# QTRB: TEAM-BASED REGION BUILDING USING Q-LEARNING TO DERIVE POLICY ON PROGRAMS PARAMETERIZED BY LOCAL REWARD SIGNAL

by

Noah Sealy

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

at

Dalhousie University
Halifax, Nova Scotia
April 2023

© Copyright by Noah Sealy, 2023

## Table of Contents

## List of Tables

## List of Figures

## List of Abbreviations Used

| | | |
|---|---|---|
| EC | Evolutionary Computation | 2 |
| GA | Genetic Algorithms | 11 |
| GP | Genetic Programming | xiv |
| NEAT | NeuroEvolution of Augmenting Topologies | 82 |
| QTRB | Team Based Region Building with Q-learning | xiv |
| RGP | Reinforced Genetic Programming | 24 |
| RL | Reinforcement Learning | xiv |

## Abstract

While attempting to solve 2-dimensional grid world maze tasks, it was observed that genetic programming is limited by its random initialization and no use of local reward. This thesis proposes a hybrid algorithm called QTRB, team-based region building with q-learning, which attempts to integrate genetic programming and reinforcement learning to use local reward during evolution. During evolution, QTRB constructs programs based directly on local environmental reward; programs are then passed to a reinforcement learning agent to learn on as a model. QTRB was tested to solve variously sized 2-dimensional maze tasks, hypothesizing that policy can be derived from an agent learning from this model. The results suggest that QTRB can derive policy on the given tasks, with fewer direct environment queries than traditional q-learning as the task size scales.

## Acknowledgements

I'd like to start this section by acknowledging the wisdom, support, and guidance of my supervisor, Dr. Malcolm Heywood. I am beyond grateful for the opportunity you have provided, and cannot stress enough that this work would not be possible without you.

Special thanks to Dr. Khurram Aziz, Dr. Mohammad Etemad, Dr. Christian Blouin, Dr. Joseph Malloch, and Dr. Raghav Sampangi for the many opportunities at Dalhousie. I'm also very grateful for my professors throughout my time here, especially Dr. Simon Gadbois, Dr. Vlado Keselj, and Dr. Thomas Trappenberg for inspiring me to pursue AI and learning systems.

I'd also like to thank the NIMS lab for always having a desk open at any hour of the day (or night). A very special thanks Ryan Amaral and Amous Qiu for offering lots of help throughout my research.

I'd like to acknowledge the passion and grittiness of the team at Virtual Hallway; from the founders to the devs, and everyone in between. I am very thankful for the opportunity you have all provided and the accommodations as I wrote my thesis in parallel.

I'd like to lastly thank my friends and family for all of the love and support throughout my endeavours. I attribute my work ethic to them and ultimately my competitive drive to my brothers. Finally; I am especially grateful for my mum and dad (and aunts) for always having a *spectacular* amount of belief in anything I do.

Thank you, I hope you enjoy :)

# Chapter 1

# Introduction

## 1.1 Definitions

There are a few concepts that require defining before stepping into the current discussion. This section will provide those definitions. It should be noted that all definitions are with respect to the current project, and will be used throughout the rest of this thesis. This section will be kept brief; more formal definitions can be found in the Background chapter.

### 1.1.1 Reinforcement Learning (RL)

Reinforcement Learning (RL) is a family of machine learning algorithms typically focused on solving control tasks. The learner, or agent, aims to derive a policy which maps states of the task to actions [24]. It inherits the psychological theory that the frequency of an agent's behaviour is proportional to the reward that the behaviour receives. Thus, the agent creates policy with respect to environmental feedback, typically some sort of local reward. Unlike supervised or unsupervised learning tasks, RL implies that a learning agent actively interacts with a task environment. The underlying goal is for the agent to maximize the cumulative reward received from the environment.

There exists a subset of RL algorithms in which the agent learns from a model, known as model-based RL [24]. The model can be anything that gives the agent some observability of the task. A model can range from simply knowing the next local reward, to a fully simulated abstraction of the task itself. Naturally, the learning agent might alternate between using the model to reduce the number of interactions with the environment (planning) and actively extending the model to incorporate new properties of the environment (learning).

### 1.1.2 Local Reward

This project heavily investigates environmental reward signals, specifically local reward. Local reward simply refers to the feedback each state yields in a given task environment. The local reward is therefore the reward immediately received when in a particular state. For example, a goal state may yield a very high local reward, whereas a state that puts the learner in danger may yield a very low, or even negative local reward. A central question of this thesis is whether local rewards can be used to incrementally adapt the representation of GP agents as deployed in 2-dimensional grid worlds.

### 1.1.3 Genetic Programming (GP)

Genetic Programming (GP) is a family of algorithms from the field of Evolutionary Computation (EC). EC algorithms consist of a population of solutions evolving over a sequence of generations [2]. This evolution is inspired by the biological concept of natural selection [5].

Unlike most machine learning, GP maintains multiple candidate solutions (agents), or the population $P$. At each generation some subset of the population survives, defining a parent pool, $PP$. A breeder model of evolution is assumed, so the offspring pool is defined by first choosing $(P - PP)$ agents from the parent pool and cloning them. Variation operators modify the cloned parents to define offspring. The union of the parent and offspring pools represents the new population. The process is elitist if the environment and performance function are non-random. Traditionally, the variation operators are stochastic.

### 1.1.4 Hybrid Algorithms (RL + GP)

In the context of the current project, hybrid algorithms refer to algorithms that in some way integrate components of both GP and RL.

## 1.2 Machine Learning Algorithms

There are three main components to consider when designing a machine learning algorithm [6].

1. Representation. This refers to how the algorithm expresses a solution. Traditionally, GP initializes individuals using some form of stochastic process for choosing instructions from an instruction set (often designed with a particular application in mind). However, the representation itself is re-configurable. This means that the topology of a solution's representation is subject to modification as well as the parameters. Thus, the instructions, arguments, order of instructions, and the number of instructions are all subject to variation. Many GP representations have been proposed that have these properties such as tree-structured GP [14], linear GP [4], push GP [20], grammatical evolution [18], Graphs [13] [1], Cartesian GP [17].

2. Performance or fitness function. This research has a specific formulation in RL scenarios in which the general objective is to discover a policy that maximizes the cumulative reward a learning agent receives during the training episode. An episode is defined as the sequence of interactions between the task environment and learning agent that take place between the start and terminal conditions. One of the premises of traditional approaches to RL is that the local reward received as a result of the learning agent interacting with the task can be immediately used for credit assignment. Conversely, GP only performs credit assignment after the terminal condition is encountered. Addressing this limitation for the specific case of grid world navigation tasks represents a goal of this thesis.

3. Credit Assignment. This refers to the process by which credit is assigned to different parts of the candidate solution's representation. There are two competing aspects to credit assignment: exploration versus exploitation. Greedy credit assignment implies that a single variation (to the agent's representation) is adopted that improves the performance of the agent. Stochastic credit assignment accepts changes to an agent's representation that do not necessarily result in immediate improvement in performance. In practice, credit assignment is never fully greedy or stochastic. GP is a population-based search in which $P$ solutions are simultaneously maintained. Thus, the first instance of credit assignment identifies which candidate solutions survive and reproduce. The second instance of credit assignment defines how offspring are constructed.[1] The final form of credit assignment establishes who from the parent

[1]For example, crossover interchanges program snippets between two parents, and mutation randomly changes instructions in an offspring.

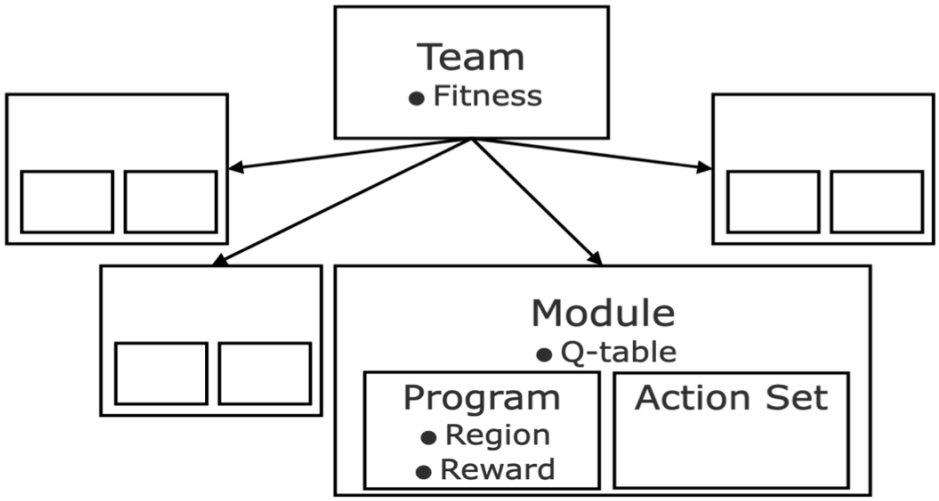


Figure 1.1: A high-level depiction of the model developed in the current project. A program holds information about the environment in the region as well as a reward. A module holds a program and a corresponding action-set, along with some information for RL. The team holds a variable number of modules as well as a fitness score, which acts as a candidate solution in evolution. A champion team will contain modules with regions that form a path from the start of the grid world to the goal.

and offspring pools will appear in the next population.[2]

The current project proposes a machine learning algorithm which integrates genetic programming and reinforcement learning, which (as mentioned), are referred to as hybrid algorithms. The current project focuses on investigating representation and credit assignment with respect to the field of hybrid algorithms.

The representation of the project will be limited to solving 2-dimensional grid world tasks. Within these tasks, teams of *modules* containing a parameterized instruction set and corresponding actions act as candidate solutions. The project's representation will be summarized in Section 1.4, but a general depiction of the model can be found in Figure 1.1.

In the scope of the current project, credit assignment is the method by which a team's module is parameterized by local reward. Moreover, teams are incrementally constructed, module by module, over multiple generations. This use of a local reward

[2]For example, the performance of the offspring is measured and only the fittest $P$ of both parent and offspring pools survive.

aims to provide novel findings in the field of hybrid algorithms.

Where the evolution is built somewhat specifically for the given tasks, the RL elements of QTRB are much more similar to traditional RL. The champion teams from GP will act as a model for traditional Q-learning to use for learning, rather than the actual environment itself. Q-learning is used to assess the quality of the entire model as currently developed by the learning agent. Conversely, local reward will be used to parameterize modules as they are incrementally added to candidate solutions. The following sections will discuss the difference in representation and credit assignment of traditional GP and the proposed approach, in order to illustrate the thesis's questions and hypotheses.

### 1.3 Traditional GP

Traditional GP algorithms possess two limitations that the current project aims to address; random initialization and no use of local reward. These limitations prevent programs from carrying the contextual meaning of the local environment, making it difficult for such algorithms to solve control tasks without the use of subgoals.

The issues in question are demonstrated during the implementation of GP on a simple $5 \times 5$ maze task. The goal of the task is to maneuver through the obstacles from the starting state to the goal. Canonical GP assumes that the representation employed by candidate solutions:

1. decomposes the state space of the task into different local regions, and

2. assigns an action to each local region that is able to move the agent across the corresponding region.

However, candidate solutions are randomly initialized, thus there is no guarantee that:

1. regions actually align with the properties of the grid world, or

2. the actions suggested by the agent are capable of 'navigating' a local region.

Canonical GP instead assumes that the population of candidate solutions is able to sample enough different parameterizations of the task for some of the region-action pairs to align with those of the task. The lack of environmental context in this initialized population does not give the programs enough environmental meaning to solve the task.

Downing addressed the action to region selection problem by using Q-learning to select actions to regions using local rewards [7]. However, random initialization of candidate solutions was still assumed, thus the resulting decomposition of the state space still resulted in regions that did not reflect properties of the grid world. Solutions could only be found if meaningful subgoals were included, i.e. agents were rewarded for partially solving a task. The agents providing partial solutions were retained and used as "building blocks" for generating new agents. This thesis aims to avoid the use of any type of subgoals.

Now that the problem has been demonstrated, this section will introduce a solution which the current project implements.

## 1.4 Proposed Solution

This thesis addresses the discussed issues relating to both representation and credit assignment in traditional GP. A team of modules, which involve programs, will be deployed to solve a 2-dimensional grid world task. In regards to representation, a program is constructed in a non-random way; building a region to represent a section of the grid world. In regards to credit assignment, a fitness value is assigned to each program using an environmental reward signal, or local reward.

As mentioned, the project's representation is shown in Figure 1.1. The following is a breakdown of each component of the figure. Finer detail relating to the representation, and algorithm, can be found in the Methodology chapter.

### 1.4.1 Program

The program directly parameterizes a portion of the grid world in a locally consistent way. It carries the actual contextual meaning of the grid world as defined by the local reward provided by the task environment within a region. The program's region itself acts as a small abstraction of the world.

The program's instruction set is what parameterizes the region. In the specific case of 2-dimensional grid worlds (the focus of this thesis), each region, depending on the selected axis, takes the form of a range in x (y) and a fixed value in y (x). These values must be positive integers and cannot exceed the number of cells in the grid world. Thus, the region represents a 1-dimensional sliver of the grid world itself.

### 1.4.2 Module

A module pairs a region to the subset of actions necessary to traverse the region. Put another way, the actions represent those actions that produce the region under the delimiting feedback provided by the environment in the form of local rewards. Note that modules need not lead the agent in the direction of the overall objective. Modules are just guaranteed to represent locally consistent regions of the grid world state space and define the action pairs that are sufficient for navigating such a local region.

### 1.4.3 Team

The team is a collection of modules. A champion team should compile these modules into an abstraction of the world, where a subset of modules creates a pathway (of regions and associated actions) from the start of the task to the goal. Once a module is added to a team that encounters the goal state, the team has the potential to form a path from start to goal. RL is then performed on the modules comprising this team's representation in order to:

1. discover a sequence of modules and actions necessary to complete the path, and

2. determine the efficiency of such a path.

Such a process assumes that the modules comprising a team represent a plan. All the credit assignment is performed relative to the modules comprising the plan. Thus, there is no interaction with the environment during team optimization. Environment interactions only take place to parameterize the properties of each module.

## 1.5 Research Questions

This section will present some research questions relating to the identified shortcomings of traditional GP, as well as a proposed solution:

1. Are local rewards sufficient for parameterizing states and actions into useful modules from which entire policies can be constructed?

2. Once a module encounters the goal state, is Q-value propagation sufficient for optimizing the plan defined by the set of modules comprising the team?

3. What implications are there for the sample efficiency of the resulting population of teams?

The thesis aims to answer these questions by testing the performance of the proposed QTRB with various metrics. The Background chapter will provide hypotheses in the scope of these questions after presenting a more thorough review of the GP, RL, and hybrid algorithms. The Methodology chapter will then discuss the specifics of the algorithm's implementation and the tasks it was tested to solve. Finally, the Results (and Appendix) and Conclusion chapters will present and analyze the algorithm's performance, as well as discuss limitations and potential future work for this project.

# Chapter 2

# Background

There exist several algorithms which integrate genetic programming (GP) and reinforcement learning (RL) to solve a variety of tasks. This project aims to expand on the field and fill in some gaps while preserving principles that previous research shares. This chapter will present a summary of prior work in the field of GP, RL, and hybrid algorithms in order to set a foundation for the current project.

An example will be followed throughout the background section to further illustrate the basic mechanics of the concepts presented. This example depicts a robot finding a way to walk back to its docking area, after being dropped at the start of a maze. Walking back consists of many paths, some containing obstacles that may be dangerous to the robot. The parameters of this example will be defined specifically for each concept presented with it. It should also be noted that the analogies used in this example are not fully representative to the exact process of each concept but are simply to provide a higher level conceptual understanding of the mechanisms at play.

## 2.1 Project Overview

Rather than integrating GP or RL into an existing algorithmic counterpart, this project builds from the ground up with the two concepts in focus. This project investigates parameterizing GP fitness using environmental reinforcement signals (local reward), to evolve an individual with a representation, or model, that RL can use to derive a policy. This is all to investigate the use of both reinforcement signals and model-based RL when integrated with GP. This project hypothesizes that an algorithm of such nature will not only successfully solve grid world tasks, but solve them in fewer environment queries than other model-free, or GP-free, RL algorithms.

This project aims to contribute novel findings to the field of integrated GP and RL systems. This will be achieved by showing how environmental reinforcement signals can be utilized to evolve models for RL to learn policy on. The following sections

provide a summary of background research in the fields, to highlight the general nature of these integrated algorithms, as well as some gaps in research in the field that this project aims to fill.

## 2.2 Evolutionary Algorithms

Evolutionary computation (EC) is a research area of computer science that deals with individuals evolving toward solutions [2]. This research area is modelled after the biological principle proposed by Darwin known as natural selection [5]. A population of individuals compete for survival based on a function that relates to how capable they are of solving a given task. The individual's capability, known as fitness, drives the population to converge onto solutions. The following example is typically the model an EC algorithm is based from [9]:

1. Randomly initialize a population of individual candidate solutions. Initialize a generation counter at zero.

2. Select individuals based on assigned fitness functions within the scope of the solution. The lesser-fit individuals are removed as candidate solutions. The surviving solutions constitute the parent pool for the next step.

3. From the parent pool, sample and clone individuals until the original population size is reached. This is the offspring pool. Variation operators are then applied to the offspring pool to define individuals that are "genotypically" distinct from the parent pool. The variation operators can transfer parts of the genotype between pairs of offspring (sexual operator) or be limited to a single individual (asexual operator). [1]

4. If the stop criterion is not met, go back to 2 and increment the generation counter.

5. If the stop criterion is met, stop, and choose the fittest candidate as the champion solution. A typical stop criterion is a limit on the number of generations.

These higher-level EC concepts can be easily visualized following the example of the robot walking home.

Rather than focusing on just one individual robot, EC functions by assessing the performance (and evolution) of a whole population of robots. In order to manage this population, the fitness function is established. The specifics of such a function are

[1] Genotype refers to the representation assumed for defining individuals.

irrelevant for this example except that it favours the robot who solves the task, with a bonus for avoiding danger.

Each candidate robot is tested against the fitness function. Through this, each will derive a solution for the task. Once the whole population has found a solution, those whose solutions had the highest-scoring fitness functions are selected. These selected robots reproduce, implying the application of the aforementioned variation operators, such as the crossover of genes, as well as gene mutation. The union of the parent and offspring pools defines the robot population for the next generation. This cycle of life continues until some external stop criterion is reached; perhaps the experiment is only run for 50 generations.

Two subfields within the field of evolutionary computation are genetic algorithms (GA) and genetic programming (GP); both will be discussed below.

### 2.2.1 Genetic Algorithms (GA)

First proposed by Holland [11], one of the distinguishable features of GAs is the use of crossover for producing variation in offspring [2]. The most classical examples of GA also represent individuals as bit-string encodings of the solution [9].

GA algorithms are the most aligned with the general principles of EC; they are biologically sound and intuitive [8]. The general structure of a GA closely follows the structure of an EC algorithm. GAs usually deal with fixed systems. In order to create offspring during evolution, a fixed number of parents are randomly paired up for reproduction. For some of these pairs, crossover randomly occurs. In order for crossover to be applied, "gene alignment" takes place. Given that all individuals have the same number of genes, then gene alignment is merely a process of aligning pairs of genes at the same location between each parent. Offspring resulting from this reproduction will go through some bit mutation, then replace the parent population [9].

GA excels at optimizing solutions on high-dimensional tasks, but only with the correct selection of the individual's representation length [11]. Individuals in GA algorithms are represented by fixed-length strings, to emulate the representation of a genotype. The length of the string determines the different number of possible genotypes within the population. The goal of GA is to find the optimal phenotype, or combination of bits in the string, by evolving based on the fitness of the phenotype.

GA sets a strong foundation within the realm of ECs [2], but representing genotypes as fixed-length strings is not always ideal for finding solutions. Koza argues that a GA individual's fixed-length encoding of a solution can severely limit their observability into it [14].

String length is an especially problematic parameter for some tasks as the optimal representation of an individual to solve the task is unknown before it is found [14]. Koza argues that without this prior knowledge, individuals need the freedom to reshape their representation as they evolve. On top of this, restrictions set to the representation of the genome can further limit the way individuals observe the task.

### 2.2.2 Genetic Programming (GP)

GP is an extension of GA [2], proposed by Koza, which offers a solution to the representation problem appearing in GA. As mentioned, in GA, fixed-length strings are used to represent individuals. GP, on the other hand, allows individuals, called programs, to assume a variable length genotype. Thus, the number of instructions comprising individuals do not need to be the same. The general structure of GP algorithms follows that of GA.

To return to the robot walking home example, "GP-based robots" would follow a very close process to the "EC-based robots". The main difference would be how the specific thinking and processing of their candidate solutions. The GP robots developed by this thesis, for instance, will assume a modular approach to constructing the representation. Different modules will then be added throughout offspring development. Moreover, these robots will be modular in a task-specific way. Robots will define a module by applying an action relative to their current location. The "dimension" of the module will be a function of the rewards received from the environment. This represents a very important distinction from previous research applying GP to RL tasks.

The algorithm designed in this project takes on an approach of GP that most closely resembles that of Linear GP (LGP). LGP is a form of GP which implements solutions using linear representations of programs, rather than the popular tree-based approach [4].

### Linear Genetic Programming (LGP)

Brameier & Banzhaf's original intention with linear GP (LGP) was to develop a new form of GP representing programs as a series of instructions from an imperative programming language [4]. Brameier & Banzhaf found that LGP could perform on par compared to neural networks on classification and generalization tasks with a standardized set of health care data [4].

Specifically, individual programs were represented as simply C code snippets of varying lengths [4]. The code snippets were based on an instruction set which could operate on two variables, or one and a constant value. The instruction types involved arithmetic operations, conditional branches (inequalities), and function calls.

The importance of linear genetic programming is that it shows that programs do not explicitly require tree-based representations. Brameier & Banzhaf introduce an important paradigm to consider throughout this project; as the programs built from the GP proposed in this project do not conform to the traditional representation of tree-based GP [4]. The programs in this project are represented closer to the programs found in LGP.

Additionally, the conditional branch (inequality) instruction type is especially important to the foundation of this project's GP. The programs used throughout the current project only operate using inequality instruction; these take on the form of checking if an agent is within a given module's region of the environment. This type of instruction type is important while passing on the context of the real environment, as the inequalities are able to carry information relating to a given state.

### Optimization vs Modelling

In Eiben's work [9], an observation is made that many other types of EC solutions are used for optimization, while GP is used for finding fit models as solutions.

The difference between optimization and modelling is based on what is known about the problem before implementing a solution. Optimization problems involve a known model and output. The goal of optimization is to find the parameters for the prior model such that a performance criterion is satisfied. A GA would represent a suitable choice for optimization tasks because the number of parameters for optimization has already been set [9]. On the other hand, modelling problems

only know the input and output. The goal is to find a formula or prediction tool (model) that maps the given input to the given output [9]. GP represents a suitable choice for modelling because the number of parameters and the parameter values are unknown.

Although Eiben was explicitly referring to tree-based GP at the time, this can also be applied to LGP. LGP aims to manipulate an instruction set which solves the problem; it creates a model which maps inputs to outputs (i.e. model is synonymous with program where the instruction set is given, but the sequencing and number of instructions is unknown). In the current project, the model is an instruction set for an RL agent to use to navigate the world. It is important to build this instruction set as it acts as an abstracted version of the environment that is detailed enough for the RL agent to form an accurate policy.

In the current project, the sole purpose of using GP is to search for a decomposition of the environment (modules) that an RL agent can learn from as a substitute for the direct environment. The paradigm of using LGP as a tool for building a model aligns with its use in this project. Although the output is not necessarily explicitly known, the agent is used to not just learn a policy to solve the task, but also to verify that the model produced by GP was parameterized in a way which allows for a solution to be found.

## 2.3 Reinforcement Learning (RL)

Reinforcement Learning (RL) is a branch of artificial intelligence that predominately deals with giving a learner, known as an agent, a control task to learn in order to solve. The fundamental theory comes from reinforcement learning in the field of psychology. The idea follows that the frequency of an individual's behaviour is directly proportional to the amount of positive (or negative) feedback assigned to that behaviour.

As mentioned, the individual learner in RL is known as the agent. RL differs from many other machine learning paradigms as an RL agent does not learn from a given data set (regardless of the label). Instead, the RL agent learns from direct (or indirect) interactions with the given environment (or model of the given environment). The environment is the task where the agent is deployed, it provides a reinforcement

signal, known as a reward, from which the agent learns. To derive rewards from the environment, the agent moves from state to state, transitioning with actions. This RL loop moves through discrete time steps starting from $t = 0$.

This section will first cover various introductory concepts of reinforcement learning, before introducing the concepts and algorithms that were used for this project.

### 2.3.1 Policy

The goal of the reinforcement learning agent is to maximize the amount of reward signal received from the environment [24]. If configured correctly, this maximized reward signal is typically found by solving the given task within the environment. After learning, the agent represents their found solutions as policies. The policy is a mapping of states to actions [24], in other words it is the strategy the agent learns to solve the given task. The policy, whether deterministic or stochastic, is the foundation of the behaviour of the agent; if a possible state is input to the policy, it will output the agent's corresponding action [24].

A stochastic policy is formally defined as the probabilities of actions being executed given a state. As the agent learns, the policy's distributions over each state should sample actions in such a way that solves the given task.

The policy, in the case of the robot example, simply determines what the robot will do given the state it finds itself in. The optimal policy for the robot would yield the best actions given each state throughout its walk home. As the robot learns, the ultimate goal is to find an optimal policy. This optimal policy may involve the robot taking some sort of path that maximizes the amount of positive reward the robot can collect on its way home.

The foundation of the agent's objective throughout learning should consist of deriving a policy which solves the given task.[2]

### 2.3.2 State, Action, Reward, and the Markov Decision Process (MDP)

The environment of the task which the agent is deployed into is composed of a set of states. Upon visiting each state, the environment returns a new state, as well as some reward signal for visiting that state. The reward establishes the ability of the

[2]The case of value functions in RL will be discussed below.

agent to learn through reinforcement learning. The relationship between the agent, the states, and reward can be described using a Markov Decision Process (MDP) [19]. As shown in Figure 2.1, the key to this MDP is that the next state of the environment is purely dependent on the previous state and completely independent of anything else.

Each step along the path on the walk home from the robot example can be considered a new state. Each new state has a distinct reward signal depending on the context of the given state. For example, a state with a dangerous obstacle will yield a negative reward signal, while a safer state will yield some positive reward. The goal state, the robot's docking area, will yield a much larger amount of reward signal than anything else. This large reward is indicative of the agent, in this case, the robot, "winning", or solving, the task.

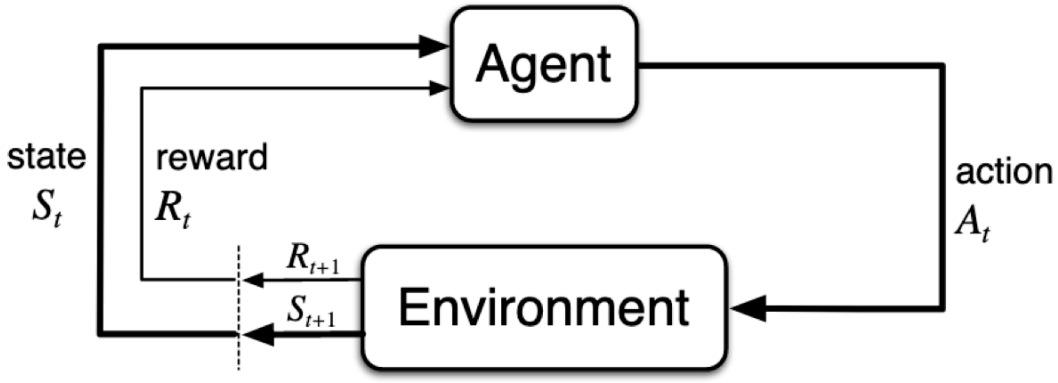


Figure 2.1: Shown is a Markov Decision Process. It is observed that the environment returns a new state and some reward signal based solely on the action of the agent. Cited from [24]

Figure 2.1 describes not just the relationship between state, action, and reward for the agent and environment, but also highlights key elements of the general reinforcement learning algorithm. These have been touched on previously, but the rest of the section aims to formally define them.

In RL, the agent is the main learner. It is deployed into the environment with the goal of deriving a policy (state action mapping) that possesses the capability of solving the given task. As mentioned, the environment is the set of states which the agent interacts with. The environment is typically comprised of a start state, goal

state, and various states in between, each with their own assigned reward signals.[3] The time step $t$ marks each interaction between the agent and environment, given a system of discrete time steps. The action $A_t$ is the selected action of the agent at time step $t$, executed onto the environment. The state, $S_t$ and $S_{t+1}$ represent the starting and resulting state of the given action at time steps $t$. The rewards, $R_t$ and $R_{t+1}$ represent the reward signals yielded by a particular state at time step $t$ and time step $t + 1$.

### 2.3.3 Value Approximation and Learning

Another key element of reinforcement learning is the approximation of the value an agent assigns to each state while learning. The foundation of learning depends on the agent's perception of the value of a given state, this is referred to as the value function.

The value function represents the agent's estimate of the long-term value a state has under a specific policy. It can be generalized to judging the relative value of the current state relative to its neighbouring states. The value function differs from a reward signal as a reward signal is immediate and does not quantify the quality of the state relative to its neighbours. Beyond that, the reward signal does not define how close to the goal the state is. The reward signal might just quantify the difference between a positive (no fail) versus a negative (fail) state. Though value function differs from reward signal, it also depends on it. The reward signal itself can be somewhat misleading, as it lacks the foresight to tell if a state is a good long-term solution; it simply gives what that exact state is independent of neighbouring states or context.

Following the robot agent example, the reward signal alone may not be sufficient feedback to learn the best possible path to the docking bay. As stated before, the best way home, or the optimal policy, should find a path that efficiently covers states to maximize the amount of positive reward collected. Judging path quality on just reward signal fails when the robot is presented with a path that has an immediately safe path, and another slightly more dangerous path that actually leads to a much faster way home. If every decision is based on immediate reward alone, the robot will

[3] In the scope of this thesis, these state-level reward signals are considered local reward.

always choose the path with no danger, losing out on the much larger reward in the other path. The value function acts as a method of assigning the larger reward path value, despite seeming as an immediate danger locally. When the robot's learned policy is based on the trajectory of each state, rather than the immediate reward signal alone, the obvious choice is the path with a slightly higher chance of danger, but much larger reward.

Expanding off of the value function, it is also useful to know the estimate of the value returned for executing a particular action on a given state. This would show the trajectory of value the given agent has in the current state while executing the suggested action. This is known as the action value function, denoted as $Q(s, a)$ [24]. If the action value function is chosen under some policy $\pi$, it is denoted as $Q_\pi(s, a)$. The action value function, or $Q(s, a)$ represents the quality of a given state-action pair. There are many ways to estimate the action value function, Q, using reinforcement learning; the method used in this project is known as Q-learning. Q-learning optimizes its Q-values through the use of Temporal Difference (TD) learning; both concepts will be briefly discussed in the following subsections.

### 2.3.4 Temporal Difference (TD) Learning

Policy iteration is a form of dynamic programming which involves a set of algorithms used to improve an agent's policy. Temporal difference (TD) learning is a policy iteration method where an agent learns from experience [22].

Monte Carlo search is another popular form of policy iteration, where the value function is updated at the end of the run. TD Learning will instead update the value function on each new time step. This key difference between TD and Monte Carlo methods is quite monumental when it comes to improving the speed of convergence in RL methods [24].

Following the work on Sutton and Barto [24], the general formula for value function updates in TD Learning is: $V(S_t) \leftarrow V(S_t)+\alpha[R_{t+1}+\gamma(V(S_{t+1}))-V(S_t)]$. Algorithm 1 describes a general TD Learning update following the same authors [24]. The given example performs a value function update at every time step.

Rather than reflecting on the path taken once the robot reaches the docking bay, TD learning involves the example robot reflecting and updating its beliefs on what

**Algorithm 1** Example of general TD learning update. Cited from [24]

Input: the policy $\pi$ to be evaluated
Algorithm parameter: step size $\alpha \in (0, 1]$
Initialize $V(s)$, for all $s \in S^+$, arbitrarily except that $V(terminal) = 0$
**for** each episode **do**
    Initialize $S$
    **for** each step of the episode **do**
        $A \leftarrow$ action given by $\pi$ for $S$
        Take action $A$, observe $R$, $S'$
        $V(S) \leftarrow V(S) + \alpha[R + \gamma V(S') - V(S)]$
        $S \leftarrow S'$
        **if** $S$ is terminal state **then**
            break
        **end if**
    **end for**
**end for**

makes a good path every step of the way. Although there is a lot more updating, the robot is able to converge to an idea of an optimal path sooner, with a finer understanding of the path itself.

### 2.3.5 Q Learning

Q-Learning is an off-policy reinforcement learning algorithm that is used to efficiently estimate Q-value (action value function) with the use of TD-Learning. Unlike other reinforcement learning algorithms, Q-learning does not learn directly using the policy. Q-learning will instead approximate the optimal action-value function with the learned action-value function [25]. Q-learning will essentially iteratively update each action-value function until they converge to the optimal Q-value.

Similar to TD updates, this example update from the work of Sutton and Barto [24] demonstrates the update used in Q-learning. $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + max(Q(S_{t+1}, a)) - Q(S_t, A_t)]$. Additionally, a sample Q-learning algorithm is shown in Algorithm 2 [24]. Overall, this highlights the importance of TD learning in the

Q-learning algorithm, as well as provides context into the method of estimating state-action values used in this project.

**Algorithm 2** Example of a general Q-learning algorithm. Cited from [24]

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\epsilon > 0$
Initialize $Q(s, a)$, for all $s \in S^+, a \in A(s)$, arbitrarily except that $Q(terminal, *) = 0$
**for** each episode **do**
    Initialize $S$
    **for** each step of the episode **do**
        Choose $A$ from $S$ using policy derived from $Q$ (such as $\epsilon$-*greedy*)
        Take action $A$, observe $R$, $S'$
        $Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma max_a Q(S', a) - Q(S, A)]$
        $S \leftarrow S'$
        **if** $S$ is terminal state **then**
            break
        **end if**
    **end for**
**end for**

Q-Learning simply involves the example-robot learning directly from its environment. The robot will act on past experiences throughout its walk home.

### 2.3.6 Model-Based Planning

To introduce an algorithm such as DynaQ, Sutton and Barto first differentiate model-free and model-based RL. The primary component of model-free RL is learning, while model-based relies on planning [23]. Although there are some differences between both planning and learning, they both mainly revolve around updating value approximations based on feedback from the state space.

The model can be anything the RL agent is able to use to predict the behaviour of the real state space. That is, given a state and action input, the model can predict the real environment's state and output [24]. In the scope of the current project, an abstraction of the state space found in GP is used as the model. The GP part of the

proposed algorithm is in some ways simply planning the RL agent's model.

Within the context of their work, Sutton and Barto define planning as "any computational process that takes a model as input and produces or improves a policy for interacting with the modelled environment" [24]. These processes share two common traits:

1. These processes involve deriving policy by approximating value functions.
2. The value functions are derived through simulated experience [24].

Taken from the same discussion, Algorithm 3 shows how an algorithm which uses planning can converge to an optimal policy. This is evidence enough that a model-based, or planning, algorithm is suitable for an agent to learn a policy within a given MDP.

**Algorithm 3** Generalized tabular Q-planning algorithm, shown to converge to the optimal policy. Cited from [24]

1: **while** true **do**
2: Select a state, $S \in S$, and an action, $A \in A(S)$, at random
3: Send $S, A$ to a sample model, and obtain a sample next reward, $R$, and a sample next state, $S'$
4: Apply one-step tabular Q-learning to $S, A, R, S'$:
$Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma max_a Q(S', a) - Q(S, A)]$
5: **end while**

### DynaQ

DynaQ is a reinforcement learning algorithm that adds a planning mechanic to traditional Q-learning. The algorithm will continuously both plan from a model as well as directly approximate the value function.

The algorithm will store the returned state and reward of actions within a model so that when they are queried again, the model can simply return the recording. During planning, Q-learning is randomly applied only on state-action pairs which are saved in the model. This way, the model is learned from real experience, which can then be used for simulation [24]. Direct action onto the environment allows for both contributing to the model and the value function approximation. During training,

both direct action and planning will occur continuously, contributing to the value function approximation.

Taken from Sutton and Barto's work, Algorithm 4 shows the general DyanQ algorithm [24]. It is also worth mentioning that DynaQ without any planning steps is simply Q-learning. From the experiments of Sutton and Barto, DynaQ was able to solve a grid world maze task in significantly fewer steps than a non-planning agent (Q-learning) [24].

**Algorithm 4** General DynaQ algorithm. Cited from [24]

Initialize $Q(s, a)$ and $Model(s, a)$ for all $s \in S$ and $a \in A(s)$
**while** true **do**
  $S \leftarrow$ current (nonterminal) state
  $A \leftarrow \epsilon\text{-}greedy(S, Q)$
  Take action $A$: observe resultant reward, $R$, and state, $S'$
  $Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma max_a Q(S', a) - Q(S, A)]$
  (e) $Model(S, A) \leftarrow R, S'$ (assuming deterministic environment)
  **for** Repeat $n$ times **do**
    $S \leftarrow$ random previously observed state
    $A \leftarrow$ random action previously taken in $S$
    $R, S' \leftarrow Model(S, A)$
    $Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma max_a Q(S', a) - Q(S, A)]$
  **end for**
**end while**

A planning algorithm such as DynaQ acts as a good benchmark for comparison with the current project's implemented algorithm. Both algorithms involve developing a model, whether it be from planning or state-space abstraction, for a Q-learning agent to derive a policy from.

Overall, these algorithms are hypothesized to reduce the number of direct environment steps taken to learn policy compared to traditional Q-learning. Thus, the amount of direct environmental queries during learning stands as a good metric while comparing traditional tabular Q-learning, DynaQ, and QTRB.

## 2.4 Hybrid Algorithms

As the following agent example will show, although the GP robots were able to cover a lot of ground faster, the RL robot would be able to have finer control over optimizing its way home. For example, the fitness of an individual robot may not account exactly for the level of safety in each individual state, whereas the value function of a learning RL agent may. There is an issue with finding such a fine value function and policy though, as the robot must take its time on the walk home, pondering every small signal yielded by its environment. This shows the trade-off of using both styles of learning but also gives rise to how they may potentially be able to complement each other.

There have been numerous studies relating to hybrid models that combine some form of GP and RL to solve tasks. The remainder of this section will discuss some notable observations from a survey of these works, in the scope of the current project.

### 2.4.1 Drugan

Drugan's survey work in the realm of hybrid algorithms sets a good stage for the work previously published in this field. Other than surveying the field for notable hybrid algorithms, Drugan offers to classify algorithms which use EC to assist RL. A notable classification from this list of subtypes is model-based RL algorithms that learn from an associated model of the environment with GP [8].

### 2.4.2 Maravall et al.

Maravall et al. introduce a hybrid algorithm of RL and GP in [16]. They propose a method of solving obstacle-based 2d control tasks with L-shaped multi-linked robots. These robots used Q-learning, which learned to optimize a situation-action-based lookup table derived through means of an evolutionary algorithm [16]. The use of the situation-action lookup table was to provide reference to the RL agent to counter the curse of dimensionality problem; thus, allowing the agent to learn the environment, without having to waste many resources exploring solutions for larger problems.

Maravell et al. [16] suggest some noteworthy rationale for proposing a hybrid algorithm. They argue that although both RL and GP alone have many advantages,

they both fall short in areas where one can help the other.

Although RL is a good choice for learning on online applications, it suffers when the state space is too large.[4] As mentioned, most RL algorithms learn by slowly stepping along the state space directly. The mentioned learning takes an increasingly long time as the state space increases. In other words, classical RL is afflicted by the curse of dimensionality.

Evolutionary computation, on the other hand, provides powerful offline optimizers, especially in the types of high dimensional problems where RL tends to struggle. Given this advantage, EC struggles with online applications.

It is apparent from Maravell et al. argument that one type of algorithm tends to thrive where the other struggles, and vice versa. This rationale provides support for most applications of hybrid algorithms.

### 2.4.3 Downing

One of the most influential studies (for hybrid algorithms) comes from Downing [7] who combines tree-based GP with reinforcement learning to solve grid world problems. The tree-based solutions would decompose the state space into regions using inequality operators, where each terminal node would contain a choice between two actions. Downing identified each action on the terminal nodes as state-action pairs, which could be updated with a value approximation according to some reward signal from the environment.

Figure 2.2 shows an example of an environment decomposed into a tree-based representation to further demonstrate Downing's Reinforced Genetic Programming (RGP). Figure 2.3 shows the corresponding program for this representation. As shown, the environment is broken down into various regions according to the given program. The terminal nodes on the program's tree contain two actions to navigate the region that their branch of the tree occupies.

The representation of Downing's programs allowed for an important modification to the applied Q-learning. The tree-based decomposition of the state space typically reduces the number of possible states an agent can be in, as being within the range

[4]Deep reinforcement learning algorithms represent an alternative approach for scaling to high-dimensional state spaces, with the trade-off of having a high computational cost.

