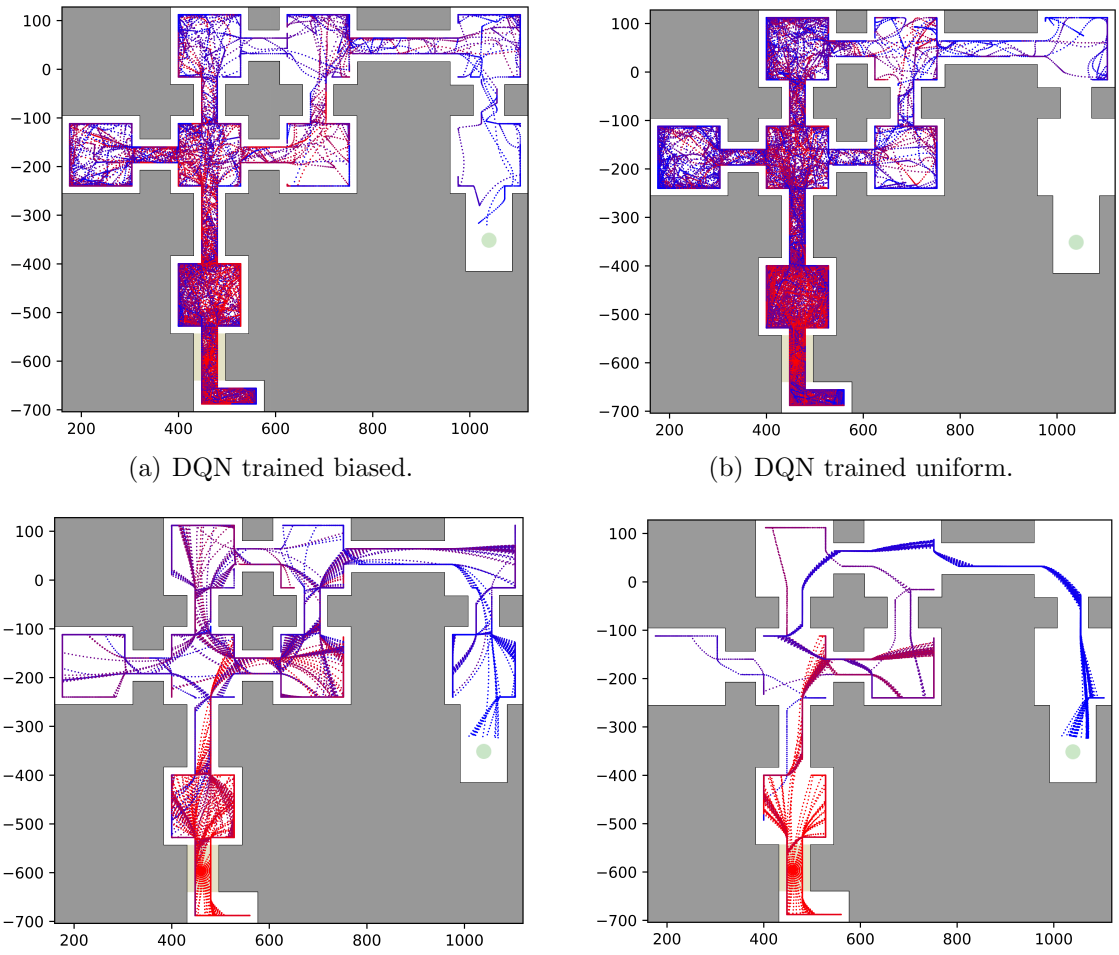
(a) DQN trained biased.

(b) DQN trained uniform.

(c) TPG trained biased.

(d) TPG trained uniform.

Figure 5.5: Spawn point 13: The paths taken from spawn point 13 by (a) DQN trained biased (b) DQN trained uniform (c) TPG trained biased (d) TPG trained uniform.



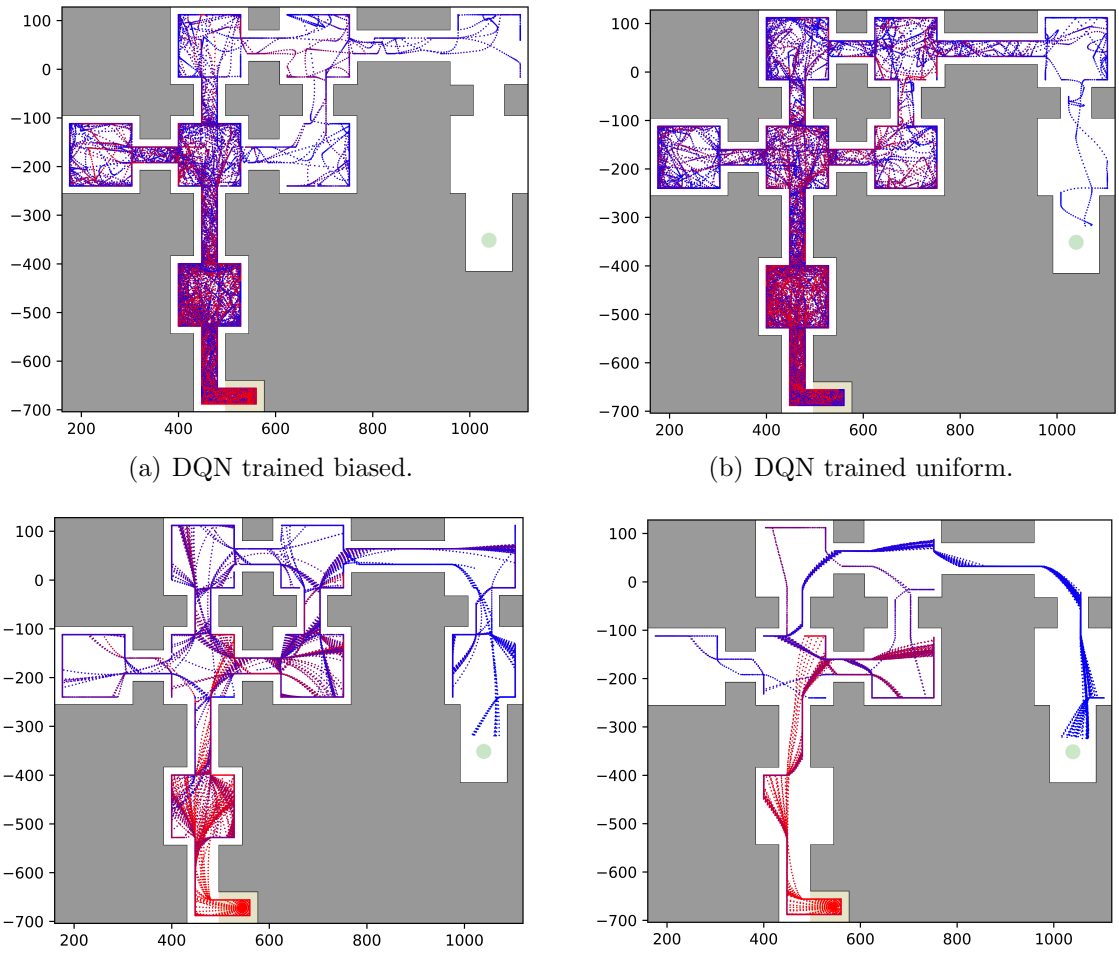
(a) DQN trained biased.

(b) DQN trained uniform.

(c) TPG trained biased.

(d) TPG trained uniform.

Figure 5.6: Spawn point 14: The paths taken from spawn point 14 by (a) DQN trained biased (b) DQN trained uniform (c) TPG trained biased (d) TPG trained uniform.



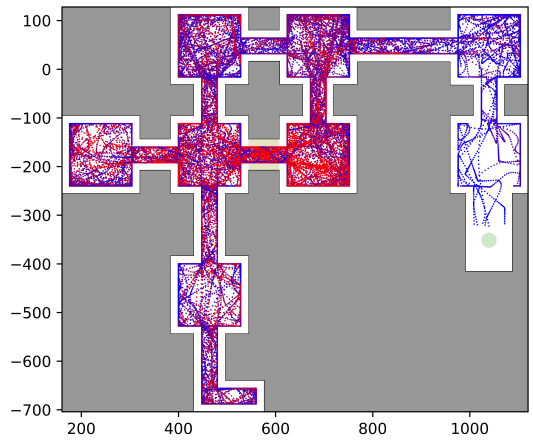
(a) DQN trained biased.

(b) DQN trained uniform.

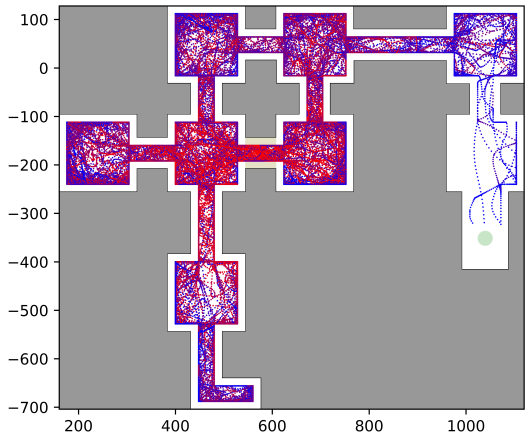
(c) TPG trained biased.

(d) TPG trained uniform.

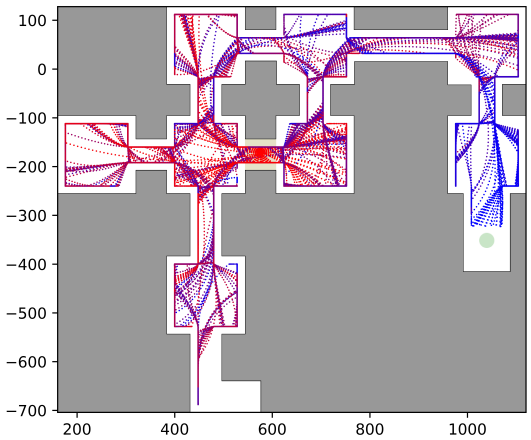
Figure 5.7: Spawn point 15: The paths taken from spawn point 15 by (a) DQN trained biased (b) DQN trained uniform (c) TPG trained biased (d) TPG trained uniform.



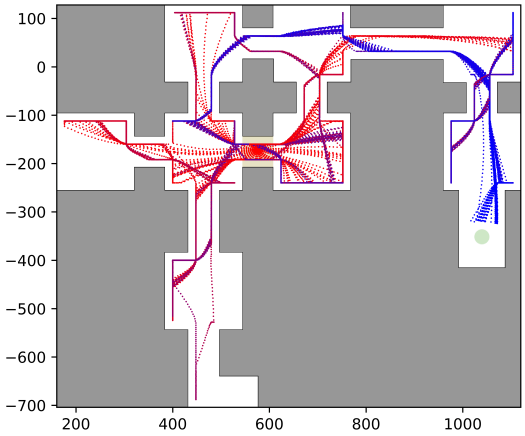
(a) DQN trained biased.



(b) DQN trained uniform.

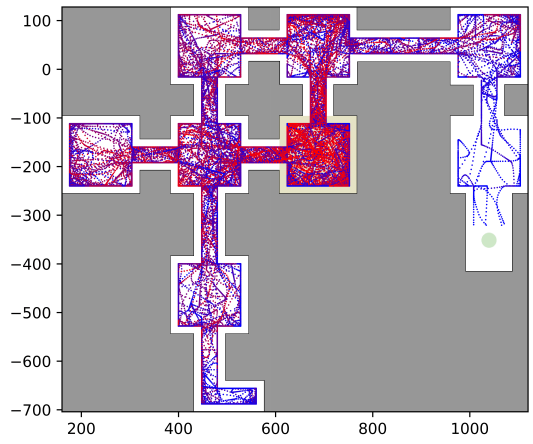


(c) TPG trained biased.

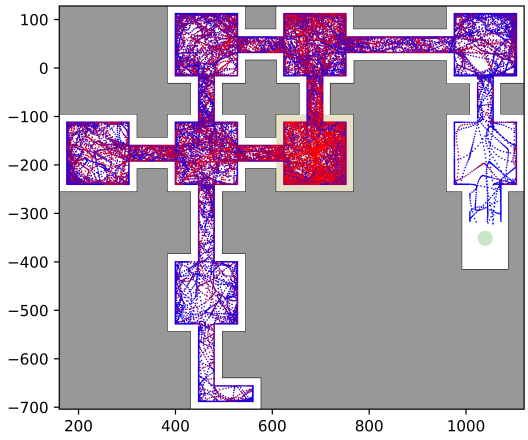


(d) TPG trained uniform.

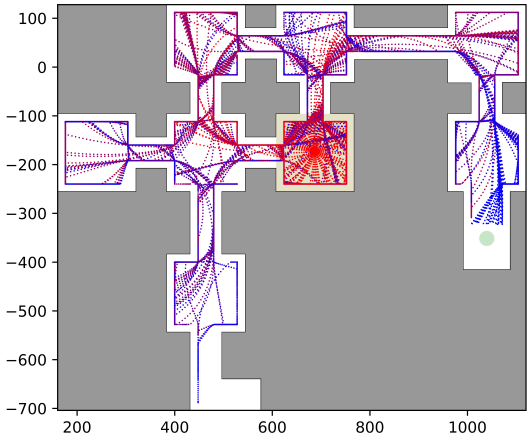
Figure 5.8: Spawn point 16: The paths taken from spawn point 16 by (a) DQN trained biased (b) DQN trained uniform (c) TPG trained biased (d) TPG trained uniform.



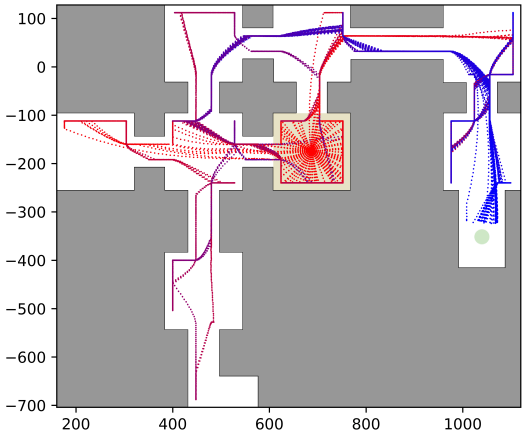
(a) DQN trained biased.



