Integrating Domain knowledge into Monte Carlo Tree Search for Real-Time Strategy Games

Yang, Zuozhi

https://researchdiscovery.drexel.edu/esploro/outputs/doctoral/Integrating-Domain-knowledge-into-Monte-Carlo/991018527910304721/filesAndLinks?index=0

Yang, Z. (2022). Integrating Domain knowledge into Monte Carlo Tree Search for Real-Time Strategy Games [Drexel University]. https://doi.org/10.17918/00001114

Repository homepage: https://researchdiscovery.drexel.edu/
Contact: libsystems.drexel.edu
Open
Downloaded On 2025/05/15 13:51:37 -0400

Please do not remove this page

GRADUATE THESIS APPROVAL FORM AND SIGNATURE PAGE

Instructions: This form must be completed by all doctoral students with a thesis requirement. This form MUST be included as page 1 of your thesis via electronic submission to ProQuest.

Thesis Title: Integrating Domain knowledge into Monte Carlo Tree Search for Real-Time Strategy Games

Author's Name: Zuozhi Yang

Submission Date: 06/22/2022

The signatures below certify that this thesis is complete and approved by the Examining Committee.

Role: Chair Name: Santiago Ontanon
Title: Associate Professor
Department: Computing
Approved: Yes Date: 06/22/2022

Role: Member Name: Vasilis Gkatzelis
Title: Associate Professor
Department: Computing
Approved: Yes Date: 06/22/2022

Role: Member Name: Edward Kim
Title: Associate Professor
Department: Computing
Approved: Yes Date: 06/22/2022

Role: Member Name: Dario Salvucci
Title: Professor
Department: Computing
Approved: Yes Date: 06/22/2022

Role: Member Name: Michael Buro
Title: Professor
Institution: University of Alberta
Approved: Yes Date: 06/22/2022

Integrating Domain knowledge into Monte Carlo Tree Search for

Real-Time Strategy Games

A Thesis

Submitted to the Faculty

of

Drexel University

by

Zuozhi Yang

in partial fulfillment of the

requirements for the degree

of

Doctor of Philosophy

June 2022

© Copyright 2022
Zuozhi Yang.

This work is licensed under the terms of the Creative Commons Attribution-ShareAlike 4.0 International license. The license is available at
http://creativecommons.org/licenses/by-sa/4.0/.

Acknowledgments

First and foremost, I would like to thank my advisor Dr. Santiago Ontañón, who has given me excellent guidance, support, and encouragement. This work would not be possible without him, especially during the tough time during the COVID-19 pandemic. Then I want to thank my friends and colleagues in the GAIMS lab, who has always been an inspiration to me. I also want to thank my dissertation committee members for their thoughtful and timely feedback. Last but not least, I want to thank my family, who has always been there for me. Drexel University and the city of Philadelphia have been my lovely home for the past five years, and I will cherish the experience and the memories forever in my heart.

Table of Contents

List of Tables

List of Figures

Abstract

Integrating Domain knowledge into Monte Carlo Tree Search for
Real-Time Strategy Games
Zuozhi Yang
Santiago Ontañón, Ph.D.

Tree search algorithms are widely applied methods to model and solve sequential decision problems. In particular, the family of sampling-based tree search algorithms called Monte Carlo Tree Search (MCTS) has had great success in problems with large branching factors. However, Real-Time Strategy (RTS) games offer a challenging testbed for tree search algorithms due to their large combinatorial action spaces, partial observability, simultaneous moves, and other factors, making them beyond the grasp of even current MCTS algorithms. This thesis makes contributions towards scaling MCTS algorithms to become more effective and efficient in the domain of RTS games. Specifically, this thesis contributes on the following problems. Firstly, we explore the problem of the integration of MCTS and domain knowledge, in the form of unit-action probability distributions, state evaluation functions, and scripted bots. Secondly, we investigate the optimization of gameplay/rollout policies for MCTS. Third, we study methods for self-learning in MCTS, where tree and/or rollout policies are bootstrapped directly from MCTS behavior iteratively.

Chapter 1: Introduction

Tree search algorithms are widely applied methods to model and solve sequential decision problems. Classic methods based on minimax search achieved impressive results in games like Chess[1]. In Chess, there are about 10^{43} game states, the branching factor is about 36, and the average game length is about 80 moves[2]. However, it is usually not efficient to systematically search the game tree in more complex domains. For example, in the game of Go, there are approximately 10^{170} possible game states, the branching factor on average is 250, and the average game length is about 150 moves[2]. This makes the game trees resulting in Go too large for minimax-based search algorithms to handle. Thus, a family of sampling-based tree search algorithms, called Monte Carlo Tree Search (MCTS), was proposed to solve the problem[3].

However, for more complicated video games like Real-Time Strategy games, MCTS struggles to scale due to many factors. First, the combinatorial structure leads to a very large branching factor in RTS games. The vanilla MCTS algorithm does not leverage the combinatorial structure in the action space. There is recent work on tackling this issue[4;5], yet there are still space for improvement in this direction. Moreover, although there are many works on incorporating domain knowledge into MCTS, how to utilize different forms of domain knowledge in RTS games still has many open problems. Specifically, domain knowledge can be policy-based and value-based. The two types require different treatment when integrated into MCTS. Additionally, in RTS games, domain knowledge is often in the form of scripted bots (scripts for short), which is one of the types of knowledge we investigate how to exploit in this thesis.

Unlike classical tree search methods, which build the full search tree systematically, MCTS builds the search tree in a stochastic way using Monte Carlo methods and prioritizes some branches over others using a *tree policy*. Developing *tree policies* to suit the needs of specific problems is one of the central topics of MCTS research. Most of the tree policies find their root in the multiarmed bandit problem (MAB)[6]. The MAB problem is a powerful framework for decision-making over time under

uncertainty. It is an online learning model concerns the problem of exploration/exploitation tradeoff. A Bandit problem captures the situation when an agent observes the reward for the choice it made at a given round, called an *arm*, but the agent does not observe the reward for the other arms. Thus, the agent needs to explore, which means it needs to try out different arms to gather information. But due to the limited trial budget, the agent needs to also exploit, which means placing more focus on the arm that looks the most promising. The bandit problem and exploration/exploitation tradeoff are important when the number of trials is limited, or the cost for each trial is high. We will introduce the commonly used optimization objective, called *regret*, and classic algorithms for various of settings.

In this thesis, we study different ways to incorporate domain and learned knowledge into MCTS in RTS games, including learning evaluation functions and tree policies, guiding MCTS directly using scripted bots, optimizing adaptive policies, and self-learning from RTS games. The specific contributions are described below.

The remainder of the document is structured as follows: In the rest of this chapter we present the list of contributions and publications. Then, in Chapter 2, we first introduce the background on RTS games, Monte Carlo Tree Search, and bandit algorithms. In Chapter 3, we describe our work on integrating domain knowledge with MCTS in two different forms. In Chapter 4, we introduce our work on guiding the MCTS using scripted bots directly. After that, we describe our work on bandit-based optimization of the gameplay and rollout policies and the work on applying advanced types of bandit algorithms for MCTS and RTS games, such as combinatorial bandits and contextual bandits in Chapter 5. In Chapter 6, the problem of self-learning via supervised learning for MCTS is investigated. Finally, we summarize and discuss future work in Chapter 7.

1.1 List of Contributions

The contributions in the category of using machine learning methods to model the domain knowledge are in Chapter 3. In this chapter, we explored the integration of two forms of domain knowledge, as the bias for tree policy and as the state evaluation function.

- A study of using statistical machine learning models to learn gameplay policies to bias the tree policy of the MCTS algorithm (Section 3.1). The study compared the performance of the models under iteration budget and time budget. And we concluded that although more complex models can have better performance in classification tasks and in biasing the tree policy under iteration budget, simpler models have better performance in more realistic settings like biasing the tree policy under time budget for its fast inference speed.

- A study of learning evaluation functions from replay data using deep learning algorithms (Section 3.2). The proposed model focused on the problem of training the evaluation function on small map sizes and generalize to larger map sizes. And it is confirmed in the RTS game environment, evaluation function learned from smaller maps can be transfer to larger maps in some degree.

The contributions in the category of guiding the MCTS directly using scripted bots are in Chapter 4. In this study, we explore a few variations of methods of integrating scripted bots into MCTS.

- Specifically, we apply the guidance from scripted bots at the expansion and selection stage of the MCTS algorithm. The integration of scripts showed great performance improvements in the experiments, especially in larger maps.

- Lastly, our experiments also showed that using the scripts as the rollout policy directly is not a good idea potentially due to the bias in the scripted bots.

The next contribution we made in Chapter 5 in the bandit-based policy optimization. In this chapter, we investigated the problem of optimizing gameplay policies from scratch using a very simple parameterization.

- Through this optimization method and parameterization, we further explore the optimization of adaptive gameplay policies using the concept of combinatorial contextual bandits. We proposed several optimization methods for combinatorial contextual bandits and compared the trade-offs between efficiency and complexity.

In Chapter 6, we described our last contribution, self-learning from MCTS in RTS games. In this chapter, we take the framework of InformedMCTS described in Section 3.1 further.

- Instead of learning from replays, we make the model learn from models of previous iterations and study what the effective ways of learning are. We tested using the model as the bias for tree policy, as the rollout policy, and as both. The result showed that using the model as the bias for tree policy in effective while using the model as rollout policy alone is not. And the most effective way to promote self-learning is to use the model as both the bias for tree policy and rollout policy at the same time.

1.2 List of Publications

- **Yang, Z.**, Ontañón, S. 2021. Contextual Combinatorial Bandits in Real-Time Strategy Games *CoG 21'*
- **Yang, Z.**, Ontañón, S. 2021. An Experimental Survey on Methods for Integrating Scripts into Adversarial Search for RTS Games *ToG 21'*
- **Yang, Z.**, Ontañón, S. 2020. Bandit-Based Policy Optimization for Monte Carlo Tree Search in RTS Games *AI for Strategy Game Workshop at AIIDE 20'*
- **Yang, Z.**, Ontañón, S. 2020. Are Strong Policies also Good Playout Policies? Playout Policy Optimization for RTS Games *AIIDE 20'*
- **Yang, Z.**, Ontañón, S. 2020. Integrating Search and Scripts for Real-Time Strategy Games: An Empirical Survey *RLG Workshop at AAAI 20'*.
- **Yang, Z.**, Ontañón, S. 2019. Guiding Monte Carlo Tree Search by Scripts in Real-Time Strategy Games *AIIDE 19'*
- **Yang, Z.**, Ontañón, S. 2019. Extracting Policies from Replays to Improve MCTS in Real Time Strategy Games *KEG Workshop at AAAI 19'*
- **Yang, Z.**, Ontañón, S. 2018. Learning Map-Independent Evaluation Functions for Real-Time Strategy Games *CIG 18'*

Chapter 2: Background

In this chapter, we first review the RTS game environment and its challenges. Then we survey the algorithmic tools used to tackle the problem starting from bandit algorithms. Based on bandit algorithms, Monte Carlo Tree Search is a family of real-time stochastic tree search algorithms that can be applied to RTS games. Lastly, we go into the detailed development of algorithm design for MCTS in RTS games.

2.1 Real-Time Strategy Games

RTS is a sub-genre of strategy games where players aim to defeat their opponents (destroying their army and base) by strategically building an economy (gathering resources and building a base), military power (training units and researching technologies), and controlling those units. From an AI point of view, the main challenges of RTS games with respect to traditional board games like Chess or Go are: (1) the combinatorial growth of the branching factor as a function of the number of units being controlled[7], (2) limited computation budget between actions due to the real-time nature, (3) they are simultaneous move games (more than one player can issue actions at the same time) with durative actions (actions are not instantaneous), and (4) lack of forward model in most of the research environments like Starcraft (necessary to perform lookahead search). Additionally, some RTS games are not fully observable or not deterministic.

Because of these reasons, RTS games have been receiving an increased amount of attention as they are an excellent platform to study the aforementioned problems. There has been many work using commercial RTS games, such as StarCraft BroodWar, WarCraft, Age of Empires, StarCraft II, etc., as the environment[8]. Also, many research environments and tools, such as ORTS[9], TorchCraft[10], SCIILE[11], μRTS[7], ELF[12], and Deep RTS[13] have been developed to promote research in the area. Specifically, in this document , to stay focused on the problem of interest of this work, we chose a single evaluation platform, μRTS, as our experimental domain, as it offers a forward model for game

tree search approaches such as minimax or Monte Carlo Tree Search, and many of the algorithms to be compared were already readily available in this platform.

2.1.1 μRTS

μRTS[1] is a simplified RTS game designed for AI research. μRTS provides the essential features that make RTS games challenging from an AI point of view: simultaneous and durative actions, combinatorial branching factors, and real-time decision making. The game can be configured to be partially observable and non-deterministic, but those settings are turned off for all the experiments presented in this work. In the default configuration, μRTS defines only four unit types and two building types, all of them occupying one tile in the map, and there is only one resource type. Additionally, as required by our experiments, μRTS allows maps of arbitrary sizes and initial configurations. Although the set of units can be configured in μRTS, we used the default configuration, with the following units:

- Base: can train Workers and accumulate resources.
- Barracks: can train attack units.
- Worker: collects resources and constructs buildings.
- Light: low power but fast melee unit.
- Heavy: high power but slow melee unit.
- Ranged: long-range attack unit.

Additionally, the environment can have walls to block the movement of units. An example screenshot of the game is shown in Figure 2.1. The squared units in green are Minerals with numbers on them indicating the remaining resources. The units with blue outline belong to player 1 (which we will call *max*), and those with red outline belong to player 2 (which we will call *min*). The light grey squared units are Bases with numbers indicating the number of resources owned by the player, while the darker grey squared units are the Barracks. Movable units have round shapes with grey units

[1] `https://github.com/santiontanon/microrts`

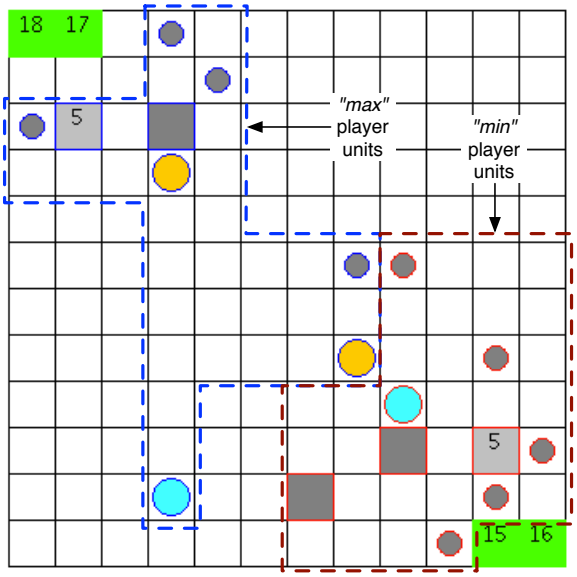


Figure 2.1: A Screenshot of μRTS.

being Workers, orange units being Lights, yellow being Heavy units (now shown in the figure), and blue units being Ranged.

2.2 The Multi-Armed Bandit Problem

The multiarmed bandit problem has been studied by many disciplines for decades[14]. The name of the problem comes by analogy with slot machines in casinos, which are often called "one-armed bandit" (that rob people of their money). Multi-armed bandit refers to the problem where the player sits in front of a slot machine with multiple arms to pull, and each of them has a different reward distribution. When an arm is pulled, a stochastic outcome will be drawn from the corresponding distribution. Thus the player faces the dilemma of exploration versus exploitation: the player can either choose to exploit the arms that have yielded higher payoffs already or continue to explore arms that have been tried less and that might be potentially more profitable, but he/she does not know yet. The goal is to design sequential strategies that balance the trade-off of exploration and exploitation and maximize the expected reward achieved by the player over time.

The standard setting of bandits is the stochastic bandits, which makes the assumption that the rewards of each arm are independent and identically distributed (IID). We will cover the general

settings of the stochastic bandit, its applications, and the metric of comparing different algorithms.

2.2.1 Stochastic Bandits

Stochastic bandit is the basic model of the bandit problem. The model assumes the rewards of each arm are independent and identically distributed (IID) random variables. An agent has K available actions, also called arms, to choose from and T total rounds of choosing, which may or may not be given beforehand. We also denote arms using a. Each arm a is associated with an unknown reward function μ_a, which takes values in the interval $[0, 1]$ following a probability distribution D_a. In each round, a stochastic reward is collected after an arm is chosen. The typical goal of the agent is to maximize the cumulative reward over the T rounds. There are three basic assumptions that are made in the stochastic bandit setting:

- The agent receives *bandit feedback*, which means that the reward is only observed for the arm chosen and nothing else.
- The reward for each arm is IID. For each arm a, there is a reward distribution D_a. The reward is sampled "independently" from this distribution when the corresponding arm is chosen.
- Per-round rewards are bounded.

2.2.2 Applications of Stochastic Bandits

Bandit problems were first introduced by Thompson[15], concerning the problem of increasing the efficiency of medical trials. Since then, the bandit problem has been one of the central topics of decision making under uncertainty and has influenced many fields, such as reinforcement learning and MCTS, that need to balance exploration and exploitation. Moreover, it has been widely applied in real-world problems like network optimization[16], recommender systems[17], and advertisement placement[18].

2.2.3 Regret

How do we compare whether an algorithm is better than another in bandit problems? The standard approach is to compare the algorithm's cumulative reward to the expected reward of always playing

an optimal arm a^* with an expected reward of μ^*. The cumulative reward of always selecting the optimal arm is thus $\mu^* \cdot T$. Formally, we can define the *cumulative regret*, $R(T)$ at round T as:

$$R(T) = \mu^* \cdot T - \sum_{t=1}^{T} \mu(a_t) \tag{2.1}$$

where a_t denotes the arm chosen by the bandit at round t. Since r_t, the reward obtained at round t, is a random variable, $R(T)$ is also a random variable. For analysis, we will typically use expected cumulative regret $E[R(T)]$, and we refer to it as regret or expected regret, and the quantity $R(T)$ is sometimes called pseudo-regret or realized regret in the literature.

However in some applications, there might be other types of regret like simple regret or instantaneous regret. Instantaneous regret is the difference between μ^* and the reward obtained by the last selected arm. And simple regret refers to the difference between μ^* and the reward of the arm believed to be the best at round T.

Lai and Robbins[19] showed in their classic paper that

$$E[\mu(a_t)] \leq \left(\frac{1}{D(D_a||p^*)} + o(1) \right) \ln t \tag{2.2}$$

where $D(D_a||p^*)$ is the Kullback-Leibler divergence of any non-optimal arm reward density D_a, and the optimal arm reward density p^*, which, since rewards range from 0 to 1 in this context, is defined as:

$$D(D_a||p^*) \coloneqq \int_0^1 D_a(x) \ln \frac{D_a(x)}{p^*(x)} dx \tag{2.3}$$

Lai and Robbins[19] proved that this regret is the best possible, which means for any policy and for any suboptimal arm a, $E[\mu(a_t)] \geq (\ln t)/D(D_a||p^*)$ asymptotically.

2.3 Overview of Bandit Algorithms

In this section, we describe a collection of basic algorithms for stochastic bandits. A strategy or algorithm used to solve the multi-armed bandit problem is often called a policy. We start with

the family of the semi-random policies, which depend on a parameter ϵ to determine how much exploration takes place. Then we describe a family of policies called Upper Confidence Bound (UCB) and one of its variants called UCB1, which achieved logarithmic regret uniformly over time. Finally, we describe a probability-matching-based policy called Thompson sampling.

2.3.1 ϵ-greedy and Variants

ϵ-greedy is the simplest and most widely used bandit algorithm in practice. With probability ϵ, the agent selects an arm at random (exploration) , and with probability $1 - \epsilon$, the agent selects the arm with the best empirical mean so far (exploitation). More generally, algorithms that maintain a distinction between the exploration phase and exploitation phase are called *semi-uniform*. In this form of ϵ-greedy the constant factor ϵ prevents the cumulative regret from getting arbitrarily close to zero. Thus a straightforward modification is to choose a function to gradually decrease ϵ to zero. Such a strategy is called ϵ-decreasing[20].

A simple variation of the ϵ-greedy algorithm is ϵ-first[20], which is also called explore-then-commit. The ϵ-first strategy does all the exploration at the beginning, then commits to the best one. Note that ϵ-first requires the total number of arm pulls T to be known beforehand. For the first ϵT trials, all the arms are tried for an equal number of times. And for the rest $(1-\epsilon)T$ trials, the agent always selects the arm with the best empirical mean. In web optimization and testing in the industry, ϵ-first is widely used. Specifically for 2-armed bandits is widely known as "A/B testing".

With a constant value of ϵ, a linear bound on regret can be achieved. Constant ϵ-greedy policies are suboptimal in terms of regret growth bound as a constant ϵ prevents the strategy from asymptotically reaching the optimal arm.

2.3.2 Upper Confidence Bounds

Both ϵ-first and ϵ-greedy are bandit algorithms that the exploration schedule does not depend on the history of the observed rewards. Upper Confidence Bounds (UCB) is a well-celebrated family of bandit algorithms that overcome this limitation and does explore adaptively[21]. The UCB family takes the form of picking the arm the maximize the surrogate function,

$$D(D_a||p^*) \coloneqq \int_0^1 D_a(x) \ln \frac{D_a(x)}{p^*(x)} dxi = \arg\max_i (\overline{X}_i + P_i) \tag{2.4}$$

where X_i is the empirical estimation of the mean reward of arm i and P_i is the confidence bound used to approximate the uncertainty of X_i. The algorithm design is based on the *optimism in the face of uncertainty* principle because we are choosing based on mean plus the confidence bound (which is the "upper confidence bound").

UCB1

Specifically, the most commonly used algorithm in the UCB family is called UCB1[6]. The policy dictates to play arm i that maximizes

$$\text{UCB1} = \overline{X}_i + \sqrt{\frac{2 \ln n}{n_i}} \tag{2.5}$$

where X_i is the average reward from arm i, n_i is the number of times arm i was played, and n is the overall number of plays so far. The reward term $\overline{X}_i$ encourages the exploitation of higher-reward choices, while the right-hand term encourages the exploration of less-visited choices. The padding function of the UCB1 algorithm is derived from the right tail distribution of the Chernoff-Hoeffding bound.

Upper Confidence Bounds for Trees (UCT)[22] is a variation of MCTS that applies UCB1 to Monte Carlo Tree Search (see later section for MCTS). Kocsis and Szepesvari showed that the bound on the regret of UCB1 still holds in the case of non-stationary reward distributions. They also showed that the probability of selecting a suboptimal action converges to zero at a polynomial rate as the number of simulations goes to infinity. Therefore, given enough time, UCT allows MCTS to converge to the minimax tree and is thus optimal[22].

Much research in regret bounds shows regret that is logarithmic "optimal" only asymptotically.[6] present that UCB1 achieves expected logarithmic regret uniformly over time, for all reward distributions, with no prior knowledge of the reward distribution required.

Algorithm 1: Thompson Sampling for Beta-Bernoulli Bandits

for $t = 1, 2, 3, \ldots$ **do**
  for $k = 1, 2, 3, \ldots, K$ **do**
    Sample $\hat{\theta}_k \sim beta(\alpha_k, \beta_k)$
  $a_t \leftarrow \arg\max_k \hat{\theta}_k$
  Choose a_t and observe r_t
  $\alpha_{a_t} \leftarrow \alpha_{a_t} + r_t$
  $\beta_{a_t} \leftarrow \beta_{a_t} + (1 - r_t)$

2.3.3 Thompson Sampling

Thompson sampling[15] is designed under the principle of probability match. The idea is to choose each arm stochastically at each round based on their currently estimated probability of being the best. The probability matching is achieved by sampling each empirical distribution once and selecting the one with the maximum sample. It has been shown that the repeated selection of the maximum of a single draw from each distribution produces an estimate and selection behavior of the optimal distribution. The algorithmic description for Thompson sampling in Beta-Bernoulli Bandits is shown in Algorithm 1. Beta-Bernoulli Bandits are one type of bandit problems that the rewards has the beta distribution.

2.4 Beyond Stochastic Bandits

So far, we have introduced stochastic bandit that assumes IID reward distributions. However, in many applications, such assumption does not hold. For example, in game AI research, the adversarial bandits can be used to address partial observability and simultaneous move. And in recommender systems, we need to personalize to cater to each customer's interest. Thus we need more general models for these problems. For example, adversarial bandits deal with non-stationary reward distributions and contextual bandit allows access to side information about each arm (ignoring such information usually make the problem highly non-stationary as well). Other variants we explore include the case where the number of arms is huge, while there is combinatorial structure that can be utilized. We also look at problems beyond bandits, such as pure exploration problems.

2.4.1 Adversarial Bandits

The adversarial bandit problem[23] is a more general bandit setting than stochastic bandits, where rather than select from an IID distribution, the rewards are selected by an adversary per-play. The problem progress in a iterative manner as follows:

- the adversary picks the reward distribution
- the agent picks an arm (without knowing the adversary's selection)
- the agent observes a reward

The interaction between learner and adversary can be framed as a two-player zero-sum game between the learner and adversary. The moves for the adversary are the possible reward distributions, and for the learner they are the policies. The pay-off for the adversary is the regret and the pay-off for the agent is the negated regret. From game theory it is obvious that the learned policies need to be stochastic to avoid being exploited.

Exp3

The first algorithm we look at for adversarial bandits is called the exponential-weight algorithm for exploration and exploitation (Exp3)[23]. This algorithm is related to an algorithm called Hedge for the full-feedback problem[24]. The full-feedback problem differs from the bandit problem that rather than selecting a single arm, in the full-feedback settings, at each round t the agent needs to decide a distribution $\mathbf{p}^t$ (known as an allocation) over all the arms, and each arm will receive feedback c_a^t (so, the agent gets a reward for all arms, rather than only for a single arm). The Hedge algorithm is shown in Algorithm 9 is designed to tackle the problem (as originally formulated, the algorithm uses *cost* rather than *reward*, but the idea is the same, just aiming to minimize cost, rather than maximize reward). The hyper-parameter γ is used to balance the exploration and exploitation of the algorithm.

The key ingredient of adapting the full-feedback Hedge algorithm to the bandit-feedback case is to utilize a mechanism to estimate the rewards for arms not played. A simple one is an importance-

Algorithm 2: Hedge algorithm for online learning with experts

parameter : $\gamma \in (0, 1)$;
Initialize the weights as $\mathbf{w}^1 \in [0, 1]^N$ with $\sum_{a=1}^{N} w_a^1 = 1$.;
for *each round* t **do**
    Decide the distribution

$$\mathbf{p}^t = \frac{\mathbf{w}^t}{\sum_{a=1}^{N} w_a^t} \tag{2.6}$$

    Observe cost vector $\mathbf{c}^t \in [0, 1]^N$ from environment. ;
    Suffer the cost $\mathbf{p}^t \cdot \mathbf{c}^t$;
    for *each arm* a **do**
        $w_a^{t+1} = w_a^t \gamma^{c_a^t}$

weighted estimator:

$$\hat{X}_{ti} = \frac{\mathbb{I}\{A_t = a\} X_t}{p_t(a)} \tag{2.7}$$

$\mathbb{I}$ is the indicator function, A_t is the action played at round t, and $p_t(a)$ is the conditional probability that arm a is played at round t. An alternative is called loss-based importance-weighted estimator:

$$\hat{X}_t(a) = 1 - \frac{\mathbb{I}\{A_t = a\} X_t}{p_t(a)}(1 - X_t) \tag{2.8}$$

After applying the estimators and exponential weighting, we have the Exp3 algorithm. Exp3 has the expected regret of:

$$E[R(T)] \leq O(\sqrt{KT \log N}), \tag{2.9}$$

where K is the number of arms, T is the number of trials, and N is the number of experts.