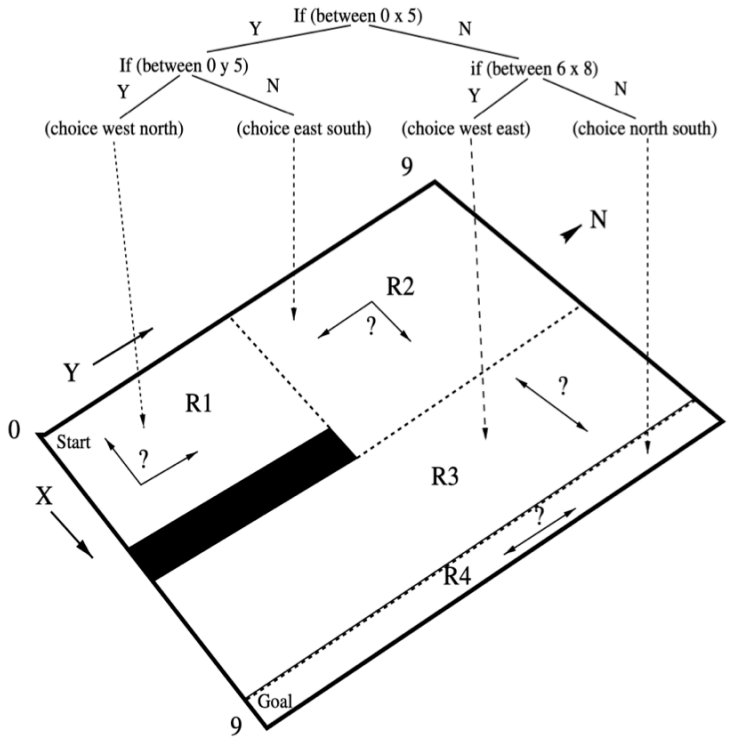


Figure 2.2: Example tree-based environment decomposition from Downing's Reinforced Genetic Programming [7]

of a region is considered within a single state. Due to this, instead of making a TD update at every time step, the update occurs at every transition of the program's branch. The amount of information stored on the Q-table is reduced, going from the environment's $length \times width$ to the varying amount of regions the programs evolve into.

With this, Downing states the purpose of both evolution and learning in this algorithm. Evolution decomposes the state space into regions, forming a new representation of the environment for learning; while learning itself finds the best action for each of the evolved regions.

Although traditional RL was able to outperform RGP in the grid worlds, Downing hypothesizes this was due to the small nature of tasks he used for experimentation [7]. Despite this, Downing does state that RGP provides benefits to both GP and RL. The benefit relevant to this project is that GP can be used with RL to create proper state abstractions [7]. As search spaces become larger, these state abstractions have

```
(if (between 0 x 5)
  (if (between 0 y 5)
    (choice (move-west) (move-north))       R1
    (choice (move-east) (move-south)))      R2
  (if (between 6 x 8)
    (choice (move-west) (move-east))        R3
    (choice (move-north) (move-south))))    R4
```

Figure 2.3: Program corresponding to Figure 2.2 from Downing's Reinforced Genetic Programming [7].

the power to reduce the amount of exploration required by the agent itself.

RGP relates to this project as it shows that GP can be used as a conduit between the environment and RL in order to reduce the complexity of the environment. In fact, one of the main advantages Downing points out is the savings this state abstraction has over the typical large search spaces of traditional RL on large-scale mazes [7]. To do this, the evolved programs must have some sort of context in the environment that is greater than randomly initialized programs. Downing, however, was not able to move beyond the use of randomly initialized programs. This meant that in order to find solutions, it was necessary to introduce subgoals into the mazes for providing additional intermediate levels of credit assignment.

This work gives rise to programs that store the context of the environment and select an action based on such context. This project expands on this concept, using an alternative approach, which is the rationale behind the region-building algorithm presented in the Chapter 3. In order for reward signals to dictate evolution, the evolution must in some way reflect the given context of the environment. This context allows evolution to decompose the environment in such a way that RL is more efficient in both time and space.

### 2.4.4 Iba

Iba et al. [12] argue that some learning simulators for real robots are imprecise, and that to capture the noise of real life, a robot must learn directly from its environment for at least some learning. Due to this real robot component of learning, the simulator where the rest of the learning first occurs does not need to be as precise as it would be without direct environmental learning.

The methods of Iba et al.'s algorithm start with a population of programs learning the task on a simplified simulation of the environment. RL is then conducted, using the fittest program to act as the agent. This allows a decomposition of the environment, established during GP, to be passed onto the RL agent to speed up learning. Similar to the current project, the RL algorithm used throughout the study is Q-learning.

This project is important to the current work as it shows an interesting speed-up in keeping RL out of the main evolution loop. Rather, it takes the champion of the given evolution process and uses only this to train on a more direct implementation of the environment. The champion program also has an advantage from evolution, as it has a decomposed plan of a simulated version of the environment. This reduces the time that would be otherwise spent in time step by time step exploration by a classical RL agent. This work introduces a core concept for the current project; using GP to plan and RL to optimize.

### 2.4.5 Elfwing et al.

Elfwing et al. trained a hierarchical RL (MAXQ) agent to solve various foraging tasks on real robots [10]. Hierarchical RL involves designing a number of subtasks based on a larger task for an agent to learn a policy of the larger task. The difference between normal hierarchical RL and this study was that rather than a human subtask designer, Elfing et al. used GP to design the sub-tasks [10].

It is also mentioned that the evolutionary search in GP constructs an abstracted representation of the state space, which is increasingly useful as the number of dimensions in the state space increases [10].

Similar to Iba's work, Elfwing et al. show an abstracted form of planning in order to make RL more efficient.

### 2.4.6 Mabu et al.

Mabu et al. propose an algorithm called Genetic Network Programming (GNP), which is a fundamental extension of GP, where programs are represented as graphs. A GNP program is comprised of judgement and processing nodes. Judgement nodes will determine which node is executed next based on some condition while processing

nodes will execute an actual action [15]. Nodes are preserved between programs, the genes for each program in the population are different combinations of node connections. This creates programs that are compact and efficient for evolution [15]. GNP is initialized by choosing the nodes, then randomly selecting connection genes for each program in the population.

On top of GNP, Mabu et al. propose a variation which uses RL, called RLGNP, to introduce online learning to derive a policy which acts as a path of execution with a program [15]. This is actually an extension of another online GNP algorithm they created, which was inefficient due to the large size of its Q-table. RLGNP programs create their own Q-tables, as each of a program's nodes is considered a state, and actions involve choosing which function on a node to execute. Each node contains multiple functions which are executed according to the agent's policy.

This variant of GNP thus evolves skeletons of programs, which RL optimizes by finding a path of execution to solve the given task [15]. An advantage to this is that evolution carries the weight of finding a diverse population of candidate solutions for RL to optimize; GP and RL work together to compensate for inefficiencies they otherwise have.

The GP in this algorithm establishes graph structures representing a subset of functions (actions) out of a broader selection of functions. In some ways, this is just another use of GP decomposing the state-action space in order to reduce the size of the Q-table for RL. RL is then used to find a policy which guides the execution of the program to solve a given task.

Overall, there are some important observations to be taken from Mabu et al. that contribute to the current work:

1. GP excels in broad, parallel searches that have the ability to find diverse populations of programs.

2. RL excels in intensified searches for optimizing local systems.

3. Storing information relating to Q-values and policy on a decomposed representation of the environment saves a substantial amount of space compared to the tabular Q-table; especially while dealing with large state spaces.

### 2.4.7 Summary

This subsection is meant to provide a summary of the findings relating to hybrid algorithms. The summary will be presented as a series of key points to be analyzed in the following section.

The first key point is that GP excels at searching for diverse populations of candidate solutions in a large environment. The compact representation of programs allows for efficient evolution amongst many programs. Moreover, GP's credit assignment may only commence after a termination condition is encountered (or episodic updating). RL, on the other hand, excels at optimizing local solutions through credit assignment over local rewards. These two components complement each other, as GP can decompose the state space to make RL more efficient, while RL can optimize candidate champion solutions constructed during evolution through online learning.

The second key point observed from the review is that saving policy information, such as Q-values, on decomposed representations of state spaces, reduces the required physical space for a Q-table. Traditional Q-tables must save information for every state-action pair, which is often redundant (when the state space is abstracted) and can lead to scaling issues on larger tasks. A proper decomposed representation of a task is able to maintain all key features while reducing the number of actual states to explore; compensating for the curse of dimensionality RL suffers from.

Lastly, hybrid algorithms have the capability of using planning mechanisms similar to DynaQ [24]. This involves the GP part of the algorithm using evolution to search for an abstracted representation of the environment, to be used as a model for RL. RL, in these cases, then only has to find solutions from the champions of evolution; thus bringing RL outside of the evolution loop. For this type of implementation, the RL agent learns within a simulation of the real environment (the searched model) rather than directly from the environment itself. As shown in Elfwing et al's work, this also eliminates the need for any sort of human designer of the model, for model-based learning [10].

## 2.5 Implications to Current Work

For context going into the remainder of this chapter, the current project's general structure can be found in Figure 1.1 in the previous chapter. To review, a team is comprised of a varying number of modules. A module holds a program, a corresponding action-set, and a Q-table to pair them during RL. The program itself is comprised of a 1-dimensional region, as well as an associated fitness score. The region is abstracted directly from the environment during evolution (region construction) in GP. A champion team should compile these modules into a complete abstraction of the world, where a subset of modules creates a pathway (of regions) from the start of the world to the goal. As mentioned, it is up to the RL agent to derive a policy that maps that subset.

This section will review the key points introduced regarding GP, RL, and hybrid algorithms. After that, the section will discuss the current project with respect to the presented background.

GP is a family of algorithms within the realm of EC which involves a population of programs evolving toward a solution. GP is particularly advantageous as it allows for variable-length representations of solutions. These algorithms are offline, meaning the quality of a candidate solution will be assessed after performing over a whole generation, usually with an ability-judging function known as fitness. GP algorithms will typically initialize a randomly seeded population of programs and manage their evolution toward a solution over many generations. GP solutions excel at broad searches of large spaces in order to achieve a diverse population of solutions.

RL algorithms typically involve a single agent learning a policy in environments that can be described with MDPs. A policy is a state-action mapping over the whole environment which acts as the main decision-maker for the agent. The policy itself is usually derived using some sort of approximation of the actual value of the current state of the agent. The environments themselves will return some local reward signal and new state with each input action to drive learning for the agent. A specific example of RL, Q-learning, derives policy to directly approximate the optimal action-value function, independent of policy; all while using TD updates to support online learning [24]. To track the Q-value for each state-action pair, a typical Q-learning solution may store these values in Q-tables. The disadvantages to RL are that it

does not typically train multiple agents within one training session, and a lot of resources may have to be allocated toward exploring the given environment. Q-learning itself struggles with space efficiency while dealing with larger Q-tables within larger environments. RL, and Q-learning, do however thrive in intensified, local searches in order to optimize some sort of action-selection process.

In this project, the scope of hybrid algorithms is algorithms which combined RL and GP. The hybrid algorithms presented typically involve optimizing some sort of action selection process within an evolved, or evolving, program. This project has a particular focus on hybrid algorithms that search, or plan, using GP, then use RL to find an efficient solution from the evolution's champions (outside of evolution). The current project uses the strategy present in various hybrid studies which involve efficiently searching for an array of candidate programs through evolution, to optimize a path of execution within the champions. This method highlights the advantages of both GP and RL, while also allowing each algorithm to rely on the other to compensate in the areas where they lack capacity.

It is usually observed that the evolutionary search process from GP creates programs which act as a decomposed representation of the state space. This concept will be discussed again, but it is important to also note the advantages this yields for Q-learning. Q-learning is no longer cursed by the dimensionality of a larger state space. The algorithm's modules can now act as the source of action-value information, which alleviates the need for an ever-growing Q-table. Thus, the modules themselves are considered to be developing these smaller Q-tables, as they will contain a subset of the original state space, with a reduced amount of actions available for selection. This advantage is also highlighted by Mabu [15]. Q-updates are also more efficient as they are directly on the program, rather than requiring a potentially expensive number of Q-table look-ups.

Overall, the current literature on hybrid algorithms provides a look at the advantages of a pipeline which executes GP and RL. The current project aims to utilize these concepts to expand on these ideas, as well as fill in some gaps that are present within this area. The concepts the current project will focus on will be discussed for the remainder of the section, first by unpacking the project's hypotheses. The motive behind the current project will be demonstrated in the two hypotheses presented in

the following sections.

### 2.5.1 First Hypothesis: Abstracted Preservation of Local Environmental Information for Policy

To act as a candidate solution during evolution, a team must provide some sort of representation of the task in order to give a meaningful solution. For a team to provide this meaning, it must collect and save information about the environment with respect to the current solution. A program can save this information within the internal parameters of its region. In the current project, regions will be parameterized from the local reward signal within the environment, to be directly contextualized by the environment. During evolution, regions will be constructed as 1-dimensional region sets to act as abstracted states, specifically for the RL agent. These regions will accumulate local reward to contribute to fitness, in order to preserve the environmental context that the local reward provides the task. In most of the reviewed work relating to EC and GP, programs are initialized with random parameters (random solutions), thus no context of the environment itself. It is hypothesized that building out candidate solutions based on the context of the current environment (local reward) will allow an RL agent to learn policy using champion solutions as models of the environment. This context is important to be maintained in solutions as the RL agent will use them to learn. This environmental context can thus be moved through the algorithm via local reward to further couple programs and RL.

The programs in this project use inequality operators to create regions in order to further decompose the state space. This representation will break the 2-dimensional state space into a set of 1-dimensional regions, where RL will be used to select one action for each region, treating each region as a state. Downing's GP [7] follows a similar structure, as shown in Figure 2.2, where the state space is broken into regions. However, there are some differences between this algorithm and Downing's work:

1. The search spaces in the current project will not use subgoals to further push programs or agents toward making meaningful regions and policies.

2. The regions formed by programs will strictly be one dimension lower than the task. That is to say, the modules correspond to 1-dimensional regions of a 2-dimensional grid world in which pairs of actions can be applied to traverse the region.

With stricter regions and a lack of subgoals, the current project must use other techniques to pass environmental context from the environment onto the agent.

### 2.5.2 Second Hypothesis: Reduction of Direct Environmental Interaction

It is hypothesized that the champion teams will provide abstracted representations sufficient enough to preserve environmental information for a learning RL agent. As mentioned, the local reward signals are passed through as the reward held on the program (summed to team fitness), allowing the environmental feedback to remain accurate even with a model. This use of model-based RL should also reduce the direct environment interactions the hybrid algorithm makes during learning because there are no longer any direct RL interactions with the environment. The metric of direct environmental interaction was not one often discussed in the reviewed research. This lack of mention is despite reduced interactions being a potential strength of these types of algorithms, as evolutionary algorithms are able to abstract state space for RL. The closest use of this hybrid model-based learning technique was applied while evolving solutions to save on real robot interactions [12]. As the tasks attempted in this project do not deal with real-life noise, the agent can base learning entirely on the evolved model. Although noise is not used in the current project's tasks, the evolved solutions come directly from the environment; any context, such as noise, that could be lost using another simulation should not apply.

As mentioned for the current project, programs are not randomly initialized but subject to parameterization through local rewards in an attempt to achieve a closer relationship between the module's region and the local reward signal. This is so RL will be able to learn directly from the team as a model. Instead, the search process during evolution (detailed in Chapter 3) builds modules from the ground up; they start at the starting state of the task and iteratively grow from there. This further couples the built program's region and the reward signals in the state space, as the proposed GP algorithm builds programs by directly slicing up the state space, and assigning fitness according to the actual reward signal. As mentioned, this state space context that is carried through allows RL to learn simply using the teams of modules as a model of the original environment. It is hypothesized that this will significantly

reduce the amount of direct environmental queries required while learning, as:

1. The RL agent will never directly touch the environment.

2. The GP search efficiently covers a lot of exploration so the agent can focus on the intensified search of optimizing the champion solutions.

Overall, this project aims to build programs (thus teams) from the ground up via iterative region construction (evolution) in order to keep a close relationship between the regions and the local reward signal from the state space. This relationship is important as the champion teams will act as models for the RL agent to learn a policy appropriate for the real state space. Overall, it is hypothesized that an RL agent should learn a policy from teams that represent the environment's state space. This type of team-based (model-based) learning is also expected to reduce the amount of direct environmental queries needed for the agent to derive a policy, compared to RL algorithms such as classic Q-learning and DynaQ [24].

# Chapter 3

# Proposed Technique

As discussed in the previous chapter, the current project has two main hypotheses. The first hypothesis is that representation in GP will preserve the environmental context of a task if local reward is used directly for credit assignment. The second is that there will be a reduction in direct environment queries when RL is trained using GP's abstraction as a model. To test these hypotheses, an algorithm was developed, called QTRB, that first evolves GP to construct a set of programs parameterized via local reward, known as teams, which are passed to a Q-learning agent to be used as a model to learn a policy for the given task.

As shown in previous chapters, Figure 3.1 depicts a high-level representation of a sample solution based on QTRB's implementation. To reiterate, a candidate solution, or team, is comprised of a varying number of modules. The team is built iteratively through evolution; a champion solution is a team that is able to solve the given task with a relatively high fitness score. The module itself is made up of a program and an action-set. The module also holds information relating to the pairing of the program and action-set during RL. Lastly, the program is comprised of a 1-dimensional region and a fitness score, which are both constructed during GP.

The following sections will discuss the finer details of this structure, as well as describe exactly how GP and RL are applied to it in the current project.

## 3.1 QTRB Overview

This section will provide a detailed overview of QTRB to investigate the current project's hypotheses. The terminology presented throughout will be relative to the scope of the model, depicted in Figure 3.1.

During GP, a population of teams evolved according to specified stop criteria, constructing modules throughout. The component of the module that was most relevant to GP was, of course, the program. As mentioned, each program had a

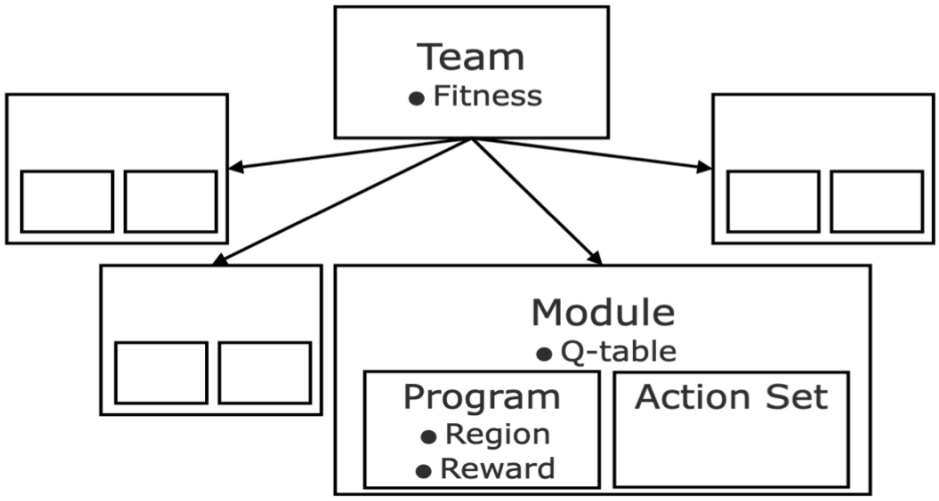


Figure 3.1: As shown in the Introduction chapter, a high-level depiction of the model developed in the current project. A program holds information about the environment in the region, along with a reward. A module holds a program and a corresponding action-set, along with some information for RL. The team holds a variable number of modules as well as a fitness score, which acts as a candidate solution in evolution. A champion team will contain modules with regions that form a path from the start of the grid world to the goal.

region, thus after evolution, a team would have a set of programs (regions) which covered the environment from the starting state to the goal. The fitness of each region was parameterized by local reward directly from the environment. A fully developed set of modules, or champion team, could thus be used as a model to deploy an RL agent. The fitness of each module was used as reinforcement signals for the Q-learning agent during this model-based learning. Using these reinforcements, the Q-learning agent assigned each module's action a Q-value in relation to its region. This ultimately associated action value with specific regions (within each module), to derive policy. This policy allowed the agent to successfully navigate the task, using a subset of modules from a champion team. Algorithm 5 provides an overview of QTRB's learning pipeline.

**Algorithm 5** QTRB Overview. The team will construct a set of modules, each program having a fitness based on environmental (local) reward signals. The RL agent will then use that team as a model to learn the environment. Given a champion team, the agent uses module fitness as reinforcement and assigns Q-values to the module's action-set to derive a policy. Local reward is passed through each step of the algorithm, driving both evolution and learning.

1: Initialize reinforcement signals onto the local environment
2: GP module-building, fitness derived from local reward
3: RL Q-Learning, Q-values derived from module fitness

The result of QTRB, shown in Algorithm 5, involves GP developing a team to act as a model of the environment for the RL agent to learn from. The respective fitness of each module was used as reinforcement signals for the agent, as they were directly derived from the environment's local reward signals. This team-based learning approach reduces the number of Q-values required for the agent, as a single Q-value was assigned to couple an action to a module's whole region, rather than to each individual state of the task. The agent can confidently couple an action to a region because regions are 1-dimensional and should move the agent along in a single direction; it would be redundant to assign the same action to each individual state within a region. As shown in Chapter 4, this team-based learning also allowed the agent to learn from a model of the environment rather than learning from directly querying the environment itself.

The next sections of this chapter will provide an overview of each component illustrated in Algorithm 5.

## 3.2 Reinforcement Signals

Although only requiring a brief discussion, reinforcement signals are extremely important for deriving policy using parameterized programs. Reinforcement signals from the environment, or local reward, will be passed from start to finish of QTRB. When properly configured, these signals drive both evolution and learning throughout this algorithm.

## 3.3 GP & Team Building

In the scope of QTRB, the goal of GP is to develop an abstract representation of the original environment for an RL agent to learn from. GP is the only learning mechanism of QTRB which directly interacts with the environment, so this representation must capture enough environmental context for the RL agent to accurately learn. Each champion team from GP consists of a constructed set of modules that can act as such a model. This set of modules should contain a subset of modules that can be used to make a path from the start state of the environment to the goal state. Thus, GP's goal is to iteratively expand its module-set so it is capable of solving the given task set, all by constructing a single new module for every team in the current population at each generation. Team building is still stochastic because regions can be split, while still constructed through interaction with the environment. Since GP is directly deployed onto the given environment, its fitness can be derived using the reward at each state of the local environment itself. Algorithm 6 shows the process in which a population of teams evolves to develop these module-sets.

The following subsections discuss various aspects of Algorithm 6.

### 3.3.1 Modules

During GP, a team's fitness was assessed based on the sum of the fitness of the programs in their module-set. These components of GP will be discussed, but it is

**Algorithm 6** Module Building GP Overview

```
1: Initialize and seed population
2: for gen in gens do
3:    for team in population do
4:       if team is not champion then
5:          child = copy(team)
6:          child.search()
7:          if win then
8:             child = champion
9:          end if
10:         population.compete(child)
11:      end if
12:   end for
13: end for
```

important to first introduce the module itself, as it is such a core part of this project's GP.

As mentioned, a single module consists of a set of directions, or actions, and a program. The program consists of a region and a reward (as received upon parameterizing the region). The module also contains a table to pair Q-values between a program's region and the module's action-set. A team developing the module, or agent learning within the module, will only be able to move in any of the directions given in the action-set. In this project, an action-set can either be North-South or East-West.

The region within a program defines the range of movement an agent in the module can travel to still be considered "within" the given module. For example, shown in Figure 3.2, a given module has a direction set containing values for North and South, and (4, 2) and (4, 4) of a two-dimensional grid world as its region's lower and upper bound, respectively. Within the grid world, an agent in this module would be able to move to states (4, 2), (4, 3), and (4, 4), in the North-South direction. Every other state of the grid world is considered to be outside of this particular module's region.

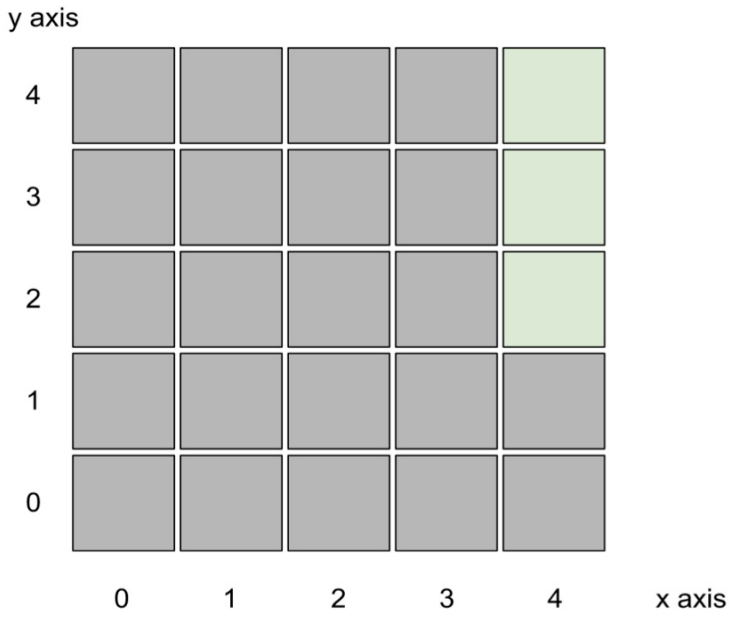


Figure 3.2: Shown is an example of a module in a $5 \times 5$ grid world. This module's region is highlighted in green and has an action-set of North-South. Given the selected module, an agent would only be able to move within the highlighted cells to be considered "within this module's region".

#### Establishing Action-sets

The logic behind the strict N-S or E-W action-sets comes from the program's region always being a projection of a lower-dimensional region of the environment. As this project only features 2-dimensional grid worlds, a region will only ever be 1-dimensional. Within these 1-dimensional regions, action-sets will only ever take on one of two directions, North-South or East-West.

### 3.3.2 Seeding population

During GP, a child is created as a copy of its parent, the selected team. For this cycle of evolution to start, the population must be seeded with starting teams.

During this project, a team is seeded in the population with an empty region. The action-set of this seeded team is important though, as it will determine the direction in which the child can move.

As mentioned, the teams in this project are limited to two options of action-sets, North-South or East-West. A child will always sample the opposite direction set of its most recently constructed module, thus the most recent module its parent constructed.

### 3.3.3 Search

In this project's scope, the search function refers to a team searching for (constructing) a new module. This involves the given individual building and parameterizing a region directly from the environment. Thus every time a team executes the search function, a new module is added to that team's individual module-set.

To add to the set of modules, the given individual must construct the action-set and region. As mentioned, the action-set is found by sampling the opposite action-set from the module created in the team's most recent search.

The region itself is developed through several environment queries. First, the team samples a starting state for its search based on the most recently constructed module. On an implementation level, this may require splitting up the most recent module into two modules, to compensate for the new region's insertion. The new module will then take steps on the environment according to the first action in the action-set. This will repeat until the environment yields some negative reward signal. This negative feedback indicates either the upper or lower bound for the module's movement with the chosen action in that particular part of the environment. The module will then move using the other action in their action-set (always in the opposite direction of the previous movement) until the environment yields another negative signal. Once again, this marks either a lower or upper bound for the module's developing region. By testing the modules's limits this way, the upper and lower bounds of movement within the region on the environment are established given the directions of the module's action-set. Throughout a single search, the fitness of a given module is the sum of all local reward accumulated from all states within that module's region. This algorithm is shown in Algorithm 7. Visual examples of this development process are shown in Figures 3.3 and 3.4.

As mentioned, the nature of GP makes the overall module-building process an

**Algorithm 7** Search function in QTRB. Steps 1-5 initialize a starting point for the module's region, while the remainder is spent searching for the bounds of the region. Note here that a state being "greater" than another state means it is further North or East, depending on the direction of the module. Similarly, a state being "lesser" than another state means it is further South or West.

1: env.curr = random(mostRecentlyMadeRegion)
2: **if** env.curr not mostRecentlyMadeRegion.lowerBound or mostRecentlyMadeRegion.upperBound **then**
3: mostRecentFirstHalf = Program(min = mostRecentlyMadeRegion.lowerBound, max = env.curr-1, fitness = mostRecent.fitness)
4: mostRecentSecondHalf = Program(min = env.curr+1, max = mostRecent.max, fitness = mostRecent.fitness)
5: **end if**
6: min, max = env.curr
7: flip = 0
8: **while** $flip < 2$ **do**
9: $(s_{t+1})$, $(r_{t+1})$ = env.step(actionSet[flip])
10: **if** $state > max$ **then**
11: max = state
12: **end if**
13: **if** $state < min$ **then**
14: min = state
15: **end if**
16: **if** $r_{t+1} < 0$ **then**
17: flip ++
18: **else**
19: env.curr = state
20: **end if**
21: $R$ += $r_{t+1}$
22: **end while**
23: team.module.program.region.max = max
24: team.module.program.region.min = min
25: team.module.program.$r_{region}$ = $R$

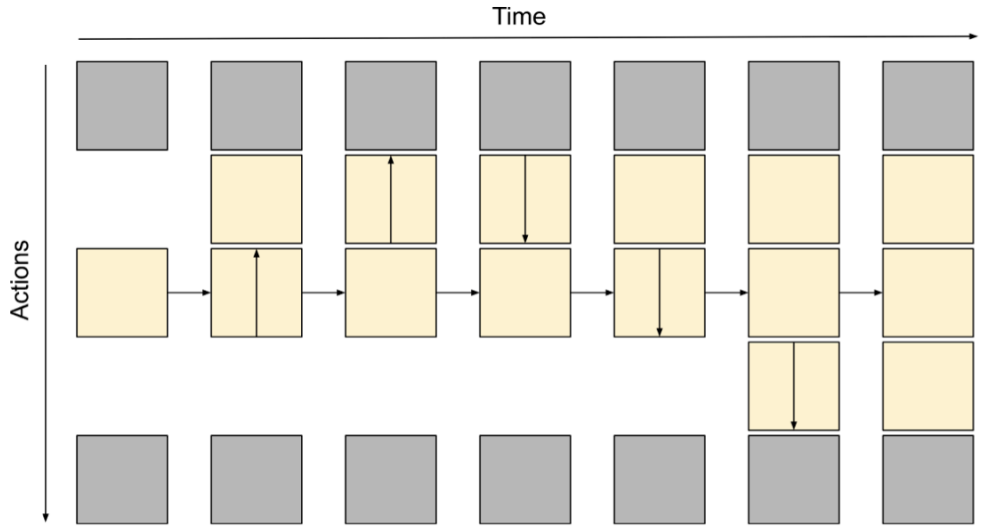


Figure 3.3: An example of searching with a North-South module. The grey squares represent cells that yield some negative local reward signal, while the yellow squares represent the module's developing region; each column is a new action execution. As shown, the module will sample a start state, then execute its first action until a negative local reward signal is received. The module will then switch actions and continue executing until a negative reward signal is received again. The result is the module's region.

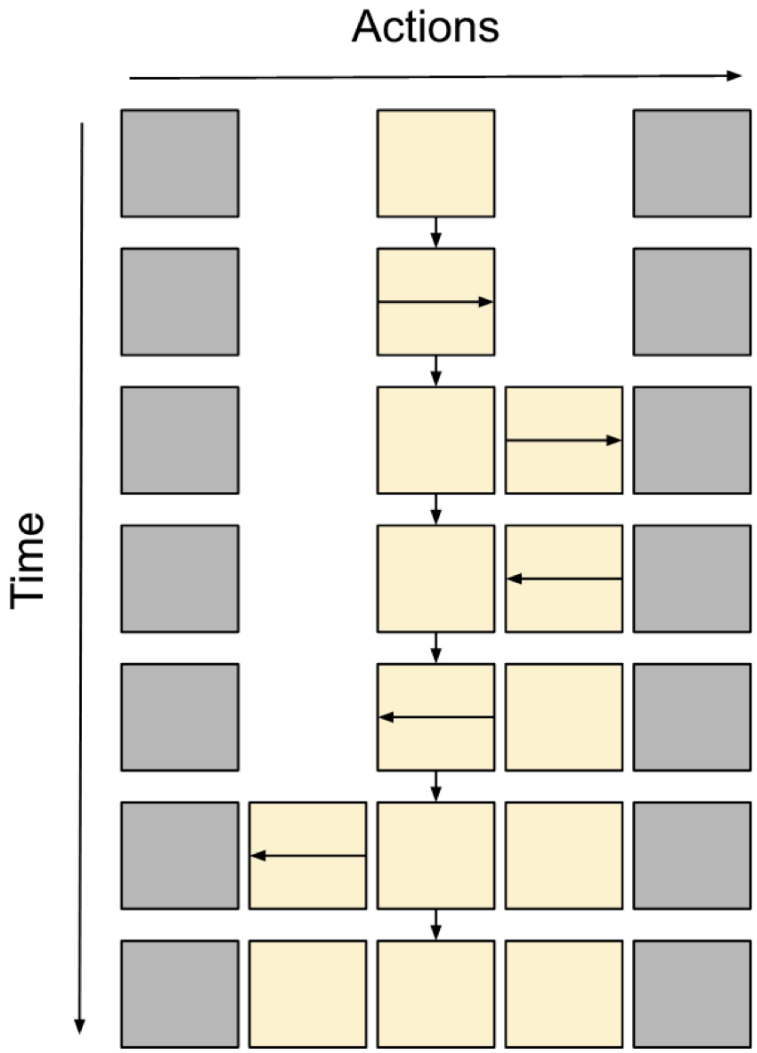


Figure 3.4: An example of searching with an East-West module. The grey squares represent cells which yield some negative local reward signal, while the yellow squares represent the module's developing region; each row is a new action execution. As shown, the module will sample a start state, then execute its first action until a negative local reward signal is received. The module will then switch actions and continue executing until a negative reward signal is received. The result is the module's region.

iterative solution. This allows a snapshot of the parent to be preserved in the population, in case the child's module is not progressive in solving the overall task. Due to this, the search each child partakes in acts as a sort of source of genetic variation to the parent and their module-set. If this variation is valuable to solving the solution, the child will survive in the population. If not, the child may be left behind while the previous snapshot of it, the parent, could very well survive to the next generation; having another chance at reproduction. Overall this allows children to experiment with many different new module compositions while posing no damage to the original structure of the parent.

### 3.3.4 Competition

After developing the new module, the child will compete against all other teams in the population. This competition is based purely on the overall fitness of the given team. As mentioned, the fitness of a team is the sum of the fitness of the programs in its set of modules.

The team with the lowest fitness is removed from the population, and the rest move on to the next round of sampling and searching until all teams from the current generation's starting population have reproduced. The process repeats according to the given stop criterion of GP.

## 3.4 RL Part

Throughout the GP part of the algorithm, any team that solves the task is saved as a champion. Once the GP stop criterion is satisfied, an RL agent will be deployed upon the champion's module-set to learn a policy to solve the task. Thus, the champion's module-set is used as a model for the RL agent to learn. The agent makes use of Q-values to determine the value for each of a module's actions associated with its region. These Q-values are assessed at a region level, rather than for each cell covered in the model, thus they can be stored on the module itself. The two main components of RL in this project were the agent's interactions with each module, and the way Q-values were stored, both will be discussed in the following subsections.

### 3.4.1 Module Interaction

During RL, a GP champion's module-set acts as a model of the environment. Thus through this model-based learning, the agent will never actually directly interact with the environment, but rather with a champion's abstracted representation of the environment.

By deploying the agent onto the module-set, it can derive the value of region-action pairs. As mentioned, the winning action is assigned to the whole region, rather than each state within it. This is logically sound as cells in a 1-dimensional region will always share the same direction, thus action, towards the goal. Thus, Q-learning finds not so much the action of a region but the optimal direction for it. The overall goal of this is to configure a subset of modules from the champion to "stitch" together a path of regions (that move in the right direction) that go from the starting state of the environment to the goal; this is the agent's policy.

The agent selects actions within each region with $\epsilon - greedy$ action selection, with the agent's available actions being only those in the given module's action-set. $\epsilon - greedy$ action selection involved also choosing the action from the action set with the higher q-value, unless an $\epsilon$ variable was sampled under a given threshold, in which case the other action was sampled. It should also be noted that $\epsilon - greedy$ action selection only occurred during training, otherwise, the agent just chose the action with the highest q-value. This use of $\epsilon - greedy$ action selection will drive the agent towards the higher value direction most of the time, but also leave room for some exploration of the champion's module-set.

The agent can transition between modules, to form the path from start to goal. Module transitions are a core component of the RL in QTRB; it is the driving mechanic of "stitching" two modules together as it is when Q-value updates occur. The policy will ultimately be a collection of module transitions that form a path from start to goal. The nature of TD-learning will form a gradient of Q-values ranging from start to goal, once the large reward from the goal is collected.

Transitions occur when the agent steps outside of the selected module's region. When this occurs, the agent will find all modules with regions (of the selected team) that contain the agent's current position in the model. The new module will be

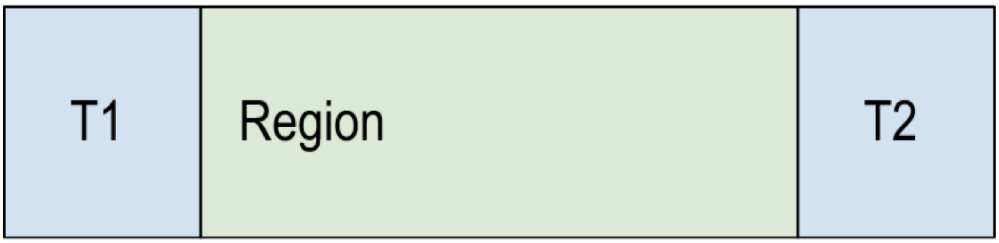


Figure 3.5: A conceptual breakdown of a region from the RL agent's perspective. T1 and T2 are the cells in which a transition occurs, while between them is the region itself.

randomly sampled from that set of modules. The agent's selected module will transition to this sampled module. As mentioned, the transition is also where the Q-value updates (stitching) occur, which will be discussed in the next subsection.

Figure 3.5 shows an example of how a module is represented during this process, transitions occur when the agent reaches the denoted transition space. This decomposes every program during RL into the region itself and two "transition" cells. These transition cells (T1 and T2 in Figure 3.5) will trigger on the first step the agent takes that is outside its currently selected program's region.

### 3.4.2 Q-Learning

Q-Learning is at the heart of this QTRB's RL, without the ability to assign value to a module's actions, a module-set would otherwise be nothing more than a random set of regions. The derived policy is simply a set of modules stitched together by Q-values. The value the module-set brings to Q-learning is that Q-values are assigned to an action associated with a whole region rather than to each cell within one. On top of this, the Q-value is stored on the module itself. This is highly efficient for module-action value look-ups, compared to a tabular approach.

It should be first noted that, because RL uses a model developed during GP, it does not need to query the environment for reinforcements. Instead, reinforcements are based on feedback from this model.

Throughout learning, the RL agent will assign the mentioned reinforcements only to the actions in the action-set of the given module. It should be clarified that this

reinforcement is simply used for inner-module action selection. As mentioned, e-greedy action selection is used, which follows Equation 3.1. Following Equation 3.1, $\beta$ is the selected module's action-set, and Q(*) are the keys associated with the selected module's action value pairs. $\epsilon$-greedy action selection is incorporated as after transitioning into a new region, the initial point within the new region might be in the middle rather than at an end. More efficient schemes for investigating these conditions are discussed in Chapter 5.

$$action = \left\{ \begin{array}{cc} argmax_{a \in \beta} Q(M_i, a) & 1 - \epsilon \\ rnd(a \in \beta) & otherwise \end{array} \right\} \tag{3.1}$$

On the module level, the Q-value that matters is the action-region value assigned to each module the agent transitions out of during RL. This value will represent the winning action for the region from its module-set. Thus, the agent will know which action to execute depending on which region it finds itself in, outside of training.

The action-region values are assigned using just the reinforcement given to the agent upon a transition. Once again, action-region value updates occur here to avoid having updates upon each state transition. The updates also support backpropagation of Q-values, so that with enough training epochs, the action-region values will make it clear not only which action to take given a module, but also which sequence of modules must be taken to solve the given task. As mentioned, TD-learning will create a gradient of Q-value from start to goal, such that following the increase in gradient will lead to the goal. This explains the logic behind the concept of "stitching" modules together and is the basis of deriving policy. The transition updates follow Equation 3.2. In Equation 3.2, Q(*) are the keys to the values associated with the values table stored on the module itself. Note that storing these values on the modules rather than a larger table significantly reduces the time to query values.

$$Q(R_t, a_t) \longleftarrow Q(R_t, a_t) + \alpha\{r_{t+1} + \gamma \max_a Q(R_{t+1}, a) - Q(R_a, a_t)\} \tag{3.2}$$

Overall, Q-learning is used to ultimately derive policy as its backpropagation mechanisms allow it to "stitch" a subset of modules together to create a directional path from the start of the task to the goal. Figure 3.6 shows the regions in Q-value space, further showing the value gradient that moves within modules from the goal

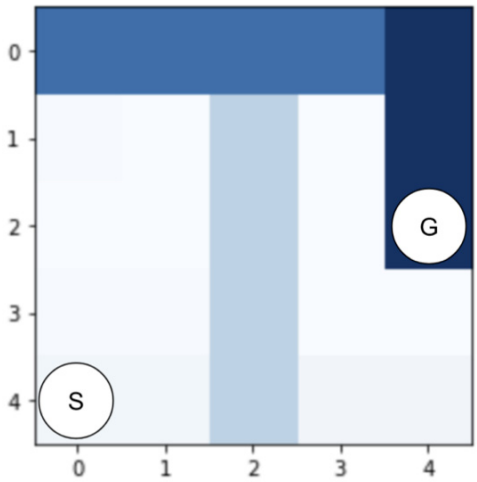


Figure 3.6: The Q-space representation from how an RL agent learned a solution from a GP champion's model on a $5 \times 5$ grid world. Concerning the figure, the environment start state is at (4, 0) and its goal is at (2, 4). The figure highlights the gradient of Q-values formed as the value is backpropagated through a stitched-together subset of modules. As shown, the modules closer to the start have a diminished return compared to the Q-values received near the goal. The trajectory of the value function is highlighted in this example.

back to the start. This also highlights the trajectory of the value function of an RL agent after learning.[1]

### 3.5 Direct Environment Queries

As previously discussed, the RL agent learns from a model of the environment. This model-based learning saves the algorithm from ever directly querying the environment during RL.

A major strength of QTRB is highlighted when total environmental queries per run are considered. All environment queries occur during module-building (GP), where a model is developed. This use of model-based RL significantly reduces the number of direct environment queries that would otherwise be necessary.

The total amount of environment queries per run follows:

[1]Note that Q-values are common to the same region, with the action of the module associated with the region defining how to transition across the region.

$$total\_direct\_queries = \#\_iterations * GP\_each\_individual\_step$$

Where #_iterations represent each time the population goes through a round of searching.