(b) DQN trained uniform.

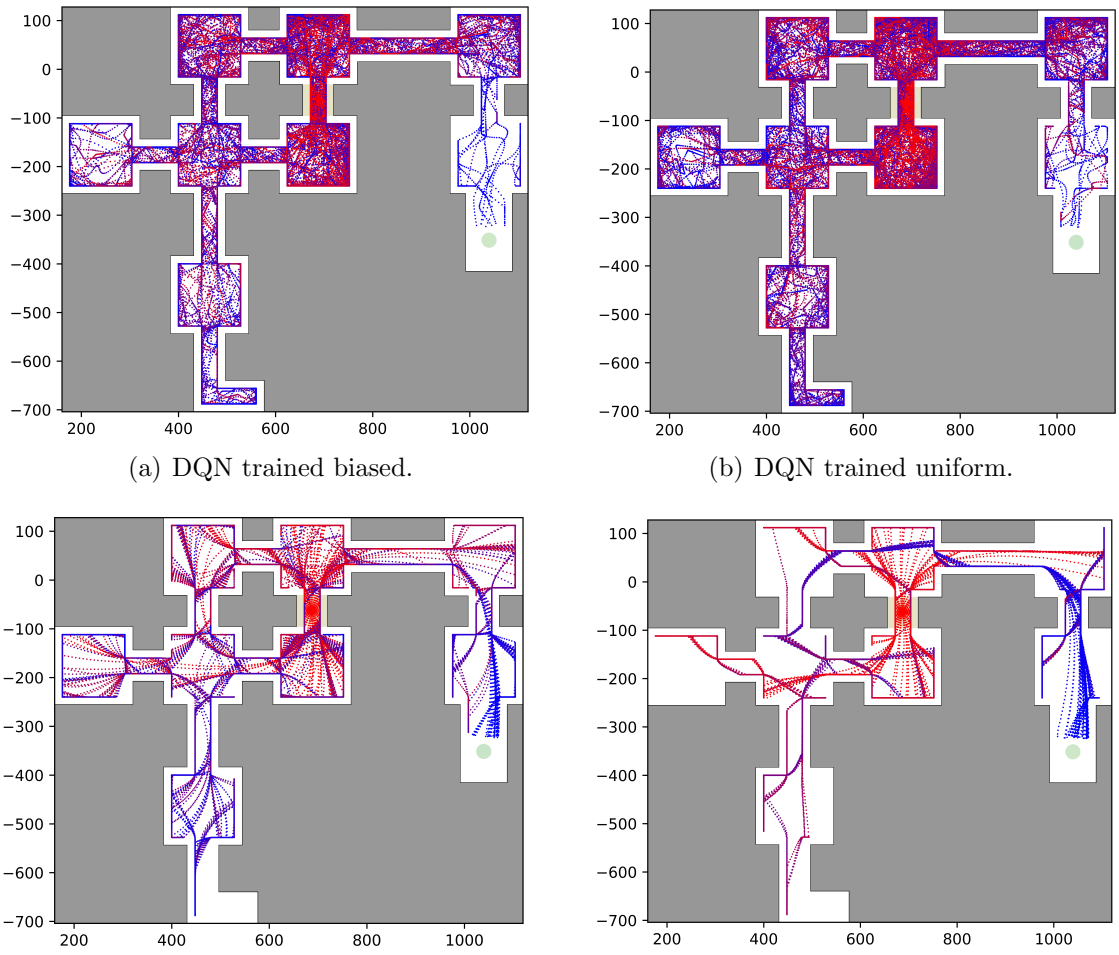


(c) TPG trained biased.



(d) TPG trained uniform.

Figure 5.9: Spawn point 17: The paths taken from spawn point 17 by (a) DQN trained biased (b) DQN trained uniform (c) TPG trained biased (d) TPG trained uniform.



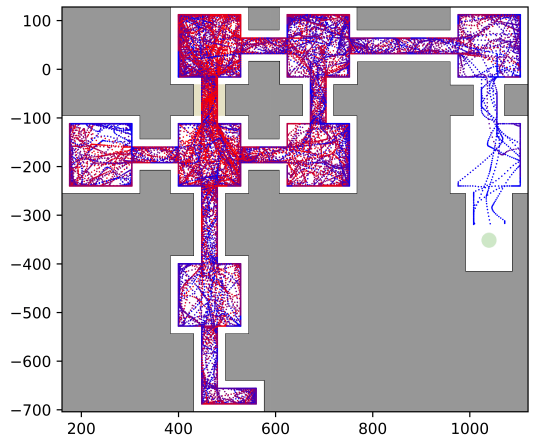
(a) DQN trained biased.

(b) DQN trained uniform.

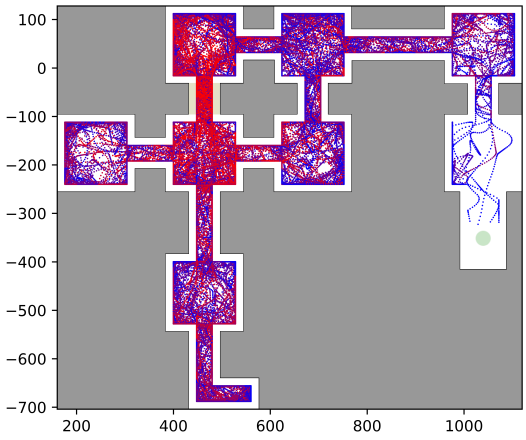
(c) TPG trained biased.

(d) TPG trained uniform.

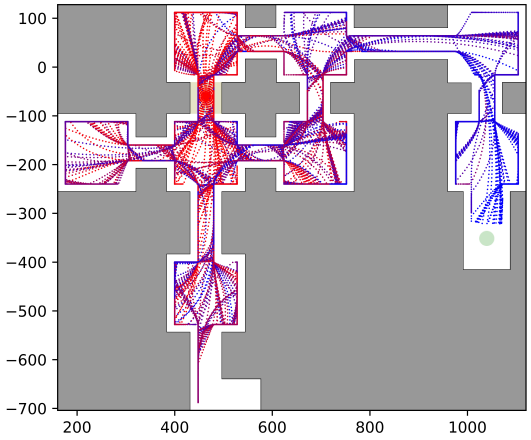
Figure 5.10: Spawn point 18: The paths taken from spawn point 18 by (a) DQN trained biased (b) DQN trained uniform (c) TPG trained biased (d) TPG trained uniform.



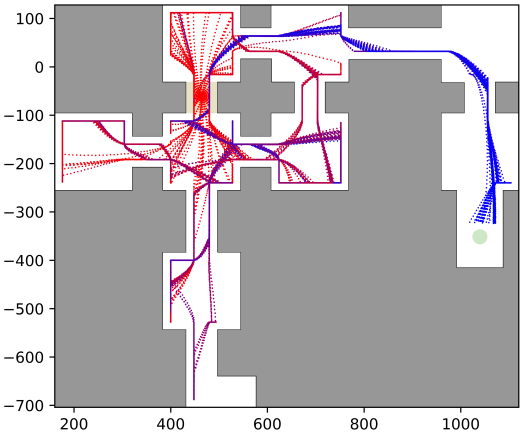
(a) DQN trained biased.



(b) DQN trained uniform.

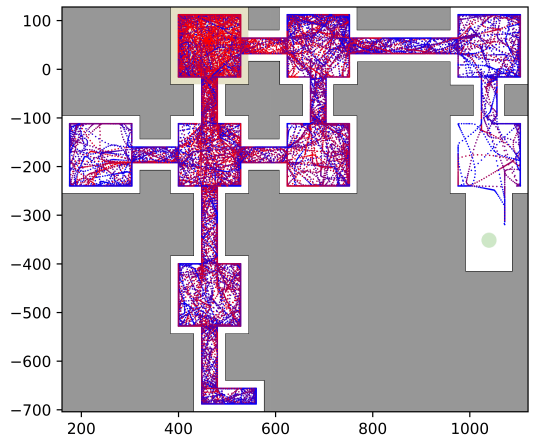


(c) TPG trained biased.

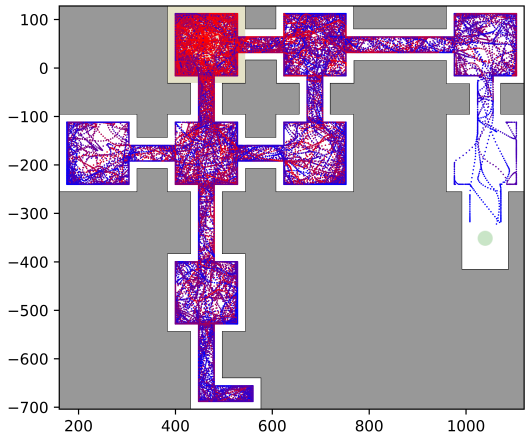


(d) TPG trained uniform.

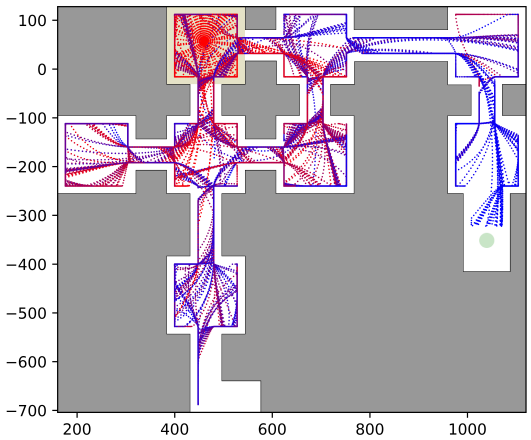
Figure 5.11: Spawn point 19: The paths taken from spawn point 19 by (a) DQN trained biased (b) DQN trained uniform (c) TPG trained biased (d) TPG trained uniform.



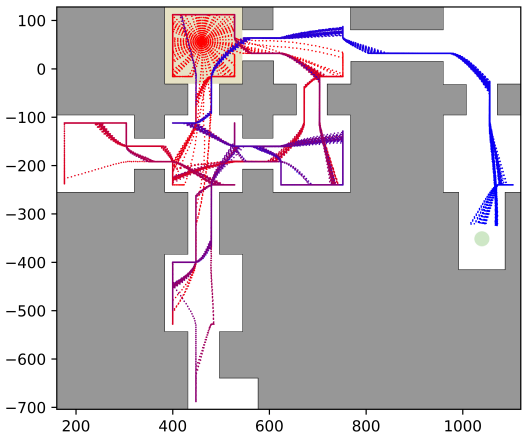
(a) DQN trained biased.



(b) DQN trained uniform.

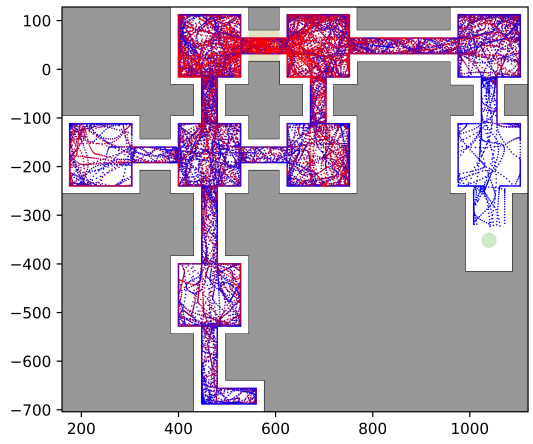


(c) TPG trained biased.

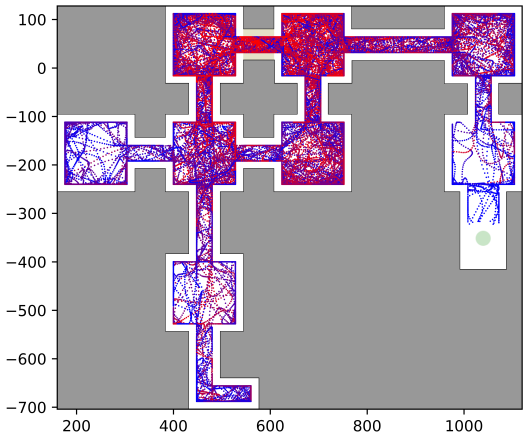


(d) TPG trained uniform.

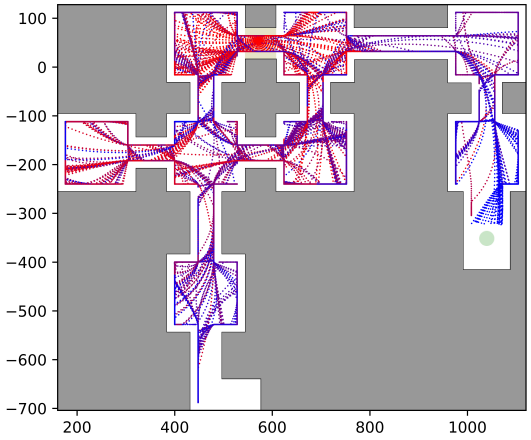
Figure 5.12: Spawn point 20: The paths taken from spawn point 20 by (a) DQN trained biased (b) DQN trained uniform (c) TPG trained biased (d) TPG trained uniform.



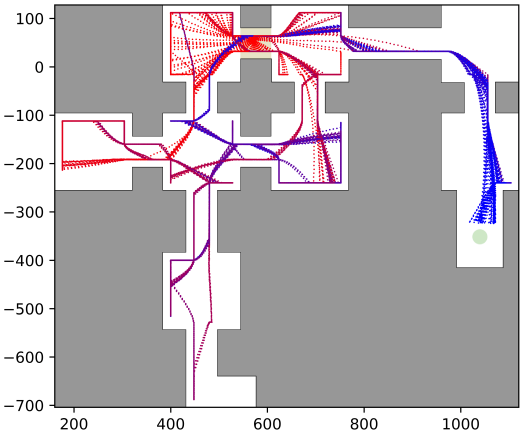
(a) DQN trained biased.



