Machine learning strategic game play for a first-person shooter video game

by

F. Huizinga

For the degree of Master of Science in Artificial Intelligence with the specialization of Intelligent Systems

in the
Faculty of Science
Informatics Institute

June 2011
“The question of whether a computer can think is no more interesting than the question of whether a submarine can swim.”

Dijkstra
The main goal of Artificial Intelligence (AI) in commercial video games is to provide a rich gaming experience for the end user. Providing this experience requires a careful balance between encoding prior knowledge, control by developers and autonomous, learning AI.

This thesis proposes a method that maintains a balance between these three objectives. The method proposed is a novel combination of Hierarchical Task Network (HTN) planning and Reinforcement Learning (RL). The HTN provides the structure to encode a variety of strategies by human developers, whereas RL allows for learning the best strategy against a fixed opponent. The method is empirically analyzed by applying it on the strategic level of Killzone 3 – a first-person shooter video game.

From the results we can conclude that the best strategy with respect to a fixed opponent indeed is learned and when a different opponent is introduced, the system has the ability to adapt its strategy accordingly.
Acknowledgments

This thesis is the final product of my time as intern researcher at Guerrilla. I would like to thank my supervisors Shimon Whiteson (Assistant professor at the University of Amsterdam) and Remco Straatman (Lead AI developer at Guerrilla). Furthermore I would like to thank all members of the AI team at Guerrilla for their technical support and constructive feedback and last but not least my family for their loving support.
## Contents

Abstract iii

Acknowledgments iv

List of Figures vii

1 Introduction 1
   1.1 Objective ........................................... 1
   1.2 Contributions ......................................... 1
   1.3 Environment ........................................... 2
   1.4 Scope ................................................ 2
   1.5 Overview ............................................ 2

2 Background 4
   2.1 Reinforcement Learning ................................. 4
      2.1.1 Markov Decision Process .......................... 4
      2.1.2 Value Functions ................................ 5
      2.1.3 Reinforcement Learning Algorithms ................. 6
      2.1.4 Action Selection Algorithms ...................... 6
      2.1.5 Contextual Bandit Problem ....................... 6
   2.2 Hierarchical Task Networks ......................... 7
      2.2.1 HTN-planner ...................................... 7
      2.2.2 Example ........................................... 7
      2.2.3 Formal Description ................................ 8
         2.2.3.1 State ....................................... 9
         2.2.3.2 Task ........................................ 9
         2.2.3.3 Operator .................................... 9
         2.2.3.4 Method ...................................... 10
         2.2.3.5 Plan ......................................... 10
         2.2.3.6 Planning Problem .............................. 10

3 Related Work 11
   3.1 Game Industry ......................................... 11
      3.1.1 F.E.A.R. ........................................... 11
         3.1.1.1 Goal-Oriented Action Planning ............... 11
      3.1.2 STRIPS ........................................... 12
List of Figures

2.1 HTN planning domain .............................................. 8
2.2 HTN initial state .................................................. 8
3.1 Hierarchical Finite State Machine ............................... 13
4.1 Killzone 3 ingame first-person view .............................. 18
4.2 Simplified design overview of the AI architecture ............. 19
4.3 Area capture example domain ................................... 21
5.1 HTN Graph .......................................................... 25
5.2 Strategy: Steamroll ............................................... 28
5.3 Strategy: Capture and Split ........................................ 29
5.4 Strategy: Divide and Conquer ..................................... 29
6.1 Corinth Highway (MP01) .......................................... 31
6.2 Pyrrhus Crater (MP02) ........................................... 31
6.3 Bilgarsk Boulevard (MP03) ......................................... 32
6.4 Method vs Baseline ................................................ 33
6.5 Comparison .......................................................... 33
6.6 HGH vs ISA ........................................................ 34
6.7 Adaptability .......................................................... 35
Chapter 1

Introduction

The main goal of Artificial Intelligence (AI) in commercial video games is to provide a rich gaming experience for the end user. Providing this experience requires a careful balance between encoding prior knowledge, control by developers and autonomous, learning AI.

1.1 Objective

This thesis proposes a method that maintains a balance between encoding prior knowledge, control by developers and autonomous, learning AI. The method proposed is a novel combination of Hierarchical Task Network (HTN) planning and Reinforcement Learning (RL). The HTN provides the structure to encode a variety of strategies, whereas RL allows for learning the best strategy against a fixed opponent. The method is empirically analyzed by applying it on the strategic level of the video game called Killzone 3 (KZ3). KZ3 is a first-person shooter (FPS), a genre of computer games where the player’s viewing perspective is shown through the eyes of their virtual character.

1.2 Contributions

In this thesis, we try to answer the following questions:

1. Can we encode good strategies into a hierarchical task network?
2. Is it possible to learn the best strategy with respect to a fixed opponent?
3. Is it possible to adapt to a different fixed opponent?
Chapter 1. Introduction

The first allows for a clear separation of domain language and problem solver as well as easy redesign of certain handcrafted strategies by the developer. The second and third enable a richer, more personal gaming experience. By building a learning mechanism on top of the HTN-Planner, the best (w.r.t. the current fixed opponent) handcrafted strategy can be learned while keeping control over the individual strategies, see chapter 5 for the method proposed.

1.3 Environment

The environment in which the proposed method is implemented and tested is a commercial first-person shooter called Killzone 3 developed by Guerrilla. KZ3 provides a good, real world test bed as its regarded as a typical, high quality FPS game throughout the world. See chapter 4 for a detailed description.

1.4 Scope

In a typical FPS game there are several game modes, each with different objectives that a team has to achieve to win the game. For the scope of this thesis, the implementation is applied to the game mode called Capture and Hold. However, it should be noted that the proposed method itself can be applied to all existing game modes.

The objective of Capture and Hold is to gain control over the key areas on the map that can be captured by either team. To gain control over an area, players must stand within the capture radius of the area and make sure that no enemy soldiers are within capture range of that same area during the capture process. If a team succeeds in keeping away the enemy long enough, the area will change ownership to the capturing team. Once an area is captured the team receives points for each time step the area is under their control. If the maximum amount of points is gained (defined by the game) the team wins or, alternatively, if the time limit is reached the team with the most points wins. When both teams have the same amount of points the game results in a draw.

1.5 Overview

Chapter 2 explains the background required for the proposed method, chapter 3 discusses related work from both the gaming industry and academic world. In chapter 4 the domain (Killzone 3) is explained in detail and chapter 5 introduces the proposed method.
Experiments and results are presented in chapter 6 and finally the conclusions, discussion and future work are given in chapter 7.
Chapter 2

Background

This chapter describes the theory of two well known fields within AI called Reinforcement Learning and Hierarchical Task Network Planning onto which the proposed method is based.

2.1 Reinforcement Learning

Reinforcement learning is a sub-area of machine learning concerned with how an agent ought to take actions in an environment so as to maximize its long term reward signal. This agent learns the mapping of situations to actions in order to maximize an expected reward \([1]\). Instead of learning by example, as is the case with supervised learning, learning desired behavior occurs using past knowledge (experience) from the environment. The current knowledge of an agent about its environment is often referred to as a state. When a state represents the environment such that it is sufficient statistic for the history, it satisfies the so called Markov property.

2.1.1 Markov Decision Process

A Markov Decision Process (MDP) is a reinforcement learning task which satisfies the Markov property. When the number of states in an environment of an agent are finite the MDP is a finite MDP. Formally the finite MDP is a tuple \((S, A, P, R)\), where:

- \(S\) is the finite state space and \(s \in S\) is a state the agent can observe.
- \(A\) is the action space and \(a \in A\) is an action that can be performed by the agent.
• $P$ holds the transition function. When an agent performs action $a$, the environment makes a transition $s \rightarrow s'$, between the current state $s$ and its successor state $s'$ with a probability $P(s'|s, a)$.