Overall, the QTRB was designed with reducing environment queries in mind. The module-building acts as a viable strategy to plan out a model in which an RL agent can freely learn from without any cost to the total amount of direct environment queries.

# Chapter 4

# Experimental Results

## 4.1 Results Overview

As discussed, the goal of this thesis is to develop an approach for incorporating local rewards into genetic programming (GP) integrated with reinforcement learning (RL); referred to as Team-based Region Builder with Q-learning, QTRB. QTRB involves constructing teams by means of local reward (from interaction with the environment during evolution), to be used as a learning model for an RL agent to derive policy. This is a divergence from previous practices in which rewards are not used interactively to adapt program representation (only used to rank individuals after an episode of interactions ends). It is important to keep in mind that the tasks are limited to 2-dimensional grid worlds with North-South and East-West action-sets.

Along with an introduction to the environments, this chapter will develop various thesis outcomes to support the project's hypothesis that such policies can be found efficiently. Specifically, this chapter includes some sample results for QTRB from a variety of tasks, the average environment queries per run for each of those tasks, and results from DynaQ for comparison.

Additionally, results are highlighted that illustrate the optimality of paths (policies) discovered, comparing different stop criteria in GP, and superimposing results over many runs of a task in an attempt to map every state of the world with action. Overall, these results act to show both the strengths and shortcomings of the QTRB.

## 4.2 Environments & Parameters

A wide array of experiments were executed to test the QTRB algorithm as described in Chapter 3. As mentioned, QTRB was designed to solve 2-dimensional grid world tasks. This project tested QTRB on a variety of such tasks, scaling in size and difficulty. In total, experimental results have been recorded for two $5 \times 5$ tasks, two

$10 \times 10$ tasks, fifteen combinations of $20 \times 20$ tasks, and five combinations of $50 \times 50$ tasks. The objective of all of these tasks is to move from the starting state to the goal, all while avoiding obstacles and staying within the environment's boundaries. This section will discuss these tasks, as well as the parameters used for QTRB itself.

Before introducing the grid worlds, and discussing the difference in properties for each size, it should be noted that all grid worlds define local reward in the same way. For each legal move, the grid worlds yield a reinforcement of 0.1, while illegal moves, such as bumping into walls, yield -0.01. Finally, the goal state of the grid worlds, hence solving the task, yielded a reinforcement of 100. This reward function is summarized in Equation 4.1.

$$r_{t+1} = \begin{cases} 100 & \text{if } s_{t+1} = \text{Goal state} \\ -0.01 & \text{if } s_{t+1} = \text{Illegal state} \\ 1.0 & \text{otherwise} \end{cases} \tag{4.1}$$

### 4.2.1 $5 \times 5$ and $10 \times 10$ Grid Worlds

Shown in Figure 4.1 is a set of grid worlds ($5 \times 5$ and $10 \times 10$ respectively) that come from the discussed Downing's work [7], and were implemented to compare his findings with QTRB. As established in Chapter 1, the main difference between Downing's work and QTRB is that QTRB does not utilize subgoals to aid solution discovery. Most importantly, QTRB interacts with the environment to discover modules that are explicitly compatible with the task. These features provide QTRB with additional abilities and potential biases that are investigated with a wider portfolio of mazes than those used by Downing.

Figure 4.2 shows a custom $5 \times 5$ and $10 \times 10$ created for the sake of experimentation during the project. The left of Figure 4.2 is a copy of Figure 4.1 (left) that is slightly modified (the obstacle at (3, 2) is removed). This was to test the module-building part of QTRB, the results of Figures 4.1 (left) and 4.2 (left) will be compared later in the chapter.

Figure 4.2 (right) is a $10 \times 10$ grid world which was implemented to challenge the module-building nature of QTRB. In this figure, a team must find the exact opening within an enclosed area to reach the goal. This figure was thought to be challenging as module-building would have to rely on sampling a region's starting state in the same

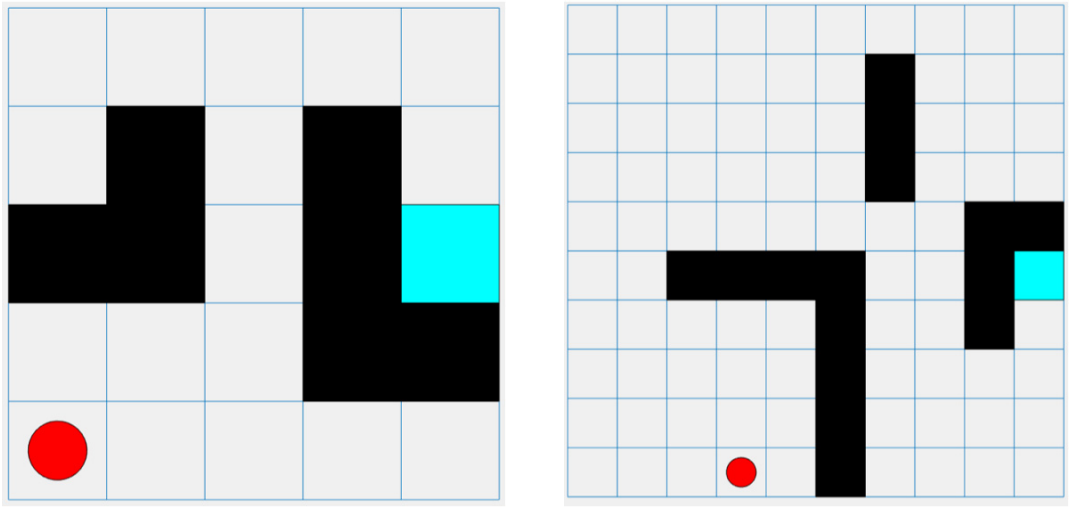

Figure 4.1: Left: $5 \times 5$ grid world task with start state (0, 0) and goal state (4, 2) (the bottom left corner is (0, 0)). Right: $10 \times 10$ grid world task with start state (3, 0) and goal state (9, 4) (the bottom left corner is (0, 0)).

1-dimensional plane as one of the openings. This task may be easier for a traditional RL agent, which would simply rely on exploration of each state to find the goal, so finding the opening would be much less based on random chance.

### 4.2.2 $20 \times 20$ and $50 \times 50$ Grid Worlds

Both the $20 \times 20$ and $50 \times 50$ grid worlds contained various aspects of random generation, overall to avoid any human bias that the creator of the task may have. The start, goal, and obstacles were randomly displaced within the available grid world space. The environments were generated with 20% obstacle coverage, 80 cells for the $20 \times 20$ and 500 cells for the $50 \times 50$ tasks. To further combat the chance of the QTRB passing on a "lucky start and goal", the start and goal state of each task was sampled five times, thus each $20 \times 20$ and $50 \times 50$ "map" generated had five different versions. The only human interaction while generating these worlds was a final approval to ensure there was indeed a possible path from start to goal.

Figure 4.3 shows the $20 \times 20$ (left) and $50 \times 50$ (right) grid worlds that QTRB was deployed within to find the results that will be shown in this section. The rest of the $20 \times 20$ and $50 \times 50$ tasks can be found in the Appendix section at the end of the thesis.

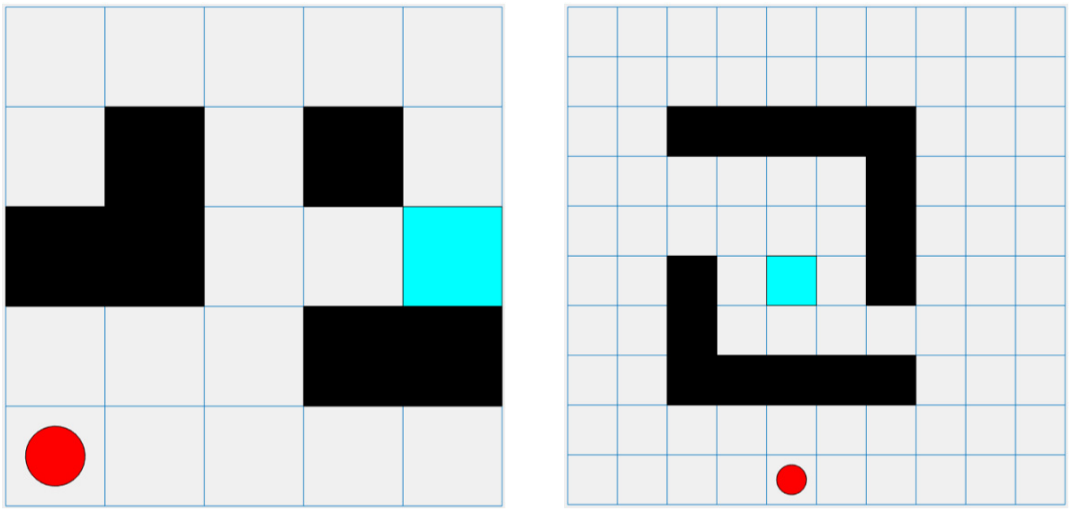

Figure 4.2: Left: Modified version of Figure 4.1 (left). It is the same everywhere except it has a "hole in the wall", the obstacle at (3, 2) was removed. Right: $10 \times 10$ grid world task with start state (4, 0) and goal state (4, 4) (the bottom left corner is (0, 0)).

### 4.2.3 GP and RL Parameters

The parameters used to collect these results, organized by task size, are shown in Table 4.1; each parameter will be briefly described for reference. The task size specifies the length of each dimension of the grid world in the given task. # Runs are the number of runs the full algorithm (GP to RL) was run on the given task, most results are presented as an average over this parameter. # Gens refers to the maximum number of generations the GP part of QTRB could run for before stopping. Max Population is the maximum number of teams that can exist in the population at any given time during a run. # of Epochs refers to the number of steps the RL agent has in each episode during learning, this thus determines the number of value function updates it can make. $\alpha$ represents the learning rate of the RL agent, from Equation 3.2. From the same equation, $\gamma$ refers to the rate at which the value of reward in future time steps are being passed back to the given Q-value is diminished. Lastly, $\epsilon$ refers to the chance of the RL agent selecting a random action, as described in Equation 3.1.

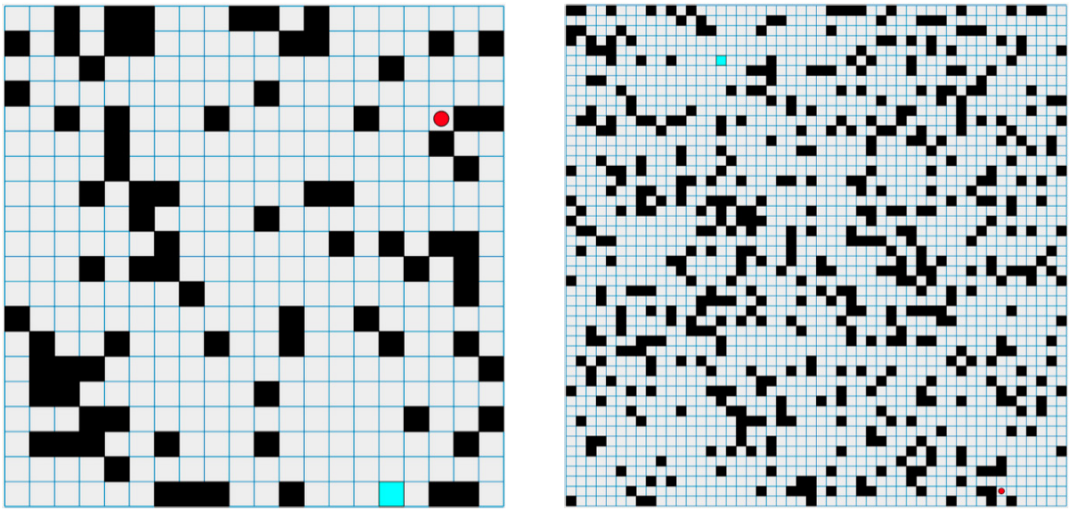

Figure 4.3: Left: $20 \times 20$ task. Right: $50 \times 50$ task. Both tasks were used to find results presented in the Results chapter, to represent their respective task size. A larger image of the $50 \times 50$ as well as additional tasks with these sizes can be found in the Appendix.

Table 4.1: Parameters used for both GP and RL parts of QTRB, grouped by environment size.

| **Task Size** | **# Runs** | **# Gens** | **Max Pop.** | **# Epochs** | $\alpha$ | $\gamma$ | $\epsilon$ |
|---|---|---|---|---|---|---|---|
| $5 \times 5$ | 30 | 50 | 5 | 5 | 0.5 | 0.9 | 0.1 |
| $10 \times 10$ | 30 | 100 | 5 | 10 | 0.5 | 0.9 | 0.1 |
| $20 \times 20$ | 30 | 150 | 5 | 20 | 0.5 | 0.9 | 0.1 |
| $50 \times 50$ | 30 | 200 | 5 | 50 | 0.5 | 0.9 | 0.1 |

## 4.3 QTRB Results

This section will provide insight into QTRB's performance from GP to RL. It will provide fitness curves, as well as sample policy and Q-value maps for each figure specifically discussed in the previous section.[1] These runs all follow the parameters above, using a generational stop criterion for GP. These policy maps will be used frequently throughout this chapter to show the results of RL on QTRB, so it should be noted here that a question mark represents states on the task QTRB did not visit during exploration of the task, and an X on the map represents an obstacle. The

[1]The fitness curves are based on the sum of reward described in Equation 4.1

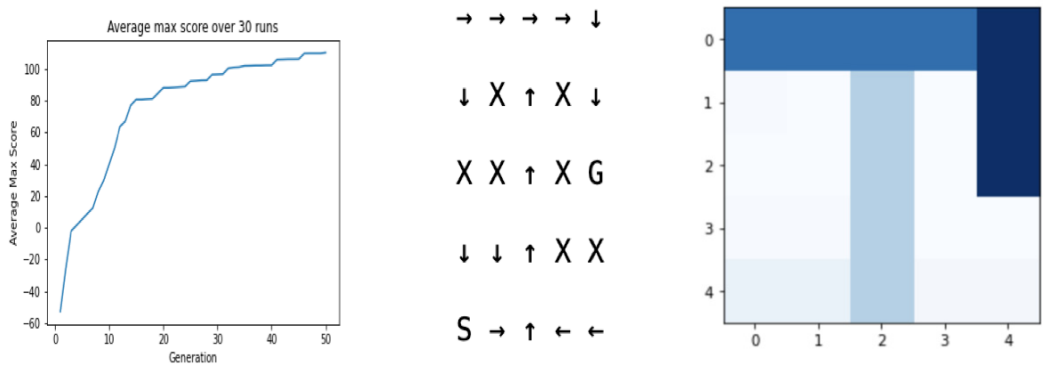


Figure 4.4: QTRB Results from Figure 4.1 (left). Shown from left to right: GP fitness curve, policy map, and Q-value map. The starting action was East.

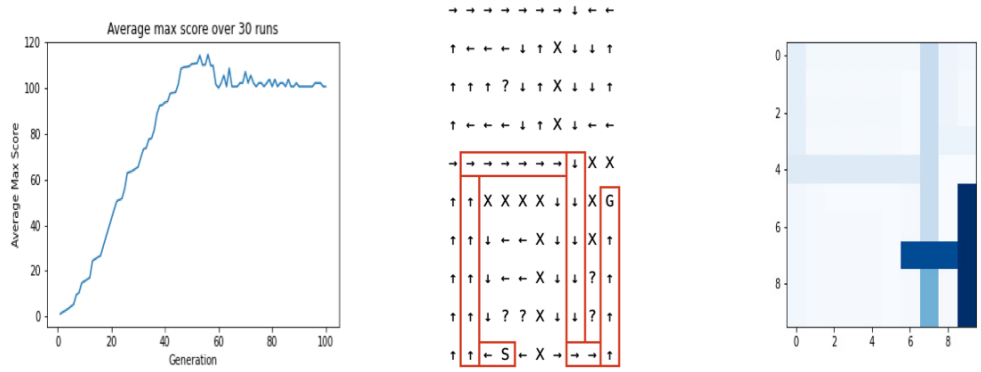


Figure 4.5: QTRB results from Figure 4.1 (right). Shown from left to right: GP fitness curve, policy map, and Q-value map. The red boxes were added to better visualize the path found by Q-learning The starting action was West.

Appendix provides these QTRB results for the tasks listed there as well. The results are shown in panels, from Figure 4.4 to Figure 4.10.

These base results alone support the hypothesis that parameterizing programs via local reward signals can be used as a model to derive task-solving policies. This is especially evident in the policy maps themselves, as they show the paths the agent has formed from the start to finish.

The fitness curves simply show the convergence of GP onto a solution, usually in fewer generations than provided. Future work may include minimizing the number of generations needed to create a viable module-set, to further reduce the number of environmental queries taken during GP. Needless to say, the number of generations used for each run was not heavily tuned throughout this thesis, so the parameter used

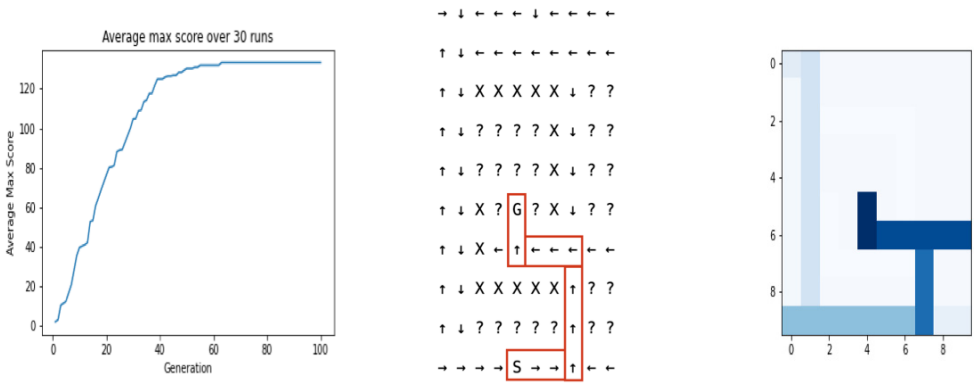


Figure 4.6: QTRB results from Figure 4.2 (right). Shown from left to right: GP fitness curve, policy map, and Q-value map. The red boxes were added to better visualize the path found by Q-learning The starting action was West.

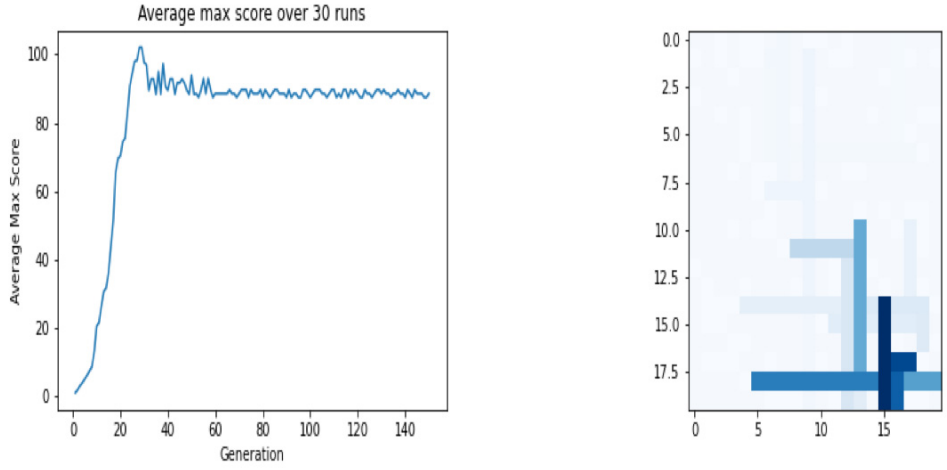


Figure 4.7: QTRB results from Figure 4.3 (left). Shown from left to right: GP fitness curve and Q-value map. The corresponding policy map can be found in Figure 4.8 for ease of visualization.

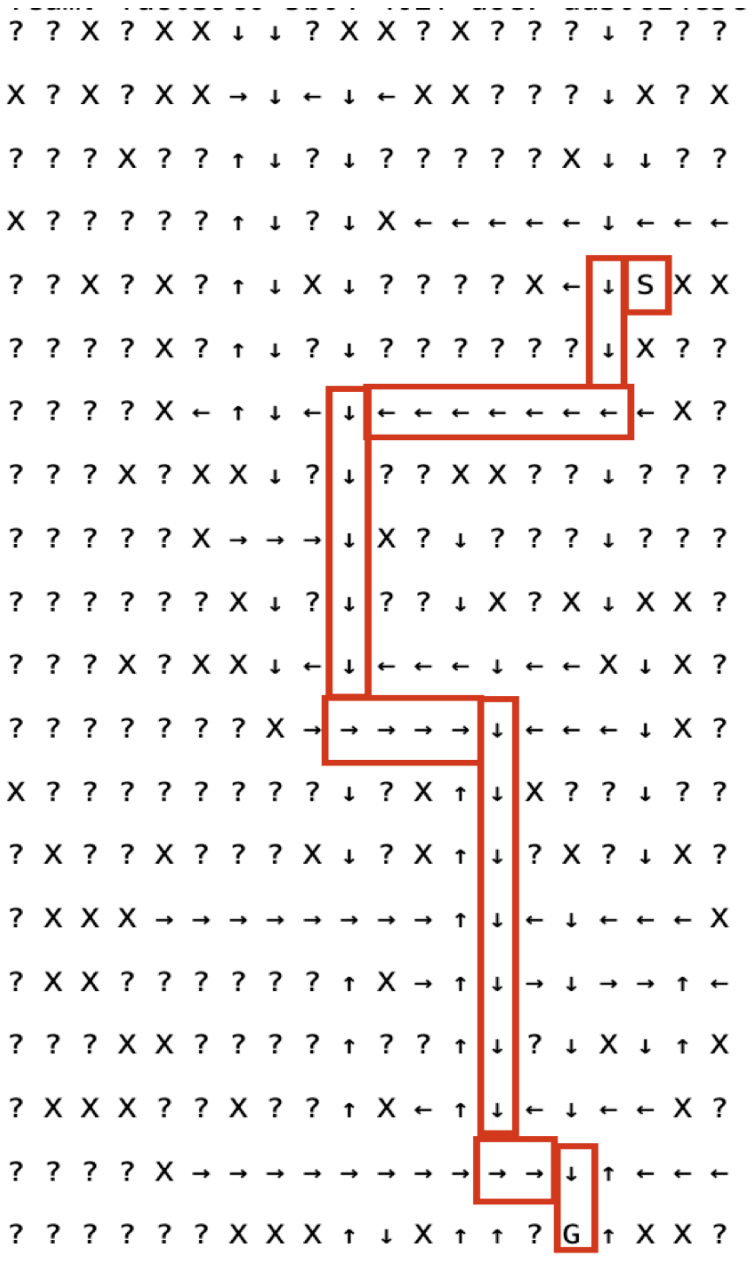


Figure 4.8: Policy map for Figure 4.3 (left). The red boxes were added to better visualize the path found by Q-learning. The starting action was West.

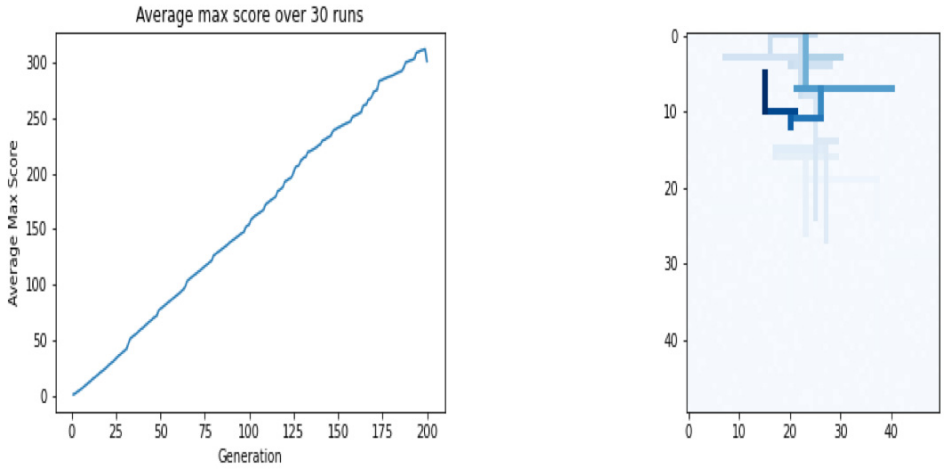


Figure 4.9: QTRB results from Figure 4.3 (right). Shown from left to right: GP fitness curve and Q-value map.

might not be optimal.

The Q-value maps themselves show the Q-value gradient being passed down from region to region of a team. The nature of the Q-learning updates will give the region which reaches the goal the most value, but that value will "trickle down" (TD learning) into the regions which were visited on the way to the goal, as shown in the Q-value maps of Figures 4.5 and 4.6. This visualization provides support to the previously discussed concept of modules being "stitched" together during RL, to form a viable path from start to goal. Additionally, different paths are clearly being considered using the Q-values associated with different module regions. Overall this provides evidence of the importance of the use of reinforcement signals throughout this project.

With reference to the policy maps in Figures 4.5 and 4.6, QTRB is exploring the grid world while also being capable of discovering the shortest paths. The approach QTRB adopts for parameterizing modules can result in large steps to different parts of the grid world. On the other hand, this does not prevent those modules from being divided up to discover different paths through the grid world.

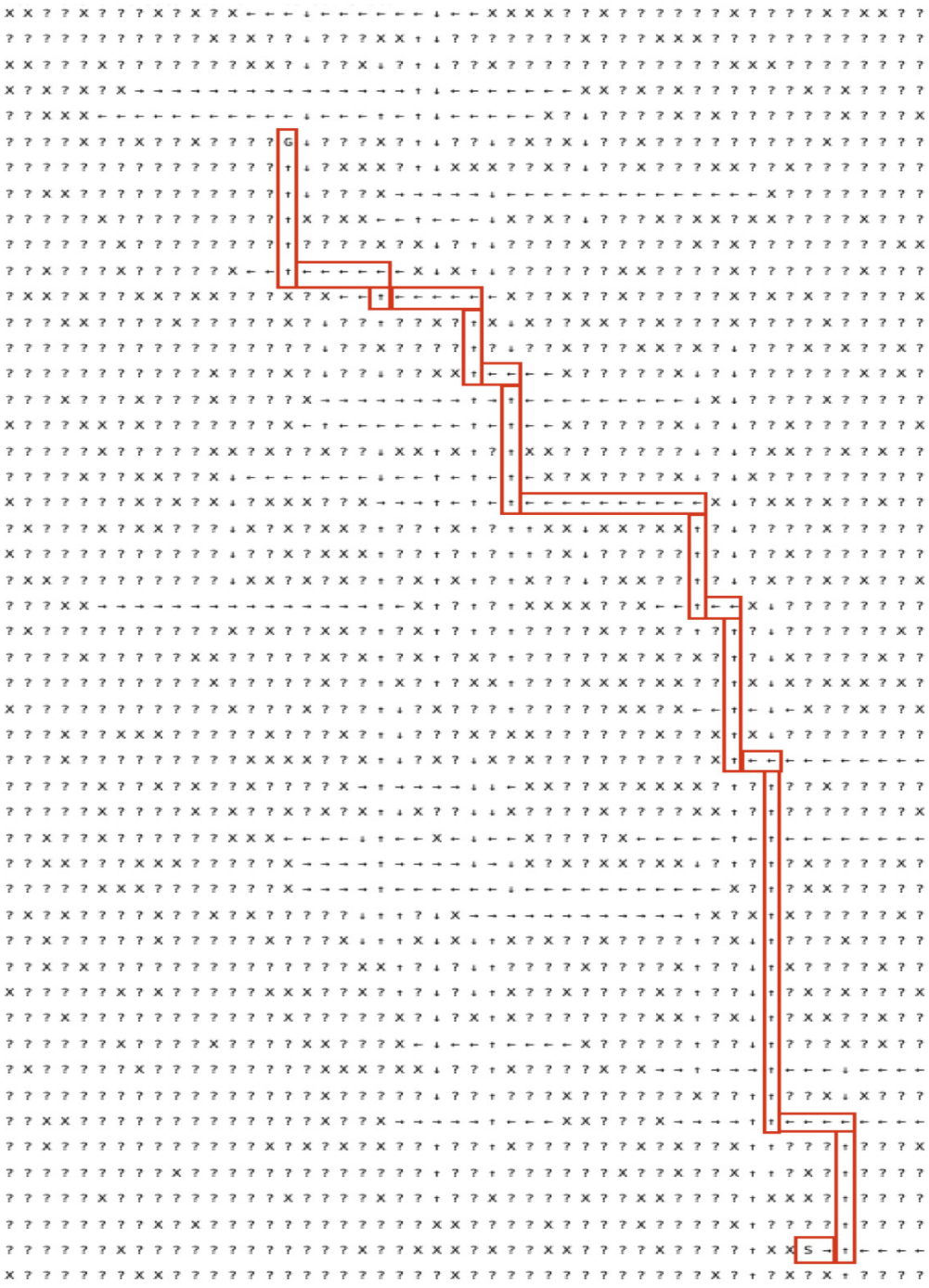

Figure 4.10: Policy map for Figure 4.3 (right). The red boxes were added to better visualize the path found by Q-learning. The starting action was East.

## 4.4 Optimal Paths

This section will provide insight into the quality of the paths the selected champions were able to find throughout QTRB's experimentation. In the scope of the current project, the optimal path simply refers to the shortest possible path from the start of the task to the goal. The optimal step count was derived from finding the optimal path from start to goal in the given task, and counting how many cells (steps) were in the given path. Table 4.2 provides the number of steps within the optimal path for each task. The number of steps taken from start to goal in a chosen sample policy for each task will be provided for comparison. These samples were chosen out of all other policies based on which had the shortest number of steps. It should be noted that these paths align with the sample policies presented for each task, in both this chapter as well as the Appendix, the tables will provide pointers to each figure for ease of reference.

The sample QTRB policy step counts presented in Table 4.2 highlight the capability of QTRB high-quality paths, but are biased towards just the best results. In order to broaden the scope of this metric, Table 4.3 presents some statistics for all policies (in a given task); depicting the quality of the average path found for each task. Though never as accurate as the step counts of the "champion of champions" from Table 4.2, the average step count for QTRB as a whole is typically not far from the theoretical optimal step count.

## 4.5 DynaQ Results

As discussed in the Background chapter, the DynaQ algorithm has a look-ahead planning mechanism that allows it to learn from a model of the environment. In this project, DynaQ was deployed on the previously mentioned tasks to set up a suite of results to compare the results of QTRB. For each task, DynaQ was deployed with planning steps of 0%, 2%, and 5% of the total cell count. It should be noted that DynaQ with 0% planning steps is just traditional tabular Q-learning; this allowed for a base RL algorithm to be analyzed as well.

Figure 4.11 shows the DynaQ on Figure 4.1 (left). Figure 4.12 shows the DynaQ on Figure 4.1 (right). Figure 4.13 shows the DynaQ on Figure 4.2 (right). Figure 4.14

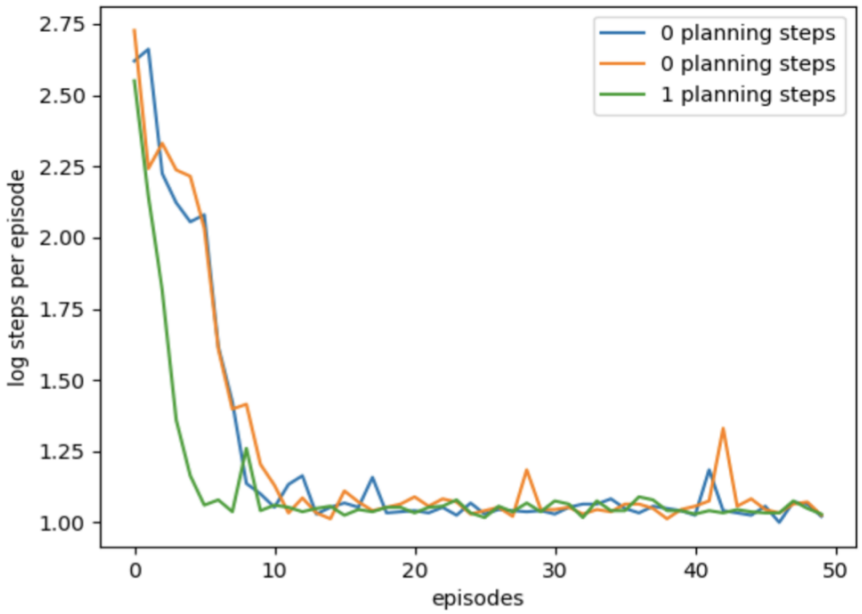


Figure 4.11: Log base 10 steps per episode curve for DynaQ deployed on Figure 4.1 (left) at varying numbers of planning steps.

shows the DynaQ on Figure 4.3 (left). Figure 4.15 shows the DynaQ on Figure 4.3 (right). The number of environment queries taken to solve each task will be compared with QTRB in the next section.

## 4.6 Average Direct Environment Queries Comparison

For the scope of this project, a direct environment query refers to any single interaction a team or agent makes with the environment during any part of QTRB. The most common example is an agent stepping through an environment during learning, to yield the new state given the step, as well as some reinforcement signal representing the value of the step.

A clear advantage of the GP-model-based RL used in this project is the reduced number of environmental queries needed to derive policy. As mentioned in previous chapters, this is because the "module-stitching" (policy building) in RL is free in terms of direct environment queries. RL can also trust that a team's model is an accurate representation of the task, as the team's modules are parameterized directly from the local reward abstracted directly from the original task.

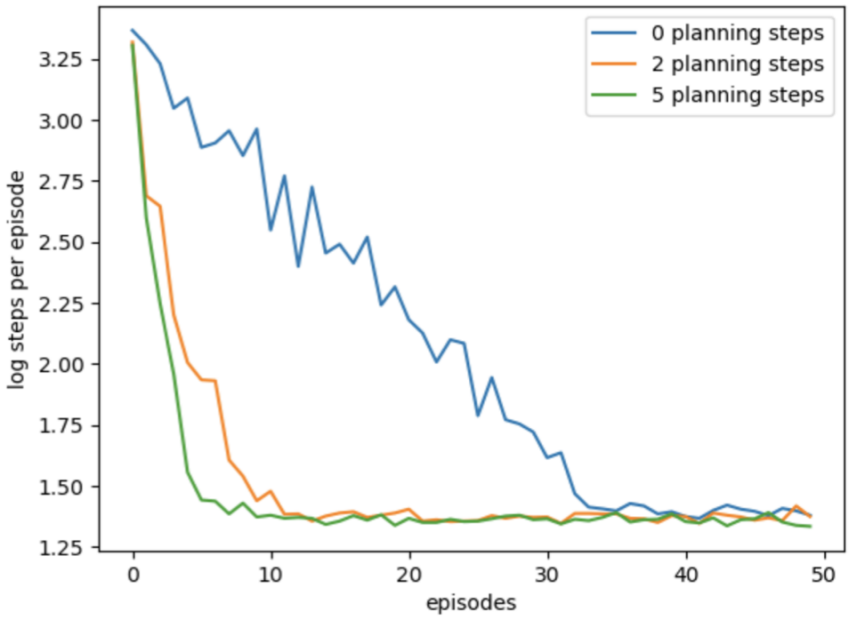


Figure 4.12: Log base 10 steps per episode curve for DynaQ deployed on Figure 4.1 (right) at varying numbers of planning steps.

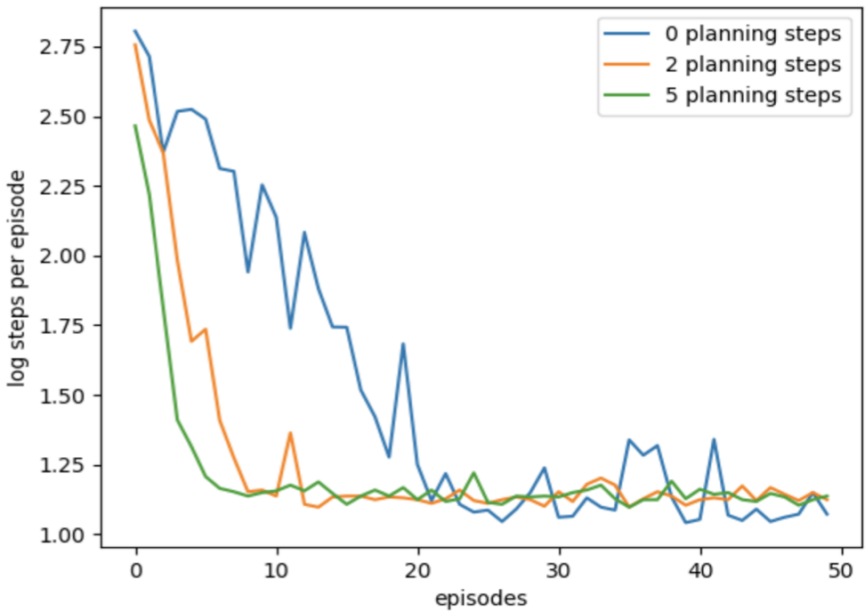


Figure 4.13: Log base 10 steps per episode curve for DynaQ deployed on Figure 4.2 (right) at varying numbers of planning steps.

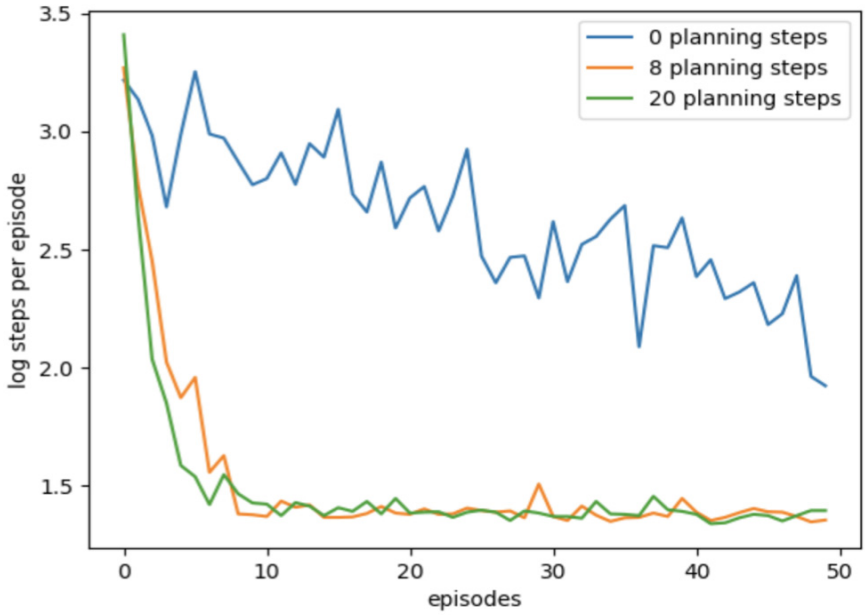


Figure 4.14: Log base 10 steps per episode curve for DynaQ deployed on Figure 4.3 (left) at varying numbers of planning steps.

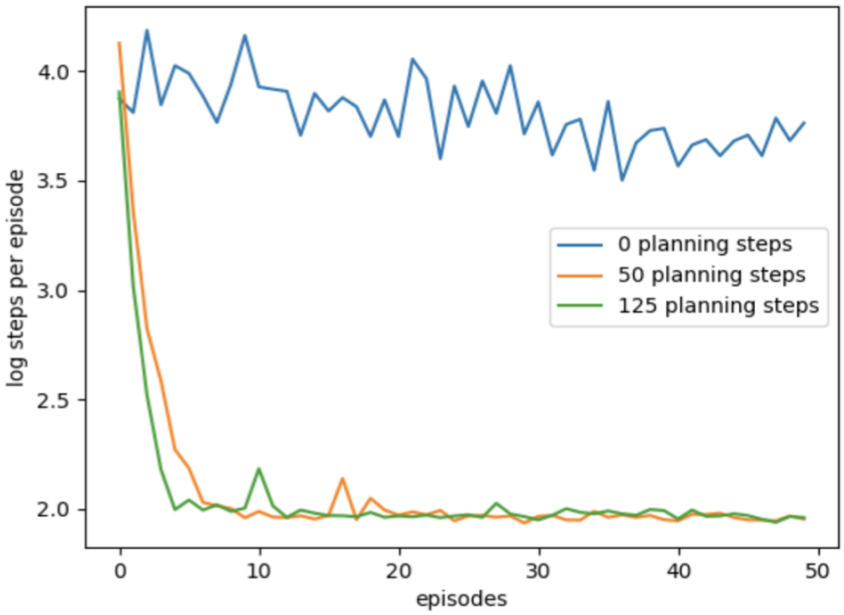


Figure 4.15: Log base 10 steps per episode curve for DynaQ deployed on Figure 4.3 (right) at varying numbers of planning steps.

The total direct environment query per run for each of the previously discussed tasks is shown in Table 4.4 for both QTRB, as well as the mentioned DynaQ algorithm at various planning steps. As shown, this algorithm uses significantly fewer environment queries, especially compared to DynaQ at 0% planning steps (traditional Q-learning).

Moreover, the tasks in Figures A.10 and A.44 are instances of $20 \times 20$ and $50 \times 50$ maze configurations, respectively. The cost of applying DynaQ under the larger mazes is clearly considerable. However, it should also be noted that introducing other heuristics into DynaQ such as Prioritized Sweeping or Trajectory Sampling would improve the sampling of DynaQ [24].

## 4.7 No Reinforcement Signals

To further show the importance of the environment's local reward signals to use as evidence to support the QTRB efficiency hypothesis, an experiment was conducted to test how having no reinforcement signals would affect policy.

For the sake of simple visualizations, the experiment was deployed on Figure 4.1 (left), as the effect on the smaller task would be easily obvious. Any accumulation of local reward by regions was set to 0. All other parameters used in this experiment were the same as those used for deploying QTRB during the base experiments.

Figure 4.16 (left) shows the resulting superimposed policies of (non-champion) teams from this experiment. Additionally, Figure 4.16 (right) shows the fitness curve for this instance of evolution. Figure 4.16 (left) is the best representation of the result for this experiment, as it clearly shows that the agent was not able to form a policy robust enough to solve the task, nor even escape the first row of the maze. On top of that, Figure 4.16 (right) further shows that the team's fitness solely relies on local reward. The importance of reinforcement signals is shown in contrast between this policy map and the policy map shown for the base results on the same task in Figure 4.4. Overall, this shows the importance of local reward signals for both GP and RL in QTRB.

