PONTIFICAL CATHOLIC UNIVERSITY OF RIO GRANDE DO SUL
SCHOOL OF TECHNOLOGY
COMPUTER SCIENCE GRADUATE PROGRAM

GENERAL GAME PLAYING WITH LEARNED RULES

GABRIEL DE ARRUDA RUBIN DE LIMA

Thesis submitted to the Pontifical Catholic University of Rio Grande do Sul in partial fulfillment of the requirements for the degree of Master in Computer Science.

Advisor: Prof. Felipe Rech Meneguzzi

Porto Alegre
2020

REPLACE THIS PAGE WITH
THE LIBRARY CATALOG
PAGE

REPLACE THIS PAGE WITH THE COMMITTEE FORMS

“The limits of my language mean the limits of my world.”
(Ludwig Wittgenstein)

GENERAL GAME PLAYING COM REGRAS APRENDIDAS

RESUMO

Pesquisas recentes sobre Aprendizagem por Reforço se concentraram em treinar agentes para jogos com a mínima supervisão humana (idealmente nenhuma). As abordagens *estado da arte* atuais podem aprender estratégias competitivas que superam as implementações anteriores e jogadores humanos profissionais em uma variedade de jogos de tabuleiro. Essas abordagens dependem que o agente saiba as regras do jogo *a priori*, através de um simulador de jogo, para antecipar o resultado das jogadas do agente durante seu treinamento. No entanto, realizar a amostragem a partir de um simulador pode ser impossível ou indesejável, nesse caso o agente deve resolver dois problemas, o de aprender a dinâmica do jogo e o de aprender uma estratégia a partir do modelo aprendido, que provavelmente possui ruído. Para lidar com essa limitação, introduzimos uma abordagem que combina random network distillation (RND) e Monte-Carlo Tree Search (MCTS) para melhorar o desempenho em jogos de um agente que utiliza amostras geradas por um modelo aprendido, quando realizar a amostragem a partir de um simulador é impossível ou indesejável.

Palavras-Chave: General Game Playing, Aproximação de Modelos, Monte Carlo Tree Search, Aprendizado por Reforço, Expert Iteration.

GENERAL GAME PLAYING WITH LEARNED RULES

ABSTRACT

Recent research on Reinforcement Learning has focused on training game-playing agents with minimal human supervision (ideally none). Current state-of-the-art approaches can learn competitive strategies that outperform those of previous implementations and professional human players in a variety of board games. Such approaches rely on providing the rules of the games *a priori* through a game simulator to anticipate the outcomes of the agent moves during training. However, sampling from a simulator might be impossible or undesirable so the agent has the dual problem of learning the game dynamics and learning the game strategy from the likely noisy learned model. In order to overcome such limitation, we introduce an approach that combines random network distillation (RND) and Monte-Carlo Tree Search (MCTS) to improve an agent's game-playing performance when sampling from a learned model, where sampling from a simulator is either impossible or undesirable.

Keywords: General Game Playing, Model Approximation, Monte Carlo Tree Search, Reinforcement Learning, Expert Iteration.

CONTENTS

1. INTRODUCTION

Much research has focused on developing agents able to compete with human players in a variety of games. Since the victory of the Deep Blue agent over Garry Kasparov in the game of Chess [Campbell et al., 2002], agents are surpassing human-level performance in a variety of games, such as: Go [Silver et al., 2016], Hex [Anthony et al., 2017], and Poker [Moravcík et al., 2017]. However, most of these agents were developed specifically for the games they play, so we cannot reuse the agent for two different games without making several modifications to its implementation [Finnsson and Björnsson, 2008]. Moreover, most expert agents are made with game-specific knowledge in order to achieve its gameplay performance, i.e. its performance depends on this expert knowledge. Agents implement game-specific knowledge in many forms: the description of the game rules [Finnsson and Björnsson, 2008], bias in the action selection with game-specific heuristics [de Lima et al., 2017], a symbolic model of the states [Silver et al., 2016], or the actual simulator to play games with self-play [Anthony et al., 2017].

In order to deal with such limitation, much effort has been made towards General Game Playing (GGP) agents [Genesereth et al., 2005]. A GGP agent should be able to play different games without the need of game-specific knowledge, and can be used as *off-the-shelf* implementations for game-playing agents. Researchers have made great progress in GGP recently, fueled by the further development of techniques such as Monte Carlo Tree Search (MCTS) and Upper Confidence Bounds for Trees (UCT) [Finnsson and Björnsson, 2008]. Noteworthy examples of recently developed agents that are based on MCTS are: the Alpha Zero agent [Silver et al., 2017], which can surpass human-level performance in the games of Chess, Go and Shogi; and a similar agent implementation called ExIt [Anthony et al., 2017], which defeated the previous benchmark agent in the board game Hex. However, the performance of such algorithms is due to their ability to plan over the future outcomes of its moves, a process that relies on a game simulator, capable of simulating a game by implementing its rules. Thus, these implementations rely on knowing the rules of the games they play *a priori*, limiting their scope and re-usability. In order to deal with such limitation, the agent could learn the rules of the game through observation and self experience, via model-based reinforcement learning (RL), and plan with a simulator that implements the rules it learned. However, model-based RL is an open field of research: learning good environment models (rules) is often challenging, even for simple environments [Kaiser et al., 2019].

In this work, we introduce an approach that improves the performance of agents that plan via sampling from an approximated game simulator by combining MCTS-based algorithms with random network distillation (RND) [Burda et al., 2018], a machine learning mechanism for measuring learning progress over an input. Our agent deals with incorrect state-transitions produced by the learned model by measuring the learning progress of a state-transition and taking such information into consideration during its planning [Katz et al., 2017]. We make the assumption that novel state-transitions, i.e. not seen during training, generated by the learned model, are likely to be incorrect samples that do not exist in the real environment. We empirically show that by incorporating the learning progress information over sampled transitions to MCTS-based algorithms, our agent is able to make better estimations over future outcomes.

Next, in Chapter 2, we provide the technical background for our work: traditional game playing algorithms, general game playing algorithms, reinforcement learning, and neural networks. Chapter 3 describe the game domains used to test our work — Hex, Othello, and Checkers. In Chapter 4 we present our agent implementation, how it learns game rules automatically, and the approaches we developed to improve its gameplaying performance when planning with learned rules. Chapter 5 present the results of our experiments in the test domains. Chapter 6 surveys related work. Finally, in Chapter 7, we conclude this work by discussing our empirical results, the limitations of our approach, interesting findings from our tests, and directions for future work. A partial version of this work was submitted for publication, where we demonstrated the applicability of our approach in the game of Hex, without previous knowledge of the game's rules.

2. BACKGROUND

This chapter introduces the technical background that is fundamental for our work. We start with a brief introduction over how games can be formally represented and some classical game-playing algorithms, focusing on those that many General Game Playing agents implement, including our agent. Later, we describe some of the theory that is key for our agent's learning process: how it can learn the game's rules and better strategies for the game by playing it, without expert knowledge.

2.1 Game Playing Algorithms

In this section, we briefly introduce formalization used by Russel and Norvig [Russell and Norvi Chapter 5, pp161-163] to describe games in general and two popular algorithms for playing competitive games. Games can be represented as an environment with many possible states, where 2 or more agents play against each-other in order to win the game (competitive game), or a single agent play to maximize its own score (single player game). For this work, we use the following notation in order to describe such games:

- S is the set of possible game states, where s_0 is an initial state.
- $S_T \subset S$ is the set of terminal game states.
- $\textsc{Player}(s)$ is a function that returns which player should make a move on state s.
- $\textsc{Actions}(s)$ returns the set of legal actions from state s.
- $\textsc{Result}(s, a)$ is the transition model, which returns the successor state s' from taking action a from state s. In stochastic games, $\textsc{Result}(s, a) = \mathcal{P}$ from MDPs.
- $\textsc{TerminalTest}(s)$ returns whether the state s is a terminal state or not.
- $\textsc{Utility}(s, p)$ is a utility function that defines how good a state s is for player p via a numerical value.

A game match starts at an initial state s_0 and continues until a terminal state s_T is reached. At each non-terminal state s_i, the $\textsc{Player}(s_i)$ function selects a player p, which selects an action $a_i \in \textsc{Actions}(s_i)$. Then, the transition function $\textsc{Result}(s_i, a_i)$ computes the next state s_{i+1} from applying action a_i at state s_i. The player's objective is to reach a terminal state s_T with the highest utility value, defined by $\textsc{Utility}(s_T)$ [Russell and Norvig, 2010, Chapter 5, pp161-163].

We can represent the game match as a tree of possible states, or game tree [Russell and Norvig, 2010, Chapter 5, pp162], where the tree's nodes are game states and its edges are actions that lead from one state to another. Each step of the tree is called a *ply*, and represents one action of a player in its turn. Games can have one or more *plys* per player's turn. In Figure 2.1, we show parts of the game tree of the game of Tic Tac Toe, a simple game tree, with few states. We can measure a game's complexity by the width and depth of its tree [Ramanujan et al., 2011]: deeper trees mean that matches can have a long duration; wider trees mean that the game have a large branching factor, and thus, its tree is computationally expensive to be explored thoroughly.

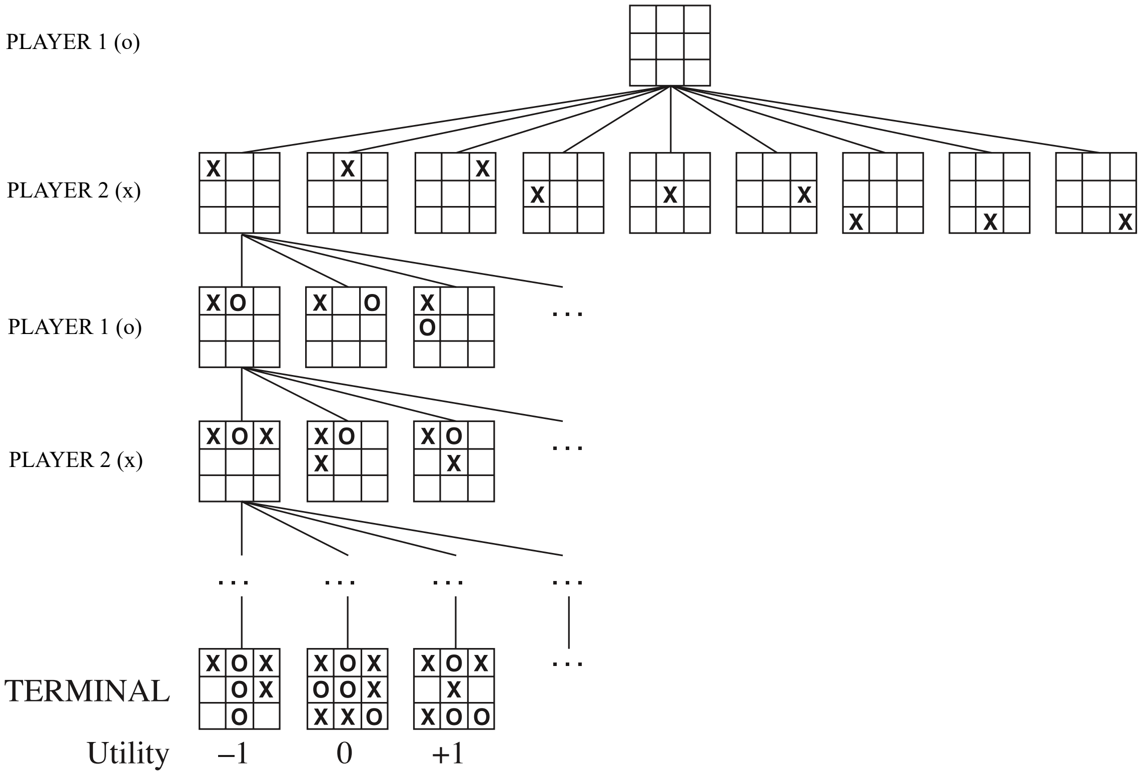

Figure 2.1 – the game tree of a Tic Tac Toe match [Russell and Norvig, 2010, Cap 5, pp163].

