Reinforcement Learning in a Generalized Platform Game

Master’s Thesis Artificial Intelligence
Specialization Gaming

Gijs Pannebakker
Under supervision of
Shimon Whiteson
Universiteit van Amsterdam

June 2010
Acknowledgements

Most of all I would like to thank my supervisor, Shimon Whiteson, whose persistent help and inspiring suggestions kept me determined to complete this thesis.

Special thanks go to my family, Ite, Jos and Pim. Thank you for your support, patience and love.
Abstract

The platform game genre is a relatively new benchmark domain in reinforcement learning. The generalized Mario domain of the Reinforcement Learning Competition is based on the classic platform game Super Mario Bros, and features a complex control task with a large state space, many possible interactions between agent and environment, and a non-trivial optimal solution. Unique to the domain are the availability of various reward systems and physics systems. This thesis contributes a detailed analysis of two novel approaches to finding control policies for this domain: Hierarchical task decomposition and direct policy search. For both approaches, a general agent and an adaptive agent are developed, the latter being able to adapt to different reward systems and physics systems. The approach using hierarchical task decomposition is not successful in this research, because the initially intuitive decomposition of the problem turns out to be a disadvantage. Empirical evidence demonstrates that the Adaptive Hill Climber, a direct policy search approach that learns different parameter vectors for different environmental parameterizations, performs significantly better than the other agents, as well as the example agent provided by the Reinforcement Learning Competition. In order to be able to win the Reinforcement Learning Competition, several aspects of the Adaptive Hill Climber still need to be improved. However, on fitness evaluations of 2000 steps, the Adaptive Hill Climber is able to beat the RL Competition winner of 2009 by scoring up to 320% more reward.
# Contents

1 Introduction 1

2 Background 4
   2.1 Reinforcement learning 4
      2.1.1 Value function estimation 5
      2.1.2 Direct policy search 6
   2.2 Hill climbing algorithm 7
   2.3 Hierarchical task decomposition 7
   2.4 Related work 9

3 Generalized Mario Domain 13
   3.1 Generalized domain 13
   3.2 Environment 14
      3.2.1 Landscape 15
      3.2.2 Mushrooms and flowers 16
      3.2.3 Enemies 17
   3.3 Agent 18
      3.3.1 Observations 18
         3.3.1.1 Tiles 18
         3.3.1.2 Entities 19
      3.3.2 Actions 20
   3.4 Differences between the RL Competition and Mario AI Competition 2009 21
   3.5 Analysis of the domain 22

4 Methods 24
   4.1 Example Agent 24
   4.2 Fixed Policy Decision Tree 27
   4.3 Adaptive Decision Tree 30
   4.4 Hill Climber 35
   4.5 Adaptive Hill Climber 44
List of Figures

3.1 Pits ................................................................. 16
3.2 Mushrooms and Flowers ..................................... 17

4.1 Fixed Policy Decision Tree .................................. 29
4.2 Adaptive Decision Tree ...................................... 34

5.1 Generalization properties of the Hill Climber ............ 51
5.2 A fitness evaluation of the Hill Climber. ................... 53
5.3 Results for the single-instance experiment ................... 54
5.4 Results for the multi-instance experiment ................... 57
5.5 Comparing the AHC with the RL Competition winner on difficulty 4 .. 60
5.6 Comparing the AHC with the RL Competition winner on difficulty 10 .. 61
List of Tables

5.1 Results for the single-instance experiment .......................... 56
5.2 Results for the multi-instance experiment .......................... 58
A.1 Variations in training data ................................................. 67
B.1 Policy parameter ranges .................................................. 69
List of Algorithms

1. Example Agent: RepeatActionSequence() ........................... 25
2. Example Agent: DecideNewAction() ............................... 26
3. Hill Climber: Main method() ................................. 36
4. Hill Climber: Fitness Evaluation() ......................... 37
5. Hill Climber: DecideNewAction() ............................... 39
6. Hill Climber: Direction() ..................................... 40
7. Hill Climber: Hesitating() ..................................... 41
8. Hill Climber: Jumping() ...................................... 42
9. Hill Climber: Speeding() ...................................... 43
10. Adaptive Hill Climber: Main method() .......................... 46
11. Adaptive Hill Climber: FitnessEvaluation() .................. 47
Chapter 1

Introduction

Reinforcement learning (RL) [1] is a sub-area of machine learning, and is one of the most active research areas in artificial intelligence (AI). It is used for solving sequential decision problems (SDPs) in a wide variety of domains, including robotics [2], system optimization [3] and gaming [4]. In RL, an agent must optimize its behavioral policy by maximizing a long term numerical reward. This reward is obtained by interacting with an uncertain environment.

In 2009 the third Reinforcement Learning Competition (RL Competition [5]) was held, an event organized by experienced researchers in the field of RL. The goal of the competition was to ‘be a forum for RL researchers to rigorously compare the performance of their methods on a suite of challenging domains’. The participants of the RL Competition 2009 had four months to develop and submit a learning agent. The results were presented during the workshop at the Multidisciplinary Symposium on Reinforcement Learning, at the 2009 International Conference on Machine Learning (ICML’09 [6]) in Montreal, Canada. The competition featured six different domains, each consisting of an environment with which an agent can interact, with the goal of maximizing the cumulative reward that it receives in a fixed number of steps. To make sure learning would be required to win the competition, the domains were generalized. A generalized domain includes a collection of SDPs that is defined by a set of parameters, forcing the agent to be flexible and robust to variations.

One of the domains was the newly introduced generalized Mario domain. Its environment was based on the open-source game Infinite Mario Bros by Markus Persson [7], which is a remake of the classic video game series Super Mario Bros by Nintendo. The first game in the series was released in 1985 and made the side scrolling platform game genre, which is a combination of side scrolling games and platform games, immensely popular. In side scrolling games the gameplay action is viewed from the side as a player-controlled
character moves through a 2 dimensional level. The view is kept on the player-controlled character by scrolling the level. In platform games, the main character must run and jump through complex levels featuring enemies and traps. It often requires sophisticated and multifaceted strategies to accomplish a good score.

Computer games can be an ideal proving ground for RL algorithms, providing a controlled environment that offers a challenging learning curve for both computers and humans [4]. Applying RL to computer games may lead to new insights in AI as well as in computer games themselves. Adaptation mechanisms can make games more fun and add to their replayability. This is the entertainment value of playing a game more than once. Different genres of games require different skills in order to be successful at them. RL algorithms have successfully learned computer games in most gaming genres, examples being Pacman [8], XPilot [9], Quake II [10], Unreal tournament [11], fighting games [12] and racing games [13, 14]. However, before 2009, no RL research was done in the platform game genre to the best of our knowledge. Currently, the Mario domain of the RL Competition 2009 is the only generalized platform game domain.

The generalized Mario domain has an enormous amount of procedurally generated levels, which, combined with various reward systems and physics handling systems, provides for an endless supply of different situations. There are several types of enemies, each of which has its own behavior and way to be dealt with. Therefore it is important for any machine learning algorithm to have an effective state representation. Interesting challenges of the domain include the use of abstraction to achieve such a state representation, ensuring a versatile learning algorithm that is capable of dealing with a wide range of situations, and making sure that lessons learned in one situation will be applied when conditions are similar. The focus of the research in this thesis lies on methods that adapt to different physics and reward systems.

In this thesis, several approaches for tackling the generalized Mario domain problem are described, analyzed and compared. These can be divided into three categories. The first category is a very simple yet effective agent that came with the competition software: the Example agent. This agent combines random moves with previously done moves in a surprisingly efficient way. The second category consists of two algorithms that follow a hierarchical task decomposition approach. They consist of several components, each specialized in a different type of behavior. Each step in the game, a decision tree chooses which component is used depending on the situation. Both the components and the decision tree are hard-coded and only involve a minimal amount of learning. The third category consists of two direct policy search algorithms, that seek the best set of parameters in a parameterized behavior space. These parameters are learned using a hill-climbing technique.
The thesis is organized as follows. Chapter 2 gives background on RL, hillclimber algorithms, hierarchical task decomposition, and related work. Chapter 3 gives a detailed description of the generalized Mario domain. Chapter 4 describes the methods used, and chapter 5 shows their respective results in different experiments. Chapter 6 discusses the results and outlines opportunities for future work.
Chapter 2

Background

In this chapter, several methods are described that form the foundation on which the research in this thesis was built. The basics of RL are described, as well as value function estimation, direct policy search, hill climbing algorithms and hierarchical task decomposition. These methods will be referenced to throughout the thesis. Also, an overview is given of related work.

2.1 Reinforcement learning

RL is learning through trial and error. The learner, called the agent, must learn a function that maps situations, known as states, to actions. This function is called a policy. The agent receives a numerical reward after every action. Its goal is to maximize the cumulative reward in the long term. This creates a dilemma for the agent. Exploring new states and actions may lead to better solutions but it is risky and may result in negative rewards as well. There is a tradeoff between favoring exploration of unknown states and actions and exploitation of already known states that yield high reward.

Everything outside the agent is called the environment. The agent continuously interacts with the environment in a sequence of discrete timesteps, $t = 0, 1, 2, \ldots$. This is known as a sequential decision problem (SDP). At each timestep $t$, the agent makes an observation $o_t \in O$ of the current state $s_t \in S$ it is in, $O$ being the set of possible observations and $S$ being the state space. The correlation between a state and its observation is defined as a general observation function. The information in observation $o_t$ and the history of observations $o_0 \ldots o_{t-1}$ can be used by the agent to decide upon action $a_t \in A$, where $A$ is the set of actions available to the agent. The next step, $t + 1$, the agent will receive a reward $r_{t+1} \in \mathbb{R}$ partly as a consequence of the taken action $a_t$. 
2.1.1 Value function estimation

If the response of the environment to an action taken at \( t \) depends only on state \( s_t \) and action \( a_t \), the environment is said to have the Markov Property. This means that future states and rewards are independent of past states and actions. Using the information captured in the present state, all future states and expected rewards can be predicted as well as would be possible by using the information captured in the entire history up to the current time. The assumption of the Markov property forms the basis for most reinforcement learning approaches. Reinforcement learning problems that satisfy the Markov property are called Markov Decision Processes, or MDPs. A finite MDP has a finite number of states and actions.

The agents’ policy \( \pi \) can be described as \( \pi : S \rightarrow A \). A policy is called greedy if it fully focuses on exploitation of known states. A greedy policy will always pick the action that maximizes the expected return. The return is a function of the reward sequence \( r_{t+1}, r_{t+2}, r_{t+3}, \ldots \). This function often includes a discount rate parameter \( \gamma \), which determines the present value of future rewards. The discounted return is defined as

\[
R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}
\]  

(2.1)

where \( 0 \leq \gamma \leq 1 \).

Most reinforcement learning approaches are based on the estimation of value functions. A value function outputs a value that is an estimation of how good it is for the agent to be in a state. This state-value \( V^\pi(s) \) is the expected return when starting in state \( s \) and following policy \( \pi \). For MDPs, \( V^\pi(s) \) is defined as

\[
V^\pi(s) = E_\pi \{ R_t | s_t = s \} = E_\pi \left\{ \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | s_t = s \right\}
\]  

(2.2)

where \( E_\pi \) denotes the expected value given that the agent follows policy \( \pi \). Alternatively, some methods use an action-value function \( Q^\pi(s,a) \) which outputs the expected return when starting in state \( s \) with action \( a \) and following policy \( \pi \).

If the expected return for a policy \( \pi \) is greater than or equal to the expected return of policy \( \pi' \) in all states \( s \in S \), then policy \( \pi \) is better or equal to policy \( \pi' \). In short, \( \pi \geq \pi' \) is true if \( V^\pi(s) \geq V^{\pi'}(s) \forall s \in S \). In policy space \( \Pi \), the collection of policies \( \pi^* \in \Pi \) for which holds that no other policy is better, are called optimal policies. As there may be other policies that are equally good, MDPs may have have more than one
optimal policy. All optimal policies have the same optimal state-value function $V^*$, and are greedy with respect to $V^*$:

$$V^*(s) = \max_{\pi} V^\pi(s)$$  \hfill (2.3)

Partly Observable Markov Decision Processes, or POMDPs, are a generalization of MDPs, where the agent cannot fully observe the state it is in. Consequently, the decisionmaking process of the agent is often based on a probability distribution over the set of possible states, known as the belief state. The belief state is based on a set of observations and observation probabilities as well as the underlying MDP.

### 2.1.2 Direct policy search

The main approach taken in this paper, direct policy search, attempts to find an optimal policy without learning value functions. Direct policy search methods can outperform value function estimation methods on some tasks [15, 16]. They present a useful alternative to value function estimation in POMDPs and large MDPs, as their value functions can be complicated and difficult to approximate.

A policy is defined as a parameterized function $\pi(s, \theta)$ with parameters $\theta$. These can be adjusted to cover a range of policies that commonly is a subset of the policy space. Learning the right parameter setting can be done in various ways (gradient descent [17], evolutionary algorithms [18]), the most straightforward one being the hill climbing algorithm, which is described in the next section. Searching the policy space usually is computationally expensive, because policies are evaluated by executing them for a period of time. The cumulative reward collected after a set amount of steps determines the value of a policy.

The size of the subset of the policy space that is searched depends on the way the parameters are implemented to affect the policy. Depending on the domain, it can be beneficial to add more constraints that are based on human knowledge, decreasing the size of the search space. The amount of human help that should be used can be seen as a tradeoff between the initial speed of learning and the number of constraints on the learner’s policy space. Too many constraints can have a negative effect on the flexibility of the algorithm, as it prevents the learner from exploring certain parts of the policy space. Sometimes human guidance provides for a good balance as it leaves the space open but encourages certain parts.
2.2 Hill climbing algorithm

The hill climbing algorithm [19] starts with an initial solution to the problem at hand, usually chosen randomly. In the case of direct policy search, this solution is a policy. The policy is mutated by a small amount. If the mutated policy has a higher fitness than the initial policy, the mutated policy is kept. Otherwise, the initial policy is retained. The algorithm iteratively repeats this process of mutating and selecting the fittest policy until a maximum in the fitness function is reached or another stopping condition is fulfilled. It returns the last kept policy.

Especially in a continuous search space, it may not be clear if a maximum is reached. Then the algorithm could be run for a set number of iterations or stop when the improvements in fitness per iteration fall below a certain number.

A local maximum consists of one or more points in the search space that have a higher fitness than their surrounding points. The hill climbing algorithm always converges toward a local maximum, making it a local search algorithm. However, unless the search space is convex, it is not guaranteed that a global maximum is found. This is the point or group of points with the highest fitness in the whole search space. Ways to overcome this problem include trying many different starting points and doing ε-greedy policies. An ε-greedy policy selects the best solution with a probability of $1 - \varepsilon$, where $\varepsilon$ is a small probability. A random solution is chosen with probability $\varepsilon$. If a local maximum is a relatively flat part in the search space, it may cause the algorithm to cease progress and wander aimlessly. This can be overcome by increasing the step size per mutation in order to look further ahead in the search space.

Compared to other learning algorithms, the hill climbing algorithm is relatively simple to implement. Although more advanced algorithms may give better results, in some situations hill climbing works just as well.

2.3 Hierarchical task decomposition

Hierarchical task decomposition is an approach in which a complex task is decomposed into hierarchies of subtasks that are easier to solve. The separate solutions to the more manageable subtasks often can be combined to form a solution for the whole problem. Hierarchical task decomposition may enable the learner to learn faster and tackle more complex tasks. However, identifying the right subtasks for a problem requires certain knowledge of the domain, which in most cases has to be manually inserted. As with direct policy search, adding human knowledge should be done carefully because it may
guide the learner away from the best solutions or incorrectly constrain the learner’s hypothesis space.

Successful ‘implementations’ of the strategy of hierarchical task decomposition can be found in nature. Organisms are comprised of organs, which are built up from cells. Cells and organs have specialized subtasks that give organisms different abilities, like seeing, eating and walking. It enables organisms to exhibit a wide range of complex behaviours.