2.4.2 Contextual Bandits

In real-world bandit problems, usually there is extra information, we refer to as context, that can be leveraged to predict the expected reward of the arms. For example, in recommender systems, user history and user demographics are useful information that can help the system to propose what to show to the user next[25]. The algorithms discussed so far do not make use of this kind of information. Thus, in this section, we introduce the contextual bandit framework that takes the

Algorithm 3: Exp3

parameter : $\gamma \in (0, 1)$
Initialize the total estimated reward as $\hat{S}_0(a) = 1$ for each arm a.
for *each round* t **do**
    Calculate the sampling distribution

$$p_t(a) = \frac{\exp\left(\gamma S_t(a)\right)}{\sum_{a'=1}^{K} w_t(a')}$$

    Sample an arm a_t from distribution $p_t(\cdot)$.
    Observe reward $c_t(a_t)$ for arm a_t.
    for *each arm* a **do**

$$S_{t+1}(a) = S_t(a) + 1 - \frac{\mathbb{I}\{A_t = a\} X_t}{p_t(a)}(1 - X_t)$$

addiction information into account.

The contextual bandit setting has taken many names, including bandits with context, bandit problems with covariates, generalized linear bandits, associative bandits, and bandit problems with expert advice. The contextual bandit problem is closely related to work in machine learning on supervised learning and reinforcement learning. In contextual bandits, rewards in each round depend on a context, which is observed by the algorithm prior to making a decision. The basic protocol of the problem is structured as follows.

For each round $t \in T$:

1. learner observes a context x_t,
2. learner picks an arm a_t according to x_t,
3. reward r_t is revealed.

The agent's aim is to collect information about the relationship between the context and rewards so that the agent can use the information to predict the best arm to play given the current context. Different algorithms are proposed to solve this problem, such as LinUCB[25], which assumes a linear dependency between the expected reward the context of an action.

2.4.3 Combinatorial Bandits

A combinatorial bandit is a bandit problem with an action set that is a subset of the d-dimensional binary hypercube: $A \subset \{0,1\}^d$. The combinatorial structure indicates each element in A is d-dimensional, with values on or off, but some combinations are not allowed. In the literature, the class of A is called global arms, and the class of $\{0,1\}^d$ is called local arms.

For the bandit feedback setting, in each round, a reward is observed for the chosen global arm. In some literature, it is assumed that the reward is a linear combination of a reward vector X on the local arms, where $R(A) = \sum_{i=1}^{d} A_i X_i$. The second type of feedback is called semi-bandit feedback, where at each round, the reward vector X is revealed for all the local arms A_i in the chosen global arm A. The challenge in these problems lies in the enormous number of global arms, i.e., in its combinatorial structure: the size of the problem with n local arms could well grow as d^n. Examples of combinatorial bandit algorithms include combinatorial upper confidence bound (CUCB)[26] and Naïve Sampling[4].

2.4.4 Best-Arm Identification

The policies discussed so far were designed to maximize the cumulative reward. Thus, the policies must carefully balance exploration against exploitation. In best-arm Identification, there is no regret received until the final decision is made by the agent to commit to a single arm. Although the problem is very similar to the bandit problem, there are also differences.

First, the definition of regret is fundamentally different. Let v be a stochastic bandit, and π be a policy. The way to measure the performance of the policy after n rounds is through simple regret in best-arm Identification setting.

$$R_n^{simple}(v, x) = \mathbb{E}_{vx}[\Delta_{A_{n+1}}(v)] \tag{2.10}$$

There are two variants of settings of best-arm identification, fixed confidence and fixed budget[27]. The fixed confidence setting is when the learner is given a confidence level $\delta \in (0,1)$ and should use as few samples as possible to output an arm that is optimal with a probability of at least

Algorithm 4: Sequential Halving

parameter : budget n, arms k
Initialize $L = \log_2(k), A_1 = [k]$
for $l = 1, \ldots, L$ **do**
    Choose each arm in A_l exactly $\frac{n}{L|A_l|}$ times.
    Computing empirical mean for each arm.
    Let A_{l+1} contain the top $|A_l|/2$ arms in A_l

$1 - \delta$. In the other variant, the learner has to make a decision after n rounds and the goal is to minimize the probability of selecting a suboptimal arm. We describe a simple algorithm for fix-budget best-arm identification in Algorithm 4[28]. The strategy works like this: we split the budget evenly across $\log_2 n$ rounds. In each round we pull arms uniformly. At the end of each round, we eliminate the worse half of the arms by comparing the empirical mean reward. The algorithm needs at most $T = O(H_2 \times log(n) \times log(\frac{log(n)}{\delta}))$ arm pulls to succeed with probability at least $1 - \delta$, where $H_2 = max_{i \neq 1}(\frac{i}{\Delta_i^2})$ and $\Delta_i = p_1 - p_i$ and $p_1 \geq p_2 \geq \cdots \geq p_i$ are the expected rewards of the arms.

2.5 Bandits and Monte Carlo Tree Search

One of the important applications of bandit algorithms is planning. The most representative algorithm is Monte Carlo Tree Search (MCTS). Tree search algorithms solve the planning problem by sequentially building a search tree. When the search space is huge, systematically searching the tree becomes very hard and time-consuming. Bandit algorithms have been used within search algorithms to decide how to sample the search space rather than exploring it systematically, hence improving performance. By introducing the bandit algorithm into the search algorithm, we can stochastically search parts of the tree that have greater potential.

2.5.1 Development of Monte Carlo Search

Monte Carlo methods have been used extensively in games with randomness and partial information[29], but they may be applied equally to deterministic games of perfect information. Following a large number of simulated games, starting at the current state and playing until the end of the game, the initial move with the highest win rate is selected to advance the game. In the majority

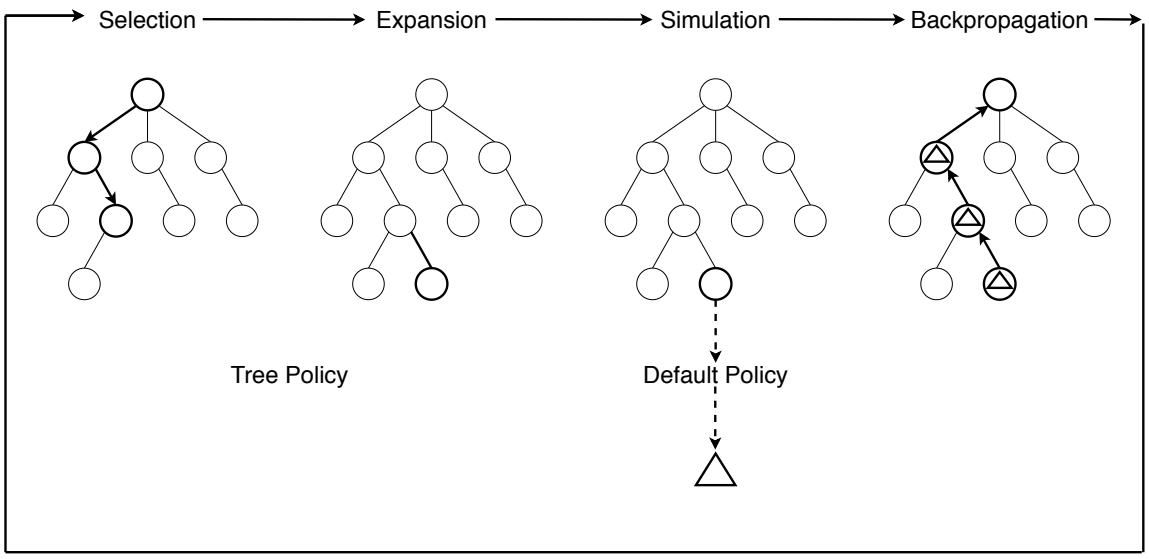


Figure 2.2: The standard MCTS algorithm

of cases, actions were sampled uniformly at random (called flat Monte Carlo) or with some game-specific heuristic bias with no game-theoretic guarantees. In other words, even in the limit, the move selected might not converge to the optimal move. Nonetheless, the power of flat Monte Carlo is demonstrated by Ginsberg[30] and Sheppard[31], who use such approaches to achieve world champion level play in Bridge and Scrabble, respectively.

Then the introduction of bandit algorithms improved the performance of MCTS and provided theoretical convergence guarantees for MCTS. The most widely applied bandit algorithm in MCTS is the UCB algorithm[6]. And the variant of MCTS combined with UCB is called UCT[22], which made huge progress in computer Go before the era of deep learning. Other bandit algorithms also have been introduced to combine with MCTS to tackle other types of problems[32].

2.5.2 Monte Carlo Tree Search

Despite the lack of game-theoretic guarantees, the accuracy of the Monte Carlo simulations may often be improved by selecting actions according to the cumulative reward of the game episodes they were part of. This may be achieved by keeping track of the states visited in a tree. In 2006, Coulom proposed a novel approach that combined Monte Carlo evaluations with tree search[3] for 9×9 Go. His proposed algorithm iteratively runs random simulations from the current state to

Algorithm 5: UCTSearch

parameter : state s_0
Create root node v_0 with state s_0
while *within computational budget* **do**
    $v_l \leftarrow$ TreePolicy(v_0)
    $\Delta \leftarrow$ DefaultPolicy$(s(v_l))$
    BackUp(v_l, Δ)
return $a(BestChild(v_0, 0))$

Algorithm 6: TreePolicy

parameter : v
while v *is nonterminal* **do**
    if v *not fully expanded* **then**
        return *Expand(v)*
    else
        $v \leftarrow BestChild(v, Cp)$
return v

Algorithm 7: Expand

parameter : v
Choose $a \in$ untried actions from $A(s(v))$
Add a new child v' to v
    with $s(v') = f(s(v), a)$
    and $a(v') = a$
return v'

Algorithm 8: BestChild

parameter : v, c

$$score = argmax \frac{Q(v')}{N(v')} + c\sqrt{\frac{2 \ln N(v)}{N(v')}} \tag{2.11}$$

Algorithm 9: BackUp

parameter : v, Δ
while s *is not null* **do**
    $N(v) \leftarrow N(v) + 1$
    $Q(V) \leftarrow Q(v) + \Delta(v, p)$
    $v \leftarrow$ parent_of v

the end of the game: nodes close to the root are added to an incrementally growing tree, revealing structural information from the random sampling episodes. In particular, nodes in the tree are selected according to the estimated probability that they are better than the current best move.

This is the very first form of MCTS.

MCTS is an algorithm proposed to find optimal decisions by taking random samples in the search space and building a search tree from the results. It has been a strong candidate for problems that can be represented as trees of sequential decisions, particularly problems that can be simulated with a forward model. The goal of MCTS is to approximate the (true) game-theoretic value of the actions that may be taken from the current state. This is achieved by iteratively building a partial search tree, as illustrated in Figure 2. How the tree is built depends on how nodes in the tree are selected. The success of MCTS, especially in Go, is primarily due to this *tree policy*.

The standard MCTS algorithm is shown in Figure 1. The basic algorithm involves iteratively building a search tree until some predefined computational budget – typically a time, memory or iteration constraint – is reached, at which point the search is halted and the best performing root action returned. Each node in the search tree represents a state of the domain, and directed links to child nodes represent actions leading to subsequent states. Four steps are applied per search iteration:

1. Selection: Starting at the root node, a child selection policy is recursively applied to descend through the tree until the most urgent expandable node is reached. A node is expandable if it represents a nonterminal state and has unvisited (i.e. unexpanded) children.

2. Expansion: One of the unexpanded child nodes is added to the tree.

3. Simulation: A simulation is run from the new node according to the default policy to produce an outcome.

4. Backpropagation: The simulation result is "backed up" (i.e. backpropagated) through the selected nodes to update their statistics.

2.5.3 Upper Confidence Bounds for Trees (UCT)

This section describes the most popular algorithm in the MCTS family, the Upper Confidence Bound for Trees (UCT) algorithm[22]. We provide a detailed description of the algorithm. In particular, Kocsis and Szepesvari proposed the use of UCB1 as tree policy[22]. The combination of MCTS and

UCB1 tree policy is called UCT. UCT treats the choice of child node as a multi-armed bandit problem. And the value of a child node is the expected reward approximated by the Monte Carlo simulations. Therefore, these rewards correspond to random variables with unknown distributions.

As described in the section for bandit algorithms, UCB1 has some promising properties: it is very simple and efficient and guaranteed to be within a constant factor of the best possible bound on the growth of regret. It is thus a promising candidate to address the exploration-exploitation dilemma in MCTS: every time a node (action) is to be selected within the existing tree, the choice may be modeled as an independent multi-armed bandit problem. A child node i is selected to maximize the UCT score (which has the same form as the UCB1 formula):

$$UCT = \overline{X}_i + C_p\sqrt{\frac{2\ln n}{n_i}} \tag{2.12}$$

where n is the number of times the parent node has been visited, n_i the number of times child i has been visited and $C_p > 0$ is a constant that can be tuned in practice.

2.5.4 Bandit-Based Variations

The tree policy in MCTS determines the most urgent node to visit at each level of the tree. Depending on the specific problems that MCTS is applied to, various of tree policy enhancements are proposed to improve MCTS in those problems.

The choice of bandit algorithm used in the tree largely decides the performance of the MCTS algorithm. We may use different bandit algorithms to suit the need of specific types of problems. For example, one can adopt UCB1-Tuned discussed before to MCTS implementations to tune the bounds of UCB1 more finely, and performance is improved in games such as Go[33], Othello[34], and Tron[35].

The Exploration-Exploitation with Exponential weights (EXP3) algorithm, as discussed in earlier sections, applies in the stochastic case. Again, The EXP3 policy operates as follows:

- Draw an arm from the probability distribution.
- Compute the estimated gain for each arm.

- Update the cumulative gain.

Then one can compute the new probability distribution over the arms. EXP3 has been used in conjunction with UCT to address games with partial observability and simultaneous moves. For example,[36] uses UCB1 to deal with states with only one player moving, and EXP3 to deal with states with simultaneous moves of two players.

Bayesian UCT[37] is another bandit-based variation, which incorporates a Bayesian framework to enable more accurate estimation of node values and uncertainties under a limited number of simulations. The Bayesian MCTS introduced two tree policies. The first equation is

$$\text{maximize } B_i = \mu_i + \sqrt{\frac{2 \ln N}{n_i}} \tag{2.13}$$

where μ_i replaces the empirical mean reward of node i with the mean of an extremum distribution P_i. And the second equation is

$$\text{maximize } B_i = \mu_i + \sqrt{\frac{2 \ln N}{n_i}} \sigma_i \tag{2.14}$$

where σ_i is the square root of the variance of P_i. Tesauro et al. claim that the first equation improves over UCT if the independence assumption holds and leaf node priors are correct, and the central limit theorem motivates the second equation.

2.5.5 Tree Policy Enhancements

Many enhancements manipulate the tree policy to change how MCTS explores the search tree. Generally, selection methods assign some numeric score, e.g., UCB1 scores, to each action to balance exploration with exploitation. In many domains it is also beneficial to influence the score for each action using domain knowledge, to bias the search towards/away from certain search space and makes use of other forms of reward estimate.

First Play Urgency

The MCTS algorithm has no way of determining the order to visit unexplored nodes. In a typical implementation, UCT visits each unvisited action once in random order before revisiting any using the UCB1 formula. This means that exploitation will rarely occur deeper in the tree for problems with large branching factors.

First play urgency (FPU) is a modification to MCTS proposed by Gelly and Wang[33] to address this issue by assigning a fixed value to score unvisited nodes and using the UCB1 formula to score visited nodes. By tuning this fixed value, early exploitations are encouraged.

Progressive Bias

Progressive bias is a technique for incorporating domain-specific heuristic knowledge to MCTS[38]. If a node is visited only a few times and its statistics are not reliable, more accurate information can come from a heuristic value H_i for a node i from the current position. A new term can be added to the MCTS selection formula of the form:

$$f(n_i) = \frac{H_i}{n_i + 1} \tag{2.15}$$

where the node with index i has been visited n_i times. As the number of visits to this node increases, the influence of this number decreases.

Move Pruning

Pruning techniques are powerful tools to make the search algorithms more efficient, e.g. $\alpha-\beta$ pruning significantly boosts the performance of minimax search in two-player zero-sum games. Move pruning can be beneficial for MCTS to eliminate obviously poor choices and focus on the better choices. Generally speaking, there are two classes of pruning techniques: soft pruning and hard pruning.

Progressive widening/unpruning[38;39] is an example of a heuristic soft pruning technique. Progressive widening initially limits the branching factor to a smaller number and as gradually increase the threshold of the branching factor. The idea is similar to First Play Urgency, which forces earlier exploitation. However, it is found that progressive widening without heuristic move ordering had

little effect on playing strength for the game of Havannah[40] and Go[41]. Given domain knowledge, the algorithms is able to prune actions known to lead to weaker positions and significantly increase the performance in many games[42–44].

Absolute pruning and relative pruning are two hard pruning strategies proposed by Huang[44] that can preserve the correctness of the UCB algorithm. They work like this:

- Absolute pruning prunes all actions from a position except the most visited one, once it becomes clear that no other action could become more visited.
- Relative pruning uses an upper bound on the number of visits an action has received, to detect when the most visited choice will remain the most visited.

2.5.6 Imitation Learning from MCTS

Expert Iteration (EXIT), independently proposed at the same time by[45] and[46], integrates MCTS into the training process of deep reinforcement learning to construct a strong agent that combines the two parts. Specifically, the training can be viewed as an imitation learning algorithm, where the MCTS agent serves as the expert demonstrator and the deep reinforcement learning agent serves as the learner. Between each training iteration, the learner learns from the demonstrations generated by the expert and the updated learner policy is used by the expert to generate better quality demonstrations. The policy can be used in tree policy to bias the search directions or used as a state value estimator, or both.

Furthermore, two learning targets, chosen-action-target (CAT) and tree-policy-target (TPT), were compared in[45]. CAT optimizes the Kullback–Leibler divergence between the output distribution of the network and this target. So the loss at position s is given by the formula:

$$L_{CAT} = -\log[\pi(a^*|s)] \tag{2.16}$$

where a^* is the action selected by MCTS.

For the other target, TPT, the tree policy target is the average tree policy of the MCTS at the root. In other words, it try to match the model output to the distribution over actions given by

Algorithm 10: Expert Iteration

$\hat{\pi}_0$ = initial_policy()
π^*_0 = build_expert()
for *i=1;i* $\leq$ *max_iterations;i++* **do**
  S_i = sample_self_play($\hat{\pi}_{i-1}$)
  $D_i = \{(s, \text{imit\_learning\_target}(\pi^*_{i-1}(s)))|s \in S_i\}$
  $\hat{\pi}_i$ = train_policy(D_i)
  π^*_i = build_expert($\hat{\pi}_i$)

$n(s,a)/n(s)$ where s is the position being evaluating (so $n(s) = 10,000$ in their experiments). This gives the loss:

$$L_{TPT} = -\sum_a \frac{n(s,a)}{n(s)} \log[\pi(a|s)] \tag{2.17}$$

Compared to CAT, TPT is cost-sensitive: when MCTS is less certain between two moves (because they are of similar strength), TPT penalizes misclassifications less severely. Cost-sensitivity is a desirable property for an imitation learning target, as it induces the IL agent to trade off accuracy on less important decisions for greater accuracy on critical decisions.

Their experiment results supported their conjecture. The classification accuracy is similar between CAT and TPT, but in terms of the gameplay strength, the model trained with TPT loss is superior.

2.6 Game Tree Search in RTS Games

Real-Time Strategy (RTS) Games is a sub-genre of strategy games where players aim to defeat their opponents (destroying their army and base) by strategically building an economy (gathering resources and building a base), military power (training units and researching technologies), and controlling those units. The main differences between RTS games and traditional board games are: they are simultaneous move games (more than one player can issue actions at the same time), they have durative actions (actions are not instantaneous), they are real-time (each player has a very small amount of time to decide the next move), they are partially observable (players can only see the part of the map that has been explored, although in this work we assume full observability) and

they might be non-deterministic.

Furthermore, compared to traditional board games, RTS games have a very large state space and action space at each decision cycle. For example, the branching factor in StarCraft can reach numbers between 30^{50} and 30^{200} [8]. Therefore, RTS games provide a challenging testbed for bandit algorithms and MCTS algorithms.

Despite the success of MCTS in many domains, most of the successful variants of MCTS, e.g., UCT [22], do not scale up well to RTS games due to the combinatorial growth of branching factor with respect to the number of units. Sampling techniques for combinatorial branching factors such as Naïve Sampling [4] or LSI [5] were proposed to improve the exploration of MCTS exploiting combinatorial multi-armed bandits. Inspired by AlphaGo, another approach to address this problem is the use of Naïve Bayes models to learn an action probability distribution as a tree policy prior to guide the search [47]. Other work to deal with this problem involves limiting the search space by introducing action abstractions. For example, instead of searching directly in the raw unit action space, Portfolio greedy search [48] and Stratified Alpha-Beta Search [49] search in the abstracted action spaces generated by hard-coded scripts.

In the next few sections, we are going to survey a collection of MCTS based algorithms for RTS games. The first two, Puppet Search and Script-Based UCT, take advantage of the scripted bots to prune the search space and make the search space feasible to apply MCTS. A scripted bot, called a script, is a rule based game-playing agent which implements the domain knowledge of human players.

2.6.1 Puppet Search

Barriga et al. [50;51] proposed Puppet Search, which combined scripts with look-ahead search. The basis of the algorithm is a hand-authored non-deterministic script, which completely defines the behavior the player should follow, except for some "choice points". Thus, the search space is defined by the set of choice points exposed by the script. Thus, as the authors described, Puppet Search works like a puppeteer that controls the limbs (choice points) of a puppet (the script).

We can control the branching factor by controlling the number of choice points exposed by the

script since the branching factor now grows with respect to the number of choice points instead of the number of units. Puppet search is thus related to the early idea of adaptive Lisp (ALisp)[52], where a script with choice points was defined and reinforcement learning was used to learn a policy for those choice points. Interestingly, ALisp was also applied to the domain of RTS games (such as learning harvesting paths for *Wargus* playing agents).

2.6.2 Script-Based UCT

Justesen et al.[53] proposed a script-based extension to UCT that searches for sequences of scripts instead of actions. Instead of searching in the full action space, script-based UCT only considers the actions proposed by the scripts. The advantage of searching in the space of actions proposed by the scripts is that the branching factor is vastly decreased. Specifically, the branching factor now grows as a function of the number of scripts rather than the number of possible raw actions units can perform. For example, if we are controlling, say 10 units, and each unit can perform 8 different moves, but we only have 4 scripts. The branching factor goes from 8^{10} for vanilla UCT to at most 4^{10} for script-based UCT. Moreover, in practice, it is likely to be lower since if more than one script proposes the same action, then the search space is even smaller.

Due to the durative action in RTS games, a variation of UCT algorithm called UCT Considering Durations (UCTCD) was used[53].

2.6.3 Naïve Monte Carlo Tree Search

NaïveMCTS[7] is a variant of MCTS specifically designed to handle RTS games. NaïveMCTS can handle durative and simultaneous actions, but most importantly, the key feature of NaïveMCTS is that rather than using standard ϵ-greedy or UCB1, it uses a sampling policy based on Combinatorial Multiarmed Bandits (CMABs) called *Naïve Sampling* in order to scale up to the combinatorial branching factors of RTS games.

MCTS algorithms employ two policies called the *tree* policy and the *default* or *playout* policy. The former is used to determine which node of the tree to explore next, and the latter is used to perform stochastic rollouts from the leaf of the tree until a terminal state is reached.

Naïve Sampling is a sampling strategy for CMAB problems designed to act as the tree policy of MCTS. Thus, it tries to find the *macro-action* (the combination of all the actions issued to game units in a given game state by the player) that maximizes the expected reward. As usual, Naïve Sampling decomposes this problem into *exploration* and *exploitation*. During exploration, a *naïve assumption* that the reward of the macro-action can be decomposed as the sum of the expected rewards of the individual unit actions is used. With this assumption, we can decompose the CMAB problem into n child MAB problem, denoted as $MAB_1, \ldots, MAB_n$. During *exploitation*, such *naïve assumption* is not assumed, and a global MAB, denoted as MAB_g (which has a macro-arm for each macro-action generated so far during exploration), is used to find the best macro-action amongst all the currently explored ones, ensuring convergence to the optimal macro-action regardless of whether the domain at hand satisfies or not the naïve assumption.

Specifically, Naïve Sampling works as follows. At each iteration, a stochastic decision is made to exploit (with probability $1 - \epsilon_0$) or explore (with probability ϵ_0):

- if exploit is selected: an ϵ-greedy policy π_g is used to select a macro-action using the global MAB.
- if explore is selected: for each local MAB, an ϵ-greedy policy π_l is used independently for each game unit to select actions and the resulting macro-action is added to the global MAB.

2.6.4 Action Abstractions for Monte Carlo Tree Search

Searching only in the space proposed by scripts as proposed in the algorithm above can help largely reduce the search space as discovered in Script-Based UCT and Puppet Search. Moraes and Lelis proposed Asymmetric Action Abstraction[49] and combined it with NaïveMCTS in their recent work[54].

The main idea is to reduce the number of legal moves by using the scripts, but only a subset of the units; legal moves of the other units remain unchanged. The authors proposed three variations of the algorithm: A1N, A2N, and A3N.

- A1N is the baseline algorithm and it searches only in the unit actions proposed by the set of scripts. Thus, it is basically the same as script-based UCT, but using NaïveMCTS instead of

UCTCD.

- A2N allows NaïveMCTS to search in an asymmetrically-abstracted action space defined by two sets of scripts, one for economy units and one for combat units.

- Finally, A3N, further relaxes the pruning of the search space, and some units are marked as "unrestricted", and the full action space of those is considered during search. A heuristic function is used to determine which units are unrestricted such as considering the unit closest to the enemy units to be unrestricted, and all the rest to be restricted by scripts.

Chapter 3: Incorporating Domain Knowledge in MCTS

In games with large branching factors and limiting computation budget, like RTS games, the search algorithms can only search a small part of the search space. Thus, it is crucial to guide the search algorithm towards a more promising area of the search space. In this chapter, we focus on introducing various of forms of domain knowledge into MCTS in RTS games.

In the context of this work, we consider *Domain knowledge* to be the knowledge obtained from human game-playing experience, for example, explicit rules, heuristics, scripts or even replays of human games. There have been many previous approaches that incorporate domain knowledge into game AI algorithms. For example, *Minimax* search and variants such as *alpha-beta search* achieved great success in computer Chess back in 1996 using a variety of heuristic functions to reduce the effective branching factor. *AlphaGo* also achieved super human game-play strength in computer Go by learning from human game replays.