Once we model the decision making of a game as a tree, we can apply tree search algorithms to make game playing agents. In the next section, we introduce a family of tree search algorithms that are commonly implemented by General Game Playing (GGP) agents.

2.2 General Game Playing Search Algorithms

A GGP agent should be able to play more than one game using the same implementation. Thus, these agents are frequently implemented using algorithms that can make competitive plans *off-the-shelf*, with little (or none) expert knowledge.

In this section, we introduce two variations of a recently developed search algorithm that are implemented by many GGP agents. These algorithms differ from classic adversarial search algorithms in their great *off-the-shelf* performance in a variety of games [Browne et al., 2012].

2.2.1 Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) is a tree search method used by many GGP approaches. The success of this algorithm in GGP is due to several factors: empirical results show that with enough samples of trajectories through the tree, its search leads to competitive decisions [Browne et al., 2012]; it is computationally cheaper than most tree searches, since its search can be stopped at any time and its scalable, i.e. divisible into smaller chunks [Chaslot et al., 2008b]; and it depends on little to no, domain knowledge [Chaslot et al., 2008a]. MCTS works by running multiple *rollouts*, or Monte Carlo simulations on the most promising trajectories of the game-tree [Browne et al., 2012]. A *rollout* is a simulation of a match from a certain state down to a terminal state, where state transitions are obtained via a game simulator.

The algorithm keeps track of promising trajectories by building a search tree $\triangle_T$, where each node n holds the tuple $\langle s, \tilde{a}, Q, N\rangle$, where s is a game state, $\tilde{a}$ is the action that led to state s from a previous game state, Q is an approximated utility value, or Q-Value, associated with node n, and N is the number of times n was visited. The search tree $\triangle_T$ starts with a root node that holds the game's current state s_i, and which the algorithm expands during the search: a node n can have child nodes for each action $a \in \textsc{Actions}(n(s))$, where $n(s)$ is the state s that n holds. A node n is fully expanded when it has one child node for each action in $\textsc{Actions}(n(s))$. The algorithm uses two different strategies, or policies, for selecting states: a tree policy for selecting states from nodes in the search tree; and a *rollout* policy for selecting states during a *rollout*. MCTS builds $\triangle_T$ iteratively, where each iteration runs the following four steps, illustrated in Figure 2.2:

- Selection: the algorithm goes through $\triangle_T$, from its root node, and selects a node according to a tree policy π_T. This policy selects a node n that is not fully expanded yet.

- Expansion: the algorithms adds a new child node n_c to n with an untried action $a \in \textsc{Actions}(n(s))$.

- Simulation: this step runs a *rollout* starting from state $n_c(s)$. The *rollout* selects actions according to a *rollout* policy π_R.

- Backup: the utility value u obtained at the end of the *rollout* from the last step is set to $n_c(Q)$ and added to the Q-Value of all nodes in its trajectory back to $\triangle_T$'s root node.

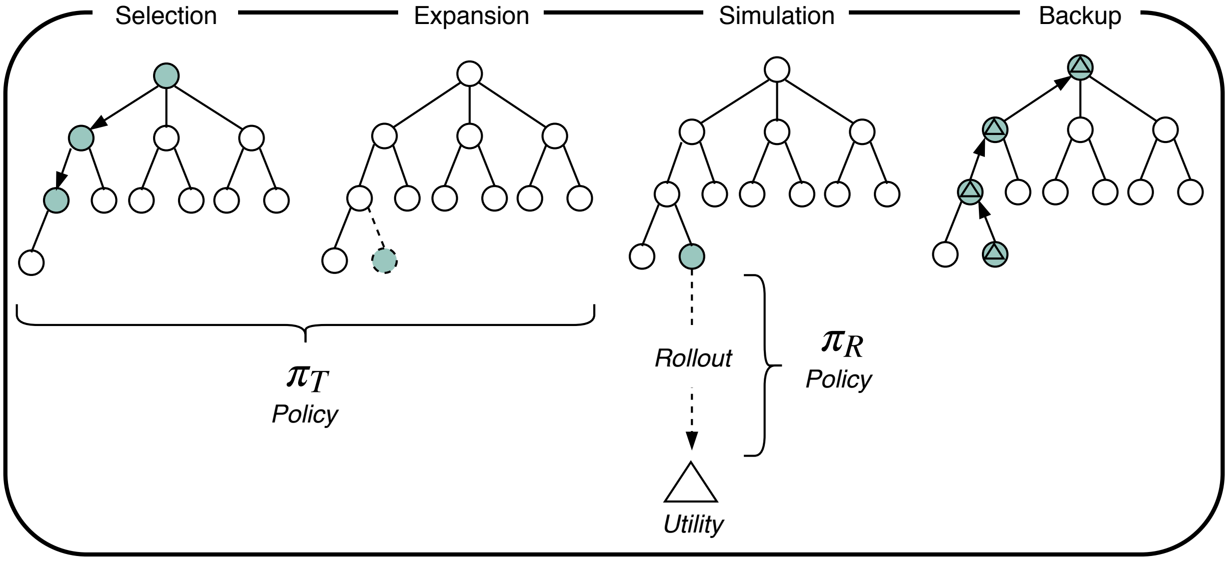

Figure 2.2 – the steps of each MCTS iteration [Browne et al., 2012].

After a *rollout* count or time constraint is reached, the algorithm selects a child node n_c from $\triangle_T$'s root node, according to the tree policy π_T, and returns the action $n_c(a)$. Algorithm 2.1 shows the MCTS method, where the BESTCHILD function, called in Lines 7 and 13, is the implementation of the node comparison made by policy π_T.

In the basic MCTS algorithm, this function always selects states with the highest mean Q-Value, as expressed in Equation 2.1. However, this selection strategy is greedy and does not take exploration into consideration, and thus, the algorithm can miss the optimal strategy by not exploring all possible trajectories [Browne et al., 2012]. In the next section, we discuss an MCTS implementation that deals with this exploration problem.

$$\overline{Q} = \frac{n(Q)}{n(N)} \tag{2.1}$$

2.2.2 Upper Confidence Bounds for Trees

Upper Confidence Bounds for Trees (UCT) is a variation of the MCTS algorithm that balances exploration and exploitation during its search by implementing the Upper Confidence Bound 1 (UCB1) method as its tree policy π_T [Browne et al., 2012]. UCB1 is a selection algorithm that was shown to balance exploration and exploitation in the context of a *k-armed Bandit Problem* [Sutton and Barto, 2018, Chapter 2, pp35-36].

Algorithm 2.1 – MCTS pseudo-code [Browne et al., 2012]

```
function MCTS( $s_0$ ) returns an action
    create root node $n_0$ with state $s_0$
    while within computational budget do
        $n_i \leftarrow$ TreePolicy( $n_0$ )
        $\triangle \leftarrow$ RolloutPolicy( $n_i(s)$ )
        Backup( $n_i$ , $\triangle$ )
    return a(BestChild( $n_0$ ))

function TreePolicy( $n$ ) returns a node
    while TerminalTest($n$) is False do
        if $n$ not fully expanded then
            return Expand( $n$ )
        else
            $n \leftarrow$ BestChild( $n$ )
        return $n$

function Expand( $n$ ) returns a node
    choose $a \in$ untried actions from Actions( $n(s)$ )
    add a new child $n_c$ to v
        with $n_c(s)$ = Result( $n(s)$ , $a$ )
        and $n_c(a) = a$
    return $n_c$
```

The k-armed Bandit Problem is a decision problem consisting of choosing a level to pull in a theoretical slot machine, such that each lever has a different probability distribution of yielding different prizes. The problem here is that, while a theoretical brute force solution consisting of pulling every lever a large number of times can yield the actual probability distribution for each lever, each time the agent pulls a lever not known to have the highest average reward so far entails a potential loss of utility. The exploration versus exploitation dilemma here is in occasionally choosing short term loss of potential utility in order to improve the estimate of the value in each lever.

The UCB1 method for the *k-armed Bandit Problem* is shown in Equation 2.2, where $\overline{X}_j$ is the average reward from lever j, N_j is the number of times that lever j was pulled, and N is the number of pulls so far [Browne et al., 2012]. The $\overline{X}_j$ term encourages the exploitation of higher-reward levers, since they yield higher mean reward, while the $\sqrt{\frac{2 \ln N}{N_j}}$ term encourages the exploration of less pulled levers, since it yields higher values as the number of pulls N increases and the N_j stays small.

$$UCB1 = \overline{X}_j + \sqrt{\frac{2 \ln N}{N_j}} \tag{2.2}$$

Algorithm 2.1 – MCTS pseudo-code [Browne et al., 2012] (continued)

```
function ROLLOUTPOLICY( $s$ ) returns a utility value
    while TERMINALTEST($s$) is False do
        choose $a \in$ ACTIONS( $s$ ) uniformly at random
        $s \leftarrow$ RESULT( $s$ , $a$ )
    return UTILITY($s$ , PLAYER($s$))

function BACKUP( n , $\triangle$ )
    while n is not null do
        $n(N) \leftarrow n(N) + 1$
        $n(Q) \leftarrow n(Q) + \triangle$
        $n \leftarrow$ parent of $n$
```

The idea of UCT is that the selection step of the MCTS algorithm can be seen as a series of *k-armed Bandit Problems*, where selecting the most promising child node is equivalent to selecting the most promising lever. Thus, UCB1 can be used as π_T policy in order to address the exploration-exploitation dilemma during the tree search. Algorithm 2.2 shows the selection function commonly used in UCT implementations, where c is an exploration constant that is added to the exploration term of the UCB1 formula to control the amount of exploration during the tree search.

Algorithm 2.2 – the selection function of the UCT algorithm [Browne et al., 2012]

```
function BESTCHILD( $n$ , $c$ ) returns a node
    return $\underset{n_c \in \text{ children of } n}{arg\ max} \frac{n_c(Q))}{n_c(N)} + c\sqrt{\frac{2 \ln n(N)}{n_c(N)}}$
```

2.3 Reinforcement Learning

Reinforcement Learning (RL) is a field of Artificial Intelligence that studies how agents can learn to make decisions in an environment without any supervision, only by interacting with it [Sutton and Barto, 2018, Chapter 1, pp1-2]. In order to do so, RL agents need to solve two related problems, prediction and control: prediction is the process of computing or estimating the utility of its actions while control is the ability to make decisions based on its predictions [Kaelbling et al., 1996].

In this section, we provide the necessary mathematical background in order to understand reinforcement learning agents. First, we detail Markov Decision Processes. Second, we discuss the differences between different types of RL agents. Third, we detail policy iteration algorithms — these algorithms are the basis for the techniques implemented by state-of-the-art GGP agents. Later, we detail SARSA and Q-Learning, which are algorithms typically used to solve models which are hard to model, and share many similarities to the MCTS algorithm. Finally. we discuss how RL algorithms approximate policies with approximation.