• $R$ is the reward function defined as $R : S \times A \rightarrow \mathbb{R}$, which returns an expected reward $r$ each time an action $a$ is performed from $s$.

A policy $\pi$ defines the learning agent’s way of behaving at a given time (a mapping between states and actions) and is formally defined as $\pi : S \rightarrow A$, so when an agent is in a certain state $s$ it can perform action $a = \pi(s)$. The objective of an agent is to maximize the return, i.e. the cumulative (discounted) reward.

### 2.1.2 Value Functions

A value function describes the expected discounted return of an agent to be in a certain state. It is defined with respect to a certain policy $\pi$. A state-value function has a value $v$ for every state, so for $s \in S$ there exists a value $v = V^\pi(s)$. Formally the state-value function is defined as:

$$V^\pi(s) = \sum_{s'} P(s'|s, \pi(s)) \left[ R(s'|s, \pi(s)) + \gamma V^\pi(s') \right],$$

(2.1)

where $s, s' \in S$ and $\gamma$ is the discount value [1]. This has been proven by Bellman (1957) and is therefore called the Bellman equation. The value function $V$ is optimal, denoted as $V^\star$, when it maximizes the expected return in all states $s$.

$$V^\star(s) = \max_a \sum_{s'} P(s'|s, a) \left[ R(s'|s, a) + \gamma V^\star(s') \right].$$

(2.2)

An optimal policy is denoted as $\pi^\star$ and it is always greedy with respect to $V^\star$, $\max_a V^\pi(s)$. Similarly, there exists an action-value function. This determines the value of taking action $a$ in state $s$ and thereafter following policy $\pi$, which is formally defined as:

$$Q^\pi(s, a) = \sum_{s'} P(s'|s, a) \left[ R(s'|s, a) + \gamma Q^\pi(s', \pi(s')) \right].$$

(2.3)

The optimal action-value function $Q^\star(s, a)$ satisfies the equation:

$$Q^\star(s, a) = \sum_{s'} P(s'|s, a) \left[ R(s'|s, a) + \gamma \max_{a'} Q^\star(s', a') \right].$$

(2.4)

Note that there can potentially exist many optimal policies.
2.1.3 Reinforcement Learning Algorithms

The algorithms used in reinforcement learning try to construct an optimal policy for an unknown MDP. They start with an initial state $s_0$ and on each time step $t$ perform an action $a \in A$ on $s$ that is executable in $s$. For episodic tasks, this will change $s$ to $s'$ and a reward $r$ is received until $s'$ is terminal, in which case the $s'$ is reset to a (new) $s_0$. For continuing tasks $s'$ is never terminal, meaning the episode will continue indefinitely.

In Q-learning, a well known algorithm $[1]$, when $s$ is observed, $a$ is chosen, $r$ is received and $s'$ observed an update is performed:

$$Q_t(s, a) = (1 - \alpha_t)Q_{t-1}(s, a) + \alpha_t \left[ r + \gamma \max_{a'} Q_{t-1}(s', a') \right],$$

(2.5)

Where $\alpha_t$ is the learning parameter, that typically decreases with time $t$, which determines how much influence the update has on the current value. As $Q_t(s, a)$ reaches optimal values, the learning rate should decrease to 0.

2.1.4 Action Selection Algorithms

The simplest action selection rule is to select the action with the highest expected value. This rule will always exploit current knowledge and never explore actions with a lower immediate reward which may end up being better. A simple alternative that does explore is called $\epsilon$-Greedy. This method, in a greedy manner, exploits the maximum expected value with a probability of $1 - \epsilon$ and explores with a probability of $\epsilon$ by selecting a random action.

2.1.5 Contextual Bandit Problem

A standard textbook problem in reinforcement learning is the $n$-Armed Bandit Problem. The $n$-Armed Bandit Problem works on a round-by-round basis. On each round:

1. A policy chooses arm $a$ from $1, \ldots, n$.
2. The world reveals reward $r_a$ of the chosen arm.

As information is accumulated over multiple rounds (or episodes), a good policy might converge on a good choice of arm. So each episode a single action is chosen and the reward $r_a$ is observed. In the Contextual Bandit Problem also a single action is chosen.
per episode, but the action may differ depending on the context or the environment. Indeed the Contextual Bandit Problem includes context in the form of a state.\

### 2.2 Hierarchical Task Networks

Whereas reinforcement learning provides a framework to compute an optimal policy, the action set can be defined using a Hierarchical Task Network (HTN). An HTN distinguishes two types of tasks: *primitive* tasks or actions that can be executed directly, and *compound* tasks which are composed of a set of simpler tasks. Instead of a single action, as is the case in conventional RL, a decomposed HTN plan consists of a specific sequence of primitive actions. These specific sequences are encoded within the structure of the HTN itself and reduce the action space compared to conventional RL.

#### 2.2.1 HTN-planner

The objective of an HTN-planner [2, 3] is to produce a sequence of actions that can solve a particular planning problem. HTN planning occurs by recursively decomposing compound tasks into smaller tasks. Eventually this process results in a sequence of primitive tasks (a plan), which need no further decomposition and can be executed directly. To construct a plan, the planner takes a planning problem and a planning domain. A planning problem is formulated as a compound task that is to be decomposed combined with a set of facts that describe the current world state. Planning domains describe the problem in a hierarchy composed of compound and primitive tasks.

#### 2.2.2 Example

Figure 2.1 shows a travel example domain and figure 2.2 shows an example planning state. Suppose our planning problem is `((travel-to uptown))`. The HTN-planner will begin calling the compound task `travel-to` with `uptown` as argument. Next the preconditions of the first branch are evaluated. In this case, as `(>= 30 60)` evaluates to false, the walking branch is chosen. This results in a decomposed plan consisting of a single primitive task: `((!walk downtown uptown))`, a long walk indeed.

Matching preconditions is similar to the matching process of Prolog [4]. Note that the planner will always try to decompose the highest branch first. If the preconditions are

\footnote{An example of this type of problem is targeted advertisement e.g. the selection of ads alongside an email. The state $s$ holds a vector of keywords from the mail, the action $a$ is one of many ads and the reward $r$ whether the ad is clicked on.}
Chapter 2. Background

Figure 2.1: HTN planning domain

met, it will decompose this branch further until it either cannot decompose any further, or a precondition fails. In the former case the planner would try to find a reduction for the compound task drive-taxi ?x ?y, which is not defined in this example. In the latter, the planner will backtrack and try the second branch in this example branch_walking.

1. (at downtown)
2. (have-cash 30)
3. (distance downtown uptown 40))

Figure 2.2: HTN initial state

2.2.3 Formal Description

This section defines the syntax and semantics used in the HTN planner. It uses the same first-order-logic definitions of variable and constant symbols, function and predicate
symbols, terms, atoms, conjuncts, most-general unifiers (mgus) as SHOP [2], an HTN-planner developed in Java. However, for the purpose of this thesis, some additional definitions are introduced.

### 2.2.3.1 State

A *state* is a set of ground atoms, and an *axiom set* is a set of Horn clauses\(^2\). If \(S\) is a state and \(X\) is an axiom set, then \(S \cup X\) satisfies a conjunct \(C\) if there is a substitution \(u\) (called a *satisfier*) such that \(S \cup X \vdash C^u\). \(u\) is a *most general satisfier* if a satisfier \(v\) can be expressed by \(v = uw\), where \(w\) is another substitution. Note that the state \(s\) in RL differs from state \(S\) in the HTN-planner. \(s\) defines world properties which are static during an episode, whereas \(S\) is an internal state within the HTN-planner that is dynamic and thus can change during an episode.