(b) DQN trained uniform.

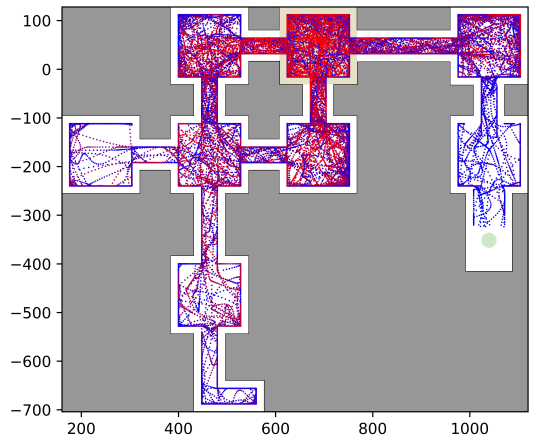


(c) TPG trained biased.

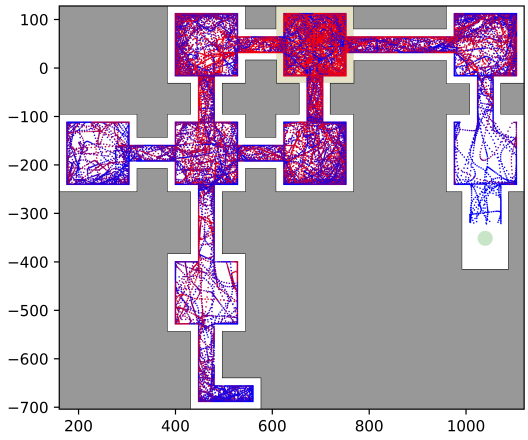


(d) TPG trained uniform.

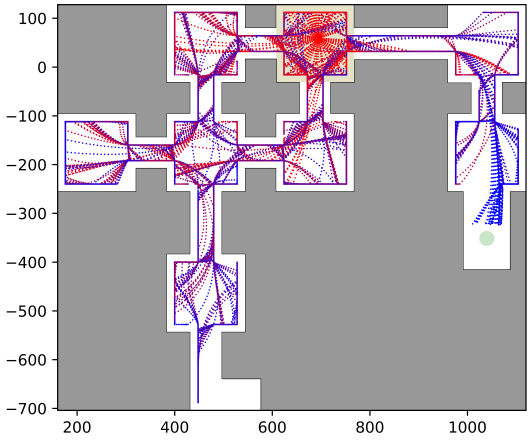
Figure 5.13: Spawn point 21: The paths taken from spawn point 21 by (a) DQN trained biased (b) DQN trained uniform (c) TPG trained biased (d) TPG trained uniform.



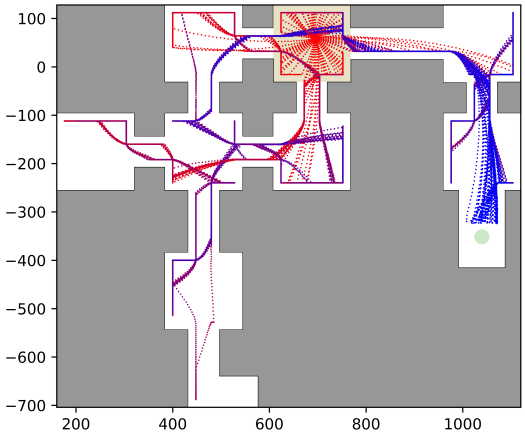
(a) DQN trained biased.



(b) DQN trained uniform.

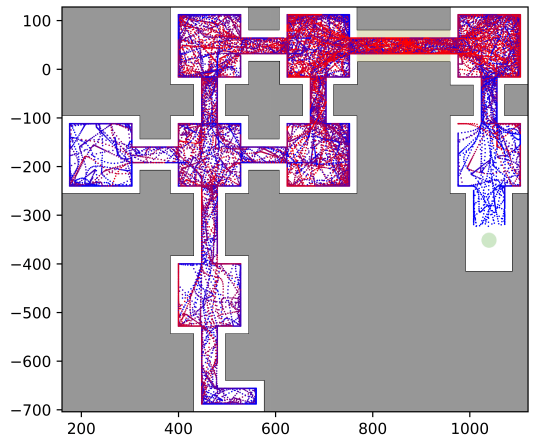


(c) TPG trained biased.

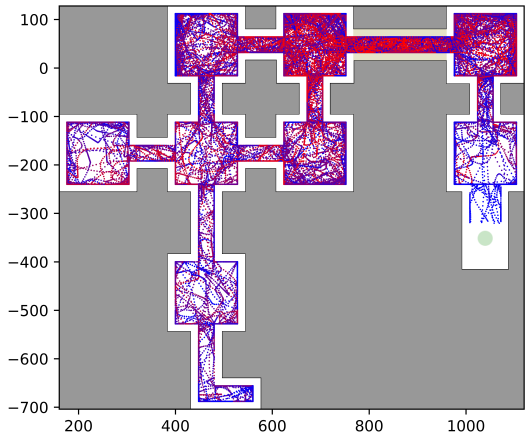


(d) TPG trained uniform.

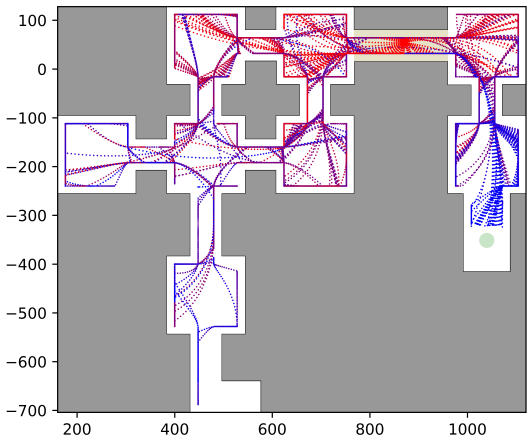
Figure 5.14: Spawn point 22: The paths taken from spawn point 22 by (a) DQN trained biased (b) DQN trained uniform (c) TPG trained biased (d) TPG trained uniform.



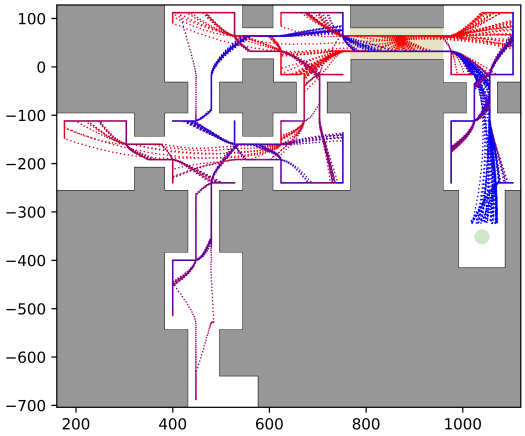
(a) DQN trained biased.



(b) DQN trained uniform.

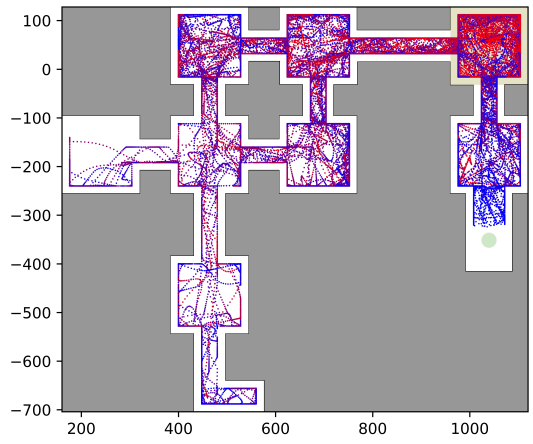


(c) TPG trained biased.

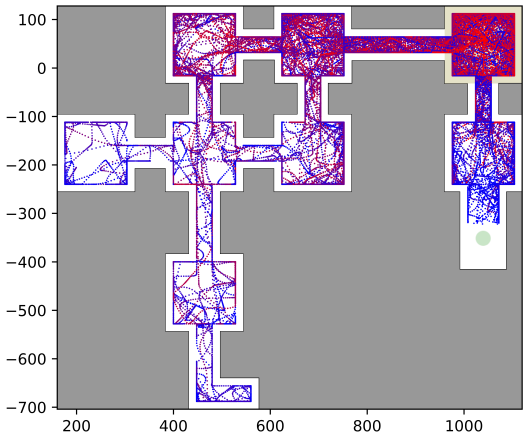


(d) TPG trained uniform.

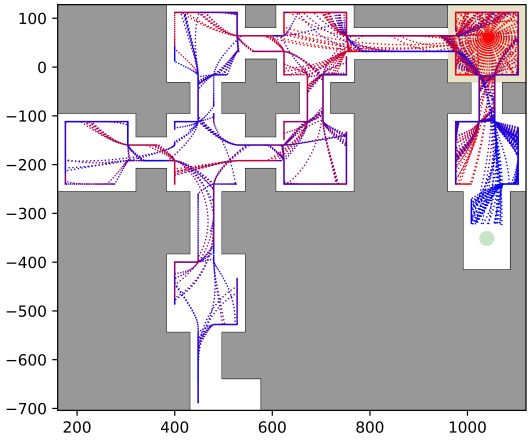
Figure 5.15: Spawn point 23: The paths taken from spawn point 23 by (a) DQN trained biased (b) DQN trained uniform (c) TPG trained biased (d) TPG trained uniform.



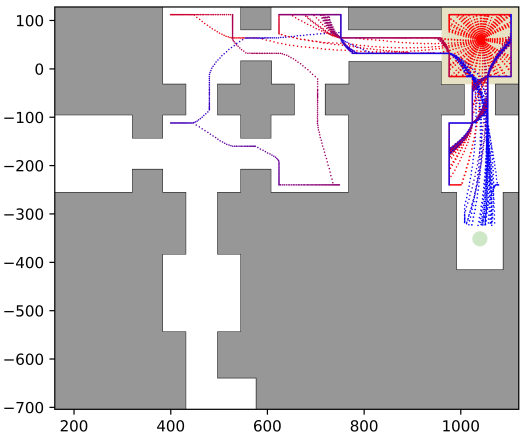
(a) DQN trained biased.



(b) DQN trained uniform.

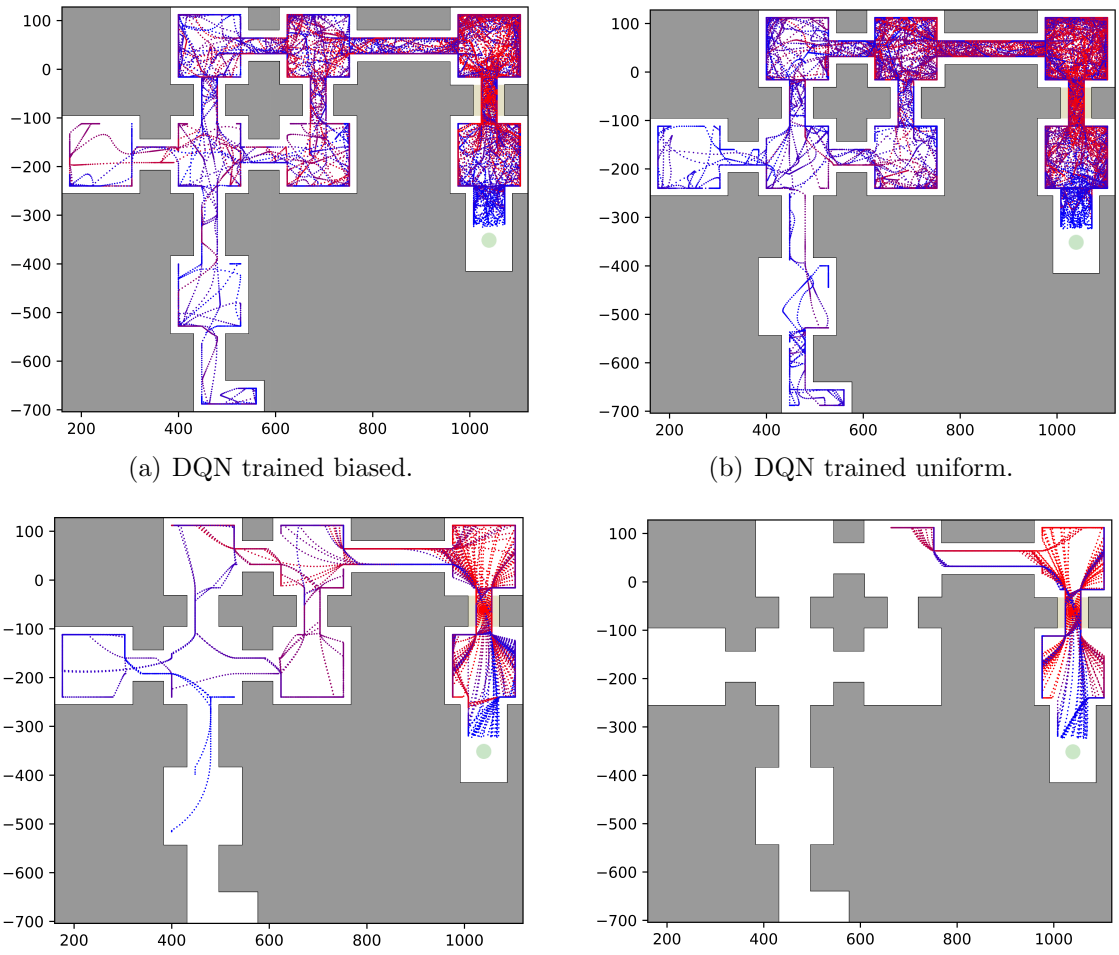


(c) TPG trained biased.



(d) TPG trained uniform.