2.3.1 Markov Decision Process

Reinforcement learning problems and domains are usually represented through a Markov Decision Process (MDP) [Sutton and Barto, 2018, Chapter 3]. MDPs are a classical formalization of sequential decision making: the problem of making a sequence of decisions under uncertainty, where each decision can depend on the outcome of previous decisions.

In an MDP, a decision maker, or agent, interacts with an environment in a sequence of discrete time steps. Taking actions yields immediate and delayed rewards to the agent, and changes the environment configuration (state). This change is expressed as a transition from one state to another, and perceived by the agent as an experience tuple $\langle s, a, r, s^{'} \rangle$, where s is the last state, a the action taken from state s, r the reward yielded by taking a at s, and transitioning to the environment's next state $s^{'}$. MDPs can be represented by the tuple $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma)$, where:

- $\mathcal{S}$ is a finite set of states that represents all possible environment configurations;
- $\mathcal{A}$ is a finite set of actions that can be taken by the agent;
- $\mathcal{P}(s' \mid a, s)$ is a transition model, a probability function that gives the probability of a transition from state s to state s' when taking action a;
- $\mathcal{R}(s)$ is a reward function, that gives the reward r from state s; and
- γ is the *discount factor*, a number between 0 and 1 which controls the agent's preference for current rewards over future ones.

The objective of an agent in an MDP problem is to find a behavior, or policy, that yields to the highest expected return: the total reward over the course of its interaction with the environment [Kaelbling et al., 1996]. If an agent $\overline{A}$ follows policy π at time step t, then function $\pi(a \mid s)$ is the probability of $\overline{A}$ selecting action a at state s. A policy is optimal if $\pi(a \mid s)$ assignts the highest probabilities to actions that lead to states with the highest rewards, and thus, leads to the highest expected return for any state $s \in \mathcal{S}$.

A *value function* returns the expected return of an agent following policy π from a starting point, which can be either a state or a state-action pair [Sutton and Barto, 2018, Chapter 3, pp 58]. With this function, the agent can evaluate states without the need of persistent rewarding from the environment at each time step, which is useful in many domain, such as video-games. In the following section, we present the two categories in which RL algorithms are typically divided.

2.3.2 Model-based and Model-free RL

Reinforcement Learning algorithms can be divided into two categories: model-based RL and model-free RL [Sutton and Barto, 2018, Chapter 8, pp159]. Model-based RL algorithms learn (or receive) the environment model and then solve the MDP for that model in order choose its actions. However, MDPs with a large state-space, like most board games, are prohibitively expensive to solve [Sutton and Barto, 2018, Chapter 4, pp85]. In order to deal with this limitation, model-based RL algorithms can instead use planning or search algorithms, such as MCTS, for choosing its actions.

Conversely, model-free RL algorithms do not depend on the environment's model in order to find optimal policies, and instead learn value functions from rewards collected from experience with the environment. Such algorithms are useful for environments that are hard to model, but are often less efficient than model-based methods.

Since we intend to implement an agent that learns how to simulate a game's match, via a learned transition model, we consider model-based RL algorithms for this work. Next, we introduce a classic model-based RL algorithm that is the base for most model-based RL algorithms, including those used by *state-of-the-art* GGP agents.

2.3.3 Policy and Value Iteration

Policy iteration is an RL algorithm that computes optimal policies given a perfect MDP model of the environment [Sutton and Barto, 2018, Chapter 4]. It is part of a class of RL algorithms known as dynamic programming methods and is guaranteed to find the optimal policy in polynomial time [Kaelbling et al., 1996].

Policy iteration is a control RL algorithm that searches for optimal policies by iteratively running two steps: policy evaluation and policy improvement [Sutton and Barto, 2018, Chapter 4, pp80-82]. Policy evaluation is a prediction algorithm, that computes the state-value function, a function that returns the estimated utility of a state, for an arbitrary policy π through approximation, while policy improvement finds an improved policy π' for π with the approximated utility values from the policy evaluation step.

The policy evaluation step starts with an initial state-value function υ_0, where the utility value of all states is zero. From υ_0, successive improved approximations υ_{i+1} are obtained for υ_π with values of υ_i. This iteration process continues until the evaluation step converges.

From an initial random policy π_0, i.e. a policy that selects actions at random, the policy improvement step iteratively improves π_i with the results of Equation 2.3, where $p(s', r \mid s, a)$ is the probability of getting reward r from state s' after selecting action a from state s, υ_π is the current state-value of state s', found by the evaluation step, and $\underset{a}{argmax}$ selects the actions that maximizes the equation. If the improved policy π_{i+1} does not change any state selection from π_i, then both policies are optimal, and any one of them can be returned. Otherwise, the policy iteration algorithm returns to the policy evaluation step.

$$\pi'(s) = \underset{a}{argmax} \sum_{s'} \sum_{r} p(s', r \mid s, a) \Big[r + \gamma \, \upsilon_\pi(s') \Big], \forall s \in S \tag{2.3}$$

The value iteration algorithm is very similar to policy iteration, however, it does not run the policy evaluation step until convergence, and instead, does one *sweep* of evaluation followed by a single policy improvement update [Sutton and Barto, 2018, Chapter 4, pp83-85]. It continues to iterate through both steps until a certain accuracy θ is reached, i.e the difference between the values of υ_i and υ_{i+1} are less than θ. Figure 2.3 shows the computation done by the value iteration algorithm on a simple 4x4 Grid World problem, where the optimal policy is found on the third iteration.

The core of these algorithms, the idea of a policy evaluation process and a policy improvement process interacting with each other, iteratively *getting closer* to the optimal policy, is the basis of most RL algorithms [Sutton and Barto, 2018, Chapter 4, pp86-87]. Both algorithms, as they are, however, require a model to be given beforehand, i.e. full domain knowledge. In the next section, we discuss two model-free algorithms that do not have this limitation (since they require no model), and the challenges involved in planning without a model.

2.3.4 Temporal-Difference Learning

Temporal-difference learning is a class of *model-free* RL algorithms that estimate value functions by *bootstraping*: estimates are improved based on learned estimates, without knowing the actual final outcome [Sutton and Barto, 2018, Chapter 6, pp119]. TD methods are widely used in practice because they are *model-free*, and most environments are *hard* to model. Many game-playing agents are implemented using TD methods, specially for board games and video-games.

TD methods search for the value function by updating a current estimate $V(S)$ based on the reward obtained at a *future* time-step and a *new* estimate made at that time

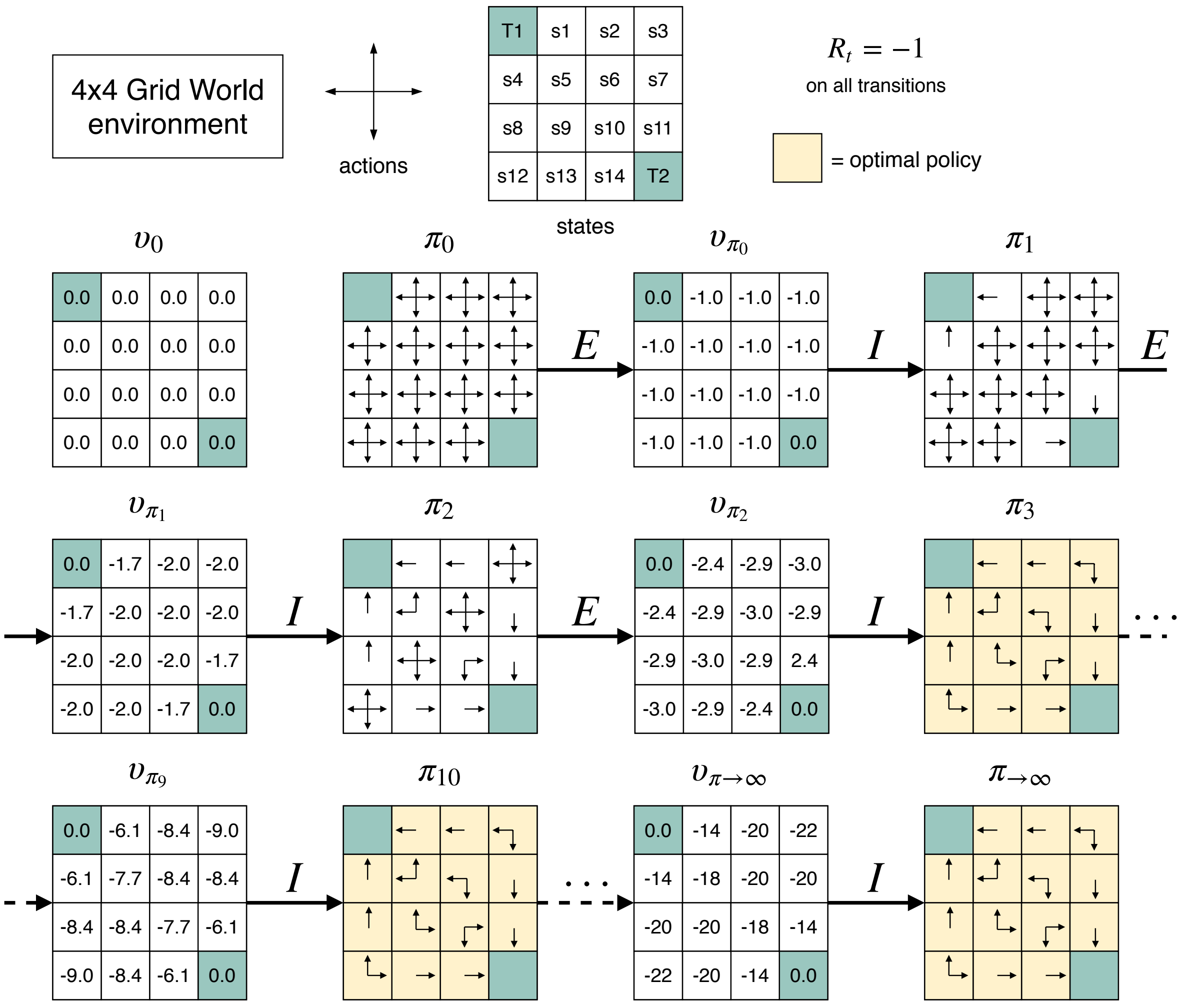

Figure 2.3 – an example of the Value Iteration algorithm running on a 4x4 Grid World.

$V(S')$. Equation 2.4 shows the update function used by TD algorithms, where t expresses the current time-step, α is a *learning-rate*, and the term $\left[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)\right]$ computes the difference from the current estimate $V(S_t)$ and the better estimate $R_{t+1} + \gamma V(S_{t+1})$. This difference, called *TD error*, is the error from the current estimate $V(S)$, which is added to $V(S)$ in order to correct it *towards* the real value function v_π. With small values of α, this algorithm is guaranteed to converge to v_π [Sutton and Barto, 2018, Chapter 6, pp126].

$$V(S_t) \leftarrow V(S_t) + \alpha\left[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)\right] \tag{2.4}$$

Control RL algorithms typically learn the action-value function rather than the state-value function, since these algorithms must learn action-selection rather than state utilities: they can keep utilities for each action a at each state s rather than transitioning through all successor states S' from s and comparing their utilities. Therefore, for TD control algorithms, we consider a modified TD update, shown in Equation 2.5, where, like the MCTS algorithm, Q is the Q-Value, the estimated utility of the state-action pair. This update rule inspired the name of a TD control algorithm which uses it, called SARSA [Sutton and Barto, 2018, Chapter 6, pp129]. Specifically, the name comes from the equation's use of every element in the quintuple $(S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1})$ that is obtained at each state-action pair transition.