### 2.2.3.2 Task

A *task* is a list of the form \((s \ t_1 \ t_2 \ldots \ t_n)\), where \(s\), the task’s name, is a task symbol and \(t_1, t_2, \ldots, t_n\), the task’s arguments, are terms. The task is *primitive* if \(s\) is a primitive task symbol and *compound* otherwise.

### 2.2.3.3 Operator

An *operator*\(^3\) is an expression that has the form \((\text{:operator} \ h \ D \ A)\), where \(h\) (the *head*) is a primitive task, and \(D\) and \(A\) (the *deletions* and *additions*) are sets of atoms containing no variable symbols other than those in \(h\).

The intent of an operator \(o = (\text{:operator} \ h \ D \ A)\) is to specify that \(h\) can be accomplished by modifying the current state of the world to remove every atom in \(D\) and add every atom in \(A\). More specifically, if \(t\) is a primitive task and there is an mgu \(u\) for \(t\) and \(h\) such that \(h^u\) is ground, then \(o\) is applicable to \(t\), and the list \((h^u)\) is a simple plan for \(t\). If we execute this plan in some state \(S\), it produces the state \(h^u(S) = o^u(S) = (S - D)^u \cup A^u\).

---

\(^2\)Note that the HTN-planner of Killzone 3 does not support axioms.

\(^3\)In Killzone 3, we *emulated* the operators that manipulate the world state during planning using specific call functions. Note that these functions are not backtrack-safe and for consistent world state manipulation, operators should be implemented.
2.2.3.4 Method

A method is an expression that has the form \( (:\text{method } h \ B) \), where \( h \), the method’s head, is a compound task, \( B = \{ b \in B : b = (C \ T) \} \) is an ordered list of branches, where \( C \) (precondition of a branch) is a conjunct, and \( T \) (the tail of a branch) is a task list. Note that the order of the branches is fixed.

The intent of a method \( m = (:\text{method } h b_1 \ldots b_n) \) is to specify that if the current state of the world satisfies one of \( C \in b_i \), then \( h \) can be accomplished by performing the tasks in \( T \in b_i \) in the order given. More specifically, let \( S \) be a state, \( X \) be an axiom set, and \( t \) be a task atom. Suppose there is an mgu \( u \) that unifies \( t \) with \( h \), and suppose \( S \cup X \models C^u \). Then \( m \) is applicable to \( t \) in \( S \cup X \), and the result of applying \( m \) to \( t \) is the set of tasks lists \( R = \{(T^v) : v \text{ is an mgs for } C^u \text{ from } S\} \). Each task list \( r \in R \) is a simple reduction of \( t \) by \( m \) in \( S \cup X \).

2.2.3.5 Plan

A plan is a list of heads of ground operator instances. If \( p \) is a plan and \( S \) is a state, then \( p(S) \) is the state produced by starting with \( S \) and executing the operator instances in the order that their heads appear in \( p \).

2.2.3.6 Planning Problem

A planning problem is a tuple \( P = (S, T, D) \), where \( S \) is a state, \( T \) is a task list and \( D \) is a set of axioms, operators and methods. Let \( \Pi(S, T, D) \) be the set of all plans for \( T \) from \( S \) in \( D \). We can define \( \Pi(S, T, D) \) recursively as follows.

If \( T \) is empty, then \( \Pi(S, T, D) \) contains exactly one plan, namely the empty plan. Otherwise, let \( t \in T \) be the first task atom, and \( R = T \setminus t \), the remaining task atoms. There are three cases. (1) If \( t \) is primitive and there is a simple plan \( p \) for \( t \), then \( \Pi(S, T, D) = \{ \text{append}(p, q) : q \in \Pi(p(S), R, D) \} \). (2) If \( t \) is primitive and there is no simple plan for \( t \), then \( \Pi(S, T, D) = \emptyset \). (3) If \( t \) is compound, then \( \Pi(S, T, D) = \bigcup \{ \Pi(S, \text{append}(r, R), D) : r \text{ is a simple reduction of } t \} \).
Chapter 3

Related Work

This chapter describes some related AI techniques used in both the gaming industry and the academic world. These techniques are not all directly related to the method proposed in this thesis, but also show the current state of the art within the gaming industry and the academic world.

3.1 Game Industry

3.1.1 F.E.A.R.

First Encounter Assault Recon (F.E.A.R.), a first-person shooter developed by Monolith Productions and published by Vivendi, was released in 2005. The game’s story revolves around a supernatural phenomenon, which F.E.A.R. – a fictional special forces team – is called to contain. The player assumes the role of F.E.A.R.’s Point Man, who possesses superhuman reflexes, and must uncover the secrets of a paranormal menace in the form of a little girl. F.E.A.R. was one of the first games in which the AI developers were able to separate their domain language from the problem solver using a technique called Goal-Oriented Action Planning [5] with the STRIPS [6] planner. However, this system was only applied on the individual level of the NPCs, the level which controls the reactive behavior of a single NPC.

3.1.1.1 Goal-Oriented Action Planning

F.E.A.R. applies a Goal-Oriented Action Planner (GOAP) for its decision making logic. An agent in F.E.A.R. constantly selects its most relevant goal to control its behavior. At each logic time step, the most relevant or best goal (a desired state of the world)
is selected and a sequence of actions (plan) is constructed which is able to satisfy that goal most effectively. The formalized process of searching for a sequence of actions that satisfies a goal used in F.E.A.R. closely resembles an automated planner called STRIPS.

### 3.1.1.2 STRIPS

STRIPS was developed at Stanford University in 1970. STRIPS is an acronym for STanford Research Institute Problem Solver [6] and can be seen as the predecessor of the HTN-planner. The crucial difference between STRIPS and an HTN-planner is that in the former, the reasoning process takes place at the level of the actions (operator space) whereas in the latter the reasoning process takes place at the level of the tasks (plan space) [7–9]. STRIPS consists of goals and actions where goals describe some desired state of the world we want to reach, and actions are defined in terms of preconditions and effects. An action may only execute if all of its preconditions are met. Each action changes the state (conjunction of literals) of the world in some way. The STRIPS planner applied to F.E.A.R. assigns costs to actions and tries to find a shortest path within the action space using A* to construct its plan. In order to achieve the desired behavior, one has to tweak these cost values such that A* finds the “right” path. This turns out to be a precise and tedious task in practice.

### 3.1.2 Halo 2

Halo 2 is a first-person shooter video game developed by Bungie Studios. Released for the Xbox video game console on November 9, 2004. The player alternatively assumes the roles of the human Master Chief and the alien Arbiter in a 26th century conflict between the human UNSC and genocidal Covenant. Players fight enemies on foot, or with a collection of alien and human vehicles.

#### 3.1.2.1 Hierarchical Finite State Machine

The control structure of Halo 2 uses a hierarchical finite state machine (HFSM) or, more specifically, behavior tree or behavior DAG (Directed Acyclic Graph) [10]. A finite state machine is a behavior model composed of a finite number of states, transitions between those states and actions similar to a flow graph. An HFSM imposes a hierarchy on the model, where non-leaf states make decisions about which children to run and leaf states perform a certain action, see Figure 3.1. There are two general approaches in the decision making of the non-leaf behavior and at different times Halo 2 uses them both:
The different characters in Halo 2 can all have different behaviors. However, as most basic behaviors are shared, the game uses custom behaviors. Each character uses the same HFSM scheme, but specific characters trigger different children. One of the main differences between an HFSM and an HTN is the domain language. The domain language of an HFSM is not strictly defined, leaving a lot of design decisions to the programmer. Whereas the HTN domain language is well defined, giving a solid framework to apply reinforcement learning on.