Figure 5.16: Spawn point 24: The paths taken from spawn point 24 by (a) DQN trained biased (b) DQN trained uniform (c) TPG trained biased (d) TPG trained uniform.



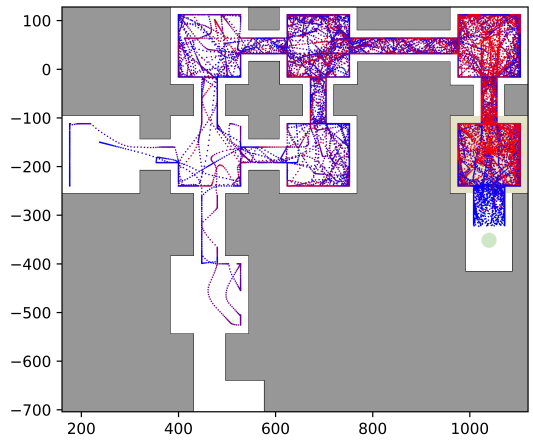
(a) DQN trained biased.

(b) DQN trained uniform.

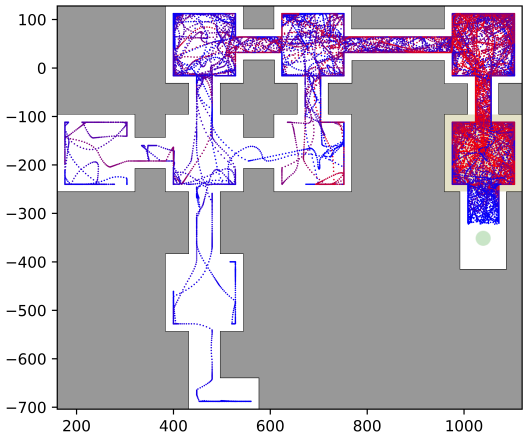
(c) TPG trained biased.

(d) TPG trained uniform.

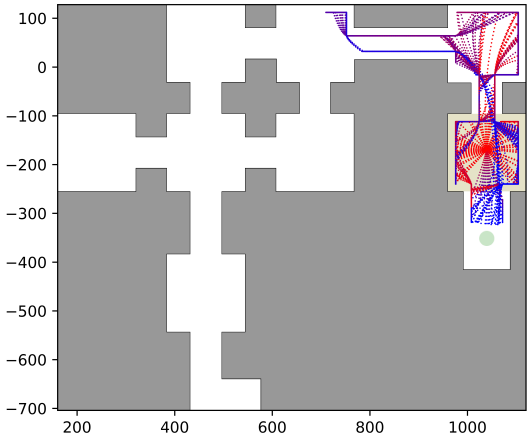
Figure 5.17: Spawn point 25: The paths taken from spawn point 25 by (a) DQN trained biased (b) DQN trained uniform (c) TPG trained biased (d) TPG trained uniform.



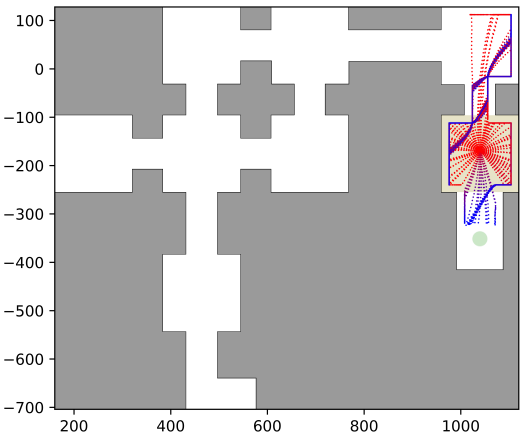
(a) DQN trained biased.



(b) DQN trained uniform.



(c) TPG trained biased.



(d) TPG trained uniform.

Figure 5.18: Spawn point 26: The paths taken from spawn point 26 by (a) DQN trained biased (b) DQN trained uniform (c) TPG trained biased (d) TPG trained uniform.

trained on the uniform scenario. The x-y coordinate of every agent, every frame, over every episode, was recorded. Example paths can be seen in Figure 5.2 to Figure 5.18 on a room by room basis for both DQN and TPG champion agents. The area that contains the spawn point is highlighted in a pale yellow colour. The armour pack is marked with a solid green circle. [1]

It is important to note that, following the example provided in the ViZDoom documentation on GitHub, DQN models were trained with a frame skip of 12 [2]. In ViZDoom, a "frame skip" is when an agent receives state information, sends an action, and that action is repeated for a number of frames equal to the frame skip. Its purpose is to decrease the number of frames an agent has to process, thus speeding up learning. However, in testing, it was found that using a frame skip lead to visualizations of paths that were not coherent. In order to capture the x-y position of the agent every frame, the frame skip (post training) was set to 0, and the sending of actions and processing of frames was handled manually; i.e. the DQN agent perceives state every 12 frames in order to operate optimally, but information is collected every frame to record paths more coherently. Every 12 frames, the agent was given the game state information, and its action was then sent to the game engine for the next 12 frames. This resulted in visualizations with discernible paths. Paths are temporally represented with a gradient between red and blue, indicating the start and end of a path, respectively. [3]

TPG has a clearly evident policy that relies on bouncing off of walls. DQN has a policy that resembles meandering until line-of sight appears with the armour, after which a direct path is taken.

In order to discuss the paths and policies displayed by the different agents, consider spawn point 15. This point is furthest from the armour pack. Look at our champion agent from TPG trained on the biased scenario, we have Figure 5.7(c). This agent

[1]The map image background used in Figures 5.2 through 5.18 was traced directly from the map file included with the My Way Home WAD file. However, the walls in ZDoom have a hitbox that is not directly viewable in the map file. This is why it appears that the agents never collide with the wall; they are colliding with the wall's hit box.

[2]A frame skip of 12 was likely selected for the example provided with the ViZDoom documentation on GitHub because of the results found in the paper ViZDoom: A Doom-based AI Research Platform for Visual Reinforcement Learning [14]. Their results, summarized in Figure 7 of their work, show that a frame skip between 10 and 20 is optimal.

[3]A frame skip of 12 was likely selected for the example provided with the ViZDoom documentation on GitHub because of the results found in the paper ViZDoom: A Doom-based AI Research Platform for Visual Reinforcement Learning [14]. Their results, summarized in Figure 7 of their work, show that a frame skip between 10 and 20 is optimal.

visits nearly every room, seemingly every episode. Despite rarely spawning at this point, the agent still manages to reach the armour pack 45 times out of 100 episodes.

Spawning at a random point with a uniform distribution during training clearly gives agents a better idea of their goal and allows them to develop a more generalist policy. Team 38985 reached the armour pack 96 times out of 100 episodes from spawn point 15. The path team 38985 took from spawn point 15 to the armour pack are seen in Figure 5.7(d).

The DQN agent trained on the biased scenario reached the armour pack 0 times out of 100 episodes. Its paths are dense and erratic, shown in Figure 5.7(a).

The DQN agent trained on the uniform scenario reached the armour pack 1 time out of 100 episodes. Similar to the DQN agent trained on the biased scenario, this agent also wandered erratically, seemingly lost. These paths are visible in Figure 5.7(b).

Figures 5.2, 5.3, 5.4, 5.5, 5.6, 5.7, 5.8, 5.9, 5.10, 5.11, 5.12, 5.13, 5.14, 5.15, 5.16, 5.17, and 5.18 illustrate the paths taken by all four agents from each different spawn point. From this collection of figures, it is evident that all four agents have little variation in their decision making from spawn point to spawn point. Patterns are largely the same. Both TPG agents seem to have developed a "one size fits most" policy. We hypothesize that this is a result of evolutionary computation maximizing the average returns over multiple episodes during training. Conversely, DQN is particularly greedy in its reward function, with rooms enabling direct visual "sight" of the goal state resulting in success. For the Basic task, a greedy reward formulation is optimal, but not under My Way Home. This topic is discussed further in Section 5.3.1.

To better understand the pathfinding policies of different agents, a brief experiment was performed where agents were spawned in a 1024x1024 room. The room featured the same ceiling and floor texture as the My Way Home map, and the same texture on the walls as the corridor sections in the My Way Home map. However, no goal was spawned. Figure 5.19 exemplifies what the agent "sees" in this custom-made scenario. Agents were spawned in the centre of the room, facing a uniformly distributed random direction. Episodes concluded after 1000 tics, and agents were tested on 100 episodes. The X-Y coordinate of each agent was recorded every tic.

This allows us to better visualize each agent's pathfinding policy. This is illustrated in Figures 5.20, 5.21, 5.22, and 5.23. As in earlier path visualizations, the gradient from red to blue represents the start and end of a path, respectively. It is now apparent that the TPG agent has a more structured approach to room investigation when the armour is not observed (most of the time). A circular orbiting policy appears in which the underlying goal appears to be to:

1. Follow a slow continuous arc from the spawn location until a room wall is encountered.

2. Orient itself to be parallel to the wall.

3. Commence another continuous arc leading away from the wall.

The final wall arc appears to either be of one of two types, shallow or tight (biased spawn points, Figure 5.20) or always shallow (unbiased spawn points, Figure 5.21). Conversely, DQN without a visual cue demonstrates a meandering behaviour throughout (Figures 5.22 and 5.23).

The empty room experiment utilized the corridor wall texture, as seen in Figure 5.19. However, to better visualize policies in different rooms, more empty rooms were created. Each room has a different texture on the wall that corresponds with a texture on the My Way Home map. For clarity, the results of these additional room experiments have been overlaid on an image of the My Way Home map.

The resulting paths seen in Figure 5.24 demonstrate that Team 29199 reacts somewhat similarly to each wall texture it is faced with. The agent walks in circles, but the radius of the circle changes depending on which wall texture it is presented with. When looking at the wall from spawn points 10, 13, 17 and 20, the agent walks in smaller circles than it does when it sees walls from the remaining spawn points.

In contrast, the TPG agent that was trained on the uniform version of My Way Home has very similar policies when faced with different wall textures, with one large exception: spawn point 24. There is an obvious discrepancy between how this agent acts when facing the wall texture from spawn point 24. In short, the TPG agent adopts a much more direct trajectory when responding to this wall texture. Figure 5.35 shows that this agent has a consistent and direct "plan" when it encounters this wall texture in the context of the entire map. It is accustomed to approaching this

Figure 5.19: The player's point of view in the empty room experiment.

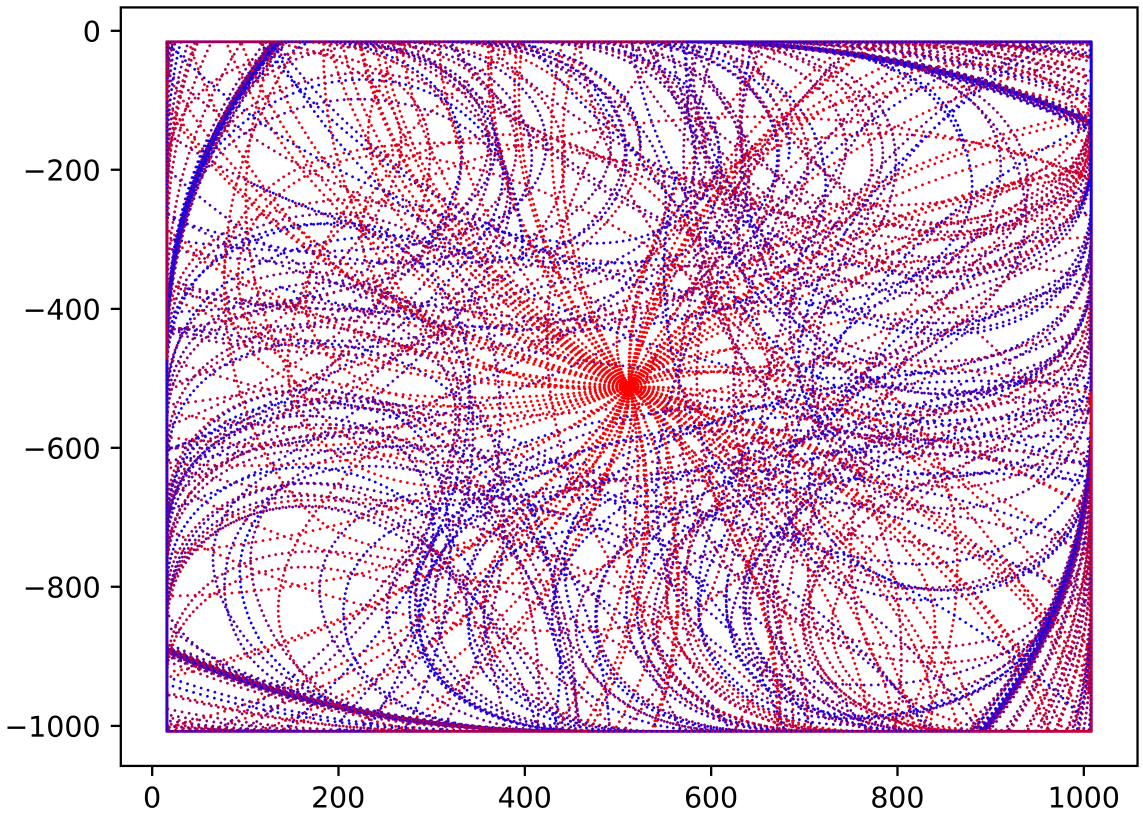


Figure 5.20: The paths taken in an empty room by TPG Team 29199; trained on the biased version of My Way Home.