There are many ways we can obtain such knowledge. For example, an *evaluation function* is a direct way of encoding human player's ability to separate good and bad game states. Also, domain knowledge can be obtained by analysing game replays of human players. And there are many more forms of encoding of human knowledge. Generally, we can classify them into two categories: *value-based* and *policy-based*.

The evaluation functions are value-based methods that take a game state and return a score that reflects the advantage (or disadvantage) of a player. An example of policy-based encoding domain knowledge is to construct a mapping from a state to actions or a probability distribution of actions. Such mapping is called a policy. For example, AlphaGo learns a policy from the replays of expert human player and use the policy network to guide search algorithm.

In this chapter, we focus on different techniques to integrate domain knowledge into Monte Carlo Tree Search in RTS games. Specifically, we will discuss two approaches to do so: learning policy and learning evaluation function from replays of human-authored scripts. In the first approach,

we study the problem of learning tree policy from the replays of round-robin tournaments of a collection of human-authored scripts. We evaluate the performance of the model by comparing prediction accuracy and gameplay strength as the tree policy. In the second approach, instead of learning the policy directly, the model learns an evaluation function by predicting the winrate of the game states. We also compare the prediction accuracy and gameplay strength as the evaluation function of the playout policy.

3.1 Learning Policies from Replays

In this section, we build upon previous work on Informed MCTS, where Bayesian models were trained from data to inform the tree policy[47]. Given that RTS games have a much more limited decision budget between each two game decisions compared to turn-based games, like most board games, the model we build must make fast predictions, apart from the requirement of accuracy.

Some times replays come from scripts with fast decision making, that can directly be used in MCTS, as explained above. However, in general, we might encounter domains where we have replays available, but no fast scripts. For example, replays might come from humans, or from search-based bots, which a high compute cost per move. Hence, by learning a policy from these replays, we can make use of the domain knowledge contained in these replays, regardless of whether they come from a fast script or not.

In consideration of the time constraint, we evaluate and compare two families of models: Bayesian classifiers and decision trees classifiers. Our results show that in experiments under the same iteration budget for MCTS, the models with higher classification performance also have better gameplay strength when used within MCTS. However, when we constrain computation budget by time, faster models tend to outperform slower, more accurate models. Specifically, the C4.5 model stands out in our experiments as the best model since it has good classification performance and fast classification speed.

3.1.1 Methods for Tree Policy Learning

As we have discussed in the Background, The *tree policy* serves as the criteria of child node selection within the search tree and balances exploration and exploitation. It can also be informed by a given prior distribution and then bias the search towards the desired action space. For example, the PUCB algorithm (Predictor+UCB)[55] extends the standard UCB1 strategy commonly used as the tree policy with an existing distribution. A variation of PUCB was used in AlphaGo, where Silver et al.[56] used temporal difference method to learn a value function and to inform the tree policy. This prior distribution can be trained offline, e.g. from existing game replays. AlphaGo, for example, trained a neural network using the human expert plays as supervision, and then refined it using reinforcement learning. Unlike the game of Go, however, the RTS games are real-time, which means the computational budget per action is much smaller than in Go. Therefore, large neural network models become impractical since they are currently too slow for the available computational budget. Thus, in this chapter, we select a collection of statistical learning algorithms which are potentially fast at classification time as our subjects of study.

We use supervised learning to model and predict the move probability of script bots and bias the tree search to mimic and hopefully improve upon script bots with the help of tree search. A supervised learning model $p_u(a_i|s)$ takes a feature vector s as the representation of the game state from the point of view of a unit u and output the probability distribution over all n possible actions $(a_1, a_2, a_3, \cdots, a_n)$ the unit can perform. The model is trained beforehand and prediction is made each time the tree policy needs to be used. Thus our learning algorithm for modeling the problem must be fast at classification time. Also, in order to increase the sampling quality, the classifier should also provide good class probability estimations. Therefore, we selected a relatively small set of categorical features (eight features) to represent the game states. Moreover, two types of classification algorithms what are previously known to be suitable for categorical data and fast at classification time are studied in this work: (semi) Naïve Bayes and (ensembles of) decision trees.

Naïve Bayes and Semi Naïve Bayes

The Naïve Bayes classifier is a simple probabilistic classifier that makes a strong assumption that all the features are independent. This allows us to write the joint density as the product of all the component densities:

$$p(\mathbf{x}|y = c, \theta) = \prod_{i=1}^{D} p(x_i|y = c, \theta_{ic}) \tag{3.1}$$

Naïve Bayes classifiers are highly scalable and can be trained and tested both in linear time. However, the assumption of independent features can be violated in most of situations. Thus, the research community has developed many semi-Naïve Bayes algorithms that have a weakened independence assumption.

The semi-Naïve Bayes algorithms tested in our work is called averaged one-dependence estimators (A1DE)[57]. Sahami introduces n-dependence estimators, where the probability of each feature is conditioned by the class and n other features.[58] The averaged one-dependence estimators work like this: for each feature $\mathbf{x}$, a classifier that assumes the other features are independent given the label and feature $\mathbf{x}$. The model makes predictions by averaging the outcome of all classifiers. Therefore, A1DE can also be viewed as an ensemble method for Naïve Bayes classifiers. We also test a more relaxed variation of A1DE called A2DE, which builds estimators conditioned by each pair of features.

Decision Trees and Ensembles

The C4.5 classifier[59] iteratively builds a decision tree by splitting the instances by the feature with the most information gain (reduction of entropy) at each internal node. Then it assigns class predictions at leaf nodes. Then the algorithm prunes the tree by a threshold confidence interval of 25%. We can estimate the class probability naturally from the label frequency of the leaves. Moreover, this estimation can be skewed towards 0 or 1 since the leaves are mostly dominated by one class. In our experiments, Laplace smoothing[60] is applied in order to produce a more accurate class probability estimation.

Bootstrapped aggregating (bagging)[61] is an ensemble method that helps to improve accuracy and stability by combining results from multiple training sets resampled with replacement. In our

experiments, we apply 10 iterations of bagging to C4.5.

Random forest [62] is a meta classifier that fits a collection of random tree classifiers on resamples of the dataset and thus improves the predictive accuracy and control overfitting by averaging. The internal random trees select a subset of features to build the model. In our case, each random tree selects a subset of $\log(n) + 1$ where n is the number of features. In our experiments, 100 random trees are built by the random forest algorithm. The class probability estimation is generated from the proportion of votes of the trees in the ensemble.

In practice, we observed that learning a different model for each different unit type in the game (workers, bases, barracks, etc. in μRTS) resulted in a better estimation of the probabilities. So, for the experiments reported in the remainder of this work, we generated the probability models in the following way: (1) For each unit type, we train a model using the subset of the training data corresponding to the unit type. (2) If this subset is empty, then just train with the whole training set (i.e., if we have no training data to model the way a specific unit is controlled, we just train a model with the whole training set for such unit, hoping it will reflect what the script we are collecting data from would have done).

3.1.2 Experiments and Results

Bots for Data Collection

We generated training data in the following way. We collected data from a round-robin tournament consisting of six bots in 8 different maps, four of which are 8×8 maps and the other four are 12×12. Four of the bots are built-in hard-coded bots: WorkerRush, LightRush, HeavyRush, and RangedRush. The other two were Monte-Carlo search bots: LSI and NaïveMCTS, whose computational budgets are configurable. We ran the experiments for LSI and NaïveMCTS bots for four times with computation budgets of 500, 1000, 2000, and 5000 playouts per game frame respectively. The description of each bot is as below:

- *Rush bots:* hardcoded deterministic bots that implement a rush strategy that constantly produces one type of unit and sends them to attack the opponent's base. Specifically, we used the *WorkerRush*, *LightRush*, *RangedRush* and *HeavyRush* bots of μRTS, which implement rushes

with worker, light, ranged, and heavy units, respectively.

- *LSI:* Monte Carlo search with Linear Side Information[5]. LSI is a two-phase scheme for CMAB planning, where the first phase generates candidate combinatorial actions, and the second phase evaluates these candidates.

- *NaïveMCTS:* MCTS algorithm with a tree policy specifically designed for games with combinatorial branching factors. The tree policy exploits Combinatorial Multi-Armed Bandits[7] to handle the combinatorial explosion of possible moves.

The feature vector $\mathbf{x}(u, s)$ used to represent each game state contains only eight features: the number of resources available to the player, the cardinal direction (north, east, south, west) toward where most friendly units are, the cardinal direction toward where most enemy units are, whether we have a barracks or not, and four features indicating the type of the unit in the cell two positions north, east, south or west (or whether these cells are empty or are a wall). Adding more features could certainly improve performance, which will be part of our future work. We employed these 8 features, as used in previous work[47], for comparison purposes.

The 12 datasets collected are described below:

- I_{WR}: consisting of all the unit-actions of WorkerRush (32679 instances)

- I_{LR}: consisting of all the unit-actions of LightRush (30139 instances)

- I_{HR}: consisting of all the unit-actions of HeavyRush (24385 instances)

- I_{RR}: consisting of all the unit-actions of RangedRush (31279 instances)

- I_{LSI}^{500},I_{LSI}^{1000},I_{LSI}^{2000},I_{LSI}^{5000}: consisting of all the unit-actions of LSI under computing budget of 500, 1000, 2000, and 5000 respectively (85918, 88816, 85611, and 68236 instances respectively)

- I_{NMCTS}^{500},I_{NMCTS}^{1000},I_{NMCTS}^{2000},I_{NMCTS}^{5000}: consisting of all the unit-actions of NaïveMCTS under computing budget of 500, 1000, 2000, and 5000 respectively (73249, 83925, 78896, and 68158 instances respectively)

Table 3.1: Model Comparison on classification accuracy and expected log-likelihood in the 12 Datasets

	Naïve Bayes		A1DE		A2DE		J48		Bagging		Random Forest	
	Accuracy	Exp. l.l.	Accuracy	Exp. l.l.	Accuracy	Exp.l.l	Accuracy	Exp. l.l.	Accuracy	Exp. l.l.	Accuracy	Exp. l.l.
WR	0.6119	-1.0749	0.6609	-0.8921	0.6698	**-0.8527**	0.6821	-0.8574	0.6828	-0.9074	**0.6882**	-0.9810
LR	0.7955	-0.6048	0.8210	-0.5059	**0.8276**	**-0.4876**	0.8188	-0.4972	0.8248	-0.5598	0.8263	-0.6504
HR	0.8490	-0.4765	0.8671	-0.3950	**0.8732**	**-0.3825**	0.8666	-0.3995	0.8703	-0.4379	0.8716	-0.4855
RR	0.8411	-0.5223	0.8585	-0.4110	0.8602	**-0.3954**	0.8593	-0.4070	0.8606	-0.4389	**0.8648**	-0.4450
LSI500	0.3637	-1.4711	0.3713	-1.4048	**0.3715**	-1.3902	0.3637	**-1.3488**	0.3618	-1.5125	0.3595	-1.9905
LSI1000	0.3743	-1.4626	**0.3795**	-1.3904	0.3776	-1.3721	0.3628	**-1.3392**	0.3619	-1.5155	0.3606	-1.9683
LSI2000	0.3832	-1.4621	**0.3920**	-1.3877	0.3915	-1.3708	0.3749	**-1.3314**	0.3764	-1.5202	0.3741	-1.9860
LSI5000	0.3977	-1.4382	**0.4062**	-1.3578	0.4051	-1.3400	0.3866	**-1.3119**	0.3903	-1.4692	0.3878	-1.9382
NaïveMCTS500	0.3747	-1.4711	**0.3832**	-1.4017	0.3811	-1.3861	0.3677	**-1.3385**	0.3672	-1.5314	0.3635	-1.9821
NaïveMCTS1000	0.3839	-1.4544	**0.3926**	-1.3809	0.3893	-1.3641	0.3740	**-1.3312**	0.3747	-1.4991	0.3709	-1.9431
NaïveMCTS2000	0.3916	-1.4527	**0.3995**	-1.3831	0.3985	-1.3665	0.3798	**-1.3296**	0.3827	-1.5034	0.3802	-1.9374
NaïveMCTS5000	0.4005	-1.4422	0.4123	-1.3652	**0.4126**	-1.3451	0.3969	**-1.3100**	0.3974	-1.4677	0.3963	-1.9043

Table 3.2: Comparison of model classification speed, where k is the number of classes, n is the number of features, and m is the number of iterations and trees built for bagging and random forest, respectively.

Model	Instances/ms.	Complexity
Naïve Bayes	303.58	$\Theta(kn)$
A1DE	174.46	$\Theta(kn^2)$
A2DE	43.07	$\Theta(kn^3)$
J48	298.05	$\Theta(log(n))$
Bagging+J48	64.37	$\Theta(m \cdot log(n))$
Random Forest	30.97	$\Theta(m \cdot log(n))$

Empirical Comparison of Models

For all six models, we used the Weka implementation[1]. In this section we compare the six machine learning models described before from the following aspects:

- action prediction (accuracy and probability estimation)
- classification speed
- gameplay strength as tree policy under iteration budget
- gameplay strength as tree policy under time budget

Unit Action Classification First, we evaluate the models on the classification and class probability estimation performance. Table 3.1 shows the predictive accuracy and expected log-likelihood of the six models in each of the 12 datasets using a 10-fold cross validation. There is a total of 69 different actions a unit can perform in μRTS, but each individual can perform only between 5

[1]The open sourced implementation of C4.5 in Weka is J48.

Table 3.3: Win rates against the baselines of the models trained by I_{WR}

Tree Policy	NB	A1DE	A2DE	J48	Bagging	RF
Random	1.000	1.000	1.000	1.000	1.000	1.000
RndBiased	0.987	1.000	1.000	0.993	0.993	0.993
WorkerR	0.668	0.650	0.662	0.637	0.675	0.668
LightR	0.843	0.875	0.806	0.900	0.868	0.850
HeavyR	1.000	1.000	1.000	1.000	1.000	1.000
RangedR	1.000	1.000	1.000	1.000	1.000	1.000
LSI	0.656	0.662	0.637	0.612	0.662	0.706
NMCTS	0.475	0.531	0.506	0.637	0.587	0.687
Average	0.829	0.840	0.827	0.848	0.848	**0.863**

Table 3.4: Win rates against the baselines of the models trained by I_{LR}

Tree Policy	NB	A1DE	A2DE	J48	Bagging	RF
Random	1.000	1.000	1.000	1.000	1.000	1.000
RndBiased	0.993	1.000	1.000	1.000	0.987	0.993
WorkerR	0.662	0.581	0.625	0.6	0.687	0.681
LightR	0.862	0.862	0.787	0.781	0.862	0.831
HeavyR	1.000	1.000	1.000	1.000	1.000	1.000
RangedR	1.000	0.993	1.000	1.000	1.000	1.000
LSI	0.618	0.656	0.618	0.650	0.650	0.687
NMCTS	0.456	0.525	0.500	0.643	0.593	0.606
Average	0.824	0.827	0.816	0.834	0.848	**0.850**

Table 3.5: Win rates against the baselines of the models trained by I_{HR}

Tree Policy	NB	A1DE	A2DE	J48	Bagging	RF
Random	1.000	1.000	1.000	1.000	1.000	1.000
RndBiased	1.000	0.994	1.000	1.000	1.000	1.000
WorkerR	0.631	0.625	0.637	0.713	0.688	0.744
LightR	0.844	0.869	0.856	0.881	0.869	0.919
HeavyR	1.000	1.000	1.000	1.000	1.000	1.000
RangedR	1.000	0.993	1.000	1.000	1.000	1.000
LSI	0.544	0.569	0.650	0.738	0.725	0.637
NMCTS	0.512	0.487	0.550	0.575	0.606	0.675
Average	0.816	0.818	0.837	0.863	0.861	**0.872**

and 33 unit-actions. We can see that the models predict the behavior of the four hard-coded bots better than the Monte Carlo search bots. But as the computing budget increases, the prediction performance also improves for the Monte Carlo search bots. This indicates that the Monte Carlo search bots converge to more stable strategies as the computing budget increases.

We also report the expected log-likelihood of the actions in the dataset given the model. This is a better metric to consider than classification accuracy, given that we want to estimate the probability

Table 3.6: Win rates against the baselines of the models trained by I_{RR}

Tree Policy	NB	A1DE	A2DE	J48	Bagging	RF
Random	1.000	1.000	1.000	1.000	1.000	1.000
RndBiased	0.994	1.000	0.994	0.994	1.000	0.994
WorkerR	0.675	0.600	0.637	0.681	0.694	0.713
LightR	0.800	0.825	0.875	0.812	0.906	0.838
HeavyR	1.000	1.000	1.000	1.000	1.000	1.000
RangedR	1.000	1.000	1.000	1.000	1.000	1.000
LSI	0.588	0.637	0.725	0.700	0.706	0.738
NMCTS	0.619	0.494	0.644	0.588	0.619	0.600
Average	0.834	0.820	0.859	0.847	**0.866**	0.860

Table 3.7: Win rates against the baselines of the models trained by I_{LSI5000}

Tree Policy	NB	A1DE	A2DE	J48	Bagging	RF
Rndom	1.000	1.000	1.000	1.000	1.000	1.000
RndBiased	1.000	1.000	1.000	1.000	0.994	1.000
WorkerR	0.650	0.675	0.700	0.706	0.675	0.725
LightR	0.856	0.944	0.894	0.931	0.894	0.844
HeavyR	1.000	1.000	1.000	1.000	1.000	1.000
RangedR	1.000	1.000	1.000	1.000	1.000	1.000
LSI	0.850	0.825	0.688	0.688	0.688	0.744
NMCTS	0.675	0.688	0.662	0.562	0.725	0.625
Average	0.879	**0.891**	0.868	0.861	0.872	0.867

Table 3.8: Win rates against the baselines of the models trained by $I_{\text{NMCTS5000}}$

Tree Policy	NB	A1DE	A2DE	J48	Bagging	RF
Random	1.000	1.000	1.000	1.000	1.000	1.000
RndBiased	1.000	1.000	1.000	0.994	0.994	0.994
WorkerR	0.731	0.700	0.738	0.600	0.688	0.688
LightR	0.875	0.831	0.881	0.887	0.887	0.875
HeavyR	1.000	1.000	1.000	1.000	1.000	1.000
RangedR	1.000	1.000	1.000	1.000	1.000	1.000
LSI	0.800	0.762	0.769	0.700	0.775	0.794
NMCTS	0.650	0.725	0.756	0.688	0.688	0.669
Average	0.882	0.877	**0.893**	0.859	0.879	0.877

distribution of the actions and not just predict the most likely action. The best possible log-likelihood would be 0. We can observe that for Bayesian models, the more we relax the independence assumption, the better the accuracy and the log-likelihood. However, for decision trees, bagging and random forest outperformed J48 in accuracy but not in the log-likelihood.

Table 3.2 report the speed in classification time, where k is the number of classes, n is the number of features, and m is the number of iterations and trees built for bagging and random forest,

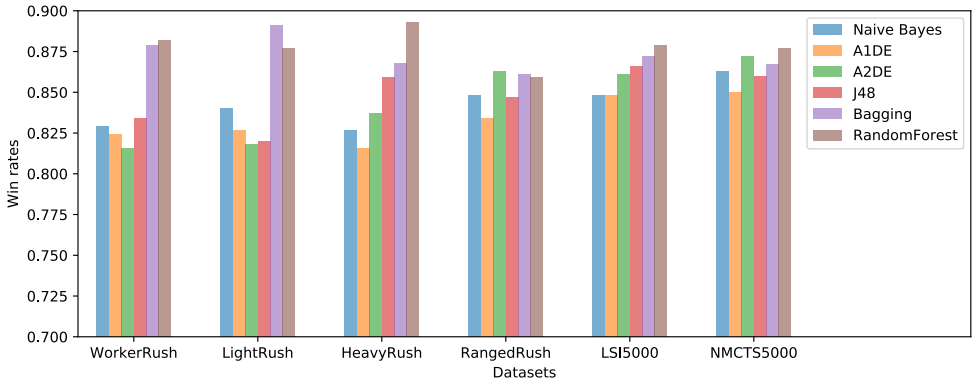


Figure 3.1: Win Rates by Dataset and Model under iteration budget.

respectively. The fastest models are Naïve Bayes and J48. The classification speed is very important in time-constrained MCTS experiments, as we will show later.

Win rates under Iteration Budget We now evaluate our models as the prior distribution for the tree policy. We use the trained models in the tree policy as the sampling prior for the Naïve Sampling process and we used RndBiased bot as the default policy. In this experiment, the NaïveMCTS bots coupled with models trained from different datasets are tested against the eight baseline bots that were used for training trace generation. The individual results for each dataset are reported in Tables 3.3-3.8. Figure 3.1 shows an aggregated bar chart view of the average win rates grouped by training set and model. The overall average performance of every model is reported in Table 3.9.

The best model overall is the random forest model with a 0.865 win rate against the baselines and being the best performing model in three out of six experiments trained separately for each dataset. A general observation is that the decision tree models, especially the tree ensembles, have better gameplay performance than Bayesian models when the training dataset is one of the rush bots. Meanwhile the Bayesian models outperformed the decision tree models in the more complex datasets, I^{5000}_{LSI} and I^{5000}_{NMCTS}. This seems to be because decision trees are better at predicting the deterministic behavior of the rush bots but struggle to estimate the probability distribution of moves of the more stochastic search bots.

Table 3.9: Overall Win rates per Model under Iteration Budget

Tree Policy	NB	A1DE	A2DE	J48	Bagging	RF
Overall	0.844	0.846	0.850	0.852	0.862	**0.865**

Table 3.10: Overall Win rates per Model under Time Budget

Tree Policy	NB	A1DE	A2DE	J48	Bagging	RF
Random	1.000	1.000	1.000	1.000	1.000	1.000
RndBiased	0.992	0.987	0.982	0.991	0.991	0.980
WorkerR	0.522	0.514	0.512	0.551	0.514	0.495
LightR	0.634	0.648	0.608	0.703	0.665	0.624
HeavyR	0.991	0.993	0.995	0.997	0.996	0.987
RangedR	0.993	0.997	0.992	0.996	0.992	0.988
NMCTS	0.708	0.688	0.634	0.736	0.594	0.579
Average	0.834	0.833	0.818	**0.853**	0.822	0.808

Win rates under Time Budget Due to the nature of RTS games, the computational budget is normally constrained by time. In that sense, complex models that are working well under iteration budget might lose their edges because they run fewer iterations than simpler but faster models. Thus, we run the gameplay strength experiments against seven[2] of the baseline bots again with the computational budget being 50 milliseconds per cycle. The results are reported in Table 3.10.

We can observe that the model performance is significantly affected by the time budget. Although the more sophisticated models like A1DE, bagging, and random forest have a better performance in experiments with iteration budget, they do not have the same performance in experiments with time budget. The reason is that those models are significantly slower than Naïve Bayes and C4.5, according to Table 2. The C4.5 model stands out as the best performing model (0.853 overall win rate) under experiments with time constraints due to its speed and better class probability estimation performance compared to Naïve Bayes model.

3.2 Learning Transferable Features using Convolutional Neural Networks

In this section, we focus on the problem of learning evaluation functions for arbitrary RTS game maps by learning general and map-independent features applicable to complex situations. This is important since scenario variability is not just a key feature of most RTS games but of many

[2]The current version of LSI does not support time budgets.

real-world problems as well. Specifically, we propose a convolutional neural network architecture specifically designed in consideration of learning features that are map size-independent and generalizable to larger maps. We report experiments training the network with data collected on smaller maps, which is cheaper to collect than data for larger maps, and then evaluate the performance of the trained network on a collection of larger maps. By utilizing generalizable features learned from small maps, we can improve the time and data efficiency of our learning approach for building agents for complex maps. All the reported experiments are carried out on the μRTS simulator[4].

Many deep neural network architectures trained for computer vision tasks learn features similar to Gabor filters and color blobs[63] in the first layer. These features tend to be generalizable to different domains and tasks. Based on this observation, in this work, we design and train our μRTS state evaluation networks for small game maps and show that they generalize to larger maps.

The remainder of this section first presents the procedure we followed to collect training data for our models, how are the game states encoded to be provided as input to the neural networks, and finally, we describe the proposed network architecture and training procedure.

3.2.1 Data Collection

We collected replay data from a round-robin tournament with 10 μRTS bots. Specifically, we used the following bots:

- *Rush bots:* hardcoded deterministic bots that implement a rush strategy that constantly produces one type of unit and sends them to attack the opponent's base. Specifically, we used the *WorkerRush*, *LightRush*, *RangedRush* and *HeavyRush* bots of μRTS, which implement rushes with worker, light, ranged, and heavy units respectively.

- *PortfolioAI:* playouts using a set of simple bots (WorkerRush, LightRush, RangedRush, and Random), are run all possible pairings, and the minimax bot is selected to produce the next action.

- *MonteCarlo:* standard Monte Carlo search approach, splitting the computation budget uniformly among all possible actions.

- *ϵ-Greedy MonteCarlo:* same as Monte Carlo search but using ϵ-greedy strategy to allocate computation budget to each action.

- *DownsamplingUCT:* Standard UCT[32], but in those nodes of the tree where the branching factor is larger than a constant k ($k = 100$ in our experiments), k of those actions are sampled at random, and only those are considered during search.

- *UCTUnitActions:* Standard UCT, but in nodes where we can issue actions to more than one unit, rather than considering the whole combinatorics, we only consider the actions of one unit per node (the children nodes of this node will consider actions of the rest of units, in turn). This follows the UCT implementation of Balla and Fern in their work on the *Wargus* RTS game[64].

- *NaïveMCTS:* MCTS algorithm with a tree policy specifically designed for games with combinatorial branching factors. The tree policy exploits Combinatorial Multi-Armed Bandits[7] to handle the combinatorial explosion of possible moves.

Four different configurations of NaïveMCTS were used, for a total of 13 different bots. For bots with configurable computing budget (PortfolioAI, MonteCarlo, DownsamplingUCT, UCTUnitActions and NaïveMCTS), we run tournaments under eight different budgets per game frame: $100ms$, $200ms$, $400ms$, $800ms$, 100 playouts, 200 playouts, 400 playouts and 800 playouts (i.e., giving the bots a certain amount of milliseconds per game frame, or a certain number of playouts, which makes sense in the MCTS-based bots).

Maps of size 8×8 with 23 different starting configurations were used. We decided to collect data only on small 8×8 maps, since it's cheaper to generate (games in larger maps take longer to execute). Additionally, in this way, we can also test if the learned evaluation function generalizes to larger maps, when trained only in smaller maps. After we discarded all replays that resulted in a draw and duplicated replays from games between two deterministic bots (to avoid biased data), this resulted in 25,200 replays.

Since states coming from the same replay are highly correlated (a state at time t and one at

time $t+1$ from the same replay are likely to be very similar), if we were to use all the states from all replays, the i.i.d. assumption of supervised learning algorithms would be violated. To avoid this problem, we randomly sample three positions in each replay. We also augmented each position in the training set with all reflections and rotations. As a result, we ultimately have a training set with 484,800 game states and a testing set with more than 15,000 game states. Each game state is labeled with the player that won the corresponding game.