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\big[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)\big] \tag{2.5}$$

Similarly to the policy iteration and value iteration methods, the SARSA algorithm improves a policy π with the updated estimated utility values, or Q-Values in this case, from its TD updates. However, this algorithm faces a exploration-exploitation dilemma similar to that of the MCTS algorithm, detailed in Section 2.2.1: it cannot make greedy decisions like DP algorithms, that always chooses the best actions, because its evaluation does not update over all the complete distribution of all successors, and thus, does not guarantee that all states are visited and updated. Therefore, without any exploration, the algorithm might not learn the optimal policy, since it does not learn the complete action-value function for the environment.

In order to deal with this dilemma, the SARSA algorithm implements a bandit algorithm, in the same way that the UCT algorithm implements the UCB1 method to solve this dilemma. It is proven that a bandit algorithm that generates soft policies is guaranteed to converge its approximations to the real action-function as the number of iterations increases toward infinity [Sutton and Barto, 2018, Chapter 6, pp129]. Soft policies are those that keeps on exploring — a simple example would be the ϵ-greedy algorithm: it selects the best action most of the time, but some times it decides to explore, i.e. selecting a random non-optimal action. The amount of exploration considered is controlled by a variable ϵ, where the chance of greedy selection is $1 - \epsilon$, while the chance of exploring is ϵ.

Algorithm 2.3 shows SARSA in pseudo-code, where Q is implemented as a table called Q-Table, and $Q(s, a)$ maps to a specific position in that table. We left the action selection function SELECTACTION (Lines 9 and 14) undefined, since it could be implemented as any soft policy bandit algorithm, such as ϵ-greedy, UCB1, or others.

The main drawback of SARSA is that it searches for the state-action function by compromising the policy that it actually learns [Sutton and Barto, 2018, Chapter 5, pp103]: in order to converge its approximations towards the optimal state-action function, it needs to follow a soft policy, one that is not optimal, since it selects states that are not optimal from time to time. This characteristic of SARSA is known as on-policy control, since it learns and behaves over the same policy.

Another TD algorithm called Q-Learning solves this drawback by learning off-policy — it utilizes two policies: the policy it learns, called target policy, and the policy that dictates its own behavior, the behavior policy [Sutton and Barto, 2018, Chapter 6, pp131-132]. Because of this distinction, Q-Learning is able to learn an optimal policy. On-policy methods can be seen as a special case of off-policy method, where the target and behavior policies are the same.

Algorithm 2.3 – *SARSA* pseudo-code [Sutton and Barto, 2018, Chapter 6, pp130]

```
function SARSA( $\alpha$ )
    for each state $s$ in S do
        for each action $a$ in Actions(s) do
            if TerminalTest(s) then
                $Q(s, a) \leftarrow 0$
            else
                $Q(s, a) \leftarrow$ a random value
    for each episode $e$ do
        $a \leftarrow$ SelectAction(s) (using policy derived from Q)
        for each step of $e$ do
            if TerminalTest(s) then
                break
            $r$, $s' \leftarrow$ Result(s, a)
            $a' \leftarrow$ SelectAction($s'$) (using policy derived from Q)
            $Q(s, a) \leftarrow Q(s, a) + \alpha\big[r + \gamma Q(s', a') - Q(s, a)\big]$
            $s, a \leftarrow s', a'$
```

The basic Q-Learning implementation follows a soft policy and learns the optimal policy by improving the target policy with a direct, greedy action selection. Algorithm 2.4 shows the pseudo-code for a basic Q-Learning agent, where the main difference from the SARSA algorithm can be seen in Line 14, where it follows a strictly greedy policy, by always selecting the action $\tilde{a}$ that maximizes $Q(s', \tilde{a})$. In the next section, we discuss the approach used by RL algorithms in order to better generalize value functions, such as the ones learned by SARSA and Q-Learning.

Algorithm 2.4 – *Q-Learning pseudo-code* [Sutton and Barto, 2018, Chapter 6, pp131]

```
 1: function Q-Learning( $\alpha$ )
 2:     for each state $s$ in S do
 3:         for each action $a$ in Actions(s) do
 4:             if TerminalTest(s) then
 5:                 $Q(s, a) \leftarrow 0$
 6:             else
 7:                 $Q(s, a) \leftarrow$ a random value
 8:     for each episode $e$ do
 9:         for each step of $e$ do
10:             if TerminalTest(s) then
11:                 break
12:             $a' \leftarrow$ SelectAction($s'$) (using policy derived from Q)
13:             $r$, $s' \leftarrow$ Result(s, a)
14:             $Q(s, a) \leftarrow Q(s, a) + \alpha\Big[r + \gamma \underset{\tilde{a}}{max} Q(s', \tilde{a}) - Q(s, a)\Big]$
15:             $s \leftarrow s'$
```

2.3.5 Policy Prediction with Linear Approximation

In this section, we discuss a novel RL approach that represents value functions via parameterized functions rather than a table [Sutton and Barto, 2018, Chapter 6, pp205-210]. Previously discussed methods relied on keeping a table, such as the Q-Table, in memory, which can be impractical for environment's with a large state-space.

In order to use such approach, we need to represent every state s as a feature vector $\mathbf{x}(s)$, which holds the features of s. Equation 2.6 shows the linear function $\hat{v}(s, \mathbf{w})$ [Sutton and Barto, 2018, Chapter 6, pp205], that approximates the state-value function $v_\pi(s)$ by the inner product of a weight vector $\mathbf{w}$ and $\mathbf{x}(s)$.

$$\hat{v}(s, \mathbf{w}) \doteq \mathbf{w}^\top \mathbf{x}(s) \doteq \sum_{i=1}^{d} \mathbf{w}_i \mathbf{x}_i(s) \tag{2.6}$$

We can approximate $\hat{v}(s, \mathbf{w})$ to the real state-value function by adjusting the values of the weight vector $\mathbf{w}$. However, since adjusting one weight can affect many states at once, its not possible to get the exact value for all states, we can only generalize. Therefore, we measure the error of the approximation over the mean difference between $\hat{v}(s, \mathbf{w})$ and $v_\pi(s)$. This error can be obtained by Equation 2.7, which is mean squared value error $\overline{VE}$ [Sutton and Barto, 2018, Chapter 6, pp199], where $\mu(s)$ is a distribution over $\mathcal{S}$ that controls the tolerance for error of each state $s \in \mathcal{S}$.

$$\overline{VE}(\mathbf{w}) \doteq \sum_{s \in \mathcal{S}} \mu(s) \Big[v_\pi(s) - \hat{v}(s, \mathbf{w}) \Big]^2, \text{ where } \mu(s) \geq 0, \sum_{s} \mu(s) = 1 \tag{2.7}$$

Many approximation methods can be considered for computing $\hat{v}(s, \mathbf{w})$. In the following chapter, we introduce the class of approximation methods considered for this work, and discuss other applications for it that are relevant to our research.

2.4 Artificial Neural Networks

In this section, we provide a brief background on the approximation method considered for this work: Artificial Neural Networks (ANN) [Tan et al., 2005, Chapter 5, pp246-256] can be used to approximate any target function, be it linear or nonlinear. This method is implemented by most modern RL algorithms for its approximation capabilities, for model-based RL it can be used to approximate the environment's model [Sutton and Barto, 2018, Chapter 8, pp159-169].

An artificial neural network is structured by three different types of layers, each composed of various nodes [Tan et al., 2005, Chapter 5, pp251]: an Input Layer that receives data and feed signals to the rest of the network; Hidden Layers are optional layers that process incoming signals and feed them forward into the network; and the Output Layer receives the signals from all previous layers and produces the network's output. Figure 2.4 illustrates a simple ANN example with 2 hidden layers, where nodes and weights are labeled.

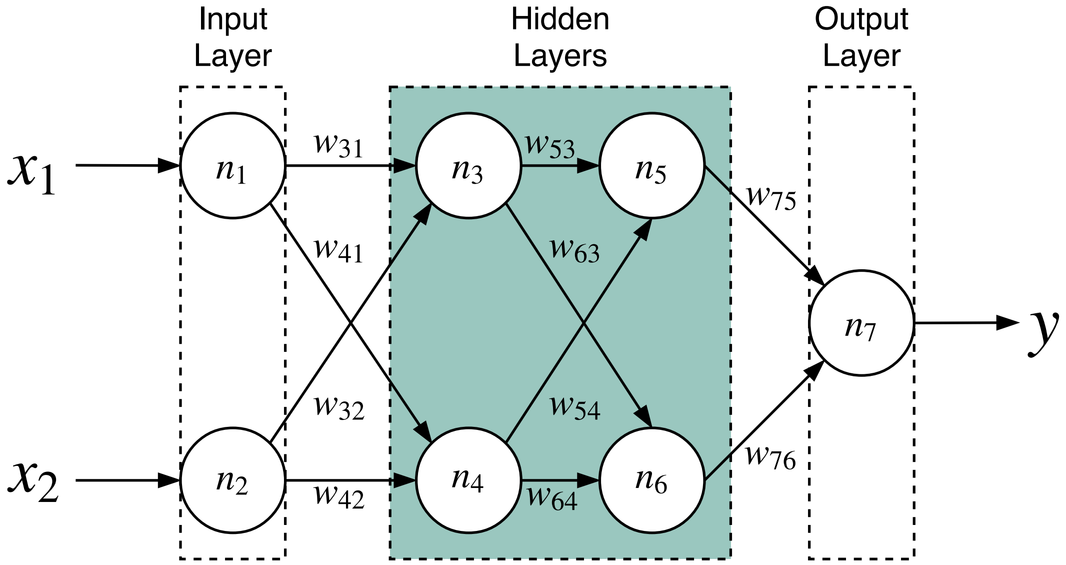

Figure 2.4 – a simple artificial neural network with 2 hidden layers [Tan et al., 2005, Chapter 5, pp253].

The goal of an ANN is to learn values for its set of weights that minimizes the error between the network's output $\hat{y}$ and the target function's output y. In order to do so, they adjust each weight iteratively, using an algorithm called backpropagation [Tan et al., 2005, Chapter 5, pp254]: at the start of each iteration, it does a forward sweep through the network, computing the output of each node, until it reaches the output layer; then, it computes the difference between its output $\hat{y}$ and the output y of the target function, and back-propagates this difference to the rest of the network.

2.4.1 Random Network Distillation

Burda et al. [Burda et al., 2018] developed random network distillation (RND) as a method for measuring state novelty, used to develop intrinsically curious RL agents. A curious agent using RND was able to achieve promising results in the Atari game *Montezuma's Revenge*, notoriously difficult for non-curious agents because of its complex, maze-like setting [Burda et al., 2018]. State novelty is a measure to indicate how dissimilar a state is in relation to those that the agent has already seen during its training process. Curious RL agents are those that use this information to explore new trajectories during training.

Previous work on curious RL agents relied on learning an approximate model of the environment, via neural networks, and state novelty was obtained by the prediction error between the predicted next state (the network's output) and the environment's next state [Stadie et al., 2015]. However, such approach was not robust against common prediction pitfalls, such as using a model that is too simple for the complexity of the environment it is trying to predict [Burda et al., 2018]. Because of this, their agent performed poorly on *Montezuma's Revenge*, and achieved worse gameplay performance in other Atari games when compared to that of the RND agent.