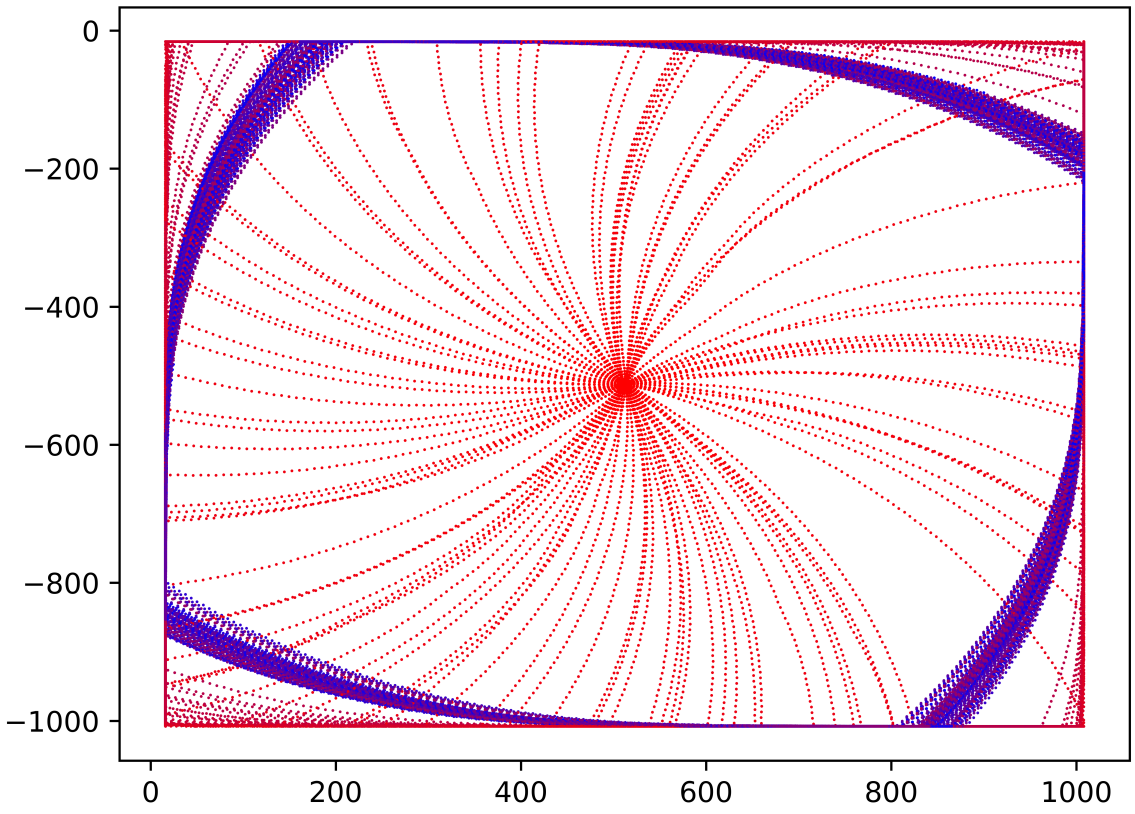


Figure 5.21: The paths taken in an empty room by TPG Team 38985; trained on the uniform version of My Way Home.

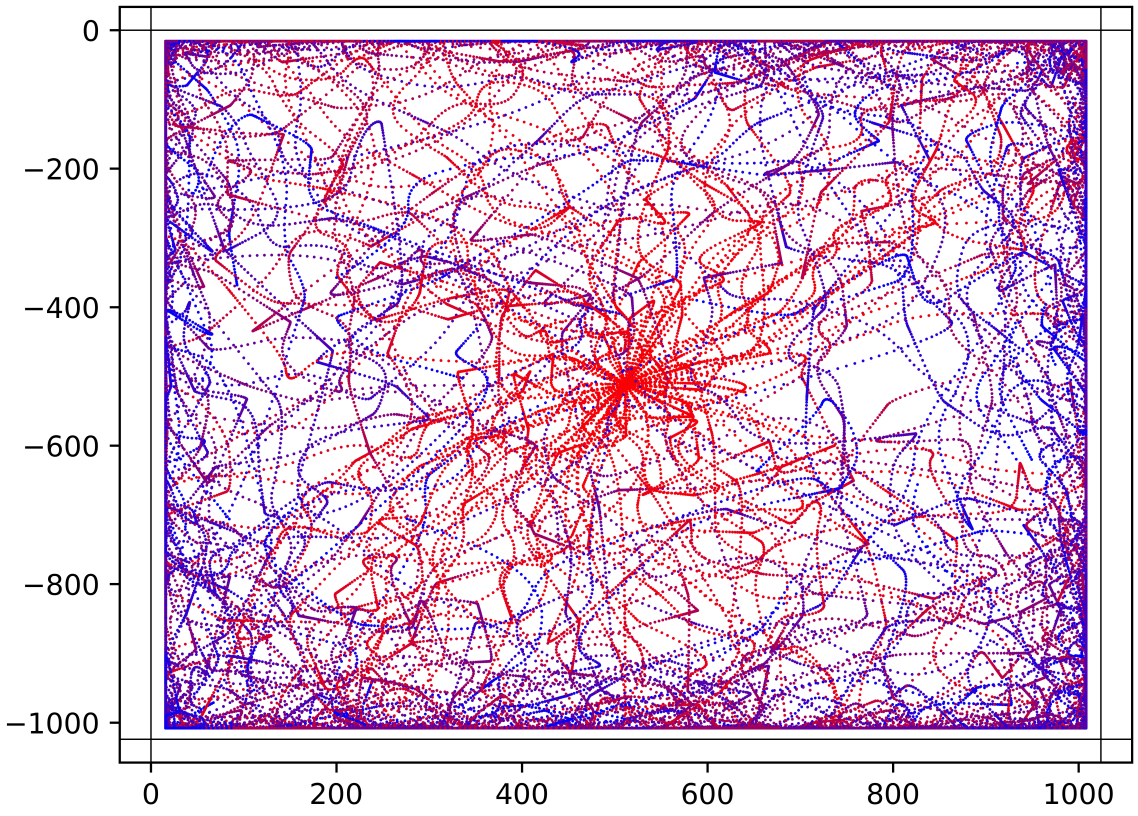


Figure 5.22: The paths taken in an empty room by the DQN agent that was trained on the biased version of My Way Home.

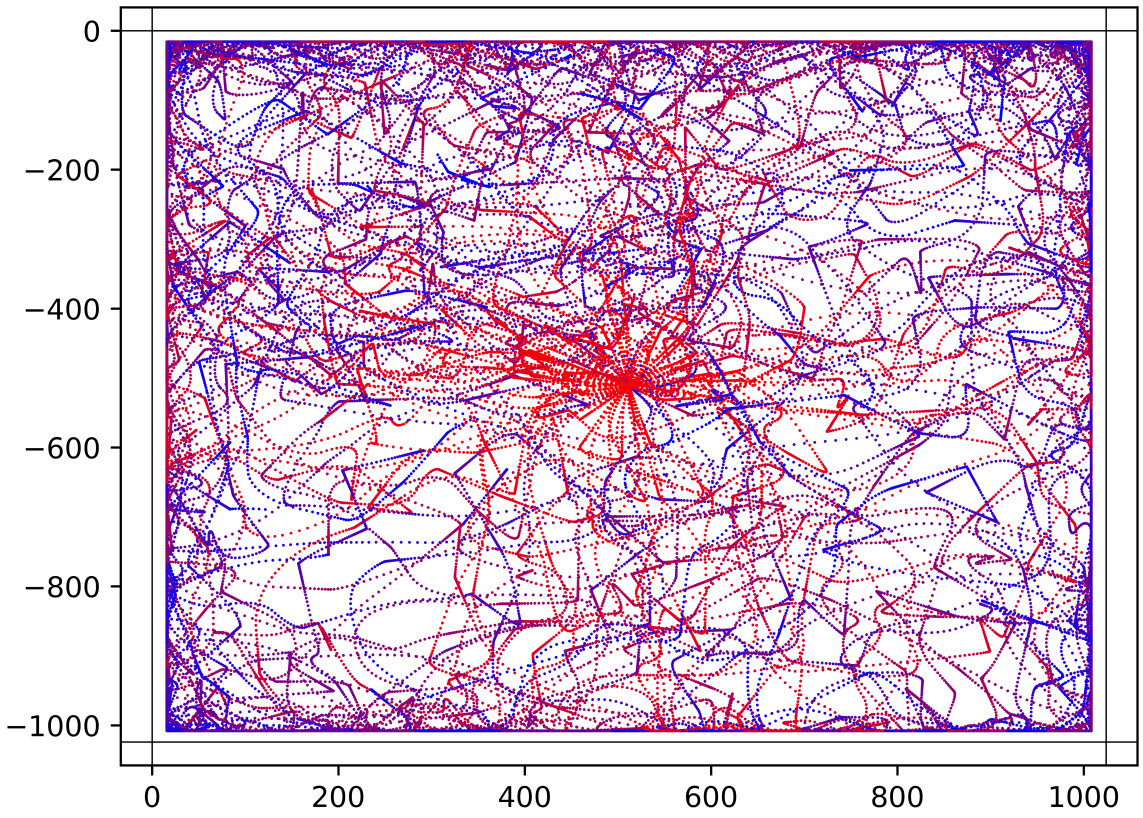


Figure 5.23: The paths taken in an empty room the DQN agents that was trained on the uniform version of My Way Home.

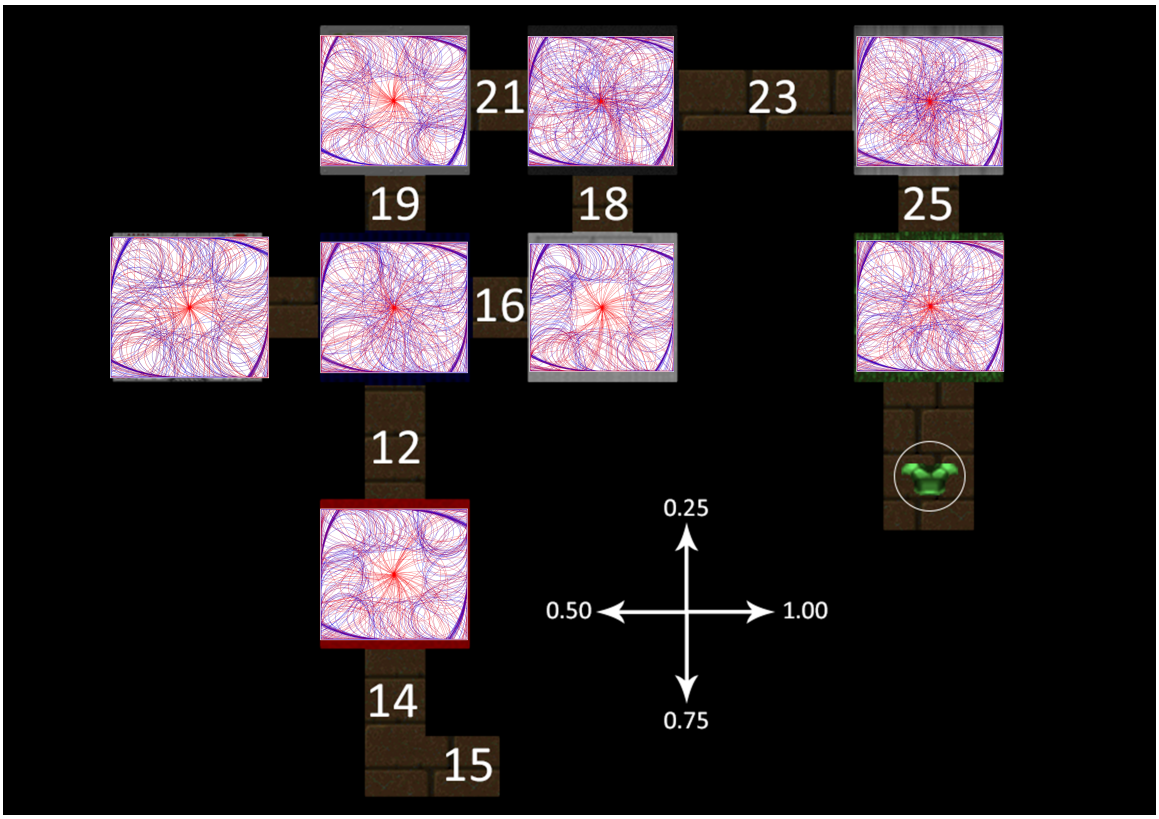


Figure 5.24: TPG Trained Biased: The paths taken when spawned in the centre of an empty room with the corresponding wall texture to the spawn point on the map.