### 3.2 Academic Work

#### 3.2.1 High Level Reinforcement Learning in Strategy Games

As human players rapidly discover and exploit the weaknesses of hard coded strategies in games, this paper presents a reinforcement learning approach for learning a policy that switches between high-level strategies \[11\]. The testbed for this paper is Civilization IV, a complex strategy game\(^1\) in which players evolve a culture through the ages.

\(^1\)Strategy video games are a genre of video game that emphasize skillful thinking and planning to achieve victory. They emphasize strategic, tactical, and sometimes logistical challenges.
3.2.1.1 Model-based Q-learning: Dyna-Q

Q-learning is a model-free method, meaning it learns a policy directly, without first obtaining the model parameters (transition and reward functions). An alternative is to use a model-based method that learns the model parameters and uses the model definition to learn a policy. Dyna-Q is a method that can learn the model and the Q-values at the same time. Thus, the agent learns both the Q-values and the model through acting in the environment. This model is then used to simulate the environment and the Q-values are updated accordingly [11]. As the model becomes a better representation of the problem, the Q-values become more accurate and convergence will occur more quickly. See algorithm 1. First the regular Q-learning update is performed and the probability and reward models are updated as averages given the new information. The model sampling occurs in the for-loop. For some designated number of iterations N the model is sampled and the Q-values are updated accordingly.

**Algorithm 1** Dyna-Q\( (Q, r, s, a) \): Returns updated Q-values, \( Q \)

**Require:** The Q-values, \( Q \), immediate reward \( r \), state \( s \) and action \( a \)

1: \( Q(s, a) \leftarrow Q(s, a) + \alpha(r + \gamma Q(s', a') - Q(s, a)) \)
2: \( P(s'|s, a) \leftarrow updatePAverage(s, a, s') \)
3: \( R(s, a) \leftarrow updateRAverage(s, a) \)
4: for \( i = 0 \) to \( N \) do
5: \( s' \leftarrow randomPreviouslySeenS() \)
6: \( a' \leftarrow randomPreviouslyTakenA(s') \)
7: \( s'' \leftarrow sampleFromModel(s', a') \)
8: \( r' \leftarrow fromModel(s', a') \)
9: \( Q(s', a') \leftarrow Q(s', a') + \alpha(r + \gamma Q(s'', a'') - Q(s', a')) \)
10: end for
11: return \( Q \)

3.2.1.2 Civilization as MDP

The statespace is defined as a set of four state features\(^2\): population difference, land difference, military power difference and remaining land. These features, \( f_1, \ldots, f_4 \) are discretized into three different values:

\[
 f_i = \begin{cases} 
 2 & \text{if } \text{diff} > 10 \\
 1 & \text{if } -10 < \text{diff} < 10 \\
 0 & \text{if } \text{diff} < -10 
\end{cases}
\]

\(^2\)State features can improve the speed of learning assuming the individual features are independent of each other.
Where $\text{diff}$ represents the difference in value between the agent and the opponent. An action is a choice of strategy that is built-in into the game. The action space was limited to four different actions. Each action represents a different type of play of the game: Aggressive and expansive, financial and organized, etc. The immediate reward is defined as the step based score of the game. That is the difference of the agent’s total score and the opponent’s total score.

Although this approach works reasonably well for the Civilization game, in a first-person shooter (FPS) game, the gameplay is much more fast-paced and changes occur in rapid succession. In order to cope with this properly, more detailed world information needs to be encoded within the state. The main challenges can therefore be found in properly modelling the FPS as an MDP without blowing up the state-action space. Another problem is the partial observability within the FPS, the enemy location can only be acquired through scouting the environment by individual agents.

3.2.2 HTN for Encoding Strategic Game AI

The paper presents a case study for HTN-Planning on a strategic level in the game called Unreal Tournament\textsuperscript{3} (UT) [12]. The game mode to which the case study is applied is called domination. In domination, there are fixed locations in the world that can be captured by letting a team member step into a location. The team gets a point for every five seconds that each domination location remains under the control of that team. The game is won by the first team that gets a pre-specified amount of points.

3.2.2.1 Strategies

Two different strategies were encoded in the HTNs. The first strategy is called Control Half Plus One Points. This strategy selects half plus one of the domination locations and sets bots to capture these points. The second strategy Control All Points requires that the team consists of at least two members. It calls for two members to capture all domination locations and patrol between them. The remaining team members are assigned to search and destroy tasks.

\textsuperscript{3}A first-person shooter video game co-developed by Epic Games and Digital Extremes. It was published in 1999 by GT Interactive.
3.2.2.2 Architecture

In order to compute the HTN for the grand strategies the Java based SHOP Planner\cite{2} is applied. An event-driven program encoded in the Javabot FSMs allows the individual bots to react to the environment while contributing to the grand task at hand.

Differentiating between strategies appears to be performed manually instead of using some adaptive method or heuristic.

3.2.3 Neural Networks and Evolving AI in FPS Games

Other approaches to optimize bot behavior in FPS games involve the application of neural networks and evolutionary algorithms. Although there have been successful attempts in applying these methods \cite{13–15}, the main problem remains the loss of control over the bots as they converge to the (local) optimal, without some form of granularity. Aside from that, neural networks require many offline training rounds and it can be difficult to understand what is going on when they become large. Making them hard to debug during development.
Chapter 4

Domain

4.1 Guerrilla

Guerrilla is a game development studio, based in the heart of Amsterdam, the Netherlands. It was formed at the beginning of 2000 as a result of a merger between 3 separate Dutch-based developers. The company now employs 130 developers, designers and artists, encompassing 20 different nationalities. The first game released by Guerrilla, Shellshock: Nam ’67 was developed for the PC, Xbox and PlayStation 2 and published by Eidos Interactive. In 2004, Guerrilla signed an exclusive deal with Sony Computer Entertainment. Under that deal, Guerrilla developed games exclusively for Sony’s consoles (PlayStation 2, PlayStation 3 and the PlayStation portable). After the release of Killzone for the PlayStation 2 (2004), the company was acquired by Sony Computer Entertainment in 2005. It went on to release Killzone: Liberation for PlayStation Portable (2006), and KillZone 2 (2009). At the time of writing this thesis, the company just released a new title for the PlayStation 3 called Killzone 3.

4.2 Killzone 3

Killzone 3 (KZ3) is a first-person shooter (FPS) game. Figure 4.1 shows a typical ingame scenario from KZ3 as viewed by the player. Like most 3D shooters, KZ3 offers a variety of playing possibilities with friends or online. It distinguishes three different modes: Singleplayer, cooperative play and multiplayer. This thesis focuses on the multiplayer mode.
4.2.1 Multiplayer

A Multiplayer online game is a multiplayer video game which is played via a game server over the Internet, with other human players around the world. Players either compete against each other (individually or in teams/clans) or cooperate with each other against a common enemy (e.g. an AI). In contrast to singleplayer mode, the game is played on one single stage only. Since these games are not centered around one player, when a player dies, the game is not restarted. Instead, these games continue and the player that died will have the opportunity to rejoin the game.