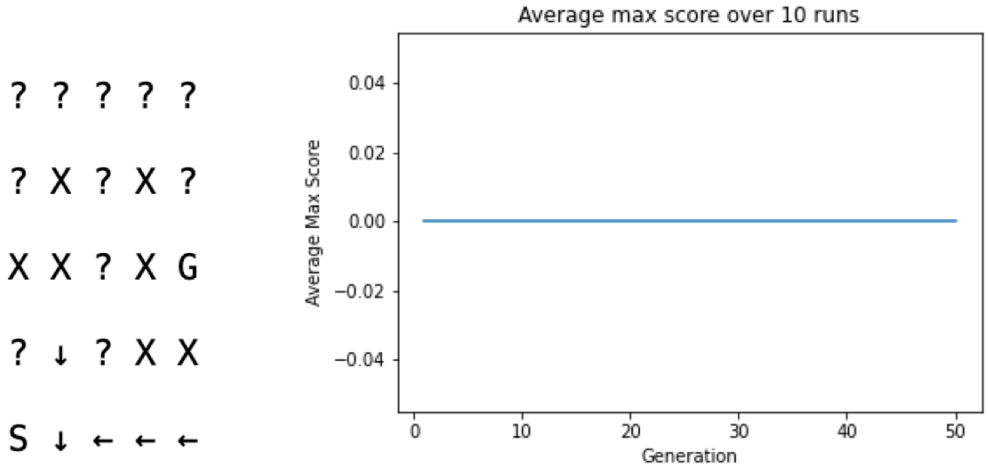


Figure 4.16: Policy map and fitness curve of QTRB deployed onto the task on the left of Figure 4.1. The task was modified so that any accumulation of reward by regions was set to 0.

### 4.8 Stop Criteria

The figures shown in the base results section were based on GP with a generation count stop criteria. This means that if QTRB has not found a solution within a certain number of generations, it will fail. With that being said, many other stop criterion can be used during GP, which may result in benefits to the quality of the policy found in Q-learning. An alternative stop criterion that was briefly investigated was the population's average region coverage. For example, GP would stop only when the average region coverage (of the given grid world) amongst teams in the population was 80%. The resulting policy for this case is shown in Figure 4.17, for QTRB deployed onto Figure 4.1 (left). Once again, this figure was chosen for sake of visual simplicity.

The benefit of a region coverage stop criterion is that it can be used to minimize the number of modules used in a champion team. This is because a team with a relatively small stop coverage will only ever reach the goal if it finds an optimal path. This however results in many teams developing module-sets that cover too much space; thus unable to ever reach the goal. This particular stop criterion can thus be used to precisely find optimal paths, with the caveat of populations as a whole having much less success, compared to the generational stop criteria. An example

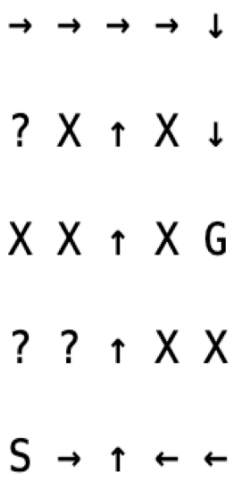


Figure 4.17: Example policy derived from average coverage stop criteria on task shown in Figure 4.1 (left). The starting action was East.

of this is shown in Figure 4.18, where a very efficient policy was found in Figure 4.1 (right). This solution had a reduced number of steps compared to many other possible solutions, using only 243 environmental queries to solve the task. The stop criterion was set to stop when teams averaged 40% coverage. Although this single team was able to find a highly efficient policy, it was the only team in the population over 30 runs to be able to solve the task. Overall this shows the strengths and weaknesses of this particular stop criterion. Furthermore, there are many possible stop functions to be examined in possible future work, all having trade-offs affecting policy.

### 4.9 Superimposed Over Many Runs

Another interesting trait QTRB possesses is the ability to construct paths to the goal from many visited states during Q-learning. This is especially interesting when the policies from champion teams over many runs are superimposed into one policy map. Two examples of this are shown in Figures 4.19 and 4.20. The two tasks shown in this figure were chosen to demonstrate this feature for two reasons:

1. visual simplicity
2. more importantly, different solutions exist within the tasks.

For both tasks, QTRB was deployed 30 times, all superimposed to one policy map. The superposition operator takes the form of the action direction corresponding to the

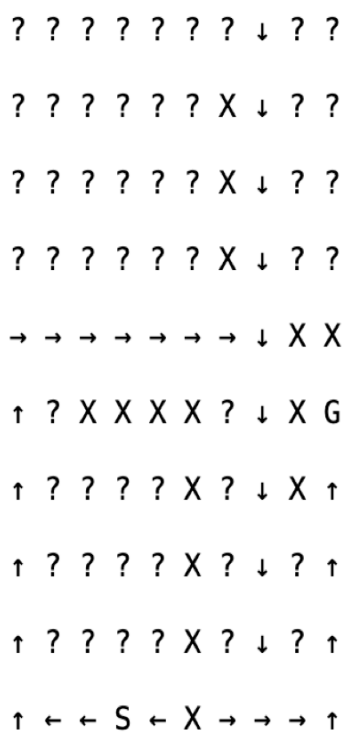


Figure 4.18: Policy map of algorithm deployed on the task shown in Figure 4.1 (right) set to stop GP at 40% average population region coverage. The total environmental query for this run was 243 queries. The starting action was West.

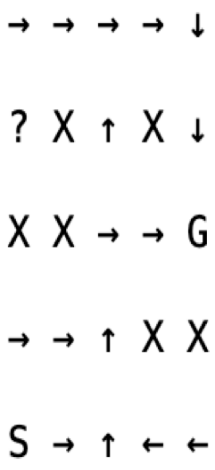


Figure 4.19: Superimposed policy maps over 30 runs of QTRB for the task shown in Figure 4.2 (left). The starting action was East. This figure shows QTRB's ability to reach the goal from a large percentage of states it has visited during RL, though at a trade-off of the sum of environmental queries over all runs. 100% of the actionable states can make it to the goal in this task.

majority of the non "?" actions suggested for that cell. Naturally, other superposition operators could be defined.

Though these superimposed solutions show a policy that leads the majority of states to the goal, there are some drawbacks. There is a trade-off in the efficiency of QTRB while creating these figures. The environmental query totals are increased by a multiple of the number of runs on QTRB, making the total query efficiency:

$$total\_queries = \#\_runs * \#\_gens * GP\_individual\_steps$$

This is ultimately due to the nature of GP in QTRB; most champion teams in a population of a single run will converge to a single solution. This typically results in similar policies among agents from the same run. While this trade-off must exist in this project due to the hard converging nature of GP, recommended future work suggests the introduction of some genetic variation operators. As discussed further in Chapter 5, this would allow QTRB to have a variety of policies within a single population of champions after Q-learning. Overall genetic variation operators may eliminate the drawback in efficiency these superimposed solutions currently have.

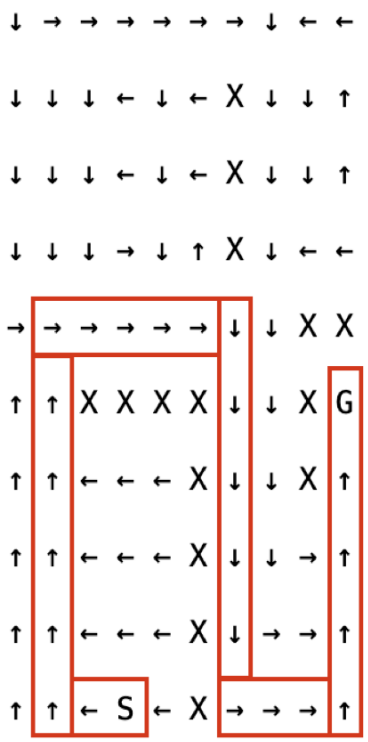


Figure 4.20: Superimposed policy maps over 30 runs of QTRB for the task shown in Figure 4.1 (right). The starting action was West. This figure shows QTRB's ability to reach the goal from a large percentage of states it has visited during RL, though at a trade-off of the sum of environmental queries over all runs. 94% of the actionable states made it to the goal in this task. The red boxes were added to better visualize the optimal path from start to goal.

Table 4.2: Number of steps needed for the shortest possible path from start to finish (optimal path) compared with the number of steps the sampled champions teams policies contained. The comparison is provided for each presented task in this project (see Appendix).

| **Task** | **Optimal Step Count** | **Sample Policy** | **Policy Step Count** | **Generation** |
|---|---|---|---|---|
| $5 \times 5$ Maze | | | | |
| 4.1 (left) | 10 | 4.4 | 10 | 5 |
| 4.2 (left) | 6 | 4.19 | 6 | 4 |
| $10 \times 10$ Maze | | | | |
| 4.1 (right) | 20 | 4.5 | 20 | 47 |
| 4.2 (right) | 10 | 4.6 | 10 | 14 |
| $20 \times 20$ Maze | | | | |
| A.1 | 18 | A.3 | 18 | 5 |
| A.4 | 19 | A.6 | 19 | 14 |
| A.7 | 9 | A.9 | 13 | 7 |
| A.10 | 21 | 4.8 | 29 | 45 |
| A.11 | 9 | A.13 | 9 | 12 |
| A.14 | 20 | A.16 | 20 | 49 |
| A.17 | 20 | A.19 | 28 | 76 |
| A.20 | 29 | A.22 | 29 | 27 |
| A.23 | 10 | A.25 | 10 | 21 |
| A.26 | 19 | A.28 | 23 | 33 |
| A.29 | 16 | A.31 | 16 | 15 |
| A.32 | 13 | A.34 | 15 | 33 |
| A.35 | 17 | A.37 | 17 | 9 |
| A.38 | 20 | A.40 | 20 | 16 |
| A.41 | 8 | A.43 | 8 | 4 |
| $50 \times 50$ Maze | | | | |
| A.44 | 73 | 4.10 | 75 | 89 |
| A.45 | 57 | A.47 | 69 | 42 |
| A.48 | 44 | A.50 | 60 | 33 |
| A.52 | 46 | A.54 | 46 | 130 |
| A.55 | 36 | A.57 | 40 | 45 |

Table 4.3: Number of steps needed for the shortest possible path from start to finish (optimal path) compared with the average steps over the task's RL winners. The comparison is provided for each presented task in this project (see Appendix).

| **Task** | **Optimal Step Count** | **Step Count Average** | **Step Count Std. Dev.** |
|---|---|---|---|
| $5 \times 5$ Maze | | | |
| 4.1 (left) | 10 | 10 | 0 |
| 4.2 (left) | 6 | 7.4 | 1.9 |
| $10 \times 10$ Maze | | | |
| 4.1 (right) | 20 | 24.5 | 3.2 |
| 4.2 (right) | 10 | 15.8 | 5.4 |
| $20 \times 20$ Maze | | | |
| A.1 | 18 | 22.8 | 4.1 |
| A.4 | 19 | 25.1 | 5 |
| A.7 | 9 | 23.5 | 10 |
| A.10 | 21 | 29.7 | 5 |
| A.11 | 9 | 11.2 | 3.4 |
| A.14 | 20 | 33.8 | 9.5 |
| A.17 | 20 | 35.9 | 2.6 |
| A.20 | 29 | 31.5 | 2.6 |
| A.23 | 10 | 19 | 11 |
| A.26 | 19 | 24.7 | 3.2 |
| A.29 | 16 | 24.45 | 10.5 |
| A.32 | 13 | 25 | 4.6 |
| A.35 | 17 | 27.2 | 7.2 |
| A.38 | 20 | 30.2 | 10.1 |
| A.41 | 8 | 12.8 | 7.9 |
| $50 \times 50$ Maze | | | |
| A.44 | 73 | 85.9 | 6.7 |
| A.45 | 57 | 80 | 5 |
| A.48 | 44 | 62 | 7 |
| A.52 | 46 | 61.7 | 12.9 |
| A.55 | 36 | 59.25 | 11.8 |

Table 4.4: Total environment queries on various tasks for various algorithms for comparison.

| **Task** | **QTRB** | **DynaQ 0%** | **DynaQ 2%** | **DynaQ 5%** |
|---|---|---|---|---|
| Fig 4.1R | 2444 | 17320 | 4506 | 3775 |
| Fig 4.1L | 409 | 1961 | 1944 | 1108 |
| Fig 4.2R | 1962 | 4077 | 1936 | 1196 |
| Fig A.10 | 2828 | 27115 | 4107 | 4360 |
| Fig A.44 | 7041 | 340717 | 21288 | 14009 |

# Chapter 5

# Conclusion

## 5.1 Summary

This thesis consisted of developing a novel algorithm that is able to use Q-learning to derive policy from teams of modules parameterized with local reward signal. Named QTRB, the algorithm was able to solve a variety of grid worlds, ranging in size from $5 \times 5$ to $50 \times 50$.

As discussed in previous chapters, the original rationale behind developing QTRB was to address shortcomings observed in traditional GP algorithms. GP solutions are typically randomly initialized for learning, thus there is no guarantee they will align with the properties of the task they set out to solve. During evolution in QTRB, solutions are built from the ground up in order to reflect the physical structures and laws of the task. Local reward is what drives this non-random construction, which conveniently allows RL algorithms, such as Q-learning, to derive information from the evolved solutions in order to find optimal paths within them. Given the structure of candidate solutions, this learning is usually much more efficient than traditional tabular RL.

The thesis set out to answer the following research questions with respect to the mentioned rationale:

1. Are local rewards sufficient for parameterizing states and actions into useful modules from which entire policies can be constructed?

2. Once a module encounters the goal state, is Q-value propagation sufficient for optimizing the plan defined by the set of modules comprising the team?

3. What implications are there for the sample efficiency of the resulting population of teams?

The next section will provide the findings in relation to each of these questions in reference to the presented results in the previous chapter. The sections to follow are the implications of such discussion, then recommended future work to improve

QTRB.

## 5.2 Interpretations

This section aims to analyze the results presented in the previous chapter, with respect to the main research questions of this study.

### Are local rewards sufficient for parameterizing states and actions into useful modules from which entire policies can be constructed?

The original hypothesis paired with this question was that local rewards would indeed be sufficient for constructing such modules. The collected results for QTRB support this hypothesis. This was touched on in the results section, but to reiterate, the policy figures for each task (shown in Chapter 4 and Appendix A), provide sufficient evidence that entire policies can be derived by the teams constructed in QTRB.

The fitness curves shown for each task also support the success of the iterative module-building nature of QTRB. As the stop criteria (generations or task coverage) went on, the fitness curves typically increased and usually converged at a value close to the goal value. This implies that the modules built around the environment were able to grow from start to goal; obtaining enough of a meaningful abstraction of the environment for the Q-learning agent to use to derive policy. The exceptions to the fitness rules will be discussed in the following section.

Additionally, the local reward was shown to play a crucial role in parameterizing the modules. This is highlighted in Figure 4.16, where the policy derived by QTRB without reward parameterization of modules is next to nonsensical in the context of the environment. This supports the theory that local reward is the driving factor of linking constructed modules with environmental meaning.

### Once a module encounters the goal state, is Q-value propagation sufficient for optimizing the plan defined by the set of modules comprising the team?

The hypothesis for this research question was that Q-value propagation would be sufficient in optimizing a team's set of modules. The so-called "bucket brigade" effect

of reward being passed from region to region, though diminished in each pass, is clearly shown in the Q-value map figures for each task. This clearly supports the hypothesis as most of the thesis Q-value maps roughly resemble a clear path from start to goal, and occasionally even go as far as being copies of the policy figures themselves.

The Q-value propagation highlights that Q-learning can indeed be used to not only stitch together the modules constructed during evolution but find the best path of modules to take from start to goal. This is shown in the policy figures throughout but is best visualized in both Table 4.2 and 4.3, which shows the cell counts of the optimal path for a task, compared to the length of the path a QTRB run was able to find. Although QTRB becomes somewhat less accurate as the task size scales (still usually within 20 cells), overall the paths Q-value finds for policies are either optimal or very close to it. Though originally unintended, this finding goes beyond the original hypothesis and is quite an impressive metric of success for QTRB.

The last set of results that supports the effective use of Q-value propagation is the figures and tables relating to the superimposed module-sets after RL. This supports the overall effectiveness of Q-learning in QTRB in a less concentrated way than the optimal path search does. Although much less efficient to find than those of a single run, the superimposed figures show the QTRB's ability to abstract environmental meaning from anywhere in the task. Once again, these findings were not an original intention of QTRB, but further highlight the effectiveness of pairing the GP technique and Q-learning.

**What implications are there for the sample efficiency of the resulting population of teams?**

Given the nature of QTRB, the hypothesis for the final question was efficiency would be a strength of the algorithm in terms of total direct environmental queries needed to derive policy. As mentioned, Q-learning can be deployed on the module-set rather than the actual environment itself. This allows for the agent to learn from a model, which reduces the need for any actual direct environment queries taken during any RL in QTRB. Thus, the direct environmental queries taken during evolution are the only queries that contribute to the total environmental queries in each task. It should also

be noted that the totals can drastically change as the maximum team population increases; this should be considered during hyperparameter tuning if maintaining efficiency is of interest.

The query totals taken during evolution in QTRB are displayed in tables throughout the Results chapter. In order to test these totals with another RL algorithm, DynaQ was deployed on various tasks. DynaQ is discussed in depth in the Background chapter, but it was chosen as it has various available planning mechanisms [24], which drew some parallels to the nature of QTRB when its evolution is considered as a form of planning. Overall, the comparison supports the claim that as the size of the environment scales up, QTRB takes significantly fewer direct environmental queries. This is a discussed drawback of RL, as it must spend a great number of resources to explore a solution space before being able to derive policy. QTRB provides the RL agent with an abstracted model of the environment, freeing up the resources needed for otherwise direct environmental exploration.

### 5.3 Implications

Overall, the research questions drove the development of QTRB. QTRB was able to use local reward to construct module-sets in which an RL agent could derive policy from using Q-value propagation. This model-based RL allowed agents to conserve direct environment sample efficiency while learning. Given these findings, this section will briefly discuss the implications of QTRB.

The most notable finding of this thesis is that programs can be built in a non-random way in order to create meaningful abstractions of a given environment. This is especially important in direct reference to Downing's discussed work, and can even be thought of as an advancement in what he set out to do [7]. QTRB successfully combines GP and RL in order to solve grid world tasks, without the use of subgoals throughout the environment. This is important as it fills in some gaps in the field of hybrid algorithms, pairing GP closer to RL than before, as a local reward now drives both program and policy development.

Another important finding of this study further supports the efficiency of planning and model-based RL. As shown, the number of direct environment queries QTRB made compared to traditional RL was significantly fewer. These findings are especially

relevant to any application of RL which requires as few direct environmental queries as possible. Overall, letting the agent learn on an abstracted version of the real environment avoids numerous direct environment queries.

## 5.4 Limitations

### 5.4.1 Fitness Function

The main limitation of local reward-driving GP was the relationship between iterative team development and the fitness function. The fitness function was based simply on the total local reward collected by a team during evolution. Thus, teams which developed aimlessly without ever reaching the goal would "bloat", despite finding very high fitness scores. These bloated teams would cause the fitness curves to trail off toward values much higher than what a successful team would actually achieve. Although fitness was high in these solutions, they realistically had little chance of ever solving the task.

Booth et al. discuss performance function and their alignment with the given task [3]. Throughout their study, Booth et al. highlight the difficulty of finding a reward function which fits a task, even finding that reward functions are usually overfit for particular tasks and agents. This thesis is a good example of Booth et al.'s work generalizing to GP. It implies that the fitness function does not always predict actual performance, which is observed here. The remainder of this subsection will discuss potential fixes for this limitation, all revolving around redefining aspects of the evolution of the fitness function to combat team bloat.

The step count penalty was actually implemented to combat these trailing fitness functions, which did somewhat solve the issue, but occasionally caused the whole population of teams to completely drop off. This drop-off was present in runs that even had champion solutions, which implies that there is some "tipping point" during the evolution of QTRB where a solution is between being underdeveloped and bloated.

This tipping point implies that through iterative development, the first champion solution may be the "peak" within their "family", as their children begin to drop off due to too many steps. This claim is supported by the tipping point shown in various fitness curves. Additionally, the majority of optimal solutions were found before half

of their task's respective generation count, which perhaps points to optimal teams developing earlier in evolution, before this tipping point.

After all, the step penalty itself was more or less a bandage over this issue, a real solution may be found by investigating the way the local reward is accessed by developing modules or changing the fitness function itself. As for solutions relating to accessing local reward, a system which limits the amount of local reward a module can access when a state is visited may suffice. For example, a cell can only assign local reward to one module in the team's module-set. This would certainly limit the amount of local reward assigned to each team, avoiding trailing fitness. This would perhaps also lead to the development of mechanisms involving modules competing for the yielded reward of a given shared state.

An alternative solution is redefining the fitness function itself to better reflect the task. A fitness function with the ability to judge candidate solutions on metrics separate from just local reward accumulation may be able to produce far superior teams. An example metric this function could keep an eye on would be team size; deleting any bloated teams from the population. More research is required to judge whether this is worthwhile, but as of now, the hypothesis is that there exists a redefined fitness function that is able to judge teams better than the simple accumulation of local reward over iterative development. Revisiting the work of Booth et al., it is important to be mindful to not overfit the performance function to QTRB itself to avoid biases in seemingly successful results [3].

### 5.4.2 Hanging Regions

As shown throughout Chapters 4 and 6, the rate at which the RL agent solves the task with the team's model, or win-rate, is low, especially as the task scales in size. After some observation, about half of these losses are attributed to "hanging regions". This is a consequence of region-building; where after RL, a region's q-value may be relatively high as it is on winning path, but only, say, half of the region is on the winning path. The rest of the region "hangs", and due to the region's high q-value, other more important regions with lower q-values (closer to the start) are blocked from getting on the winning path. An example of this is shown in Figure 5.1, where the left moving region (in red) is blocking the right moving region (in green). The key

properties of these regions are that the green is the starting region, so it will usually have a much lower q-value despite being part of the winning path, and half of the red region is also actually leading to the goal, which would contribute to a higher q-value for that region.

Similar to the fitness function limitation, there are a variety of solutions for the hanging regions for future work. One solution involves implementing penalties for the q-learning agent, which assigns penalties to regions that can potentially block winning regions. As these penalties could only be assigned once the final solution is known, these penalties would have to at least run after evolution. This may suggest that QTRB needs some sort of region clean-up step between GP and RL. Additionally, this given solution is only applicable in problems where the solution is already known, such as grid worlds. Future work may also involve expanding on this concept for cases where the solution is not known.

### 5.4.3 Novel Algorithm

Lastly, another obvious limitation of QTRB is it is a novel algorithm. A lot of mechanisms within the algorithm were a product of requirement during iterative design, shaped around the research questions. The continuous development of features and improvements for QTRB will overall improve the robustness of the algorithm, hopefully someday hosting it as a piece of "out-of-the-box" software. The next section provides insight into the next best set of tasks for future work, but other general improvements to QTRB include a proper development testing suite, increased performance tracking, and more runs against more variants of larger grid worlds.

Though this section discussed larger issues with the project, the next section will provide some additional limitations, but with direct suggestions for future work to amend them.

## 5.5 Future Work

This section will review improvements that can be implemented into the current project in the future. QTRB is a novel algorithm, so there are a lot of possibilities for future work to advance this project. However, two main improvements will be


Figure 5.1: An example of a hanging region, where the red outlined region blocks the green outlined region from moving to the winning path.

focused on for the remainder of the section. The improvements relate to population diversification in evolution and exploration in RL.

### 5.5.1 Diversifying Emerging Populations during Evolution

The GP used in the project converges quite hard into the first champion solution that emerges from evolution. Although other solutions occasionally emerge to increase the variety within the population, there is typically not too much difference between champion solutions. This is not so much an issue on smaller tasks, where a smaller set of actual solutions exist, but becomes more relevant as the size of the grid world grows, thus the number of varying solutions grow too. The lack of diversification may also be contributing to the previously discussed issue of teams becoming bloated. Overall, this observation is somewhat expected, as there are no diversity operators present in the current implementation of QTRB.

The obvious future improvement for the problem is to add some sort of diversity mechanic to the evolving population. It is hypothesized this improvement would give room for more emerging solutions, creating a much more diverse champion pool at the end of a run compared to the current implementation.

There are no specific diversity mechanisms planned for implementation, but an interesting example comes from the NEAT algorithm [21]. Simply put, before assessing each population pool during evolution, NEAT will organize solutions into "species", and have solutions compete only within their own species. This allows emerging solutions that go against the status quo to live for further development, while still allowing the strongest solutions to dominate most of the population.

Additionally, this improvement is hypothesized to further reduce the total environment query time of the QTRB in the superimposed case shown in the Results and Appendix chapters. A diversified module-set from a single run could potentially cover the entire task with visited regions, thus fewer QTRB runs would be required to produce such coverage.

### 5.5.2 Exploration

As mentioned, the weakest point of the result set seems to be the RL win-rate metric. To reiterate, as shown in various tables throughout the Results and Appendix chapter,

the RL win rate represents the amount of champion module-sets the RL agent was able to solve. This subsection will propose a mechanism which is hypothesized to improve this win rate.

The current implementation of QTRB uses "out-of-the-box" Q-learning, where agent exploration exists on the state level; the agent rolls for $\epsilon$-greedy action selection at each cell of the grid world. This is an issue since QTRB's agents execute meaningful movement on the region level, where the Q-value updates occur. The issue is highlighted when the agent rolls to choose a random, or exploratory, action in the middle of a region. The current implementation of the agent will take that random action within the region, then continue on with mostly exploitative actions. Other than edge cases, these random actions are not occurring frequently enough to actually drive any meaningful random movement for the agent to properly explore.

The implementation of this improvement would involve the learning agent ever only existing in a region's lower bound or the region upper bound. All other cells within a region are actually redundant, as the policy will always assign them the action (and fitness) from the lower or upper bound. Thus it is theorized that the agent is not exploring the module-set frequently enough, and is occasionally even getting stuck in exploitative loops that do not result in completing the task within the specified number of time steps.

From this analysis, it is hypothesized that the RL win rate will increase if the current implementation of the Q-learning agent functioned solely on a region level. An auxiliary hypothesis for this work is that the overall number of steps needed for the RL agent to solve the task will decrease compared to the current implementation. The mentioned number of steps is predicted to decrease because most steps in a region would be reduced by the number of cells in that given region. Overall, changing the agent to operate at the region level seems like a step in further coupling the GP and RL in QTRB, while also improving the overall efficacy and efficiency of the learning agent.

## Bibliography


[1] Timothy Atkinson, Detlef Plump, and Susan Stepney. Probabilistic graph programs for randomised and evolutionary algorithms. pages 63–78.

[2] Thomas Back, David B. Fogel, and Zbigniew Michalewicz. *Handbook of Evolutionary Computation*. IOP Publishing Ltd., GBR, 1st edition, 1997.

[3] Serena Booth, W Bradley Knox, Julie Shah, Scott Niekum, Peter Stone, and Alessandro Allievi. The perils of trial-and-error reward design: Misdesign through overfitting and invalid task specifications. 2023.

[4] M. Brameier and W. Banzhaf. A comparison of linear genetic programming and neural networks in medical data mining. *IEEE Transactions on Evolutionary Computation*, 5(1):17–26, 2001.

[5] C. Darwin. *The Origin of Species.* John Murray, 1859.

[6] Pedro M. Domingos. A few useful things to know about machine learning. *Communications of the ACM*, 55:78 – 87, 2012.

[7] Keith L. Downing. Reinforced genetic programming. *Genetic Programming and Evolvable Machines*, 2:259–288, 2004.

[8] Madalina M. Drugan. Reinforcement learning versus evolutionary computation: A survey on hybrid algorithms. *Swarm and Evolutionary Computation*, 44:228–246, 2019.

[9] Agoston E Eiben, James E Smith, et al. *Introduction to evolutionary computing*, volume 53. Springer, 2003.

[10] Stefan Elfwing, Eiji Uchibe, Kenji Doya, and Henrik I. Christensen. Evolutionary development of hierarchical learning structures. *IEEE Transactions on Evolutionary Computation*, 11(2):249–264, 2007.

[11] John H. Holland. *Adaptation in natural and artificial systems.* University of Michigan Press, 1975.

[12] Shotaro Kamio and Hitoshi Iba. Adaptation technique for integrating genetic programming and reinforcement learning for real robots. *Evolutionary Computation, IEEE Transactions on*, 9:318 – 333, 07 2005.

[13] 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.

[14] Jong R. Koza. *Genetic programming: on the programming of computers by means of natural selection*. The MIT Press, 1st edition, 1992.

[15] Shingo Mabu, Kotaro Hirasawa, and Jinglu Hu. A Graph-Based Evolutionary Algorithm: Genetic Network Programming (GNP) and Its Extension Using Reinforcement Learning. *Evolutionary Computation*, 15(3):369–398, 09 2007.

[16] Darío Maravall, Javier de Lope, and José Antonio Martín H. Hybridizing evolutionary computation and reinforcement learning for the design of almost universal controllers for autonomous robots. *Neurocomputing*, 72(4):887–894, 2009. Brain Inspired Cognitive Systems (BICS 2006) / Interplay Between Natural and Artificial Computation (IWINAC 2007).

[17] Julian Miller and Andrew Turner. Cartesian genetic programming. In *Proceedings of the Companion Publication of the 2015 Annual Conference on Genetic and Evolutionary Computation*, GECCO Companion '15, page 179–198, New York, NY, USA, 2015. Association for Computing Machinery.

[18] Conor Ryan. Grammatical evolution. In *Annual Conference on Genetic and Evolutionary Computation*, 2001.

[19] David Silver. Lectures on reinforcement learning. https://www.davidsilver.uk/teaching/, 2015.

[20] Lee Spector, Erik Goodman, A. Wu, W. Langdon, H. Voigt, M. Gen, S. Sen, M. Dorigo, S. Pezeshk, M. Garzon, E. Burke, and Morgan Publishers. Autoconstructive evolution: Push, pushgp, and pushpop. *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001)*, 05 2001.

[21] Kenneth O. Stanley and Risto Miikkulainen. Evolving neural networks through augmenting topologies. *Evolutionary Computation*, 10(2):99–127, 2002.

[22] Richard S. Sutton. Learning to predict by the methods of temporal differences. *Machine Learning*, 3(1):9–44, 1988.

[23] Richard S. Sutton. Integrated architectures for learning, planning, and reacting based on approximating dynamic programming. In Bruce Porter and Raymond Mooney, editors, *Machine Learning Proceedings 1990*, pages 216–224. Morgan Kaufmann, San Francisco (CA), 1990.

[24] Richard S. Sutton and Andrew G Barto. *Reinforcement Learning: an Introduction*. Cambridge, Massachusetts: The MIT Press, 2nd edition, 2018.

[25] Christopher Watkins and Peter Dayan. Q-learning. *Machine Learning*, 8:279–292, 1992.

# Appendix A

## A.1 $20 \times 20$ Figures

This section contains any $20 \times 20$ tasks discussed, but not explicitly mentioned, throughout the Results chapter. For a quick reference, the red dot in each figure is the start state while the blue square is the goal state. For the policy maps, the red boxes were added to better visualize the path found by Q-learning.

Table A.1 and Table A.2 show various performance statistics from QTRB's runs on the given tasks. In Table A.1, the average step count refers to the number of environmental queries QTRB made in total, averaging over 30 runs. The # of RL runs refers to the number of champion teams GP produced for RL to learn on. % of RL wins refers to the win rate of RL on the given champion teams. Step Penalty refers to the number of steps a given team could take in GP before receiving large negative reinforcements. In Table A.2, # Superimposed Actionable Cells refers to the number of cells containing an action when all champions teams over all runs were superimposed onto the given task. The % Superimposed cells leading to the goal refers to the subset of those actionable cells that were connected to a path that led to the goal.

## A.2 $50 \times 50$ Figures

This section contains any $50 \times 50$ tasks discussed, but not explicitly mentioned, throughout the Results chapter. For a quick reference, the red dot in each figure is the start state while the blue square is the goal state. For the policy maps, the red boxes were added to better visualize the path found by Q-learning.

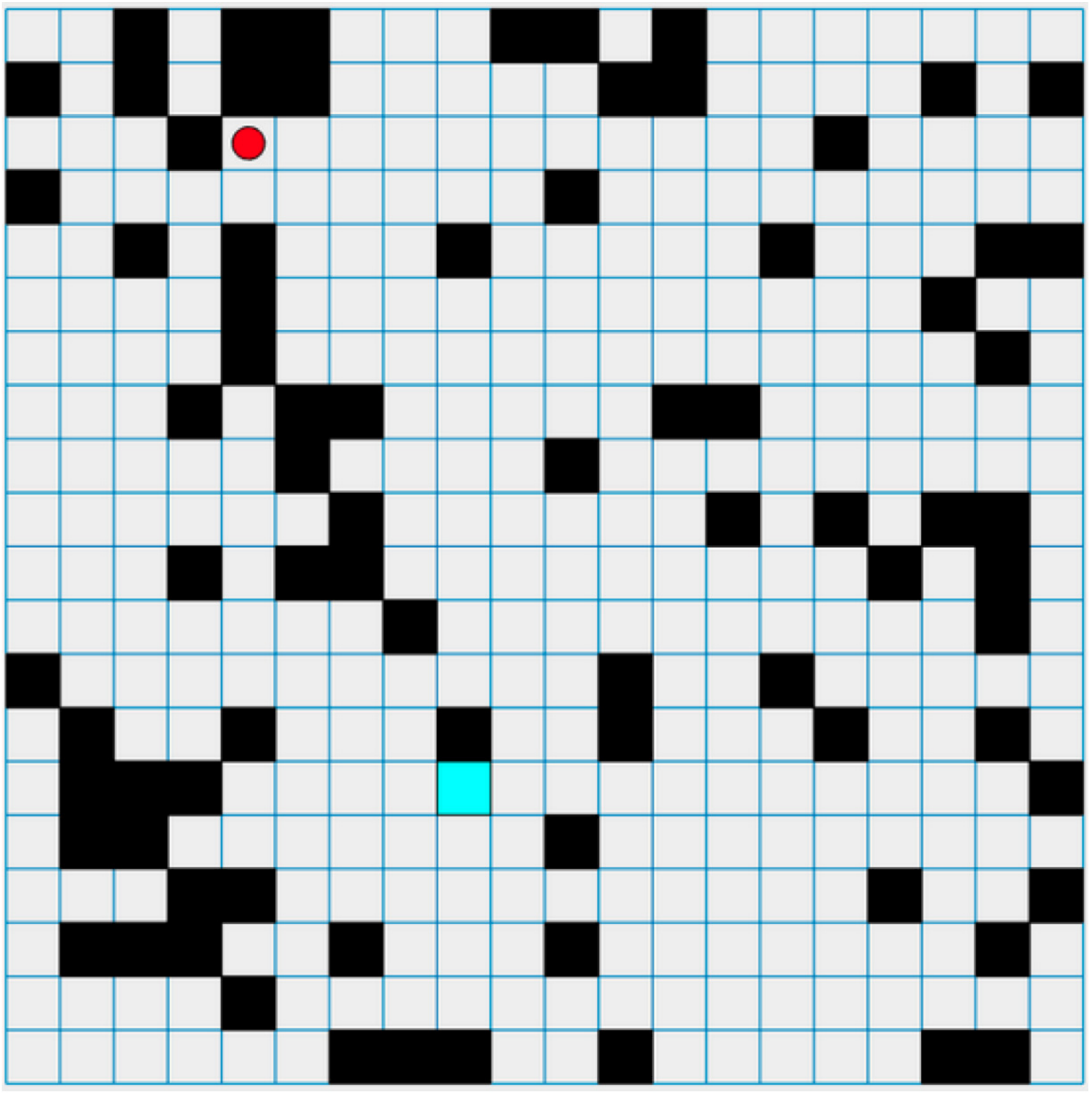

Figure A.1: $20 \times 20$ grid world task with start state (4, 17) and goal state (8, 5) (the bottom left corner is (0, 0)).

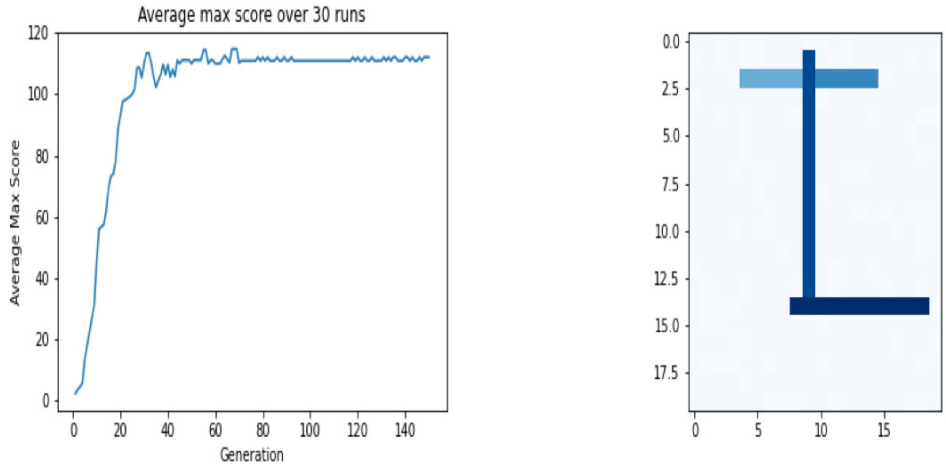


Figure A.2: QTRB results from Figure A.1. Shown from left to right: GP fitness curve and Q-value map.

Table A.1: Performance data for QTRB, all averaged over 30 runs for each given task.

| **Task Figure** | **Average Step Count** | **# RL Runs** | **% RL wins** | **Step Penalty** |
|---|---|---|---|---|
| A.1 | 3283 | 479 | 70 | 400 |
| A.4 | 3233 | 122 | 86 | 400 |
| A.7 | 2521 | 161 | 58 | 400 |
| A.10 | 4374 | 77 | 72 | 400 |
| A.11 | 2301 | 166 | 75 | 400 |
| A.14 | 2735 | 115 | 43 | 1000 |
| A.17 | 3102 | 94 | 54 | 1000 |
| A.20 | 4786 | 37 | 57 | 1000 |
| A.23 | 2624 | 106 | 60 | 1000 |
| A.26 | 4243 | 41 | 44 | 1000 |
| A.29 | 5187 | 36 | 86 | 1000 |
| A.32 | 4206 | 164 | 57 | 1000 |
| A.35 | 2683 | 110 | 70 | 1000 |
| A.38 | 5091 | 30 | 63 | 1000 |
| A.41 | 1400 | 131 | 61 | 1000 |

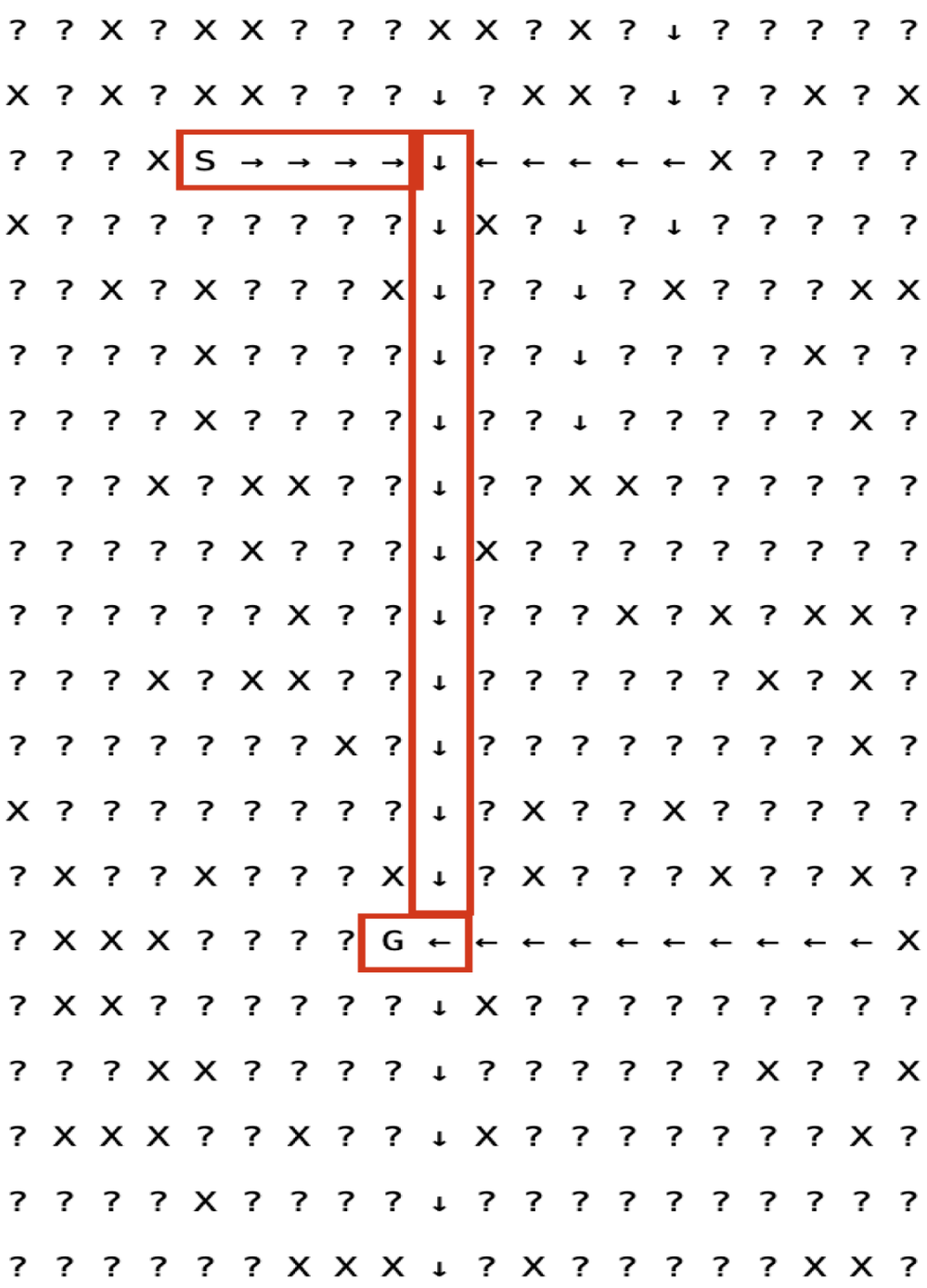


Figure A.3: Policy map for Figure A.1. The starting action was East.