Whiteson et al. (2005) [20] describes various ways to implement a learning agent that uses hierarchical task decomposition. It is often useful to hand-code self-evident parts of the hierarchy and use machine learning for the less trivial parts. Depending on the domain, a switch network that learns high-level decisions based on global observations might work, while for other problems it may be advantageous to apply machine learning for one or more specific subtasks. High-level decisions can also be made using a decision tree: a tree-like structure of rules that branches out, each leaf representing a subtask. In offline learning, a machine learning algorithm learns a policy in an initial training phase. Once this phase is absolved, the algorithm does not further adapt its policy. Learning subtasks offline in a controlled environment can speed up the learning process. Systems which employ online learning update their policy every time a reward is received.

In coevolution, no human assistance is provided beyond the task decomposition, and the different components are trained simultaneously. This can be done in a competitive or a cooperative way. The evaluation of a coevolutionary algorithm can be done by putting together the components into one big network and evaluating that as a whole.

In layered learning, human assistance consists of constraints and guiding. Components are learned in a more structured, sequential fashion, which is specified in a model called a layered learning hierarchy. Special training environments are developed to train the lower layers of the hierarchy on. When the lowest level components have learned their subtasks sufficiently, higher level components can be learned that use output of the lower levels for more global decisions. Each layer directly affects the next layer by either constructing a set of training examples for that layer, providing the features used for learning, or pruning its output set.

In concurrent layered learning, lower layers are allowed to continue to adapt while higher layers are being trained. This combines the advantages of using a layered learning hierarchy with the flexibility of coevolution.
2.4 Related work

An Object-Orientated Representation For Efficient Reinforcement Learning (Diuk et al. [21]) presents an object-oriented approach to solving reinforcement learning problems that is applicable to a broad set of domains. The method is demonstrated in the videogame Pitfall!, which was released in 1982 by Activision for the Atari 2600 game console and was one of the first platform games ever created. All transitions in the game are deterministic. In the first level, a man must find a path from the left of the screen to the right of the screen using walking and jumping actions, while interacting with a hole, a ladder, a log and a wall. Utilizing an object-oriented representation of the environment that features Object-Oriented MDPs, a reinforcement learning algorithm DOORMAX is capable of learning the fastest path through the level faster than most state-of-the-art learning algorithms. This approach gives a natural way of modeling environments and offers important generalization opportunities. It can only be applied to deterministic environments.

In the presentation An Approach to Infinite Mario [22], Paul Ringstad describes the design of the algorithm that won the Infinite Mario category of the Reinforcement Learning Competition\(^1\). The approach consists of a Q-learning algorithm that learns the values of state-action pairs using a heavily abstracted state space. Every step, the action-values of the 100 last visited state-action pairs are updated with a discounted reward. The policy of the algorithm is greedy. The abstraction is done in two steps. First, the parameters that define the raw state space are discretized. Second, every step in the game, the three discretized parameters that are most important for staying alive at that moment form the final abstract state that is used for learning. The importance of each element in a discretized state is determined by a deterministic heuristic ranking system that is based on implicit domain knowledge. In addition to the rewards from the environment, the algorithm artificially rewards itself for the distance covered in the level to aid the learning process. Also, the algorithm uses knowledge from previous trials by limiting exploration to the end of the path Mario previously took.

In August 2009 another competition was held that involved implementing an AI for a Mario playing agent: The Mario AI Competition 2009 [23]. The winner of the Mario AI Competition, Robin Baumgarten [24], managed to implement an AI involving the A* algorithm [25] that was able to finish levels on the highest difficulty degree without dying once. The heuristic used was based on moving to the right of the screen as fast as possible while trying to avoid being hurt. The algorithm does not work perfectly. Sometimes Mario gets hurt or dies. There is no learning involved in its policy. The

\(^1\)The results of the competition can be found at http://2009.rl-competition.org/results.php
algorithm leans very much on its model of the environment that is used to predict the next states of the game given Mario’s actions. The search space is updated every few steps of the game and predictions are done several steps ahead, depending on the speed of the computer it is run on. Section 3.4 discusses the exact differences between the RL competition software and the Mario AI Competition.

In *Super Mario Evolution* (Togelius et al. [4]), nine types of neural networks are evolved on the domain of the Mario AI Competition 2009. Two categories of neural networks were tested: Multi-Layer Perceptron (MLP) and Simple Recurrent Network (SRN). The MLPs were evolved using Evolution Strategies (ES). The SRNs were evolved using either ES or HyperGP, the latter being a method used for evolution of neuron weights. For the MLPs and both types of SRNs, three sizes of state space were tested: small, medium and large. The size of the state space corresponded directly with the size of the area around Mario observed by the agent. The fitness function only looked at the distance covered along a number of levels with increasing difficulty, disregarding the number of coins and powerups collected and the number of kills made. The ES-based agents obtained similar results. For these algorithms, the small networks performed best, followed by medium and large networks in that order. The HyperGP networks performed a little lower than the small ES networks. However, the HyperGP approach proved to be able to evolve networks with larger amounts of input data, as it obtained similar results regardless to the size of the state space of its networks. The problems with all networks included generalization over different levels, and spatial and temporal reach. In order to solve the latter two, the HyperGP approach seems the best option, as more input data are needed to look further ahead (and back) in time and space.

A successful example of hierarchical task decomposition in a gaming environment is described in *Evolving Soccer Keepaway Players through Task Decomposition* (S. Whiteson et al. [20]). The keepaway domain is a subtask of Robot Soccer. One team of agents, the *keepers*, must try to maintain possession of the ball while another team of agents, the *takers*, tries to get it. The game is played within a fixed area. The article compares several different approaches: a fully hand-coded strategy, three hierarchical task decomposition methods, and *tabula rasa learning*. In tabula rasa learning, a single monolithic neural network tries to learn the game using the least amount of human guidance. The three hierarchical task decomposition methods were coevolution, layered learning, and concurrent layered learning. Each task decomposition method came in two versions: One version with a hand-coded decision tree as the top layer, and a second version with a switch network as the top layer. The top layer decides between different subtasks, which were: Intercepting the ball, passing the ball, and getting to the right position to receive a pass. These subtasks were learned by neural networks, using *neuroevolution*, in which a population of neural networks evolves. One additional neural network was
used to decide to which of the teammates the ball should be passed. The results showed that learning low-level behaviors while learning the switch network at the same time is much more difficult than only learning the low-level networks while using a hand-coded decision tree for the high-level decisions. The best performing algorithms were concurrent layered learning with the decision tree and coevolution with the decision tree. Tabula rasa learning performed worst of all, indicating that the task decomposition is essential for obtaining high results in this domain. The hand-coded strategy performed better than some of the methods with a switch network, but could not reach the heights of the concurrent layered learning and coevolution when using a decision tree, proving that machine learning can outperform hand-coded approaches in complex control tasks. In *Reinforcement Learning for RoboCup-Soccer Keepaway* (P. Stone et al. [2]), the same domain is also solved by using a task decomposition approach.

Hierarchical task decomposition is mainly used for multi-agent problems. A computer game genre that has seen much research with this approach is *Real Time Strategy games* (RTS) [26–30], in which multiple agents must work together to build a base, build units, harvest resources, and attack the enemy. RTS games involve complicated teamwork between agents because attacks need to be coordinated and sophisticated planning is needed for economical decision-making. These games are ideal for using hierarchical task decomposition because subtasks are relatively independent and can be carried out by different agents. An example of hierarchical task decomposition used in a single-agent problem is described in *It Knows What You’re Going To Do: Adding Anticipation to a Quakebot* (Laird [31]). The article describes the AI of an agent in the three-dimensional shooting game Quake II, which relies on a large dynamic system of more than 100 subtasks structured in several layers. Subtasks are defined using rules, and the system allows addition and deletion of rules based on observations. By adding and subtracting rules, the agent learns to predict actions of its opponent, and is able to anticipate to this information.

In the future, computer games will increasingly make use of RL methods to generate content. *Computational intelligence in games* (Miikkulainen et al. [32]) discusses the achievements and future prospects of neuroevolution in video games. The replayability of a game can be greatly enhanced by adding adapting intelligent non-player characters or by adding a training system that adapts its strategy as the player plays more games and gets better. In *Making Racing Fun Through Player Modeling and Track Evolution* (Togelius et al. [33]), an evolutionary algorithm called *Cascading Elitism* is trained to generate racing tracks that are fun to play. The fitness function determining the amount of fun is based on the sensation of speed, the amount of challenge, the amount of possible drift in a turn, and the variety of a track. The article also talks about evolving an agent that can race on the generated tracks. For future work, the article suggests using a
theory of renowned game designer Raph Koster in the fitness function. In *A theory of fun for game design* (Koster and Wright [34]), Koster describes that ‘playing and learning are intimately connected, and a fun game is one where the player is continually and successfully learning’ [34]. In other words, if a learning agent shows a long and gradual learning-curve, this indicates that a track is fun to play. Another example of evolving content in games is given in *Evolving content in the galactic arms race video game* (Hastings et al. [35]). A game is presented in which players pilot space ships and battle enemies in a galactic encounter. By killing enemies, new weapons can be obtained. As the game progresses, a neuroevolutionary algorithm keeps track of which weapons are used most by a player, and evolves new weapons to increasingly suit the player’s tastes. The variety in weapons is very large and more flexible than simple content randomization, which increases the fun of the game.
Chapter 3

Generalized Mario Domain

This chapter describes the details of the generalized Mario domain that help understand the goals and challenges of the research presented in this thesis. First, the notion of a generalized domain will be defined, and it will be explained how the generalized domain was used in the RL Competition to encourage participants to use RL in their agents. Second, the environment and the agent are described, with detailed information on the landscape, enemies, observations and actions. By then, enough information has been given to make an insightful comparison between the RL Competition and Mario AI Competition 2009, which discusses the differences and makes an argument why the RL Competition presents a greater challenge. Finally, an analysis of the domain is done, discussing the complexity and the challenges of the domain.

3.1 Generalized domain

In a generalized domain, a class of SDPs is defined by a set of parameters. Altering these allows for a range of variations within the class of SDPs.

The RL Competition was broken up into three phases. In the first phase, competitors built their agents, testing and training them on their local systems using SDPs that were provided with the competition software. The second phase was the proving phase, in which competitors tested their agents a limited number of times, on a set of newly parameterized SDPs. In the final phase, each competitor had one run on yet another set of SDPs. Each SDP was evaluated on 100,000 steps. The goal of the competition was to obtain as much reward as possible over all SDPs in the final phase.

The changing of parameterizations of SDPs was done to encourage competitors to use online learning in their agents. In order to win, agents needed to be robust to variations
between the different SDPs. Non-learning agents that use hand-coded strategies have a hard time adapting to new parameter settings. To emphasize this even more, the exact specifications of the parameters and the dynamics for altering the parameter settings were kept secret during the competition.

3.2 Environment

The environment consists of four elements: the landscape, the entities, the reward system and the physics system. A level is a configuration of the landscape and the entities.

Landscape The landscape is made of square tiles. Tiles define the graphics of the landscape and if Mario can go through a certain part of the level or not. The usual level size is 320 by 16 tiles.

Entities The entities are the movable parts of the level. They include enemies, mushrooms, flowers, fireballs and Mario himself.

Reward system The reward system contains five distinct rewards for five events:

- Mario reaches the end of the level (usually gives a big positive reward),
- Mario dies (usually gives a big negative reward),
- Mario collects a coin or mushroom or flower (usually gives a small positive reward),
- Mario kills an enemy (usually gives a small positive reward),
- a step of the agent (usually gives a small negative reward).

For the numerical values of the rewards in the reward systems used for the first phase of the RL Competition, see Appendix A.

Physics system The physics system determines the way Mario moves. This includes walking speed, running speed, jumping height and speed, and falling speed. For the specifics on the different physics systems, see Appendix A.

The environment is episodic, dividing the learning process into independent subsequences of steps called episodes. An episode always starts with Mario placed at the beginning (the left side) of a level, and ends when Mario dies or when he reaches the finish line on the right side. The parameterization of the environment does not change during an episode. The environment is parameterized with level seed, level type, difficulty and instance.
Level seed The level seed is the random seed used by the level generator. It can be set to any integer, providing millions of different levels. Levels are generated by probabilistically choosing a series of idiomatic pieces of levels and fitting them together [36].

Level type There are three level types, which specify the graphical appearance of the landscape of the level and alter its configuration. The level type parameter allows for more levels but does not change anything relevant to the way the game should be played.

Difficulty The difficulty parameter ranges from 0 to 10, 0 being a very easy environment and 10 being very difficult. Higher difficulty means more enemies, harder enemies, and more and harder pits in the landscape.

Instance The instance parameter determines a set of parameters, as it specifies the reward system and the physics system, as well as the width of the level and the maximum number of steps that a trial may take. In the RL Competition software package ten instances were given to train on, though in the proving and testing phase of the competition the rewards and physics were changed to unknown values, making flexibility of the agent essential. The exact differences between the ten training instances are described in Appendix A.

3.2.1 Landscape

The landscape consists of several elements:

Air tiles Air tiles are transparent and they provide space for the entities to move through.

Hard matter Nothing can go through hard matter. Examples of hard matter are grass, stones, pipes and used questionmark blocks.

Semi-hard plateaus These tiles are only hard when coming in from above. They can be walked on but are passable from the bottom and sides.

Smashable bricks Bricks can be destroyed by bumping Mario’s head into them. The tile then becomes an air tile.

Question mark blocks When Mario slams his head into question mark blocks a power-up will come out of it. This can be a coin that is picked up automatically, or a mushroom or flower. When bumped into, these blocks become hard matter.
Some question mark blocks are secret. Instead of having a question mark graphic, they look like a destructible brick.

**Coin tiles** When Mario passes a coin tile, the coin is collected and the tile is converted into an air tile.

**Pits** Each level with difficulty higher than 0 will have several open spaces in the landscape that need to be jumped over. Everything that falls into them is destroyed. When Mario falls into a pit, the episode ends immediately. Pits sometimes have stone stairs around them to make them harder to jump over. The height of the stair depends on the difficulty of the level.

**Finish line** This indicates the end of the level.

![Figure 3.1: Some variations of pits in the landscape](image)

### 3.2.2 Mushrooms and flowers

Mushrooms and flowers are only found in question mark blocks. When a mushroom comes out of a question mark block, it will start moving to the right. When it bumps into a wall it will start moving in the opposite direction. Flowers stay in place above the question mark block. Mushrooms are easier to collect than flowers, because their movement makes the chance of running into them quite large, and flowers are often in places that are hard to reach.

Each episode, Mario starts out being small. When Small Mario is hurt by an enemy, he dies, ending the episode. When small, picking up a mushroom will make Mario become big. When Big Mario is hurt by an enemy, he will be invincible for a short time, and become small again. When Big Mario picks up a flower, he will be come Fiery Mario. Fiery Mario has the ability to shoot fireballs. When Fiery Mario is hurt by an enemy, he will be invincible for a short time, and become Big Mario again.
3.2.3 Enemies

The Mario domain features five different types of enemies:

**Goomba** A Goomba will walk to the left while possible, even if this means running into a pit. When a it bumps into a wall it will go back. It can be killed by jumping on them or hitting them with a fireball. When it collides with Mario in another way than jumping on it, Mario will be hurt.

**Green Koopa** Green Koopas have the same walking behavior as Goombas. When jumped on, a Green Koopa turns into a Shell. When a Green Koopa collides with Mario in another way, it will hurt Mario. When shot with a fireball, it will die instantly.

**Red Koopa** Red Koopas have the same behavior as Green Koopas, with one exception: when facing a cliff drop, they will turn back.