A multiplayer game mode is defined by a set of rules and regulations that specify game objectives, win/lose scenarios and conditions for scoring and ranking on team and individual basis. For any game mode, points are rewarded for killing enemies. However, many game modes define multiple different (primary) objectives and therefore require different strategies to win. The nature and amount of objectives vary among the different game modes and can even be different for the opposing teams. For instant in the symmetric game mode “Bodycount”, the teams have to kill as many players in the opposing team as possible, where each kill gains a point. In the game mode “Assassination”, a non-symmetric mode, team one has to assassinate a single key player in team two, which team two has to defend. For this thesis, we will create strategies for the symmetric game mode “Capture and Hold”.

Figure 4.1: Killzone 3 ingame first-person view
4.2.1.1 Capture and Hold

In this game mode, there are three key areas on the map that can be captured by either team. To gain control over an area, players must stand within the capture radius of the area and must make sure that no enemy soldiers are within capture range of that same area during the capture process. If a team succeeds in keeping away the enemy long enough, the area will change ownership to the capturing team. Once an area is captured the team receives points for each time step the area is under their control. If the maximum amount of points is gained (defined by the game) the team wins or, alternatively, if the time limit is reached the team with the most points wins. When both teams have the same amount of points the game results in a draw.

4.2.2 Multiplayer AI Design Overview

Singleplayer and coop AI differ greatly from multiplayer AI. The reason for this difference is that NPCs found in singleplayer games, have a different role from those in multiplayer games.

In order to put the multiplayer bot behavior – at a strategic level – in perspective, this section gives a general overview of the entire AI architecture [16, 17]. Figure 4.2 shows a simplified overview of the AI hierarchy as defined in Killzone 3. The top layer shows the

Figure 4.2: Simplified design overview of the AI architecture
decision system at the strategic level, often referred to as commander or general\textsuperscript{1}. This is the level where the method, proposed in Chapter 5, is focused on. The subsections below will explain each layer in more detail (though still at high level). Note that both the commander and the group are concepts within the AI architecture, they do not represent actual entities in the virtual world.

4.2.2.1 Individual

The individuals define the actual NPCs. They observe their surroundings through visual and auditive stimuli modeled after how humans observe the world. This prevents stupid mistakes such as an NPC “mysteriously” knowing that an enemy is behind him. These observations about the world are stored in a world fact database. The world fact database is local and different for each individual and used by the HTN-planner to create behavioral plans using orders. These orders include reloading weapon, firing weapon, going in cover, blind fire, etc.

4.2.2.2 Group

A group, as the name implies, defines a set of individuals. These individuals form a small military unit often referred to as a squad\textsuperscript{2}. The group is responsible for e.g. defending an area or capturing a strategic point. Groups are both created and controlled at the commander layer using commands. Each group also uses a unique world fact database which stores information such as where each individual is located and which group it belongs to. The squad is responsible for the coherency of the individuals that belong to it during movement.

4.2.2.3 Commander

The commander is the top layer in the AI. Its actions are performed on the strategic level. The commander is responsible for the following tasks:

- Squad creation
- NPC to squad allocation
- Commands for squads

\textsuperscript{1}“Strategy without tactics is the slowest route to victory. Tactics without strategy is the noise before defeat.” – Sun Tzu.

\textsuperscript{2}The terms squad and group are used interchangeable throughout this thesis.
Chapter 4. Domain

The commands that can be sent to squads include: Going to a certain waypoint, attacking an entity, defending a marker or entity. The squads, in turn, can report back the status of their progress. These commands form the primitive actions which, when put in the right sequence, form a sensible plan. We expanded the HTN architecture to support the commands above for the commander such that we can encode a strategy in a domain. Given this domain, the HTN-planner can construct a plan at the start of e.g. a capture and hold game. Figure 4.3 shows how the command sequence of a squad capturing an area is build. The method requires a list of areas indices as argument and

```
(:method (capture_areas ?inp_area_list)
  (branch_areas_captured
   (and (call eq (call get_list_length ?inp_area_list) 0)
        (= ?squad_index (call get_last_created_squad_index))
        (= ?squad (call get_squad ?squad_index))
   )
   ( (!end_command_sequence ?squad)
   )
  )

(branch_capture_area
  (and (= ?area_index (call get_list_item ?inp_area_list 0))
       (= ?area_list (call remove_list_item ?inp_area_list 0))
       (= ?squad_index (call get_last_created_squad))
       (= ?squad (call get_squad ?squad_index))
  )
  ( (!order_squad_custom ?squad capture ?area_index)
      (!order_squad_custom ?squad advance ?area_index auto)
      (capture_areas ?area_list)
  )
)
```

Figure 4.3: Area capture example domain

orders a squad to capture the area and advance. While there are still areas that require capturing, the method assigns a squad. Finally an “end of command sequence” is given.

Applying the HTN-planner at this layer separates the domain language from the problem solver. This separation allows for the application of reinforcement learning integrated within the HTN-planner. Integrating the RL algorithm on this level enables machine learning the best strategies across multiple game modes against various fixed opponents.
4.2.2.4 C++ Strategy Implementation

Currently the strategical logic is implemented in C++. This is done using an objective based approach. An objective is implemented as a C++ class which defines the desired amount of bots and squads to be accomplished. Some examples of objectives are:

- Attack Entity
- Defend Marker
- Escort Entity

Every logical update, the objectives and their importances are computed using hard-coded heuristics and squads get assigned or reassigned to these objectives. This process continues until the game is over.
Chapter 5

Approach

This Chapter introduces the learning algorithm and the handcrafted strategies for Killzone 3’s multiplayer game mode capture and hold. The proposed method is a modified version of the HTN planner combined with a simple form of reinforcement learning, a basic description follows. At the start of a multiplayer game of Killzone 3 (e.g. capture and hold), a strategy is constructed using the HTN planner which was either selected at random during exploration or greedily during exploitation. The strategy is executed and at the end of the round or episode, its reward is observed and the value of the weight belonging to the strategy is updated accordingly. So each episode a single action is chosen and executed, depending on environmental variables, making this a contextual bandit problem.

5.1 Learning AI

As stated in Chapter 2, the ordering of branches of the methods in the original HTN planner is fixed. This fixed ordering makes sense if there is a clear order in which a planning problem should be decomposed given a state $S$ and an axiom set $X$. At the strategic level of the Killzone 3 AI, there exist methods for which there is no clear ordering of their branches beforehand. That is, given some method $m = (:\text{method } h b_1 \ldots b_n)$ from the strategic domain, the order in which $C$ from $(C T) \in b_i \forall i$ is entailed for some mgu $u$ that unifies $t^n \in T$ with $h$ such that $S \cup X \models C^u$, cannot be defined in terms of $S$ and $X$ alone. This section proposes an algorithm that can adapt the ordering by applying reinforcement learning.
5.1.1 Branch Ordering and Selection

By assigning weights (or values) $w$ to each branch $b_{im}$ in a method $m$ we can sort the branches on their respective weight in descending order and execute the first branch whose preconditions are met. A branch is a tuple $b_{im} = (C_{im}, T_{im}, w(s)_{im})$, where $C_{im}$ holds the preconditions, $T_{im}$ holds the task list and $w(s)_{im}$ is a weight for a certain contextual state $s$. As explained in chapter 2, state $s$ defines world properties which are static during an episode (the map onto which the episode is played and the team or faction), whereas $S$ is a world state defining potential non-static properties during an episode. During exploration by e.g. $\epsilon$-Greedy, branches are randomly selected, while during exploitation branches are sorted by their weight in descending order and the first branch for which precondition $C_{im}$ is met can be further decomposed.