The RND technique measures novelty by means of two neural networks: a predictor and a target. The predictor network is trained to predict the output of a fixed and randomly initialized target network on the next state when given the next state itself. This target network is never trained, its weights remain as they were initialized. Figure 2.5 shows an overview of the RND architecture, where the same state is given as input to both networks and the novelty is obtained by the mean square error of their outputs. The idea is to *distill* the randomly initialized target network into the predictor network — distill in the sense that the predictor network eventually learns to replicate the target network, since it is trained to fit the output of the target. The novelty of a state *s* is measured by the predictive error of the predictor network over the output of the target network for *s*. Novel states (states not frequently seen during training) are more likely to yield high prediction error, since the predictor network may not be fitted to the target network's output yet — or not as fitted as states that are frequently seen.

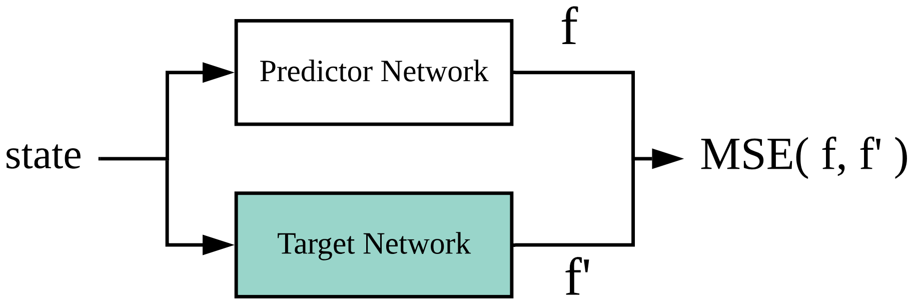

Figure 2.5 – the structure of the RND network [Burda et al., 2018].

The novelty measured by the RND can also be interpreted, in a more general interpretation, as learning progress. With this in mind — and the impressive results obtained by Burda et al. in *Montezuma's Revenge* — we believe that RND is a promising technique that could potentially benefit many RL agents. Later in this work, we detail how our agents implement RND in order to make better plans with learned models, demonstrating the versatility of this technique.

3. GAME DOMAINS

This chapter details the game domains that we use as test domains for this work, and where we executed our experiments. Here, we explain the rules of each game, and discuss the challenges and different characteristics of each game and how they differ from each other.

First, we describe the game of Hex, the *simplest* game used for this work. Later, we describe Othello, which is another classical board game that is similar to Hex in many ways, but has more complicated rules. Finally, we cover the game of Checkers, a *simple* classical game that has an element that previous games lack: piece movement — which makes its rules more challenging to be approximated.

3.1 Hex

Hex is a classic board game where two players connect two edges of an $n \times n$ hexagonal grid by placing pieces on it. One player controls white pieces while the other controls black pieces, and they place one piece per turn. Players can pick any spot on the board in order to place pieces. Pieces that are placed on the board stay there for the rest of the match. The first player to connect a string of pieces from one side of the board to the other, wins.

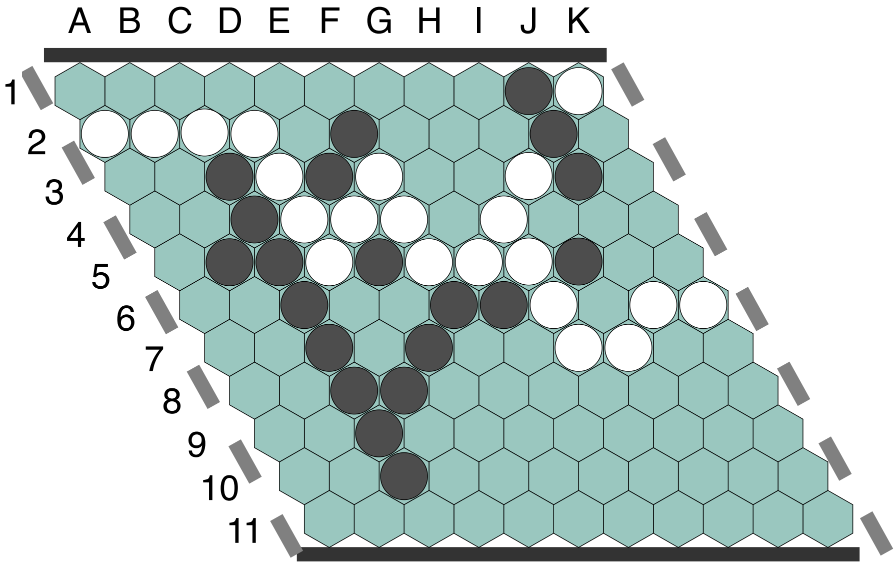

Figure 3.1 – an example of a terminal state in an 11×11 Hex game, where the white player wins.

Figure 3.1 shows a terminal state in a game of 11×11 Hex, a typical board size in competitive play, where the player with white pieces wins. Hex is particularly challenging because there are many positions to consider for each move and there are many moves which are very hard to counter, such as blocking the greatest string made by the opponent or trapping the opponent into a corner.

We use Hex for this work for the following reasons. First, it has simple piece-positioning rules; players can play Hex on any $n \times n$ board, where $n \geq 3$, so the complexity of the decision-making process can be easily scaled using the same set of rules. Second, there are no ties in a match of Hex, so the utility of endgame positions are either +1 or -1, simplifying the decision-making process. Third, it is a challenging game, especially for larger board sizes. Thus, we believe that it is a good archetype of classical board games.

3.2 Othello

Othello (also known as Reversi) is a classical board game that is played on an 8×8 square grid, where players place pieces of two different colors, black and white. One player controls the black pieces, while the other controls the white ones, similarly to Hex. The game starts at an initial board setup, illustrated in Figure 3.2. As in Hex, at each turn, players must place one of their pieces in an empty position on the board, and pieces already placed on the board are never removed. However, in Othello, players can capture each other's pieces: when a player places a piece on the board, this player can capture any opponent piece that is in between the recently positioned piece and another one of its own pieces already on the board, either in a straight line or diagonal. When a capture occurs, the captured pieces change their color to that of their captor's pieces. Players can only put pieces at positions that will result in at least one capture, and if that cannot occur, the player must pass its turn. The game continues until each player can no longer move, either by the capturing restriction or lack of empty spaces. The player with most pieces placed on the board is the winner. While matches may end in a draw, they are rare.

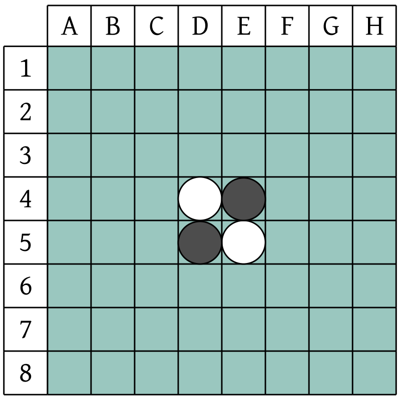

Figure 3.2 – the initial board configuration in a game of Othello.

Figure 3.3 shows an example of a complex multi-piece capture, where the player with black pieces successfully captures most of its opponents pieces with a single move: the black player puts a new black piece at position C6, and from there, all white pieces in between another black piece are subsequently captured, in a straight line upwards, to the right, and diagonally to the right. Notice that the white piece at position F6 is not captured, and that is because that piece was already in between two black pieces before the new piece at C6 is placed — that is the only exception to the capturing rule.

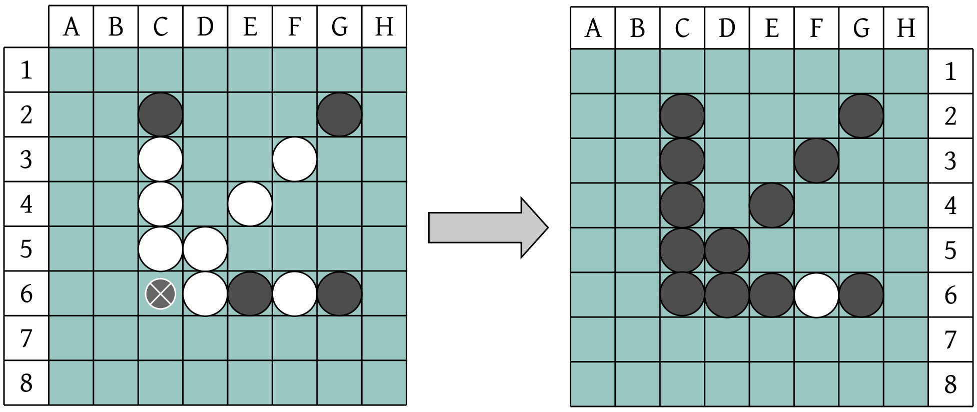

Figure 3.3 – an example of a complex capture in Othello.

We believe that, like Hex, Othello has simple rules, but can be quite challenging and competitive. However, its rules can be viewed as a superset of those of Hex — specifically, the RESULT function. The ACTIONS function is also much more complicated than that of Hex, since the capture condition for each move. The only exception is the TERMINALTEST, which is much simpler: it is simply to count each player's pieces on the board and compare both quantities. Therefore, we believe that Othello is a perfect step up in complexity when compared to Hex, it present more complicated rules, but also many similarities, which enables us to draw comparisons between the two to better analyze them.

3.3 Checkers

Checkers is the third, and last, game use in this work. Like previous games, Checkers is a classical board game that is very popular worldwide. This game has many regional variants, and is played with a variety of different rules. For this work, we use the Brazilian variant of the game, which is played in an 8×8 square board with 12 starting pieces positioned on the board for each player, this starting board setup is illustrated in Figure 3.4. Checker's rules are more complex than those of both Hex and Othello, however, that does not mean that the previous games are less competitive than this one.

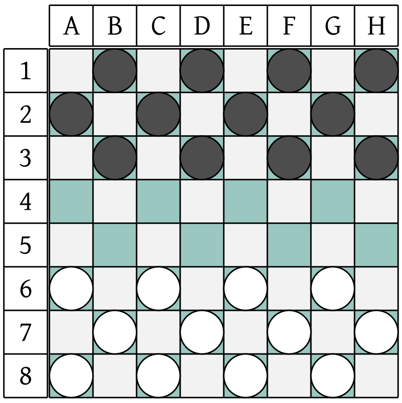

Figure 3.4 – the setup board for a game of Checkers.

The key difference between Checkers and previous games is that players don't place pieces in Checkers, but rather move them in 4 possible diagonal directions: diagonally up, down, right, and left. Like Othello, pieces can be captured, but captured pieces are removed from game, and not *converted*. Another distinct feature is that Brazilian Checkers have two different types of pieces: regular pieces and kings (also known as *flying kings* or *ladies* in Brazil); regular pieces can only move toward the opponent side of the board, while kings can move in all four directions. These features (piece capturing, movement, and types) make Checkers rule harder to approximate than previous games: the RESULT function is more complex, with special rules that we detail later in this section; the ACTIONS function is also more complex and have its own conditions; and the TERMINALTEST function is more complex than that of Hex.