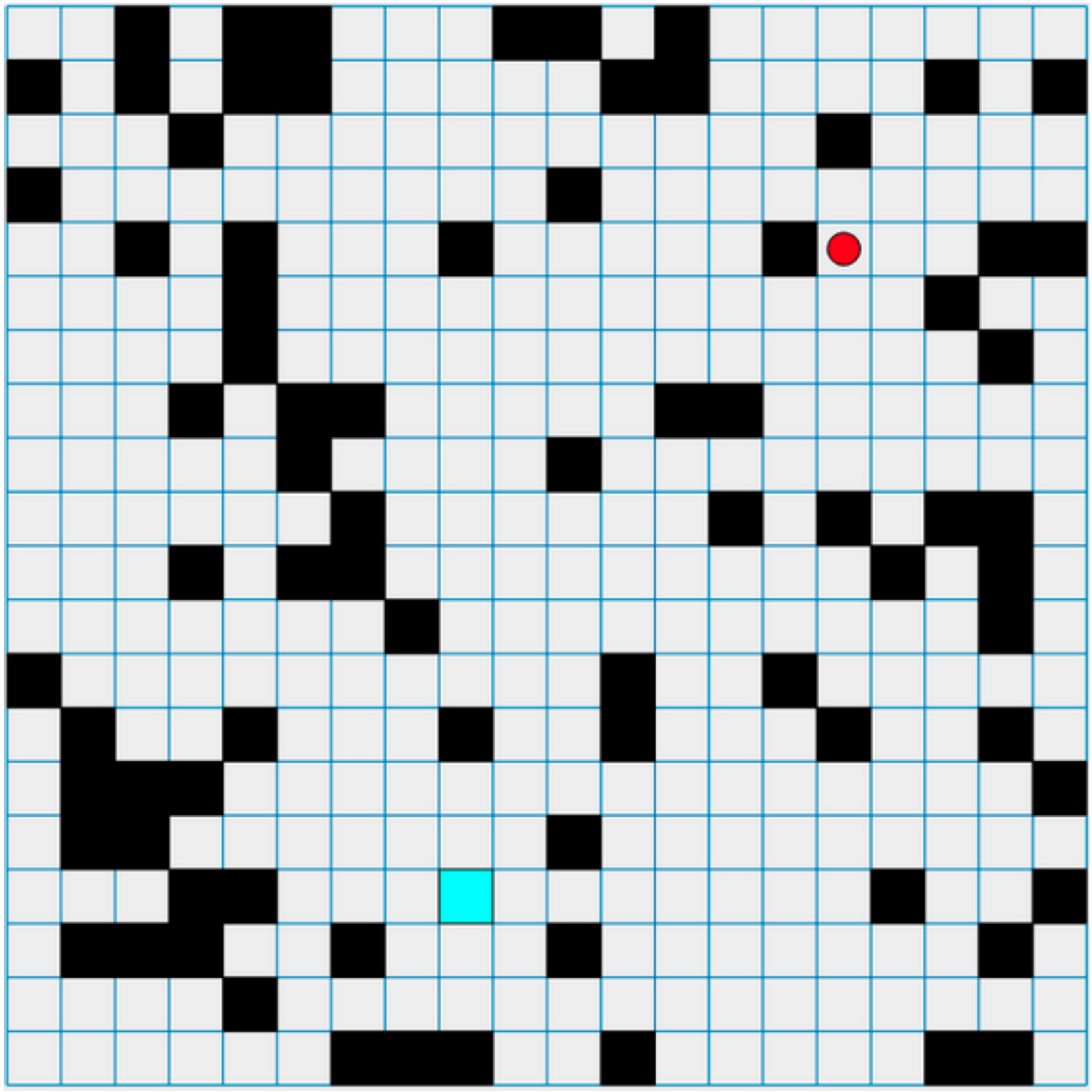

Figure A.4: $20 \times 20$ grid world task with start state (15, 15) and goal state (8, 3) (the bottom left corner is (0, 0)).

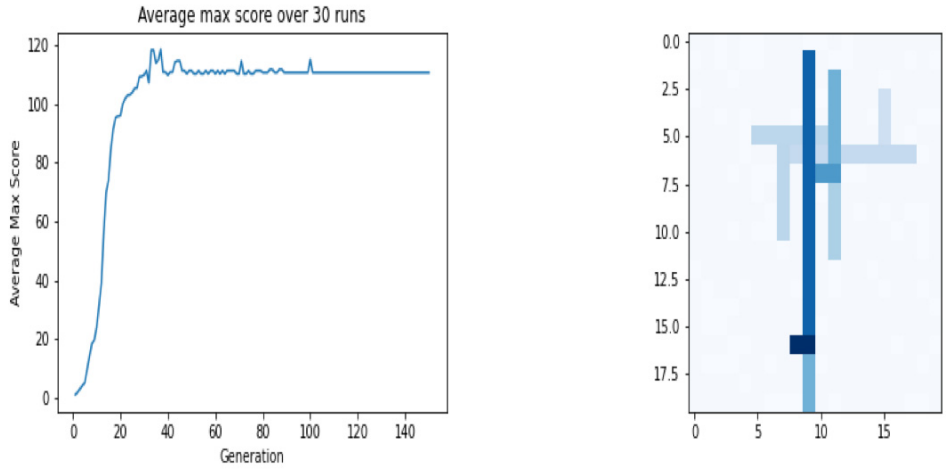


Figure A.5: QTRB results from Figure A.4. Shown from left to right: GP fitness curve and Q-value map.

Table A.2: Shown are the number of actionable cells each task had once champion solutions from each run of a given task were superimposed onto one grid world. The percentage of cells that were connected to a path which lead to the goal of the task is also shown.

| **Task Figure** | **# Superimposed Actionable Cells** | **% Superimposed cells leading to goal** |
|---|---|---|
| A.1 | 237 | 99 |
| A.4 | 246 | 95 |
| A.7 | 286 | 94 |
| A.10 | 254 | 95 |
| A.11 | 282 | 96 |
| A.14 | 308 | 64 |
| A.17 | 243 | 100 |
| A.20 | 203 | 63 |
| A.23 | 265 | 62 |
| A.26 | 299 | 100 |
| A.29 | 273 | 76 |
| A.32 | 285 | 94 |
| A.35 | 293 | 91 |
| A.38 | 261 | 87 |
| A.41 | 291 | 71 |

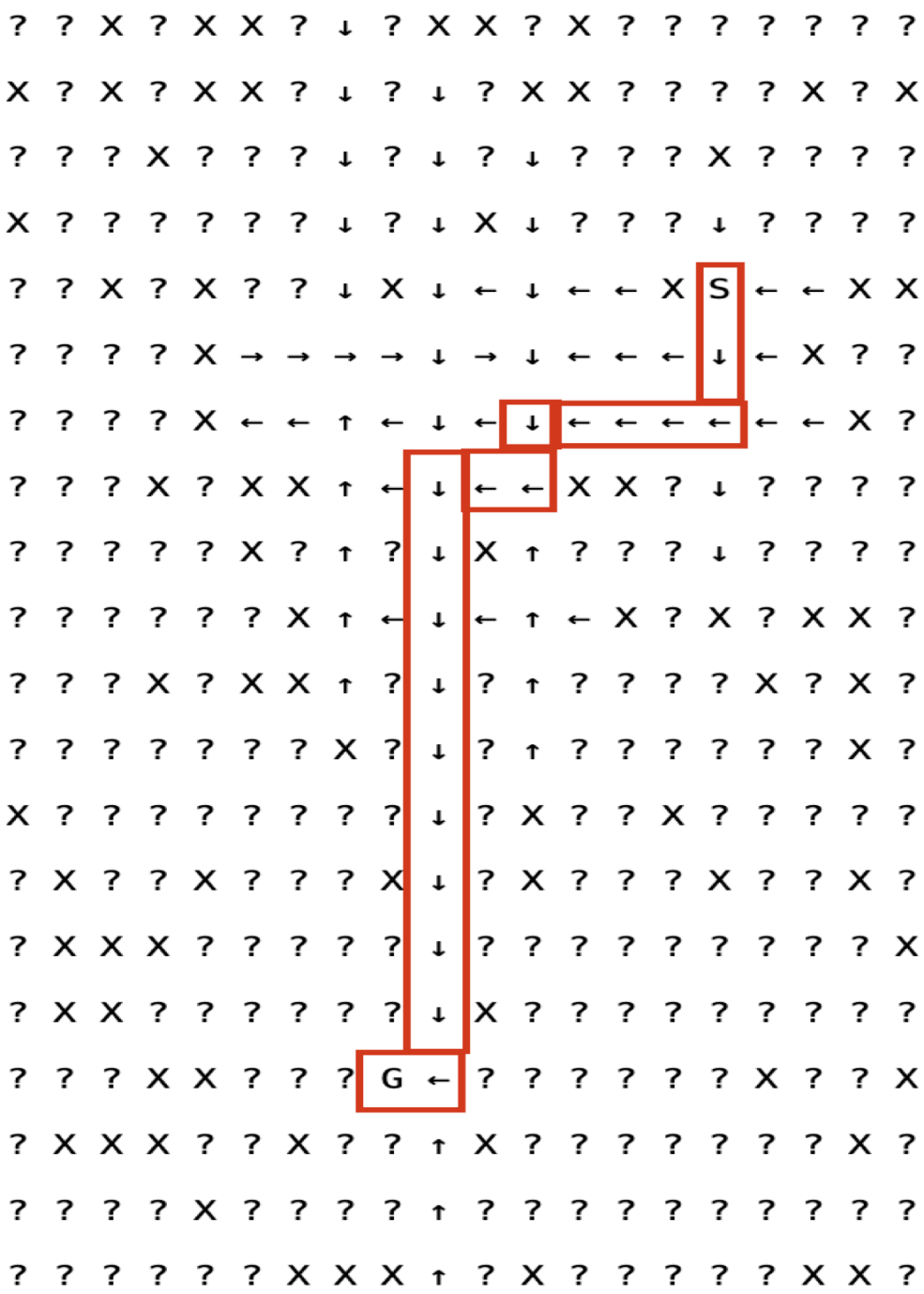


Figure A.6: Policy map for Figure A.4. The starting action was South.

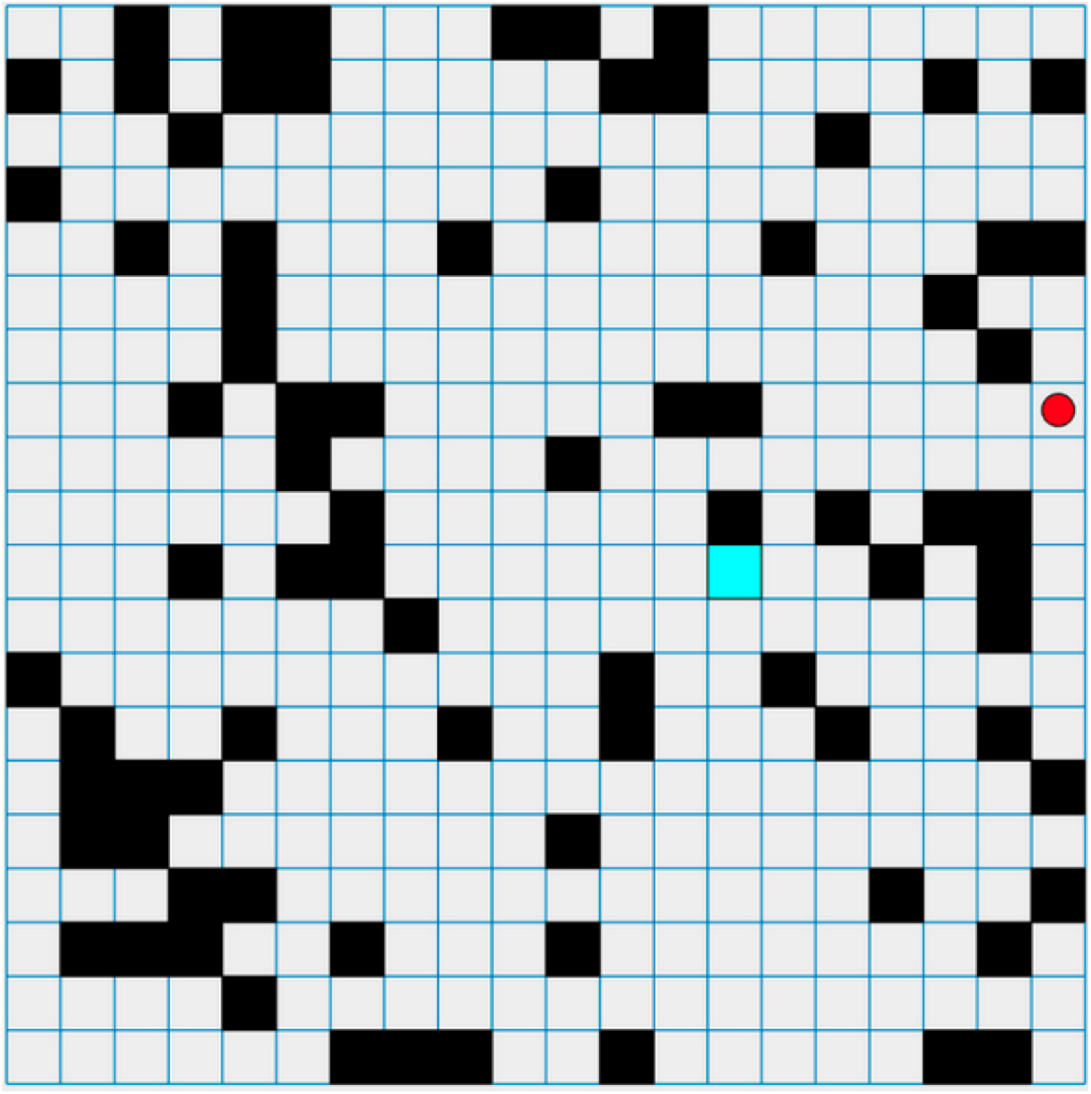

Figure A.7: $20 \times 20$ grid world task with start state (19, 12) and goal state (13, 9) (the bottom left corner is (0, 0)).

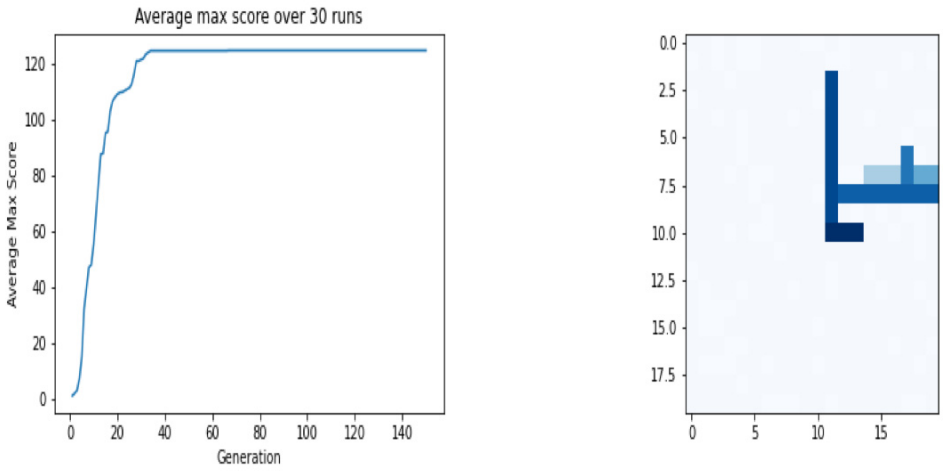


Figure A.8: QTRB results from Figure A.7. Shown from left to right: GP fitness curve and Q-value map.

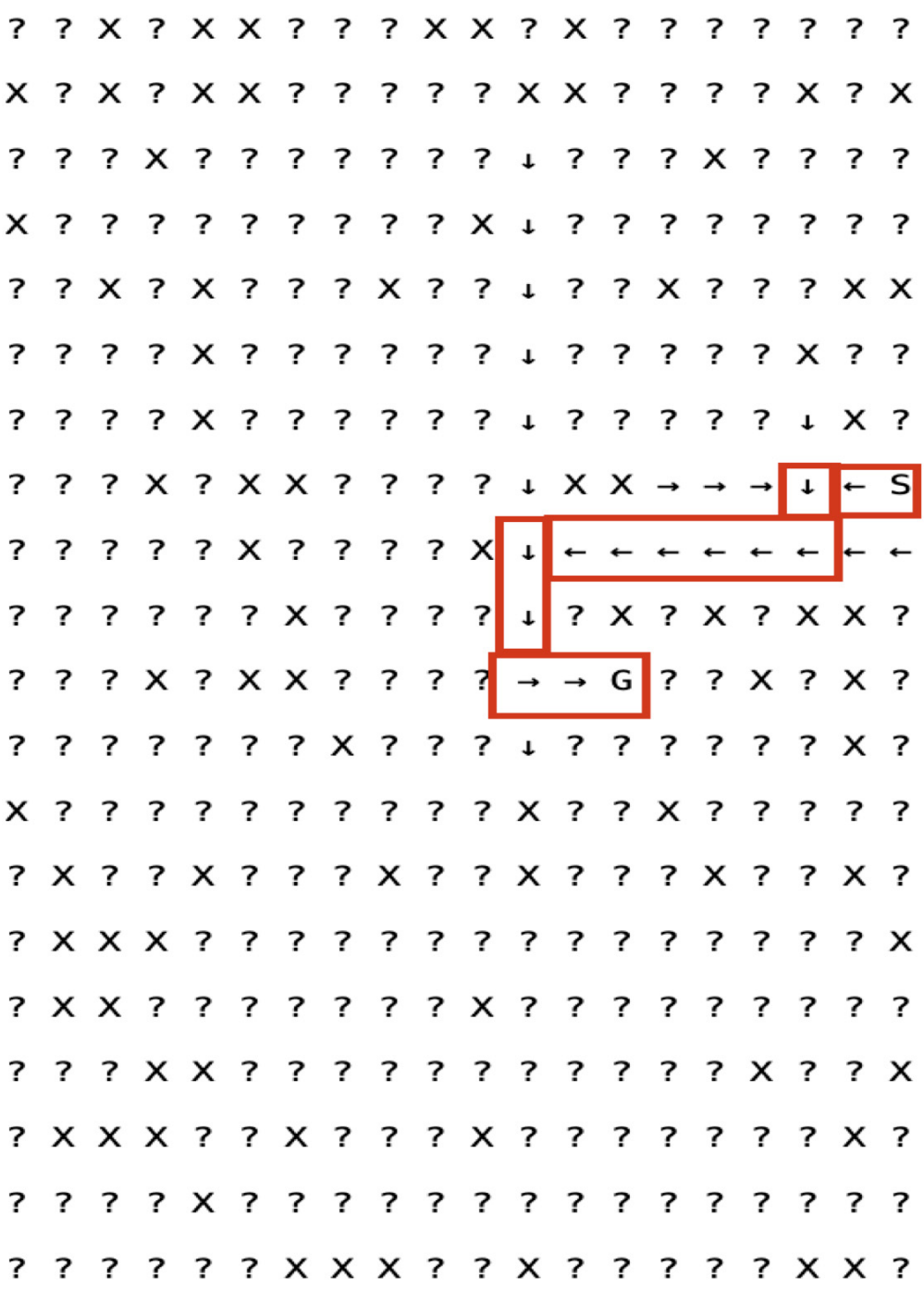


Figure A.9: Policy map for Figure A.7. The starting action was West.

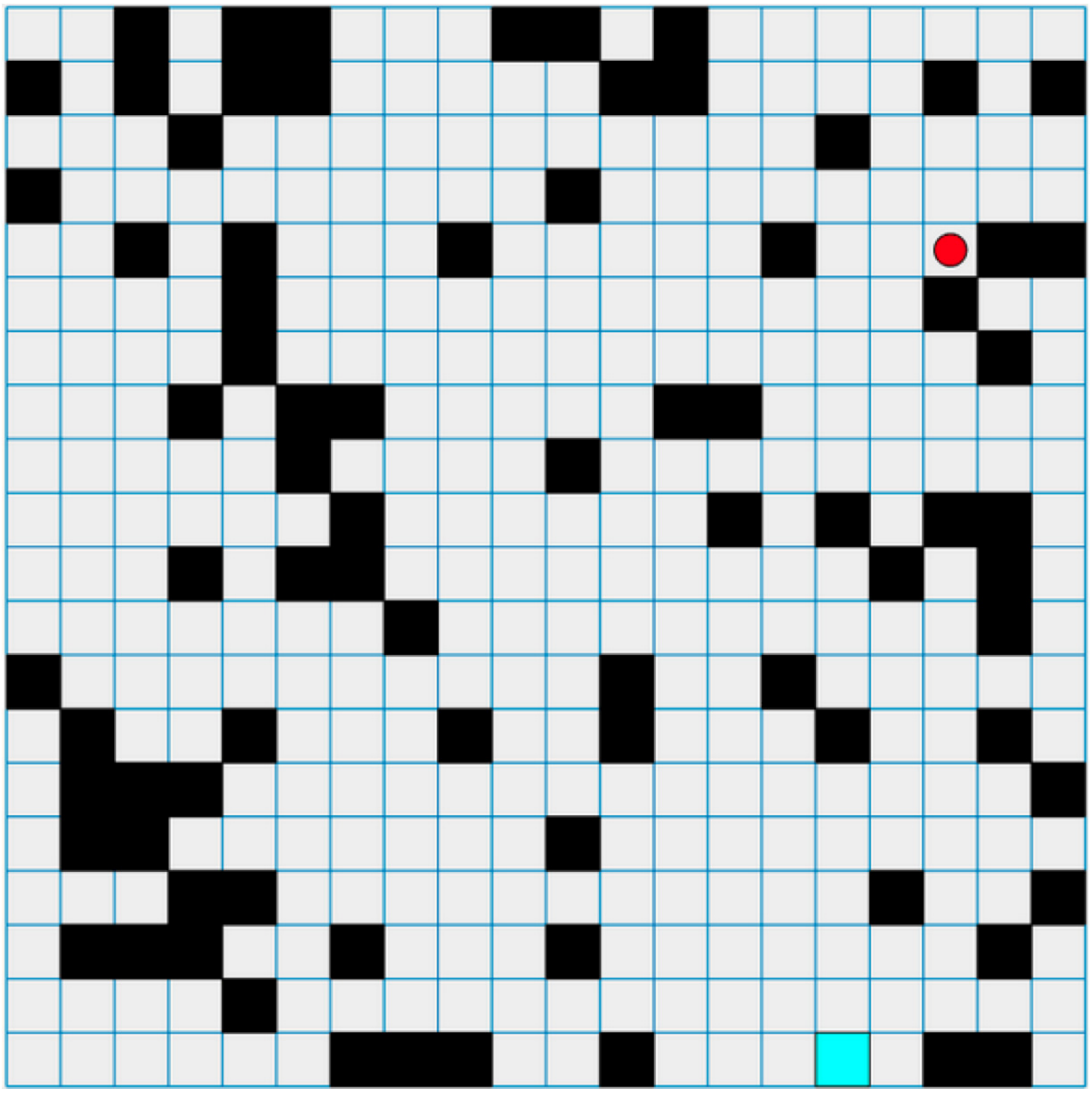

Figure A.10: $20 \times 20$ grid world task with start state (17, 15) and goal state (15, 0) (the bottom left corner is (0, 0)). This figure is in reference to the results presented in the Results Chapter.

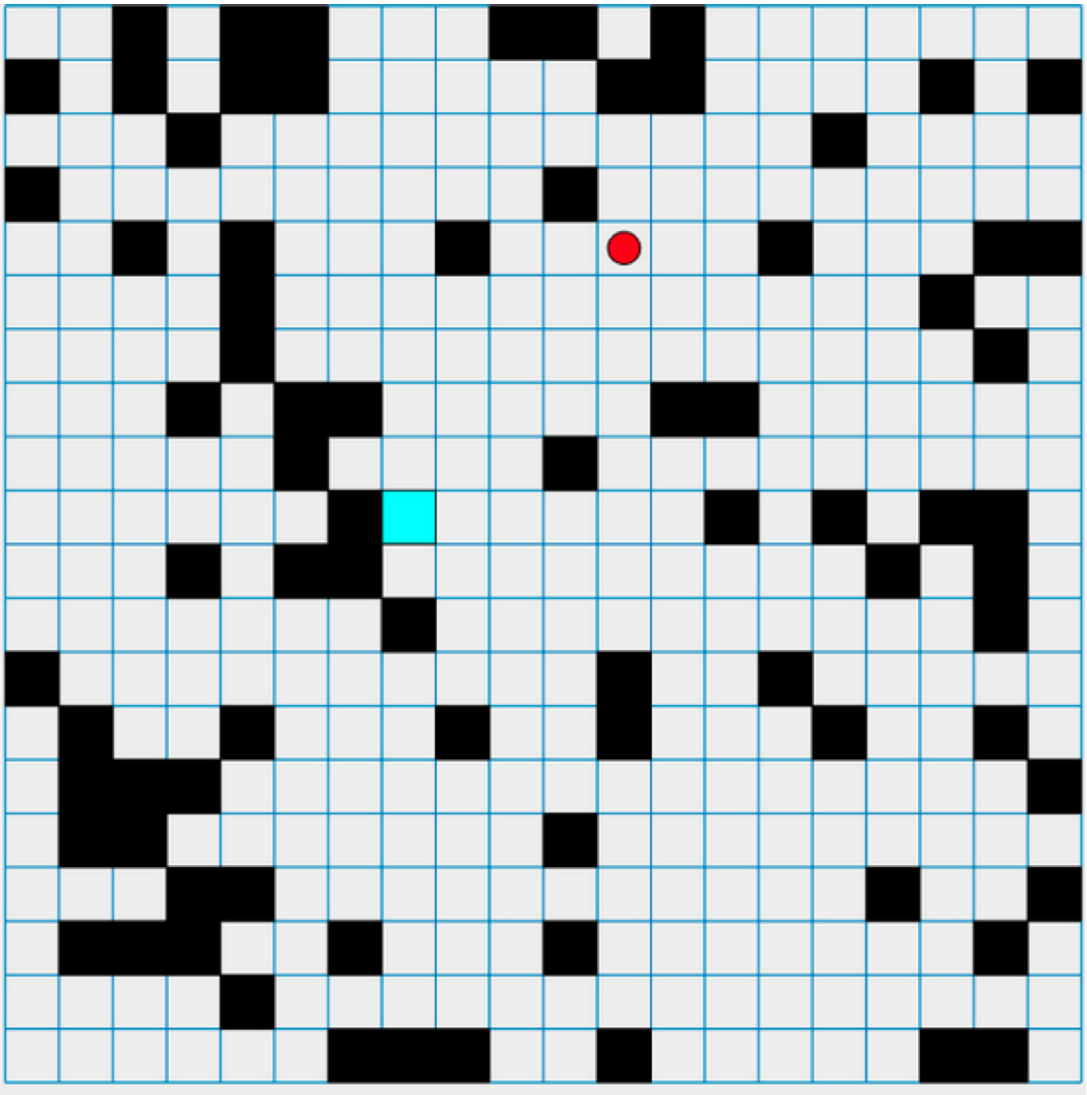

Figure A.11: $20 \times 20$ grid world task with start state (11, 15) and goal state (7, 10) (the bottom left corner is (0, 0)).

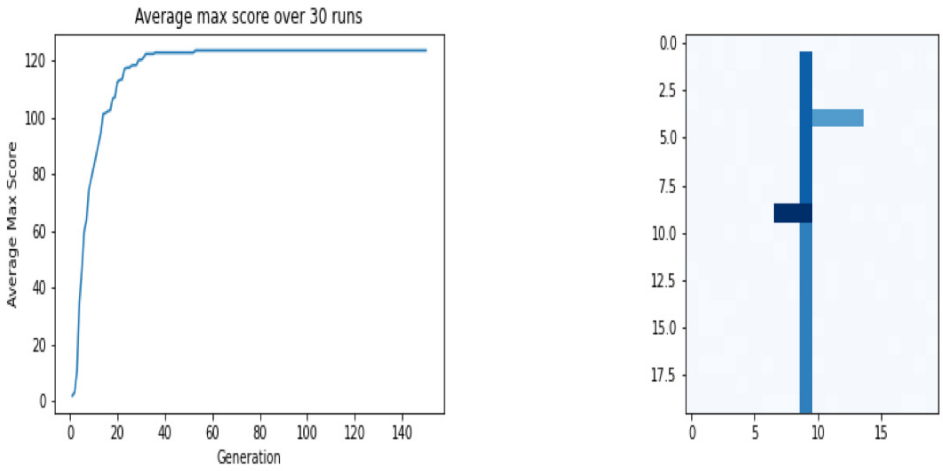


Figure A.12: QTRB results from Figure A.11. Shown from left to right: GP fitness curve and Q-value map.

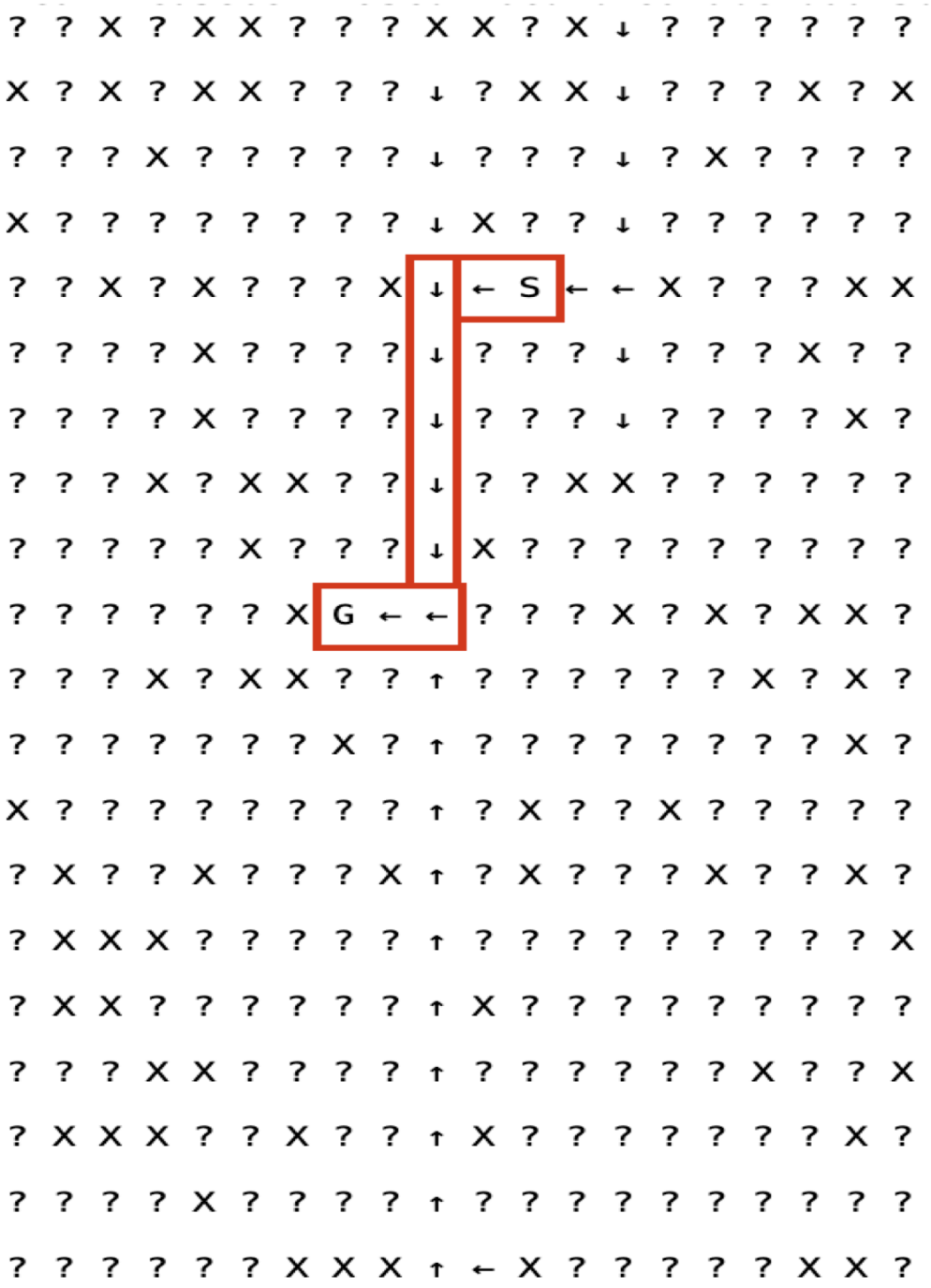


Figure A.13: Policy map for Figure A.11. The starting action was West.

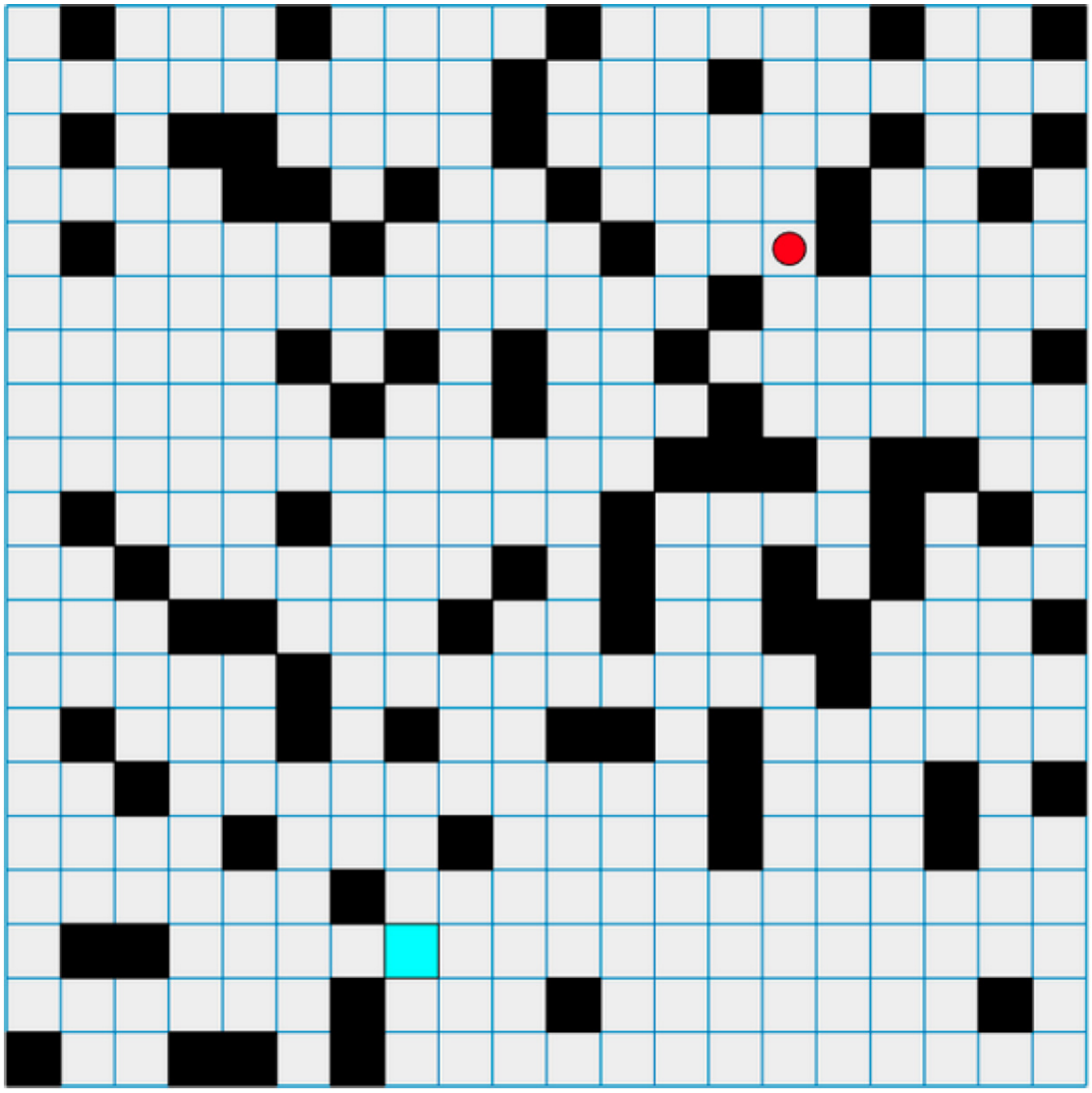

Figure A.14: $20 \times 20$ grid world task with start state (14, 15) and goal state (7, 2) (the bottom left corner is (0, 0)).

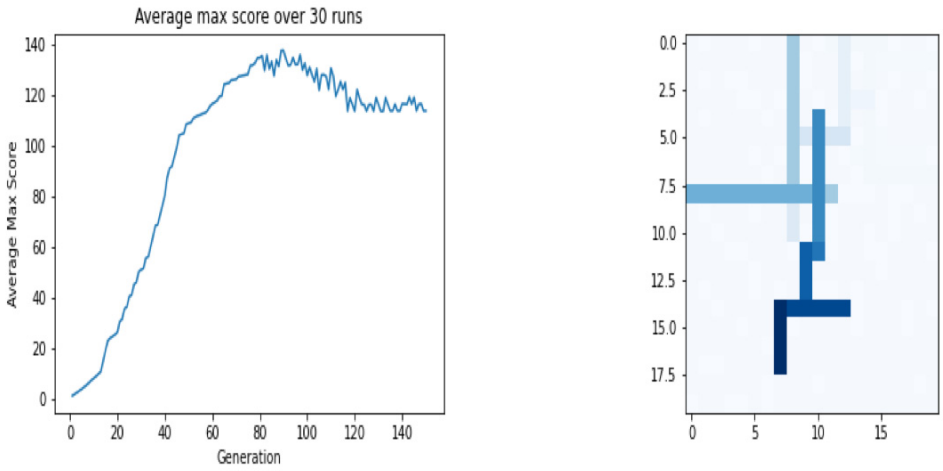


Figure A.15: QTRB results from Figure A.14. Shown from left to right: GP fitness curve and Q-value map.

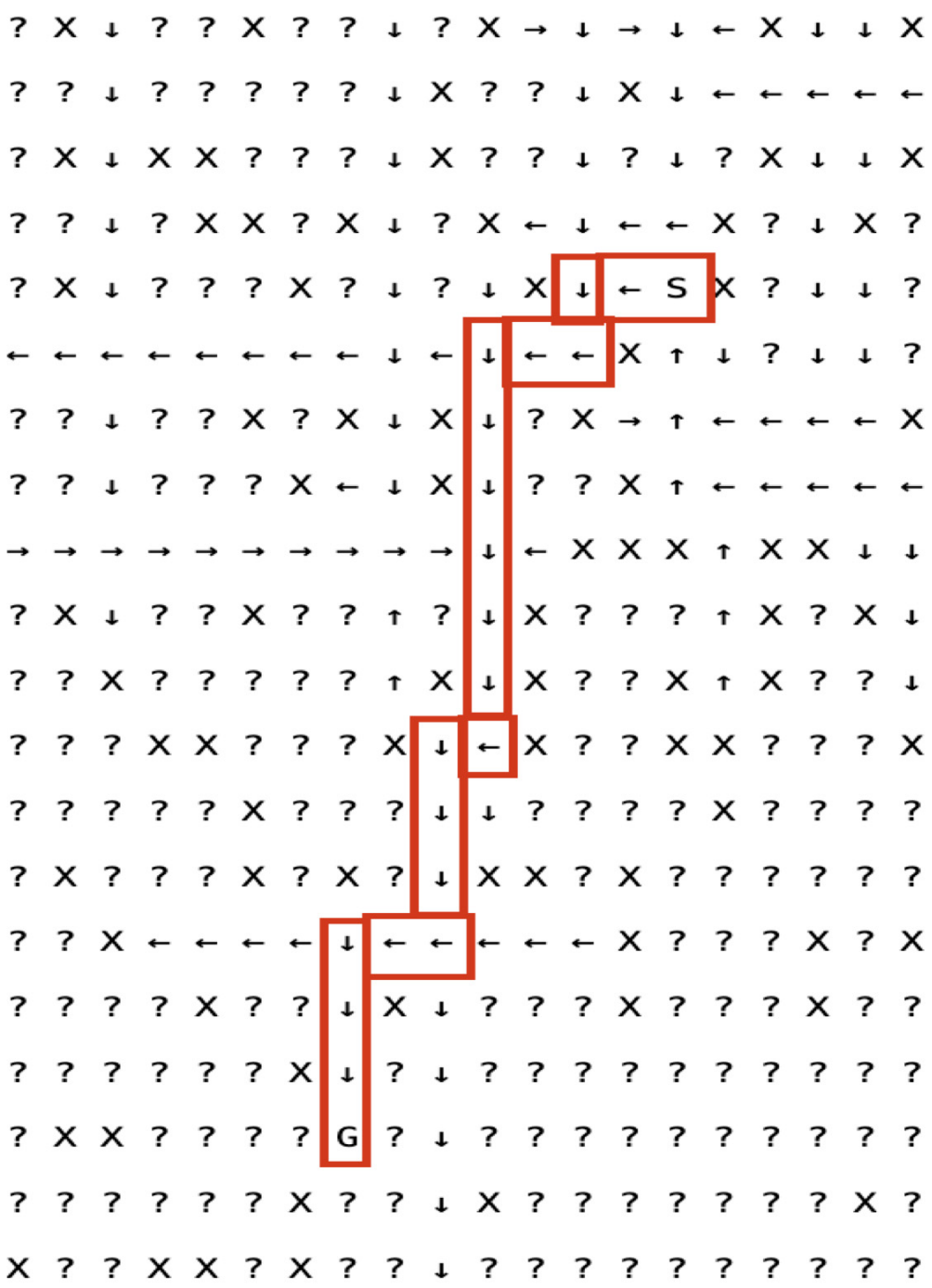


Figure A.16: Policy map for Figure A.14. The starting action was West.