Given the Q-learning update rule in 2.5, we will now present how we apply an adapted version in our algorithm. As our problem is modelled as a contextual bandit problem, we only chose one action per episode:

$$Q_t(s, a) = (1 - \alpha_t)Q_{t-1}(s, a) + \alpha_t\left[r + \gamma \max_{a'} Q_{t-1}(s', a')\right]$$

(5.1)

$$= (1 - \alpha_t)Q_{t-1}(s, a) + \alpha_t\left[r + 0\right]$$

(5.2)

$$= Q_{t-1}(s, a) - \alpha_t Q_{t-1}(s, a) + \alpha_t r$$

(5.3)

$$= Q_{t-1}(s, a) + \alpha_t\left[r - Q_{t-1}(s, a)\right]$$

(5.4)

As such there is no transition from $s$ to $s'$ within an episode, reducing the right term in the square brackets to 0. At the end of an episode, a branch update occurs by observing the immediate reward $r$ on the current leafnode after which the highest value is propagated upwards to the rootnode:

$$w(s)_{im} = \begin{cases} 
    w(s)_{im} + \alpha \left[r - w(s)_{im}\right] & \text{if } w(s)_{im} \text{ is a leafnode} \\
    \max_{c \in \text{children}(m)} w(s)_{ic} & \text{otherwise}
\end{cases}$$

(5.5)

Where $\alpha$ denotes the learning rate, $c$ denotes a child method under parent $m$ and $i$ the branch index, with $im$ or $ic$ uniquely identifying the action. Thus equation 5.5 successfully implements the reinforcement learning update rule within an hierarchical environment.

5.1.2 Pruning the Action Space

As stated in chapter 2, the HTN is able to define the action set for RL. Instead of allowing all possible sequence of actions to be chosen, as is the case in regular reinforcement
learning, a plan (specific sequence of primitive tasks) is encoded using the HTN. This greatly reduces the action space, allowing only sensible sequences of tasks to be executed as defined by the prior knowledge of the developer or expert.

The pruning of the action space does come at a potential cost. Figure 5.1 shows a graphical depiction of the HTN planner in action, the nodes represent (composite) methods and the edges represent the branches of a method. The planner starts at the root node and tries to decompose the ordered branches from the highest weight to the lowest at each level in the tree in a depth first manner. In this case, the most left leaf-node cannot be reduced as its preconditions are not met. The planner performs a backtrack and tries to decompose its sibling, which is of weight 2, whereas the best leaf-node would be the most right node with a weight of 3. This is the potential pitfall that is the result of pruning the action space by HTN encoding and it is up to the prior knowledge of the developer to ensure a proper encoding. The sibling nodes should have a contextual relation with each other that enforces equal constraints. In this case, the parent node of the node with weight 5 should have this constraint resulting in the right most node with weight 3.

5.2 Implementation

The method is implemented using two algorithms execute-plan and seek-plan. The method first tries to find a plan, executes it and performs the weight update.

5.2.1 Execute Plan

Algorithm 2 first calls subroutine seek-plan which returns a plan \( p \) and its traversed path of weights \( W \) (in reverse order) in the form of a list that require updates. If the
method corresponding to \( w(s)_m \in W \) is a leaf-node, the immediate reward update is applied, otherwise the max weight of the children of \( m \) is assigned to \( w(s)_m \). This propagates the highest weight to the root node, making sure the best path is selected during exploitation. The method \( c \) variable is used for keeping track of the child method.

**Algorithm 2** \( \text{execute-plan}(S, T, D) \): Executes a plan and updates the weights

**Require:** The state \( S \), task list \( T \) and \( D \) a set of operators, axioms and methods

1. \((p, W) \leftarrow \text{seek-plan}(S, T, D, \text{nil}, \text{nil})\)
2. observe world state \( s \)
3. run episode with plan \( p \), observe reward \( r \)
4. method \( c \leftarrow \text{nil} \)
5. for all \( w(s)_m \in W \) do
6. if is-leaf \( (m) \) then
7. \( w(s)_m \leftarrow w(s)_m + \alpha[r - w(s)_m] \)
8. \( c \leftarrow m \)
9. else
10. \( w(s)_m \leftarrow \max\{w(s)_c : w(s)_c \in \text{branches}(c)\} \)
11. \( c \leftarrow m \)
12. end if
13. end for

Next, the episode (simulation) is executed with plan \( p \) applied and reward \( r \) is observed at the end of the episode. This reward is a numerical value and defined as follows:

\[
    r = m - e, \tag{5.6}
\]

where \( m \) defines the mission points of the current team and \( e \) the mission points of the enemy.\(^1\) This results in a reward \( r \in [-50, 50] \), where \( r < 0 \) indicates a loss and \( r > 0 \) indicates a victory and \( r = 0 \) indicates a draw. Finally the traversed weight path is updated according to equation 5.5.

### 5.2.2 Seek Plan

The subroutine \( \text{seek-plan} \) shown in Algorithm 3 returns a tuple \((p, W)\), where \( p \) is the plan found and \( W \) the corresponding list of weights of the traversed path in reverse order. The algorithm is an extended version of the HTN planner which was already available in Killzone 3. It recursively decomposes the given task list \( T \) using state \( S \) and the set of operators \( D \) until the entire plan consists of primitive actions only. Along the way it stores the weights \( w \in W \) that require updating. Since a method \( m \) can be used in different domains, the corresponding runtime callstack \( c \) of method \( m \) is used to

\(^1\)The actual formula for computing mission points in Killzone 3 varies per game mode. In the case of capture and hold, mission points are assigned per timestep to teams that control captureable areas.
Algorithm 3 seek-plan\((S, T, D, p, W)\): Returns a plan and the weights of the learnable methods along the path

**Require:** The state \(S\), task list \(T\) and \(D\) a set of operators, axioms and methods, \(p\) the plan, \(W\) the set of weights of the learnable methods found in the traversed path

\[
\begin{align*}
1: & \text{if } t = \text{nil} \text{ then} \\
2: & \quad \text{return } (p, W) \\
3: & \text{end if} \\
4: & t \leftarrow \text{first task in } T, R \leftarrow T - t \\
5: & \text{if } t \text{ is primitive then} \\
6: & \quad \text{if there is a simple plan } q \text{ for } t \text{ then} \\
7: & \quad \quad \text{return } \text{seek-plan}(q(S), R, D, \text{append}(p, q), W) \\
8: & \quad \text{else} \\
9: & \quad \quad \text{return } \bot \\
10: & \text{end if} \\
11: & \text{else} \\
12: & \quad \text{observe callstack } c \\
13: & \quad \text{for all } m \in D \text{ that can reduce } t \text{ in } S \text{ do} \\
14: & \quad \quad \text{if } m \text{ is learnable then} \\
15: & \quad \quad \quad \text{sort branches on } w_{imc} \text{ in descending order} \\
16: & \quad \quad \text{end if} \\
17: & \quad \quad r \leftarrow \text{reduction of } t \text{ using } m \text{ in } S \\
18: & \quad \quad (p', W') \leftarrow \text{seek-plan}(S, \text{append}(r, R), D, p, \text{append}(w_{imc}, W)) \\
19: & \quad \quad \text{if } p' \neq \bot \text{ then} \\
20: & \quad \quad \quad \text{return } (p', W') \\
21: & \quad \quad \text{end if} \\
22: & \quad \text{end for} \\
23: & \text{end if}
\]

discriminate between the different contexts. Thus the callstack and method provide a unique key for a weight.

5.3 Strategies

For this thesis, three main strategies were developed for the game mode *capture and hold* using three areas, namely: Steam Roll, Capture and Split, Divide and Conquer. Each of the strategies contains multiple variations which determine area capturing sequences, number of squads and relative sizes of the squads. The domain file for the *capture and hold* strategies is listed in appendix A.