3.2.2 Features

For this work, we assumed each map can have an arbitrary size $w \times h$ where w and h is the width and height of the map. Each game state is thus converted into a $15 \times w \times h$ tensor, composed of 15 feature planes (each of them of size $w \times h$), to be fed as input to the neural network. The feature planes that we use (see Table 3.11) come directly from the raw representation of the board information (unit types, unit health, unit owners, unit positions, and resources carried by workers). Many of the features are split into multiple planes of binary values. For example, in the case of unit positions, there are six feature planes for each type of unit indicating position information. In addition to the 15 feature planes, we have a *global feature array*, consisting of the following features:

- Resources owned by each player
- Proportion of game time left scaled from 0-20

For example a *global feature array* $[5, 6, 4]$ can be interpreted as that the max player has 5 resources, the min player has 6 resources, and the game has about 20% of its maximum pre-defined length (5000 cycles in our experiments) left.

In comparison to the input of Stanescu's work[65], we employ a similar one-hot encoding for spatial information. The difference between our work and theirs is that we separated the global information from the spatial information and they encoded them all together. In the next section, we will describe the design of our model and usage of the inputs.

Table 3.11: Description of feature planes for game state representation

Features	Planes	Description
Unit Type & Position	7	Base, Barracks, Resource, worker, light, ranged, heavy
Unit Health	5	1, 2-3, 4-5,6-7, ≥ 8
Unit Owner	2	Mask each unit to its player owner
Worker Resources	1	1 if the worker is carrying a resource, 0 otherwise

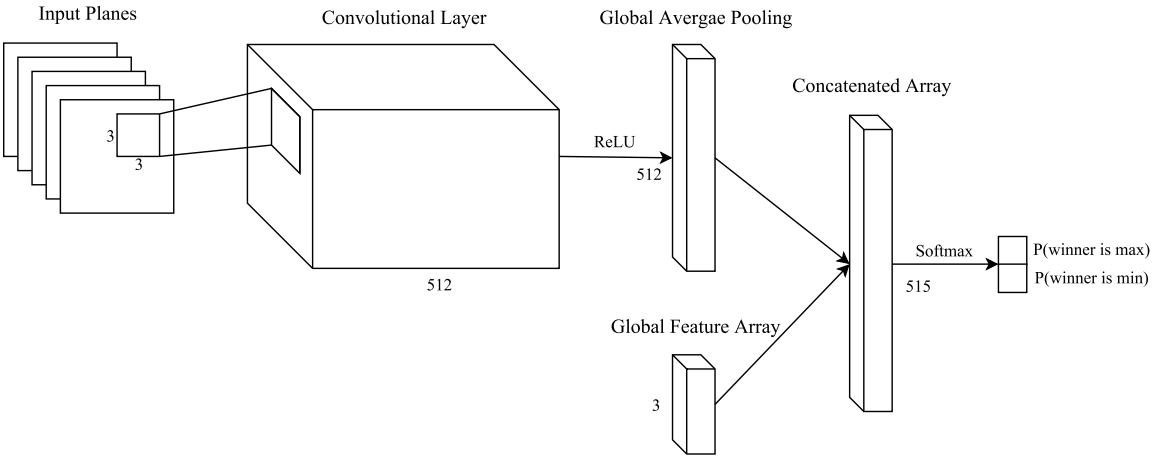


Figure 3.2: Proposed neural network architecture for maps-size independent game state evaluation.

3.2.3 Neural Networks Architecture and Training

Since our goal is to train a neural network that can handle different map sizes as input and generalize across them, we made the following design decisions:

1. Structurally: we employed a convolutional neural network (CNN) approach[66]. Usually, CNNs consist of a set of *convolutional layers*, followed by a set of fully connected layers. This assumes a fixed input size. We replaced the fully connected layers (typically at the end) with a global average pooling layer (described below), which converts different-sized convolutional feature maps into a fixed size vector. In this way, we can provide maps of different sizes as input.

2. Algorithmically: only using one but thick convolutional layer with 512 filters to focus on learning general features. After this layer, we feed the concatenation of the output of the convolutional layer (after having been converted to a fixed size vector by the pooling layer)

and the global feature array to the output layer (as shown on Figure 3.2).

As described above, the input is a stack of $15 \times w \times h$ spatial features where $w \times h$ is the size of the input map plus a global feature array of size 3. First, each spatial feature plane is padded with 0s to obtain a $(w+1) \times (h+1)$ plane. Then the input is convolved with 512 3×3 filters with stride 1. The resulting feature maps are also padded with 0s to keep the same size as the input. After the convolutional layer, rectified linear units (ReLU) are applied to calculate activations and a 0.5 dropout ratio is applied to prevent overfitting. We then directly output the spatial average of the feature maps from the convolutional layer as the confidence of categories through a global average pooling layer[67] instead of adopting the fully connected layers for classification that are usual in CNNs. This results in a vector of size 512 (one per spatial feature map). This vector is then concatenated with the non-spatial global feature array. Finally, the concatenated vector is fed into the softmax layer of 2 units, which predicts the winning probability for each player. The final architecture of the proposed neural network is shown in Figure 3.2.

We use Xavier algorithm to initialize weights[68], and adaptive moment estimation (ADAM)[69] with hyperparameters $\beta_1 = 0.9$, $\beta_2 = 0.999$, and $\alpha = 0.001$ as the optimizer. We employ cross entropy loss for backpropagation. The whole training process takes about 10 minutes to converge on a NVIDIA GTX 1080 Ti GPU with the dataset of 484,800 training instances.

3.2.4 Experimental Evaluation

In order to evaluate our approach, we conduct two evaluation experiments:

1. *winner prediction*: we compare the winner prediction accuracy on the collected dataset on 8×8 maps using a test set of 15,000 game states) between the neural network and several human-designed baseline evaluation functions.

2. *game play*: we evaluate the game play strength of a Monte Carlo Tree Search bot using our proposed evaluation function versus using the baseline evaluation functions on three sets of maps of different sizes: 10×10, 12×12, and 16×16.

Experimental Setup

In both experiments, we use two simple human-designed evaluation functions and the Lanchester[70] model as the baselines, all of which evaluate the game states by assigning a numercal score to each player and comparing these values. These evaluation functions work as follows:

- *Simple evaluation function*[3]: The scoring system is shown in equation (1). Given a state s and a player p , U_p is the set of units owned by p in state s, R_p is the total amount of resources for player p, $W_p \subseteq U_p$ is the set of player p's workers is s, R_u is the amount of resources each worker unit u is carrying, C_u is the cost of unit u, HP_u the current health points of unit u, and $MaxHP_u$ its maximum health points of u. W_{res}, W_{work}, W_{unit} are constant weights from human experience, which are 20, 10, and 40 respectively (we omit the state s in the equations for simplicity).

$$E_p^S = W_{res}R_p + W_{work}\sum_{u\in W_p} R_u + W_{unit}\sum_{u\in U_p}\frac{C_u HP_u}{MaxHP_u} \tag{3.2}$$

  Then the state evaluation for player p ($p \in \{0, 1\}$) is

$$Simple_p = E_p^S - E_{1-p}^S \tag{3.3}$$

- *SimpleSqrt evaluation function*[4]: The scoring system is very similar to *Simple evaluation function* but takes a square root on the term for unit health.

$$E_p^Q = W_{res}R_p + W_{work}\sum_{u\in W_p} R_u + W_{unit}\sum_{u\in p}\sqrt{\frac{C_u HP_u}{MaxHP_u}} \tag{3.4}$$

  Also, the final evaluation is normalizaed between 0 to 1.

$$SimpleSqrt_p = 2E_p^Q/(E_p^Q + E_{1-p}^Q) - 1 \tag{3.5}$$

[3]This corresponds to the *SimpleEvaluationFunction* function of μRTS.
[4]This corresponds to the *SimpleSqrtEvaluationFunction3* function of μRTS.

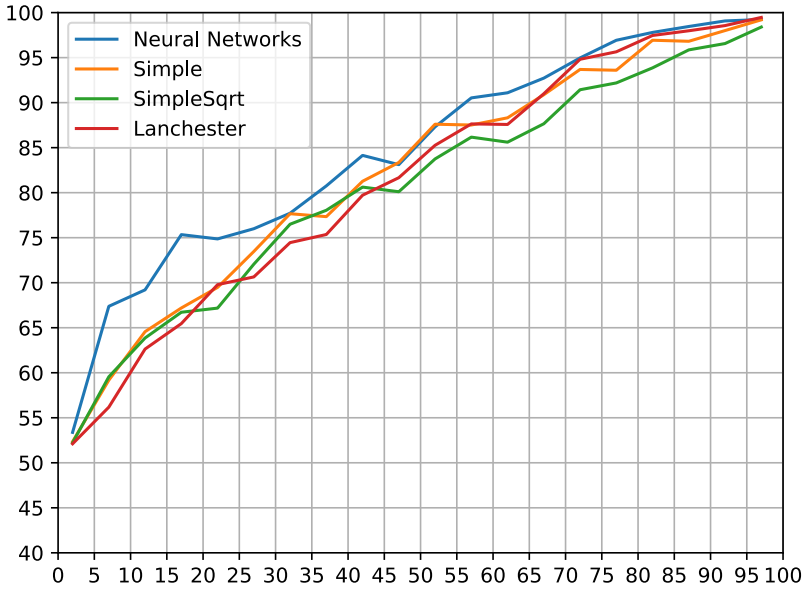


Figure 3.3: Winner prediction accuracy comparison. Vertical axis shows prediction accuracy, and horizontal axis shows game time (as a percentage of total length of a replay).

- *Lanchester evaluation function*[5]: The Lanchester model is a strong heuristic evaluation model in RTS games since it can be optimized using logistic regression according to specific situations. In Equation 3.6, α_u is the strength value for the type of unit u, and o is the Lanchester attrition order. We employ the parameters and weights trained by Stanescu et al.[65] for our dataset which are optimizaed for 8×8 maps. Then again, the state evaluation score for player p is calculated as $Lanchester_p = E^L_p - E^L_{1-p}$.

$$\begin{aligned} E^L_p = \quad & W_{res} R_p + W_{work} \textstyle\sum_{u \in W_p} R_u + \\ & W_{base} \tfrac{HP_{base}}{MaxHP_{base}} + W_{barracks} \tfrac{HP_{barracks}}{MaxHP_{barracks}} + \\ & \textstyle\sum_{u \in U_p} \alpha_u \tfrac{C_u HP_u}{MaxHP_u} |U_p|^{o-1} \end{aligned} \tag{3.6}$$

As for the neural network evaluation function, it calculates the state evaluation score for player p by simply taking the difference of winning probability of the output neurons for both players.

[5]This corresponds to the *LanchesterEvaluationFunction* function of μRTS.

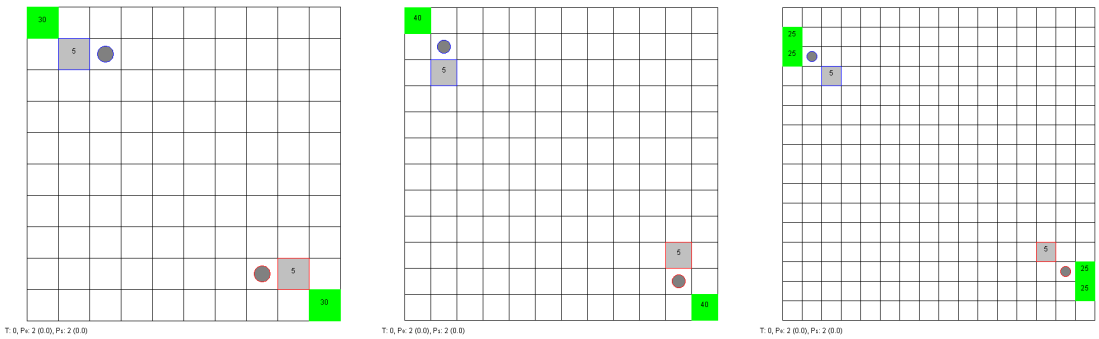


Figure 3.4: Example 10×10, 12×12 and 16×16 maps used in our experimental evaluation.

Winner Prediction Evaluation

In the first experiment, we evaluate our model by comparing the winner prediction accuracy with the baseline evaluation functions. We use the trained neural network and the baseline evaluation functions to predict the final winner of the instances in the test set (see Section 3.2.1). Figure 3.3 plots the prediction accuracy for all the compared evaluation functions as a function of how close the game was to the end, with 0 being the beginning of the game and 100 being the end of the game. To calculate this plot, we aggregated the results in 5% buckets.

First, we can see that the performance of baselines is very close to each other, especially at the beginning. Towards the end, Lanchester and Simple seem to edge SimpleSqrt. Our neural network consistently outperforms the baselines, especially at the beginning of the game, which usually has a significant impact on the game, and it is when the prediction task is harder. It is worth noting that we are using the same parameters and weights for Lanchester model from Stanescu et al.'s work[65], where they specifically optimized for the dataset used in their experiments. In their work, they report the Lanchester model to significantly outperform the Simple and SimpleSqrt baselines, which we do not observe in our evaluation. This indicates that there might be some amount of overfitting in their training procedure towards the maps used in that work, and that this evaluation function does not seem to generalize well to our dataset which includes a larger variety of maps, and AIs playing with different amounts of computational budget. Also interestingly, Simple seems to outperform SimpleSqrt, even if in the literature of RTS games it has been shown several times[71] that

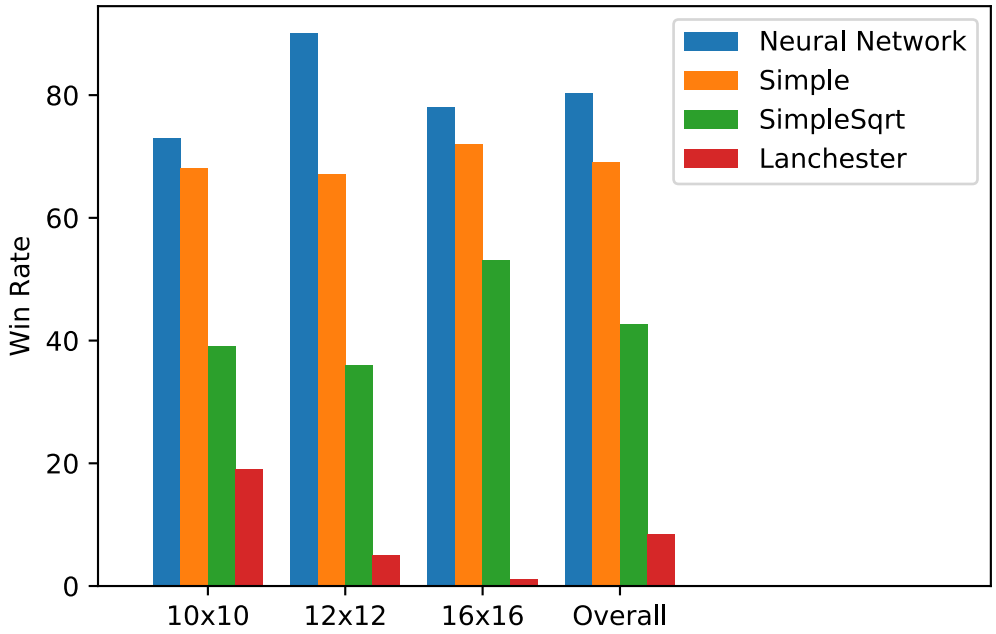


Figure 3.5: Gameplay strength results with playout budget

using the square root of the hitpoints of units results in stronger evaluation functions (as SimpleSqrt does). This does not seem to be the case in μRTS.

Gameplay Strength Evaluation

The second, and perhaps more important, experiment is to test the neural network evaluation function in actual game play under maps of different sizes. In this experiment, we have 10 maps with different initial states for each testing size (10×10, 12×12, and 16×16). Examples of maps of each size is given in Figure 3.4. Thus in total we have 30 maps for testing. We run a round-robin tournament between the neural network evaluation function and three baseline evaluation functions, all of which are coupled with *NaïveMCTS* with the same parameter configuration (the default parameters in μRTS for this algorithm). Each pair of bots play four games against each other in the tournament per map and are given the same computational budgets for both. Therefore, every pair of bots will play $10 \times 3 \times 4 = 120$ games.

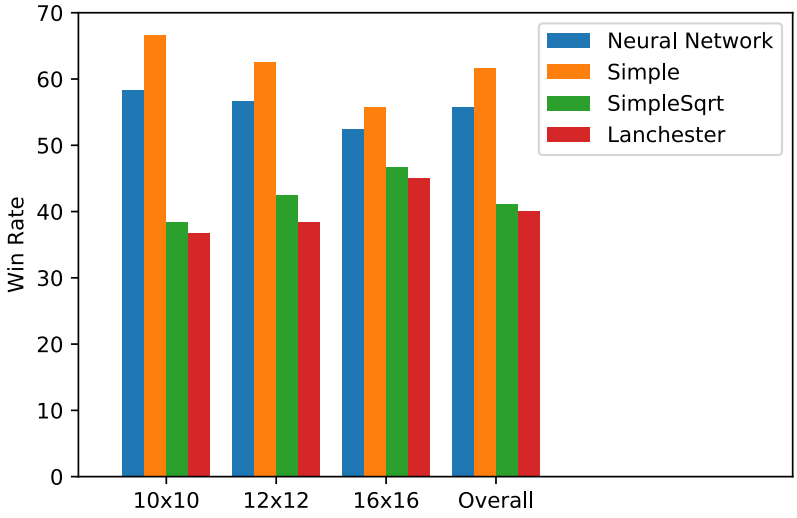


Figure 3.6: Gameplay strength results with time budget

We performed these experiments limiting the computational budget in two different ways: (1) the first is giving each bot only 200 playouts per game frame, and (2) giving each bot only 200 milliseconds per game frame. We use the win rate as the metric for game play strength. If there is a draw, both bots will receive a 0.5 win. The winning rate results are reported in figures 3.5 and 3.6 and the win/draw/loss matrix for each bot pair is reported in tables 3.12 and 3.13.

As we can observe from the results, when limiting the computation budget by the number of playouts, the neural network has a significant advantage over the baselines on gameplay strength. We can also see that SimpleSqrt and Lanchester perform particularly poorly when compared to the neural network. Additionally, it might seem that the performance of the neural network degrades when in 16x16 maps as compared to 12x12 maps. However, notice that what happens is that in 16x16 maps there are a lot of ties, since the map is large and sometimes players do not find each other. However, in 16x16 maps, as Table 3.5 shows, the neural network only lost 4 (out of 40) games against Simple, 1 against SimpleSqrt and 0 against Lanchester. This is a limitation of using playouts of length 100 in *NaïveMCTS*, rather than of the evaluation functions.

Concerning limiting the computation budget by time (as would happen in real-world settings),

Table 3.12: Gameplay strength results with playout budget

Result Matrix for 10 × 10 Maps				
	Neural Network	Simple	SimpSqrt	Lanchester
Neural Network	N/A	19/12/9	29/5/6	22/12/6
Simple	9/12/19	N/A	28/3/9	26/11/3
SimpSqrt	6/5/29	9/3/28	N/A	17/6/17
Lanchester	6/12/22	3/11/26	17/6/17	N/A

Result Matrix for 12 × 12 Maps				
	Neural Network	Simple	SimpSqrt	Lanchester
Neural Network	N/A	21/9/10	35/2/3	35/3/2
Simple	10/9/21	N/A	28/4/8	36/2/2
SimpSqrt	3/2/35	8/4/28	N/A	30/3/7
Lanchester	2/3/35	2/2/36	7/3/30	N/A

Result Matrix for 16 × 16 Maps				
	Neural Network	Simple	SimpSqrt	Lanchester
Neural Network	N/A	13/23/4	12/27/1	40/0/0
Simple	4/23/13	N/A	22/14/4	40/0/0
SimpSqrt	1/27/12	4/14/22	N/A	38/2/0
Lanchester	0/0/40	0/0/40	0/2/38	N/A

the Simple evaluation outperformed the neural network. This is due to two main reasons. First, the intrinsic reason that the neural network works less dominantly is that the time needed for the neural network to evaluate a single state is longer than that required by the baseline evaluation functions. However, a second significant contributing factor is that μRTS is written in Java and the neural network model is written and trained in Python. Thus on average the evaluation function spent around 37% of time budget on communication between the languages, which happens via a socket. We believe, that if this communication time is removed, the problem can be mitigated and our model will outperform all the baselines. In our experiments, the average ratio of playouts completed within the same time budget between the neural network and baselines are around 16:1, 13:1, and 9:1, respectively for maps of size 10×10, 12×12, and 16×16. If the communication time is eliminated, these ratios will go down to about 10:1, 8:1 and 5.5:1.

3.3 Conclusions

In this chapter, we investigated two approaches of learning domain knowledge from human-authored scripts and integrating the knowledge into MCTS. The first approach is policy-based, which learns

Table 3.13: Gameplay strength results with time budget

Result Matrix for 10 × 10 Maps				
	Neural Network	Simple	SimpSqrt	Lanchester
Neural Network	N/A	14/4/18	22/6/12	20/14/6
Simple	18/4/14	N/A	28/6/6	22/14/4
SimpSqrt	12/6/22	6/6/28	N/A	16/8/16
Lanchester	6/14/20	4/14/22	16/8/16	N/A

Result Matrix for 12 × 12 Maps				
	Neural Network	Simple	SimpSqrt	Lanchester
Neural Network	N/A	12/12/16	22/8/10	20/8/12
Simple	16/12/12	N/A	22/6/12	24/8/8
SimpSqrt	10/8/22	12/6/22	N/A	20/4/16
Lanchester	10/8/22	24/8/8	16/4/20	N/A

Result Matrix for 16 × 16 Maps				
	Neural Network	Simple	SimpSqrt	Lanchester
Neural Network	N/A	10/18/12	16/14/10	12/18/10
Simple	12/18/10	N/A	10/20/10	16/20/4
SimpSqrt	10/14/16	10/20/10	N/A	6/26/8
Lanchester	10/18/12	4/20/16	8/26/6	N/A

a policy from a script as the tree policy in the MCTS algorithm. The second one is value-based, which learns an evaluation function from the script and is used in the playout stage of the MCTS algorithm. Both approaches learn from the replays of the scripts playing against each other.

Our empirical results on the first approach showed that both classification accuracy and speed play a significant role when integrating machine learning models into MCTS. Specifically, a random forest model has the best classification performance and best gameplay performance as tree policy when the models are compared under iteration constraints. However, in experiments under time constraints, the random forest model has much worse performance compared to faster models. Instead, the C4.5 model, which has logarithmic complexity in classification time, has the best performance.

There are a number of future work directions we intend to pursue. For example, the models in this work ignore the interdependence of actions (when selecting an action for a unit, knowing which action was selected for another unit on the same game state might be relevant). Training a model that takes action interdependence into account should potentially result in better performing agents. Another possible improvement can come from models learned online using techniques like

contextual bandit algorithms [72;73].

For the second approach, our empirical results showed that we have successfully learned features from game replays collected under simple maps that can be transferred to larger maps. Specifically, the result of gameplay strength under time budget showed promise, but also showed that there are still challenges to be addressed. The main challenge is that the evaluation of the neural network is slower than the baseline evaluation functions used in our experiments. Possible solutions involve using smaller networks (we are currently using 512 filters, which could be reduced trading accuracy by speed). Another idea to explore further is to better exploit the parallel power of the GPU, for example, exploiting existing work on parallelization of the MCTS search algorithm [74], the evaluation function could be executed in batches, rather than state by state, shortening the speed gap between neural networks model and human-engineered evaluation functions.

Chapter 4: Guiding MCTS using Scripts

While in the previous chapter we studied how we can learn policies from scripts, in this chapter we study how we can use scripts directly. The motivation of the problem is two fold. First, a complex scripted bot may have very different behavior in different environments (e.g. initial map configurations), thus learning a robust policy from a complex scripted bot become very difficult. Using the scripts in the search algorithm enables us to build the search tree with the guidance of the scripts. Second, bandit algorithms tend to choose actions at random initially. This is problematic in games where we have a limiting search budget and need to search very deep, as a lot of the budget is lost in exploration. The key idea we want to explore is whether we can use scripts to help guide this initial exploration, by exploring actions that are likely to be good.

Due to the limited search budget in RTS games and the massive branching factor, it is usually unfeasible to produce a reliable estimation of the expected reward of the possible actions from sampling. Meanwhile, scripted bots are hard-coded agents using human knowledge, which follow simple and fast strategies, sometimes achieving a good level of gameplay strength. For example, scripted bots performed among the best bots in the first μRTS AI competition[75]. In this work, we propose two tree policies to incorporate scripted bots (which we will just call "scripts") as guidance for game tree search, namely single script guided naïve sampling (1-GNS) and mixed scripts guided naïve sampling (k-GNS). Note that with some tuning, a bot based on 1-GNS was able to achieve second place in the full observation track of μRTS competition at the 2019 CoG conference.

4.1 Single Script Guided Naïve Sampling (1-GNS)

This tree policy utilizes a single script when expanding the tree. The first time the tree policy is used on a search node to determine which is the first child that will be added to such node, instead of using stochastic sampling (like Naïve Sampling or ϵ-greedy do), or choosing the first non-expanded child (as UCB1 does), we propose to choose the macro-action suggested by the script. In all other

Algorithm 11: 1-GNS Node Expansion(n_0, s)

```
1 if n_0.children = ∅ then
2 |   n_1 = n_0.newNode(s.getAction);
3 else
4 |   n_1 = n_0.newNode(NaïveSampling(n_0));
5 end
```

circumstances, the tree policy selects a player action that consists of unit actions selected by Naïve Sampling. The resulting tree policy is shown in Algorithm 11.

The key idea behind this is that while the top nodes of the tree might be explored heavily by MCTS, as we go deeper into the tree, most nodes would only be visited once or a handful of times. Thus, by first selecting an action based on a script, we ensure that if a node is only visited once, at least the action selected is an action that makes sense, and not a random action. Moreover, since the script is only used the first time a node is visited, in the long-run, MCTS can ignore it if other actions turn out to have a higher reward. Thus, this does not change the theoretical convergence guarantees of MCTS.

4.2 Mixed Scripts Guided Naïve Sampling (k-GNS)

This tree policy utilizes a collection of scripts when expanding the tree. So, the first time the tree policy needs to be used in a tree node, rather than using a single script, k-GNS chooses a script uniformly from a pool of scripts, and uses the action produced by that script.

Moreover, k-GNS does not use the script just the first time, but also has a probability ϵ_0 of using them at any other iteration.

Notice that the ϵ_0 probability can also be applied in the 1-GNS policy. When the ϵ_0 is applied, 1-GNS becomes a special case of k-GNS, where the set of scripts contains only one script. The default ϵ_0 we use is 0.33. When we use larger ϵ_0, the behavior of the search will be forced towards the scripts, which can be useful in larger maps as we will show later in the experiments.

The resulting tree policy is shown in Algorithm 12. The idea behind having more than one script, and inserting this probability ϵ_0 of using the scripts at any iteration is to bring the actions

Algorithm 12: k-GNS Node Expansion(n_0, S)

```
if n_0.children = ∅ or with probability ε_0 then
    s = randomly select from S
    n_1 = n_0.newNode(s.getAction);
else
    n_1 = n_0.newNode(NaïveSampling(n_0, S));
end
```

proposed by the scripts to the attention of MCTS more often, and thus more strongly guide the search process. As we will show in our experimental results, this has very positive effects, especially for larger maps where the baseline NaïveMCTS gets lost in the search space.