**Shell** A Shell is created when a Koopa is jumped on. Shells can be green or red, but this does not have an effect on their behavior. When Mario jumps on a Shell, it will be propelled very fast across the level. When it bumps into a wall it will go back. A moving Shell can be stopped by jumping on it again. When it collides with Mario in another way than jumping on it, Mario will be hurt. When a moving Shell comes across another enemy, that enemy is killed. This provides a fast way for killing a big pack of enemies. Fireballs have no effect on Shells.

**Spikey** Spikeys have the same walking behavior as Goombas. Colliding with them in any way will hurt Mario. Fireballs have no effect on Spikeys. The only way to kill them is using a Shell.
**Piranha Plant** Piranha Plants only move vertically. They periodically come out of a pipe and quickly return. When Mario is very close to a pipe that contains a Piranha Plant it will not come out. A Piranha Plant can be killed by jumping on it or hitting it with a fireball. When it collides with Mario in another way, it will hurt Mario. Not all pipes contain a Piranha Plant.

Goombas, Koopas, and Spikeys can have wings, which will make them bounce across the level. Winged enemies will always bounce toward Mario. When winged Goombas or Koopas are jumped on, they will lose their wings and acquire their normal walking behavior.

### 3.3 Agent

The agent can be thought of as a human player playing the game with a Nintendo controller in his hands that is used to steer Mario. In a normal game of Mario, the speed of the game is defined at 24 ticks per second. A *tick* is an atomic timestep in the environment: Every tick all entities on the screen, including Mario himself, do an atomic action. Once every five ticks, the agent does an observation of the state of the environment in the current tick, and is able to press the buttons of the virtual controller. This section describes the specifications of the agent’s observations and actions.

#### 3.3.1 Observations

Each observation of the agent consists of information about all tiles and entities on the screen in the current tick. The four states of the environment in between two steps are not observable.

##### 3.3.1.1 Tiles

In every observation, 352 tiles are visible, lined up in 16 rows of 22 tiles. The position of each tile is defined by integer $x$ and $y$ coordinates. The $x$ coordinate is determined by the number of the column that the tile is in, counting from the beginning of the level. The $y$ coordinate represents the row, starting at the lowest row.

The tiles are represented with chars and can be any of the following types:

- 0 to 7: a 3 bit vector determining the type of hardness of the tile. A bit is 0 if can pass, 1 if cannot pass.
– The first bit indicates whether an entity can pass through this tile from the top,
– The second bit determines if an entity can pass through from the bottom,
– The third bit determines if an entity can pass through from either side.

• $M$: the tile that Mario is standing on.
• $b$: a smashable brick, or a secret questionmark block.
• $?$: a question mark block.
• $\$$: a coin.
• $\mid$: a pipe. Different than 7 because piranha plants often come out of pipes.
• $!$: the finish line.
• $\backslash 0$: tile out of the visible region.

Tiles with $x$ position $< 0$ are considered always solid.

### 3.3.1.2 Entities

For every entity on the screen, six attributes are known to the agent:

- $type$, the type of the entity that determines his behavior and look. There are 12 different types: Small Mario, Red Koopa, Green Koopa, Goomba, Spikey, Piranha Plant, Mushroom, Flower, Fireball, Shell, Big Mario, Fiery Mario.

- $winged$, a boolean that is true if the enemy has wings.

- $x$ and $y$ coordinates, the entity’s position on the x and y-axis of the level. The coordinates are aligned with the tile positions, but can have in-between floating point values. The speed of the entity is also defined by floating point values.

- $x$-speed and $y$-speed, the entity’s current speed in the $x$ and $y$ direction in tiles per step floating point values.

Note that Mario is defined both as an entity and a tile. Mario’s current state is defined by the $type$ attribute. His other known attributes are the same as the other entities.
3.3.2 Actions

Every step, the agent decides on an action. This action is composed of an integer array of length 3: \([-1, 1, [0, 1], [0, 1]]\). The values correspond with the buttons on a Nintendo controller: \{direction pad, A, B\}. The first value refers to the direction Mario is heading, with -1 for left, 0 for neither and 1 for right. The second value refers to not jumping (0) or jumping (1). Pressing the jump button while already jumping increases the length and height of Mario’s jump. The third value refers to the speed button being off (0) or on (1). Pressing the speed button will increase Mario’s speed when moving, and when Mario is fiery, pressing it will also make him shoot fireballs. In total this gives the agent 12 different actions to consider every step.

The five atomic actions executed by Mario during one step are not equal to five times the action specified by the agent. The execution of the action specified by the agent is delayed for one or two ticks. After deciding on an action, there will be one tick wherein the previous action is still executed. Then follows one tick wherein the direction of the new action is used, but the jump and speed buttons are set to 0. The last three ticks are exactly like the new action specified by the agent.

Mario’s movement in a tick is determined not only by the current atomic action. His speed in the previous tick as well as the environment also play a role. The speed in the previous tick gives Mario momentum. For example, if Mario starts running from a standstill, it will take up to 30 ticks until his maximum speed is reached in most instances.

Combining the action given by the agent with the environment, several interactions are possible:

- Bumping into walls or blocks which stops Mario.
- When Mario kills an enemy by jumping on it, he will automatically jump up next tick.
- Picking up power-ups, which will then disappear.
- When becoming big or fiery Mario the environment freezes for about 3 steps.
- Destroying bricks by bumping his head into them.
- Activating blocks with a questionmark on them by bumping his head into them.
- Becoming invincible for a few seconds by getting hurt.
- Launching / stopping shells by jumping on them.
• Mario can get killed by getting hurt when small.

• Mario can get killed by falling in a pit.

3.4 Differences between the RL Competition and Mario AI Competition 2009

Section 2.4 mentions the agent of Robin Baumgarten [24] that won the Mario AI Competition 2009 [23]. Like the software of the RL Competition, the software of the Mario AI Competition was based on Infinite Mario Bros. The alterations that the RL Competition introduced to the Infinite Mario Bros software make it improbable to implement such a successful agent in the RL Competition using the same deterministic techniques. To clarify this, I will discuss the differences between the RL Competition software and the Mario AI Competition.

In the Mario AI Competition, all entities will always start an episode at the same coordinates. In the RL Competition, the starting location of entities can slightly vary between episodes. Having a different starting position means that remembering a successful action sequence from a previous episode will not guarantee the same result in the new episode.

The Mario AI Competition is not generalized: There is only one physics system and one reward system. The physics system is implemented such that Mario can reach every part of the level. In the RL Competition, this is not the case in most instances, which may cause a deterministic algorithm to loop Mario’s behavior. An example of this is when Mario keeps trying to grab a coin that he cannot reach. The reward system in the Mario AI Competition is measured as the average distance travelled on a number of previously unseen levels, with a set number of trials per level. This means that Mario does not have to worry about collecting coins and killing enemies. The physics system and the reward system are easier to deal with in the Mario AI Competition. However, the biggest advantage of having no instances is that a model of the world is available that can be used to predict Mario’s next position given an action and a state. In the RL Competition, this model must be learned which adds an extra layer of complexity.

In the Mario AI Competition, an agent gives an action to Mario on every tick of the game. This means that errors in predicting the next state the agent will see will be five times smaller than in the case of the RL Competition, where the agent gives one action every five ticks. This makes controlling Mario a harder problem for the RL Competition. Additionally, in the Mario AI Competition there is no delay between
deciding on an action and executing it, and the action executed is always equal to the action given.

### 3.5 Analysis of the domain

The environment satisfies the Markov property. Additionally, if a model of the physics system would always be available to the agent, it would be possible to predict a large part of the next state with complete certainty. For these reasons, the most sensible way to model the Mario domain is modeling it like a deterministic MDP.

The size of the state space is huge. Consider the following features of the four elements of the environment:

- 352 visible tiles on the screen with 13 different types.
- The number of entities on screen, which has a maximum of about 10. There are 12 entity types, four of which can be winged. All entities move in a continuous space.
- 7 variables determining the physics system.
- 5 variables determining the rewards system.

The entities move in a continuous space, which can be simplified by applying a discretization on the position $x$, $y$, $x$-speed and $y$-speed variables. Since a pixel is the least significant difference in checking for collisions, it is reasonable to include as many possible positions in the representation as there are pixels on the screen, which is 90,112. For the speed variables, about 20 categories are needed for each direction. Without taking into account the different physics and reward systems, the number of states of the state space would be more than $26 \cdot 10^{12}$.

With such a large state space and an action space of 12 actions, it would take very long to learn an optimal policy. The agent would suffer from the *curse of dimensionality*: the exponential explosion in the number of states as a function of the number of state variables. Faster learning can be achieved by reducing the number of options available to the learner. Therefore it is essential to drastically reduce the size of the state space while maintaining the information that is vital to obtaining a good policy.

In the generalized Mario domain, the agent starts without any knowledge of the parameter settings of the MDP. In order to be able to decide on a good policy, it is essential
for the agent to obtain information about the physics and reward systems. Doing this as fast and reliable as possible is one of the challenges of the domain. An experiment may consist of several different MDPs. The agent will know when another MDP is initiated, but will not know what the new parameter setting is, or if the parameter setting was used before.
Chapter 4

Methods

This chapter gives a detailed description of five agents. Discussed are the motivations behind each agent, differences in approach, and expectations. The agents will be evaluated later on in the thesis, and their performances will be compared and discussed by referencing back to this chapter.

First, the example agent is described. This agent was provided by the RL Competition. It has a simple yet effective strategy of combining random moves with previously done moves. Because this agent has a simple policy, it will be interesting to compare it to more complicated algorithms. It provides a base line for comparisons.

The other four agents use methods that have not been evaluated on this domain before. The first two agents, the Fixed Policy Decision Tree (FPDT) and the Adaptive Decision Tree (ADT) use a hierarchical task decomposition approach. The second two, the Hill Climber (HC) and the Adaptive Hill Climber (AHC) use a direct policy search approach. Aside from the two different approaches, these four algorithms can be divided into a group of adaptive algorithms and a group of non-adaptive algorithms. The adaptive algorithms, the ADT and AHC, are designed to adapt to different instances. The other two algorithms use a general strategy for every instance.

4.1 Example Agent

The example agent was delivered with the RL Competition software. Its policy evolves around stochastically choosing new actions and remembering successful actions from previous episodes.

Algorithm 1 describes how the Example Agent uses knowledge from previous episodes to its advantage. Each episode after the first, the agent repeats the action sequence of the
previous episode up to the last 7 actions. For example, if Mario died after 20 steps, the first 13 steps are considered to be successful and will be repeated in the next episode. The last actions of the previous episode in which Mario died are called the death zone.

After repeating the successful actions from the previous episode, the agent will decide on the actions using the policy described in algorithm 2. Because that policy is sufficiently randomized, Mario will eventually find a way to reach the end of a level.

Algorithm 1 Example Agent: RepeatActionSequence()

\begin{verbatim}
previousActionSequence = list of all actions of the previous episode
currentActionSequence = list of all actions of the current episode
stepNumber = step number of the current episode

if previousActionSequence.size() > stepNumber + 7 then
    action = previousActionSequence.get(stepNumber)
else
    action = DecideNewAction()

return currentActionSequence.add(action)
\end{verbatim}

Algorithm 2 describes how the Example Agent decides on a new action. The agent will never decide to move to the left. Its core policy consists of walking to the right with a 0.95 probability of not jumping. This probability, \textit{jumpHesitation}, is lowered as certain observations are made in the vicinity of Mario, making it more likely to jump for coins, hills, blocks, pits, and monsters. When Mario is jumping, the probability of pressing the virtual jump button is doubled. There always is a probability of 0.1 to stand still, and when monsters are near this probability is doubled. The speed button is pressed when no monsters are near or when Mario is jumping over a pit.

The Random() function returns a random decimal value between 0 and 1.
Algorithm 2 Example Agent: DecideNewAction()

\[ \text{jumpHesitation} = 0.95 \]

\[ \text{areaTiles} = \text{area to Mario’s right} \]

\[ \text{for all} \ t \in \text{areaTiles} \ \text{do} \]
\[ \quad \text{if} \ t \ \text{is a coin} \ \text{then} \]
\[ \quad \quad \text{jumpHesitation} *= 0.7 \]
\[ \quad \text{if} \ t \ \text{is not air or Mario} \ \text{then} \]
\[ \quad \quad \text{jumpHesitation} *= \text{right}/7 \]

\[ \text{isPit} = \text{false} \]
\[ \text{if} \ \text{there is a pit in front of Mario} \ \text{then} \]
\[ \quad \text{jumpHesitation} = 0 \]
\[ \quad \text{isPit} = \text{true} \]

\[ \text{areaMonsters} = \text{area around Mario} \]
\[ \text{monsterNear} = \text{false} \]
\[ \text{for all} \ \text{monsters in} \ \text{areaMonsters} \ \text{do} \]
\[ \quad \text{jumpHesitation} *= (\text{monster.x} - \text{mario.x} + 2)/12 \]
\[ \quad \text{monsterNear} = \text{true} \]

\[ \text{if} \ \text{Mario is going upwards} \ \text{then} \]
\[ \quad \text{jumpHesitation} *= 0.5 \]

\[ \text{if} \ \text{walkHesitating} \ \text{then} \]
\[ \quad \text{if} \ (\text{not} \ \text{monsterNear}) \ \text{or} \ \text{Random()} > 0.8 \ \text{then} \]
\[ \quad \quad \text{walkHesitating} = \text{false} \]
\[ \quad \text{else if} \ \text{Random()} > 0.9 \ \text{then} \]
\[ \quad \quad \text{walkHesitating} = \text{false} \]
\[ \quad \text{else if} \ \text{monsterNear and} \ \text{Random()} > 0.8 \ \text{then} \]
\[ \quad \quad \text{walkHesitating} = \text{true} \]
\[ \quad \text{else if} \ \text{Random()} > 0.9 \ \text{then} \]
\[ \quad \quad \text{walkHesitating} = \text{true} \]

\[ \text{action}[0] = \text{walkHesitating} \ ? \ 0 : 1 \]
\[ \text{action}[1] = \text{Random()} > \text{jumpHesitation} \ ? \ 1 : 0 \]
\[ \text{action}[2] = (\text{isPit and} \ (\text{not} \ \text{monsterNear})) \ ? \ 1 : 0 \]

\[ \text{return} \ \text{action} \]
4.2 Fixed Policy Decision Tree

The main task of the agent is to maximize reward per step. Several different subtasks can be derived from the five reward types of the generalized Mario domain that help to accomplish this goal. These subtasks are reaching the end of the level, staying alive, collecting bonuses, killing enemies and making sure all is done using the least amount of steps as possible. The FPDT uses the hierarchical task decomposition approach, in which a decision tree prioritizes these subtasks depending on abstracted features of the state. The tree chooses between several macro actions, each providing a policy that is designed for a distinct task and situation. The algorithm uses a fixed policy on both high and low level decisions. The algorithm is largely deterministic, but it contains some minor random factors in low level decisions.

The FPDT does not consider which parameter setting is used for the environment. Instead, it assumes the parameter values that are most likely to be used, which are the values of instance 0. These settings for physics systems and reward systems stay true to the original game and the other nine training instances provided by the RL Competition adopted most of these values, as seen in Appendix A.

The FPDT uses knowledge from previous episodes to its advantage in the same way the Example Agent does (see Algorithm 1) with two minor differences. First, if the end of the level was reached in the previous episode, no more exploration is done at all to prohibit the action sequence from getting longer. The same action sequence will be executed. Second, the action sequence is repeated until the last 5 actions instead of 7. The optimal number of actions in the death zone varies between policies. Tests with the FPDT gave the best results for 5 actions, which is why this number was used.

It was a conscious choice to adopt the action of the previous episode instead of using the longest action sequence or the best rewarded action sequence. Using the longest action sequence would often lead to unnecessary extra length of the action sequence, which would result in lower reward. Using the best rewarded action sequence would slow down Mario’s progress through the level, favoring short episodes with high reward over longer episodes that bring Mario closer to the end of the level where the big positive reward awaits.