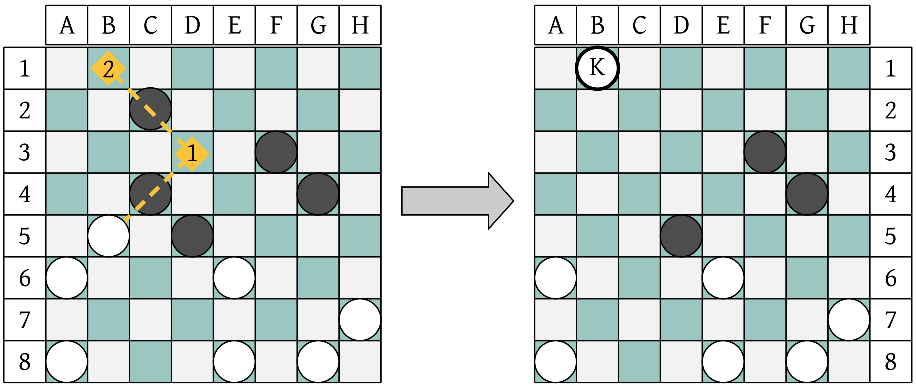

Figure 3.5 – an example of chained piece capture and piece promotion in a game of Checkers.

From the initial board setup, players must move their pieces diagonally toward the opponent side of the board. Players can move pieces in two different ways: a regular move is one that moves the piece to an empty space immediately next to it in one of the possible diagonal directions; a capture move is one that moves the piece to an empty space immediately next to an opponent piece, by following one of the possible directions two consecutive times. In Brazilian Checkers, when an opponent piece **can** be captured, it **must** be captured, and captures must be *chained*, if one capture leads to the opportunity of another one, it must happen immediately. When a piece arrives at the opposite end of the board (the opponent's end), it is promoted to a king. The game continues until one player can no longer make moves, and the player with more pieces still on the board, wins. Figure 3.5 illustrates a chained piece capture and a piece promotion to king: the white piece at 6A moves to position 1, at D3, captures the black piece at position C4, and immediately moves to position 2, at 1B, capturing a second black piece at C2; Then, the same white piece that made this move, is promoted to king, represented with a capital 'K'.

Now that we described each game, it is easier to see how we ordered them by the complexity of their rules: Hex has the simplest rules, in spite of being very competitive and challenging to play; Othello is similar to Hex in many aspects, however, its state-transition is much more complex; Checkers is very different than previous games, with piece capturing (that removes pieces from the board), movable pieces, piece categories, and some special rules. However, they share many similarities — all of them are played on square boards (Hex's board can be represented as a square board) and have simple rules (for a human player), involving colored pieces on the board — and thus, it is easier to understand how the small differences between their rules can impact an agent that plans over the learned rules.

4. PLANNING WITH LEARNED RULES

We now develop our approach for improving an agent's game-playing performance when planning with learned rules. First, we discuss how our agent learns the rules of the classical board games described in Chapter 3, the details of our implementation, and how each game is represented.

Later, we develop our approach to detect inconsistencies in the learned model output via a method inspired by RND. Finally, we introduce a modification to the base UCT algorithm that is able to make better plans by taking inconsistent simulations into consideration.

4.1 Learning game rules

Our agent learns the rules of games by training an artificial neural network with the data obtained by interacting with the game and observing traces from other agents. For this work, we relaxed the problem of learning the entire model and focused on the RESULT(s, a) function (the state-transition function), so our agent is given the actual TERMINALTEST(s) and ACTIONS(s) functions and does not have to learn them.

This is because the state-transition function is what drives the game simulation, and we believe it is better to investigate its performance in isolation before introducing errors and noise generated from other approximated functions. Moreover, the ACTIONS(s) function is only necessary for board games, as discussed earlier, and the TERMINALTEST(s) function could also be ignored in most environments, such as video-games.

In order to represent each game's states generically, we developed a *vectorized* representation for each game. This representation is implemented by each game's simulator and given to our agents as observations. We use such representation in order to have a standard form of observations, so the same agent implementation can be used to learn the rules of each game — training different models separately, instead of training a universal transition function. Our agent could be more *general* if it could learn entirely from images, but that would also introduce an unnecessary variable in our learning of the rules and the evaluation of the effectiveness of our reinforcement learning algorithm.

Figure 4.1 shows the vectorized representation of states and actions in the game of Hex that are implemented by our network: each state is represented by a vector of size $n \times n \times 2$, where n is the size of the game board, and the first $n \times n$ positions represent the first player's pieces on the board, while the rest represents the same information for player 2, where 1 indicates that a piece is positioned on that position and 0 otherwise. Actions are similarly represented, where the index of 1 indicates the position and the player that made that action. This figure represents an example of a training data for our network, the network receives a vector of size $n \times n \times 4$ (state and action) and outputs a vector of $n \times n \times 2$ (the successor state).

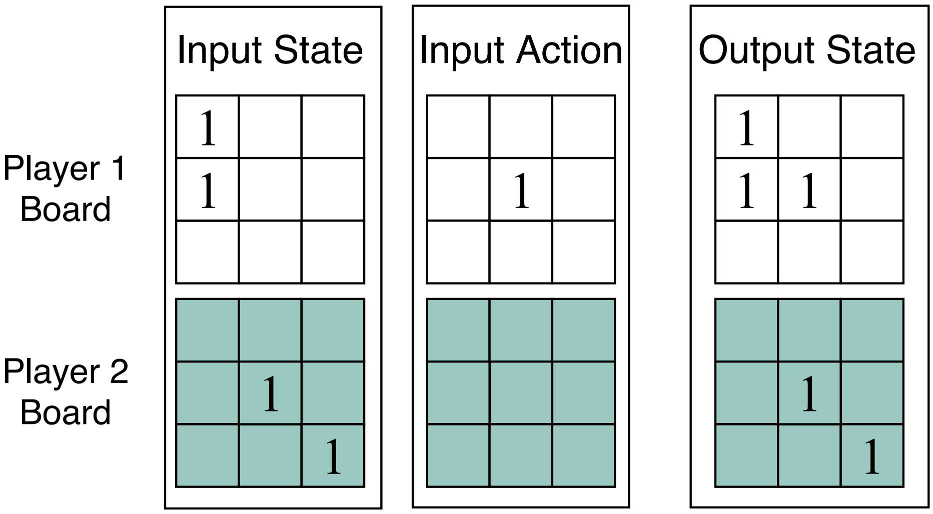

Figure 4.1 – our vectorized representation of state and actions in a game of 3 x 3 Hex.

We implemented our neural network with 4 fully connected layers: an input layer with the size $n \times n \times 4$ (state and action), two hidden layers with two times the size of the input data, and an output layer with the size $n \times n \times 2$ (the successor state), using *ReLU* as activation function for all layers. Network parameters were optimized by stochastic gradient descent, using the mean squared error of its prediction and the correct successor state as loss function. A state-transition example for this network is represented in Figure 4.1.

The vectorized representation of Othello is very similar to that of Hex, but output states change more than one value at a time (because of piece capturing), so the prediction task is more challenging than in Hex. Because of this, the same network architecture used in Hex was not able to learn Othello's rules so easily, so we added two more hidden layers and changed the length of hidden layers to be 16 times that of the input layer.

The same network architecture used for Othello was applied to learn the rules of Checkers, but we had to change the representation of actions in that game. Actions in Checkers are represented differently because in this game players move pieces, instead of placing them. So we represent actions with two signs: a -1 on the board position where the piece is, and a 1 where the piece should move. This is a slightly informative representation, but as discussed earlier, training a complex model to solve a complex prediction task is not the main interest of this work, rather, we want to show that our agent can play games using approximated models.

4.1.1 Trained models and accuracy

With our networks, we trained a variety of predictive models for each game, with different accuracy values, which served as base for our experiments. We trained each model with different datasets of state-transitions: with state-transition data gathered from only 30 games ($5,690$ state-transitions), our Hex network was able to achieve 100% accuracy in 11×11 Hex, and models with lower accuracy were obtained by training with less game data. In order to measure each model accuracy, when tested each one against a dataset of 2000 games ($378,944$ state-transitions). We obtained similar results with our networks for Checkers and Othello, but with data from more games (1000 for Othello, and 250 for Checkers).

While some models achieved 100% accuracy in our tests, this does not mean that the network learned the perfect model, just that it does not made a single error in test data, although this is a good indicator of the quality of the learned model. To further ensure the quality of the learned models, we tested them as a simulator for an UCT agent that played 200 games against a regular UCT agent (one that used the real game simulator), and achieved the same results as the real simulator for the game, with minor increase in the duration of rollouts.

4.2 State-Transition RND

In order to deal with inconsistent state-transitions generated by the learned model, our agent also trains an RND model during its training process. While previous work used this method for measuring the novelty of states to encourage exploration, our aim in this work is to do the opposite: measure the novelty of state-transitions produced by an approximated model when compared to state-transitions frequently observed in the real environment, and use this information to *discourage* novel state-transitions. Novel state-transitions, i.e. not seen during training, generated by the learned model, are likely to be incorrect samples, which do not exist in the real model.

Like our previous network, we also implement our RND model over the vectorized representation of states and actions, but instead of using state s and action a as inputs, we train our RND with action a and state delta $s_{\Delta} = s - s'$, where s' is the successor state. Thus, our RND model learns which actions were frequently seen during training and how they transform the game state, by means of s_{Δ}. This contrasts with the original RND technique, which measures the novelty of a state s, while our model need to detect the novelty of state-transitions, since an inconsistent transition often lead to a valid state s', i.e. s' should not be reachable from s, even if both states exist in the game.

Since this work is focused on board games, and most board games (like Hex) have discrete states, we developed a method for our RND to classify state-transitions as either valid or invalid. Algorithm 4.1 shows this method, where a is an action, s_{Δ} is a state delta, k is the distance threshold, $\overline{e}$ is the mean novelty value of state-transitions obtained during training with the real environment, $out_{predictor}$ and out_{target} are the output of the predictor and target networks, and e is the state-transition novelty value measured by the mean square error of $out_{predictor}$ and out_{target}, finally, the method classifies the transition as valid if the normalized distance between e and $\overline{e}$ is within threshold k.

Algorithm 4.1 – RND state-transition classifier

```
function IsTransitionValid( $a$, $s_{\Delta}$, $k$, $\overline{e}$ ) returns a boolean
    $out_{predictor} \leftarrow$ PredictorNet($a$, $s_{\Delta}$)
    $out_{target} \leftarrow$ TargetNet($a$, $s_{\Delta}$)
    $e \leftarrow$ MSE($out_{predictor}$, $out_{target}$)
    return $\frac{e - \overline{e}}{\overline{e}} <= k$
```

With this method, our RND model has better performance and sample efficiency when compared to a traditional classifier network of comparable complexity, and can learn effectively with very few training examples: in games of Hex, it correctly classifies all state-transitions from our test dataset containing 86994 valid and invalid examples after training with data of just 10 games played. In comparison, a traditional classifier achieves 75% accuracy against the test dataset when trained with the same training data, but with labeled transitions. These results were also obtained in the games of Othello and Checkers, with a more complicated network architecture and more training examples. Moreover, training an RND is simpler than a regular classifier, since the later must be trained over labeled data — and thus it would need to be trained with invalid states explicitly labeled as so, after comparing the output of the model the agent is learning with the output of the environment — while the RND model can learn just by *observing* a and s_{Δ} after each transition from the environment.

4.3 Addressing incorrect states in UCT

For this work, we experimented with two different modifications to the base UCT algorithm: tree pruned UCT (TP-UCT) and selective rollout UCT (SR-UCT). Both of them are adaptations of the base UCT algorithm (Algorithms 2.1 and 2.2) that take advantage of a trained RND state-transition classifier to bias its search to valid transitions.