We would like to point out that the idea of defining sampling policies that incorporate external policies is not new, as mentioned above in Section 2.6. Also, another multiarmed bandit strategy that incorporates this is PUCB (Predictor + UCB)[55], which modifies the standard UCB1 policy to incorporate weights from a provided predictor. Moreover, these approaches assume the external policy can provide a probability distribution of actions. The key contribution of this work is how to incorporate simple (potentially deterministic) scripts (that are simpler to specify and available for many RTS game domains) in a way that search can greatly benefit from them.

4.3 Experiments

We did experiments on five different maps, 8×8, 12×12, 16×16, 32×32, and 128×128. For each map, both players always start with one base and one worker). For experiments comparing 1-GNS/k-GNS against NaïveMCTS, we used 8×8, 12×12, and 16×16 maps. For experiments comparing search algorithms against the scripts, we used all five maps. For each map, 100 games are played between guided NaïveMCTS and baseline NaïveMCTS. Thus, each variation of guided NaïveMCTS is tested in 100 games per map size. Games that go beyond 5000 cycles are considered a draw and in each cycle we give players a computation budget of 500 iterations of MCTS expect 128×128 map, where 100 iterations are given.

Scripted Bots used for Guidance

We used four simple scripted bots provided by μRTS as guidance bots:

- *WorkerRush*: hardcoded deterministic bot that constantly produces Worker units and sends them to attack the opponent's units and base
- *LightRush*: hardcoded deterministic bot that constantly produces Light units and sends them to attack the opponent's units and base
- *HeavyRush*: hardcoded deterministic bot that constantly produces Heavy units and sends them to attack the opponent's units and base
- *RangedRush*: hardcoded deterministic bot that constantly produces Ranged units and sends them to attack the opponent's units and base

Experiments with 1-GNS

We run experiments of 1-GNS with four different scripts playing against vanilla NaïveMCTS in three sets of maps with increasing sizes. The aggregated win ratio results are shown in Table 4.1 where 1.0 means 1-GNS wins 100% of the times, and 0.0 would mean NaïveMCTS wins 100% of the times. Results show that the performance of this approach gets better as the map size grows. The reason is that the action space grows with the map size and makes it even harder for the search to get accurate estimations, thus the script helps in the search process. In smaller maps MCTS search alone can find good actions by itself and the improvement is not significant.

Figure 4.1 contains the winrate and the standard error [1] of NaïveMCTS and three variations of GNS versus the scripts in five maps with increasing maps sizes. We can see that in all sizes of maps, 1-GNS achieved better performance than the baseline. However, 1-GNS cannot defeat the scripts in the 16×16 and the 128×128 maps. Especially in the 128×128, 1-GNS hardly win against the script, suggesting the guidance is insufficient for such huge search space. In the next section, we show that k-GNS can mitigate the problem by incorporating more and varied guidance.

Experiments with k-GNS

The combination of multiple scripts provided diversified yet good directions to explore. The results shown in Table 4.2 continue the positive trend shown in the results of 1-GNS, but we see even larger

[1] Where standard error is defined as the standard deviation divided by the square root of the number of samples.

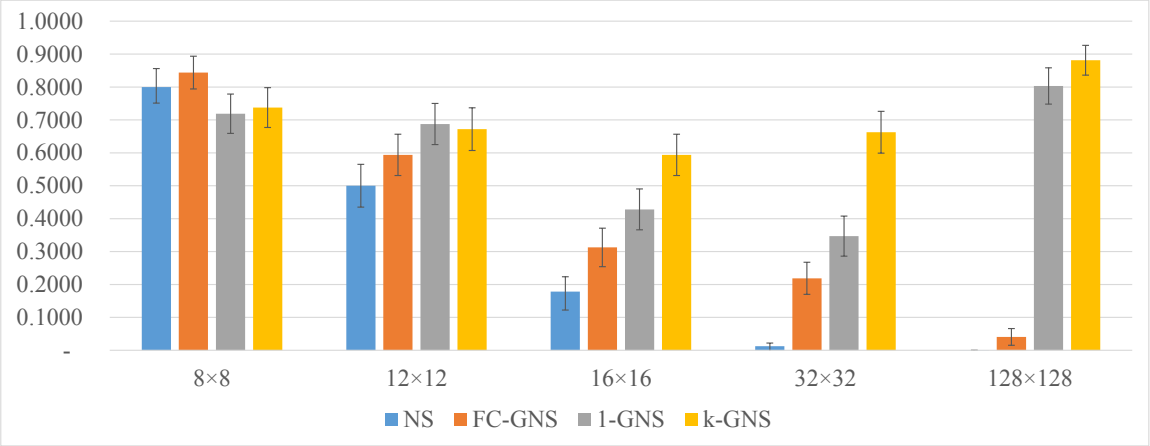


Figure 4.1: Win rates of NaïveMCTS, 1-GNS and k-GNS against the four rush scripts.

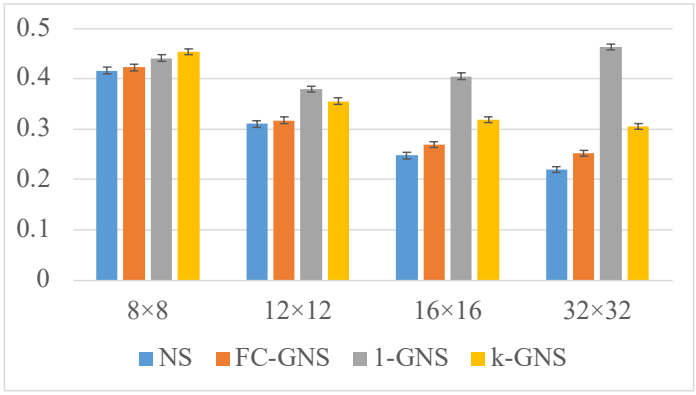
Table 4.1: Win Rates of 1-GNS guided by different scripts against NaïveMCTS.

	8×8	12×12	16×16
WorkerRush	0.5313	0.6521	0.6188
LightRush	0.5708	0.6438	0.7688
HeavyRush	0.5396	0.6125	0.7063
RangedRush	0.5042	0.5708	0.7188

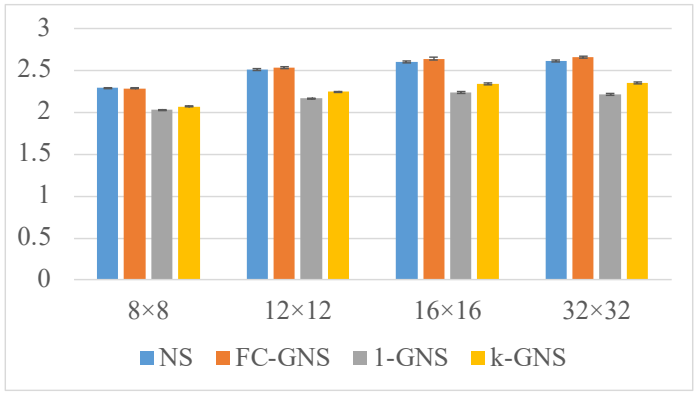
gains. Specifically, we see that our agents do no better than vanilla NaïveMCTS in 8×8 maps but has better performance in 12×12 and 16×16 maps (win rate of 95.83%, which is remarkable).

Additionally, we also compared the performance of 1-GNS (with and without ϵ_0) and k-GNS against the scripts in all five maps for reference. The ϵ_0 we used in the first four maps is 0.33 and for 128×128 maps we used 0.75. For the NaïveMCTS and k-GNS, we run experiments against each of the rush scripts, and for 1-GNS, we run experiments against the corresponding guiding script. The aggregated results are shown in Figure 4.1. We found that 1-GNS improved upon NaïveMCTS on their performance against the scripts in all maps. And k-GNS had even larger improvements except for the 8×8 map. This is because in very small maps, the search algorithm is able to find a better solution than the scripts and adding the bias of the scripts to the search will hurt the performance. It is worth noting that the superiority of the 1-GNS with ϵ_0 and k-GNS really starts to show in larger maps like 32×32 and 128×128. While in maps as large as 128×128, a larger ϵ_0 will help the search get focused in a huge search space.

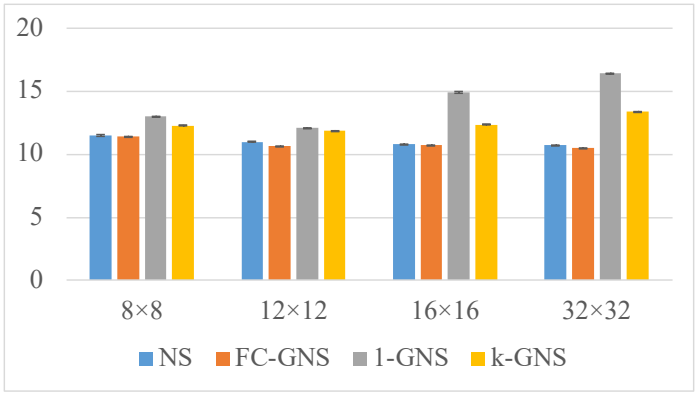
In order to understand where do the benefits in performance of 1-GNS and k-GNS, we analyze



(a) Average Tree Width



(b) Max Tree Depth



(c) Proportion of the Best Action being Visited at the Top Level

Figure 4.2: Comparison of Search Tree Statistics between Tree Policies with and without Guidance

the shape of the resulting search trees below.

Table 4.2: Win Rates of k-GNS against NaïveMCTS.

	8×8	12×12	16×16
Win Rate	0.4813	0.7896	0.9583

Comparison of the Shape of the Search Tree

To further understand the behavior of the proposed algorithms, we collected statistics of the shape of the search trees. The statistics are collected from 48 games between single/mixed scripts guided NaïveMCTS and vanilla NaïveMCTS (24 games for both strategies). We use data from the first 1000 game cycles of each game.

In Figure 4.2, we compare the shape of the trees generated by 1-GNS and k-GNS to the shape of vanilla NaïveMCTS using the mean and standard error of the following three statistics: (a) the average number of children per node, (b) maximum tree depth, and (c) the proportion of times that MCTS visited the selected action at the root of the tree. We can observe that the difference in the average number of children per node is insignificant between 1-GNS and the baseline. And the search tree of 1-GNS is slightly deeper than the baseline. While for k-GNS, due to more enforcement on the scripts, the average number of children per node is smaller and the tree is deeper. This could be because in 1-GNS, the node is only guided by the script at creation, and it does the same thing as the baseline the rest of the time. For k-GNS, the scripted action can be reinforced multiple times, thus the search is more concentrated. Finally, looking at the proportion of time that the selected action was visited at the root of the tree (the larger, the more concentrated the search is in one action), we see that in 8x8 maps, it seems that the scripts tended to distract MCTS rather than guide it, as adding more guidance made the search being less concentrated. However, in the larger maps, the more guidance, the higher the ratio of playouts that went through the selected action, which indicates that the scripts were helping MCTS in focusing the search.

Table 4.5 shows the proportion of times MCTS selected the action proposed by one of the scripts as the selected action. As can be seen, k-GNS selects actions from the scripts a larger percentage of times (up to 51.28% in 12x12 maps, whereas 1-GNS only selects the action proposed by the script

up to 18.29% of the time. We also see that in the smaller 8x8 map, MCTS overpowered the scripts more often in k-GNS than for the larger maps.

So, in summary, we see that 1-GNS generates trees that are very similar to vanilla NaïveMCTS, but k-GNS generates narrower and deeper trees. Moreover, the guidance has a stronger influence in larger maps, where we can see that the percentage of times the final action is one of the actions proposed by the script is higher than the percentage of times vanilla NaïveMCTS would have selected one of those actions.

NaïveMCTS with Guided Playout Policy

Finally, we explored the idea of using the scripts as the playout policy. In most of the MCTS variations, the playout policies are normally random plays to the end or a pre-defined depth. Due to the limited computing budget in RTS games, it would be beneficial to have a strong playout policy that can provide a good evaluation of the current state. However, as[76] pointed out, a stronger policy, whether it is learned or hard-coded, does not necessarily lead to stronger gameplay because of the bias in the playout policy.

In previous experiments, the playout policy we used is RandomBiased, which, despite its name, is less biased than the rush policies as a playout policy. In Table 4.4 we report the performance of the single script guided MCTS, each with the corresponding guiding scripts as the playout policies. We can see the result is quite catastrophic when the guiding scripts are used as playout policies. Our hypothesis of the results is that despite the scripts achieving good strength in gameplay, they are still not optimal, and the proposed algorithms work under the premise that the MCTS will refine the player action provided by the scripts. However, applying the scripts as playout policy can prevent the MCTS from recovering from these suboptimalities, since it will assume players will behave as the script indicates. Also, the following situation could happen: 1) when the playout indicates the agent has the edge, the agent plays boldly as the script suggests, however 2) when the playout indicates otherwise, the player action from the scripts gets overridden. This could create a "zigzag" effect, where the agent plays actions that cancel each other out, since in RTS games, macro goals, like "harvest" or "attack", take a series of actions to accomplish.

Table 4.3: Winrate of k-GNS against 1-GNS

	8×8	12×12	16×16
k-GNS	0.2833	0.6583	0.8417

Table 4.4: Win Rates of Guided NaïveMCTS with Scripted Playout

	8×8	12×12	16×16
WorkerRush	0.3875	0.2271	0.0708
LightRush	0.4938	0.2792	0.5104
HeavyRush	0.5208	0.1188	0.3229
RangedRush	0.5292	0.1354	0.4167

4.4 Conclusions

In this chapter, we studied how to incorporate scripts that contain human knowledge into the tree policy of MCTS in order to improve its performance in the context of RTS games, while still maintaining the original search space of MCTS, and ensuring MCTS would converge to the optimal action in the limit. We presented two tree policies (FC-GNS and k-GNS) that do such integration, one using a single script and the other using a collection of scripts. We also reported our experiments on using the scripts in the playout policy.

The experimental results showed that both tree policies could improve the gameplay strength upon the baseline algorithm. They also scale well to larger maps (the larger the map, the larger the increase in performance with respect to NaïveMCTS), where the baseline algorithm struggles to perform. Additionally, experimental results on using scripts for the playout policy suggested that the bias should be carefully considered when designing scripts for playout policies.

We applied this technique in the 2020 CoG μRTS AI competition. Our entries for the competition are called *MentalSeal* and *MentalSealPO* (one for the fully observable track and one for the partial observable track). We employed 1-GNS in the competition and instead of using the existing scripts in

Table 4.5: Proportion of Script Action as Best Action of Search

	8×8	12×12	16×16
1-GNS	0.1739	0.1829	0.1791
k-GNS	0.4300	0.5128	0.5075

*mu*RTS, we designed a parameterized script, and optimized the values of those parameters separately for each map of the competition. The script is parameterized by the following parameters:

- The number of harvesting workers to build at the beginning.
- The number of attack unit needed before starting attacking (waving strategy).
- The number of bases to build at the beginning.
- The number of barracks to build at the beginning.
- The probability of producing Light unit against other types of attack units.

For the optimization of the parameters, we use a technique called fictitious play, which will be introduced in Chapter 5. The fictitious play algorithm produces a mixture of a collection of scripts, we pick the one script in this collection that has the highest occurrence during the optimization process.

For the partial observable track, the script has the same parameterization and has hard-coded scouting behaviour to deal with the partial observability. MentalSeal achieved the third place (and the first among search-based bots) in the fully observable track and MentalSealPO achieved first place in the partial observable track.

This work opens a wide range of possible future work. In particular we would like to study the effects of the ϵ parameter in k-GNS, which controls how much guidance is provided by the scripts. Additionally, another interesting problem is how to design a strong and less biased script for the playout policy. Finally, we want to compare with other ways to exploit scripts in the literature both from a practical and theoretical point of view.

Chapter 5: Bandit-Based Policy Optimization

Despite the success MCTS had in the domain of game AI, there still has been much room for a better understanding of the different components of MCTS. In previous chapters, we have studied how to leverage the domain knowledge in the form of scripted bots and use them as the tree policies. However, it is not clear this is the best way to obtain and apply a policy. The intuition we had in the previous chapters is that if a policy is strong in game playing, then it would also be a good tree policy or playout policy. In this chapter, we take a closer look into the problem of what makes a good playout policy and tree policy.

Motivated by the question of what makes a good playout policy, we first empirically study the effect of optimizing playout policies with different objectives for MCTS in the domain of real-time strategy (RTS) games. In almost all variations of MCTS, playout policies, also called simulation policies, are used to select actions for both players during the forward simulation phase of the search process. Since the quality of the playout policy has a great impact on the overall performance of MCTS, previous work has covered various of methods to generate these policies such as handcrafted patterns[77], supervised learning[39], reinforcement learning[78], simulation balancing[76;79;80], and online adaptation[56;81]. However, there is little generalizable understanding about how to design or learn good playout policies in systematic ways. Optimizing directly on the gameplay strength of the playout policy often yields decreased performance[78] (an effect we also observed in preliminary experiments, and which partially motivated this work). In recent Go research, playout policies are sometimes abandoned and replaced by refined evaluation functions[46;82].

In this chapter, we first introduce the methods that we use to optimize the policies. Then we use the methods to optimize for different objectives and compare their performance as playout policies. Specifically, we explore two approaches: one based on combinatorial bandits to optimize policies in general, and one based on contextual combinatorial bandits to try to make policies adaptive to specific maps.

5.1 Playout Policy Optimization in MCTS

5.1.1 Policy Optimization in μRTS

In order to study the differences between policies optimized for gameplay and those optimized directly as playout policies and tree policies, we define a very simple parametrized policy, and use an optimization process to optimize these parameters.

Policy Parameterization

We employ a simple parameterization to represent a space of stochastic policy, where we define a weight vector $\mathbf{w} = (w_1, ..., w_6)$, where each of the six weights $w_i \in [0, 1]$ corresponds to each of the six types of actions in the game:

- NONE: no action.
- MOVE: move to an adjacent position.
- HARVEST: harvest a resource in an adjacent position.
- RETURN: return a resource to a nearby base.
- PRODUCE: produce a new unit (only bases and barracks can produce units, and only workers can produce new buildings).
- ATTACK: attack an enemy unit that is within range.

A policy is totally represented by the vector $\mathbf{w}$. During gameplay, the action for each unit is selected proportionally to this weight vector. To choose the action for a given unit, the following procedure is used: given all the available actions for a unit, a probability distribution is formed by assigning each of these actions the corresponding weight in $\mathbf{w}$, and then normalizing to turn the resulting vector into a probability distribution. If the weights of all the available actions are 0, then action is chosen uniformly at random. Notice that this defines a very simple space of policies, but as we will see below, it is surprisingly expressive, and includes policies that are stronger than it might initially seem.

The goal of keeping the policy space simple is to be able to find near-optimal policies (within the policy space), in a computationally inexpensive way. The same ideas presented here would apply to more expressive policies, parameterized by larger parameter vectors, such as those represented by a neural network, for example (although a different optimization algorithm might be required, such as reinforcement learning).

Policy Optimization

Given the parameterization, we can optimize the policy for many purposes using different optimization algorithms. We propose two methods, fictitious play (FP) and repeated game of bandits (RGB)[83;84].

Fictitious play works as in Algorithm 13, where in each iteration we optimize a new policy against the pool of previous policies. RGB works as in Algorithm 14, where two regret-minimizing agents repeatedly play against each other. And if the repeated game is zero-sum, the empirical distribution of FP and RGB converges to Nash Equilibrium[83]. The motivation of both algorithms is that if a policy is optimized to maximize win rates against a single other agent, cycles might be created, where we have three policies A, B, and C, and A beats B, B beats C, and C beats A. To avoid these cycles and compute the least exploitable agent, we need to approximate the Nash Equilibrium. The details of the two algorithms will be discussed later.

In particular, in this work we use multiarmed bandits as a way to compute the best response in FP and RGB. For bandit optimization, we discretized the search space, allowing each weight to take values in $\{0, 1, 2, 3, 4, 5\}$. Specifically, we model the problem using combinatorial bandits, since the problem has a combinatorial structure where there are 6 types of actions and for each action type there are 6 different weights to choose from. Moreover, notice that if we multiply a weight vector by a scalar strictly larger than zero, the resulting policy is identical in behavior. Internally, when interpreting the weight vectors as a policy, the vector will be normalized to a probability distribution (that sums up to one).

Algorithm 13: Naïve Sampling Fictitious Play(m)

```
Initialize the set of the Nash Equilibrium strategy N as a set with one policy at random
  from the space of policies.
for k = 1, 2, 3, ..., T do
    CMAB = new NaïveSampling() bandit
    for k = 1, 2, 3, ..., K do
        Randomly pick one opponent policy π_o ∈ N
        Choose arm π_k = CMAB.sample()
        r = play a game π_k vs π_o in map m
        CMAB.observeReward(π_k, r)
    a* = CMAB.bestMacroArm()
    N ← N ∪ {a*}
```

Fictitious Play with Bandit Optimization. In order to find the optimal policy within the space of policies defined by our 6-parameter vector, we use Naïve Sampling within a fictitious play framework. Specifically, we use Algorithm 13. Given a target map m (notice that, in principle, we can use a set of maps, but for simplicity, we just optimized policies for specific maps in this work), we use fictitious play as follows. We initialize a set of policies N with a single policy chosen at random, and then execute T iterations of fictitious play. At each iteration, we optimize a policy a^* to be the best response against the policies in N. To do this, we use K iterations of NaïveSampling, after which we add this new best response policy to N, and iterate again. As we discussed above, it has been shown that N converges to the Nash Equilibrium. Finally, since in this work we just want to select a single policy at the end, in our experiments, we then ran a final NaïveSampling optimization process with a very large number of iterations, trying to find the best response policy against the Nash Equilibrium N. We do this, in order to obtain a policy represented just as a vector of 6 numbers, and make results interpretable, so we can compare the result of optimizing for gameplay strength, versus optimizing for playout strength.

In order to optimize a policy for being a strong playout policy, rather than a strong gameplay policy, we use the same exact procedure, except that when playing a game between π_o and π_k, we use MCTS agents where π_o and π_k are used as the playout policies.

Algorithm 14: Repeated Game of Bandits (with Naïve Sampling)

```
Initialize Nash Equilibrium strategy set N = ∅.
Initialize two bandit agents B_1 and B_2.
CMAB_1 = new NaïveSampling() bandit
CMAB_2 = new NaïveSampling() bandit
for k = 1, 2, 3, ..., T do
    Choose arm π_k^1 = CMAB_1.sample()
    Choose arm π_k^2 = CMAB_2.sample()
    r = play a game π_k^1 vs π_k^2 in map m
    CMAB_1.observeReward(π_k^1, r)
    CMAB_2.observeReward(π_k^2, 1 − r)
    N ← N ∪ {π_k^1, π_k^2}
```

Zero-Sum Repeated Game of Bandits In order to find the optimal policy within the space of policies defined by our 6-parameter vector, we use Naïve Sampling within the RGB play framework. Specifically, we use Algorithm 14. Given a target set of maps m, we use RGB as follows. We initialize a set of policies N. And then execute T iterations of repeated games between two regret-minimizing bandit agents B_1 and B_2. At each iteration k, two arms, π_k^1 and π_k^2, are pulled from each bandit independently and simultaneously. Then 10 games are played between the policies and the averaged reward $r \in [0, 1]$ is revealed to both bandits (r to B_1, $1 - r$ to B_2). Both of the selected arms are added the N. As we discussed above, it has been shown that N converges to the Nash Equilibrium.

However, in this study, we stick to the single policy for analysis and in order to obtain a policy represented just as a vector of 6 numbers, and make results interpretable, so we can compare the result of optimizing for gameplay strength, versus optimizing for playout strength. The final weight vector will be the most visited arm after the bandit optimization process.

In order to optimize a policy for being a strong playout policy, rather than a strong gameplay policy, we use the same exact procedure, except that when playing a game between π_k^1 and π_k^2, we use MCTS agents where π_k^1 and π_k^2 are used as the playout policies.

Furthermore, we also experiment with optimizing the policy as the tree policy of the MCTS, in order to observe its difference to policies optimized for game-playing strength. The research question to ask is whether it is enough to optimize only for game-playing strength to have a good playout or tree policy.

Finally, we optimize the tree policy and playout policy directly at the same time. The purpose is to see if there are possible interactions between the two types of policies and potentially obtain policy combinations that work better than optimizing them separately.

5.1.2 Experiments and Results

In our previous work[85] we presented the result of bandit optimized policies of the same parameterization. However, we did not compare with policies optimized using other techniques, such as *simulation balancing*[76] in order to assess if just using simulation balancing is enough to obtain strong playout policies. Thus, in this work, we first establish a baseline using simulation balancing and show that it does not scale well in RTS games. Then, we further investigate the bandit based optimization approach and the effect of the different optimization objectives described above.

Three different maps are used to test the generalizability of our comparison. The maps are:

- Map 1: *8x8/basesWorkers8x8A.xml*: In this map of size 8 by 8, each player starts with one base and one worker. Games are cut-off at 3000 cycles.
- Map 2: *8x8/FourBasesWorkers8x8.xml*: In this map of size 8 by 8, each player starts with four bases and four worker. Games are cut-off at 3000 cycles.
- Map 3: *NoWhereToRun9x8.xml*: In this map of size nine by eight, each player starts with one base and the players are initially separated by a wall of resources, that needs to be mined through in order to reach each other. Games are cut-off at 3000 cycles.

Monte Carlo Simulation Balancing

Simulation Balancing (SB)[76;79;80] approach the problem of optimizing for good playout policy by not optimizing policy strength but optimizing policy balance. It is a policy gradient-based method that minimizes "imbalance" in the policies so that the small errors cancel each other out during the whole playout. The pseudocode is given in Algorithm 15. The algorithm first constructs a training set of state/state value pairs. The true state value can be estimated by performing a deep MCTS search when expert play is not available Then the algorithm uses Monte Carlo simulation to calculate the actual state value estimation of the given policy. Finally, the algorithm calculates the difference

Algorithm 15: Simulation Balancing

```
θ ← 0
for t = 0 to T do
    (s_1, V*(s_1)) ← Random choice from training set
    V ← 0
    for j = 0 to M do
        simulate (s_1, a_1, ··· , s_N, a_N, z) following π_{θ_t}
        V ← V + z
    V ← V/M
    for i = 0 to M do
        simulate (s_1, a_1, ··· , s_N, a_N, z) following π_{θ_t}
        g ← g + z Σ_{n=1}^{N} ψ_{θ_t}(s_n, a_n)
    g ← g/M
    θ_{t+1} ← θ_t + α(V*(s_1) − V)g
```

of the true state value and estimated state value to do policy gradient update. The policy gradient $\psi_{\theta_t}(s_n, a_n)$ is the following

$$\psi_{\theta_t}(s_n, a_n) = \nabla_\theta \log \pi_\theta(s, a) = \phi(s, a) - \sum_b \pi_\theta(s, b)\phi(s, b) \tag{5.1}$$

In the equation, ϕ is the feature vector and θ is the policy.

Now we experiment with the performance of SB. We first collect a dataset of estimated true state values from the three maps using NaïveMCTS of 100000 iterations and the value estimation of the root node is recorded as the estimated true state value. 1000 states are sampled from 200 self-played games of two random agents. During training, we first calculate the state value estimated by the playout policy V by averaging 1000 playouts. Then we run another 1000 playouts to calculate policy gradient ψ_{θ_t}. The resulting policy of SB optimization is characterized by the parameter vector $[0.02, 0.32, 0.18, 0.18, 0.17, 0.13]$. Together with a purely random and the built-in RandomBiased bot in μRTS, the result from SB will be used as the baseline in our study.