When the algorithm does not use an action from a previous episode, the new actions are decided upon using the decision tree shown in Figure 4.1. This decision tree contains eight task specific macro actions:

JumpPit Specifically designed to jump over any types of pits.
**Retreat** Mario walks to the left. This can be useful because, when facing large clusters of enemies, going back a bit will leave only the enemies on the front to deal with.

**JumpOnEnemies** Mario will try to jump on an enemy.

**JumpHill** Mario jumps up a hill if needed. This is mainly used to not get stuck.

**GetMushroomFlower** Mario will go in the direction of the closest mushroom or flower.

**GetQuestionmarks** Mario will go in the direction of the question mark block, and try to jump at the right time to hit it.

**GetCoins** Mario will go in the direction of the closest coin.

**HeuristicMove** This is the main method for moving around. It is based on heuristics that will steer Mario toward the end of the level.

For each of the 12 possible actions, a short path is predicted, representing Mario’s movement in the next steps if that action is executed. Each position in the path is given a heuristic value based on Mario’s predicted position, relative enemy positions and relative bonuses positions. Each heuristic value is weighted such that the weights decline as the position is more steps away in time. The value of an action is determined by the average weighted heuristic values of the positions in its path. The reason a short path is constructed is that, by looking forward only one step, sometimes it cannot be avoided that Mario jumps in an undesirable position.
Figure 4.1: Overview of the Fixed Policy Decision Tree.
4.3 Adaptive Decision Tree

Like the FPDT, the ADT uses the hierarchical task decomposition approach in which the problem is split up into separate subtasks. A decision tree selects which of several task specific macro actions will provide the policy depending on abstracted features of the state. The main difference with the FPDT is that instead of applying a fixed policy, the macro actions provide a policy that is a function of five properties of the environment: walking speed, running speed, jumping speed, falling speed, and bonus reward. These properties correlate with the parameterization of the MDP. The agent has no direct access to this parameterization, but can perform online measurements of properties of the environment. The measured properties are taken into account in both the high-level and low-level the decision-making process, enabling the ADT to adapt to different physics and reward systems. This should result in the ADT having a higher average reward per step than the FPDT when tested on different instances. For example, the algorithm will only make an effort to grab coins if the bonus reward is higher than zero. Another example: The distance to jump for coins, question mark blocks, mushrooms, flowers, enemies, pits and hills depends on the physical properties of the environment, decreasing the chance that Mario will miss a good item or die from a wrong jump.

The properties of the environment are learned online, meaning that they are learned while the policy is evaluated. Therefore, it is essential to have a fast and accurate way of measuring them. This is done in the following way:

**Walking speed** Mario’s speed in the $x$-direction after walking right four steps in a row.

**Running speed** Mario’s speed in the $x$-direction after running right four steps in a row.

**Jumping speed** At the start of an episode, if the jumping speed has not yet been measured, a jump is performed to measure this value, which is set to the maximum speed in the $y$-direction during the jump.

**Falling speed** Lowest speed value of Mario in the $y$-direction. This value is measured very accurately and fast because Mario always starts an episode at the top of the screen, falling down to the ground.

**Bonus reward** The reward for a step where a coin was picked up.

The properties of the environment start out with default values. These are later overwritten by the measured values. For example, bonus reward has a default value of $+1$,
to make sure Mario will at least pick up one coin to measure its real reward. If the real reward appears to be zero or less, the policy is adapted to that.

The physical properties of the environment are mainly used in linear formulas for distances. These formulas are based on intuitive notions. For example, the distance from a hill at which Mario should start to jump can be larger if Mario can move faster. Therefore, the maximum distance to jump for a hill was set to $w + r$ where $w$ is the walking speed property and $r$ is the running speed property of the environment. Each formula was tested and tweaked until an acceptable solution was achieved that seemed to perform well in all instances.

There were other properties of the environment that could be measured online, such as the rewards for killing an enemy, dying and reaching the finish line. However, there was no use for them in affecting the policy. Trying to kill an enemy was considered too hard and too risky to be successful. Instead, the policy of the ADT focuses on avoiding enemies altogether. Dying is almost never in the interest of maximizing reward and is always avoided by the ADT because it terminates the episode, prohibiting further exploration and additional rewards. The ADT always tries to get to the finish line because in all instances the rewards were configured such that reaching the finish line would be the best way to end an episode. The assumption was done, based on comments the by the developers of the RL Competition\textsuperscript{1}, that this would also be the case in yet unseen instances.

The ADT uses its previous action sequence in the same way the FPDT does, except that the action sequence is repeated until the last 6 actions. This gave the best results in initial tests. It decides on new actions using a decision tree, which is shown in Figure 4.2. This decision tree is based on the FPDT. It contains a hierarchy of high-level and low-level macro actions. The low-level macro actions are macro actions taken from the FPDT that are used within the high-level macro actions. They are shown to make clear how the FPDT evolved into the ADT. However, they have been changed to take into account the physical properties of the environment in their decisions.

The ADT contains less high-level macro actions than the FPDT due to three changes. First, dealing with enemies is made simpler. The FPDT contained three macro actions for dealing with enemies: Retreat, HeuristicMove and JumpOnEnemies. The ADT contains only AvoidEnemies, which does not try to kill enemies one by one by jumping on them, but tries to avoid them while going forward in the level. This is less risky and puts the focus more on getting to the end of the level where the big reward is. Second,
jumping for hills is made more flexible by making JumpHill is a low-level macro action that is used by multiple high-level macro actions. Jumping up a hill may be useful for moving fast, collecting bonus items as well as jumping pits. Therefore, the high level macro actions MoveFast, GetBonuses and JumpPit use this part of the code in their own context. Third, while the ADT deals with collecting bonus items in a similar manner as done in the FPDT, it is presented in the ADT in a hierarchical way which gives a better overview. First, it is checked if all constraints are satisfied to perform a bonus-greedy policy. If so, GetBonuses decides which of the three low-level macro actions GetMushroomFlower, GetQuestionMarks and GetCoins will provide the main direction, using JumpHill to make sure the terrain is handled smoothly.

The ADT contains five high-level macro actions, the last four of which are based on the macro actions of the FPDT:

**HelpMeasuringPhysics** This macro action is only executed if the situation is considered safe and one or more of the physical properties of the environment are not yet measured. It performs actions to be able to measure them, for example the jump at the beginning of the level. Note that the properties of the environment are also measured when this macro action is not active, it just helps speeding up the process.

**JumpPit** This macro action is specifically designed to jump over any types of pit, taking into account the physical properties of the environment.

**AvoidEnemies** This macro action provides a policy that tries to avoid enemies while moving forward in the level. This is done by computing a heuristic value for each of the possible next actions in a similar way as the HeuristicMove macro action of the FPDT did.

For each possible next action, a path is predicted that looks two steps ahead, taking into account the physical properties of the environment, current speed and landscape. Then linear interpolation is used to approximate the positions of Mario and the enemies for every tick in between those steps and these positions are added to the path. For every position in the path, a heuristic value is measured based on the relative enemy positions, taking into account the type of the enemy. For example, a position directly above a Goomba will get a higher value than a position directly above a Spikey because Mario cannot die from jumping on a Goomba but can from jumping on a Spikey. The heuristic value for a possible next action is the sum of the heuristic values of the individual positions in their path. On top of that, a mechanism is put into place that assures that action sequences in the
death zone are not repeated. This is done to make sure Mario will eventually find a way through a difficult pack of enemies.

**GetBonuses** This macro action provides a policy that tries to collect all valuable bonus items on the screen, ignoring pits and enemies. It is only activated if collecting mushrooms, flowers or coins will give an advantage, like a reward or an upgrade to Mario’s status (being small, big or fiery). The priority lies with mushrooms and flowers, then question mark blocks, and coins last. Executing this macro action, Mario will go directly towards the target bonus item, taking into account the physical properties of the environment. This is especially important for this particular macro action because it is essential to have the right distance from a question mark block when jumping for it while running in order to hit it with Mario’s head. In most instances, situations exist where Mario is unable to reach the target, which may cause him to go into a loop. To counter this, Mario’s positions in the last 25 steps are always saved to check if a loop is occurring. If this seems to be the case, GetBonuses cannot be executed in the next 5 steps.

**MoveFast** This macro action utilizes the policy to run right all the time, jumping for enemies, hills and sometimes randomly. It takes into account the physical properties of the environment.

Because the only non-deterministic macro action in the decision tree is MoveFast, a random action is performed with a probability of 0.05 to decrease the chance that Mario repeats a mistake twice. Every time Mario dies at approximately the same point, this probability is increased with 0.1. In other words, the more Mario dies at the same place, the larger the probability becomes of taking random actions instead of doing the decision tree policy. If Mario manages to get through the difficult part, this probability will be back to 0.05. This should help to avoid getting stuck at one point in a level for a large number of episodes.
Chapter 4. Methods

Figure 4.2: Overview of the Adaptive Decision Tree.
4.4 Hill Climber

The decisions made by the FPDT and ADT are mostly based on distances. The values of these distances were hard-coded or based on linear functions of properties of the environment. Both hand-coded values and functions were human estimations based on considerable amounts of testing. However, using machine learning these values could be further optimized, increasing the effectiveness of the policy. This is the main idea behind the HC. The HC is a direct policy search algorithm that uses the hill climbing method to search a policy space that is defined by 20 parameters. Each parameter represents a distance, count, or probability that is used to determine the policy. The HC returns bestParams, which is the best parameter vector it could find within a run.

The FPDT and ADT decompose the generalized Mario domain into more manageable subtasks. The solutions of each subtask are recombined by a decision tree to solve the whole problem. This decomposition demands prioritizing of subtasks, which decreases the flexibility of the algorithm. For example, both the FPDT and the ADT algorithms assign a higher priority to dealing with enemies than gathering bonus items. Most levels are filled with enemies, leaving little time to focus on collecting the bonus items. In other words, a choice has to be made: enemies or bonus items. The same goes for pits. A pit that is surrounded by enemies represents a problem that is very hard to solve using separate macro actions for jumping pits and handling enemies. In order to be able to take different aspects of the domain into account at once, the HC does not use task decomposition. Instead, an action is determined in a four step process. In the first step, the best direction to go to is decided: right or left? The other three steps base their decision on the this direction. They respectively decide whether to wait or not, to jump or not, and to speed or not. Their order does not matter for the decision-making process. Each of the four steps uses a couple of parameters in its decisions.

Combining a task decomposition approach with machine learning would have been a difficult task. Learning a switch network that replaces the current decision tree while keeping the current macro actions would not lead to better performance: The decision tree of the ADT is so simple that there is not much to optimize about it, while the macro actions are very complex and employ suboptimal policies. Applying layered learning in the generalized Mario domain is impossible because no special training environments are available for learning subtasks offline. For example, there are no levels with only pits, without enemies, to train JumpPit on. Learning only one macro action or doing a coevolutionary approach seem to be the most viable options in this domain for combining task decomposition with machine learning. However, keeping in mind the arguments given above on why decomposing the problem into subtasks cannot be done in an optimal way, it is questionable if an algorithm that learns subtasks could outperform an algorithm
that approaches the problem as a whole, like the HC does. Additionally, the subtasks of the decision trees are not fully independent. Learners assigned to different subtasks would need to learn largely overlapping behaviors while using largely overlapping state spaces. For example, when Mario is avoiding enemies, jumping pits, or collecting bonus items, the most important part remains to successfully predict Mario’s next position given the state and action, in order to prevent him from dying. By learning distances at which actions should be taken, the approach presented by the HC aims to learn a policy that prevents Mario from doing the wrong actions in certain situations. Because these distances should generally be the same in similar situations, the learned policy should generalize across different levels / different parts of one level, provided that it has had enough representative training examples.

Algorithm 3 describes the main method of the HC. First off, \( p \) parameter vectors are randomly generated and saved in a set of parameter vectors \( \Theta \). The initiation range for each parameter value can be found in Appendix B. One iteration features an evaluation process and a mutation process. In the evaluation process, all parameter vectors in \( \Theta \) are tested to find their average reward per step. The parameter vector with the highest average reward per step is saved in \( bestParams \). In the mutation process, \( p-1 \) copies are made of \( bestParams \). For each parameter in each copy, there is a chance that its value is slightly altered. Finally, all parameter vectors in \( \Theta \) except \( bestParams \) are replaced by the mutated copies of \( bestParams \). If a parameter vector attains the highest score on more than one iteration, it is evaluated by using the average score over all iterations it was evaluated in.

```
Algorithm 3: Hill Climber: Main method()

\[ \Theta = p \text{ randomly generated parameter settings} \]

for a fixed number of iterations do

\[ results = \text{empty list to save the results of the current iteration in} \]

for all \( \tilde{\theta} \in \Theta \) do

for all MDPs to be evaluated do

for \( t \) fitness evaluations do

FitnessEvaluation(\( \tilde{\theta} \))

\[ results_{\tilde{\theta}} = \text{average reward per step for } \tilde{\theta} \]

\[ bestParams = \arg\max_{\tilde{\theta} \in \Theta} results_{\tilde{\theta}} \]

mutate \( bestParams \) into a new pool of parameter vectors \( \Theta \) that includes \( bestParams \)

return \( bestParams \)
```
Each parameter vector $\vec{\theta} \in \Theta$ is evaluated on one or more MDPs. For every MDP, it is evaluated on $t$ fitness evaluations. The procedure of a fitness evaluation is described in Algorithm 4. Every fitness evaluation, the game is played for $s$ steps using the policy defined by parameter vector $\vec{\theta}$. This means the number of episodes can vary between fitness evaluations. Each episode after the first in a fitness evaluation, the policy of the HC is to repeat the action sequence from the previous episode until the last $\vec{\theta}_0$ actions, $\vec{\theta}_0$ being a parameter in $\vec{\theta}$ with range $[0, \infty)$. The ranges of all policy parameters can be found in Appendix B. Each fitness evaluation, the algorithm starts with an empty action sequence. The score of one parameter vector in one evaluation thus consists of the average reward per steps over a total of $t \cdot s$ steps wherein Mario has started from scratch $t$ times.

Algorithm 4 Hill Climber: Fitness Evaluation()

<table>
<thead>
<tr>
<th>input $\vec{\theta}$ = vector containing the current parameter setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{stepsThisFitnessEvaluation} = 0$</td>
</tr>
<tr>
<td>$\text{episodesThisFitnessEvaluation} = 0$</td>
</tr>
<tr>
<td>$\text{actionSequence} = \text{empty list}$</td>
</tr>
</tbody>
</table>

while $\text{stepsThisFitnessEvaluation} < s$ do

stepNumber = 0

for all steps in 1 episode do

if $\text{stepNumber} < \text{length(actionSequence)} - \vec{\theta}_0$ then

action = $\text{actionSequence}_{\text{stepNumber}}$

else

action = DecideNewAction($\vec{\theta}$, $\text{stepNumber}$, $\text{episodesThisFitnessEvaluation}$)

execute action

$\text{actionSequence}_{\text{stepNumber}} = \text{action}$

$\text{stepNumber} += 1$

if $\text{stepsThisFitnessEvaluation} + \text{stepNumber} \geq s$ then

break

$\text{stepsThisFitnessEvaluation} += \text{stepNumber}$

$\text{episodesThisFitnessEvaluation} += 1$

Algorithm 5 describes how the HC decides on a new action. The first four steps of each episode, the action $\{-1, 0, 1\}$ (running left) is always given. By doing this, Mario will always get stuck on the bottom left of the level, providing a steady starting position at the cost of four steps. This pays off because the method of repeating the previous action sequence relies on an environment that is exactly the same every episode. With
monsters occasionally starting in slightly different positions, the probability exists that a previously successful action sequence is unsuccessful in a new episode because Mario misses some coins and dies after a few steps. This probability must be minimized.