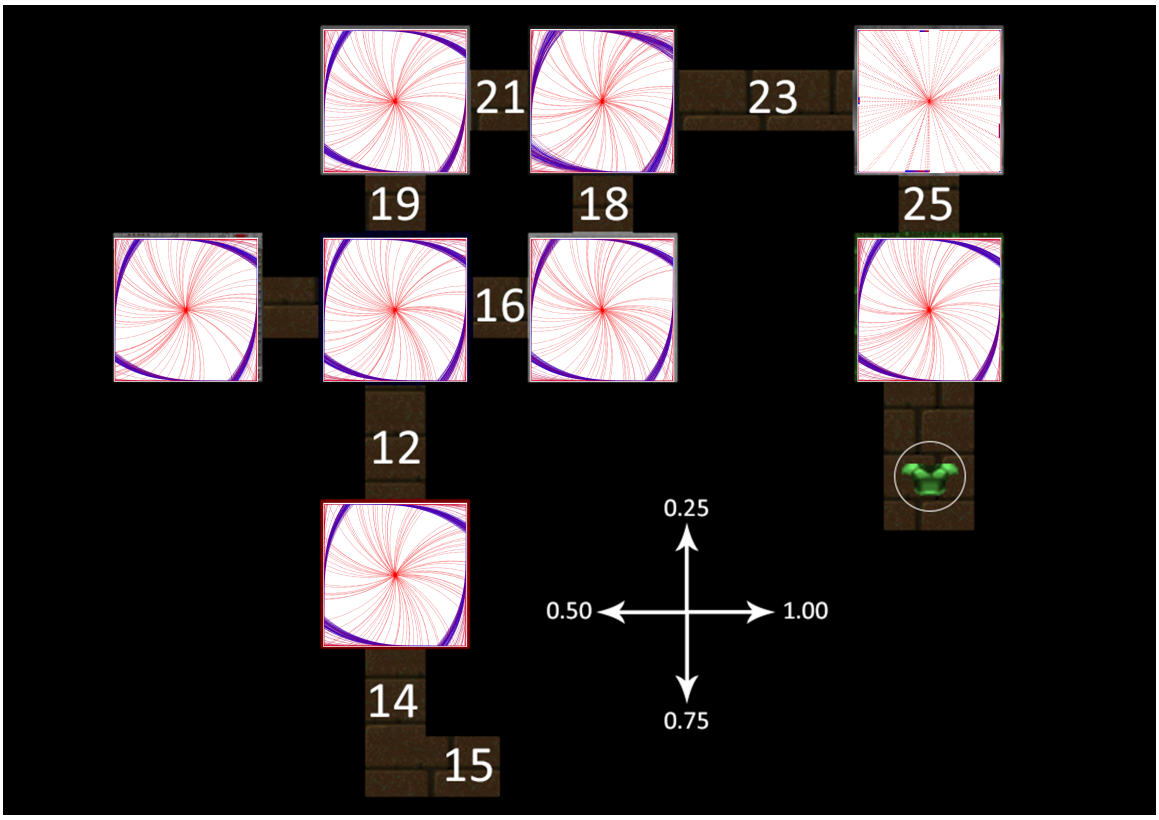


Figure 5.25: TPG Trained Uniform: The paths taken when spawned in the centre of an empty room with the corresponding wall texture to the spawn point on the map. Note that the ending points (blue) at spawn point 24 have been emphasized in post for visibility.

wall texture while travelling in the 1.0 ("east") direction. Hence, when in an empty room with the spawn point 24 wall texture, it heads directly towards the wall because it believes it is travelling "east".

5.3.1 Discrepancy in DQN and TPG Performance

DQN's shortcomings in the My Way Home task scenario are likely due to its greedy credit assignment. As stated in Section 2.1.1, 0.0001 points are removed per elapsed tic in My Way Home, with a 1 point reward for finding the armour pack. Given that DQN makes adjustments with every incremental reward, until the armour pack is found, every action simply leads to a worsening score. Compare Figures 5.12(a) and 5.12(b) to Figures 5.13(a), 5.13(b), 5.14(a), 5.14(b), 5.15(a), 5.15(b), 5.16(a), 5.16(b),

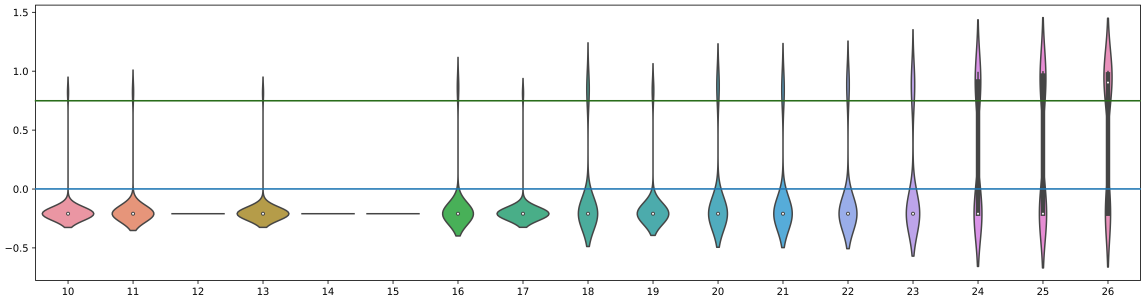


Figure 5.26: A violin plot depicting the frequency of scores achieved at each spawn point in the My Way Home task by the DQN trained on the biased version of My Way Home.

5.17(a), 5.17(b), 5.18(a), and 5.18(b). Regardless of the scenario a DQN agent was trained on, there is a clearly visible difference in the density of paths approaching the armour pack. Both DQN agents perform best when they are close to the armour.

In contrast, TPG takes a more holistic approach to credit assignment. Reward has no effect until after every individual in the TPG population has completed all of their training episodes. This rewards TPG agents for maximizing the average returns across entire episodes, all incremental rewards are ignored. DQN on the other hand, adopts a Q-value approach to accumulating rewards. Simply put, policies are developed that optimize reward accumulation between pairs of agent actions (hence also the significance of proper parameterization for frame skipping). When visual line of sight is established to the armour, particularly effective polices appear. When the armour is not present, it is more difficult to discover useful intervening objectives.[4]

Figures 5.26, 5.27, 5.28, and 5.29 depict violin plots illustrating the distribution of scores achieved by our four agents while being tested at each different spawn point. The horizontal blue line marks ($x = 0$) and the horizontal green line marks ($y = 0.79$). 0.79 is the minimum score achievable while still being successful ($0 - (0.0001 * 2100) + 1 = 0.79$). -0.21 is the lowest possible score, indicating that the player was not able to find the armour pack before its time expired. The highest score possible depends on the spawn point, since spawn points further from the armour pack will obviously require more travel time.

[4]Other gradient approaches can be effective, but more sophisticated credit assignment formulations are also necessary.

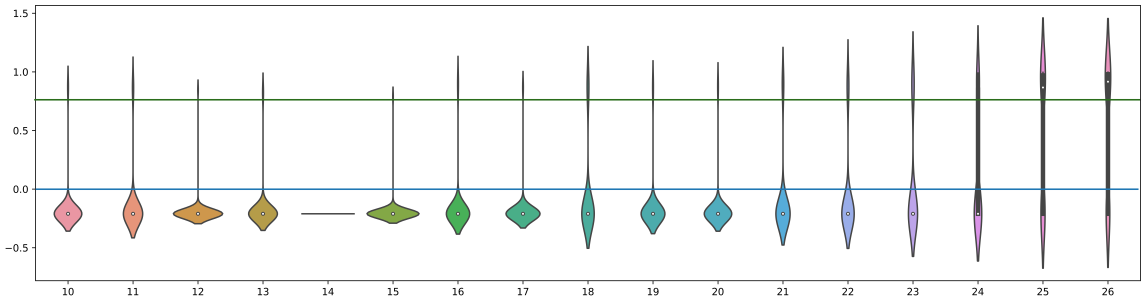


Figure 5.27: A violin plot depicting the frequency of scores achieved at each spawn point in the My Way Home task by the DQN trained on the uniform version of My Way Home.

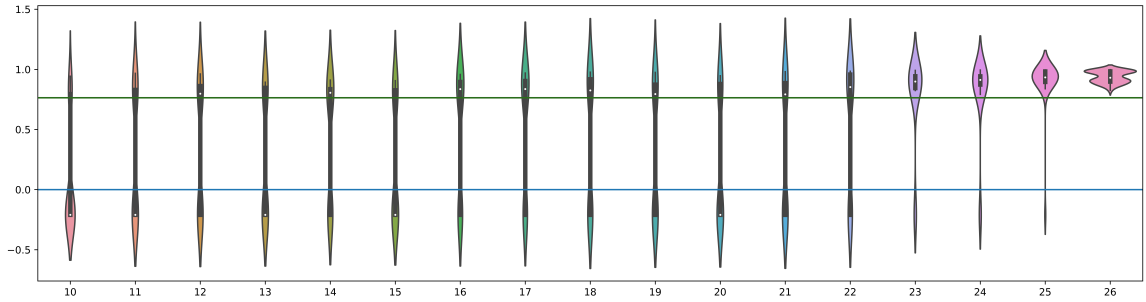


Figure 5.28: A violin plot depicting the frequency of scores achieved at each spawn point in the My Way Home task by TPG Team 29199, which was trained on the biased version of My Way Home.

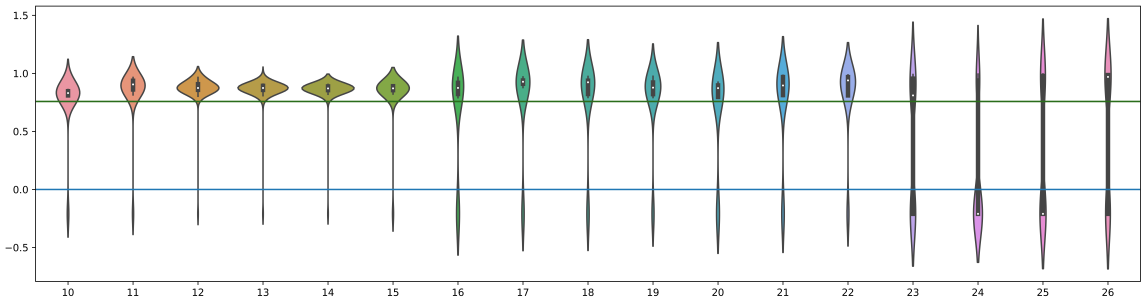


Figure 5.29: A violin plot depicting the frequency of scores achieved at each spawn point in the My Way Home task by TPG Team 38985, which was trained on the uniform version of My Way Home.

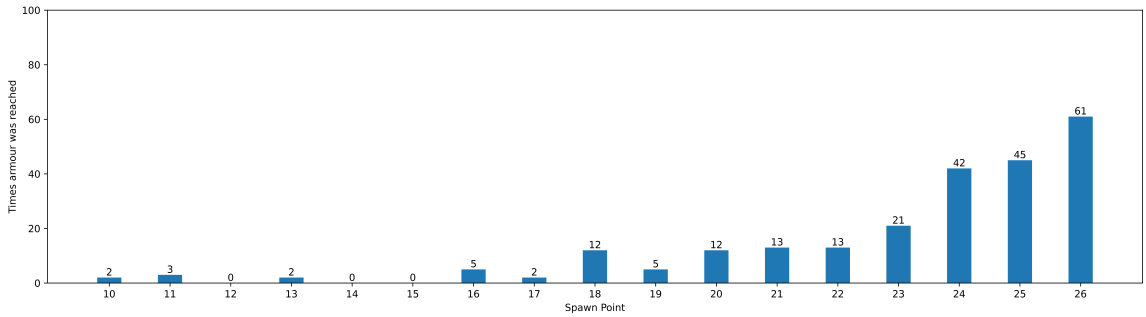


Figure 5.30: A plot depicting the number of times the armour pack was reached from each spawn point in the My Way Home task by the DQN trained on the biased version of My Way Home.

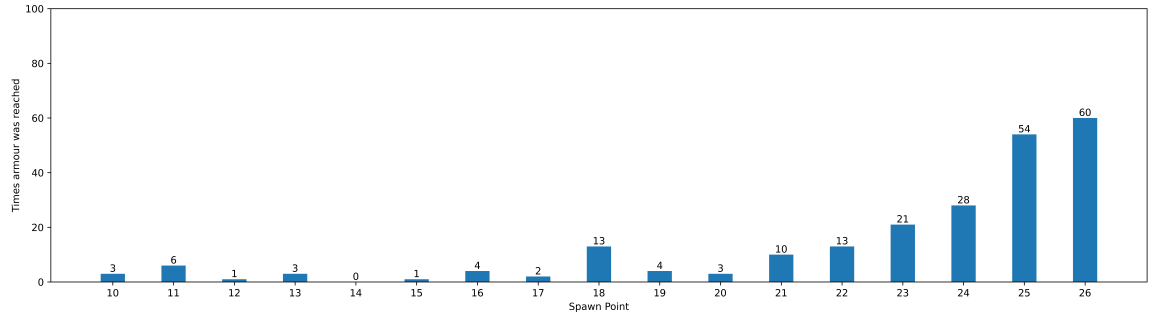


Figure 5.31: A plot depicting the number of times the armour pack was reached from each spawn point in the My Way Home task by the DQN trained on the uniform version of My Way Home.

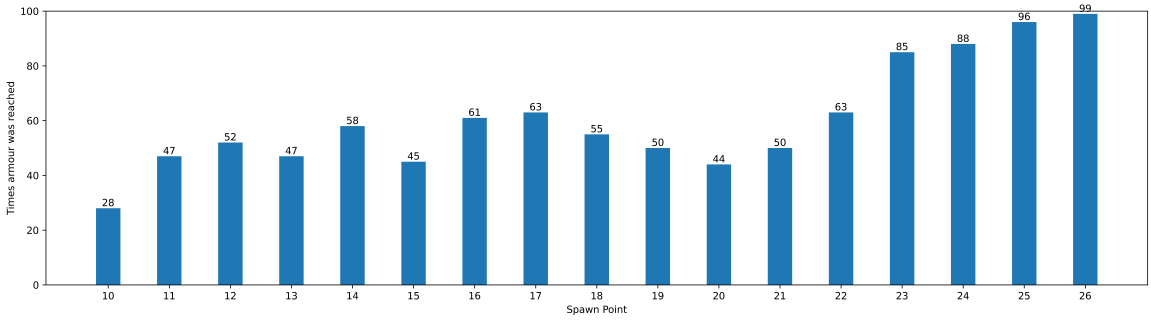


Figure 5.32: A plot depicting the number of times the armour pack was reached from each spawn point in the My Way Home task by TPG Team 29199, which was trained on the biased version of My Way Home.

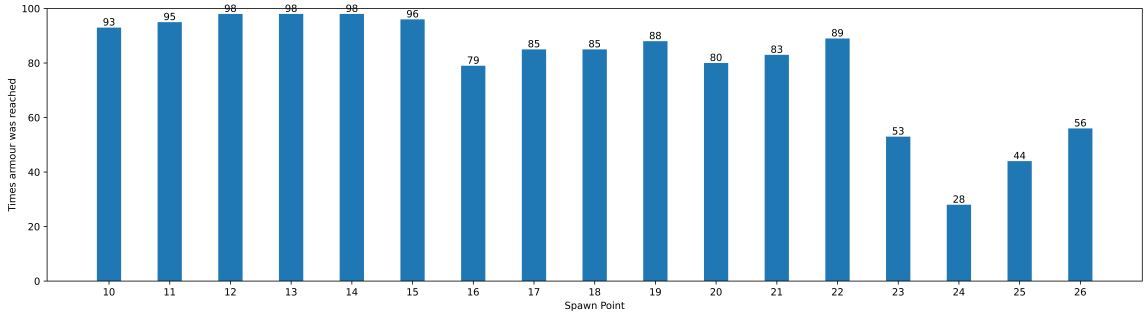


Figure 5.33: A plot depicting the number of times the armour pack was reached from each spawn point in the My Way Home task by TPG Team 38985, which was trained on the uniform version of My Way Home.