The first one, TP-UCT, modifies UCT's tree structure, by flagging invalid nodes in the tree, and use such flag to bias node selection during the selection step of the UCT algorithm. Figure 4.2 illustrates a modified UCT expansion step in TP-UCT, where the algorithm

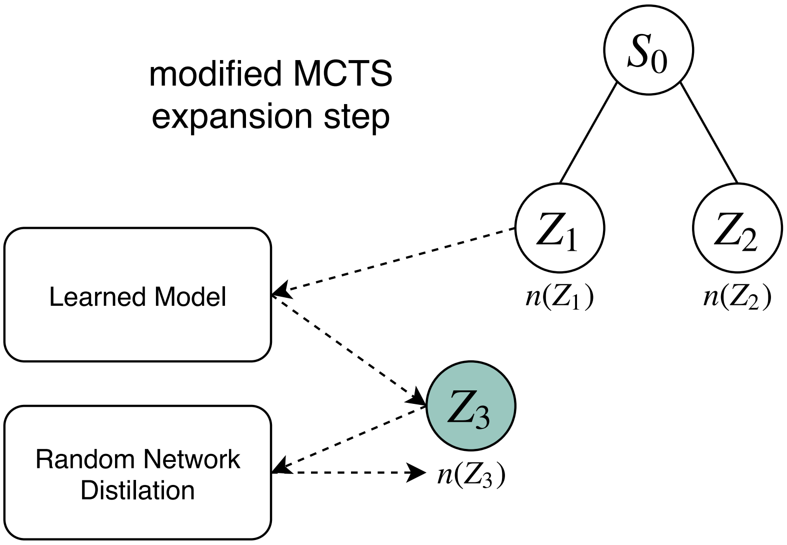

Figure 4.2 – UCT with a modified expansion step for computing novelty of states.

creates a new node with state Z_3 obtained from the learned model, and the validation flag $n(z_3)$ through the RND method. If the transition is valid, $n(z_3) = 1$ (or *True*), and $n(z_3) = 0$ (or *False*) otherwise. Then, if the recently expanded node is invalid, TP-UCT skips the rollout step from this node — since it would have been an invalid rollout from the start — and instead, calculates its reward with the mean Q-Value of all of its *siblings* (nodes at the same level of the tree), and then continues to the backup step, where this reward value is propagated upwards through the tree. Our intuition is that, in most cases, sibling states have similar reward values, and using this value is more informative than a randomly generated one or zero.

Finally, TP-UCT ignores invalid nodes during the selection step, since any rollout starting from that node is already invalid, we cannot be sure if it will reach a terminal state or not, and even if it does, the utility value of that state might not be accurate. The idea is to force the UCT algorithm to make *better* use of each rollout, without *wasting* rollouts on potentially uninformative trajectories.

Algorithm 4.2 – SR-UCT simulation policy

```
function SimulationPolicy( $s$ ) returns a utility value
    while TerminalTest( $s$ ) is False do
        $A \leftarrow$ actions $\in$ Actions( $s$ )
        for $a \in A$ do
            $s' \leftarrow$ Result( $s$ , $a$ )
            if IsTransitionValid( $a$, $s - s'$, $k$, $\overline{e}$ ) then
                $s \leftarrow s'$
                break
        if no state-transition was valid then
            return Utility($s$ , Player($s$))
    return Utility($s$ , Player($s$))
```

Our second UCT modification, SR-UCT, works by selecting valid state-transitions during UCT's rollout. The idea is that the agent should try to *steer* rollout trajectories towards accurate predictions of future outcomes, so that the rollout results are informative, i.e. they yield an accurate utility value. This agent does this by modifying UCT's simulation policy method, using our state-transition RND to classify each state-transition during the rollout. If there are no valid transitions left in the current trajectory, SR-UCT returns the utility value of the state from the last valid state-transition. Algorithm 4.2 shows this modified simulation policy, where A is a buffer that holds actions to try at each rollout step, while the other variables are the same as in Algorithm 2.1, Line 21.

5. RESULTS

In this chapter, we detail the experiments we carried out and their results. For these experiments, we considered the following UCT-based agents, with the exploration constant $c = 0.25$:

- *Regular-UCT*: Default UCT algorithm with a perfect Hex simulator.
- *A-UCT*: UCT algorithm with an approximate Hex simulator.
- *TP-UCT*: our UCT agent that prunes the tree during the expansion step.
- *SR-UCT*: our UCT agent that selects rollout trajectory during the simulation step.

First, we show the shortcomings of using an approximate model for UCT search, and how our agents address such shortcomings. Secondly, we compare the gameplay performance of TP-UCT and SR-UCT against an A-UCT agent. Finally, we compare the win rate of our agents that use approximate models against the Regular-UCT agent.

We tested these agents with various different settings in games of Hex with a 9×9 board, the same board size in the related work [Anthony et al., 2017] (Chapter 6), Othello in a traditional 8×8 board, and Checkers with a 8×8 board. Both TP-UCT and SR-UCT use our state-transition RND model to detect invalid state-transitions, with $k = 0$. This RND model has 100% accuracy against our test datasets for every game domain.

5.1 UCT performance with learned models

In order to simulate the effects of imperfect models, we trained our state-transition network with a reduced number of training examples during training, and measure its accuracy with our test datasets. There are various practical advantages of simulating this situation with our existing game implementations: we already have a viable network architecture and know how a good model performs. Our trained models are almost as fast as our game simulators, so our agents can make many *rollouts* per move, and we can get fast test results.

We generated 5 approximate models with increasing accuracy: 1%, 15%, 45%, 75%, and 100%. These accuracy values correspond the percentage of correct transitions that the approximate model made with data from our test datasets, which contained data from 100,000 matches for each game. We then used these models as transition-functions for the UCT search in various matches for each game to measure the percentage of invalid rollouts generated by each model. An invalid rollout is any rollout that contains at least one invalid state-transition. We classify state-transitions by comparing those made by the approximate model with the output of the game simulator, but, in case the simulator is not available, such procedure can also be done with the RND state-transition classifier. Figure 5.1 shows the results obtained in this experiment.

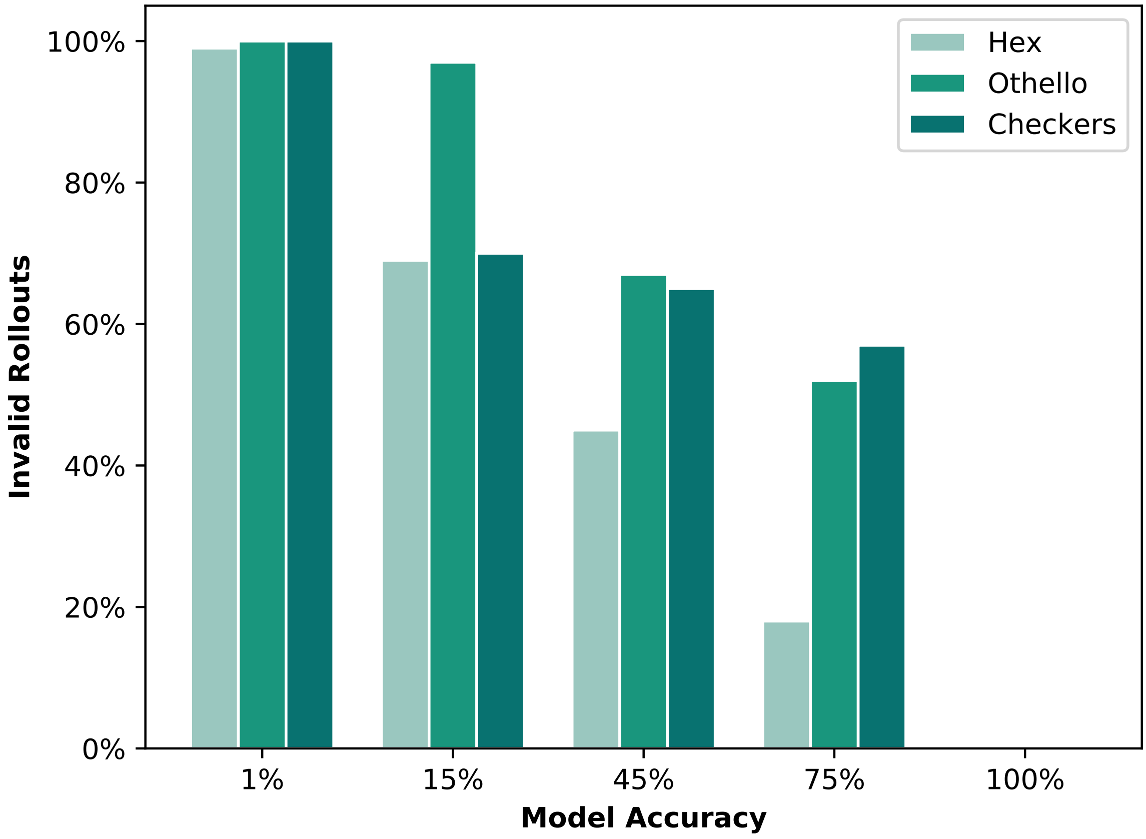

Figure 5.1 – the percentage of invalid *rollouts* of MCTS agents with varying model accuracy.

From this test, we found that the percentage of invalid rollouts varies between games even if their approximate models achieve the same accuracy. We believe that this is due to differences in the tree structure of each game: Hex's game tree has a large breadth, since after every move, all free positions on the board are available, so there are many possible trajectories to continue; Othello has fewer possible moves per ply, and has more possible states than Hex, because of piece capturing rules; Checkers has *sink states* in its tree, where only one move is available, because of the obligatory capture rule, so, if this sink transition is invalid, the entire rollout becomes invalid.

We found that invalid rollouts also contain a fatal flaw for planning algorithms: they might loop indefinitely, never reaching an end state. This means that during the simulation step of the tree search, our UCT algorithm can enter an infinite loop induced by the errors in the approximate model and the rollout never terminates. We overcome such limitation by doing truncated rollouts for our A-UCT and TP-UCT agents, i.e. rollouts that end before the actual end of a match and result in an estimated utility value. This impacts the performance of both agents, which are already hindered by the imperfections of the under-trained model. Similarly, our SR-UCT agent also truncates rollouts, but only if its selective strategy was unable to *correct* rollout trajectories, so it truncates less often than the other agents.

5.2 Comparison of TP-UCT and SR-UCT to A-UCT

Before we measure the gameplay performance of our agents against the Regular-UCT agent, we compare their performance against the A-UCT agent, so we can rank the agents that use approximate models for rollouts. For these experiments, all agents simulate the game using our approximate model with 45% accuracy against our test dataset.

Table 5.1 shows the results from our experiments, specifically, the win rate of each agent in different scenarios, with varying rollouts per UCT decision. These results show that TP-UCT actually performs worse than A-UCT in both Hex and Checkers, winning only 19.7% and 34.6% of the games respectively. We believe that this is because even if state-transitions are invalid, the model might retain some useful information of the previous state s in the successor state s' produced by the approximated model. By discouraging the exploration of every invalid transition, without measuring how much information was *lost* between s and s', this method fails to explore viable strategies during its search.