From the fifth step of an episode onward, the policy for new actions is determined by $\vec{\theta}$. First, a main direction is chosen in Direction() which determines to which side the algorithm looks. Then, given the main direction, in Hesitating() it is decided if Mario will move or stand still. Next, it is decided whether to press the jump and speeding buttons in Jumping() and Speeding() respectively. These functions are explained below. Finally, a check is done if Mario is in the death zone: the last five actions before he will die if all goes like the previous episode. Actions taken in the death zone will vary between different episodes, creating opportunities to find another way around problems. A random policy in the death zone would lead to large variance in the results, which means that a lot of fitness evaluations are needed to determine the true fitness of the policy. Doing many fitness evaluations would slow down the learning process. By deterministically rotating the death zone actions, a policy will use the same rotation in every fitness evaluation. Therefore, no extra fitness evaluations have to be done because no extra variance is brought in the results. A disadvantage of this method is that some combinations of actions are skipped, which makes this method slightly less flexible.

The remainder of this section explains in detail how the functions Direction(), Hesitating(), Jumping() and Speeding() determine a new action by using the policy parameters.

Algorithm 6 describes the Direction() function, which determines if Mario wants to move right or left. The default is to move to the right. If Mario’s horizontal progress during the last 25 steps was less than 10 tiles, a loop might have occurred and the direction will stay at value 1 (to the right). Otherwise, the parameters $\vec{\theta}_1$, $\vec{\theta}_2$ and $\vec{\theta}_3$ determine whether any observed flowers, mushrooms, coins or question mark blocks are worth it to go after.

The highest priority lies in pursuing flowers, because they decrease the risk that Mario will die, enable the shooting of fireballs, and may give a reward even if Mario is already fiery. Second priority is given to mushrooms, because they also decrease the risk that Mario will die and may give a reward. Next are coins, which are easy to pick up and may give reward. Question mark blocks can be difficult to activate depending on other parameter settings, which is why they have the lowest priority.

Three functions are used, which take as argument either an entity type, a tile type or a group of types, such as ‘enemy’ which can be any enemy. The Count() function outputs how many entities or tiles of that type are counted in the last observation. The Distance() function outputs the distance in tiles from Mario to the closest entity or
Algorithm 5 Hill Climber: DecideNewAction()

\[ \text{input } \vec{\theta} = \text{vector containing the current parameter setting} \]
\[ \text{input } \text{stepNumber} = \text{the number of steps this episode} \]
\[ \text{input } \text{episodesThisFitnessEvaluation} = \text{the number of episodes this fitness evaluation} \]

if \( \text{stepNumber} < 5 \) then
  \[ \text{direction} = -1 \]
  \[ \text{jump} = 0 \]
  \[ \text{speed} = 1 \]
else
  \[ \text{direction} = \text{Direction}(\vec{\theta}, \text{stepNumber}) \]
  \[ \text{direction} = \text{Hesitating}(\vec{\theta}, \text{direction}) \]
  \[ \text{jump} = \text{Jumping}(\vec{\theta}) \]
  \[ \text{speed} = \text{Speeding}(\vec{\theta}) \]

if Mario died previous episode and this is not the first episode of the restart then
  \[ \text{stepOfDeath} = \text{number of steps last episode} \]
else
  \[ \text{stepOfDeath} = -1 \]

if \( \text{stepOfDeath} - \text{stepNumber} > 0 \) and \( \text{stepOfDeath} - \text{stepNumber} < 5 \) then
  \[ \sigma = (\text{episodesThisFitnessEvaluation} + \text{stepNumber}) \]
  if \( \sigma \mod 3 = 0 \) then
    \[ \text{direction} = \sigma \mod 2 \]
  if \( \sigma \mod 3 = 1 \) then
    \[ \text{jump} = \sigma \mod 2 \]
  if \( \sigma \mod 3 = 2 \) then
    \[ \text{speed} = \sigma \mod 2 \]

\[ \text{action}[0] = \text{direction} \]
\[ \text{action}[1] = \text{jump} \]
\[ \text{action}[2] = \text{speed} \]

return \( \text{action} \)

tile of a type. For objects to the right of Mario, the Manhattan distance is returned. For objects to the left, the negative Manhattan distance is returned. The Distance\(_x()\) function returns the difference on the \( x \) direction, which has a negative value for objects to the left of Mario.

For each item, the choice to go after it is based on whether the number of enemies that are around is smaller than the respective parameter. This way, the parameter influences how much of a risk the policy wants to take to get the item. If an item is pursued, \( \text{direction} \) is set to -1 or 1 depending on which side of Mario the item is situated. This is described in the pseudocode using the notation of the conditional operator ‘?’.
the condition before the operator is true, the first value after the operator is returned. Otherwise, the second value is returned. A colon separates the first and second values.

**Algorithm 6** Hill Climber: Direction()

```
input $\vec{\theta} =$ vector containing the current parameter setting
input stepNumber = the step number of the current episode

direction = 1

if mario_{stepNumber}^{x} - mario_{stepNumber-25}^{x} \geq 10 then
    if Count(enemy) < $\vec{\theta}_1$ and Count(flower) > 0 then
        direction = Distance(flower) > 0 ? 1 : -1
    else if Count(enemy) < $\vec{\theta}_1$ and Count(mushroom) > 0 then
        direction = Distance(mushroom) > 0 ? 1 : -1
    else if Count(enemy) < $\vec{\theta}_2$ and Count(coin) > 0 then
        direction = Distance(coin) > 0 ? 1 : -1
    else if Count(enemy) < $\vec{\theta}_3$ and Count(questionmark) > 0 then
        direction = Distance(questionmark) > 0 ? 1 : -1

return direction
```

Algorithm 7 describes the Hesitating() function, which determines whether `direction` should be zero. This slows down or stops Mario and does not change the direction he is facing. This facing direction is captured in variable $d$, which is multiplied with the Distance() function to make sure policy parameters work independently of Mario’s facing direction, a positive value being in front of Mario and a negative value being behind Mario.

If Mario is too close to a Spikey or a Piranha Plant, `direction` remains unchanged, to minimize the chance that Mario will jump on a Spikey and make sure Mario will not stand still under a Piranha Plant. Parameter $\vec{\theta}_4$ determines how close Mario can be to a Spikey in front of him while not hesitating.

The argument ‘enemy common’ that is used for the Distance() function represents the group that consists of Goombas, Red Koopas and Green Koopas. These enemies can be killed by jumping on them. When such an enemy is close, it is a good strategy to slow down horizontal movement and jump a lot, minimizing the chance to run into them from the side and increasing the chance to jump on them. Parameter $\vec{\theta}_5$ determines the size of the zone around the enemy wherein the deceleration of horizontal movement is applied. Once within the range, parameter $\vec{\theta}_6$ is the probability that hesitation occurs per step.
When Piranha Plants come into vision, they move upward out of their pipe. Their movement speed and the time they stay up are such that, in most cases, they block several paths that Mario can take to move over the pipe. The pathfinding process is complicated by the fact that Mario can move underneath the Plant with the risk that the Plant goes down again and Mario is hurt. These risks and complexities can all be avoided by waiting until the Piranha Plant has gone back into its pipe before jumping on the pipe. However, this costs a few steps of time. The Hesitating() function determines if Mario will wait using three parameters. Parameter \( \vec{\theta}_7 \) determines the distance from the pipe at which Mario will wait. The variable \( \text{hesitatingTime} \) is a counter by which the time that Mario stands still is measured. Parameters \( \vec{\theta}_8 \) and \( \vec{\theta}_9 \) determine how long it will be. Parameters \( \vec{\theta}_7 \) and \( \vec{\theta}_8 \) cannot mutate lower than value 1 and \( \vec{\theta}_9 \) cannot mutate lower than value 2. This makes sure that Mario cannot stay still forever.

**Algorithm 7 Hill Climber: Hesitating()**

```plaintext
input \( \vec{\theta} \) = vector containing the current parameter setting
input direction

d = direction

if \((d \cdot \text{Distance(enemy spikey)}) > -3 \text{ and } d \cdot \text{Distance(enemy spikey)} < \vec{\theta}_4)\)
or \((|\text{Distancex(enemy plant)}| < 1)\) then
   return direction

if \(d \cdot \text{Distance(enemy common)} > -1\)
and \(d \cdot \text{Distance(enemy common)} < \vec{\theta}_5\) then
   direction = (Random() < \vec{\theta}_6) ? 0 : d

if \(d \cdot \text{Distancex(enemy plant)} > \vec{\theta}_7\)
and \(d \cdot \text{Distancex(enemy plant)} < \vec{\theta}_7 + 2\)
and \(\text{hesitatingTime} = 0\) then
   \(\text{hesitatingTime} = \vec{\theta}_8\)

if \(\text{hesitatingTime} > \vec{\theta}_9\) then
   direction = 0

\(\text{hesitatingTime} -= 1\)
if \(\text{hesitatingTime} < 0\) then
   \(\text{hesitatingTime} = 0\)

return direction
```

Algorithm 8 describes the Jumping() function which returns whether Mario will jump (1) or not (0). Mario may jump for pits, hills, coins, question mark blocks and enemies.
These all have another optimal distance for jumping, which is why they all get a different parameter assigned.

There are several different parameters for different enemies. One parameter is used for enemies in the ‘enemy common’ group because they should be dealt with in the same way. Spikeys, Piranha Plants and Shells have distinct behaviors which is why they all get a separate parameter.

Algorithm 8 Hill Climber: Jumping()

\[
\text{input } \vec{\theta} = \text{vector containing the current parameter setting}
\]

\[
d = \text{facing direction}
\]

\[
\text{if } (d \cdot \text{Distance}(\text{pit}) > 0 \text{ and } d \cdot \text{Distance}(\text{pit}) < \vec{\theta}_{10}) \\
\text{or } (d \cdot \text{Distance}(\text{hill}) > 0 \text{ and } d \cdot \text{Distance}(\text{hill}) < \vec{\theta}_{11}) \\
\text{or } (d \cdot \text{Distance}(\text{coin}) > 0 \text{ and } d \cdot \text{Distance}(\text{coin}) < \vec{\theta}_{12}) \\
\text{or } (d \cdot \text{Distance}(\text{questionmark}) > 0 \text{ and } d \cdot \text{Distance}(\text{questionmark}) < \vec{\theta}_{13}) \\
\text{or } (d \cdot \text{Distance}(\text{enemy common}) > 0 \text{ and } d \cdot \text{Distance}(\text{enemy common}) < \vec{\theta}_{14}) \\
\text{or } (d \cdot \text{Distance}(\text{enemy spikey}) > 0 \text{ and } d \cdot \text{Distance}(\text{enemy spikey}) < \vec{\theta}_{15}) \\
\text{or } (d \cdot \text{Distance}(\text{enemy plant}) > 0 \text{ and } d \cdot \text{Distance}(\text{enemy plant}) < \vec{\theta}_{16}) \\
\text{or } (d \cdot \text{Distance}(\text{enemy shell}) > 0 \text{ and } d \cdot \text{Distance}(\text{enemy shell}) < \vec{\theta}_{17}) \text{ then}
\]

\[
\text{return } 1
\]

\[
\text{else}
\]

\[
\text{return } 0
\]

Algorithm 9 describes the Speeding() function which returns whether Mario will run (1) or not (0). When Mario is in fiery mode, it also determines if he will shoot or not. However, this is quite rare and the algorithm does not consider this scenario.

There are two parameters involved. Parameter $\vec{\theta}_{18}$ determines how much speeding is used while jumping over pits. Parameter $\vec{\theta}_{19}$ determines how far Mario should be from enemies to start running. In general, it is good to go through the level as fast as possible because fewer steps per episode will result in higher average reward per step. However, dealing with enemies often requires slow movement. Also, it is easier to learn good jumping parameters when Mario’s velocity is stable.
Algorithm 9 Hill Climber: Speeding()

**input** \( \vec{\theta} \) = vector containing the current parameter setting

\( d \) = facing direction

if \( (d \cdot \text{Distance}(\text{pit}) > 0 \text{ and } d \cdot \text{Distance}(\text{pit}) < \bar{\theta}_{18}) \)

or \( (|\text{Distance(enemy)}| > \bar{\theta}_{19}) \) then

return 1

else

return 0
4.5 Adaptive Hill Climber

The AHC is based on the HC algorithm, but like the ADT it takes into account properties of the environment in order to be able to adapt its policy to different environments. It is a direct policy search algorithm that uses the hill climbing method to search a policy space. This policy space is defined by the same 20 parameters the HC uses. The AHC adapts to environments in a different way than the ADT. Instead of using hand-coded formulas based on intuitive notions, the AHC learns a different set of parameters for each group of environments with the same type of physical characteristics. This automatic optimization means the AHC should eventually be more accurate than the ADT. It should also be more accurate than the HC algorithm because parameters do not have to be a compromise between differing physics systems. The AHC returns bestParams, which is the best parameter vector it could find within a run.

The AHC is designed to adapt to different classes of instances. Instances of the same class have roughly the same physical characteristics. The algorithm learns a different policy for each class of instances. To do this, a parameter matrix \( \theta \) is used that has \( c \) rows and 20 columns, where \( c \) is the current number of classes of instances. The notation \( \theta_{\psi,*} \) will be used to indicate the row in \( \theta \) that holds the 20 parameters for the class of instances with characteristics \( \psi \). These parameters are \( \theta_{\psi,0} \) to \( \theta_{\psi,19} \). In the beginning of each fitness evaluation, the physical characteristics of the current instance are measured. The policy that is used in the fitness evaluation is defined by parameters \( \theta_{\psi,*} \), where \( \psi \) is the type of physical characteristics that is most similar to those of the current instance.

Algorithm 10 describes the main method for the AHC. The variable \( \Psi \) is an empty set of types of physical characteristics, each type defining a class of instances. The variable \( \Theta \) is a set of parameter matrices. It is initiated with \( p \) empty parameter matrices. Parameters will be added to them as classes of instances are discovered.

In the evaluation process, all parameter matrices in \( \Theta \) are tested to find their average reward per step. Each parameter matrix is evaluated on a set of MDPs that is ordered randomly. The parameter settings of the MDPs are unknown. For each instance, \( t \) fitness evaluations are done. During a fitness evaluation, it is checked which class of instances the current instance belongs to. If it is the case that no classes are available, or no classes have similar physical characteristics, a new class defined by \( \psi \) will be introduced along with the necessary parameters \( \theta_{\psi,*} \). This is described in more detail below. Each fitness evaluation returns type of physical characteristics \( \psi \) that was used. The averaged reward per step for \( \theta \) during the fitness evaluations of MDP \( m \) is saved in \( \text{results}_m^{\theta} \). The type of physical characteristics returned by the last fitness evaluation of each MDP
is saved as $m_{\psi}$. It is assumed the other fitness evaluations returned the same type of physical characteristics.

When all parameter matrices have been evaluated on all instances, the parameter matrix with the highest average reward per step over all instances is saved in $bestParams$. Then it is checked whether any other parameter matrices performed better on specific classes of instances. If so, $bestParams$ will receive the parameter settings for that specific class. If the evaluated parameter matrix was already used previously, its results are averaged with its previous results like in the HC. Additionally, if a part of the evaluated parameter matrix was already used previously, its results are also averaged with previously acquired results.

In the mutation process, $p - 1$ copies are made of $bestParams$. Second, each parameter in each copy has a probability to be slightly altered. Finally, the parameter matrices in $\Theta$ are replaced. One of the replacements is $bestParams$, the other $p - 1$ replacements are the mutated copies of $bestParams$.

The procedure of a fitness evaluation is described in Algorithm 11, which is similar to the fitness evaluation of the HC (Algorithm 4). The game is played for $s$ steps using a policy defined by parameter matrix $\theta$. However, depending on the class of instances, a different row of $\theta$ will be used. Therefore, it is essential to quickly determine the class of instances that the current MDP belongs to.