Optimization for Gameplay Strength

In the first experiment, we optimize the policy with multiple maps together and compare with the policies in[85]. Specifically, we run 10000 iterations of the repeated game of bandits between two Naïve Sampling agents to obtain a history distribution of the process. The arms pulled by the two bandits

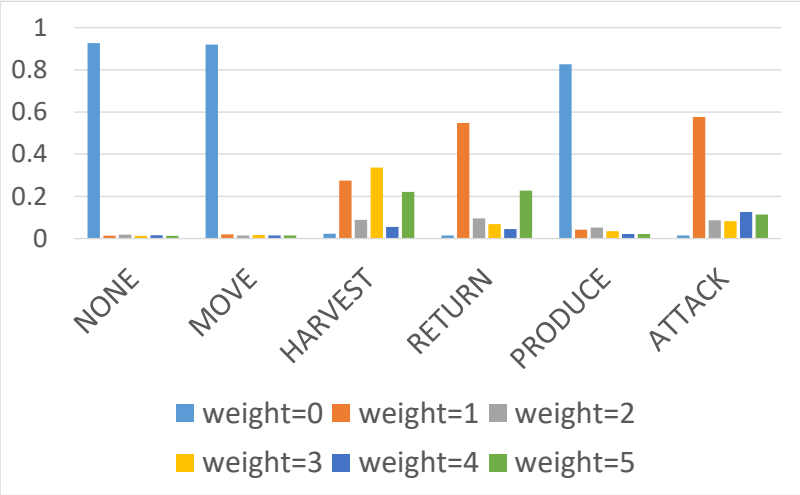


Figure 5.1: Gameplay Strength Optimization

correspond to the game-playing policies and play against each other for 10 games to calculate the reward. The result is 20000 policies (the policy of each of the two players over 10000 itertions). We visualized the weight distribution of these 20000 policies.

Since the strategy consists of a pool of parameterized policies, we visualize this pool of 20000 policies by plotting the proportion of times each action is assigned to each of the different possible weights ($\{0, 1, 2, 3, 4, 5\}$). For example, the result for gameplay strength optimization is shown in Figure 5.1. We observe that NONE, MOVE, and PRODUCE are most frequently assigned with a weight of 0 in most of the policies in the pool. RETURN and ATTACK are more frequently given a weight of 1. HARVEST and RETURN are given more diverse weights, probably due to the fact that we use different maps, and some values might work better in some maps than in others. Later in the document, we will evaluate how strong these policies are in actual gameplay.

Optimization for Tree Policy

In the second experiment we optimize the tree policy directly as opposed to optimizing for gameplay strength. The experimental setup is similar to the gameplay strength optimization but the performance of the policies is measured directly by using them as tree policies of MCTS. Again, we run 10000 iterations of the repeated game of bandits between two Naïve Sampling agents. The arms pulled by the two bandits are used as tree policies and used by MCTS agents to play against each

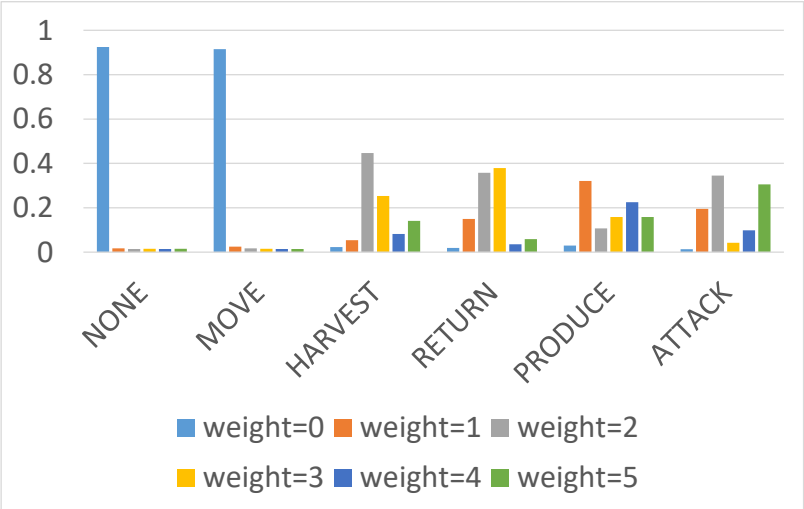


Figure 5.2: Tree Policy Optimization

other for 10 games to calculate the reward.

The result of the optimization is visualized in Figure 5.2. It is easy to see that the results agree with the gameplay strength optimization that NONE and MOVE should assign a 0 weight with high probability, but disagree that PRODUCE should have a low probability for 0. Also, HARVEST, RETURN, and ATTACK have more spread weights than gameplay optimization. This shows that strong gameplay policies tend to be different from strong tree policies.

Optimization for Playout Policy

In the third experiment we optimize for the playout policy with a similar experimental setup as in tree policy optimization. We run 10000 iterations of the repeated game of bandits between two Naïve Sampling agents and the arms pulled are interpreted as playout policies and used by MCTS agents to play against each other for 10 games to calculate the reward.

The result of the optimization is visualized in Figure 5.3. We can observe that the weight distribution is very different to the distribution of optimization of tree policy or gameplay policy. In this weight distribution, NONE is mostly assigned to weight 1. MOVE and PRODUCE are mostly assigned weight 0. And ATTACK is mostly assigned to the highest weight of 5. HARVEST is spread between 1, 2, and 3. RETURN has most of the weights assigned to 1 and 5. Again, we see that

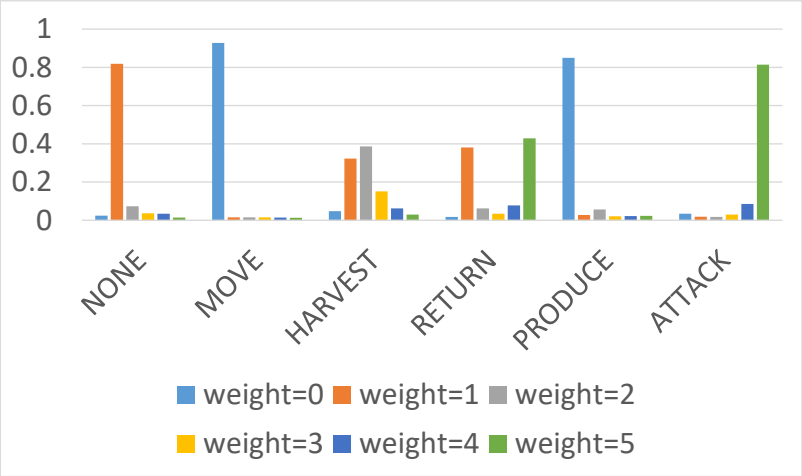


Figure 5.3: Playout Policy Optimization

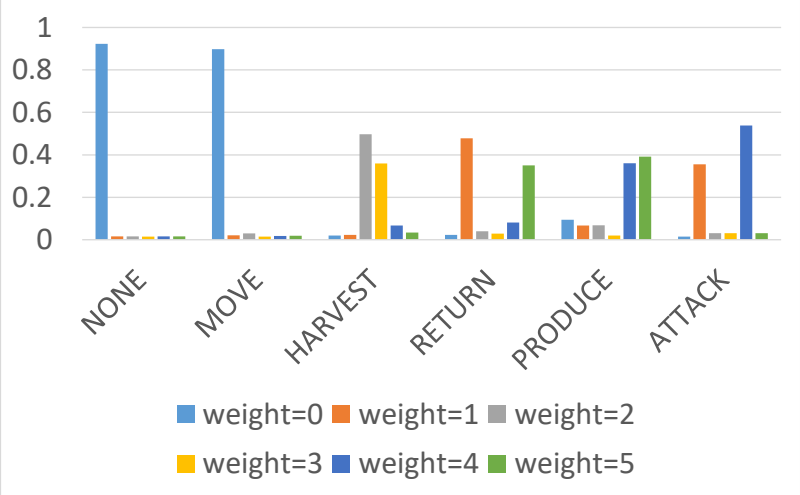


Figure 5.4: Tree Policy in the Joint Optimization

strong playout policies are very different from strong gameplay policies.

Joint Optimization for Tree Policy and Playout Policy

To test whether tree policy and playout policy interact with each other, we further investigate by optimizing both at the same time. Similarly, We run 10000 iterations of the repeated game of bandits between two Naïve Sampling agents that choose values for 12 parameters rather than 6, and the arms pulled will be interpreted as two policies, one for tree policy and the other for playout policies, and used by MCTS agents to play against each other for 10 games to calculate the reward. Thus, in this experiment, arms pulled by bandits have 12 parameters and the first six parameters

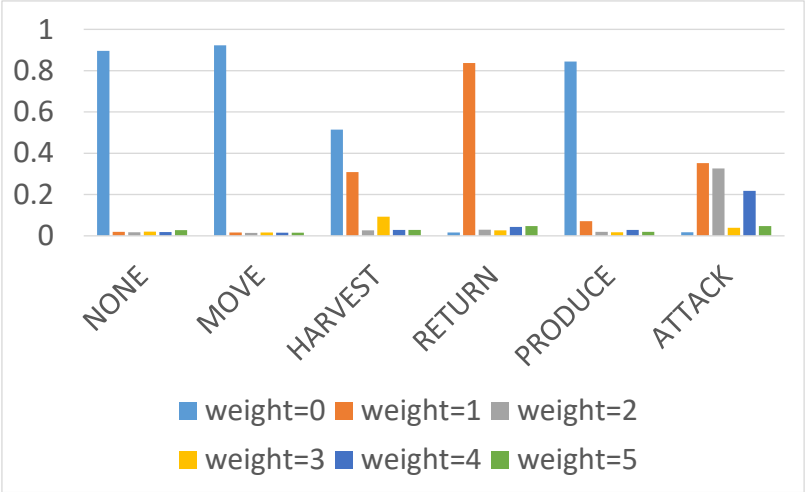


Figure 5.5: Playout Policy in the Joint Optimization

are interpreted as tree policy and others are interpreted as playout policy.

The result of the optimization is visualized in Figure 5.4 and Figure 5.5, showing that the jointly optimized policies are different from the policies obtained when optimizing them separately. Let us now compare how strong these policies are in actual gameplay.

Comparing Performance as Gameplay Policies vs. Playout Policies

So far we have policies optimized for different objectives:

- Optimizing "simulation balance".
- Optimizing gameplay strength of the policy.
- Optimized as tree policy of MCTS.
- Optimized as playout policy of MCTS.
- Joint optimization of tree policy and playout policy.

Now, together with the two baselines, Random and RandomBiased, we compare their policies in two tasks: gameplay strength when used directly to play (without MCTS), and gameplay strength when used as playout policies within MCTS. We run 10 rounds of round-robin between all the

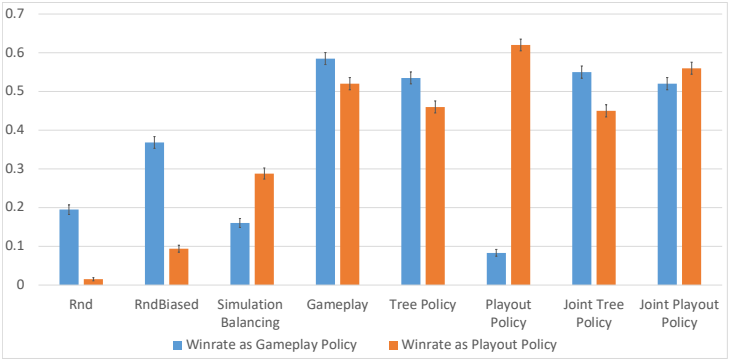


Figure 5.6: Comparison on winrates of policies serving as game-playing policy and playout policy of an MCTS agent.

policies. The winrates and the standard error are reported in Figure 5.6 (we tested the tree policies separately as reported below).

First, simulation balancing has a winrate of 0.16 as game-playing policy and a winrate of 0.29 as the playout policy, which are outperformed by the two baselines as game-playing policy (winrate of 0.20 and 0.37 respectively), but is better than baselines as playout policy of MCTS (winrate of 0.02 and 0.09 respectively). This is expected, as simulation balancing is supposed to design strong playout policies.

Second, the policy optimized for gameplay strength outperformed the baselines by a large margin and has the best gameplay winrate (0.58) and third best winrate as playout policy (0.52). For performance as playout policy, it is only worse than the two optimized as playout policies directly. The policy optimized as tree policy of MCTS also outperformed baselines, but has a worse winrate than gameplay optimized policy in both tracks (winrate of 0.53 and 0.47).

Now we look at the policy optimized for playout policy directly. The result is interesting since it is very weak in terms of gameplay (winrate of merely 0.08), but very strong as playout policy (winrate of 0.62). This suggests that a strong gameplay strength is not a requirement of being a good playout policy, and that simulation balancing does not capture all that is required for a strong playout policy.

Lastly, we have the pair of policies that are optimized together, one as tree policy and the other as playout policy. The tree policy achieved similar winrates (winrate of 0.55 and 0.45 respectively)

as singly optimized. The policy optimized as playout policy is interesting that not only is it good as playout policy (winrate of 0.56), but also it has a good gameplay strength (winrate of 0.52).

Strength of Tree Policies

The result of the jointly optimized policies suggests there could be some factor of "match" between the tree policy and the playout policy for them to work well together. Thus, to further verify this hypothesis, we take the best pairs of singly optimized tree policy and playout policies to play against the pair of jointly optimized policies in two MCTS agents.

We run the jointly optimized pair against the pair of best gameplay policy (as tree policy) and best singly optimized playout policy (as playout policy) for 1000 games, and the jointly optimized pair has a winrate of **0.64**. We also run the jointly optimized pair against the pair of best tree policy and best singly optimized playout policy for 1000 games, and the jointly optimized pair has a winrate of **0.67**.

Moreover, we run gameplay optimized policy against an optimized tree policy as the tree policy of an MCTS agent for 1000 games, both with the optimized playout policy. We found the gameplay-policy has a winrate of **0.44**, which means that a gameplay optimized policy does not necessarily make for a good tree policy.

5.2 Optimizing Adaptive Policies

In this section, we describe our approach of optimizing policies that are adaptive (specifically, the initial map configurations). The adaptive policies are obtained by leveraging the contextual combinatorial multiarmed bandits (CCMAB)[86]. The problem deals with the situation where the bandit problem with combinatorial structure and a context is given before observing the reward. The context given can be at different levels of the combinatorial bandits. First, a context can be given at the macro arm level which is potentially used to infer reward distribution of other macro arms. Second, contexts can be given at the micro arm level and the learning agent should combine these contexts to infer the reward of other macro arms. For this work, we focus on the first scenario where a context is given at the macro arm level.

In CCMAB, the reward r_t in each round t depends both on the context σ_t and the chosen macro arm V_t, and V_t consist of a set of n micro arms $V_t = \{v_t^1, \ldots, v_t^n\}$.

The difference between CCMAB and contextual bandits is that in contextual bandits there is a single variable representing the arm to choose, while in CCMAB there are n variables to choose values for, defining a combinatorial space. And the difference between CCMAB and combinatorial bandits is that in combinatorial bandits, the reward r only depends on the chosen combination of n variables and in CCMAB the reward depends on both the context vector σ and the chosen combination of arms.

To measure the performance of the strategies to address CCMAB, we use the notion *regret*, which is the difference between the expected reward of the selected macro arm V_t, and the expected reward of an optimal macro-arm V^*. There are many ways to compute the regret and the most commonly used and what we used in this work is the *cumulative regret*, which is the sum of differences between r^* and the reward obtained by the selected macro arms at each iteration.

$$R_T = \sum_{t=1}^{T} (r^* - r_t) \tag{5.2}$$

Thus, in our problem settings, we will optimize the CCMAB for T iterations to find the best macro arm with the lowest cumulative regret or highest cumulative rewards.

We present and compare three approaches for contextual combinatorial bandits: naïve aggregation, pair-wise aggregation, and contextual global optimization approach.

5.2.1 Naïve Aggregation

In naïve aggregation, we make the naïve assumption that reward distribution of the macro arm can be factored by estimating a reward distribution for each micro-arm, i.e. we assume the reward for each micro-arm is independent. Because of the naïve assumption, we can break down the CCMAB to n contextual bandit problems. As shown in Algorithm 16, at each round t, the agent first observes a context vector σ_t. σ_t has d dimensions to represent d features of the context. Then each micro MAB observes σ_t and picks a micro arm v_t^n according to the pre-defined contextual bandit algorithm. For

Algorithm 16: Naïve Aggregation

```
for each round t do
    1. macro MAB observes the context σ_t
    2. each micro MAB X_n picks an arm v_t^n according to learner L^{X_n}
    3. macro MAB V_t picks the combination formed by {v_t^1, ..., v_t^n}.
    3. reward r_t observed for the chosen marcro arm, and r_t is fed back to each v_t^n.
```

example, if we use contextual bandit algorithms with a policy class as mentioned before. A policy is based on a machine learning model L, and the learning algorithm is used to predict the mean reward. The macro arm is selected as the legal combination of micro arms with the highest sum of mean rewards. Since we have n micro MABs, for each micro MAB X_n we have a learner L^{X_n}.The reward r_t observed is fed back to each bandit policy of micro MABs.

The advantage of naïve aggregation is that thanks to the naïve assumption, the problem is simplified from $O(K^n)$ to $O(nK)$, with n being the number of micro MABs and K being the number of arms of each micro MAB. However, the efficiency we gained is because we trade-off the interdependence between the local MABs. Thus, the next technique, called pair-wise aggregation, tries to bring back some of the interdependence on top of the naïve aggregation.

5.2.2 Pair-Wise Aggregation

As we have discussed, the naïve aggregation trade-off interdependence of micro MABs for efficiency. Here we describe pair-wise aggregation that brings back partial interdependence. We consider a pair of micro arms of two different micro MABs $\{a_i^{X_n}, a_j^{X_m}\}$ to be an arm in the pair-wise MAB. Thus, for a macro MAB with n K-armed micro MABs, the number of legal pairs will be $O(K^2n^2)$. And each macro arm V_t picked at time t, will generate training examples of K^2 pairs.

Algorithm 17 demonstrates how pair-wise aggregation works. On top of naïve aggregation, we build pair-wise learners on the side and pick arms by merging the two. First, the agent observes the context vector σ_t and reveal σ_t to both micro MAB learners L and all the relevant pair-wise learners P. L and P will separately predict the expected reward. Then we can pick the macro arm according to the prediction. Finally, the observed reward is revealed to L and P to update the learners.

The advantage of adding pair-wise dependence is that it is a more realistic modeling of many

Algorithm 17: Pair-Wise Aggregation

for *each round* t **do**
- 1. macro MAB observes the context σ_t
- 2. each micro MAB X_n picks an arm v_t^n according to learner L^{X_n} and all relevant pair-wise learner $P^{X_n \times X_m}$
- 3. macro MAB V_t picks the combination formed by $\{v_t^1, \ldots, v_t^n\}$.
- 3. reward r_t observed for the chosen macro arm, and r_t is fed back to each v_t^n and $P^{X_n \times X_m}$

real world problems. For example in the news placement example, it is very likely that the news shown together are in the same category or theme. Adding modeling of pair-wise relations allows us to capture some interdependence of the micro MABs.

5.2.3 Contextual Global Optimization

In this section we describe a simple approach of incorporating the full interdependence between the micro arms. In this strategy, we "flatten" the combinatorial structure and only consider the macro arms. We refer to this method as contextual global optimization. For a problem with context vector σ that has d dimension and m possible values in each dimension, the number of contexts is $O(m^d)$. And for combinatorial bandits, the number of macro arms is $O(K^n)$. To apply contextual global optimization, we construct *context-macroarm* vectors, which consist of the concatenation of the context vector σ to the vector representing a given macro-arm, and then we can train a machine learning model (the *context learner* L) to predict the expected reward of a given macro-arm in a given context by learning to predict the reward given this context-macroarm vector. Essentially, we convert the CCMAB problem into a contextual bandit problem where the macro-arm is represented by the specific micro-arms that compose the macro-arm.

Specifically, during the bandit optimization process, we first receive the context vector σ, then we select the macro arm according to the contextual learner. When the reward r_t is observed, we construct a training example as $\{[\sigma, V_t], r_t\} = \{[\sigma_1, \ldots, \sigma_d, v_t^1, \ldots, v_t^n], r_t\}$. This example is given to a context learner, which is retrained every iteration with the set of collected training examples to enable the model to generalize. The learner can be any machine learning model like a decision tree, a Bayesian net, neural network depending on the need (for simplicity, we used decision trees

Algorithm 18: Contextual Global Optimization

for *each round* t **do**
- 1. macro MAB observes the context σ_t
- 2. Pick the macro-arm V_t according to the context σ_t using the context learner L as $argmax_{V_t \in X} L([\sigma, V_t])$.
- 3. reward r_t observed for V_t.
- 4. Construct training example $\{[\sigma, V_t], r_t\}$ and feed it back to the context learner..

in our experiments, which gave us the best tradeoff of speed/performance in our experiments given the amount of data we collect).

By applying contextual global optimization we convert the problem to a contextual bandit problem with $O(K^n)$ arms and context space of $O(m^d)$. In this way, we have the benefit of capturing the full interdependence of the micro MABs and we can apply contextual bandit algorithms like ϵ-greedy with classification algorithms, as shown in Algorithm 18.

When applying ϵ-greedy, in each iteration, we have ϵ chance picking a random arm, and $1 - \epsilon$ chance, we find the one for which the context learner predicts the highest reward given the current context. This requires an *argmax* operation on L. While some differentiable models allow for an implementation of such argmax operation in an efficient way, in our experiments with a decision tree, we just iterate over all possible macro arms and obtain the one for which L reports the highest reward (this can be implemented more efficiently than full iteration over the complete combinatorial macro-arm space, by iterating over the set of leaves of the decision tree, for example).

In this section, we describe the experimental setup and report results. In order to evaluate the three methods described above, we perform experiments in μRTS. A collection of 6 maps of different starting configurations are selected as our different "contexts". The goal is to use proposed bandit algorithms to choose good parameterized stochastic game-playing policies for a new map they had not seen before. To test this, we train the learning models of our bandits in five of those maps, and then we test them in the sixth, unseen map in a leave-one-out setting.

5.2.4 Bandit Optimization of Gameplay Policy

We employ the discretized parameterization of the policies where each value, w_i, in $\mathbf{w}$ has six values $[0, 1, 2, 3, 4, 5]$. Thus, the optimization of the policy can be modeled as a combinatorial bandit where the macro MAB is all of the possible policies and each macro arm consists of six micro MABs with six micro arms to represent the discretized values. The reward are calculated from the game play results against our target bot called `RndBiased` bot, which is built-in intro μRTS. `RndBiased` is actually expressible in our space of policies. It is a biased random agent where HARVEST, RETURN, and ATTACK has five times the weights than other action types (approximate weight vector $[0.06, 0.06, 0.28, 0.28, 0.06, 0.28]$).

To test whether the optimized contextual combinatorial bandit can generalize to unseen contexts, we employ the *leave-one-out* cross validation technique, which means we select one of the maps as testing map, train the agent on the rest of the maps, and repeat the optimization for each map. Thus, the optimization cycle works like this:

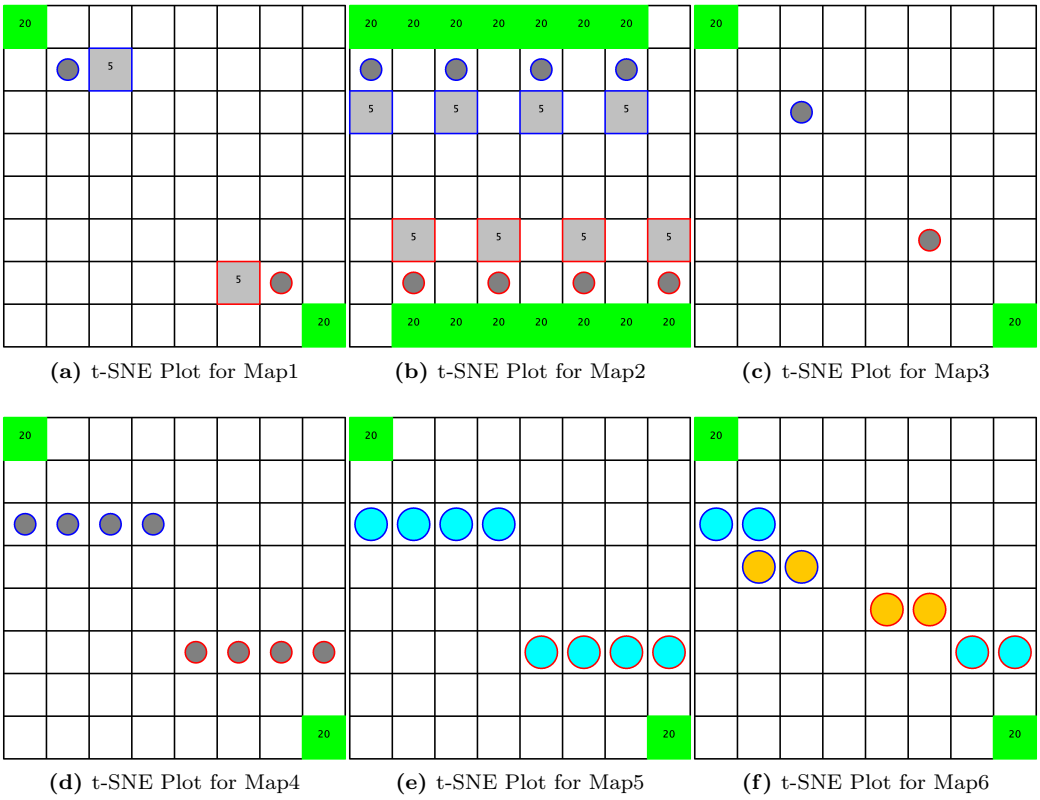
1. A random map from the training set is chosen and the corresponding context vector σ is revealed to the agent.

2. The policy picked by the bandit algorithm will play 10 games against the target bot in the chosen map.

3. The average winrate is recorded as the reward and revealed to the bandit algorithm.

This optimization process is repeated for 10000 iterations. After the optimization, the context vector of the testing map is revealed to the bandit. The selected testing policy will run 1000 games against the target bot in the testing map. And the average winrate is recorded as the final performance of the bandit algorithm for further comparison.

Additionally, we add two more baseline algorithms for comparison: naïve sampling (with global optimization)[7] and naïve sampling without global optimization. The two algorithms serve as ablation studies of the contextual bandits, because they capture the combinatorial structure but not the contextual information. Naïve sampling is a combinatorial multiarmed bandit strategy that is *not*



(a) t-SNE Plot for Map1 **(b)** t-SNE Plot for Map2 **(c)** t-SNE Plot for Map3

(d) t-SNE Plot for Map4 **(e)** t-SNE Plot for Map5 **(f)** t-SNE Plot for Map6

Figure 5.7: The six maps used in CCMAB experiments.

contextual, hence it will just find the best macro-arm in the set of training maps, and use that in the test map, without adapting the policy to the test map. Naïve sampling without global optimization is the same as Naïve sampling, but removing the *global MAB* (this is to have a direct non-contextual comparison to our Naïve Aggregation approach). We select six maps, listed below, as our set of context maps, as shown in Figure 5.7.