5.3.1 Steam Roll

The strategy *steam roll* is depicted in Figure 5.2. In this strategy, a single squad \(A\) is created and sequentially traverses each of the areas that require capturing. In this
case, the squad captures the areas in the following sequence: 1, 2, 3. Variations on this strategy are the different capture sequences of the areas, indicated by areaXYZ, where XYZ can be any permutation of 1, 2, 3. The created squad is indeed very strong and will most likely capture the area it’s going for. However, its squad can only capture a single area at a time, making it a slow strategy. Secondly, the captured areas will be left undefended completely.

### 5.3.2 Capture and Split

This strategy creates three squads instead of a single squad as is the case in steam roll. First the three squads capture a single area together. Next a single squad stays behind and the remaining squads go to the second area and capture it, where the second squad also stays behind and the third squad finally captures the last area. This strategy poses two variables along which it can differ, capture sequences as in steam roll, and the various number of individuals per squad. A specific strategy from capture and split is thus defined as areaXYZ_sqdABC, where areaXYZ is equally defined as in steam roll and sqdABC defines the distribution of the individuals among the squads. For instance sqd121 means that the individuals are divided over the squads as \( \{1, 2, 1\} \). See Figure 5.3 for an example of this strategy.

### 5.3.3 Divide And Conquer

The last strategy is shown in Figure 5.4. Divide and conquer divides its forces into three squads and tries to capture each of the areas in parallel. This strategy also has
the ability to vary along the squad distributions like *capture and split*. A variation of this strategy is thus defined as *sqdABC*. This type of strategy, when successful, will
capture each of the areas the fastest. However, each with a much weaker force that can
be overrun by the enemy with less effort than the previous strategies.
Chapter 6

Experiments and Results

In this chapter we empirically analyze the method using the *Killzone 3* environment. We devised three different experiments that will answer our questions posed in the first chapter:

1. Can we encode good strategies into a hierarchical task network?
2. Is it possible to learn the best strategy with respect to a fixed opponent?
3. Is it possible to adapt to a different fixed opponent?

The first is a training run against the baseline, which is the current C++ strategy version, on various maps. The second is a comparison between the method and averaged reward data from the strategies against the baseline. Finally we apply the trained version from the first experiment to a different fixed opponent and observe its adaptive capabilities.

6.1 Settings

For our experiments we used three different multiplayer maps MP01, MP02 and MP03. Most experiments were run on MP01, also known as *Corinth Highway*\(^1\). Figure 6.1 shows a top view of MP01, on the left we can see the ISA base and on the right the HGH base. In the center we can see the capturable areas CH1, CH2 and CH3. Figure 6.2 and 6.3 show two more maps, with the captureable areas more widespread across the map. The detached images in figure 6.3 represent different floors of the map, in this case both the ISA and HGH bases are located on the first floor. As can be seen in all the

\(^1\)Killzone 3's exo mounting was disabled for both teams as exos were introduced after the strategies were developed.
figures, the maps are symmetric with respect to the starting positions for both teams in order to ensure a fair battle. Each episode had a timelimit of 10 minutes and 16 bots divided over two factions or teams (8 versus 8).

6.2 Training

The method was trained against the baseline, which is the current C++ strategy implementation, for 150 episodes on three different multiplayer maps averaged over 3 runs.
During training, the learning-rate $\alpha$ was set to $1/4$ and exploration-rate $\epsilon$ started at 1.0 and gradually decayed to 0.1 after each episode.\footnote{To compute the decay-rate $d$ we used $d = (\epsilon_f/\epsilon_s)^{1/E}$, where $\epsilon_s$ is the start exploration-rate, $\epsilon_f$ is the final exploration-rate and $E$ is the amount of episodes.} Figure 6.4 shows the maximum reward gained after each episode for the three different multiplayer maps. As stated in chapter 4, the reward is defined as $r = m - e$. In the case of capture and hold it represents the amount of time capturable areas were under our control minus the areas that were under enemy control. Thus when $r > 0$ indicates a win and $r = 50$ is the maximum score achievable, meaning the capturable areas were never under enemy control. By examining the weights corresponding to this graph, it showed that on all three maps the divide and conquer substrategies are most successful. This is probably due to the parallel nature of the strategy. As each area is assigned a designated squad at start and points are gained per timestep for captured areas.
6.3 Comparison

To determine how well the trained weights reflect the real success of the strategies, we compare it to the averaged rewards for each of the individual strategies against the baseline. Figure 6.5(a) shows the weight distribution obtained for the different strategies during training against the baseline on MP01 and figure 6.5(b) shows the rewards per strategy averaged over 10 episodes. From these results we can conclude that the method is consistent with the averaged rewards w.r.t. the divide and conquer strategies (indicated in blue). For the other strategies, bigger differences can be observed as not every strategy is explored as thoroughly due to the decreasing exploration-rate. Furthermore,
these differences are the result of fluctuations in the rewards of the strategies itself per episode. Some episodes the strategy performed better than others. These fluctuations can also be seen in figure 6.4, e.g. the red line from episode 100 to 150 jumping up and down between 40 and 50. The stream roll type strategies were most “unstable”, e.g. area021 varied from −24 to 18 against the baseline.

The figure below shows the weight distribution of two training runs on the map MP02 (Pyrrhus Crater). MP02 was chosen as the capture points are further away from each other, see figure 6.2. As both factions always start on the same side of the map, we expect to see some mirrored behavior in the area capturing sequence for capture and split and steam roll. The substrategies of divide and conquer barely show a difference as they capture each area simultaneously. On the left figure 6.6(a) the ISA faction was trained against the baseline and on the right figure 6.6(b) the HGH faction was trained against the baseline. A clear mirrored behavior can be seen in area201_sqd212 and area201_sqd111. Both start at area 2 and the ISA faction first captures area 1, whereas the HGH faction first captures area 0. This mirrored behavior is however not shown for the steam roll sub strategies as they are more unstable as stated above.

6.4 Adaptability

To determine the adaptability against a different fixed opponent we chose the best strategy according to figure 6.5(a), strategy sqd121 as the new adversary. For this experiment α was set to 1/3 to allow for more agressive changes and ε was fixed at 1/3 in order to prevent getting stuck in local optima. Figure 6.7(b) shows the new weight distribution after 20 episodes against sqd121 on MP01. Most stream roll strategies dropped further
below zero, indicating more loss and the divide and conquer strategies hover around zero as they are equally strong. The strategies area201_sqd212 and area210_sqd111 pose decent counters and show that the algorithm adapted to the different fixed opponent.
Chapter 7

Conclusions

In this thesis, we proposed a method that maintains a balance between control by developers and autonomous, learning AI. The method is a novel combination of hierarchical task network planning and reinforcement learning. Various handcrafted strategies were encoded in an HTN within a commercial game environment called *Killzone 3*. We showed that it is possible to learn the best strategy with respect to a fixed opponent and that, when a different fixed opponent is introduced, the method has the ability to change its counterstrategy accordingly. Initially the method can be trained offline against a baseline to bootstrap it e.g. before shipment. From that basis, it has the ability to learn against a different fixed opponent as humans apply their strategies against it.

What was somewhat surprising is the good performance of the simple handcrafted strategies against the baseline, which is the current C++ version in *Killzone 3*. We think this is because the C++ baseline has the ability to change its priorities during an episode and often flip flops between them without first finishing one, whereas our method selects a single strategy at the start of an episode using the HTN-planner and sticks to this strategy during the entire episode. Details like how to approach a certain area with a squad do vary between episodes, but the high level strategy remains the same. Without proper heuristics that can aid in making decisions on when to (partially) switch your strategy or sufficient information about the environment encoded in the state without blowing up the state space, it might be better to stick to the plan constructed at start. A potential pitfall lurks in the design of the hierarchical task network. As stated in chapter 5, when a branch is unable to decompose due to failing preconditions the sibbling branch that gets decomposed because of the recursive nature of the algorithm might not be optimal. Currently the method does not support multiple learnable tasks in the tasklist within a single branch. This would cause several values to return under a branch.
and the method has no way of coping with that. Although the current strategies do not require this type of structure, more complex strategies might. A possible solution would be to apply some linear function to the values returned, but this might not be as trivial as it sounds in terms of desired behavior.