The physical characteristics of the instance are measured by Mario’s jumping speed in the beginning of the episode. Using only one property of the environment, the current MDP can be classified quickly. More importantly, it is enough to split the 10 training instances into several classes with distinct physical characteristics. Like the HC algorithm, Mario always runs left $\{-1, 0, 1\}$ during the first four steps of each episode, to make sure the starting position is fixed. In the AHC, Mario performs a jump $\{1, 1, 0\}$ at $stepNumber = 5$ and $stepNumber = 6$ each episode, enabling an early measuring of the jumping speed in the instance. The speed in the $y$-direction at $stepNumber = 7$ during the first episode of each fitness evaluation determines the physical characteristics of the current MDP, $\psi_{current}$.

If these characteristics differ too much from existing definitions of classes of instances, a new instance class will be created. For this to happen, the Euclidian distance between most similar class of instances $\psi_{closest} \in \Psi$ and the current physical characteristics $\psi_{current}$ must be higher than threshold $threshold$. If this is the case, $\psi_{current}$ is added to $\Psi$, and a new row are added to each parameter matrix. These parameters will have randomly generated values. The policy during the rest of the fitness evaluation will be
Algorithm 10 Adaptive Hill Climber: Main method()

Ψ = empty
Θ = set of $p$ empty parameter matrices
$mdps = \text{set of MDPs to be evaluated, ordered randomly, with unknown parameters}$

for a fixed number of iterations do

$\text{results} = \text{empty list to save the results of the current iteration in}$

for all $m \in mdps$ do

for all $\theta \in \Theta$ do

for $t$ fitness evaluations do

$\psi, \Psi, \Theta = \text{FitnessEvaluation}(\theta, \Theta, \Psi)$

// in FitnessEvaluation, $\Psi$ and $\Theta$ are filled depending on the physical characteristics of $m$

update $results^m_\theta$ to be average reward per step for $\theta$ in $m$

$m_\psi = \psi$

$bestParams = \arg\max_{\theta \in \Theta} \sum_{m \in mdps} results^m_\theta$

for all $\theta \in \Theta$ do

for all $\psi \in \Psi$ do

$M = \text{set of MDPs } m \in mdps \text{ for which } m_\psi = \psi$

if \[
\left( \sum_{m \in M} results^m_\theta \right) > \left( \sum_{m \in M} results^m_{bestParams} \right)
\] then

$bestParams^\psi = \theta_{\psi,*}$

mutate $bestParams$ into a new pool of parameter matrices $\Theta$ that includes $bestParams$

return $bestParams$

---

determined by the 20 new parameters that were added to the current parameter matrix $\theta$.

After the first iteration, all instances will have been evaluated once, and all classes of instances will have been defined. The distance between $\psi_{\text{current}}$ and $\psi_{\text{closest}}$ will stay below threshold, and the policy during the fitness evaluation will be determined by the row of the current parameter matrix that is assigned to $\psi_{\text{closest}}$.

Each fitness evaluation, the algorithm starts with an empty action sequence. Each episode after the first in a fitness evaluation, the action sequence of the previous episode will be repeated until the last $\theta_{\psi,0}$ actions. The policy for new actions is determined by parameters $\theta_{\psi,*}$ in the function DecideNewAction(). This function is exactly as the Hill Climbers DecideNewAction() (Algorithm 5), except for the jump in steps six and seven.

The function Direction() differs from the Hill Climber’s Direction() (Algorithm 6) in one way: in determining Mario’s direction, coins are only considered if the bonus reward is greater than zero. The bonus reward is the reward for the last step in which a coin was
Algorithm 11 Adaptive Hill Climber: FitnessEvaluation()

input \( \vec{\theta} = \) the current parameter vector
input \( \Psi = \) vector containing all known classes of instances

\[ \text{stepsThisFitnessEvaluation} = 0 \]
\[ \text{episodesThisFitnessEvaluation} = 0 \]
\[ \text{actionSequence} = \text{empty list} \]

while \( \text{stepsThisFitnessEvaluation} < s \) do
    \( \text{stepNumber} = 0 \)
    for all steps in 1 episode do
        if \( \text{episodesThisFitnessEvaluation} = 0 \) and \( \text{stepNumber} = 7 \) then
            \( \psi_{\text{current}} = \) the physical characteristics of the current instance
            \( \psi_{\text{closest}} = \arg\min_{\psi \in \Psi} \text{EuclidianDistance}(\psi_{\text{current}}, \psi) \)
            if \( \text{EuclidianDistance}(\psi_{\text{current}}, \psi_{\text{closest}}) < \text{threshold} \) then
                \( \psi = \psi_{\text{closest}} \)
            else
                \( \psi = \psi_{\text{current}} \)
                add \( \psi \) to \( \Psi \)
                \( \forall_{\theta \in \Theta} \) add a new row \( \theta_{\psi,*} \) to \( \theta \) with randomly generated values
        \end{if}
        \( \theta_{\psi,*} = \) the row in \( \theta \) used for the current class of instances \( \psi \)
        if \( \text{stepNumber} < \text{length(actionSequence)} - \theta_{\psi,0} \) then
            \( \text{action} = \text{actionSequence}_{\text{stepNumber}} \)
        else
            \( \text{action} = \text{DecideNewAction}(\theta_{\psi,*}, \text{stepNumber}, \text{episodesThisFitnessEvaluation}) \)
        execute \( \text{action} \)
        \( \text{actionSequence}_{\text{stepNumber}} = \text{action} \)
        \( \text{stepNumber}++ \)
        if \( \text{stepsThisFitnessEvaluation} + \text{stepNumber} \geq s \) then
            break
        end
        \( \text{stepsThisFitnessEvaluation} += \text{stepNumber} \)
        \( \text{episodesThisFitnessEvaluation}++ \)
    end
return \( \psi \)
picked up. Coins have no function other than giving reward. Mushrooms and flowers are usually lucrative to get, because they provide protection from dying. Therefore, mushrooms, flowers and question mark blocks are considered in the same way as in the Hill Climbers Direction() function.

The functions Hesitating(), Jumping() and Speeding() are identical in the Hill Climber and the Adaptive Hill Climber.

This chapter has described five agents with different approaches. The reasoning behind each approach has been explained, motivating why the agent theoretically should perform better than previously described agents. The example agent should provide a baseline of how well a simple stochastic policy performs in the generalized Mario domain. The FPDT and ADT are much more deterministic and should give an idea of how well a carefully thought out a hand-coded policy can perform. The HC and AHC can improve themselves by learning policy parameters, which is an advantage over the other algorithms. Two of the algorithms, the ADT and AHC, use an adaptive strategy that should be more robust to differences between environments. In the next chapter two experiments are described that expose the strengths and weaknesses of each approach.
Chapter 5

Experiments

This chapter describes and analyzes two experiments that feature the agents presented in Chapter 4. The main goal of these experiments is to make a comparison between the different approaches, emphasizing their ability to adapt to different instances. Additionally, the last section of the chapter compares the HC to the algorithm that won the RL Competition. The expected outcome of these comparisons is specified in the following three hypotheses:

- The algorithms with the task decomposition approach FPDT and ADT should obtain a higher reward than the example agent.
- The offline learning algorithms HC and AHC should eventually outperform the non-learning example agent, FPDT and ADT.
- The adaptive agents ADT and AHC should be more robust to different environments than the other algorithms, resulting in relatively higher scores than the other algorithms in experiments with multiple MDPs.

A short summary of the reasons for these hypotheses given in Chapter 4: The example agent should have the lowest score because of its simple stochastic policy. The FPDT should perform better because it has task specific deterministic macro actions that have been tested to perform well on their task. The ADT should attain higher scores than the FPDT because it adapts to different parameterizations, and takes less risk by avoiding enemies instead of killing them. It should perform relatively better than non-adaptive agents in experiments featuring multiple MDPs, because its policy is a function of the properties of the environment. The hill climbing algorithms should eventually outperform the non-learning algorithms because they continue to tweak their parameter settings, opposed to the fixed parameters used in the non-learning algorithms
that are established by human intuition. The AHC should obtain the highest score in experiments with multiple MDPs, because it learns a different policy for different classes of instances.

In order to test if the adaptive agents ADT and AHC are indeed more robust to different environments than the other algorithms, two experiments were set up: a single-instance experiment and a multi-instance experiment. Apart from the instances used in the experiments, the two experiments used exactly the same parameter settings. For the single instance, instance 0 was taken. The multi-instance experiment featured all 10 instances provided by the RL Competition software. Both experiments are done using level seed 121 with difficulty 4, level type 0. Level 121 was picked arbitrarily. Difficulty 4 was chosen because it seemed to offer a reasonably solvable problem that is not too easy and not too hard. The level type does not influence the MDP in a relevant way which is why one type was used. The main reason for learning only on one level and one difficulty setting is the computational time it takes to do these experiments. Each iteration of the algorithm, different policies are evaluated. In order to get accurate results, policies have to be tested thoroughly. To be able to make an evaluation of a policy, different fitness evaluations need to be done on various instances. Each fitness evaluation, multiple episodes need to be run. All this takes a lot of time. Experiments were run with 198 steps per second. This means, using the settings of the experiments described in this chapter, that an evaluation of 1 parameter matrix on 1 instance in the AHC takes about 61 seconds. Doing one full iteration for 10 instances took 1 hour and 36 seconds. In both experiments the algorithms were run for 20 hours, which resulted in 197 iterations for 1 instance and 19 iterations for 10 instances.

While each level is different, this difference mainly lies in the order of the series of challenges that the player has to conquer. Picking more than one level would not have made much difference in the challenges themselves, and would be more of the same type of challenges in a different order. This is supported by the fact that level generation is based on welding together idiomatic level parts. Although using more than one level would make the problem harder, the tests on one level should give a good indication on how efficient the different algorithms are in learning the Mario domain. This also depends on how well learned policies can be used in other levels. A small experiment was done to get insight in the generalization properties of learned policies of the hill climber algorithms. In the generalized Mario domain, this is very important as there are a huge amount of levels and it was not announced in advance which levels would be used for the final testing phase. Figure 5.1 provides empirical evidence that the HC is capable of learning policies that generalize between different levels. The $x$-axis shows the number of iterations and the $y$-axis shows the average reward per step for bestParams in that iteration. The blue line displays the results for the HC while learning a control policy
Figure 5.1: Generalization properties of the HC across levels. The blue line is a training phase in which the algorithm learns a policy for 30 levels. After every iteration, this policy is tested in a test phase on 30 other levels. This test phase is the thin black line. Both lines are the average of 25 runs, with settings $p = 5$, $t = 4$, $s = 800$, parameter mutation probability = 0.5, maximum mutation range = 2. On levels 0 to 29, all with level type 0, difficulty 4, instance 0. This is the training phase. After every iteration, this policy is tested in a test phase on levels 60 to 99 with the same type, difficulty and instance. The results of the test phase are displayed by the thin black line. Information from the test phase results was not used for learning. Both lines are the average of 25 runs, with settings $p = 5$ policies evaluated per iteration, $t = 4$ fitness evaluations per evaluation and $s = 800$ steps per fitness evaluation, parameter mutation probability = 0.5, maximum mutation range = 2. These were the best settings after a few initial tests. The graph shows that, while the HC is improving its policy based on the training phase, the testing phase scores are also rising. This proves that the learned policy generalizes to the levels of the test phase. As the AHC only differs from the HC in its adaptation mechanism to classes of instances, its generalization properties should be roughly the same.

5.1 Single-instance experiment

The configurations used in the single-instance experiment gave the best result for the hill climbing algorithms after extensive testing. Each algorithm performed 25 runs, with
settings $t = 6$ and $s = 2000$. For the hill climbing algorithms, $p = 5$ parameter vectors were evaluated per iteration. The mutation settings used for the HC and AHC were the same. The probability to change a policy parameter was set to 0.5. The maximum mutation step size was 2, allowing parameters to change by a random value in the range $[-2, 2]$. To help dealing with local maxima, the maximum mutation step size was increased by 0.1 for each iteration that the current best parameter setting remained the best. For the AHC, the threshold used for separating classes of instances threshold was set to 5.

Figure 5.2 shows the course of a fitness evaluation of the HC algorithm after 20 hours of learning. Two graphs are shown that are generated by the same fitness evaluation. On the $x$-axis of both graphs, the number of steps is displayed, which goes from 1 to 2000. On the $y$-axis, the left graph shows the cumulative reward, and the right graph shows the average reward per step. The fitness evaluation was run on instance 0, level type 121, difficulty 4 and level type 0.

Looking at the left graph, a small ‘stairway’ can be seen on the left. This is caused by Mario dying early in the first eight episodes, receiving -10 reward for every death. However, from episode nine, which starts at step number 198, Mario suddenly runs the whole way through the level, receiving the +100 points of reward at the finish line. From that point on, the last action sequence is repeated, which is 164 steps long in this case. During every episode, a small reward is collected by killing enemies and collecting coins, but the most important reward is the reward for finishing the level. The graph makes clear how finishing the level as fast as possible is very important for stacking up this +100 reward in this instance.

The right graph shows a curve, which is caused by dividing the cumulative reward by the number of steps done. As the number of steps increases, the curve stabilizes and becomes an asymptotic function, due to the fact that from step 198 the same episode is repeated. The reason for choosing 2000 steps per fitness evaluation is that at this point the curve already gives a good indication of its point of stabilization. In order to mimic the RL Competition, fitness evaluations of 100,000 steps would be needed. By taking only 2000 steps the value of each policy parameter setting in average reward per step will approximate the value of each policy parameter setting over 100,000 steps.

Doing several fitness evaluations of the same policy will often lead to different results in terms of average reward per step. In other words, the fitness function used for evaluation of policies is noisy. There are two reasons for this. The main reason is that the policies often contain random elements. By doing one action randomly different, the whole rest of an episode will be different, often resulting in a completely different score. The other reason is that there are small variations in starting positions of entities which sometimes
Figure 5.2: A fitness evaluation of 2000 steps for the HC. Both graphs are from the same fitness evaluation. Left: Cumulative reward. Right: Average reward per step. The fitness evaluation was done on instance 0, level seed 121, difficulty 4, level type 0.

may lead to another outcome of the game. Every iteration, the parameter setting with the highest score is used to generate the population of $p = 5$ parameter settings for the next iteration. The highest scoring parameter setting will often have a score that is at the top-end of its variance. Both algorithms systematically select the top-end of the variance of the scores, not knowing if high scores are caused by noise or by factors that are relevant to the learning process. The scores used for learning could significantly overestimate the quality of the generation champion, because the guy selected may have just had a lucky evaluation, without actually being any better. This noise influences the learning process, which is called overfitting. In order to visualize the amount of overfitting, a test phase was added at the end of each iteration of the HC and AHC algorithms. In this test phase, the winning parameter vector of the current iteration $\text{bestParams}$ is evaluated for $t$ fitness evaluations on all instances in $\text{instances}$ to test for a second time what its score is. Information from the test phases was not used for the learning processes. Without overfitting, the score of the test phase should on average be the same as the initial evaluation score. However, if the evaluation score goes up while the test phase stays low, this means the algorithm is only learning random noise. Comparing these scores will provide insight in the amount of overfitting taking place.

Figure 5.3 shows the results for all algorithms in the single-instance experiment. Each line is the average of 25 runs. The $x$-axis shows the number of iterations and on the $y$-axis the average reward per step for $\text{bestParams}$ in that iteration is shown. To give a clear view, the graph is smoothed using uniform moving averages with a window of width 6. Table 5.1 lists the average score in the last iteration and the 95% confidence interval for that score.