Figures 5.30, 5.31, 5.32, and 5.33 allow us to compare and contrast how often an agent was able to reach the armour pack from each room. This is a binary (yes/no) function of measuring performance compared to examining the scores of each agent; in other words, was the agent able to complete its given task? Each agent was given 100 episodes at each spawn point, where they spawned facing a uniformly distributed random angle. Scores were still tracked, but these figures only depict the number of times they were successful in reaching the armour. In Figure 5.30, it is evident that the DQN agent trained with the biased RNG is able to reach the armour from spawn point 24, spawn point 25, and spawn point 26 roughly half the time. This is much better than its performance from other spawn points. The discrepancy in performance is due to the bias in how this DQN agent was trained. The DQN agent was most frequently spawned at spawn point 24 and therefore learned best how to navigate to the armour from spawn point 24. As seen in Figure 2.3, in order to reach the armour from spawn point 24, one has to navigate through spawn point 25 and spawn point 26. Referring to Figure 5.16(a), it is clear that, while this agent sees some success from this spawn point, it still frequently wanders off to parts of the map that it rarely saw during training. Subsequently, the agent gets "lost" and is unable to find its way. Unlike the DQN agent that was trained on the biased version of the Basic task, this agent is unable to extrapolate the information it needs to find its way back to the armour pack.

The performance of the DQN agent trained on the uniform version of My Way

Home does not show much improvement. However, we cannot attribute its performance to not having been exposed to all of the spawn points equally. The spawn points were selected randomly with a uniform distribution, so the agent had the opportunity to develop a more holistic policy. As discussed earlier, the greedy nature of DQN's weight adjustment method is a likely culprit. To reiterate, DQN adjusts itself based on the current frame's reward. In My Way Home, a positive reward is only given upon reaching the armour; each frame gives a negative score to penalize sluggishness. As such, DQN likely "focuses" its efforts on spawn points that are close to the armour pack because it sees a positive reward sooner from those spawn points.

Looking at the TPG agent trained on the biased version of My Way Home, we can see that, like both DQN agents, it gets better as it gets closer to the armour. In contrast with the DQN agents, this TPG agent is able to reach the armour roughly half the time, or more, from every spawn point, with the exception of spawn point 10. This is an interesting result because the TPG agent saw spawn point 24 a severely disproportionate number of times during training. Despite training with a disadvantage, the agent was still able to develop a policy that allows it to see success relatively frequently. Figure 5.20 allows us to see this policy in action, unobstructed by walls. Its policy appears to rely on using the intersection with a wall as a guide to angle itself in a particular direction, where it then navigates in a circular pattern until it collides with another wall.

Interestingly, the TPG agent that was trained on the uniform version of My Way Home struggles with the complement of the set of spawn points that TPG team 29199 excels at. Figure 5.34 provides a visualization of where TPG team 38985 is skilled and where it is not. Spawn points 11, 20, 22, and 17 make up "the cross", (due to the cross shape made by the negative space). The cross is where team 38985 starts to struggle. By no means does it perform poorly when spawning in this area, but there is a marked decline seen in Figure 5.33 from earlier spawn points. There is another plummet in performance starting at spawn point 23. Team 38985 gets worse as it gets closer to the armour. This trend is interesting because the player has to navigate through spawn points 23, 24, 25 and 26 in order to reach the armour from any other spawn point on the map. Why is it not able to recognize where it is when spawning at those points?

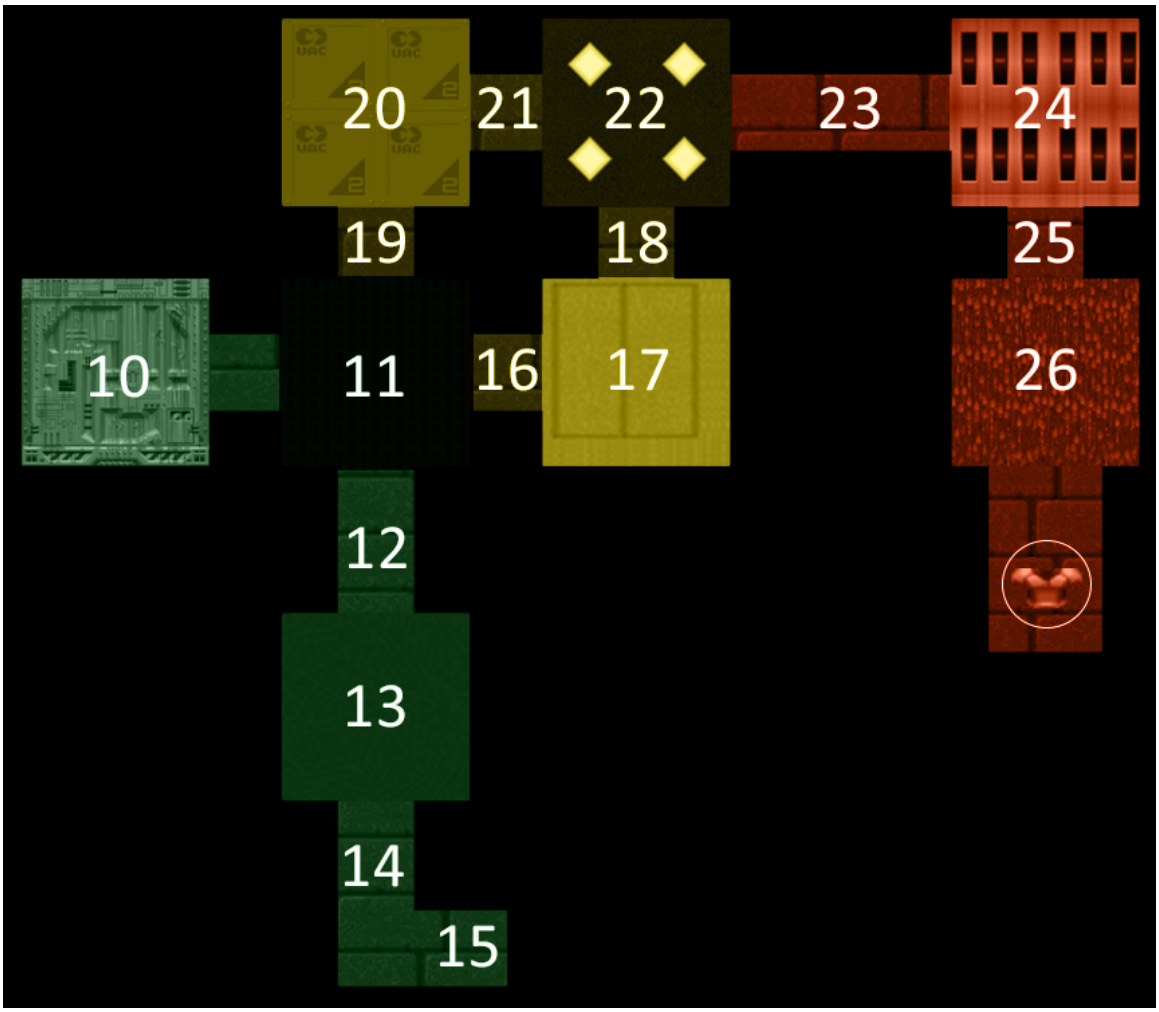


Figure 5.34: The My Way Home map with portions highlighted to illustrate where TPG team 38985 is successful and where it struggles. Areas highlighted in green are the spawn points where this individual was able to score highly, while areas in red show spawn points where this team scored relatively poorly.

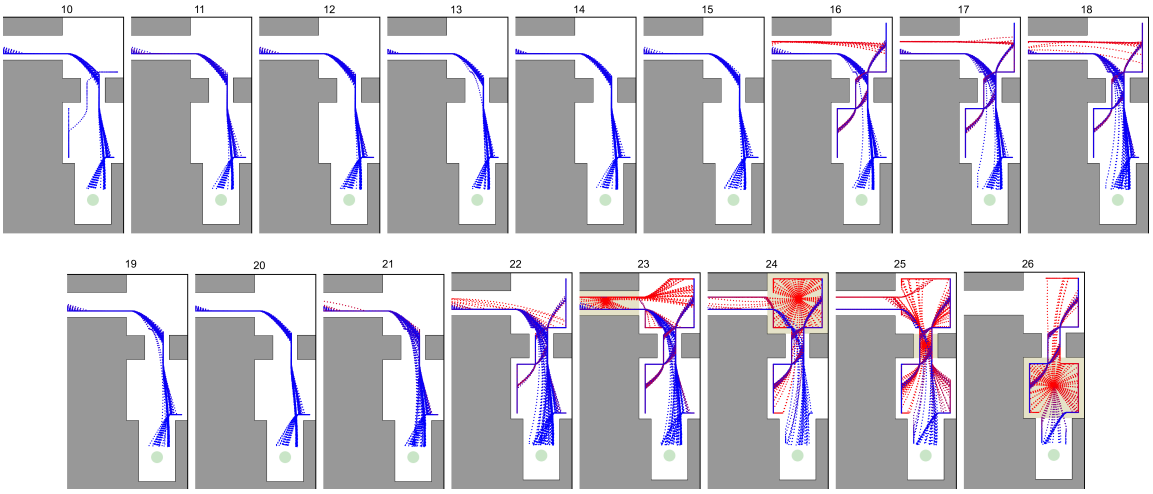


Figure 5.35: An aggregation of the end of paths taken from each spawn point by Team 38985.

One possible explanation is that the agent knows how to navigate the area highlighted in orange in Figure 5.34 when it approaches from *one* angle, but not every angle. To better visualize the evidence that supports this theory, the rightmost portion of all 17 path images has been compiled into a single image, Figure 5.35.

Team 38985 has clearly developed a policy where it is expecting to approach spawn point 24 from the bottom of the corridor that contains spawn point 23. Because Team 38985 saw each spawn point uniformly during training, we can assume that it needed to traverse through spawn point 23 in roughly 76% of the training scenarios it saw. As such, it developed a policy that is very good at traversing through spawn point 23, but that policy is not as effective in the minority of cases where it starts in or after spawn point 23.

5.3.2 TPG Agent Complexity

When examining TPG solution, there are certain properties we can examine to further quantify the emergent process by which solutions are evolved. Continuing with the two TPG individuals we have been analyzing, team 29199, trained on the biased version of My Way Home, and team 38985, trained on the uniform version of My Way Home.

Team ID	Division	Product	Sum	Difference	Conditional	Total
29199	107	156	68	86	88	505
38985	288	189	215	174	105	971

Table 5.6: Bid Programs: Comparison of frequency of instruction types in the bid programs of our TPG champion teams.

Team ID	Division	Product	Sum	Difference	Conditional	Total
29199	112	76	80	84	40	392
38985	53	53	62	58	54	280

Table 5.7: Action Programs: Comparison of frequency of instruction types in the action programs of our TPG champion teams.

In Tables 5.6 and 5.7, we see an interesting discrepancy in the overall quantity of instructions contained within each individual. Team 29199 has overall fewer bid instructions, but more action instructions.

Figures 5.36 and 5.37 depict the visual representation of how a root team (green) relate to learners (blue) and non-root teams (orange). Both of these structures are noticeably smaller than solutions seen in previous research.

Team ID	Visual Buffer	Local Registers
29199	164	341
38985	365	606

Table 5.8: Bid Programs: Comparison of input sources in bid programs from the TPG champion teams.

Team ID	Visual Buffer	Local Registers
29199	116	276
38985	64	216

Table 5.9: Action Programs: Comparison of input sources in action programs from the TPG champion teams.

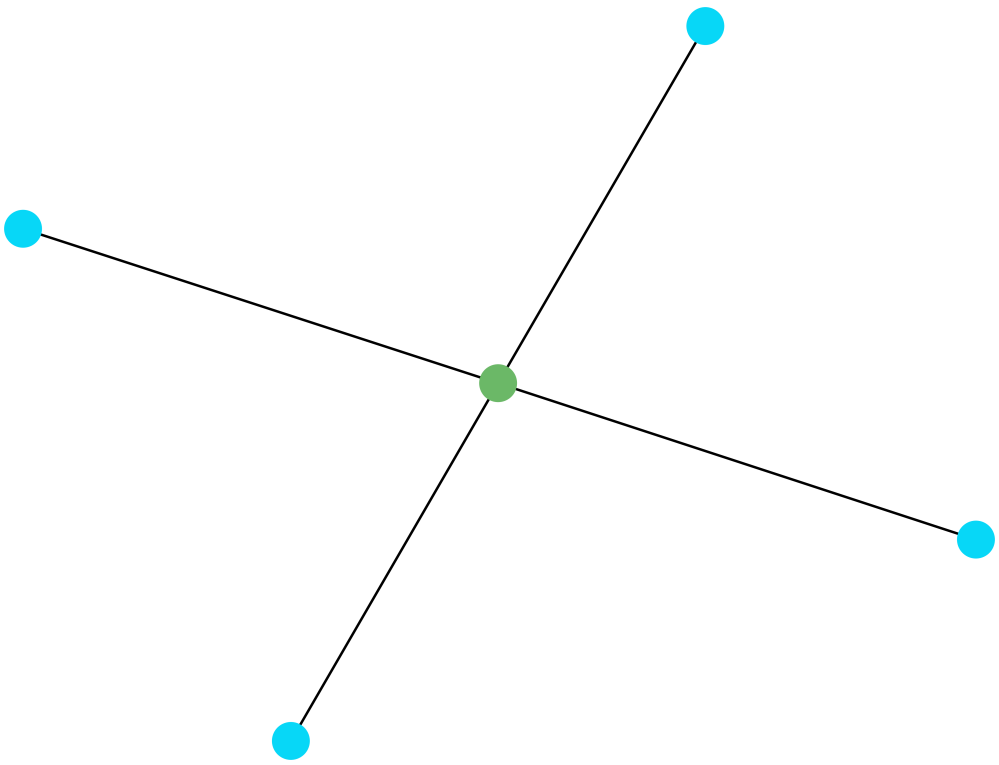


Figure 5.36: A visualization of the learner-team structure that comprises team 29199.