- Map 1: *8x8/basesWorkers8x8A.xml*: In this map of size 8 by 8, each player starts with one base and one worker. Games are cut-off at 3000 cycles.
- Map 2: *8x8/FourBasesWorkers8x8.xml*: In this map of size 8 by 8, each player starts with four bases and four worker. Games are cut-off at 3000 cycles.
- Map 3: *OneWorker8x8.xml*: In this map of size 8 by 8, each player starts with one worker and more resources. Games are cut-off at 3000 cycles.

- Map 4: *FourWorker8x8.xml*: In this map of size 8 by 8, each player starts with four workers. Games are cut-off at 3000 cycles.
- Map 5: *FourRanged8x8.xml*: In this map of size 8 by 8, each player starts with four ranged units. Games are cut-off at 3000 cycles.
- Map 6: *TwoRangedLight8x8.xml*: In this map of size 8 by 8, each player starts with two ranged units and two light units. Games are cut-off at 3000 cycles.

The context vector calculated for the maps is simple: a vector of the number of each type of units for either player. For example, Map 1 has the corresponding context vector $[1, 0, 1, 0, 0, 0]$.

5.2.5 Implementation Details

In this part, we describe the implementation details for the three algorithms. Overall, we use decision trees as the machine learning model as our policy learner. The reason is that during the bandit optimization, the dataset is often unbalanced, and decision trees are less sensitive to an unbalanced dataset. Also, the computational complexity of training and testing decision trees is low compared to more complicated methods. Specifically, we use the J48 model provided by Weka.

For naïve aggregation, we trained a J48 classifier for each micro arm. Thus, in total we have $K \cdot n = 36$ classifiers. We use ϵ-greedy to balance exploration and exploitation, which means for each iteration we have ϵ probability of selecting a random macro arm, and $1 - \epsilon$ probability of selecting the best macro arm. We use a linearly decaying ϵ in our experiments, starting at $\epsilon = 0.5$ and reaching $\epsilon = 0.00$ in the last optimization iteration. In naïve aggregation, the best macro arm is the combination of the micro arms with the best winrate separately. Thus, we perform a greedy selection of micro arms to construct the best macro arm.

For pair-wise aggregation, we have two parts. In the first part is identical to the naïve aggregation model. The second part we have the modeling for pair-wise relations. We train one J48 classifier for each pair of micro arms from different macro MABs. In total, we have $\frac{n(n-1)}{2} \cdot K^2 = 540$ classifiers. Exploration/exploitation is the same as naïve aggregation, the same reward will be revealed to the pair-wise learners. For naïve aggregation, greedy selection will give us the macro arm with the

highest average winrate. For pair-wise aggregation, we go over all combinations to find the macro arm with the highest average score. Once we have the naïve aggregation and the pair-wise score for each macro-arm, we calculate the average score for each macro-arm and select the one with the highest score. Again, notice that this requires iterating over all possible macro-arms (which is feasible in our experiments), although mode efficient ways to do this process will be devised in future work.

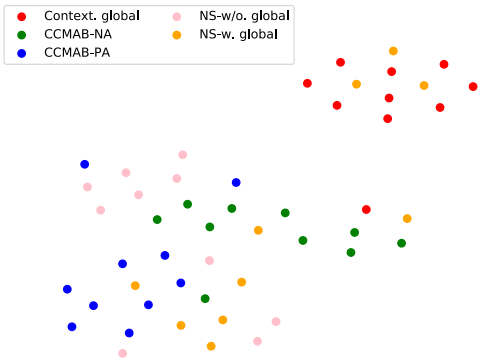
For contextual global optimization, we simply have one classifier that takes the context-macroarm vector as the input. Again, we use ϵ-greedy to balance exploration and exploitation, with linearly decaying ϵ starting at $\epsilon = 0.5$ and going down to zero. When we are making exploitation selections, the macro arm of the highest predicted winrate with respect to the current context is selected. The picked augmented macro arm and reward received together will be the training examples for the classifier.

5.2.6 Experimental Results

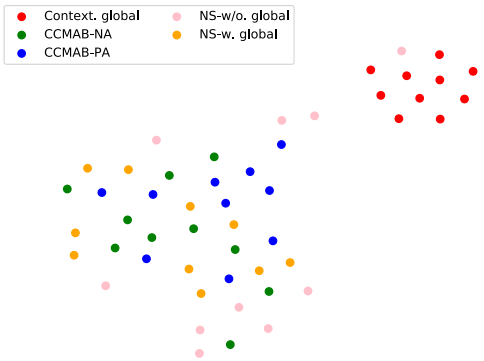
In our experiments we compare the policies resulting from the different bandit strategies using the cross-validated win rate against the target bot. As we can see in Table 5.1, the best performing method overall is global contextual optimization with 0.836 total winrate and being the best bot in four maps. The next is pair-wise aggregation strategy with 0.819 total winrate, followed by naïve aggregation with 0.812 winrate. We can see that considering the interdependence between the micro MABs are clearly helpful for the performance of the optimization, since pair-wise aggregations outperformed naïve aggregation, and global optimization outperformed pair-wise aggregation. Moreover, the contextual global optimization approach performed the best or close to the best in all maps except for Map 1 (we explain the reason for this below).

The two baseline methods that do not consider contextual information have the worst performance. Interestingly, Naïve Sampling without global optimization slightly outperformed Naïve Sampling with global optimization, with 0.806 and 0.798 winrate respectively, probably due to overfitting.

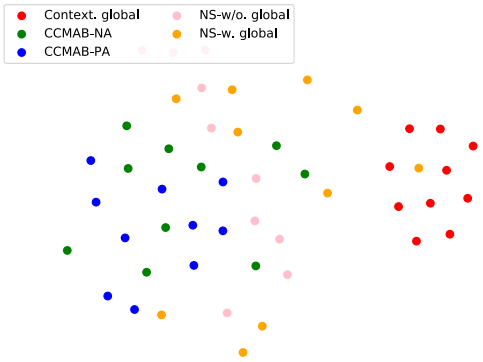
Then we used *t-SNE*[87] to visualize the resulting macro arms of the algorithms in the six maps,



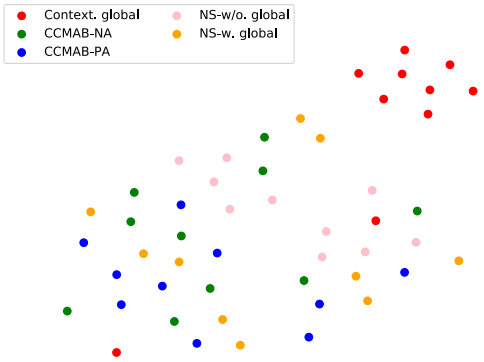
(a) t-SNE Plot for Map1



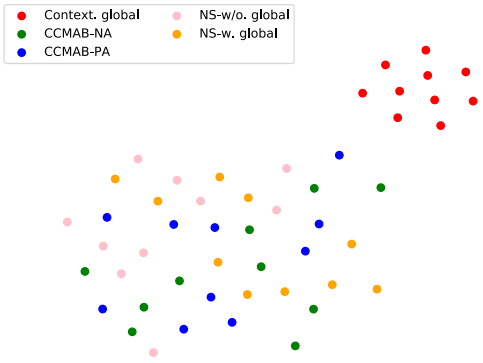
(b) t-SNE Plot for Map2



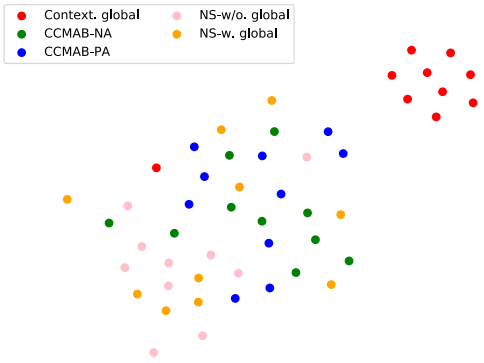
(c) t-SNE Plot for Map3



(d) t-SNE Plot for Map4



(e) t-SNE Plot for Map5



(f) t-SNE Plot for Map6

Figure 5.8: t-SNE visualizations of optimized policies in the six maps

Table 5.1: Cross-validated winrates for naïve aggregation, pair-wise aggregation, contextual global optimization, and two baseline algorithm based on naïve sampling.

	Naïve Agg.	Pair-wise Agg.	Contextal Global Opt.	NS w/o. Global Opt.	NS w. Global Opt.
Map 1	0.839	**0.886**	0.587	0.851	0.759
Map 2	0.528	0.515	**0.898**	0.487	0.524
Map 3	0.738	0.738	**0.747**	0.728	0.735
Map 4	0.922	0.922	**0.932**	0.910	0.914
Map 5	0.946	0.941	**0.949**	0.939	0.941
Map 6	0.903	0.914	0.907	**0.921**	0.915
Average	0.812	0.819	**0.836**	0.806	0.798

shown in Figure 5.8. Sub-figure (a) to (f) in Figure 5.8 represents the results in Map 1-6 respectively. And in each sub-figure, the results of each algorithm are labeled with a different color. There each algorithm is run 10 times, thus for each algorithm, there are 10 points representing results from separate runs. A clear phenomenon we can observe in this visualization is that the results of the global contextual optimization are quite different from the ones of other algorithms. In fact, contextual global optimization almost always selects the macro arm that is effectively equivalent to $[0, 0, 0, 0, 0, 1]$ across all of the maps.

The reason can be found from the visualization of one of the trees that were learned in our experiments, shown in Figure 5.9. In this decision tree, any policy with the first two elements being 0 and the last element greater than 0 will be classified to win with high conviction. Due to the order in which we search the macro-arms for selecting the best, this results in selecting the macro-arm mentioned above. Our interpretation is that the decision tree finds a solution that is performing well and generalizable in most of the maps that we use. And Map 1 is an example of this solution that fails to generalize. We believe this phenomenon occurs as the set of maps we used for experimentation are not varied enough for the model to learn generalizable policies, and context learners often learned that the $[0, 0, 0, 0, 0, 1]$ policy does well in all maps. Hence, this points out the need for a more varied training set. However, it is worth noting that it is interesting that such a simple strong policy exists in the space and Naïve Sampling did not find it. We believe this is because the machine learning model is able to perform generalizations such as ("arm $1 > 2$") that the probability estimation procedure done in Naïve Sampling cannot capture.

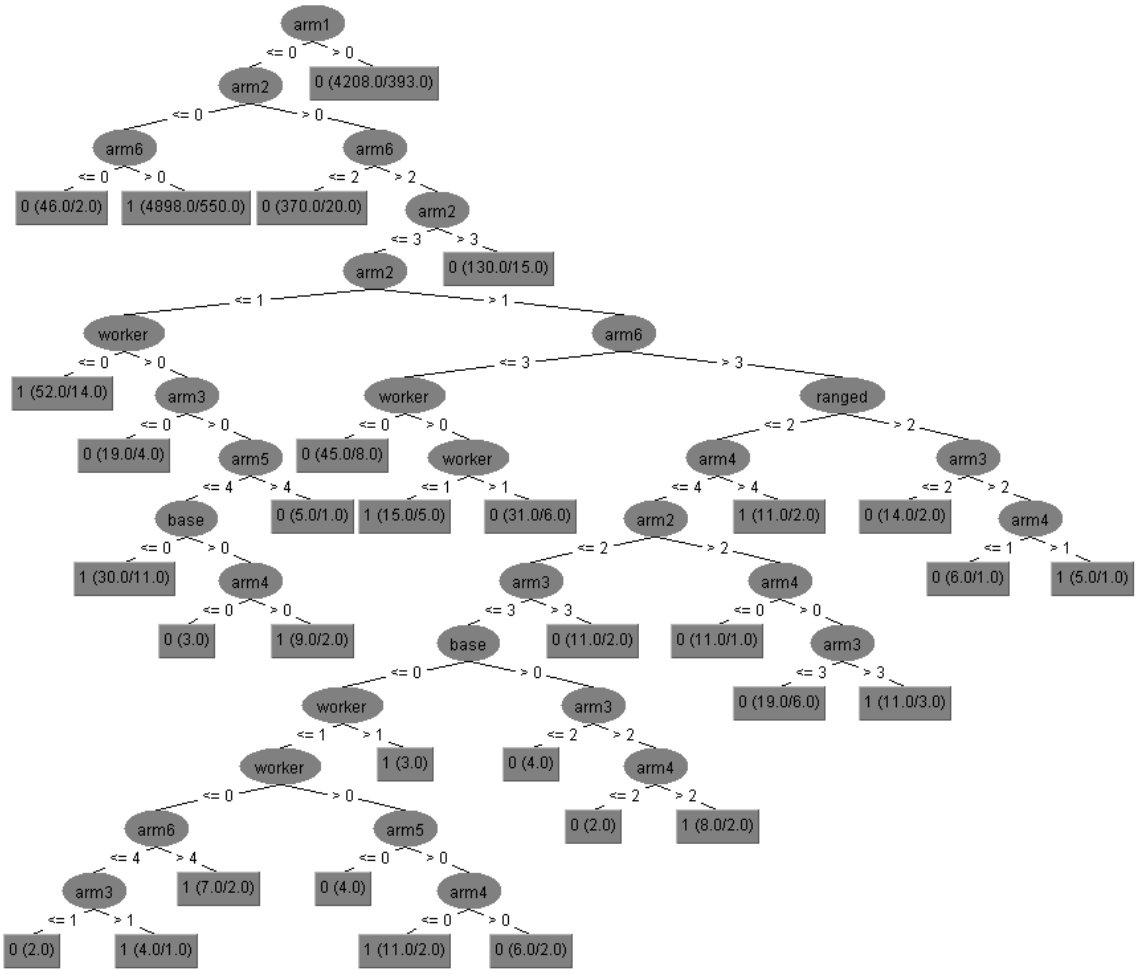


Figure 5.9: Example visualization of the decision tree trained from contextual global optimization.

5.3 Conclusion

In this chapter, we introduced two applications bandit-based policy optimization, tree/playout policy optimization and contextual combinatorial bandits.

Specifically, in the first part, we have studied policy optimization in several settings. First, we tried simulation balancing for playout policy optimization. We found that although it is better than the baselines as playout policy, its performance is not comparable to optimizing as playout policy directly. We also tried optimizing game policy, tree policy, and playout policy in three maps at the same time. We observed that for some action types like NONE and MOVE, the

weight distribution is in consensus for all maps, but for others, weight distribution is spread to multiple categories. This might be because certain weights are good for some maps. Furthermore, we compared the performance as tree policy between optimized gameplay policy and optimized tree policy, and confirmed that optimizing tree policy directly does help. Finally, we optimized the tree policy and playout policy jointly. The resulting pair of policies outperform the combination of the best of tree policy and playout policies, which suggests that the "match" of the tree policy and playout policy can also play an important role in the performance of the MCTS.

For future work, we want to further investigate the simulation balancing algorithm, since there has been some encouraging advance in gradient policy algorithms that might help scaling up SB. Also, the joint optimization of different component of the MCTS algorithm seemed to be beneficial. It will be interesting to take more factors, like the evaluation function tuning and exploration parameters, into the optimization process to see if we can push the progress further and gain insight into the interplay between the different pieces of MCTS.

In the seceond part, we have introduced the CCMAB problem, which corresponds to the bandit problem with contextual information and combinatorial arm structures. Then we proposed three strategies to tackle the CCMAB problem: naïve aggregation, pair-wise aggregation, and contextual global optimization. The three strategies consider no micro arm interdependence, partial micro arm interdependence, and full micro arm interdependence. Specifically, naïve aggregation only considers micro MABs separately and does greedy aggregation. Pair-wise aggregation adds the pair-wise relations of the micro MABs upon the naïve aggregation. Lastly, contextual global optimization considers the full interdependence by treating the combinatorial structure of arms as additional contexts.

Then we designed experiments in μRTS to compare the three strategies. Specifically, the task is to find the best map-specific game-playing policies whose parameterization has the combinatorial structures in unseen maps given a set of training maps. The result showed that the more interdependence that the strategy considers, the better performance it can obtain. Also, through the t-SNE visualization, we observed that the strategy that considers the full interdependence of the micro

arms generates policies that differ more from the ones that do not consider full interdependence or contextual information.

For future work, we believe there is still a large space for improvements for the CCMAB strategies. For example, the scalability of the algorithms can be crucial in more complex problems, as our implementations often rely on macro-arm enumeration. Also, for exploration/exploitation balancing, we simply used ϵ-greedy strategies. Further, we want to study the regret bounds of these algorithms, which we did not study in this work, where we limited ourselves to empirical experimentation. Finally, we want to apply our solution to the CCMAB problem to domains like player modeling, where MAB approaches have shown promise[88] and to Monte Carlo Tree Search, by selecting contextualized policies for either the tree policy or the default policy.

Chapter 6: Self-Learning From MCTS in RTS Games

In previous chapters, we have discussed work toward incorporating domain knowledge with MCTS in RTS games. More specifically, we have discussed learning a tree policy from replays, learning an evaluation function from replays, bandit-based tree/playout policy optimization, and contextual combinatorial bandits. In this chapter, we turn to a different related problem. Namely, that of self-learning from MCTS in the context of RTS games: can a search-based agent (based on MCTS) bootstrap itself, learning better tree or playout policies by learning from its own gameplay?

In Chapter 3.1, we have investigated and compared various of machine learning model that can be used to train a policy to extract knowledge from scripted bots and to bias the tree search. The work presented in this chapter builds upon that line of work. However, instead of learning from scripts that encode domain knowledge, we explore the problem of learning improving policies over iterations of self-play by learning from the agent's own game play. Previous studies of computer Go that combines MCTS and reinforcement learning have shown that we can generate knowledge that is beyond that the model learns from human knowledge[46]. Thus in this chapter, we construct a self-learning process using a simple machine learning model and study the effect that changing different parameters of MCTS have on the learning process. The remainder of the chapter is structured as follows: we first discuss the problem of learning policies directly from MCTS behavior in the context of the RTS games. Then we describe the experimental designs and results.

6.1 Learning from MCTS in RTS games

So far in this document we have explored various ways to obtain knowledge-encoded policies for MCTS in RTS games: from replays, from hard-coded scripts, and generating them from scratch via bandit optimization. There are many interesting problems to explore apart from work having been described in this document. One of them is learning from MCTS itself. Thus, as the last chapter of the dissertation, we investigate the problem of learning from MCTS via self-play.

Learning from MCTS can be achieved by simple behavior cloning from replays like described in InformedMCTS (as in Chapter 3.1). To take a step further, we can construct a feedback loop between MCTS and the machine learning model, where a policy is learned form MCTS, which is then in-turn used to generate new replays by using them inside MCTS, and so on. Specifically, we investigate machine learning model for learning from MCTS in RTS games to address the special problems in RTS games.

As we have established in the previous sections, evaluation speed of the machine learning model is crucial for applying machine learning models within MCTS in RTS games due to the limiting time budget. Since the training is done beforehand, it is the time required used to evaluate the model during the execution of MCTS that is important during gameplay. Thus, we investigate machine learning models with fast inference speed, such as decision trees, as the algorithms for self-learning in the experiments of this chapter.

Note that self-learning from MCTS is not new in the area of game AI. AlphaGo Zero and AlphaZero combine MCTS and reinforcement learning to construct self-learning in the game of Go, Shogi, and Chess, and have shown that this method has great performance in these games[82]. In this work, we use supervised learning instead of reinforcement learning as the learning scheme. Moreover, we focus on studying how different components of MCTS affects the performance of self-learning.

6.2 Self-Learning from MCTS via Supervised Learning

In this section, we describe our methods to learn a unit-action probability model to guide the tree policies from replays of self-play of MCTS agents. Given a game state s, a player p, and a unit u, the model returns the probability $P(a|p, s, u)$. Then the MCTS agent uses this model to bias the selection process of the tree policy. To ensure the simplicity and inference speed of the model, we use the decision tree model in this chapter to learn the unit-action probability model.

As shown in Algorithm 19, the learning process is initialized with the self-play of a standard NaïveMCTS agent M_0. The replays of the games played are recorded and used to train the first iteration of learning model M_1. To avoid the *Rock, Paper, Scissors* effect, where agents have cyclical win-lose relations and it is hard to determine which is the best one, we determine the best model

Algorithm 19: Policy Learning through Self-Play

Initialize the set of models $S = \{M_0\}$, with M_0 being the standard NaïveMCTS.
Run self-play for the initial S, and train model M_1 to learn from M_0. And add M_1 to S.
for $i = 1, 2, 3, \cdots, T$ **do**
    a. Run round-robin tournaments between MCTS agents that use models $\{M_0, \cdots, M_i\}$ to find the best model so far
    b. Train the model M_{i+1} to learn from the replays of the best model in the round-robin.
    c. Add M_{i+1} to S.

based on the overall winrate against all of the past models generated so far during the process. Then for a total number of T iterations, we first run a round-robin between models $\{M_0, \cdots, M_i\}$ to find the best model, and then we train a new model using the replays of the best bot in this round-robin experiments. We repeat the process after adding model M_{i+1} into the collection S of all the generated models. Specifically, we explore three different ways of applying the model in Monte Carlo Tree Search: using the model to bias the tree policy, using the model as the rollout policy, and using the model to bias the tree policy and as the rollout policy. We also try different depths for the rollouts to study its effects on the learning process.

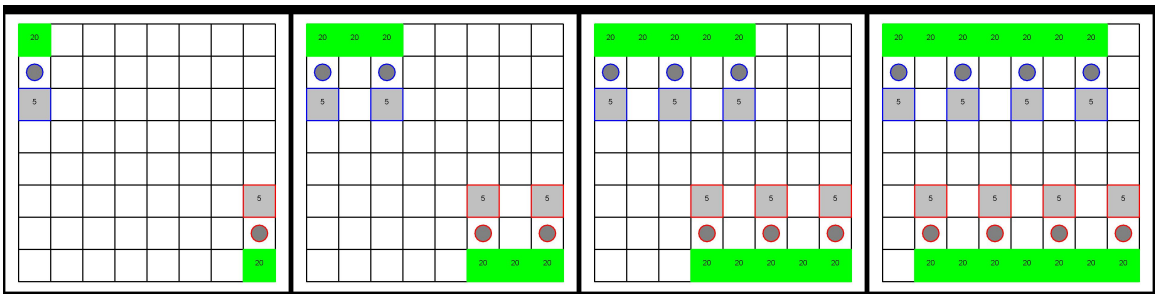
6.3 Experiments and Results

To empirically study the properties of self-learning, we design a few experiments to test the performance under different settings. The self-learning process consist of five iterations of agents. In each iteration, we run round-robin tournaments repeatedly for 25 times to obtain (1) winrate statistics and (2) replays of each agent. Then, we pick the agent that has the highest winrate overall in the current iteration, as the agent to learn from. To mitigate the variance in the experiment, we repeat the full self-learning process four times with four different maps shown in Figure 6.1. Thus, when calculate the aggregated winrate for each agent, there are statistics from $25 \times 4 = 100$ rounds of round-robin tournaments. Note that learning is only done after all round-robin tournaments for the current iteration are complete. To learn from the picked agent, we train a decision tree model to predict unit actions given a set of features representing the environment around the unit. The set of features we used is: the number of resources available to the player, the cardinal direction (north,

east, south, west) toward where most friendly units are, the cardinal direction toward where most enemy units are, whether we have a barracks or not, four features indicating the type of the unit in the cell two positions north, east, south or west (or whether these cells are empty or are a wall), and 24 features indicating the unit type and friendly or enemy status of the unit type in the five by five grid surrounding the unit. Thus the total number of features used is 32. And the label for each entry is one of the 69 all possible unit actions. The machine learning model we use is the J48 implementation in the Weka library. Also, as the number of iteration grows, the number of games played by each agent increases. Thus, to keep the size of the training dataset for each model at the same level, we resample the training dataset for each iteration of agents to 100,000 data points. To keep diversity of the configuration of the training maps, we selected four 8×8 maps, described as follows and shown in Figure 6.1.

- *OneBaseWorker8x8*:In this map of size 8 by 8, each player starts with one base and one worker. Games are cut-off at 3000 cycles.
- *TwoBaseWorker8x8*:In this map of size 8 by 8, each player starts with two bases and two workers. Games are cut-off at 3000 cycles.
- *ThreeBaseWorker8x8*:In this map of size 8 by 8, each player starts with three bases and three workers. Games are cut-off at 3000 cycles.
- *FourBaseWorker8x8*:In this map of size 8 by 8, each player starts with four bases and four workers. Games are cut-off at 3000 cycles.

The first experiment we run is using the learned model to bias the tree policy only. In this experiment, the J48 model is used as the probability distribution to bias the unit-level action sampling. In the first iteration, we run self-play games between two standard NaïveMCTS agents and collect the replay data. Then we train the model using the replay data to generate our first generation of learned agent by combining this model with a NaïveMCTS agent. The generated agent is added to the pool of agents together with the standard NaïveMCTS agent. In the following four iterations, we first run a round-robin tournaments within the pool of agents. Then we create and resample a



(a) **(b)** **(c)** **(d)**

Figure 6.1: Four maps used in the experiments

100,000 training data from the replays of the best performing agent in the round-robin. Lastly, we add the NaïveMCTS agent combined with the new model into the pool. After five iterations of model is generated, we run another round-robin tournaments to test the final performance of all agents, including the standard NaïveMCTS agent, using the metric of winrate. Due to the stochastic nature of the experiments we repeat the process four times and take the average winrate as the final results (however we observe that the results do not fluctuate much from trial to trial). Also, to study the effect of the depth of the rollouts, we repeat the experiments three times with the rollout depth of the MCTS agent being 25, 50, and 100. As shown in Figure 6.2, we plot the winrate statistics of the round-robin tournaments collected from the last iteration of self-play, grouped by the experiments ran with three different rollout out depth. In each group of bars, there are six bars representing the winrate of each bot. The bar labeled as "Iteration 0" stands for the performance of the vanilla MCTS (without any learned model) and the rest stand for the performance of MCTS agent using the model trained in each iteration. We can observe that the learning agents are able to learn to beat the baseline agent. But the learning saturates very soon (right after the first iteration). By looking at the performance of the standard NaïveMCTS agent, we can see that the winrate of the baseline agent decreases as the rollout depth increases.