Some other points for future work are the following. Currently the static state holds the faction and the map on which the episode is played. It might be interesting to add some features about the enemy which can discriminate between the strategies the enemy plays. Features like the amount of enemy squads and the size of the squads, or how aggressive the enemy is during attacks. These features could paint a profile of the enemy that allows for more specific behavior which require less adjustments during online learning resulting in faster convergence.

Another area that might be interesting for further investigation is to treat the problem like a full MDP, thus moving away from the contextual bandit approach. The main challenge would lie in defining the state space such that it will not blow up i.e. determining all the relevant variables that describe the environment.
Appendix A

C & H Domain

```plaintext
(: domain capture_and_hold
// ================================================= =======================
// CAPTURE_AND_HOLD
// ================================================= =======================
(: learnable-method (capture_and_hold)
  ( branch_steam_roll
    (steam_roll)
  )
  ( branch_capture_and_split
    (capture_and_split)
  )
  ( branch_divide_and_conquer
    (divide_and_conquer)
  )
)
// ================================================= =======================
// CREATE_SQUADS
// ================================================= =======================
(: method (create_squads ?inp_distribution ?inp_sum)
  ( branch_squads_created
    (call eq (call get_list_length ?inp_distribution) 0)
  )
  ( branch_create_squad
    (call create_squad)
  )
)
```
(add_players ?player_list ?size ?squad_index)
(!start_command_sequence ?squad 2 1)
(!order_squad_custom ?squad start_new_operation)
(create_squads ?distribution (call get_list_sum ?distribution))
)

// ADD_PLAYERS
(:method (add_players ?player_list ?inp_size ?inp_squad_index)
  (branch_players_added
    (or (call eq (call get_list_length ?inp_player_list) 0)
      (call lt ?inp_size 1.0)
    )
    (call dec ?inp_size)
  )
  )
  (branch_add_player
    (and (= ?player_index (call get_list_item ?inp_player_list 0))
      (call add_member ?inp_squad_index ?player_index)
    )
    (add_players ?player_list (call dec ?inp_size) ?inp_squad_index)
  )
)

// STRATEGY
(:learnable-method (capture_and_split)
  (branch_area_sequence_012_distribution_122
    (create_squads (1.0 2.0 2.0) 5.0)
    (assign_areas (0 1 2))
  )
  (branch_area_sequence_201_distribution_212
    (create_squads (2.0 1.0 2.0) 5.0)
    (assign_areas (2 0 1))
  )
  (branch_area_sequence_120_distribution_311
    (create_squads (3.0 1.0 1.0) 5.0)
    (assign_areas (1 2 0))
  )
  (branch_area_sequence_210_distribution_111
    (create_squads (1.0 1.0 1.0) 3.0)
    (assign_areas (2 1 0))
  )
  (branch_area_sequence_102_distribution_211
    (create_squads (2.0 1.0 1.0) 4.0)
    (assign_areas (1 0 2))
  )
)
(branch_area_sequence_021_distribution_311
 (create_squads (3.0 1.0 1.0) 5.0)
 (assign_areas (0 2 1))
 )
)

// ASSIGN_AREAS
(:method (assign_areas ?inp_area_sequence)
 (branch_areas_assigned
 (and (call eq (call get_list_length ?inp_area_sequence) 0)
 )
 )
)

// ASSIGN_AREA
 (:method (assign_area ?inp_area_index ?inp_squad_list)
 (branch_area_assigned
 (and (call eq (call get_list_length ?inp_squad_list) 0)
 )
 )

// ASSIGN_AREA
 (:method (assign_area ?inp_area_index ?inp_squad_list)
 (branch_capture_and_hold
 (and (= ?squad_index (call get_list_item ?inp_squad_list 0))
 (call eq ?squad_index ?inp_area_index)
 (call eq ?squad_index (call get_list_item ?inp_area_sequence 0))
 )
 )

// ASSIGN_AREA
 (:method (assign_area ?inp_area_index ?inp_squad_list)
 (branch_capture_and_move_on
 (and (= ?squad_index (call get_list_item ?inp_squad_list 0))
 (call eq ?squad_index ?inp_area_index)
 (call eq ?squad_index (call get_list_item ?inp_area_sequence 0))
 )
 )
// order clear area filter
(! order_squad_custom ? squad advance (call get_area_waypoint ? inp_area_index) auto)
(assign_area ? inp_area_index ? squad_list)

// STRATEGY
// DIVIDE_AND_CONQUER

(:learnable-method (divide_and_conquer)
  ( branch_squad_distribution_111
    (create_squads (1.0 1.0 1.0) 3.0)
    (defend_areas (0 1 2))
  )
)

(:method (defend_areas ? inp_area_list)
  (branch_areas_defended
    (and (call eq (call get_list_length ? inp_area_list) 0)
        )
  )
)

(:branch_defend_area
  (and (= ?area_index (call get_list_item ? inp_area_list 0)
      (= ?area_list (call remove_list_item ? inp_area_list 0))
      (= ?squad (call get_squad ? area_index))
    )
  (order_squad_custom ? squad start_new_operation)
  // order clear area filter
  (order_squad_custom ? squad announce destination_waypoint (call get_area_waypoint ? area_index))
  (order_squad_custom ? squad defend (call get_area_marker ? area_index)
    auto)
  // order clear area filter
  (order_squad_custom ? squad advance (call get_area_waypoint ? area_index)
    auto)
  (end_command_sequence ? squad)
  (defend_areas ? area_list)
  )
)

// STRATEGY
(: learnable-method (steam_roll)
  ( branch_area_sequence_012_distribution_1
    ( create_squads (1.0) 1.0)
    ( capture_areas (0 1 2))
  )
  ( branch_area_sequence_201_distribution_1
    ( create_squads (1.0) 1.0)
    ( capture_areas (2 0 1))
  )
  ( branch_area_sequence_120_distribution_1
    ( create_squads (1.0) 1.0)
    ( capture_areas (1 2 0))
  )
  ( branch_area_sequence_210_distribution_1
    ( create_squads (1.0) 1.0)
    ( capture_areas (2 1 0))
  )
  ( branch_area_sequence_102_distribution_1
    ( create_squads (1.0) 1.0)
    ( capture_areas (1 0 2))
  )
  ( branch_area_sequence_021_distribution_1
    ( create_squads (1.0) 1.0)
    ( capture_areas (0 2 1))
  )
)

// CAPTURE AREAS

(: method (capture_areas ?inp_area_list)
  ( branch_areas_captured
    (call eq (call get_list_length ?inp_area_list 0)
      (call get_last_created_squad)
      (call get_squad ?squad_index)
    )
    (! end_command_sequence ?squad)
  )
)

(branch_capture_area
  (call get_list_item ?inp_area_list 0)
  (call get_last_created_squad)
  (call get_squad ?squad_index)
  (! order_squad_custom ?squad announce destination_waypoint)
  (! order_squad_custom ?squad capture)
)
// order clear area filter
order_squad_custom ?squad advance (call get_area_waypoint ?area_index auto)
(capture_areas ?area_list)
)
Bibliography