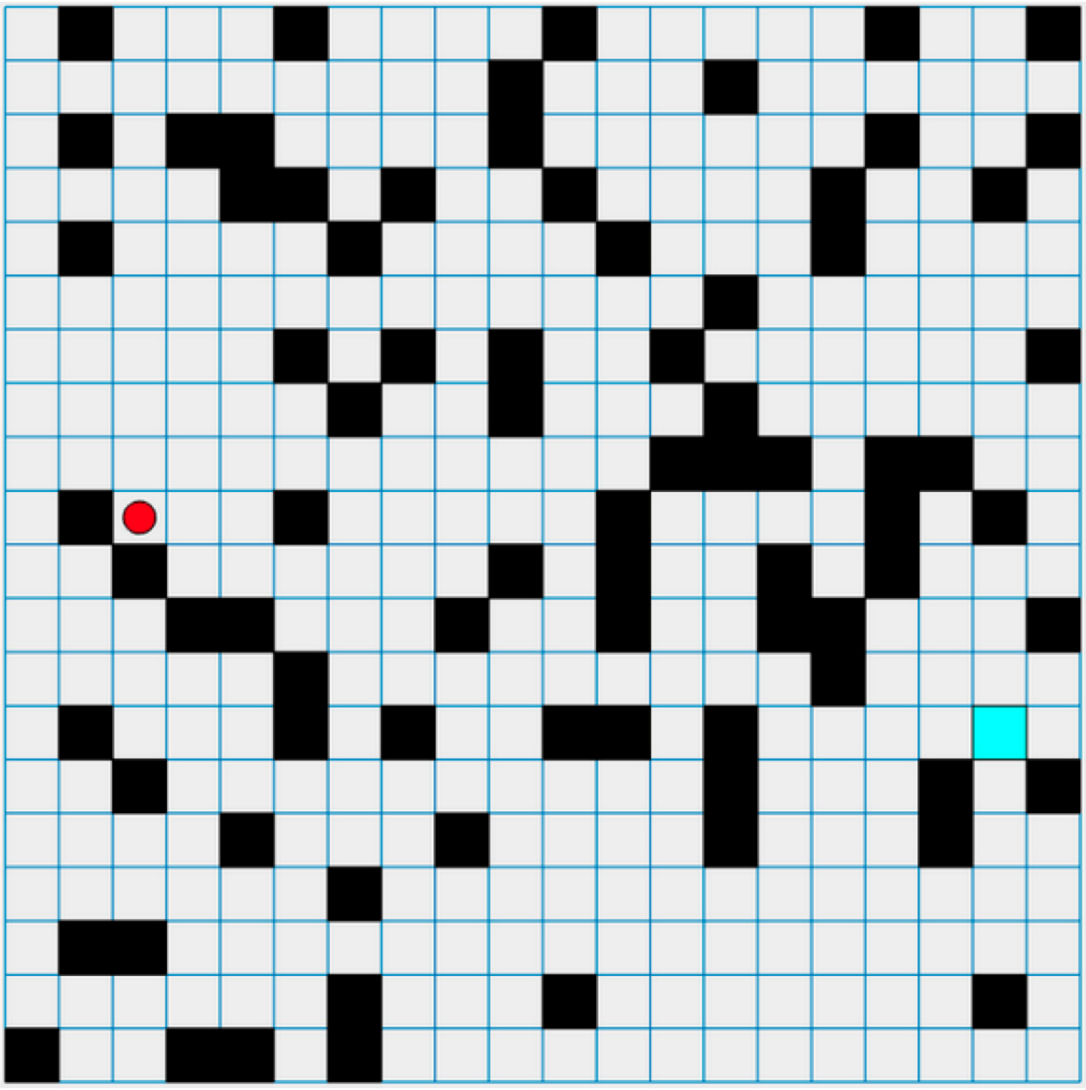

Figure A.17: $20 \times 20$ grid world task with start state (2, 10) and goal state (18, 6) (the bottom left corner is (0, 0)).

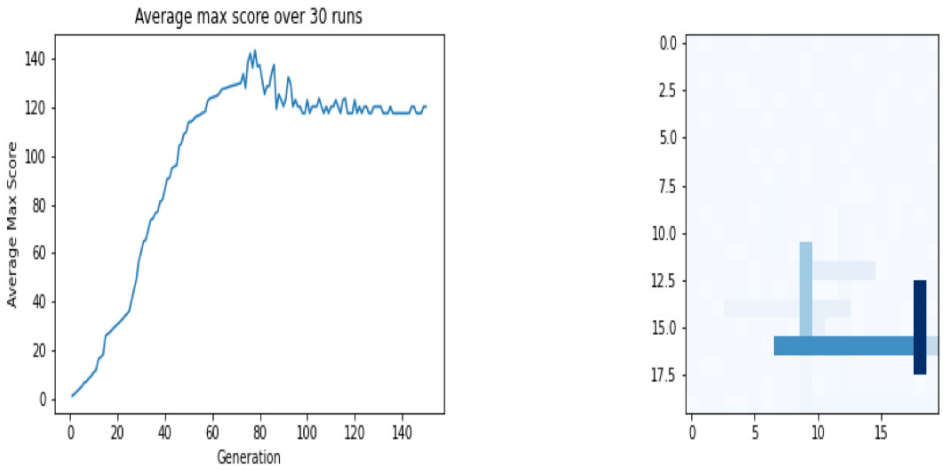


Figure A.18: QTRB results from Figure A.17. Shown from left to right: GP fitness curve and Q-value map.

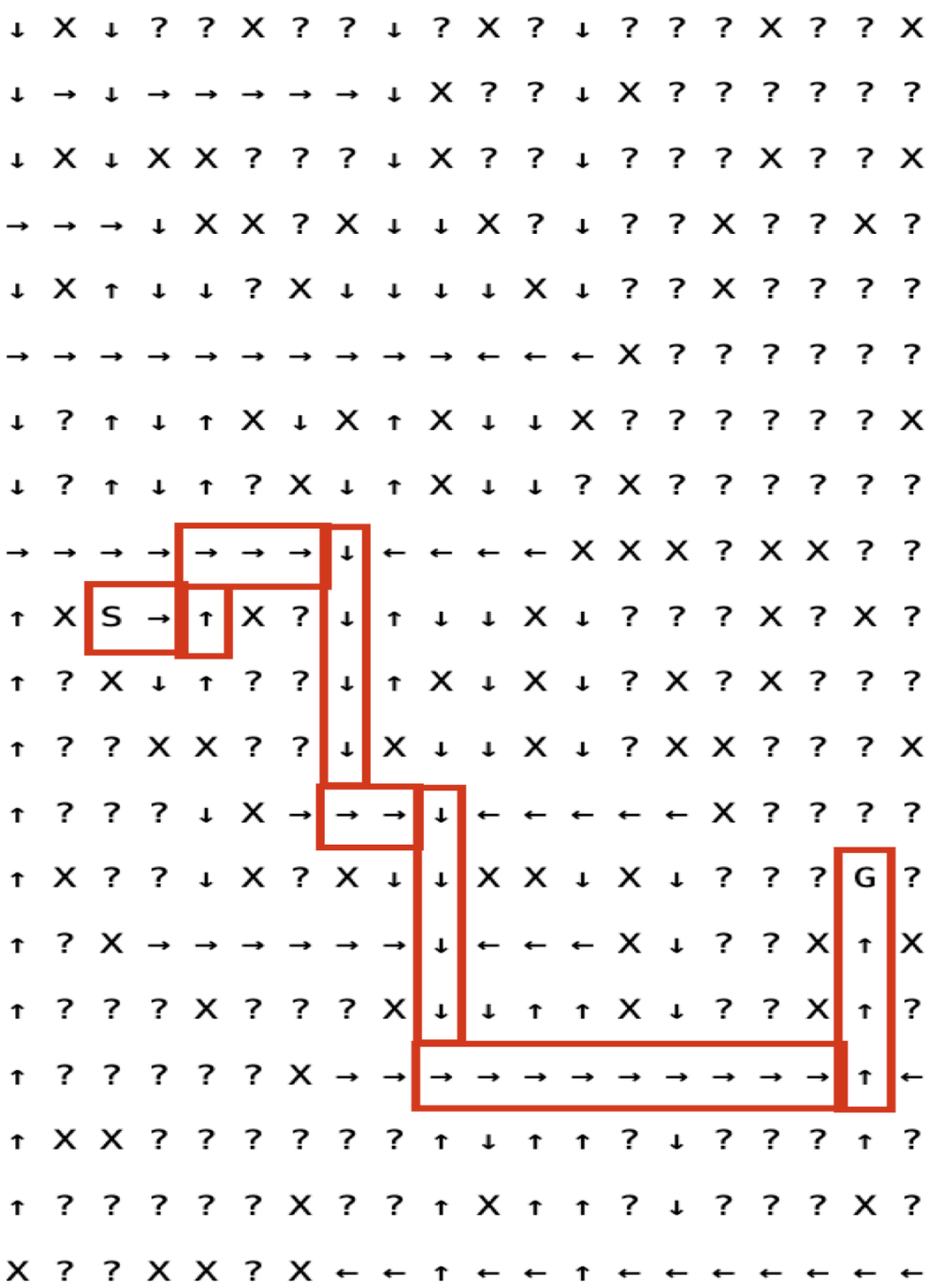


Figure A.19: Policy map for Figure A.17. The starting action was East.

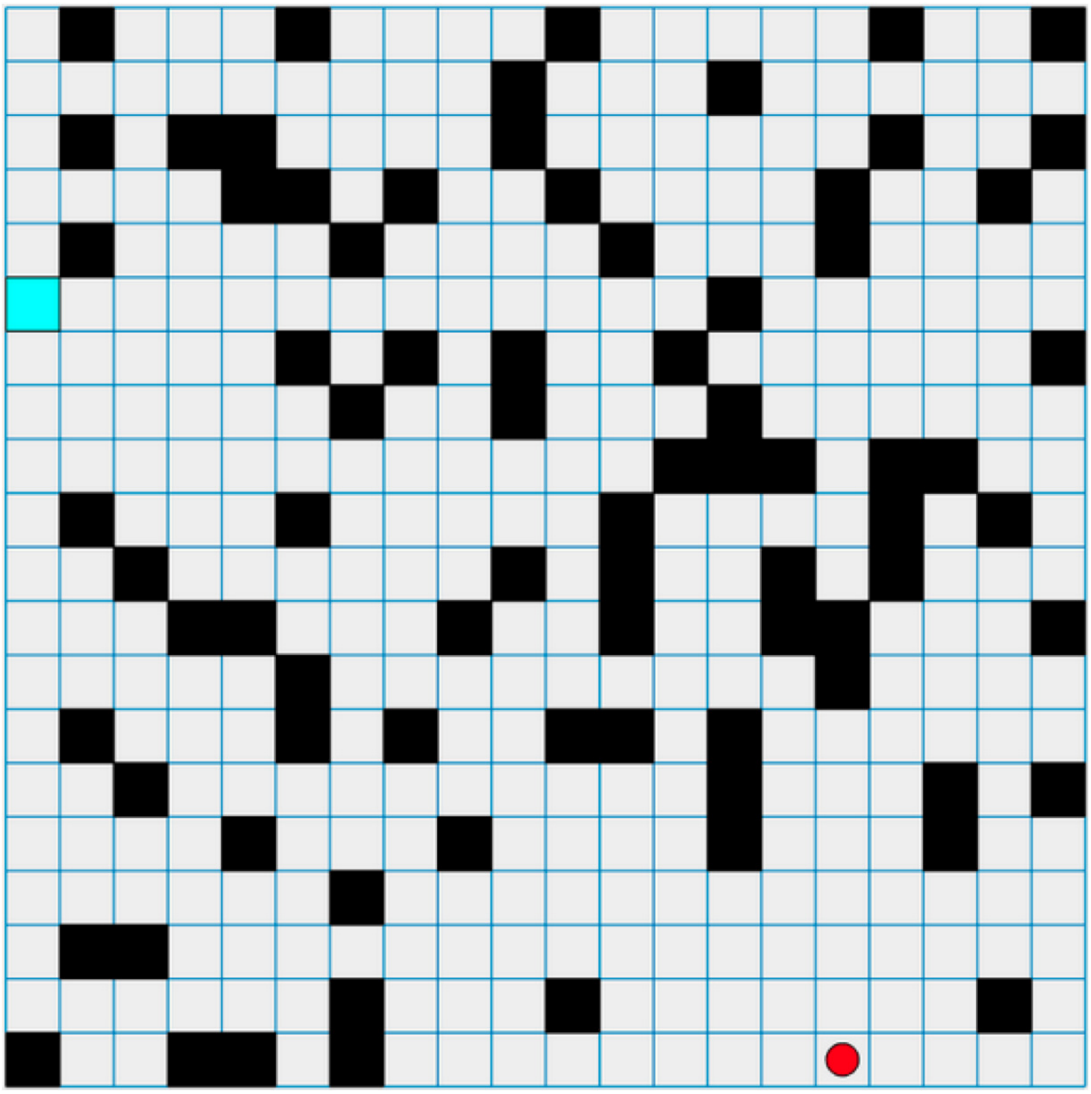

Figure A.20: $20 \times 20$ grid world task with start state (15, 0) and goal state (0, 14) (the bottom left corner is (0, 0)).

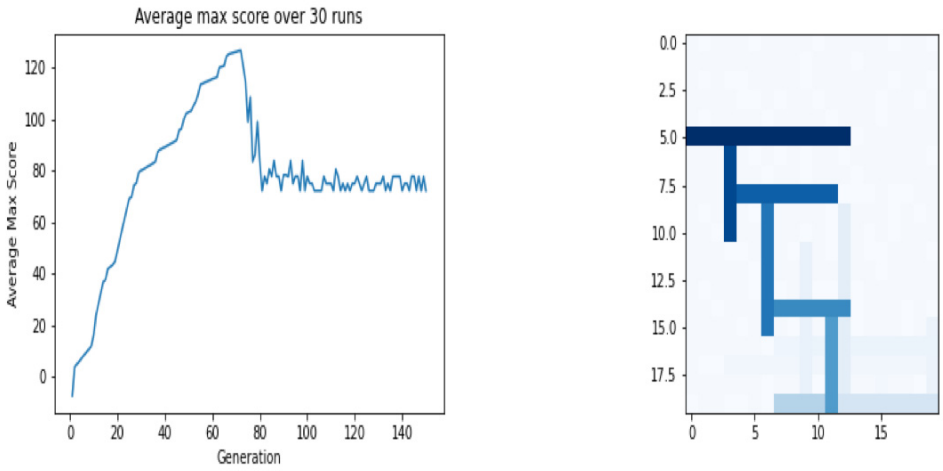


Figure A.21: QTRB results from Figure A.20. Shown from left to right: GP fitness curve and Q-value map.

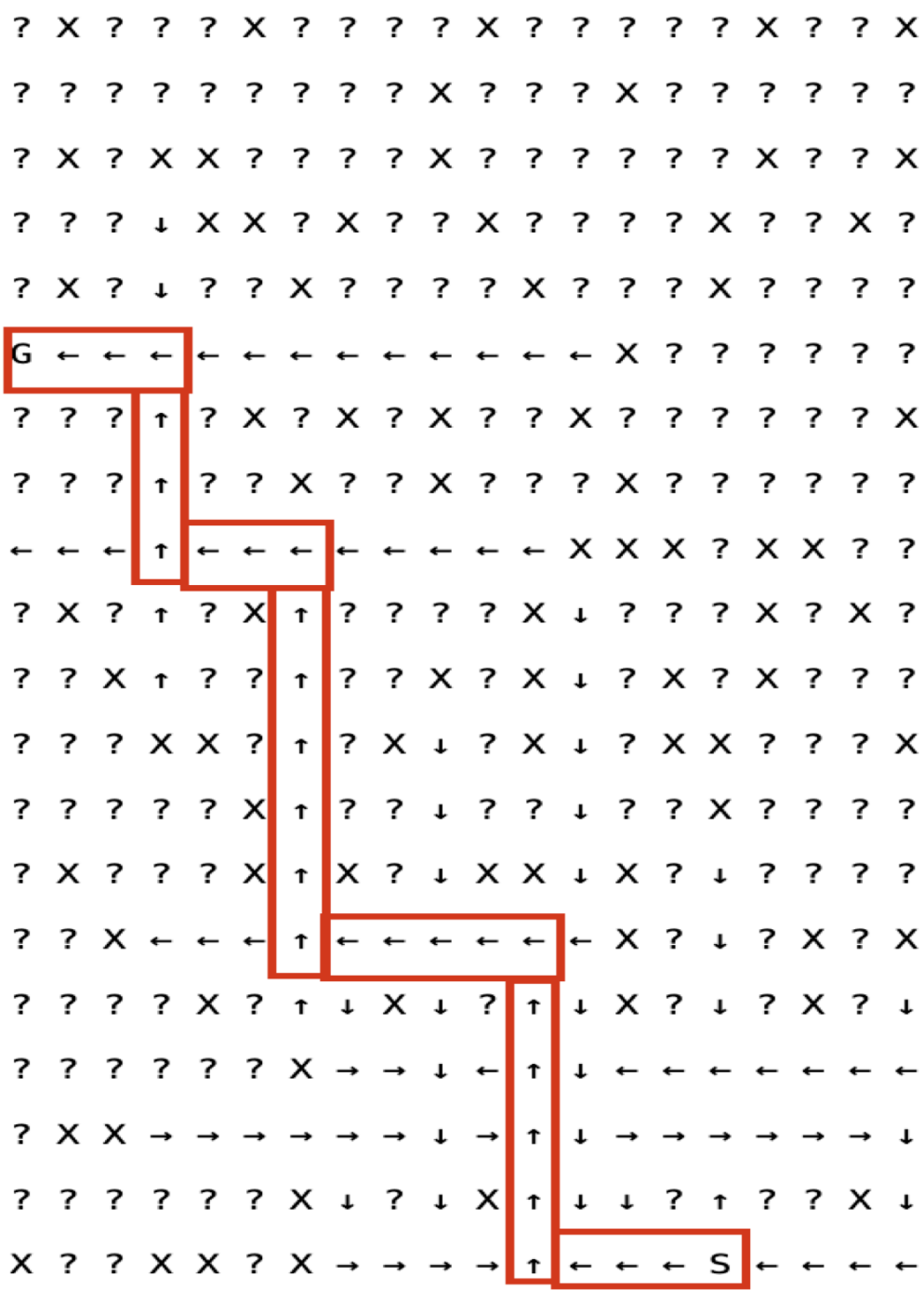


Figure A.22: Policy map for Figure A.20. The starting action was West.

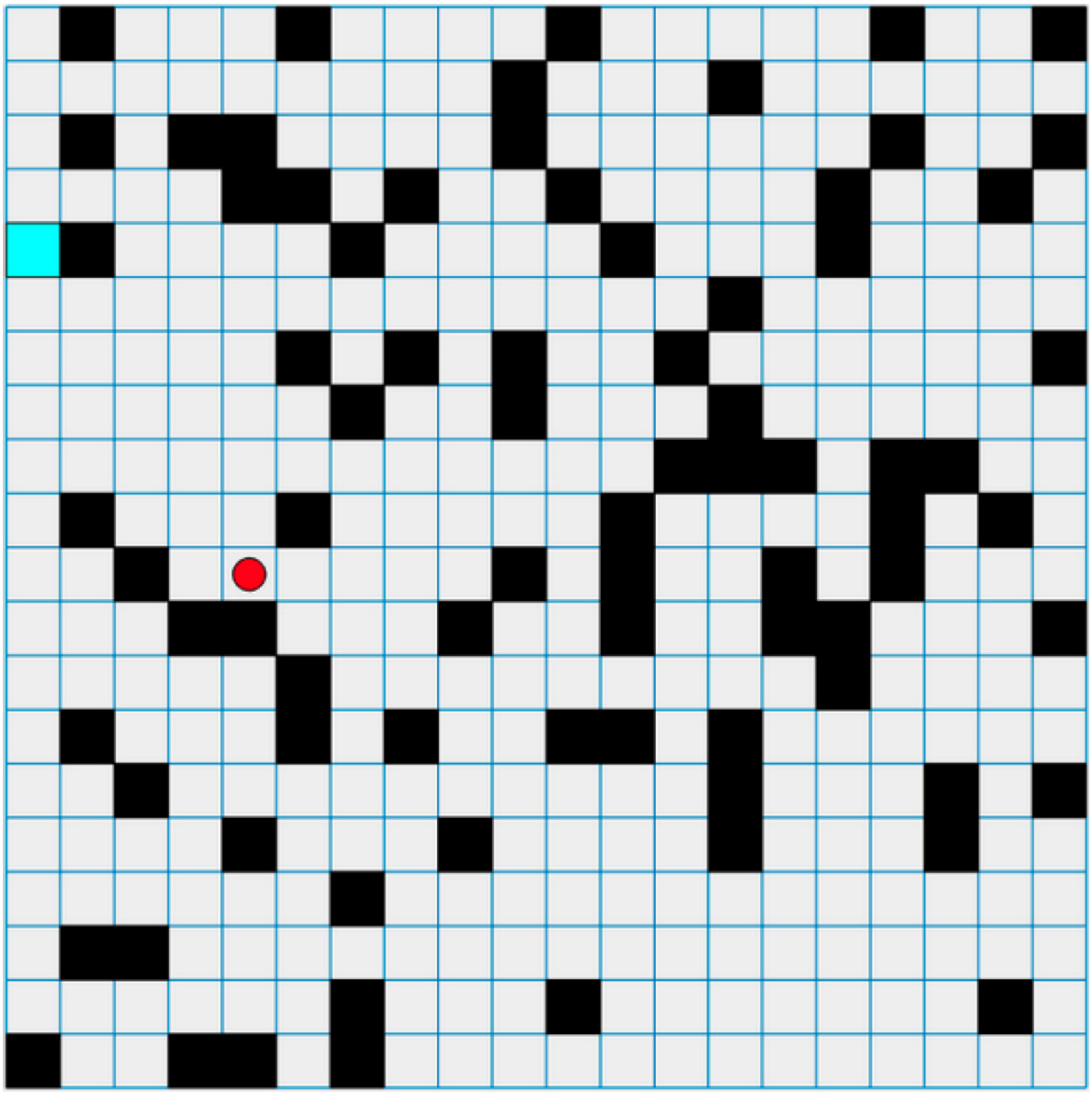

Figure A.23: $20 \times 20$ grid world task with start state (4, 9) and goal state (0, 15) (the bottom left corner is (0, 0)).

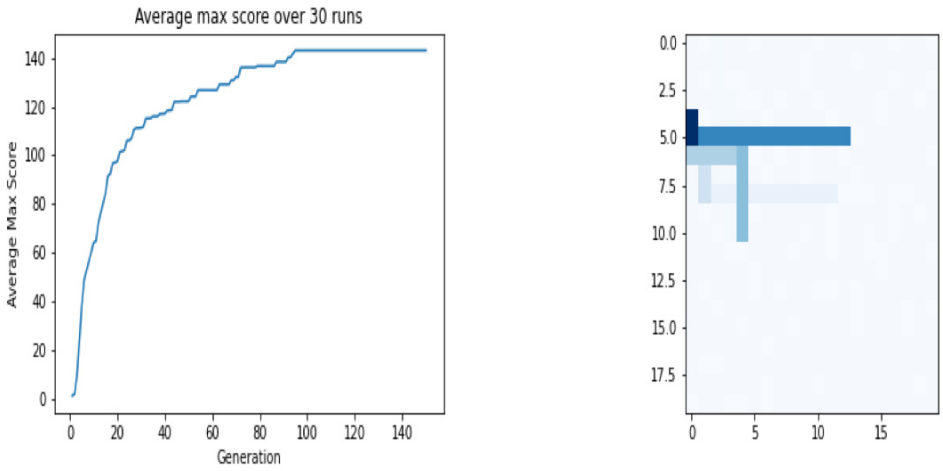


Figure A.24: QTRB results from Figure A.23. Shown from left to right: GP fitness curve and Q-value map.

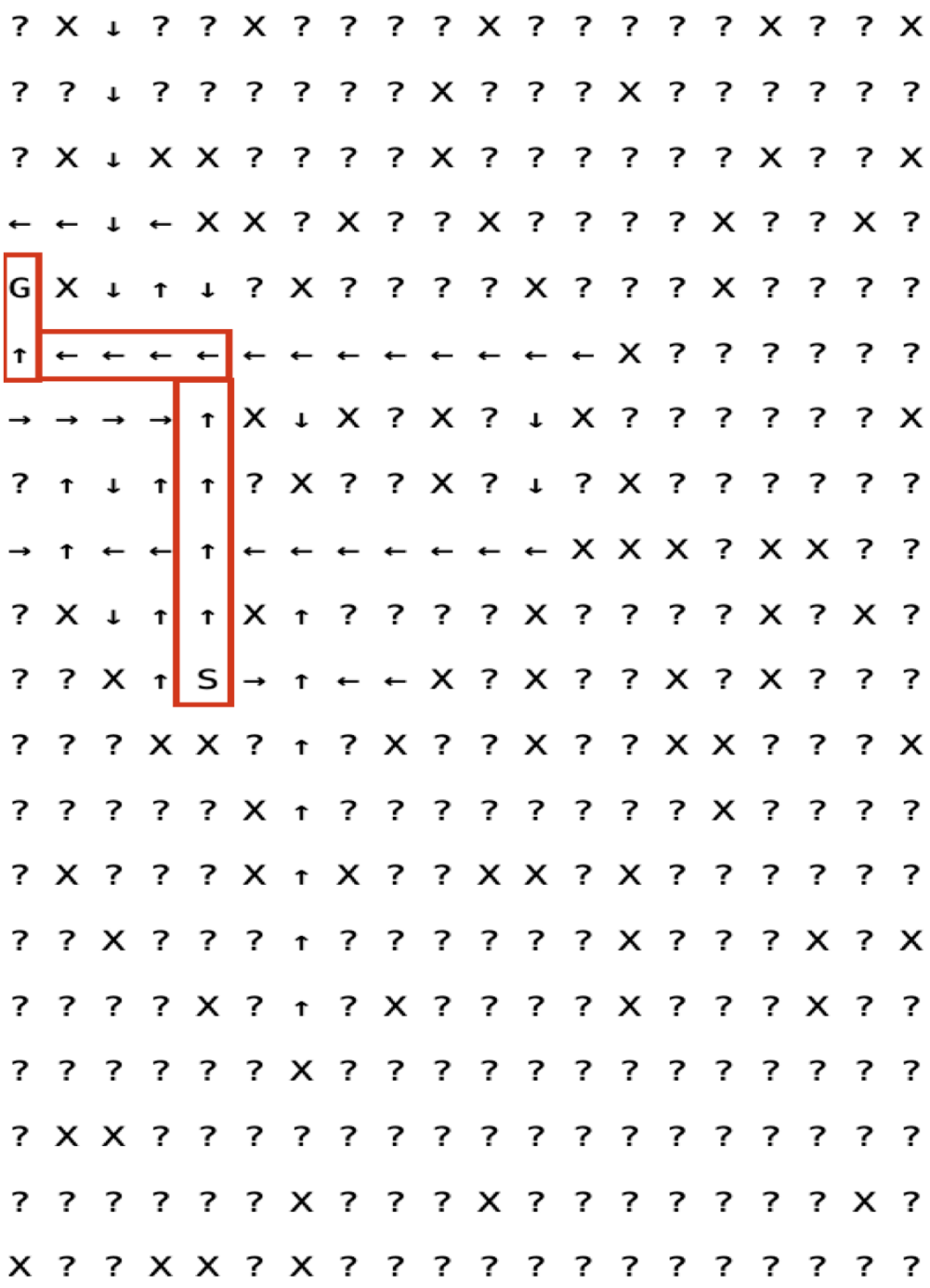


Figure A.25: Policy map for Figure A.23. The starting action was North.

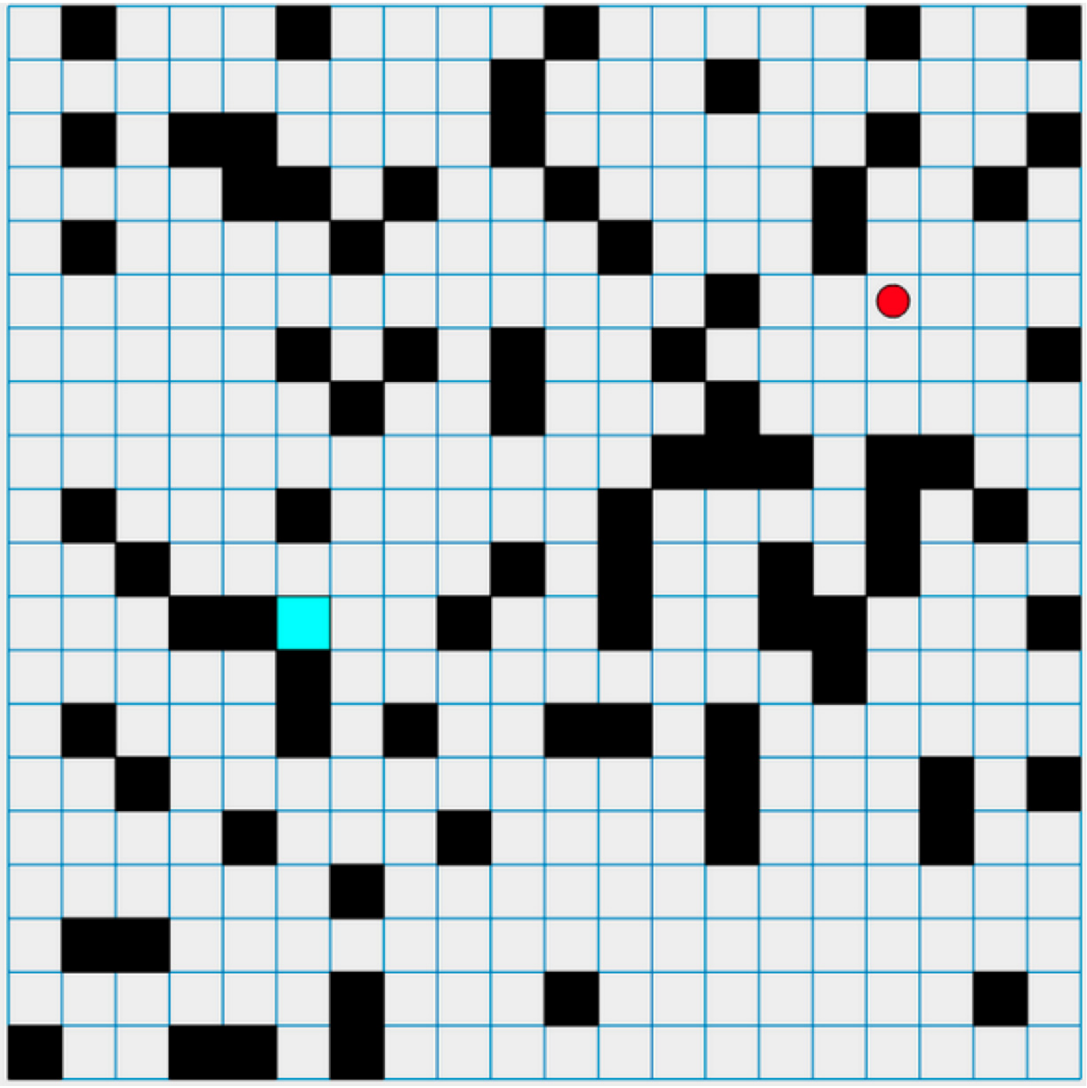

Figure A.26: $20 \times 20$ grid world task with start state (16, 14) and goal state (5, 8) (the bottom left corner is (0, 0)).

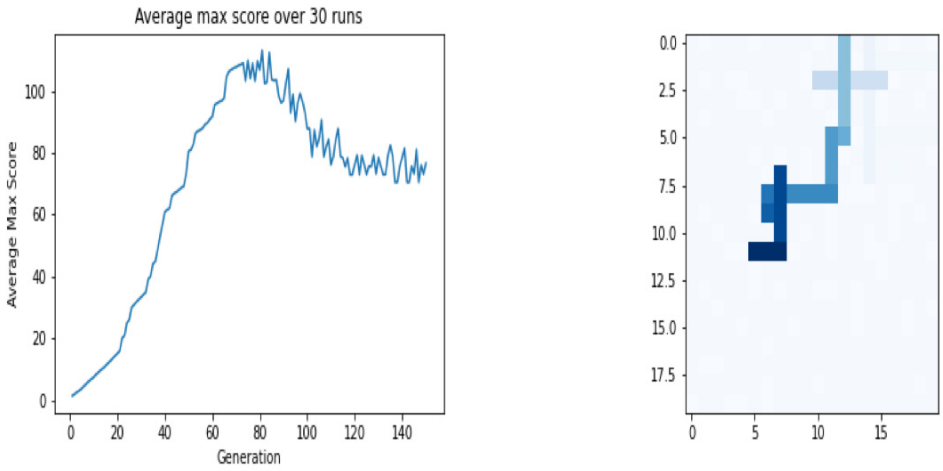


Figure A.27: QTRB results from Figure A.26. Shown from left to right: GP fitness curve and Q-value map.

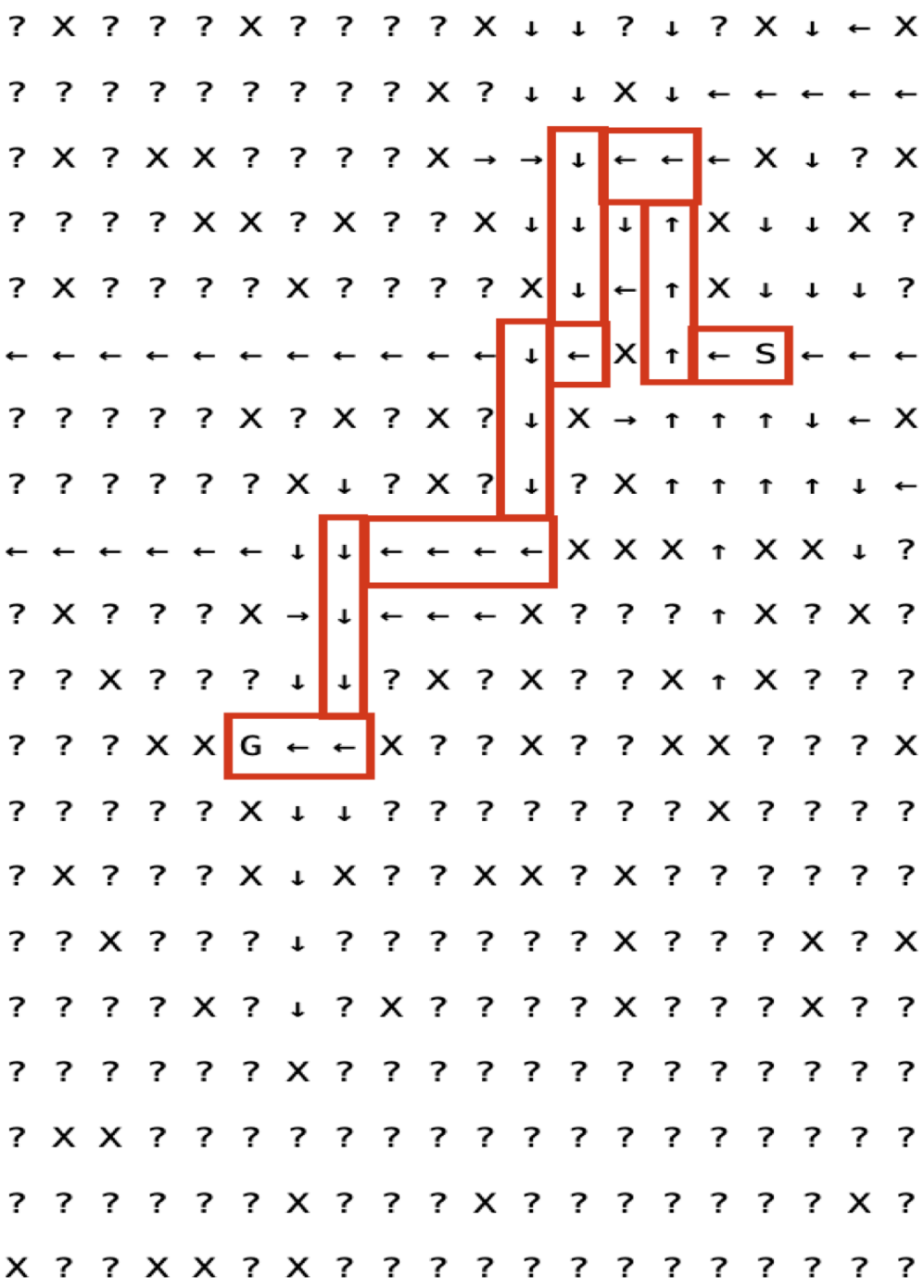


Figure A.28: Policy map for Figure A.26. The starting action was West.

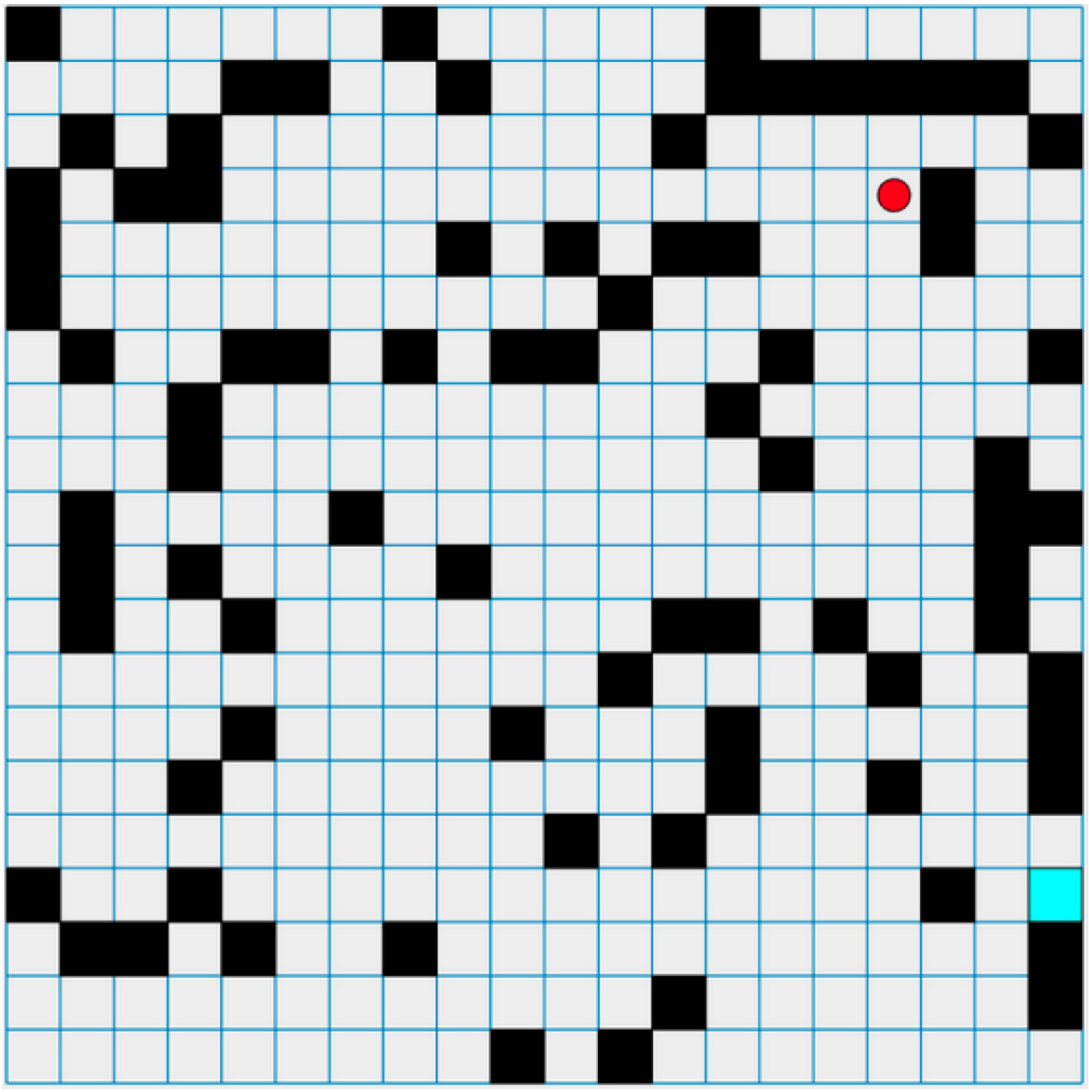

Figure A.29: $20 \times 20$ grid world task with start state (16, 16) and goal state (19, 3) (the bottom left corner is (0, 0)).

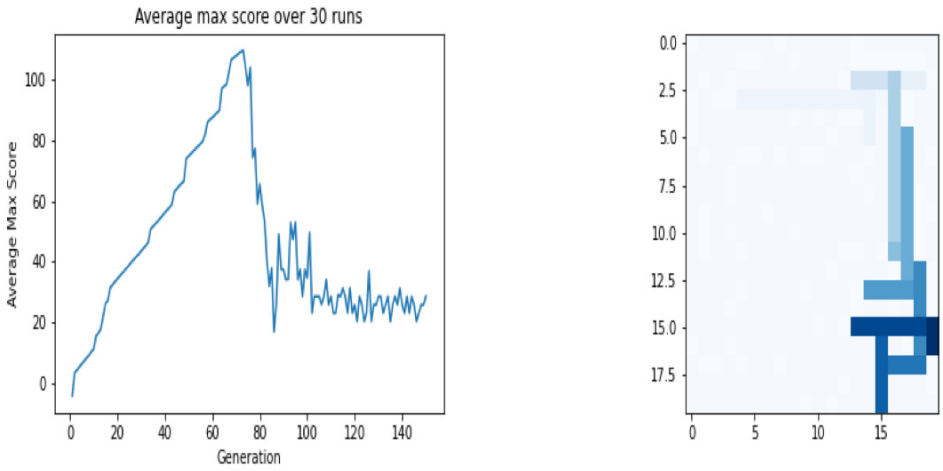


Figure A.30: QTRB results from Figure A.29. Shown from left to right: GP fitness curve and Q-value map.