The second set of experiments we run is the same as the first one, except that instead of using the model to bias the tree policy, we use it directly as the rollout policy. The result us shown in Figure 6.3. We find in this setting, the model does not improve over the five iterations under all

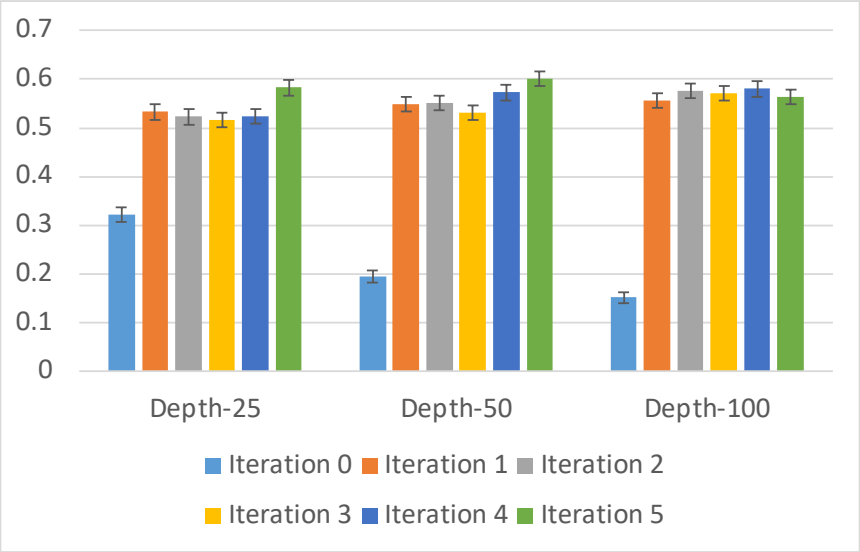


Figure 6.2: Winrate comparisons with model used to bias tree policy under different rollout depths.

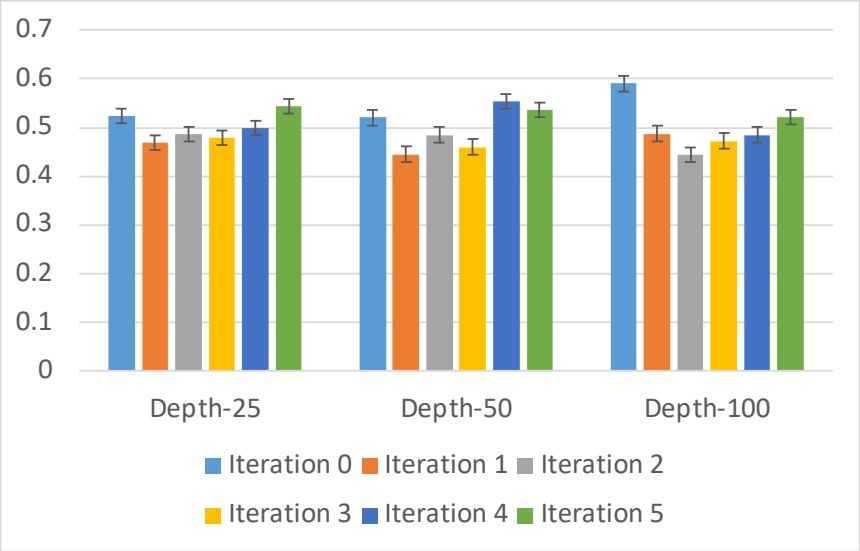


Figure 6.3: Winrate comparisons with model used as the rollout policy under different rollout depths.

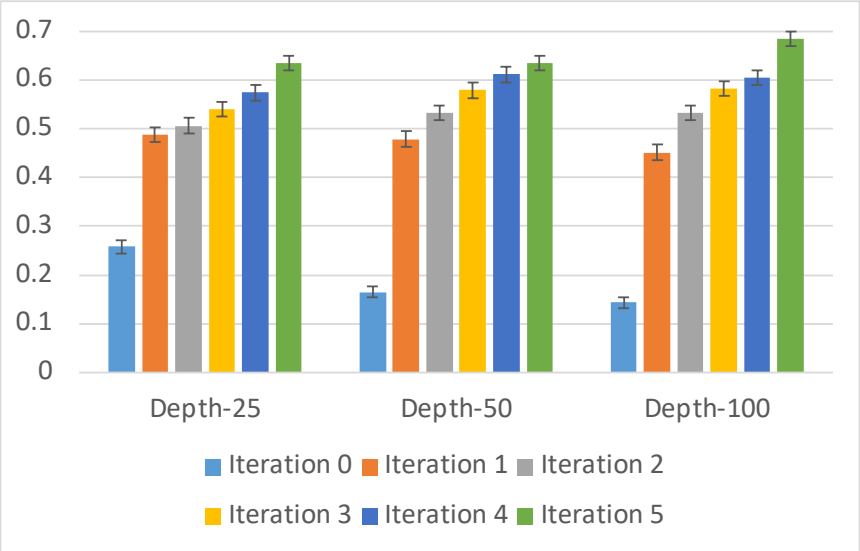


Figure 6.4: Winrate comparisons with model used to bias tree policy and as rollout policy under different rollout depths.

three rollout depths.

The final set of experiments we run is that we use the model both to bias the tree policy and as the rollout policy. The result us shown in Figure 6.4. By applying the model as both component, we find that the agents constantly improve over the five iterations in all three rollout depths. Also, if we compare Figure 6.3 and Figure 6.4, we can observe that the winrate of the baseline agent is lower in the setting that the model applied to both components, indicating that the agents trained in this way has a better gameplay strength.

6.4 Conclusion and Discussion

In this chapter, we have investigated the problem self-learning from MCTS in the domain of RTS games. In the long term, we would like to learn more about the dynamics of how different parameter of MCTS and learning schedules affect the performance of self-learning from MCTS in RTS games. And in this chapter, we looked at how applying the learned policies in different parts of MCTS (tree policy, rollout policy) affects the self-learning process together with the effects of the rollout depth.

Specifically, we used a simple machine learning model to learn probabilistic gameplay policies from MCTS. Then we empirically compared the performance of the self-learning process via several

approaches of applying the model to MCTS, including biasing the tree policy, as rollout policy, and both. We found that using the model only to bias the tree policy promotes the learning while using the model only as the rollout policy does not. And the most effective way of applying the model is to use the model to bias the tree policy and as the rollout policy at the same time. In the experiments, we employ round-robin tournaments in each iteration, where the number of games needed to be run grows quadratically as the number of agent grows (although, technically, only the games involving the new agent for the current iteration need to be run, and the other games can be reused from previous iterations). And this is part of the reason why we only run five iterations of self-play. For the scale of our experiments, five iterations of robin-robin tournaments are sufficient for showing learning trends. In practice, a full experiment of self-play with five iterations takes between four to six hours depending on the map used and the playout depth of the experiment on a single machine with 12 cores and 32G of RAM. For future work of larger scale, a match-making scheme that runs faster with respect to the number of agents should be employed. For example, AlphaStar[89] uses a variation of fictitious self-play, which in each iteration of the training of new agent, the new agent is trained against the mixture of all previous agents. This leads to many open problems as well, such as, whether the supervised learning models can handle training examples collected from a mixture of different agents.

For future work, there are several interesting lines of work to pursue. First, in this work we used the default exploration factors for exploration/exploitation balancing in MCTS. It would be interesting to see how different exploration factors play out in the self-learning process. In other words, is it more effective to do more exploration or to do more exploitation? Another interesting problem is to integrate enhancement techniques of MCTS to see if they promotes the efficiency of the self-learning. For example, rapid action value estimations (RAVE) is a very useful technique in computer Go. It is important to see if these techniques can be apply to a different type of games and be helpful in a different learning problem.

Chapter 7: Conclusions

In this chapter, we present a summary of the work and contributions in this dissertation and a discussion of potential lines of future work.

7.1 Summary

In summary, this dissertation explored different types of domain knowledge and how they can be integrated into MCTS in the domain of RTS games. Specifically, we first tried learning gameplay policy and game state evaluation function from the replays of scripted bots, and using them to bias the tree policy and evaluating the game states, respectively. Apart from learning models from game replays, we also tried using the scripted bot directly to guide MCTS to search near the action space proposed by the scripts in the expansion and selection phase. The next problem we investigated is that whether strong game policy makes good rollout policy for MCTS. We constructed a simple parameterization for gameplay policy and optimized for different objectives. And we found that strong rollout policies are not necessarily strong gameplay policy. Finally, we studied the problem of self-learning from MCTS. We learn a model of gameplay policy from the replays and applied the model to different components of MCTS to study its effects on the performance of the self-learning.

7.2 Contributions

This dissertation presents the following contributions:

- We proposed a convolutional neural network that can take varied map size as input and studied the feasibility of learning evaluation function in small maps and transfer the knowledge to larger maps.
- We surveyed the problem of learning tree policy bias models using a collection of machine learning models with varied model complexity. Compared the models in terms of classification performance and performance of biasing MCTS under iteration/time budget.

- We proposed the family of Guided Naïve Sampling (GNS), which can leverage the scripted bots to guide the MCTS directly. GNS significantly improved the performance of MCTS in large maps, exceeding the performance of MCTS and the guidance script separately. However, using the scripts as the rollout policy has very unstable performance.

- We proposed a simple parameterization of gameplay policies and used a bandit-based optimization method to optimized for two objectives: performance as gameplay policy and performance as rollout policy. We observe that optimizing for gameplay strength leads to good performance as the bias for tree policy but no necessarily good as the playout policy. And optimizing for rollout policy strength, although works bad as gameplay policy, leads to the best performance as rollout policy.

- We study the contextual combinatorial bandit problem and proposed three strategies of its optimization: naïve aggregation, pair-wise aggregation, and contextual global optimization.

- We proposed a framework for self-learning from MCTS via supervised learning algorithm. Under this framework, we studied the effects of applying the model as the bias for tree policy, as the rollout policy, and applying both together. We observe that the self-learning process has the best performance with the model applied as both the bias for tree policy and as the rollout policy.

7.3 Future Work

This dissertation opened a spectrum of next steps could be taken in the field of applying MCTS in RTS games and beyond. Future work includes:

- Considering games where the player controls many units simultaneously, modeling of the interdependence of the unit actions in policy learning processes can be a very useful and important problem to pursue (all our learned policies in this work chose actions independently for each unit, and it was left to MCTS to integrate them).

- Contextual bandits is another idea to consider for integration into the current MCTS algo-

rithms for RTS games. We have explored using contextual bandits to optimize for adaptive policies. It is interesting to see if it can be applied to the MCTS algorithm to improve the scalability of MCTS in RTS games.

- There are still many properties of the RTS games we did not cover in this dissertation. For example, partial observability (in the form of fog-of-war) is an important property in commercial RTS games. We have alread had some initial results in the direction of combining scripted bots and MCTS under partial observability. Our bot MentalSealPO competed in the μRTS competition in 2020 and 2021 and achieved the first place in the partial observable track. Note that this bot depends on the scripted for the exploration of the map. The potential future work in this direction is to try to involve the search algorithm to solve the partial observability problem.

Bibliography

[1] Murray Campbell, A Joseph Hoane Jr, and Feng-hsiung Hsu. Deep blue. *Artificial intelligence*, 134(1-2):57–83, 2002.

[2] Jay Burmeister and Janet Wiles. The challenge of go as a domain for ai research: a comparison between go and chess. In *Proceedings of Third Australian and New Zealand Conference on Intelligent Information Systems. ANZIIS-95*, pages 181–186. IEEE, 1995.

[3] Rémi Coulom. Efficient selectivity and backup operators in monte-carlo tree search. In *International conference on computers and games*, pages 72–83. Springer, 2006.

[4] Santiago Ontañón. The combinatorial multi-armed bandit problem and its application to real-time strategy games. In *Ninth Artificial Intelligence and Interactive Digital Entertainment Conference*, 2013.

[5] Alexander Shleyfman, Antonín Komenda, and Carmel Domshlak. On combinatorial actions and cmabs with linear side information. In *ECAI*, pages 825–830, 2014.

[6] Peter Auer, Nicolo Cesa-Bianchi, and Paul Fischer. Finite-time analysis of the multiarmed bandit problem. *Machine learning*, 47(2-3):235–256, 2002.

[7] Santiago Ontanón. Combinatorial multi-armed bandits for real-time strategy games. *Journal of Artificial Intelligence Research*, 58:665–702, 2017.

[8] Santiago Ontanón, Gabriel Synnaeve, Alberto Uriarte, Florian Richoux, David Churchill, and Mike Preuss. A survey of real-time strategy game AI research and competition in starcraft. *IEEE Transactions on Computational Intelligence and AI in games*, 5(4):293–311, 2013.

[9] Michael Buro. ORTS: A hack-free RTS game environment. In *International Conference on Computers and Games*, pages 280–291. Springer, 2002.

[10] Gabriel Synnaeve, Nantas Nardelli, Alex Auvolat, Soumith Chintala, Timothée Lacroix, Zeming Lin, Florian Richoux, and Nicolas Usunier. TorchCraft: a library for machine learning research on real-time strategy games. *arXiv preprint arXiv:1611.00625*, 2016.

[11] Oriol Vinyals, Timo Ewalds, Sergey Bartunov, Petko Georgiev, Alexander Sasha Vezhnevets, Michelle Yeo, Alireza Makhzani, Heinrich Küttler, John Agapiou, Julian Schrittwieser, et al. StarCraft II: a new challenge for reinforcement learning. *arXiv preprint arXiv:1708.04782*, 2017.

[12] Yuandong Tian, Qucheng Gong, Wenling Shang, Yuxin Wu, and C. Lawrence Zitnick. ELF: An extensive, lightweight and flexible research platform for real-time strategy games. *Advances in Neural Information Processing Systems (NIPS)*, 2017.

[13] Per-Arne Andersen, Morten Goodwin, and Ole-Christoffer Granmo. Deep RTS: a game environment for deep reinforcement learning in real-time strategy games. In *2018 IEEE Conference on Computational Intelligence and Games (CIG)*, pages 1–8. IEEE, 2018.

[14] Herbert Robbins. Some aspects of the sequential design of experiments. *Bulletin of the American Mathematical Society*, 58(5):527–535, 1952.

[15] William R Thompson. On the likelihood that one unknown probability exceeds another in view of the evidence of two samples. *Biometrika*, 25(3/4):285–294, 1933.

[16] Yi Gai, Bhaskar Krishnamachari, and Rahul Jain. Combinatorial network optimization with unknown variables: Multi-armed bandits with linear rewards and individual observations. *IEEE/ACM Transactions on Networking*, 20(5):1466–1478, 2012.

[17] Djallel Bouneffouf. Online learning with corrupted context: Corrupted contextual bandits. *arXiv preprint arXiv:2006.15194*, 2020.

[18] Eric M Schwartz, Eric T Bradlow, and Peter S Fader. Customer acquisition via display advertising using multi-armed bandit experiments. *Marketing Science*, 36(4):500–522, 2017.

[19] Tze Leung Lai, Herbert Robbins, et al. Asymptotically efficient adaptive allocation rules. *Advances in applied mathematics*, 6(1):4–22, 1985.

[20] Joannes Vermorel and Mehryar Mohri. Multi-armed bandit algorithms and empirical evaluation. In *European conference on machine learning*, pages 437–448. Springer, 2005.

[21] Peter Auer. Using confidence bounds for exploitation-exploration trade-offs. *Journal of Machine Learning Research*, 3(Nov):397–422, 2002.

[22] Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. In *European conference on machine learning*, pages 282–293. Springer, 2006.

[23] Peter Auer, Nicolo Cesa-Bianchi, Yoav Freund, and Robert E Schapire. The nonstochastic multiarmed bandit problem. *SIAM journal on computing*, 32(1):48–77, 2002.

[24] Yoav Freund and Robert E Schapire. A decision-theoretic generalization of on-line learning and an application to boosting. *Journal of computer and system sciences*, 55(1):119–139, 1997.

[25] Lihong Li, Wei Chu, John Langford, and Robert E Schapire. A contextual-bandit approach to personalized news article recommendation. In *Proceedings of the 19th international conference on World wide web*, pages 661–670, 2010.

[26] Wei Chen, Yajun Wang, and Yang Yuan. Combinatorial multi-armed bandit: General framework and applications. In *International Conference on Machine Learning*, pages 151–159. PMLR, 2013.

[27] Victor Gabillon, Mohammad Ghavamzadeh, and Alessandro Lazaric. Best arm identification: A unified approach to fixed budget and fixed confidence. In *NIPS-Twenty-Sixth Annual Conference on Neural Information Processing Systems*, 2012.

[28] Zohar Karnin, Tomer Koren, and Oren Somekh. Almost optimal exploration in multi-armed bandits. In *International Conference on Machine Learning*, pages 1238–1246. PMLR, 2013.

[29] Bruce Abramson. Expected-outcome: A general model of static evaluation. *IEEE transactions on pattern analysis and machine intelligence*, 12(2):182–193, 1990.

[30] Matthew L Ginsberg. Gib: Imperfect information in a computationally challenging game. *Journal of Artificial Intelligence Research*, 14:303–358, 2001.

[31] Brian Sheppard. World-championship-caliber scrabble. *Artificial Intelligence*, 134(1-2):241–275, 2002.

[32] Cameron B Browne, Edward Powley, Daniel Whitehouse, Simon M Lucas, Peter I Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. A survey of Monte Carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in games*, 4(1):1–43, 2012.

[33] Sylvain Gelly and Yizao Wang. Exploration exploitation in go: UCT for monte-carlo go. In *NIPS: Neural Information Processing Systems Conference On-line trading of Exploration and Exploitation Workshop*, 2006.

[34] Philip Hingston and Martin Masek. Experiments with Monte Carlo Othello. In *2007 IEEE Congress on Evolutionary Computation*, pages 4059–4064. IEEE, 2007.

[35] David Robles, Philipp Rohlfshagen, and Simon M Lucas. Learning non-random moves for playing Othello: Improving Monte Carlo tree search. In *2011 IEEE Conference on Computational Intelligence and Games (CIG'11)*, pages 305–312. IEEE, 2011.

[36] Olivier Teytaud and Sébastien Flory. Upper confidence trees with short term partial information. In *European Conference on the Applications of Evolutionary Computation*, pages 153–162. Springer, 2011.

[37] Gerald Tesauro, VT Rajan, and Richard Segal. Bayesian inference in monte-carlo tree search. *arXiv preprint arXiv:1203.3519*, 2012.

[38] Guillaume M JB Chaslot, Mark HM Winands, H Jaap Van Den Herik, Jos WHM Uiterwijk, and Bruno Bouzy. Progressive strategies for monte-carlo tree search. *New Mathematics and Natural Computation*, 4(03):343–357, 2008.

[39] Rémi Coulom. Computing "ELO ratings" of move patterns in the game of go. *ICGA journal*, 30(4):198–208, 2007.

[40] Fabien Teytaud and Olivier Teytaud. Creating an upper-confidence-tree program for havannah. In *Advances in Computer Games*, pages 65–74. Springer, 2009.

[41] Chang-Shing Lee, Mei-Hui Wang, Guillaume Chaslot, Jean-Baptiste Hoock, Arpad Rimmel, Olivier Teytaud, Shang-Rong Tsai, Shun-Chin Hsu, and Tzung-Pei Hong. The computational intelligence of mogo revealed in taiwan's computer go tournaments. *IEEE Transactions on Computational Intelligence and AI in games*, 1(1):73–89, 2009.

[42] Broderick Arneson, Ryan B Hayward, and Philip Henderson. Monte Carlo tree search in hex. *IEEE Transactions on Computational Intelligence and AI in Games*, 2(4):251–258, 2010.

[43] Suoju He, Yi Wang, Fan Xie, Jin Meng, Hongtao Chen, Sai Luo, Zhiqing Liu, and Qiliang Zhu. Game player strategy pattern recognition and how UCT algorithms apply pre-knowledge of player's strategy to improve opponent ai. In *2008 International Conference on Computational Intelligence for Modelling Control & Automation*, pages 1177–1181. IEEE, 2008.

[44] Jing Huang, Zhiqing Liu, Benjie Lu, and Feng Xiao. Pruning in UCT algorithm. In *2010 International Conference on Technologies and Applications of Artificial Intelligence*, pages 177–181. IEEE, 2010.

[45] Thomas Anthony, Zheng Tian, and David Barber. Thinking fast and slow with deep learning and tree search. In *Advances in Neural Information Processing Systems*, pages 5360–5370, 2017.

[46] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. Mastering the game of Go without human knowledge. *Nature*, 550(7676):354, 2017.

[47] Santiago Ontanón. Informed Monte Carlo tree search for real-time strategy games. In *2016 IEEE Conference on Computational Intelligence and Games (CIG)*, pages 1–8. IEEE, 2016.

[48] David Churchill and Michael Buro. Portfolio greedy search and simulation for large-scale combat in StarCraft. In *2013 IEEE Conference on Computational Inteligence in Games (CIG)*, pages 1–8. IEEE, 2013.

[49] Rubens O Moraes and Levi HS Lelis. Asymmetric action abstractions for multi-unit control in adversarial real-time games. In *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[50] Nicolas Arturo Barriga, Marius Stanescu, and Michael Buro. Puppet search: Enhancing scripted behavior by look-ahead search with applications to real-time strategy games. In *Eleventh Artificial Intelligence and Interactive Digital Entertainment Conference*, 2015.

[51] Nicolas A Barriga, Marius Stanescu, and Michael Buro. Game tree search based on nondeterministic action scripts in real-time strategy games. *IEEE Transactions on Games*, 10(1):69–77, 2017.

[52] Bhaskara Marthi, Stuart Russell, and David Latham. Writing stratagus-playing agents in concurrent ALisp. *Reasoning, Representation, and Learning in Computer Games*, page 67, 2005.

[53] Niels Justesen, Bálint Tillman, Julian Togelius, and Sebastian Risi. Script-and cluster-based UCT for StarCraft. In *2014 IEEE Conference on Computational Intelligence and Games*, pages 1–8. IEEE, 2014.

[54] Rubens O Moraes, Julian RH Marino, Levi HS Lelis, and Mario A Nascimento. Action abstractions for combinatorial multi-armed bandit tree search. In *Fourteenth Artificial Intelligence and Interactive Digital Entertainment Conference*, 2018.

[55] Christopher D Rosin. Multi-armed bandits with episode context. *Annals of Mathematics and Artificial Intelligence*, 61(3):203–230, 2011.

[56] David Silver, Richard S Sutton, and Martin Müller. Temporal-difference search in computer go. *Machine learning*, 87(2):183–219, 2012.

[57] Geoffrey I Webb, Janice R Boughton, and Zhihai Wang. Not so naive Bayes: aggregating one-dependence estimators. *Machine learning*, 58(1):5–24, 2005.

[58] Mehran Sahami. Learning limited dependence Bayesian classifiers. In *KDD*, volume 96, pages 335–338, 1996.

[59] J Ross Quinlan. *C4.5: programs for machine learning*. Elsevier, 2014.

[60] Jiawei Han, Jian Pei, and Micheline Kamber. *Data mining: concepts and techniques*. Elsevier, 2011.

[61] Leo Breiman. Bagging predictors. *Machine learning*, 24(2):123–140, 1996.

[62] Andy Liaw, Matthew Wiener, et al. Classification and regression by randomforest. *R news*, 2 (3):18–22, 2002.

[63] Jason Yosinski, Jeff Clune, Yoshua Bengio, and Hod Lipson. How transferable are features in deep neural networks? In *Advances in neural information processing systems*, pages 3320–3328, 2014.

[64] Radha-Krishna Balla and Alan Fern. UCT for tactical assault planning in real-time strategy games. In *Twenty-First International Joint Conference on Artificial Intelligence*, 2009.

[65] Marius Stanescu, Nicolas A Barriga, Andy Hess, and Michael Buro. Evaluating real-time strategy game states using convolutional neural networks. In *Computational Intelligence and Games (CIG), 2016 IEEE Conference on*, pages 1–7. IEEE, 2016.

[66] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[67] Min Lin, Qiang Chen, and Shuicheng Yan. Network in network. *arXiv preprint arXiv:1312.4400*, 2013.

[68] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, pages 249–256, 2010.

[69] Diederik P Kingma and Jimmy Ba. ADAM: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[70] Marius Stanescu, Nicolas Barriga, and Michael Buro. Using Lanchester attrition laws for combat prediction in StarCraft. In *Eleventh Annual AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE)*, pages 86–92, 2015.

[71] Alexander Kovarsky and Michael Buro. Heuristic search applied to abstract combat games. In *Conference of the Canadian Society for Computational Studies of Intelligence*, pages 66–78. Springer, 2005.

[72] Wei Chu, Lihong Li, Lev Reyzin, and Robert Schapire. Contextual bandits with linear payoff functions. In *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, pages 208–214, 2011.

[73] Yusaku Mandai and Tomoyuki Kaneko. LinUCB applied to Monte Carlo tree search. *Theoretical Computer Science*, 644:114–126, 2016.

[74] Guillaume MJ-B Chaslot, Mark HM Winands, and H Jaap van Den Herik. Parallel monte-carlo tree search. In *International Conference on Computers and Games*, pages 60–71. Springer, 2008.

[75] Santiago Ontañón, Nicolas A Barriga, Cleyton R Silva, Rubens O Moraes, and Levi HS Lelis. The first MicroRTS artificial intelligence competition. *AI Magazine*, 39(1), 2018.

[76] David Silver and Gerald Tesauro. Monte-carlo simulation balancing. In *Proceedings of the 26th Annual International Conference on Machine Learning*, pages 945–952, 2009.

[77] Sylvain Gelly—Yizao Wang—Rémi Munos and Olivier Teytaud. Modification of UCT with patterns in monte-carlo go. *Technical Report RR-6062*, 32:30–56, 2006.

[78] Sylvain Gelly and David Silver. Combining online and offline knowledge in UCT. In *Proceedings of the 24th international conference on Machine learning*, pages 273–280, 2007.

[79] Shih-Chieh Huang, Rémi Coulom, and Shun-Shii Lin. Monte-carlo simulation balancing in practice. In *International Conference on Computers and Games*, pages 81–92. Springer, 2010.

[80] Tobias Graf and Marco Platzner. Monte-carlo simulation balancing revisited. In *2016 IEEE Conference on Computational Intelligence and Games (CIG)*, pages 1–7. IEEE, 2016.

[81] Hendrik Baier and Peter D Drake. The power of forgetting: Improving the last-good-reply policy in Monte Carlo go. *IEEE Transactions on Computational Intelligence and AI in Games*, 2(4):303–309, 2010.

[82] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. A general reinforcement learning algorithm that masters Chess, Shogi, and Go through self-play. *Science*, 362(6419):1140–1144, 2018.

[83] Nicolo Cesa-Bianchi and Gábor Lugosi. *Prediction, learning, and games*. Cambridge university press, 2006.

[84] Aleksandrs Slivkins. Introduction to multi-armed bandits. *arXiv preprint arXiv:1904.07272*, 2019.

[85] Zuozhi Yang and Santiago Ontañón. Are strong policies also good playout policies? playout policy optimization for rts games. In *Sixteenth Annual AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE)*, 2020.

[86] Lixing Chen, Jie Xu, and Zhuo Lu. Contextual combinatorial multi-armed bandits with volatile arms and submodular reward. *Advances in Neural Information Processing Systems*, 31:3247–3256, 2018.

[87] Laurens van der Maaten and Geoffrey Hinton. Visualizing data using t-SNE. *Journal of Machine Learning Research*, 9:2579–2605, 2008. URL `http://www.jmlr.org/papers/v9/vandermaaten08a.html`.

[88] Robert C Gray, Jichen Zhu, Danielle Arigo, Evan Forman, and Santiago Ontañón. Player modeling via multi-armed bandits. In *International Conference on the Foundations of Digital Games*, pages 1–8, 2020.

[89] Oriol Vinyals, Igor Babuschkin, Wojciech M Czarnecki, Michaël Mathieu, Andrew Dudzik, Junyoung Chung, David H Choi, Richard Powell, Timo Ewalds, Petko Georgiev, et al. Grandmaster level in starcraft ii using multi-agent reinforcement learning. *Nature*, pages 1–5, 2019.