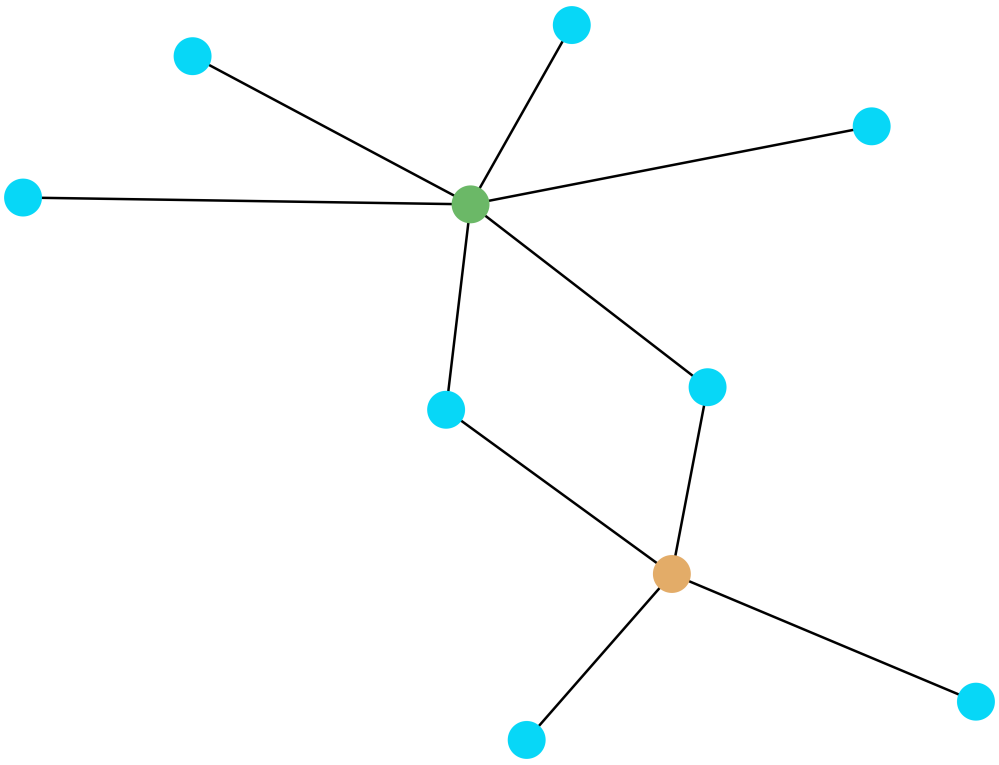


Figure 5.37: A visualization of the learner-team structure that comprises team 38985.

Chapter 6

Conclusion

Visual reinforcement learning problems are characterized by high dimensional state spaces that are typically composed from pixels alone. This places a lot of emphasis on the ability of a reinforcement learning framework to develop appropriate 'embeddings' (the deep learning approach) or perform feature construction using subsets of the feature space (the tangled program graph approach). Moreover, the DQN approach to credit assignment is based on the Q-learning formulation of temporal difference. This implies that optimal policies may result when the Markov assumption holds (complete information). Conversely, TPG may only perform credit assignment after the terminal state is encountered (episodic accumulation of average returns). We hypothesize that this will result in rather different learning biases when the agents are exposed to visual reinforcement learning tasks with different properties.

Given the above background, this thesis evaluates DQN and TPG on two tasks from the ViZDoom game engine that were chosen to exhibit complete information (Basic) versus partial information (My Way Home). We are interested in identifying to what degree DQN and TPG are able to develop effective solutions to each task. As part of this study, we also uncovered operating systems specific biases in the ViZDoom game engine itself. Specifically, under the Windows operating system, the (pseudo) uniform random number generator for agent spawn point generation appears to instead be a normal random number generator. This implies that most of the time the opponent 'monster' in Basic is initialized to the left of the agent. Under the My Way Home task the agent tends to be initialized in the top right hand side room of the labyrinth. Conversely under the Linux operating system the ViZDoom game engine does initialize spawn points using a uniform random number generator. Furthermore, we proposed an effective workaround for replacing spawn point values with value from an external uniform random generator. In this thesis we benchmark each learning algorithm under both tasks with both the VizDoom random number generator and

an external random number generator, resulting in four environments in total.

Under the Basic task DQN identified a strategy that was considerably more effective than that identified by TPG. Moreover, the DQN strategy was less sensitive to the biased initialization of monsters. At some level this was anticipated given that the Basic task is defined in terms of complete information. The Q-learning formulation of DQN in combination with the known effectiveness of a convolutional deep neural network architecture has been repeatedly shown to be particular effective under these conditions. TPG on the other hand performed considerably worse on the Basic task. Moreover, TPG was much more sensitive to the biased versus uniform sampling of monster spawn states. We attribute this difference to the requirement for TPG to learn how to index the state space in such a way that monsters could be detected across the width of the visual frame. In the past TPG has been able to achieve this, but only under training curricula that exposed TPG to the wide range of monster start states for long enough to develop suitable indexing schemes [23].

In the case of the partially observable My Way Home labyrinth navigation task, TPG was considerably more effective than DQN. Some of this we attribute to differences in the approach to credit assignment. TPG ranks candidate solutions in proportion to overall episodic performance whereas DQN attempts to develop a policy through maximizing the pairwise temporal difference in Q values. As such, with the goal (terminal) state 'in sight' DQN will be very effective at moving the agent to the goal. However, without any direct view to the goal state, DQN will at best find a policy for avoiding the most negative immediate rewards; whereas under the My Way Home task rewards are sparse, making it difficult for DQN to develop a general approach to labyrinth navigation. Conversely, TPG sparsely indexes state, forming a policy that defines navigation as a set of arcs that interact with the labyrinth in such a way that systematic movement between rooms is possible. Moreover, for the most part infinite cycles between the same rooms are also avoided. This points to on the one hand to the capacity of TPG to compose strategies using symbolic expressions based on a low number of pixels. On this other hand this reflects strategy discovery for which a modular decomposition of the task is particularly appropriate. Finally, this also reflects the bias of TPG towards accumulation of reward maximization over an entire episode as opposed to greedy goal directed policy discovery.

Future research could expand both the set of ViZDoom tasks evaluated and the types of deep reinforcement learning algorithm employed. Examples might include asynchronous advantage actor-critic (A3C) and/or Trust Region Policy Optimization methods; both of which have demonstrated improved performance over the original DQN approach to deep reinforcement learning. Other future work could include an examination of the impact of the wall textures on agent performance in the My Way Home task. To what degree are these reactive agents dependent on unique wall textures? What happens if we replace the wall texture with something they have never encountered before (such as a purely white or black wall), or replace all of the wall textures with a texture that exists in the current map? Additional future work may measure the importance of map geometry: will changing the width and or height of each section of the My Way Home map impact the reactive agents? This may provide insight as to how purely reactive these agents are, or if they have simply developed a "script", so to speak, that is effective at solving the maze.

Bibliography

[1] Caleidgh Bayer, Ryan Amaral, Robert J. Smith, Alexandru Ianta, and Malcolm I. Heywood. *Finding Simple Solutions to Multi-Task Visual Reinforcement Learning Problems with Tangled Program Graphs*, pages 1–19. Springer Nature Singapore, Singapore, 2022.

[2] Marc G. Bellemare, Yavar Naddaf, Joel Veness, and Michael Bowling. The arcade learning environment: An evaluation platform for general agents. *CoRR*, abs/1207.4708, 2012.

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

[4] Brian Christian. *The Alignment Problem: Machine learning and human values.* Norton, 2021.

[5] John Doucette, Peter Lichodzijewski, and Malcolm I Heywood. Benchmarking coevolutionary teaming under classification problems with large attribute spaces. In *Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation*, GECCO '09, page 1901–1902, New York, NY, USA, 2009. Association for Computing Machinery.

[6] John A. Doucette, Peter Lichodzijewski, and Malcolm I. Heywood. Hierarchical task decomposition through symbiosis in reinforcement learning. In *Proceedings of the 14th Annual Conference on Genetic and Evolutionary Computation*, GECCO '12, page 97–104, New York, NY, USA, 2012. Association for Computing Machinery.

[7] Thomas Haynes, Sandip Sen, Dale Schoenefeld, and Roger Wainwright. Evolving a team. In E. V. Siegel and J. R. Koza, editors, *Working Notes for the AAAI Symposium on Genetic Programming*, Cambridge, MA, November 1995. AAAI.

[8] Stephen Kelly and Malcolm I. Heywood. On diversity, teaming, and hierarchical policies: Observations from the keepaway soccer task. In Miguel Nicolau, Krzysztof Krawiec, Malcolm I. Heywood, Mauro Castelli, Pablo García-Sánchez, Juan J. Merelo, Victor M. Rivas Santos, and Kevin Sim, editors, *Genetic Programming*, pages 75–86, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

[9] Stephen Kelly and Malcolm I. Heywood. Knowledge transfer from keepaway soccer to half-field offense through program symbiosis: Building simple programs for a complex task. In *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation*, GECCO '15, page 1143–1150, New York, NY, USA, 2015. Association for Computing Machinery.

[10] Stephen Kelly and Malcolm I. Heywood. Emergent tangled graph representations for atari game playing agents. In James McDermott, Mauro Castelli, Lukás Sekanina, Evert Haasdijk, and Pablo García-Sánchez, editors, *Proceedings of the European Conference on Genetic Programming*, volume 10196 of *Lecture Notes in Computer Science*, pages 64–79, 2017.

[11] Stephen Kelly and Malcolm I. Heywood. Multi-task learning in atari video games with emergent tangled program graphs. In *Proceedings of the Genetic and Evolutionary Computation Conference*, GECCO '17, page 195–202, New York, NY, USA, 2017. Association for Computing Machinery.

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

[13] Stephen Kelly, Robert J. Smith, Malcolm I. Heywood, and Wolfgang Banzhaf. Emergent tangled program graphs in partially observable recursive forecasting and vizdoom navigation tasks. *ACM Transactions on Evolutionary Learnering and Optimization*, 1(3):11:1–11:41, 2021.

[14] Michał Kempka, Marek Wydmuch, Grzegorz Runc, Jakub Toczek, and Wojciech Jaśkowski. Vizdoom: A doom-based ai research platform for visual reinforcement learning, 2016.

[15] Ibrahim Kushchu. Genetic programming and evolutionary generalization. *IEEE Transactions on Evolutionary Computation*, 6(5):431–442, 2002.

[16] William B. Langdon, Riccardo Poli, Nicholas F. McPhee, and John R. Koza. *Genetic Programming: An Introduction and Tutorial, with a Survey of Techniques and Applications*, pages 927–1028. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[17] Peter Lichodzijewski and Malcolm Heywood. *The Rubik Cube and GP Temporal Sequence Learning: An Initial Study*, pages 35–54. Springer New York, New York, NY, 2011.

[18] Peter Lichodzijewski and Malcolm I. Heywood. Pareto-coevolutionary genetic programming for problem decomposition in multi-class classification. In *Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation*, GECCO '07, page 464–471, New York, NY, USA, 2007. Association for Computing Machinery.

[19] Peter Lichodzijewski and Malcolm I. Heywood. Managing team-based problem solving with symbiotic bid-based genetic programming. In *Proceedings of the 10th Annual Conference on Genetic and Evolutionary Computation*, GECCO '08, page 363–370, New York, NY, USA, 2008. Association for Computing Machinery.

[20] Peter Lichodzijewski and Malcolm I. Heywood. Symbiosis, complexification and simplicity under gp. In *Proceedings of the 12th Annual Conference on Genetic and Evolutionary Computation*, GECCO '10, page 853–860, New York, NY, USA, 2010. Association for Computing Machinery.

[21] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin A. Riedmiller, Andreas 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. *Nat.*, 518(7540):529–533, 2015.

[22] Riccardo Poli, William B. Langdon, and Nicholas Freitag McPhee. *A field guide to genetic programming*. Published via `http://lulu.com` and freely available at `http://www.gp-field-guide.org.uk`, 2008. (With contributions by J. R. Koza).

[23] Robert J. Smith and Malcolm I. Heywood. Scaling tangled program graphs to visual reinforcement learning in vizdoom. In Mauro Castelli, Lukás Sekanina, Mengjie Zhang, Stefano Cagnoni, and Pablo García-Sánchez, editors, *Genetic Programming - 21st European Conference, EuroGP 2018, Parma, Italy, April 4-6, 2018, Proceedings*, volume 10781 of *Lecture Notes in Computer Science*, pages 135–150. Springer, 2018.

[24] Robert J. Smith and Malcolm I. Heywood. Evolving dota 2 shadow fiend bots using genetic programming with external memory. In Anne Auger and Thomas Stützle, editors, *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO 2019, Prague, Czech Republic, July 13-17, 2019*, pages 179–187. ACM, 2019.

[25] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An introduction*. MIT Press, 2nd edition, 2018.

[26] Dennis G. Wilson, Sylvain Cussat-Blanc, Hervé Luga, and Julian F. Miller. Evolving simple programs for playing Atari games. In Hernán E. Aguirre and Keiki Takadama, editors, *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 229–236. ACM, 2018.

[27] Marek Wydmuch, Michał Kempka, and Wojciech Jaśkowski. Vizdoom competitions: Playing doom from pixels. *IEEE Transactions on Games*, 2018.