Comparing the decision tree algorithms to the example agent, the decision trees perform
Figure 5.3: Results for the single-instance experiment, on instance 0, level 121, difficulty 1, level type 0: each line is the average of 25 runs learning 20 hours, with \( t = 6 \) fitness evaluations per evaluation and \( s = 2000 \) steps per fitness evaluation. For the HC and AHC, \( p = 5 \) parameter vectors were evaluated per iteration. The probability to mutate a policy parameter was 0.5, maximum mutation size 2. To give a clear view, the graph is smoothed using uniform moving averages with a window of width 6.

worse than expected and the example agent performs relatively better than expected. With an score of -0.174 average reward per step, the FPDT clearly is the worst performing agent. The example agent scores 0.005, and the ADT is slightly better than that, with a score of 0.014. The confidence intervals shown in Table 5.1 indicate that there is no significant difference between the example agent and the ADT, as the results overlap by variance. This empirical evidence indicates that the first hypothesis, which stated that the FPDT and the ADT would obtain a higher reward than the example agent, has failed for the single-instance experiment. To get an idea of why the example agent is surprisingly good, and what exactly went wrong with the decision trees, we looked at some games played by the agents.

The FPDT is largely deterministic, with some random factors in low level decisions. This gives the algorithm an advantage over the example agent as long as it faces situations its policy was tested on. This can be seen in its play style, as it moves around faster while paying more attention to collecting bonus items. However, when its policy fails at some part of the level, it will take a lot of trials to get through that part because there is a lower probability of changing the actions in the death zone. At difficulty 4, level 121 brings too much of these challenges to let the FPDT find a good action sequence within
2000 steps. In fitness evaluations of 2000 steps, the FPDT reaches the end of the level only half as much as the example agent, missing out on the large reward.

The ADT was based on the FPDT but featured several improvements, like taking less risks around enemies and having more varied policies in the death zone. This helped a lot, as the ADT performs much better than the FPDT. It does not repeatedly fail at the same point in the level. As a result, the ADT reaches the end of the level about as much as the example agent does. The slight benefit the ADT has over the example agent is that it dies less often. There are two reasons for this. First, the ADT has a strategy to avoid enemies. Because of this the agent kills less enemies, but it is safer and results in less deaths for Mario. Second, the ADT is able to move backwards. Therefore it may sometimes go back and forth a bit, wandering around while doing nothing useful. This results in more steps per episode, which means less episodes per fitness evaluation, which means that Mario dies less often. This was unintended but still surprisingly effective.

Both algorithms have a low priority on collecting coins, and there does not seem to be a huge difference in the amount of coins collected by the agents. With tenfold the amount of code trying to steer Mario in the right direction, the performance of the ADT should have been much higher than the score of the example agent. This is a clear indication of how difficult it is to hard-code an agent in this domain.

The strategy of the example agent works so good because it is balanced very well. It collects coins, kills enemies, and manages to get further in the level using a stochastic policy. If the algorithm was altered to improve one of these goals, some of its flexibility in the death zones would have to be sacrificed. It is this flexibility, combined with only going in one direction, that makes the algorithm very effective. The following examples show that improving the example agent on one of these goals is non-trivial. If the example agent was rewritten to put more priority on collecting coins, it would sometimes have to move toward enemies that guard coins, which means dieing more often. If the example agent was rewritten to hesitate less often in order to finish the level faster, it would collect less coins and less kills because it would just jump over coins and enemies. If the algorithm was altered to go back for missed items, it would reach the finish line less often. Without using machine learning, it is hard to design a better algorithm because it requires lots of testing which would take too much time to do manually. Therefore the use of machine learning is essential in the generalized Mario domain.

The hill climbers start around the same point as the example agent and the ADT, but a learning curve is visible that rises significantly higher within a few iterations. The main difference between the HC and the AHC lies in learning over different instances. Learning only on one instance, the HC and AHC are nearly the same algorithms and
their performances are comparable. After 197 iterations the HC scores 0.454 and the AHC scores 0.450. For the HC and AHC, the test phases are shown in thin lines. Here, the HC scores 0.422 and the AHC scores 0.415 after 197 iterations, which is lower than their training phase results for both algorithms. This means that the algorithms are overfitting. Most importantly though, the test phase results prove that the algorithms actually learn valuable information. The overfitting could be reduced by doing more fitness evaluations per parameter setting per iteration. However, this would slow down the learning process. The cost of reducing the overfitting outweighs the benefit, because adding more fitness evaluations did not result in higher scores within the 20 hours of the experiment. Looking at Table 5.1, the confidence intervals of the hill climber algorithms are larger than the confidence intervals of the other algorithms. This means that in some of the 25 runs, the algorithms performed a lot better than the curve shown in Figure 5.3, while other runs performed a lot worse. As discussed in section 2.2, unless the search space is convex, hill climbing algorithms are only able to find local maxima. The search space defined by the parameters of the hill climbers seems to be filled with local maxima. Although the mutation size was increased slightly for each iteration that the current best parameter setting remained the best, some runs did get stuck in low lying local maxima. Despite this, the large variance does not undercut the conclusion that HC and AHC perform significantly better than the other algorithms of the comparison. This means the hypothesis that the hill climbing algorithms should eventually outperform the non-learning algorithms is true for the single-instance experiment.

### 5.2 Multi-instance experiment

The multi-instance experiment was done using the same settings as the single-instance experiment, except for the fact that all algorithms were tested on 10 instances. Figure 5.4 shows the results for the experiment with 10 instances, with iterations on the x-axis and on the y-axis the average reward per step for bestParams for each iteration. Each iteration cost a factor 10 more time than in the single-instance experiment, which is why

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Last iteration</th>
<th>95% conf. int.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Example Agent</td>
<td>0.005</td>
<td>± 0.019</td>
</tr>
<tr>
<td>Fixed Policy Decision Tree</td>
<td>-0.174</td>
<td>± 0.003</td>
</tr>
<tr>
<td>Adaptive Decision Tree</td>
<td>0.014</td>
<td>± 0.014</td>
</tr>
<tr>
<td>Hill Climber</td>
<td>0.454</td>
<td>± 0.040</td>
</tr>
<tr>
<td>Hill Climber test phase</td>
<td>0.422</td>
<td>± 0.052</td>
</tr>
<tr>
<td>Adaptive Hill Climber</td>
<td>0.450</td>
<td>± 0.045</td>
</tr>
<tr>
<td>Adaptive Hill Climber test phase</td>
<td>0.415</td>
<td>± 0.049</td>
</tr>
</tbody>
</table>

Table 5.1: Results for the single-instance experiment.
the \(x\)-axis is 10 times shorter. The result for the FPDT was left out to be able to zoom in to the interesting part. It is a flat line on -0.412. Table 5.2 lists the average score in the last iteration and the 95% confidence interval for that score. These results cannot be directly compared to the results of the single-instance experiment, because the reward system varies between instances. However, several conclusions can still be drawn.

The FPDT scored relatively lower than its performance in the single-instance experiment. Compared with the example agent and the ADT, the FPDT is more deterministic and has less adaptation capabilities. The stochastic example agent adapts easier to different environments than the hard-coded FPDT because it will eventually randomly do a good move. The ADT has both an adaptation mechanism and increased randomness in the death zone policy, which the FPDT lacks.

Compared to the example agent, the ADT performed relatively better than it did in the single-instance experiment. In the single-instance experiment, the ADT did not significantly outperform the example agent. Looking at Table 5.2, there is only an overlap of 0.002 between the confidence intervals of the results of the algorithms. This means that in a single fitness evaluation, there is a very slight chance that the example
agent performs better than the ADT, but the ADT will most of the time be superior. This indicates that the adaptation mechanism of the ADT works. Additionally, taking into account that the ADT is largely based on the FPDT, which scored relatively lower in the multi-instance experiment, it is remarkable that the ADT scores relatively higher. The other agent that adapts to different instances, the AHC, also has a relatively higher score than in the single-instance experiment. In the single-instance experiment, the AHC and the HC were practically the same. In the multi-instance experiment, the AHC adapts its policy to the classes of instances it finds, while the HC keeps on learning one policy for all instances. Despite the fact that the confidence intervals given in Table 5.2 are large, this does not undercut the conclusion that the AHC significantly outperforms the HC. This means the adaptation mechanism of the AHC works. The AHC divided the 10 instances of the experiment into four classes of instances. This means that it learned 80 parameters in total, while the HC learned 20 parameters. Learning more parameters does not always guarantee a better learning curve. If the parameters were not chosen correctly, it could have made the algorithm learn four times slower, and could even have resulted in the algorithm having no learning curve at all. The results show that the AHC learns faster than the HC, starting around the same point but quickly going up faster. This is consistent with the assumption that instances that have different physical characteristics have different ideal policy parameter settings. By learning a different policy for each class of instances, the AHC is able to outperform the HC within the same period of time. Like in the single-instance experiment, the test phase results are slightly lower than the training phase results. However they retain the fact that the AHC is the best algorithm, followed by the HC.

This section showed that both the AHC and the ADT have a relatively higher score than other algorithms when tested on multiple instances. Therefore the third hypothesis, which states that the adaptive agents should be more robust to different environments than the other algorithms, is true.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Last iteration</th>
<th>95% conf. int.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Example Agent</td>
<td>-0.093</td>
<td>± 0.009</td>
</tr>
<tr>
<td>Fixed Policy Decision Tree</td>
<td>-0.412</td>
<td>± 0.006</td>
</tr>
<tr>
<td>Adaptive Decision Tree</td>
<td>-0.076</td>
<td>± 0.010</td>
</tr>
<tr>
<td>Hill Climber</td>
<td>-0.013</td>
<td>± 0.019</td>
</tr>
<tr>
<td>Hill Climber test phase</td>
<td>-0.026</td>
<td>± 0.020</td>
</tr>
<tr>
<td>Adaptive Hill Climber</td>
<td>0.055</td>
<td>± 0.020</td>
</tr>
<tr>
<td>Adaptive Hill Climber test phase</td>
<td>0.031</td>
<td>± 0.021</td>
</tr>
</tbody>
</table>

Table 5.2: Results for the multi-instance experiment. These results cannot be directly compared to the results of the single-instance experiment, because the reward system varies between instances.
5.3 Comparison with the RL Competition winner

How would the most successful algorithm of this research, the AHC, have fared in the 2009 RL Competition? Would it have won? This section provides insight in the answer to this question by presenting four more experiments which involve the AHC and the algorithm that won the RL Competition 2009 [22]. A short description of this algorithm was given in section 2.4.

In the first experiment, the AHC is compared with the RL Competition winner on a MDP it has already learned on for 20 hours in the single-instance experiment: seed 121, difficulty 4, instance 0, type 0. Both algorithms were tested in one fitness evaluation of 100,000 steps, which is the number of steps used in the competition. The policy used for the AHC was provided by the parameter matrix from the last iteration of a successful run in the single-instance experiment. This parameter matrix scored 0.618 average reward per step over 2000 steps in fitness evaluations of the single-instance experiment. The results of the comparison between the the AHC and the RL Competition winner are shown in Figure 5.5, which features two graphs with on the $x$-axis the number of steps and on the $y$-axis the cumulative reward during the course of the fitness evaluation. Both graphs are from the same fitness evaluation. The left graph shows only the first 4000 steps of the experiment, while the right graph shows the whole experiment. The reason only one fitness evaluation was done is that for both algorithms, multiple trials resulted in roughly the same outcome.

Looking at the left graph, it is clear that the AHC has an advantage at the beginning of the experiment. This is sensible as the agent has already spent 20 hours of learning on this level. After 454 steps, the AHC reaches the finish line for the first time, and starts to collect reward fast by continuing to finish the level every episode. The RL Competition winner reaches the finish line after 2840 steps. At that time, the AHC has built up a nice lead. However, the number of steps that the RL Competition winner needs to finish a level is shorter than that of the AHC. As seen on the right graph, the RL Competition winner passes the AHC around the 10,000th step of the fitness evaluation. After 100,000 steps, the RL Competition winner has accumulated a total reward of around 92,842.82 which beats the 75,897.91 total reward collected by the AHC. However, the AHC still beats the RL Competition winner on the setting it was trained for, which is 2000 steps. At step number 2000, it accumulated 320% more reward than the RL Competition winner. Apparently, the fitness function used for learning the parameter matrix of the AHC was not ideal for this experiment. By evaluating policies by the average reward per step in 2000 steps, it becomes very important to be able to complete the level as early as possible. In experiments with 100,000 steps, the angle in which the cumulative reward line finally rises becomes much more important, as there is enough time to catch
up on other algorithms. For most instances, this angle is determined for a large part by the amount of steps per episode. In other words, it is most important to run as fast as possible to the finish line, without spending time to grab coins and kills. By changing the fitness function of the AHC, it might be able to learn a policy that beats the RL Competition winner. For example, a bonus could be given for lower amounts of steps in the penultimate episode if that episode was finished by reaching the end of the level. The reason for taking the penultimate episode is that the last episode is generally cut off at the end of a fitness evaluation. Learning with fitness evaluations of 100,000 steps is not recommended as it would take far too much time.

![Figure 5.5:](image)

**Figure 5.5:** Cumulative reward gathered by the AHC and the RL Competition winner in a fitness evaluation of 100,000 steps. Both graphs are from the same evaluation. Left: the first 4000 steps. Right: all 100,000 steps. Both graphs have the number of steps on the x-axis and the cumulative reward on the y-axis. The settings of the environment were level seed 121, difficulty 4, instance 0, type 0. The policy used by the AHC was provided by a successful run from the single-instance experiment, in which 25 runs of the AHC trained for 20 hours on the same level.

In the second experiment of this section, the AHC was compared to the RL Competition winner on a more difficult level. The difficulty was raised to 10, and the other environment parameters were set to level seed 121, instance 0, type 0. The AHC was trained on this level for 20 hours in 25 runs. The best run was selected to provide the AHC policy for this experiment. Like in the previous experiment, the agents were compared in one fitness evaluation of 100,000 steps. More than one evaluation was not needed because the results contained very little variance. The results are shown in Figure 5.6, which has the number of steps on the x-axis and the cumulative reward on the y-axis. Besides the results for the AHC and the RL Competition winner, the graph also displays the results for the AHC with randomized death zone. This is explained below. The AHC is able to outperform the RL Competition winner in the first 2000 steps. However, after 9900 steps, the RL Competition winner takes the lead. Over 100,000 steps, the RL
Competition winner accumulated 56,326.34 total reward which beats the 9653.70 total reward of the AHC.

Two interesting things can be seen in these results. First, the line of the RL Competition winner becomes steeper around 30,000 steps in the fitness evaluation. This is because up to that point, the agent occasionally reaches the finish line but keeps dying a lot too. After 30,000 steps, the agent does not die anymore. The RL Competition winner does repeat the previous action sequence until the last couple of steps. When an action sequence is found that brought Mario to the finish line, it is repeated, but without success. This failure is caused by variance in starting positions of entities. After 30,000 steps, the agent finally finds an action sequence that works independent of the variance in starting positions. Clearly it was very hard to find such a path in this level. Second, the almost horizontal line of the AHC indicates that it did not reach the finish line at all in this fitness evaluation. The positive reward it received came only from picking up coins and killing enemies. This is the main reason for the low total reward for the AHC. Mario dies at one point in the level and keeps repeating this, despite the mechanism that rotates actions in the death zone. In order to see what would happen if the policy of the AHC would get past this point, more tests were done using a random policy in the death
zone. Five fitness evaluations were done using a randomized death zone, which gave largely differing results depending on how fast the finish line was reached in the fitness evaluation. The results of these fitness evaluations are shown with the dotted lines. The uninterrupted black line shows the average of the five evaluations. The AHC with randomized death zone significantly outperforms the normal AHC in this experiment. As explained in section 4.4, the reason that the HC and the AHC use a deterministic mechanism to rotate actions in the death zone is to speed up the learning process. This experiment shows that in order to handle the highest difficulty levels, the AHC would need a better death zone mechanism. In the RL Competition, the AHC would not be able to train on the levels it will be evaluated on, because these are not announced. Even though the policy learned by the AHC does seem to generalize across levels, a better death zone mechanism would help to overcome unanticipated situations. With a death zone of 5 steps and 12 different actions, 60 action combinations are possible. The AHC currently does not evaluate all combinations, because waiting 60 episodes to find the ideal combination takes too much time. In order to quickly find a good combination, heuristics could be used, or a form of online learning. The power of the RL Competition winner comes from its online learning. By learning state-action pairs online, it will eventually find an action sequence that reaches the end of the level.