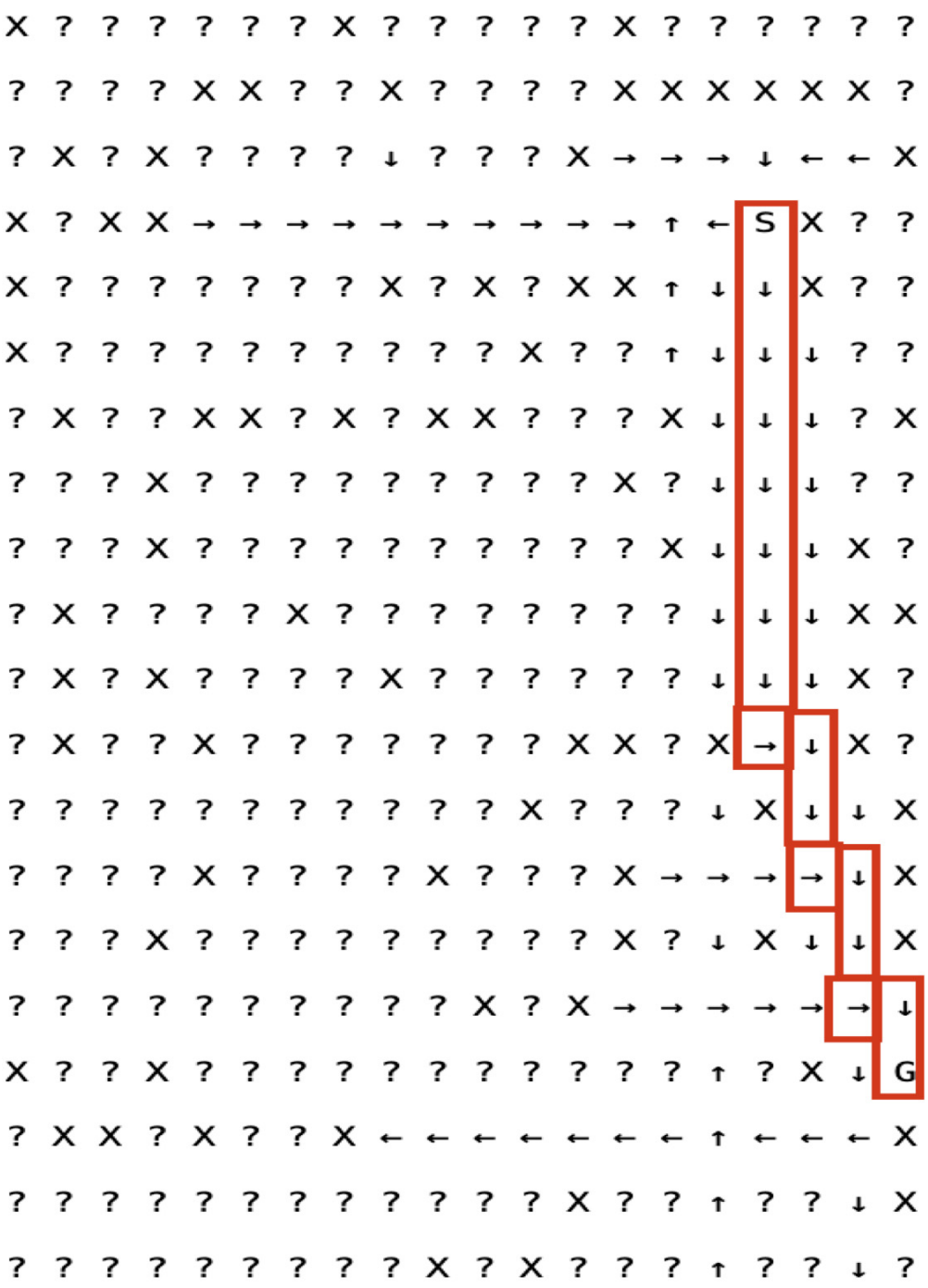


Figure A.31: Policy map for Figure A.29. The starting action was South.

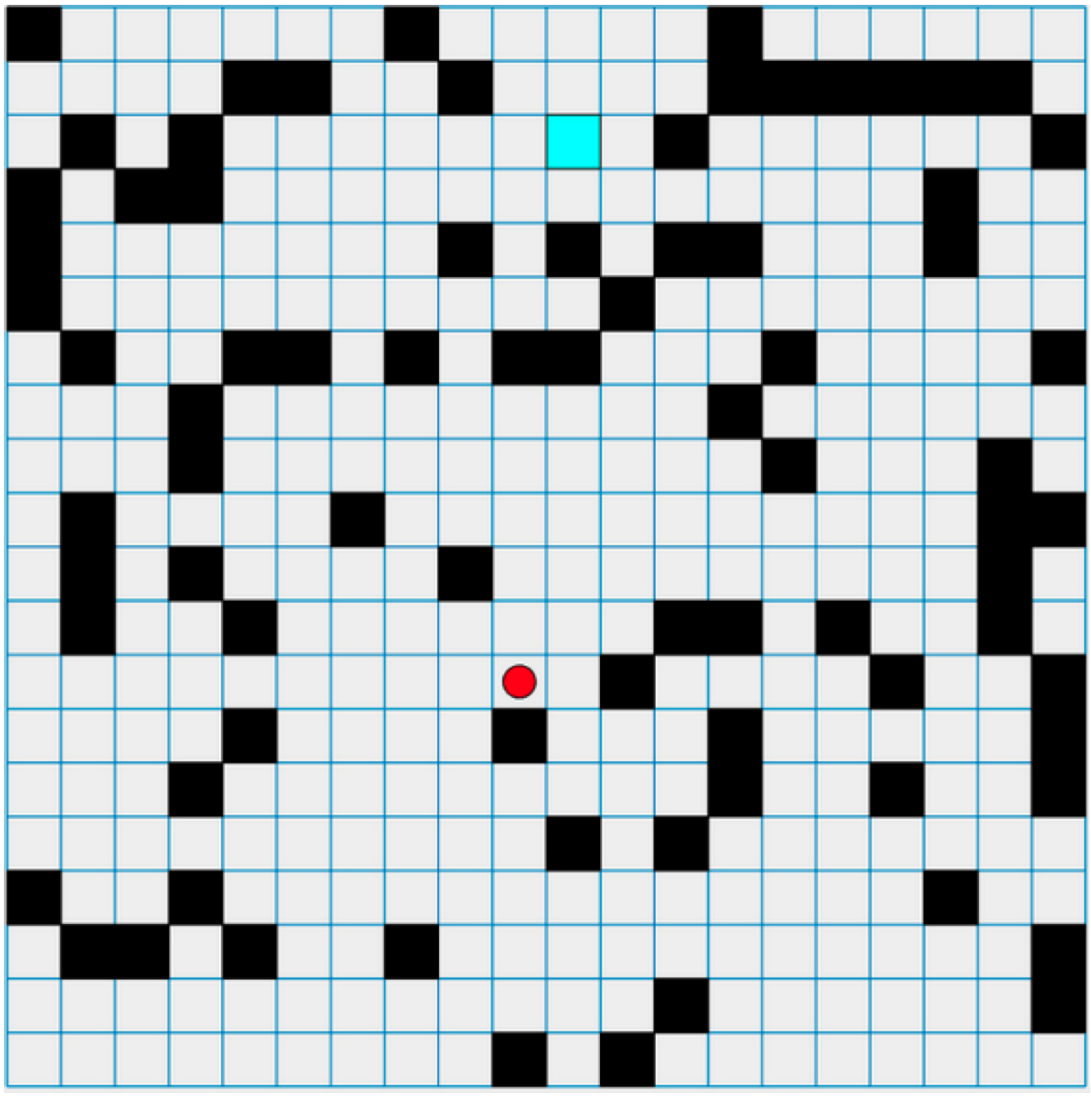

Figure A.32: $20 \times 20$ grid world task with start state (9, 7) and goal state (10, 17) (the bottom left corner is (0, 0)).

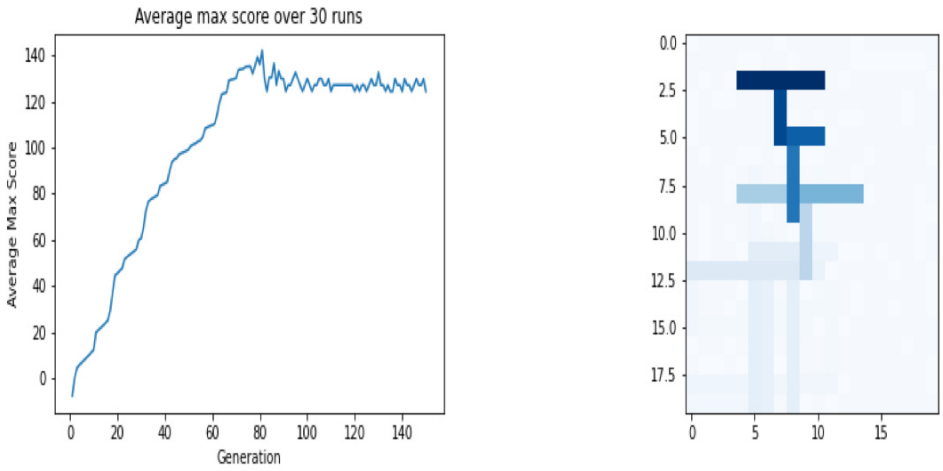


Figure A.33: QTRB results from Figure A.32. Shown from left to right: GP fitness curve and Q-value map.

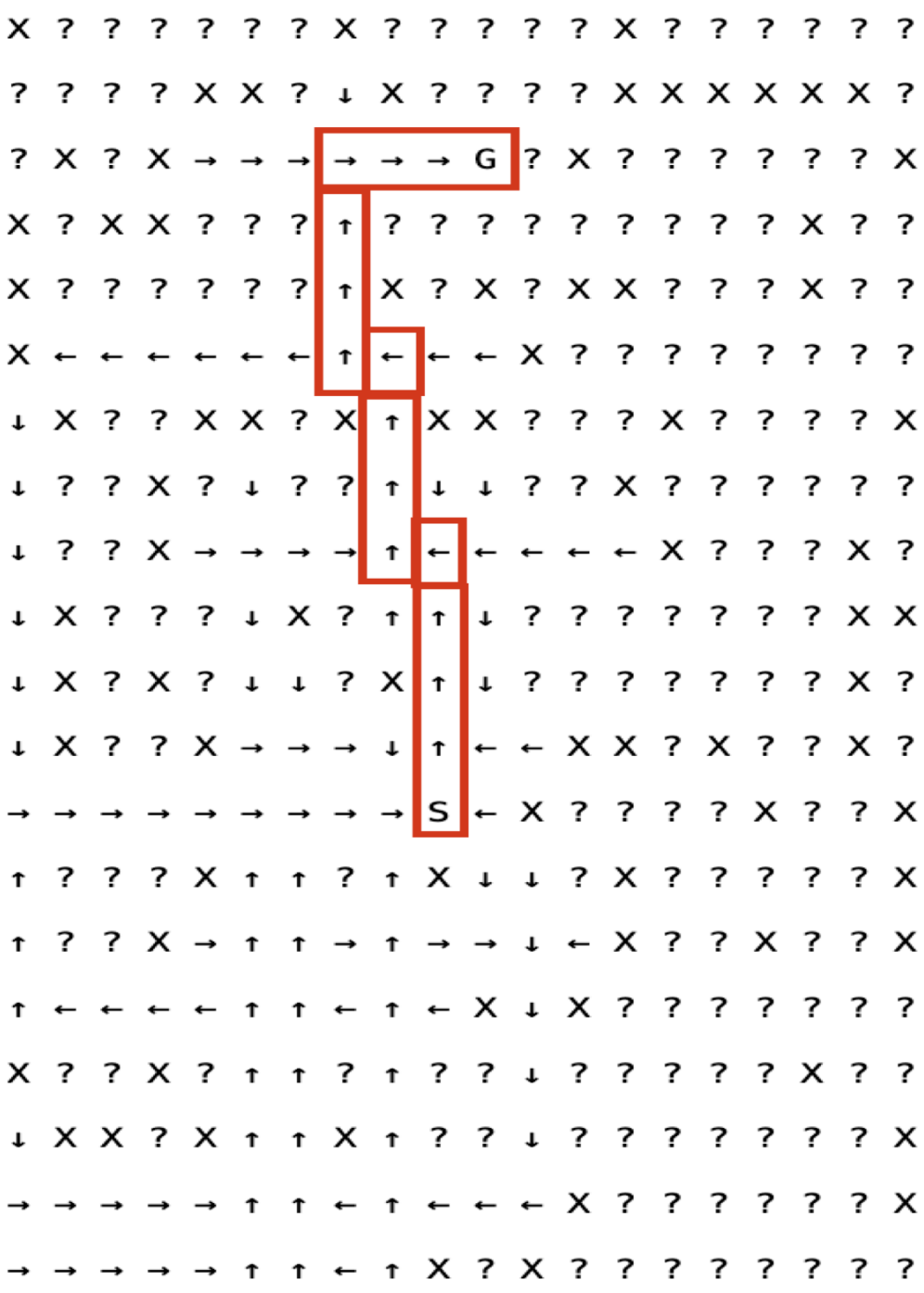


Figure A.34: Policy map for Figure A.32. The starting action was North.

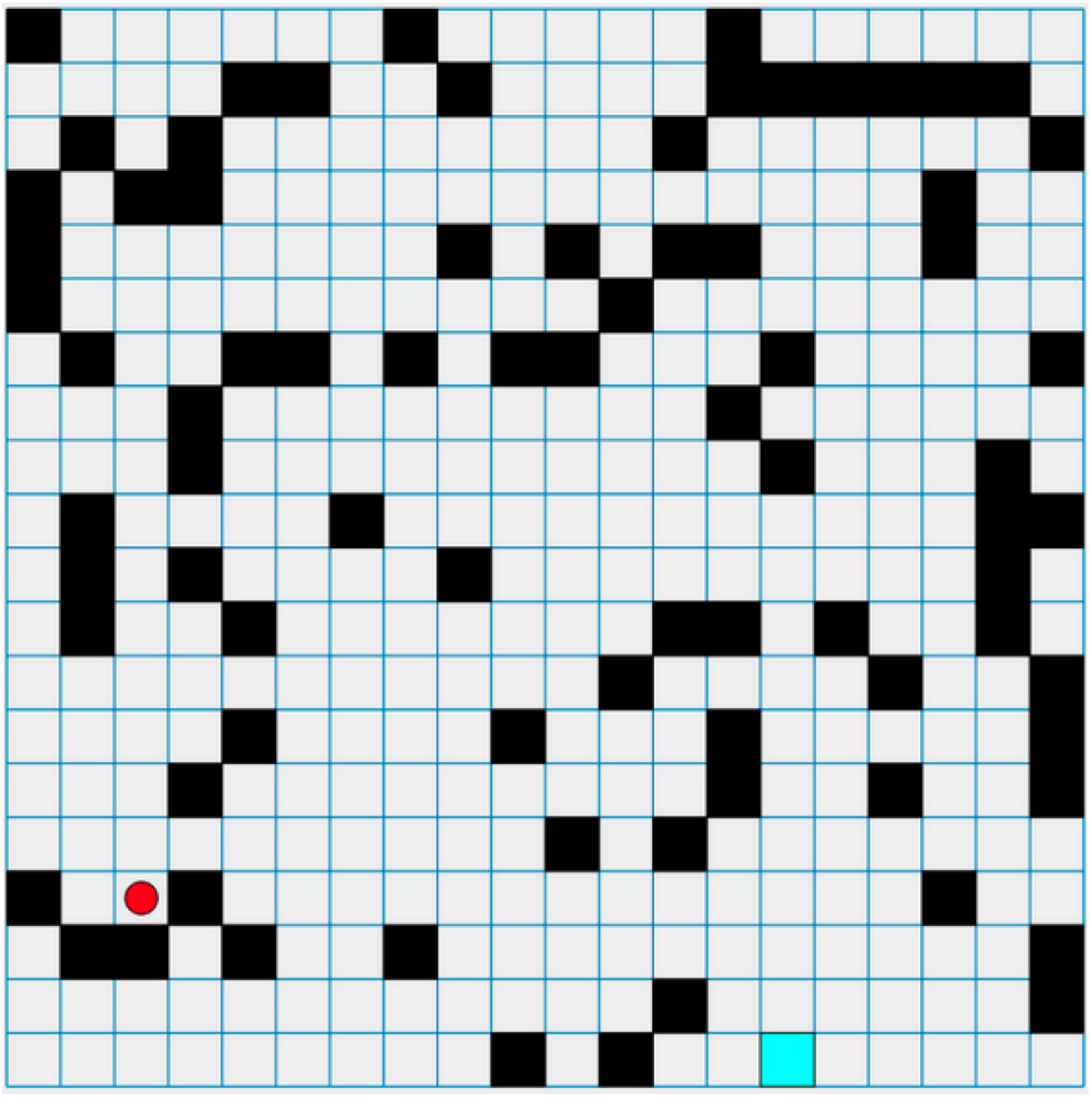

Figure A.35: $20 \times 20$ grid world task with start state (2, 3) and goal state (14, 0) (the bottom left corner is (0, 0)).

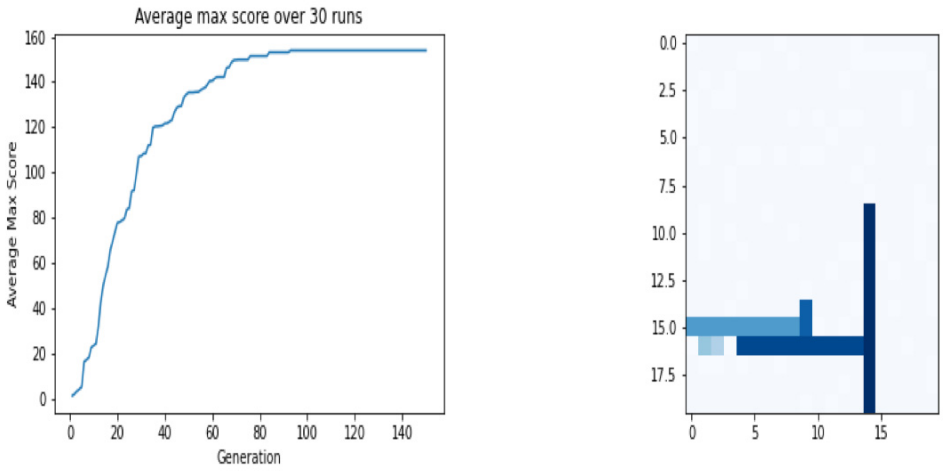


Figure A.36: QTRB results from Figure A.35. Shown from left to right: GP fitness curve and Q-value map.

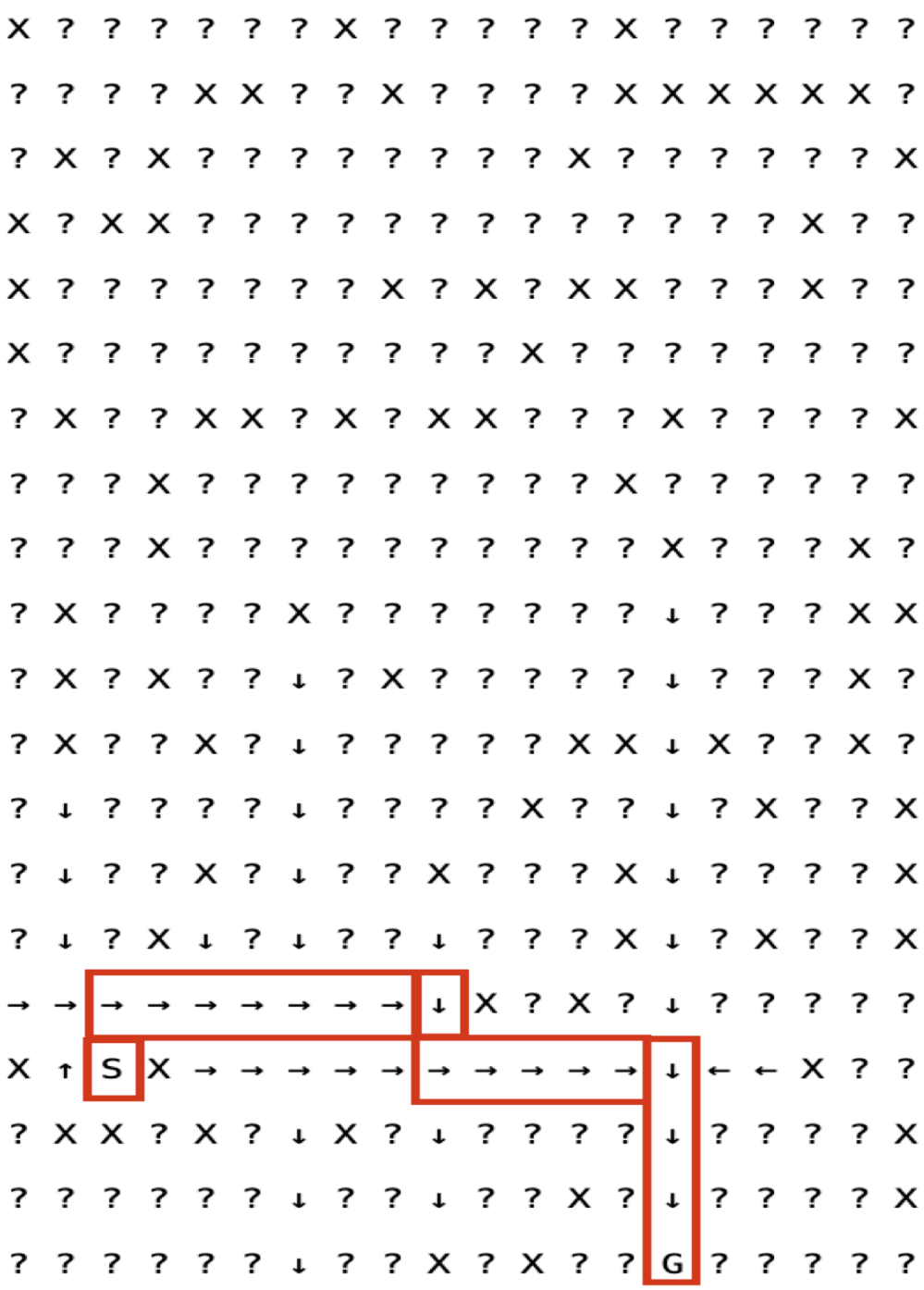


Figure A.37: Policy map for Figure A.35. The starting action was North.

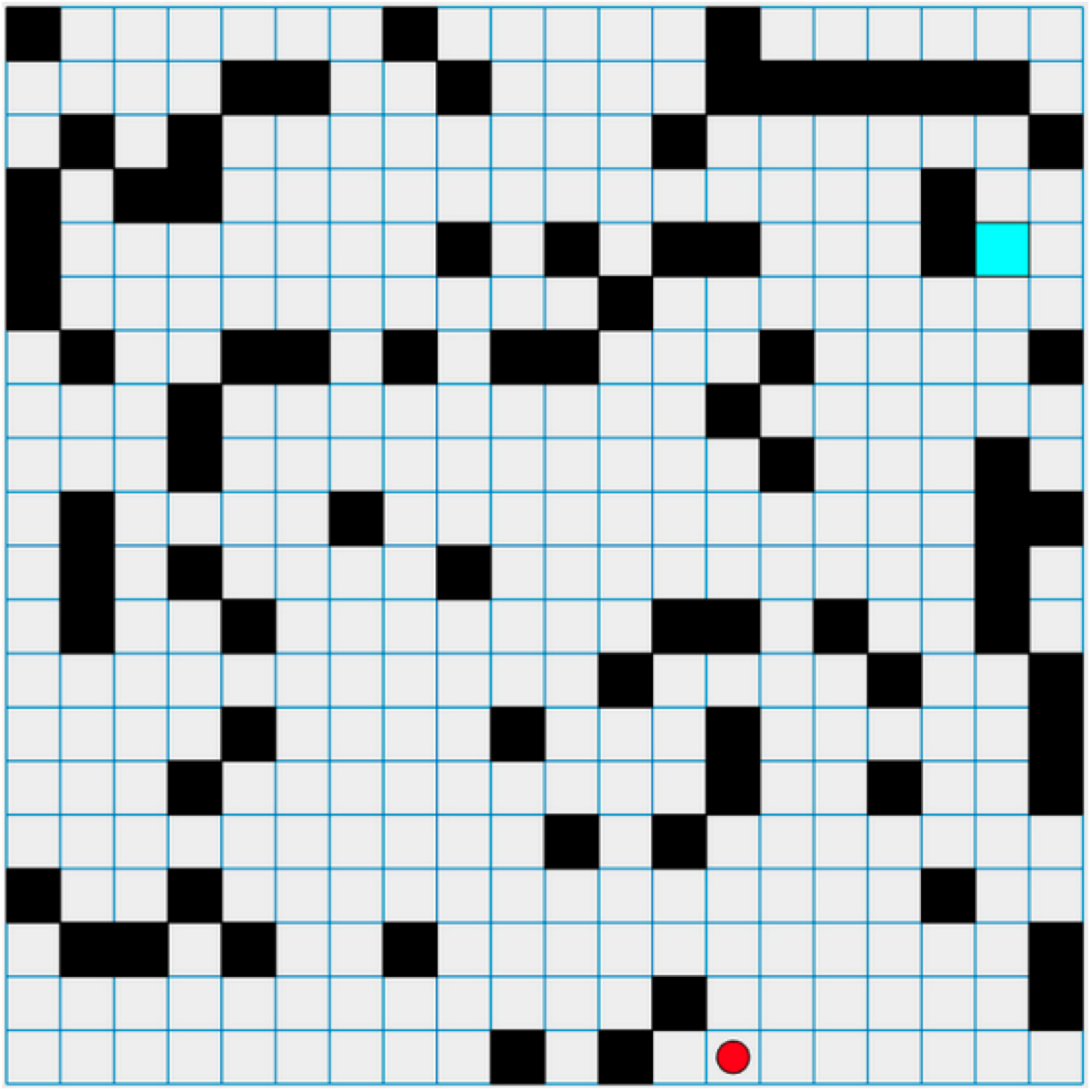

Figure A.38: $20 \times 20$ grid world task with start state (13, 0) and goal state (18, 15) (the bottom left corner is (0, 0)).

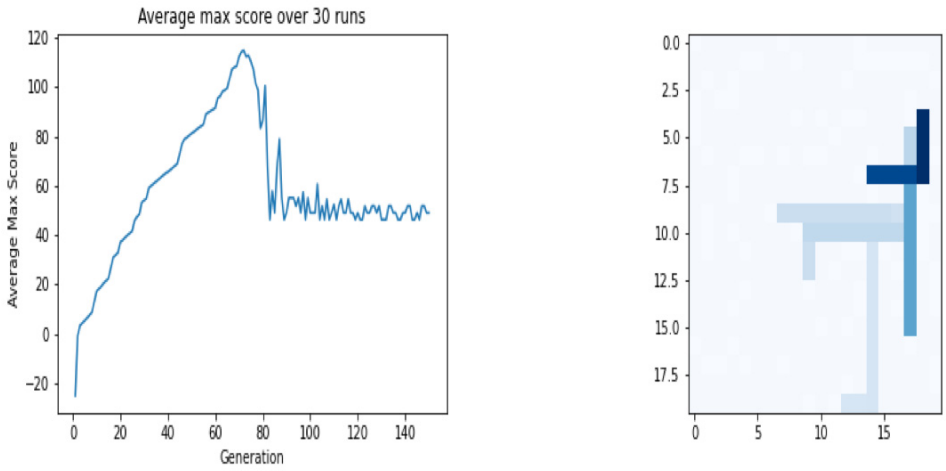


Figure A.39: QTRB results from Figure A.38. Shown from left to right: GP fitness curve and Q-value map.

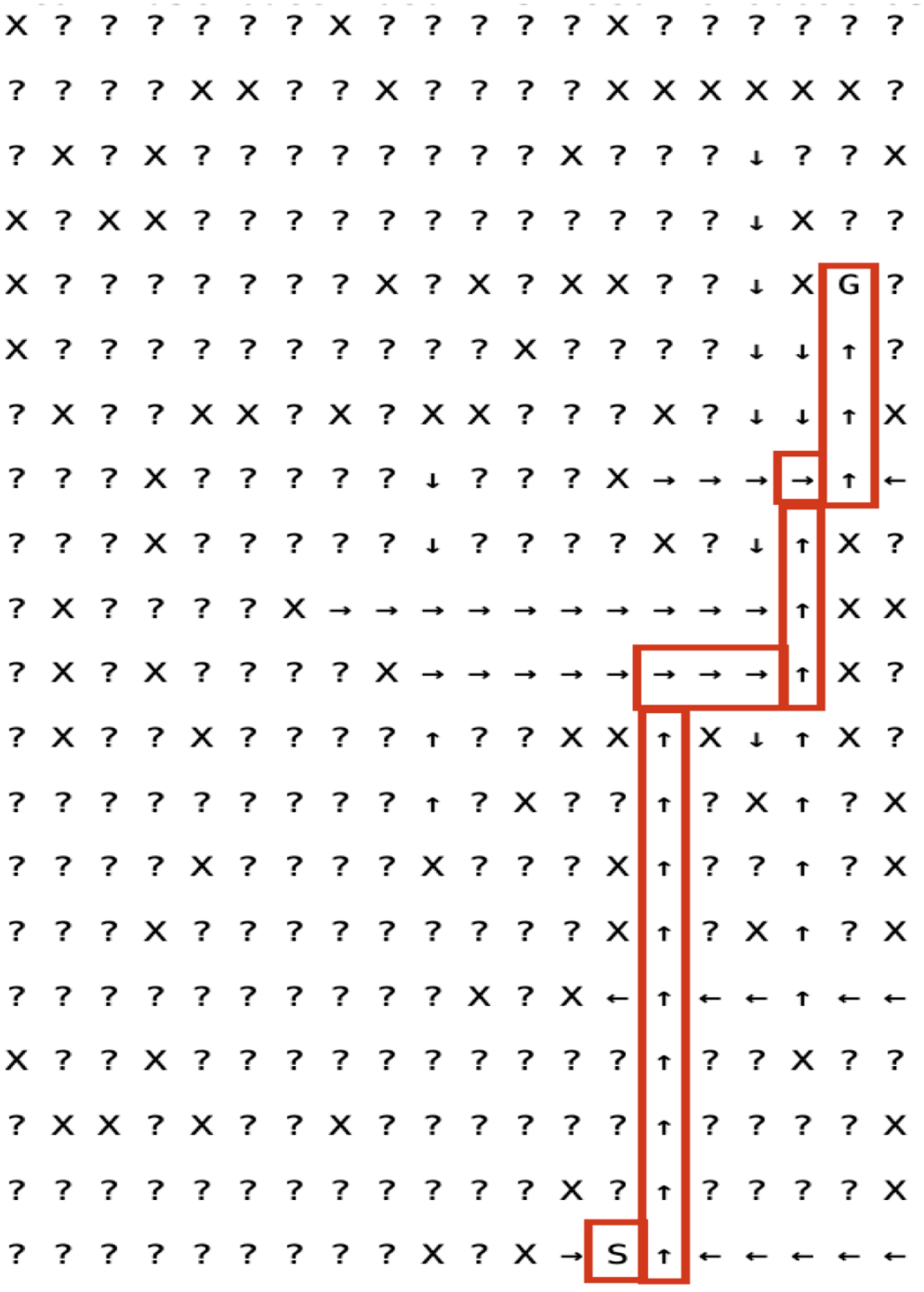


Figure A.40: Policy map for Figure A.38. The starting action was East.

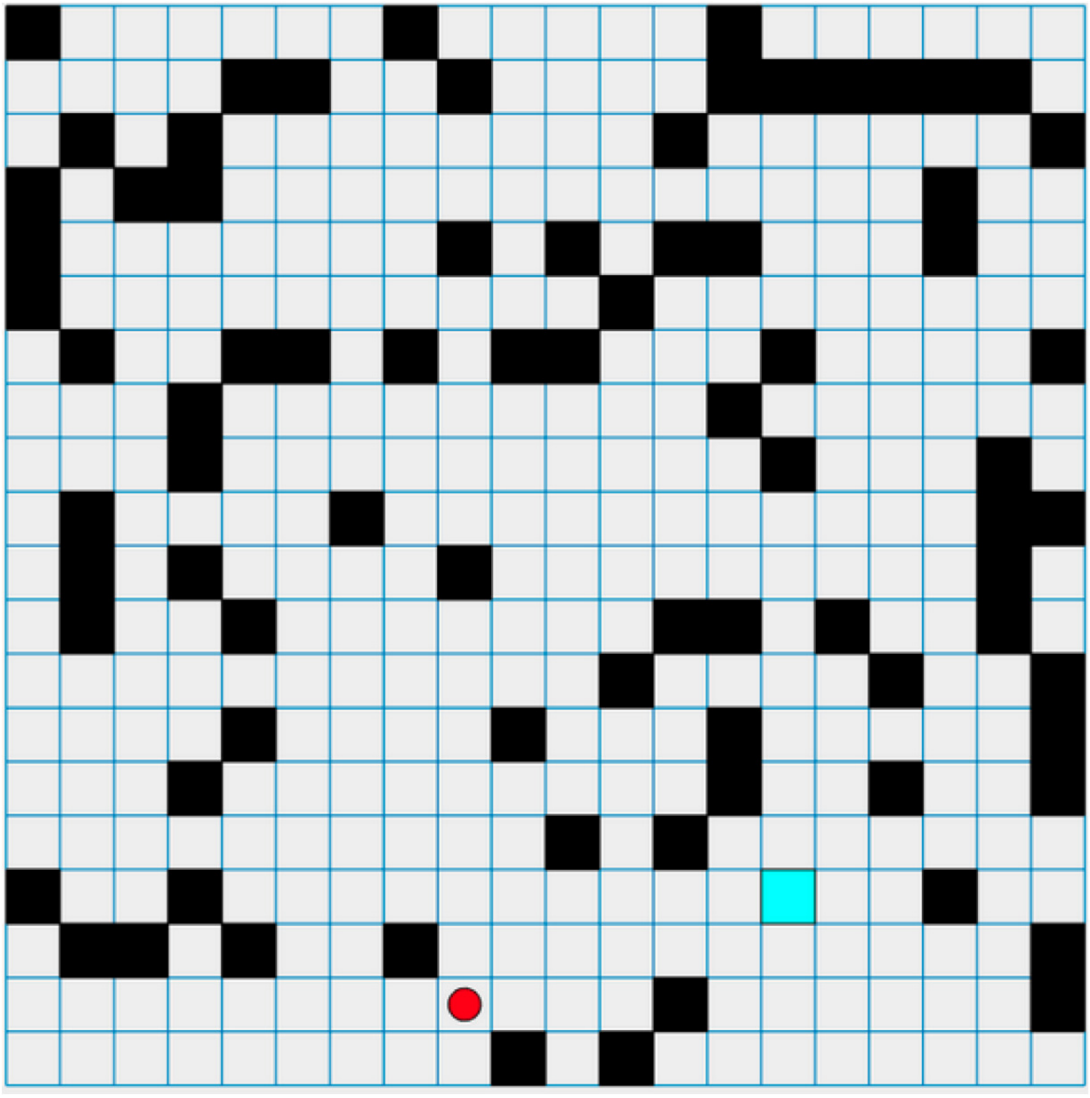

Figure A.41: $20 \times 20$ grid world task with start state (8, 1) and goal state (14, 3) (the bottom left corner is (0, 0)).

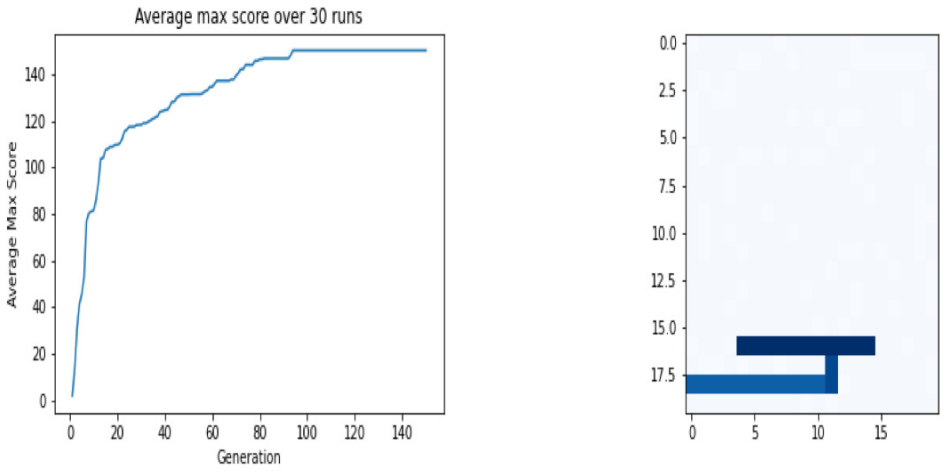


Figure A.42: QTRB results from Figure A.41. Shown from left to right: GP fitness curve and Q-value map.

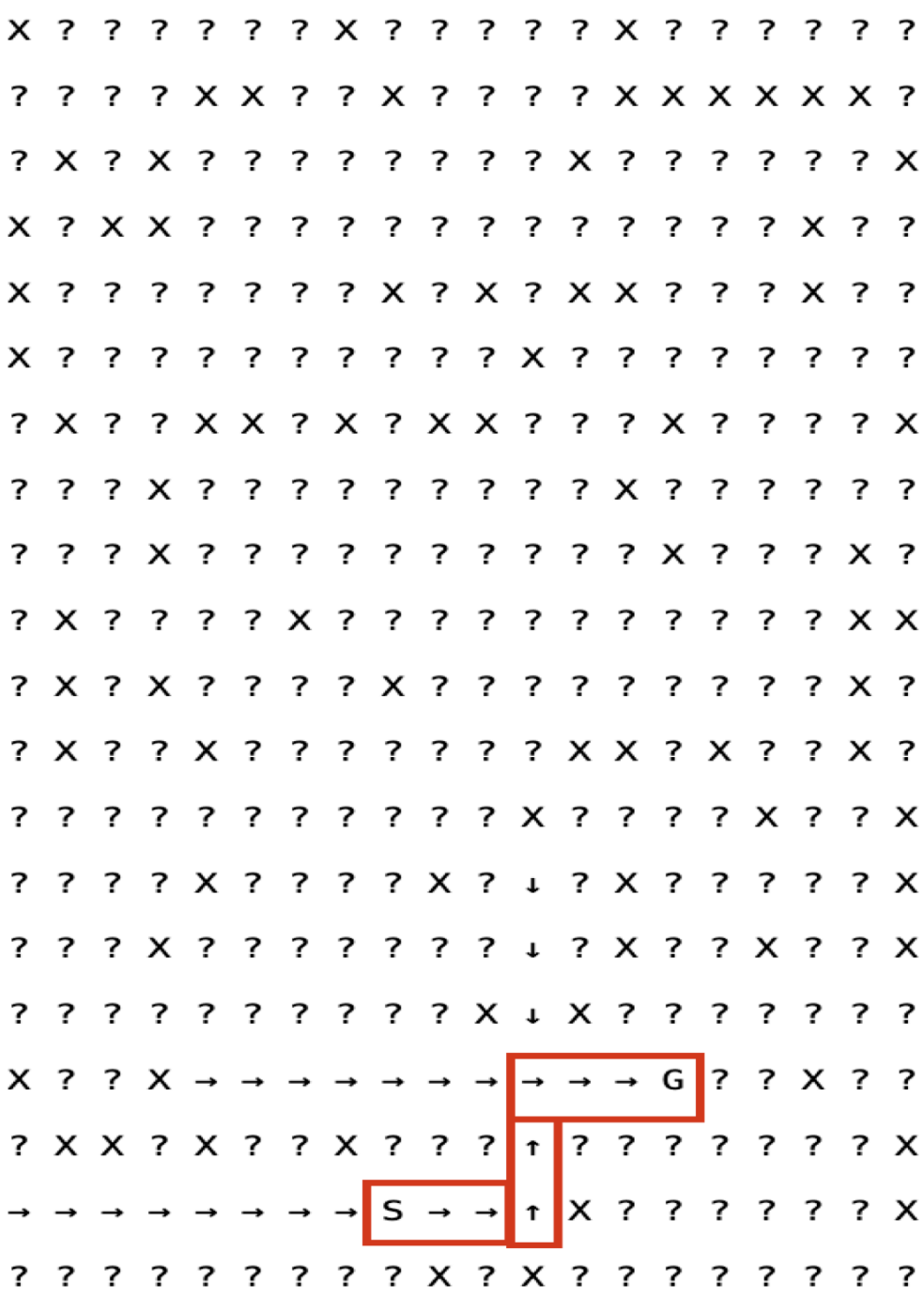


Figure A.43: Policy map for Figure A.41. The starting action was East.