How would the RL Competition winner have performed in the single-instance and multi-instance experiments described in sections 5.1 and 5.2? Had the RL Competition winner been included in the single-instance experiment, it would have shown a very short learning curve that would go up very quickly. In the single-instance experiment one iteration consists of 12,000 steps. On the same level, the RL Competition winner finds its best policy after about 2840 steps, which means that from the second iteration on, it would be at its maximum score. This was confirmed by doing an extra experiment, using the settings of the single-instance experiment. The RL Competition winner scored an average reward per step of 0.936 with a 95% confidence level of ±0.003, which is significantly higher than the score of the test phase of the AHC: 0.415 average reward per step with a 95% confidence level of ±0.049. However, the AHC’s specialty lies in adaptation to different instances. The RL Competition winner learns an entirely new policy from scratch every time a new instance is loaded. Using the settings of the multi-instance experiment, the RL Competition winner obtained an average reward per step of 0.389, with a 95% confidence level of ±0.002. The score of the test phase of the AHC in this experiment was an average reward per step of 0.031 with a 95% confidence level of ±0.021. This means, that learning 10 instances from scratch over 2000 steps, the RL Competition winner significantly outperforms the AHC which has been training offline for 20 hours on the same problem. In the single-instance experiment, the learning curve of the AHC did not show much sign of improvement after 20 hours. However, in the multi-instance
experiment the line still seems to be going up. The AHC learns slower which can be explained by the fact that the AHC, having found four classes of instances among the 10 instances, has to learn four times as many parameters. Even though the score obtained by the AHC after 20 hours is probably not the maximum it can eventually reach, it is unlikely that it reaches a higher score than the RL Competition winner by training for a longer period.

This section has provided the insight that the AHC would not have won the RL Competition 2009. Even when evaluated on levels it was trained on for 20 hours, it could not outperform the RL Competition winner. Especially with higher difficulties, the algorithm is not able to learn a policy that is good enough. In an experiment involving multiple instances, the AHC also obtained a lower score. In order to win the RL Competition, the AHC would need a more flexible death zone mechanism. Also, the fitness function used for training a policy would have to take into account how well a policy would do over 100,000 steps instead of 2000 steps. The offline learning mechanism of the AHC does have the advantage that, at the start of a fitness evaluation, it immediately uses a policy that is better than starting policies of any other algorithm presented in this thesis.

Looking back on the three hypotheses specified at the beginning of this chapter, one failed, and two are true. The first hypothesis, which says that the algorithms with the task decomposition approach should obtain a higher reward than the example agent, failed because the results of the FPDT were lower. The results of the ADT were a disappointment because it did not outperform the example agent in the single-instance experiment. The second hypothesis is true, because the HC and AHC eventually outperformed the non-learning agents. The third hypothesis stated that the adaptive agents should be more robust to different environments than the other algorithms, resulting in relatively higher scores than the other algorithms in experiments with multiple MDPs. The ADT did obtain a relatively better result in the multi-instance experiment, even though its improvement was very little. The results of the AHC showed that it was able to outperform all other algorithms when tested on multiple MDPs. This makes the last hypothesis true.
Chapter 6

Discussion & Future Work

In this thesis, six different approaches to solving the generalized Mario domain have been analyzed and compared. Initially Mario may seem like a simple game, which can be solved by a short hard-coded rule-based system. However, the results of the FPDT and ADT algorithms made clear that without machine learning, it is very hard to even outperform a largely random agent like the example agent. Also, the results prove that the hill climbing technique is able to outperform hand-coded approaches.

The Mario domain seems not very suited for a hierarchical task decomposition approach. While the low results of the FPDT and ADT were mostly due to the fact that the algorithms did not use machine learning, there were lots of difficulties during development that indicated that the way the decision trees decomposed the domain into smaller problems was not ideal. Difficult choices had to be made about prioritization of subtasks. At higher difficulty levels it becomes harder to separate subtasks because bonus items, enemies and pits are more packed together at the same places. Learning one or more subtasks of the decision tree could increase the performance of the ADT, however the solution presented with the hill climbers is both easier and more effective. As explained in section 4.4, learners assigned to different subtasks would overlap both in behavior and state space. This raises the question if another decomposition would have been better. The decompositions used for the FPDT and ADT were developed with the assumption that the prediction of Mario’s next position given a state and an action would be trivial. After all, the domain is a deterministic MDP and Mario’s movement looks simple. However, it was very hard to hand-code a JumpPit macro action, and the final version still cannot take all different pits in one trial. Keeping this in mind, it could be beneficial for any approach to add a subtask for learning to predict the next position of Mario, or even the whole next state. In the hill climbing algorithms, the main task is also divided into subtasks. Instead of basing the decomposition on the reward system...
like the decision trees did, the subtasks are based on the different aspects of the action space: direction, hesitating, jumping and speeding. Also, in a way the AHC constructs a new macro action for every class of instances it can find. This worked very well in the experiments of Chapter 5. Currently the agent relies solely on jumping speed to distinguish between classes of instances. This works well for the training instances because it can be measured quickly and splits the instances into several classes with distinct physical characteristics. However, other properties like walking speed and falling speed could be added.

Looking at the behavior of some of the best parameter settings of the hill climbers from individual evolutionary runs on instance 0, the reward system seems to stimulate fast completion of the level. Bonus items in regions with enemies are often ignored. In regions with few enemies most bonus items are picked up. Killing enemies is clearly not a priority, which is logical because of the risk it brings. The AHC does not consider all aspects of the game. The algorithm could be expanded by adding extra policy parameters for handling winged enemies and determining when to shoot. Mario’s status is not taken into account as well. It could be beneficial to learn different parameter settings for different statuses. For example, when big, getting hurt is less of a problem than when Mario is small. All in all, within 20 hours the algorithms are able to learn a policy that is not optimal, but can complete a level of difficulty 4 within a few tries. On the highest difficulty the algorithm has problems to find a way through the level. One way to solve this is to give the algorithm more flexibility in the death zone. A stochastic death zone handling could have bad effects on the learning process. Therefore, a better solution would be to use a deterministic system that eventually tests all action combinations, starting with the most likely one. In order to quickly find a good action sequence in the death zone, heuristics could be used, or a form of online learning.

The results presented in section 5.3 made clear that the most successful one of the agents presented, the AHC, would not have been able to win the competition in its current form. However, there is room for improvement in the algorithm. How successful would the AHC have been in the RL Competition if it had a fitness function that works good for fitness evaluations of 100,000 steps, combined with better death zone handling? The AHC is mostly an offline learning algorithm, and its success would depend largely on the time spent on learning a parameter matrix that performs well on different MDPs. In order to develop a parameter matrix that performs well on new, unseen levels, the AHC should be trained on a large number of levels of varying difficulty. Initial experiments indicate that, when training on 30 levels, policies learned by the hill climber algorithms do generalize to other levels. More experiments need to be done to determine the optimal set of MDPs to train on in order to learn a policy that performs well on most MDPs. The success of the AHC in the RL Competition would also depend on the extent to
which the training instances differ from the instances used in the final test. The current
instance classification system, which only relies on Mario’s jump speed, works well for
the training instances but might be less effective in the final test, depending on the
parameterization of the MDPs.

The fact that the AHC is mainly an offline learning algorithm is both an advantage
and a disadvantage. The main advantage is that, by training the agent beforehand, the
agent will have a head start in the competition because it should already be prepared
for the MDP and will not have to try out different policies. However, using the settings
used in Chapter 5, the algorithm needs 60,000 steps in order to evaluate one parameter
matrix on one MDP, which takes more than 5 minutes. Depending on the generalization
capabilities of the algorithm, it could take weeks to train the agent sufficiently. This
is the main disadvantage. The presented approaches all featured a very simple form
of online learning: Repeating the previous action sequence. An interesting find is that
repeating the previous action sequence works better than repeating the best rewarded
action sequence. Doing more research with online learning would mean the algorithm
learns every step, instead of every 60,000 steps, which has the advantage that a policy
can be evaluated faster.

The research presented in this thesis has evaluated approaches that were not considered
before in any platform game domain. This chapter has discussed the advantages and
disadvantages of these approaches. Also, various ideas and alternatives for future re-
search were presented. Super Mario has only very recently become an AI benchmark,
and this research only scratches the surface of what is possible when it comes to tack-
ling the challenges of this domain. These challenges include the abstraction of the huge
state space, unknown MDP parameterizations, generalization over different levels, and
mastering the many possible interactions with the environment. The main goal of the
research presented in this thesis was to develop a learning algorithm that could adapt
to different instances. Empirical evidence has been presented in which is shown that the
adaptation mechanism of the AHC is a viable technique for this purpose. The AHC is
able to dramatically outperform the example agent. The AHC was trained on fitness
evaluations of 2000 steps, and on this setting it is able to beat the RL Competition
winner.
Appendix A

The Training Environments

Table A.1 contains the parameters of the 10 instances that were predefined in the RL-competition software.

Explanation of values:

**level width** The width of the level in tiles.

**max episode steps** An episode will be ended if it lasts more than this number of steps.

**reward finish** The reward given by reaching the finish line of the level.

**reward death** The reward given when Mario dies.

**reward step** This reward is given every step of the episode.

<table>
<thead>
<tr>
<th>Instance number</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>level width</td>
<td>320</td>
<td>320</td>
<td>320</td>
<td>150</td>
<td>150</td>
<td>320</td>
<td>320</td>
<td>320</td>
<td>620</td>
<td>320</td>
</tr>
<tr>
<td>max episode steps</td>
<td>1000</td>
<td>1000</td>
<td>1000</td>
<td>1000</td>
<td>1000</td>
<td>1000</td>
<td>1000</td>
<td>1000</td>
<td>2000</td>
<td>1000</td>
</tr>
<tr>
<td>reward finish</td>
<td>100</td>
<td>0</td>
<td>10</td>
<td>0</td>
<td>0</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>reward death</td>
<td>-10</td>
<td>0</td>
<td>-10</td>
<td>0</td>
<td>-10</td>
<td>-10</td>
<td>-10</td>
<td>-10</td>
<td>-10</td>
<td>-10</td>
</tr>
<tr>
<td>reward step</td>
<td>-0.01</td>
<td>-0.01</td>
<td>-0.01</td>
<td>-0.01</td>
<td>-0.01</td>
<td>-0.01</td>
<td>-0.01</td>
<td>-0.01</td>
<td>-0.01</td>
<td>-0.01</td>
</tr>
<tr>
<td>reward bonus</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>reward kill</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>speed walk</td>
<td>0.6</td>
<td>0.6</td>
<td>0.6</td>
<td>0.6</td>
<td>0.6</td>
<td>0.3</td>
<td>0.6</td>
<td>1.2</td>
<td>0.6</td>
<td>0.6</td>
</tr>
<tr>
<td>speed run</td>
<td>1.2</td>
<td>1.2</td>
<td>1.2</td>
<td>1.2</td>
<td>1.2</td>
<td>0.6</td>
<td>1.2</td>
<td>2.4</td>
<td>1.2</td>
<td>1.2</td>
</tr>
<tr>
<td>speed jump</td>
<td>-1.9</td>
<td>-1.9</td>
<td>-1.9</td>
<td>-1.9</td>
<td>-1.9</td>
<td>-1</td>
<td>-1.9</td>
<td>-3.9</td>
<td>-1.9</td>
<td>-1.9</td>
</tr>
<tr>
<td>speed jump sliding</td>
<td>-2</td>
<td>-2</td>
<td>-2</td>
<td>-2</td>
<td>-2</td>
<td>-1</td>
<td>-2</td>
<td>-2</td>
<td>-2</td>
<td>-2</td>
</tr>
<tr>
<td>accel gravity</td>
<td>0.82</td>
<td>0.82</td>
<td>0.82</td>
<td>0.82</td>
<td>0.82</td>
<td>0.42</td>
<td>0.82</td>
<td>0.92</td>
<td>0.82</td>
<td>0.42</td>
</tr>
<tr>
<td>jump time</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>15</td>
<td>7</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>jump time sliding</td>
<td>-6</td>
<td>-6</td>
<td>-6</td>
<td>-6</td>
<td>-6</td>
<td>-6</td>
<td>-6</td>
<td>-6</td>
<td>-6</td>
<td>-6</td>
</tr>
</tbody>
</table>

Table A.1: Variations in training data over the 10 instances.
reward bonus  The reward given by picking up a coin, mushroom or flower.

reward kill  The reward given when Mario kills an enemy.

speed walk  Determines Mario’s walking speed.

speed run  Determines Mario’s running speed.

speed jump  Determines Mario’s jumping speed.

speed jump sliding  Determines Mario’s jumping speed while sliding.

accel gravity  Determines Mario’s falling speed.

jump time  Influences the time length of a jump of Mario.

jump time sliding  Influences the time length of a jump of Mario while sliding.
Appendix B

Policy Parameter Ranges

Some policy parameters cannot mutate lower than a certain value. These values are shown in Table B.1. All policy parameters have no maximum value to which they can mutate. The values are the same for the HC and the AHC.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Minimum mutation value</th>
<th>Initiation range</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\vec{\theta}_0$</td>
<td>0</td>
<td>$[0, 10)$</td>
</tr>
<tr>
<td>$\vec{\theta}_1$</td>
<td>$-\infty$</td>
<td>$[0, 9)$</td>
</tr>
<tr>
<td>$\vec{\theta}_2$</td>
<td>$-\infty$</td>
<td>$[0, 9)$</td>
</tr>
<tr>
<td>$\vec{\theta}_3$</td>
<td>$-\infty$</td>
<td>$[0, 9)$</td>
</tr>
<tr>
<td>$\vec{\theta}_4$</td>
<td>$-\infty$</td>
<td>$[0, 10)$</td>
</tr>
<tr>
<td>$\vec{\theta}_5$</td>
<td>$-\infty$</td>
<td>$[0, 8)$</td>
</tr>
<tr>
<td>$\vec{\theta}_6$</td>
<td>$-\infty$</td>
<td>$[0, 8)$</td>
</tr>
<tr>
<td>$\vec{\theta}_7$</td>
<td>1</td>
<td>$[1, 9)$</td>
</tr>
<tr>
<td>$\vec{\theta}_8$</td>
<td>1</td>
<td>$[1, 10)$</td>
</tr>
<tr>
<td>$\vec{\theta}_9$</td>
<td>2</td>
<td>$[2, 5)$</td>
</tr>
<tr>
<td>$\vec{\theta}_{10}$</td>
<td>1</td>
<td>$[1, 10)$</td>
</tr>
<tr>
<td>$\vec{\theta}_{11}$</td>
<td>1</td>
<td>$[1, 10)$</td>
</tr>
<tr>
<td>$\vec{\theta}_{12}$</td>
<td>0</td>
<td>$[0, 10)$</td>
</tr>
<tr>
<td>$\vec{\theta}_{13}$</td>
<td>0</td>
<td>$[0, 7)$</td>
</tr>
<tr>
<td>$\vec{\theta}_{14}$</td>
<td>1</td>
<td>$[1, 10)$</td>
</tr>
<tr>
<td>$\vec{\theta}_{15}$</td>
<td>1</td>
<td>$[1, 10)$</td>
</tr>
<tr>
<td>$\vec{\theta}_{16}$</td>
<td>1</td>
<td>$[1, 10)$</td>
</tr>
<tr>
<td>$\vec{\theta}_{17}$</td>
<td>1</td>
<td>$[1, 10)$</td>
</tr>
<tr>
<td>$\vec{\theta}_{18}$</td>
<td>$-\infty$</td>
<td>$[0, 10)$</td>
</tr>
<tr>
<td>$\vec{\theta}_{19}$</td>
<td>$-\infty$</td>
<td>$[0, 10)$</td>
</tr>
</tbody>
</table>

Table B.1: Minimal values of policy parameters.

69
Bibliography