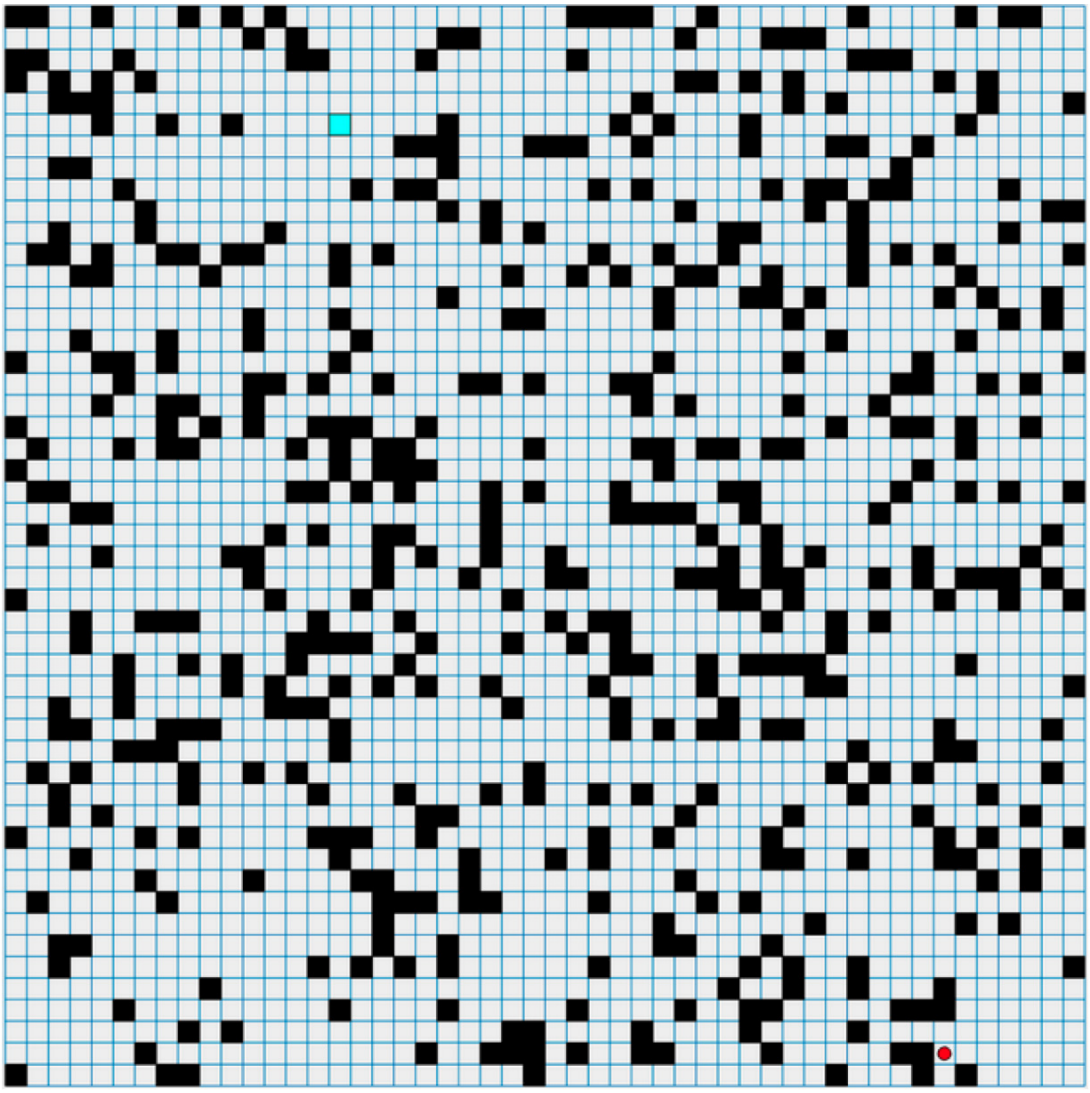

Figure A.44: $50 \times 50$ grid world task with start state (43, 1) and goal state (15, 44) (the bottom left corner is (0, 0)). This task is in reference to the results shown in the Results Chapter, and is shown in Figure 4.3 (right).

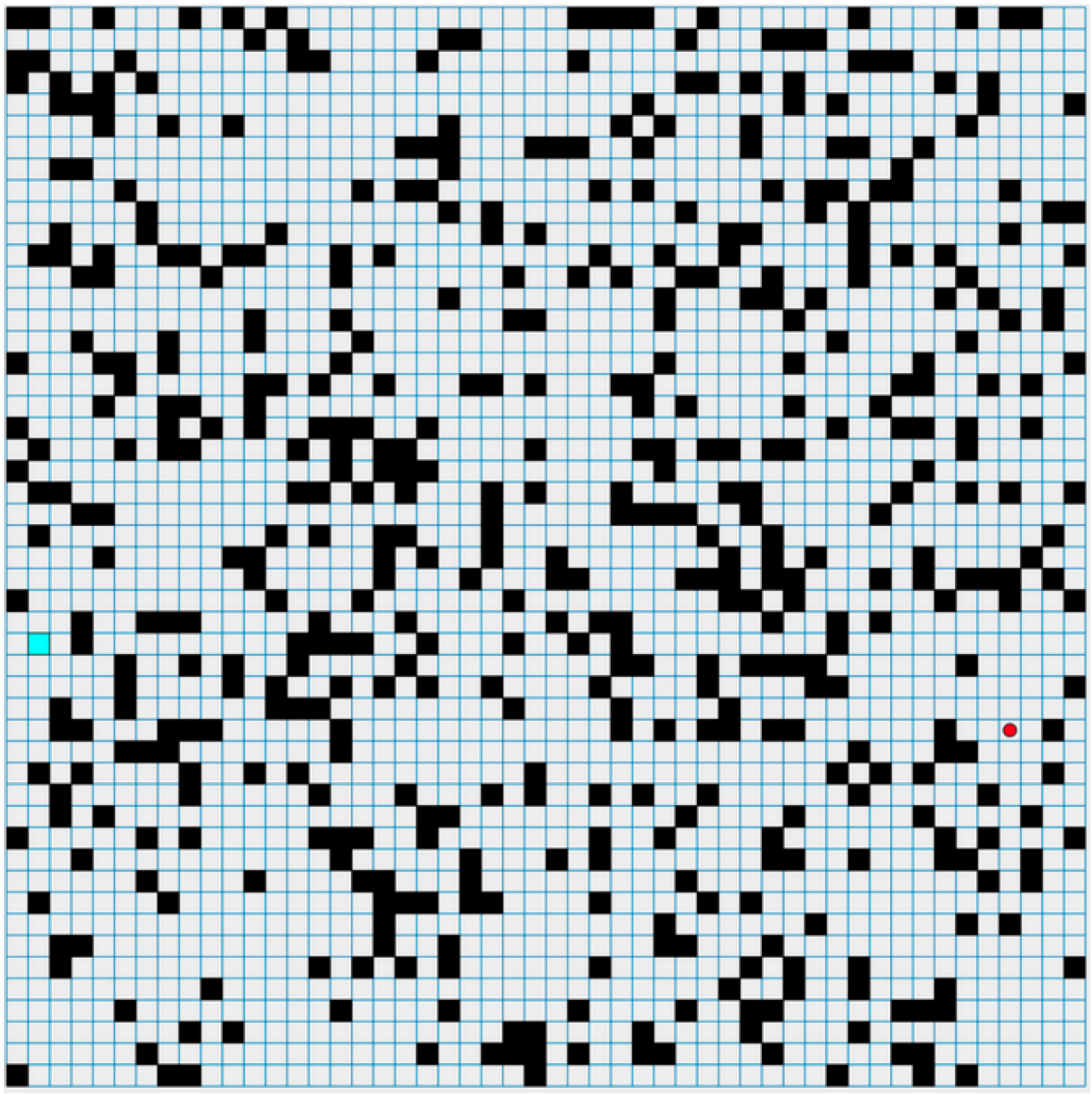

Figure A.45: $50 \times 50$ grid world task with start state (46, 16) and goal state (1, 20) (the bottom left corner is (0, 0)).

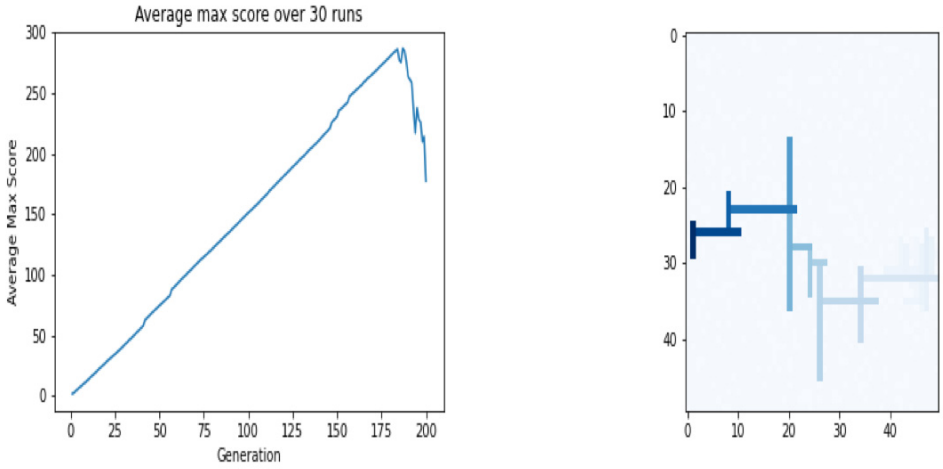


Figure A.46: QTRB results from Figure A.45. Shown from left to right: GP fitness curve and Q-value map.

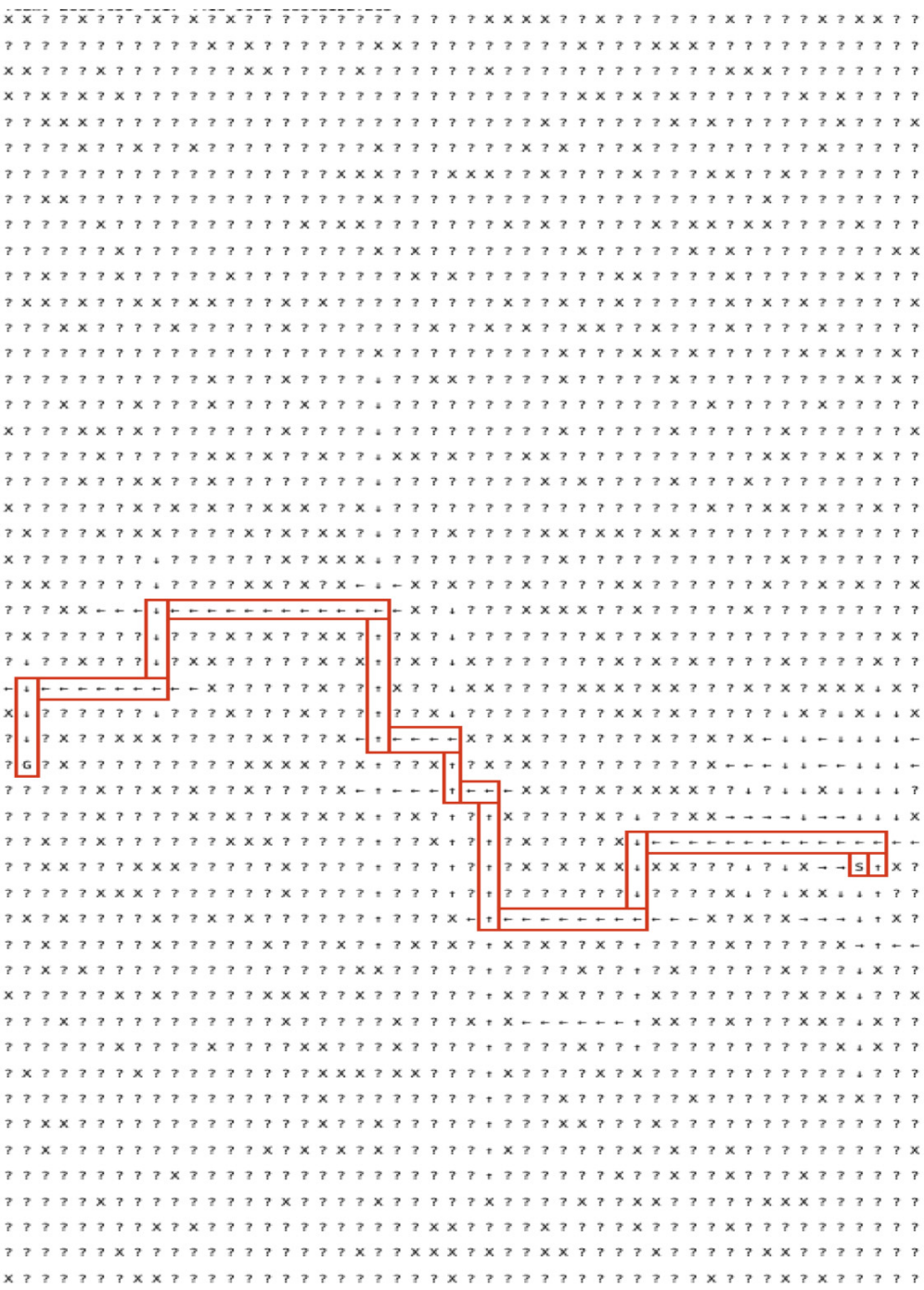

Figure A.47: Policy map for Figure A.45. The starting action was East.

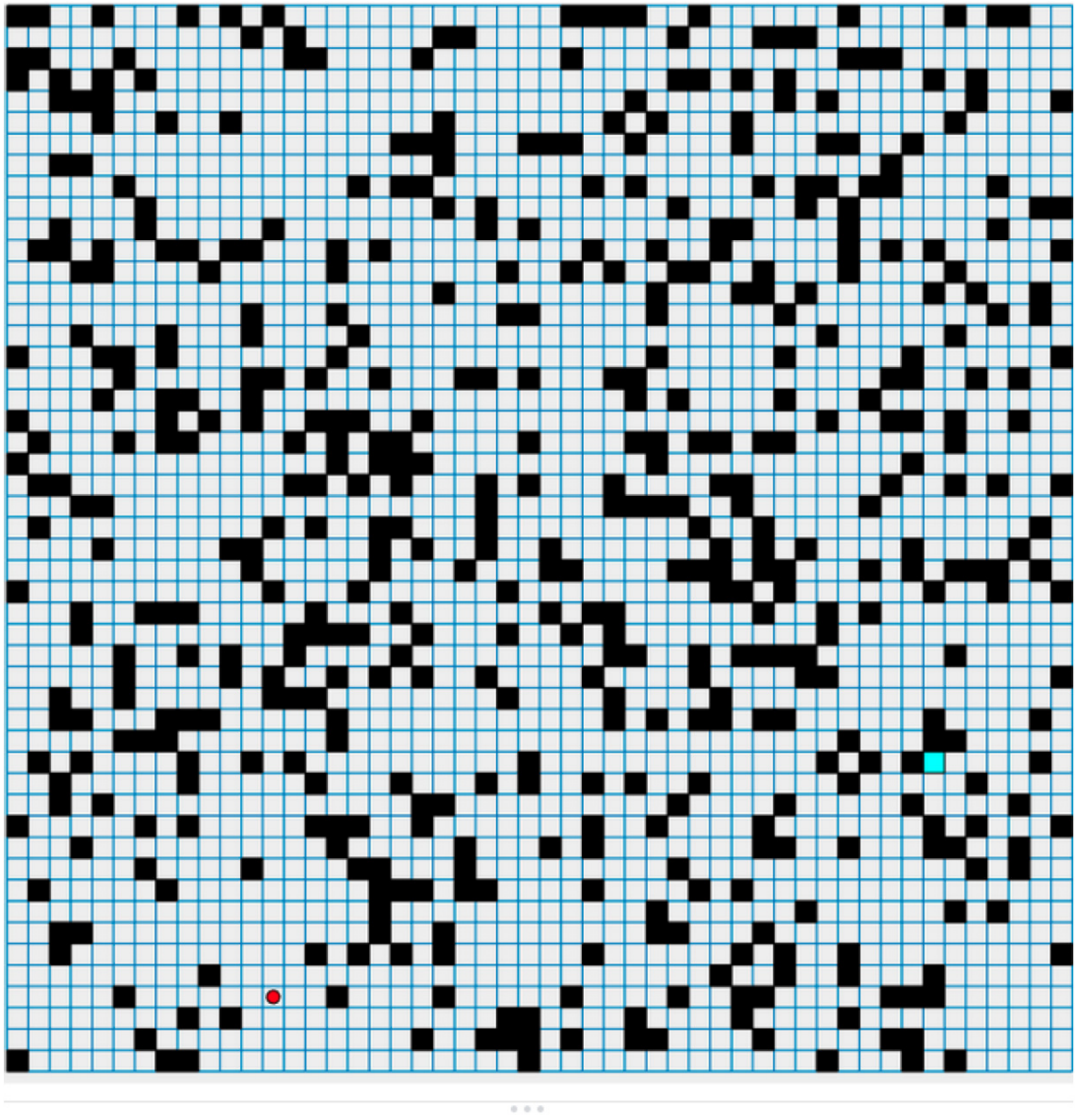

Figure A.48: $50 \times 50$ grid world task with start state (12, 3) and goal state (43, 14) (the bottom left corner is (0, 0)).

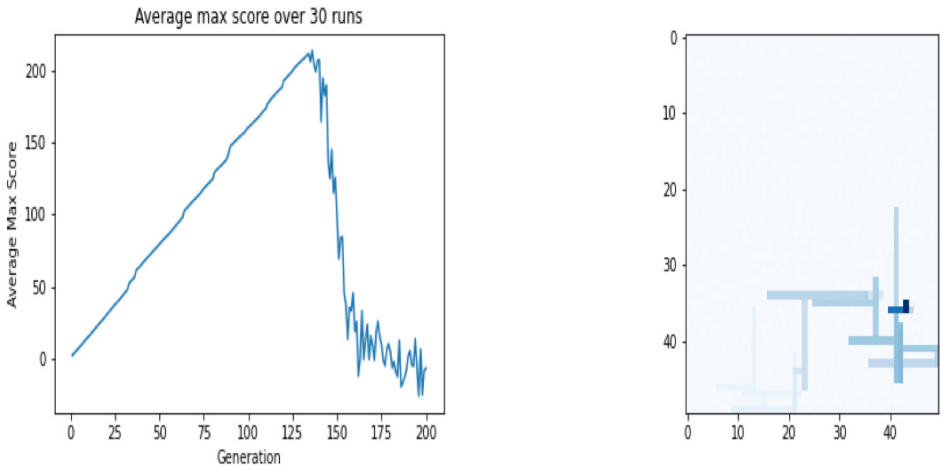


Figure A.49: QTRB results from Figure A.48. Shown from left to right: GP fitness curve and Q-value map.

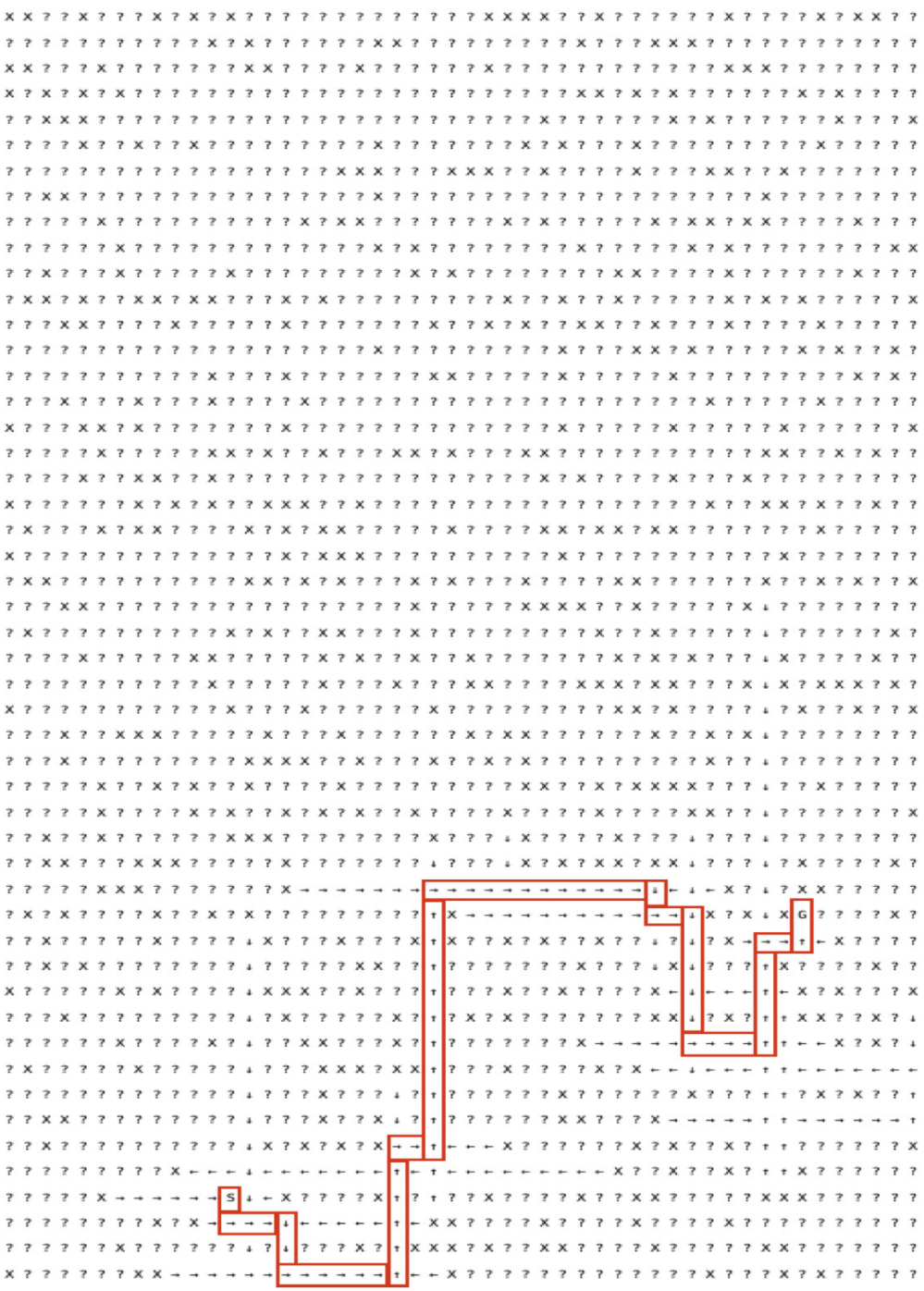

Figure A.50: Policy map for Figure A.48. The starting action was South.

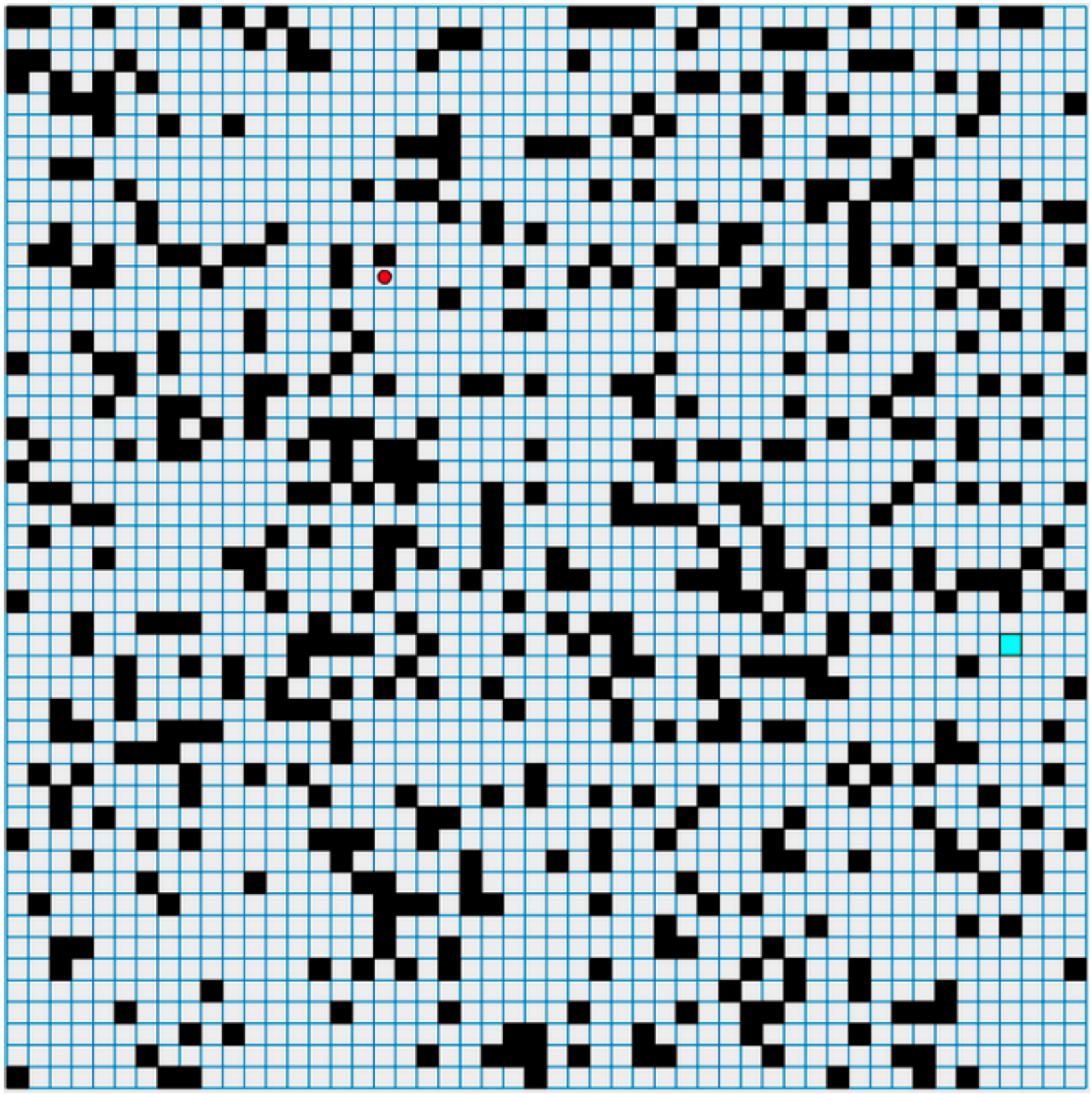

Figure A.52: $50 \times 50$ grid world task with start state (17, 37) and goal state (46, 20) (the bottom left corner is (0, 0)).

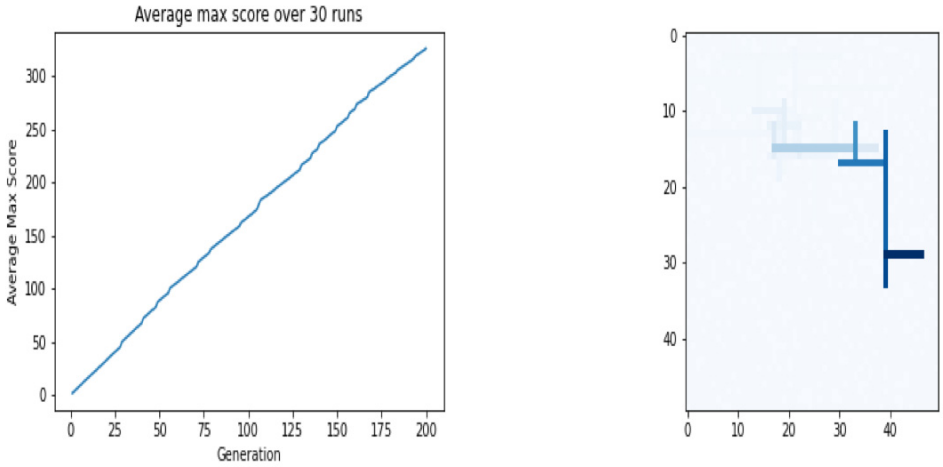


Figure A.53: QTRB results from Figure A.52. Shown from left to right: GP fitness curve and Q-value map.

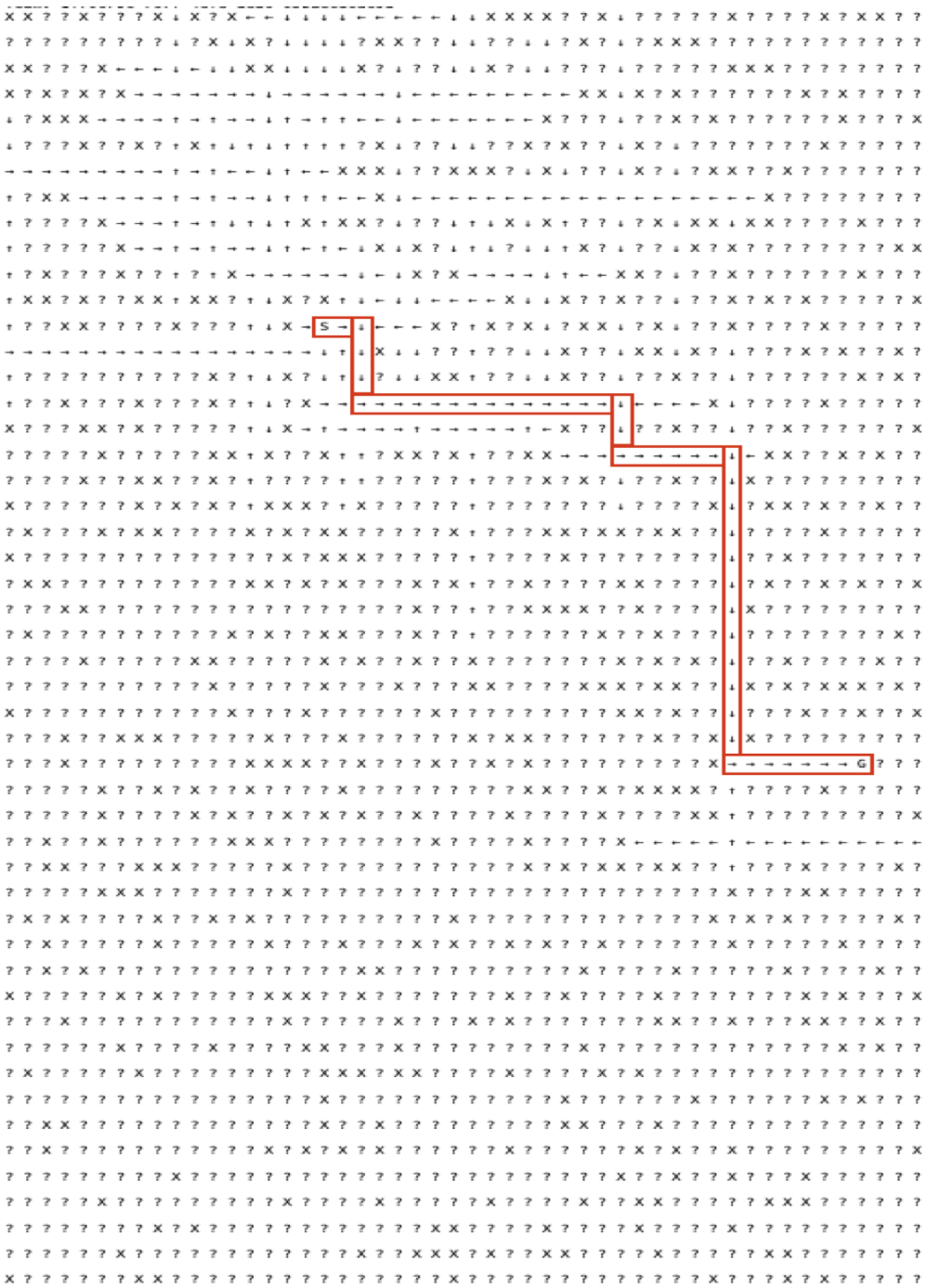

Figure A.54: Policy map for Figure A.52. The starting action was East.

Table A.3 and Table A.4 show various performance statistics from QTRB's runs on the given tasks. In Table A.3, the average step count refers to the number of environmental queries QTRB made in total, averaging over 30 runs. The # of RL runs refers to the number of champion teams GP produced for RL to learn on. % of RL wins refers to the win rate of RL on the given champion teams. Step Penalty refers to the number of steps a given team could take in GP before receiving large negative reinforcements. In Table A.4, # Superimposed Actionable Cells refers to the number of cells containing an action when all champions teams over all runs were superimposed onto the given task. The % Superimposed cells leading to the goal refers to the subset of those actionable cells that were connected to a path that led to the goal.

Table A.3: Performance data for QTRB, all averaged over 30 runs for each given task.

| **Task Figure** | **Average Step Count** | **# RL Runs** | **% RL wins** | **Step Penalty** |
|---|---|---|---|---|
| A.44 | 7486 | 69 | 35 | 3500 |
| A.45 | 9159 | 5 | 40 | 3500 |
| A.48 | 9064 | 15 | 47 | 2500 |
| A.52 | 8868 | 33 | 46 | 7000 |
| A.55 | 6191 | 114 | 47 | 3000 |

Table A.4: Shown are the number of actionable cells each task had once champion solutions from each run of a given task was superimposed onto one grid world. The percentage of cells that were connected to a path which lead to the goal of the task is also shown.

| **Task Figure** | **# Superimposed Actionable Cells** | **% Superimposed cells leading to goal** |
|---|---|---|
| A.44 | 1481 | 58 |
| A.45 | 859 | 75 |
| A.48 | 772 | 35 |
| A.52 | 1300 | 47 |
| A.55 | 1494 | 54 |

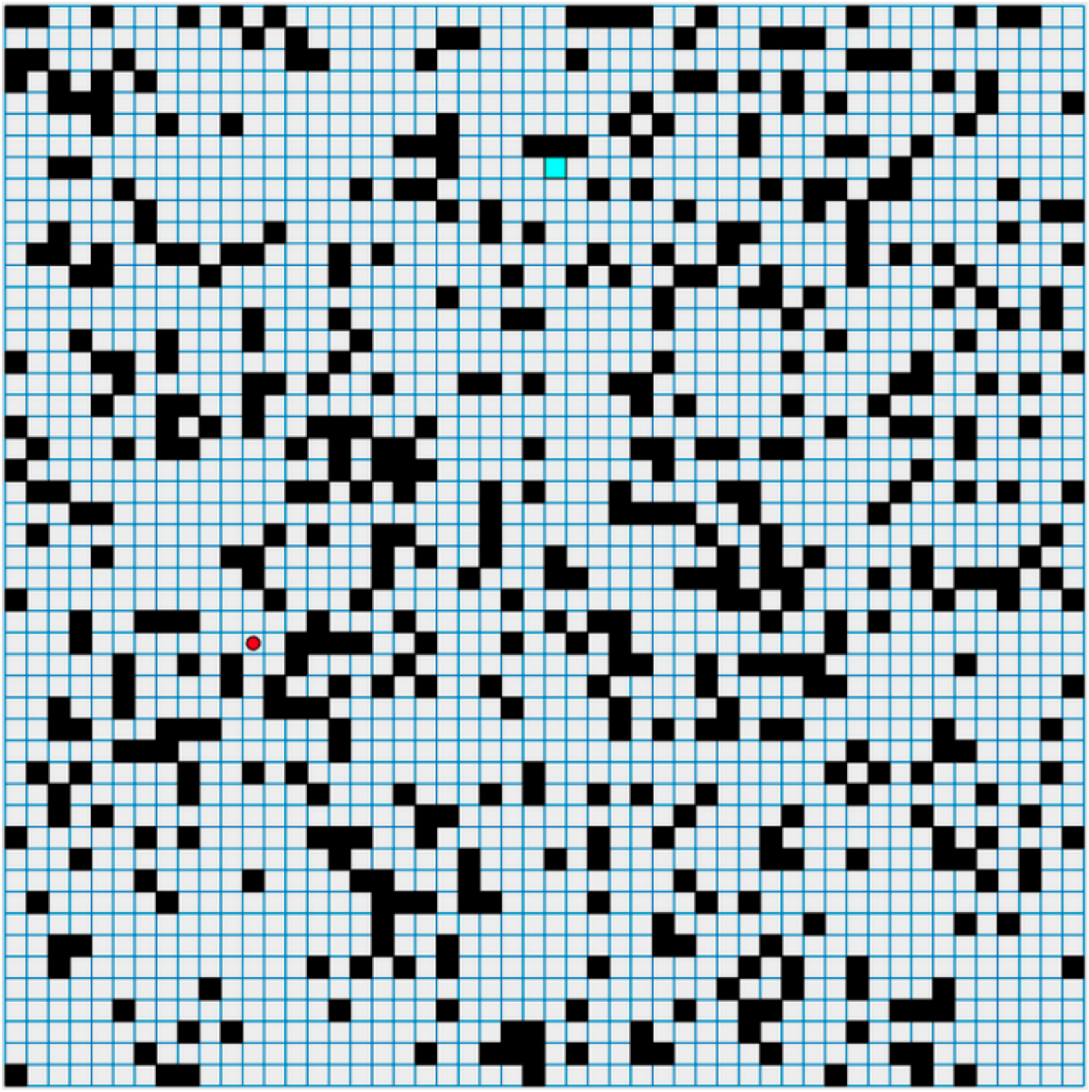

Figure A.55: $50 \times 50$ grid world task with start state (11, 20) and goal state (25, 42) (the bottom left corner is (0, 0)).

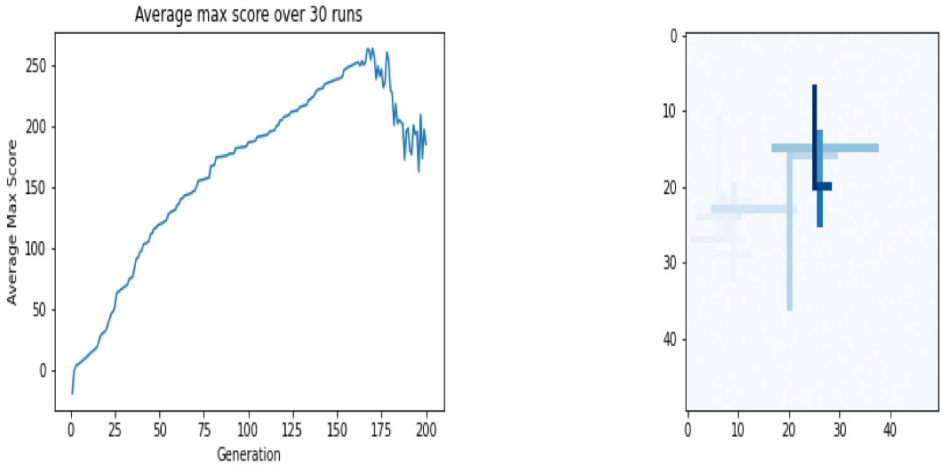


Figure A.56: QTRB results from Figure A.55. Shown from left to right: GP fitness curve and Q-value map.

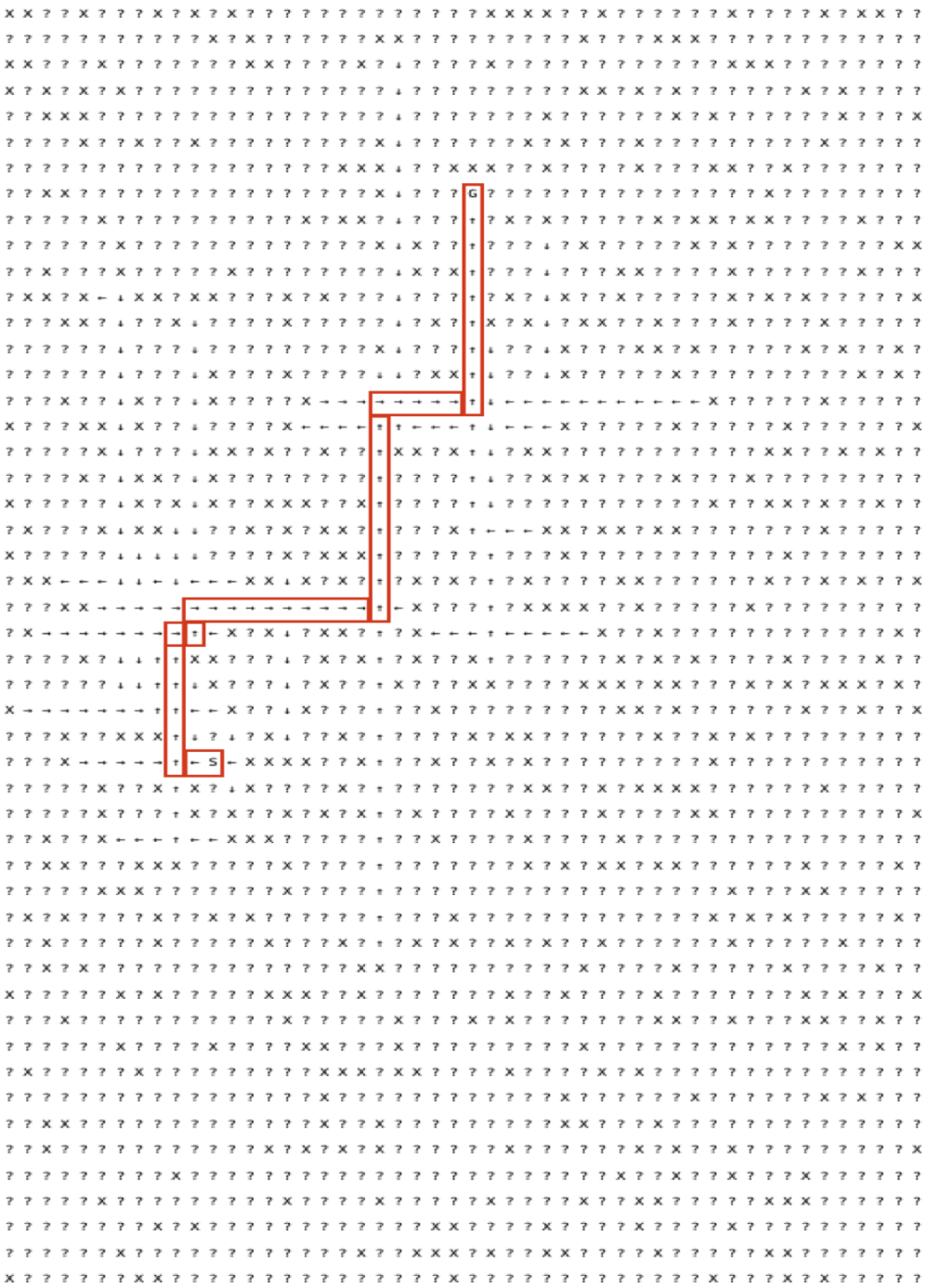

Figure A.57: Policy map for Figure A.55. The starting action was West.