The opposite occurs in Othello, where TP-UCT wins 57.5% of the games, and we believe this is due to Othello states being more *fragile* to inconsistencies: a small change in a Othello state might not only invalidate it completely, it might also change the *lead* of the game, i.e. the player who is closer to victory. Moreover, the tree pruning gives the TP-UCT advantage in this game over the A-UCT since it *wastes* fewer rollouts in invalid rollouts, and at 45% accuracy, our Othello model still has a high percentage of invalid rollouts. This percentage is also high in Checkers, but its states are less prone to losing game information, even less so than in Hex, since pieces are few and scattered on the board.

Conversely, the SR-UCT show an expressive advantage over the A-UCT agent in Hex and Checkers. Such results shows that our selective rollout strategy is effective in keeping the rollout valid, and thus, avoiding truncated rollouts, that don't reach actual terminal states. By truncating fewer rollouts, SR-UCT is able to better predict the outcomes of its moves, and make decisions that are more competitive than those of the A-UCT agent in such games.

In all games, the RND based agent that has advantage over A-UCT (SR-UCT for Hex and Checkers and TP-UCT in Othello) achieves better result as the number of rollouts grows. These results suggest that both RND agents performance depends on the characteristic of the games they play. The RND solution achieved better results in Hex (specifically, SR-UCT) than in Othello and Checkers. We believe that this is due to the relation of model accuracy and invalid rollout percentage in the approximate models used for this game Hex: they are linearly correlated, unlike Othello and Checkers. From our experiments, we found that, even if model accuracy impacts the agent's performance less than invalid rollouts, with higher accuracy less information is lost when invalid transitions occur, and this gives the A-UCT algorithm an advantage, since more of its invalid rollouts may actually give the correct reward.

Table 5.1 – Win rate of agents against A-UCT agent, where all agents use an approximate model.

Game	Agent	Rollouts	Win Rate
Hex	TP-UCT	100	19.7% (± 4.59%)
Hex	SR-UCT	100	54.8% (± 5.74%)
Hex	SR-UCT	1000	65.1% (± 5.52%)
Hex	SR-UCT	2000	71.8% (± 5.19%)
Othello	TP-UCT	100	52.3% (± 6.92%)
Othello	TP-UCT	1000	57.5% (± 6.84%)
Othello	SR-UCT	100	33.4% (± 6.54%)
Checkers	TP-UCT	100	34.6% (± 6.59%)
Checkers	SR-UCT	100	53.2% (± 6.89%)
Checkers	SR-UCT	1000	58.0% (± 6.84%)

5.3 Gameplay performance against Regular-UCT

After ranking the UCT agents that use approximate models, we now compare them to the Regular-UCT agent, which uses the correct game simulator to compute its rollouts. For these comparisons, we focus on A-UCT and the RND agent that achieved best result in each game, SR-UCT for Hex and Checkers, and TP-UCT for Othello. Since the number of invalid rollouts grows significantly in models with poor accuracy (less than 15%), we do not consider such models in our experiments, as they perform significantly worse than the Regular-UCT agent. Similarly, we do not compare the win rate of agents that use models with 100% accuracy, since they do not produce invalid rollouts. All of the proximate model agents are expected to perform worse than Regular-UCT since they are not given a perfect simulator to compute their rollouts.

Table 5.2 shows the results from these experiments, where both agents are tested with varying model accuracy and rollouts per decision. Results corroborate our earlier experiments that show that model accuracy and invalid rollout percentage have a substantial impact on the performance of the UCT search. This performance gap grows with the number of rollouts per decision, since each Regular-UCT rollout is much more informative than a truncated or approximate rollout executed by RND agents and A-UCT.

These results also confirm RND agents superiority in relation to the A-UCT agent, and that this superiority is more pronounced with a *mediocre* model accuracy, specifically 45%. We believe this is due to the percentage of invalid rollouts at this accuracy value: SR-UCT selective rollout strategy has little effect on models with poor accuracy (15% or lower), since it might truncate frequently for not having valid transitions to select at the rollout step; conversely, when using a model with good accuracy value (75% or more), the same strategy is less effective because fewer rollouts truncate, so its gameplay performance is closer to that of the A-UCT agent.

Table 5.2 – Win rate of agents using approximate model against a Regular-UCT agent.

Game	Agent	Model Accuracy	Rollouts	Win Rate
Hex	A-UCT	15%	100	11.4% (± 3.69%)
Hex	A-UCT	15%	1000	4.3% (± 3.02%)
Hex	A-UCT	45%	100	24.7% (± 4.98%)
Hex	A-UCT	45%	1000	10.5% (± 3.54%)
Hex	A-UCT	75%	100	31.3% (± 5.35%)
Hex	A-UCT	75%	1000	24.4% (± 4.96%)
Hex	SR-UCT	15%	100	12.0% (± 4.02%)
Hex	SR-UCT	15%	1000	4.8% (± 3.18%)
Hex	SR-UCT	45%	100	32.2% (± 5.39%)
Hex	SR-UCT	45%	1000	19.1% (± 4.54%)
Hex	SR-UCT	75%	100	36.6% (± 5.56%)
Hex	SR-UCT	75%	1000	28.4% (± 5.20%)
Othello	A-UCT	15%	100	5.1% (± 3.05%)
Othello	A-UCT	15%	1000	2.0% (± 1.94%)
Othello	A-UCT	45%	100	7.7% (± 3.69%)
Othello	A-UCT	45%	1000	2.5% (± 2.16%)
Othello	A-UCT	75%	100	23.3% (± 5.86%)
Othello	A-UCT	75%	1000	11.8% (± 4.47%)
Othello	TP-UCT	15%	100	4.4% (± 2.84%)
Othello	TP-UCT	15%	1000	1.3% (± 1.56%)
Othello	TP-UCT	45%	100	10.9% (± 4.32%)
Othello	TP-UCT	45%	1000	6.6% (± 3.44%)
Othello	TP-UCT	75%	100	21.5% (± 5.69%)
Othello	TP-UCT	75%	1000	10.3% (± 4.21%)
Checkers	A-UCT	15%	100	1.6% (± 1.74%)
Checkers	A-UCT	15%	1000	0.4% (± 1.13%)
Checkers	A-UCT	45%	100	3.6% (± 2.58%)
Checkers	A-UCT	45%	1000	1.1% (± 1.45%)
Checkers	A-UCT	75%	100	5.3% (± 3.1%)
Checkers	A-UCT	75%	1000	2.5% (± 2.16%)
Checkers	SR-UCT	15%	100	1.2% (± 1.51%)
Checkers	SR-UCT	15%	1000	0.4% (± 1.13%)
Checkers	SR-UCT	45%	100	4.2% (± 2.78%)
Checkers	SR-UCT	45%	1000	1.4% (± 1.63%)
Checkers	SR-UCT	75%	100	4.9% (± 2.99%)
Checkers	SR-UCT	75%	1000	1.7% (± 1.79%)

6. RELATED WORK

In this chapter, we introduce work that is related to our research, namely, *expert iteration* methods, which are *state of the art* GGP implementations for board games, and approaches for learning environments models. Our work aims to be a *link* between these two subjects: *expert iteration* algorithms relies on environments models, which can be learned.

Lastly, we discuss the recent MuZero algorithm, which was introduced just as we were finishing our research, and aims to solve the same problem — but relies on a much more complex and computationally expensive solution than our own. Because this algorithm was introduced recently, and many of its details are still unclear, we cover it to the best of our current understanding of it.

6.1 Learning Environment Models

Learning a predictive model for the environment enables agents to simulate future scenarios, which can be used by planning techniques to yield better performance in a variety of RL problems when compared to model-free RL algorithms [Weber et al., 2017, Ha and Schmidhuber, 2018]. Learned models are also useful for encouraging exploration in RL tasks, and such approach has shown promising results for agents that play Atari games [Stadie et al., 2015]. These works differ from our approach since they learn a model over image observations [Weber et al., 2017, Ha and Schmidhuber, 2018], and aim at more complex environments.

A straightforward approach for learning models is to use function approximation methods, such as artificial neural networks. The agent explores the environment using a base policy to collect observations, and then fit those observations to the neural network. Most model-based RL agents learn models with this approach, however, this is an open area of research: for most environments, learned models yield high prediction error [Kaiser et al., 2019]. Our work focuses on investigating how these errors affect the agent's planning performance, and how agents can adapt their planning strategies accordingly. It does not aim to address the challenge of learning better models.

6.2 Expert Iteration

Expert iteration (ExIt) is a *state-of-the-art* algorithm for GGP agents that effectively combines game planning with MCTS and reinforcement learning in order to learn optimal policies for various board games. This algorithm was independently developed by two recent efforts: the Alpha Zero agent [Silver et al., 2017], which is capable of playing Chess, Go and Shogi better than human players and other agent implementations; and the work which gave the algorithm its name [Anthony et al., 2017], where an ExIt agent defeats the benchmark agent for the game of Hex (at the time of its publication), MoHex 1.0.

This algorithm learns optimal policies with the combination of two processes: approximating policies (via artificial neural networks) and MCTS planning. This combination is inspired by dual-process theory [St B. T. Evans, 2011], which states that humans think via two different processes: a fast, automatic process known as intuition, or System 1; and a slow, conscious process known as reasoning, or System 2. In ExIt, the policy learning neural network *plays the part* of System 1, while MCTS represents the slow and analytic System 2.

ExIt learns by interacting with an environment via *self-play* (playing against itself) in an RL fashion, improving both systems at each time-step: System 1 approximates learned strategies of System 2, while System 2 uses System 1's learned strategies for its planning. Figure 6.1 illustrates the loop the ExIt algorithm uses to find an optimal policy. The idea is that the agent learns intuition by planning, and then makes improved plans based on learned intuition. System 1 approximates the strategies of System 2 by a process known as imitation learning (IL) [Anthony et al., 2017]. Our approach, on the other hand, does not learn strategies over time, and instead, focuses on learning the environment's model. We intend to introduce similar strategy learning in future work.

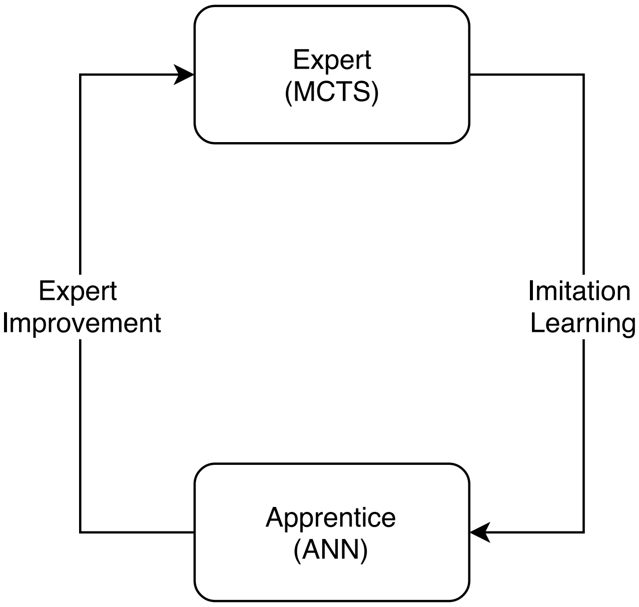

Figure 6.1 – a representation of the loop performed by the ExIt algorithm.

There are several approaches of imitation learning, the ExIt algorithm considers the following IL algorithm [Ross et al., 2010]: an agent, or *apprentice*, learns a policy from an *expert* with approximation methods, such as artificial neural networks. Algorithm 6.1 is a simplified *pseudo-code* for the ExIt algorithm, where S_i (Line 5) is a set of game states created by self-play with the apprentice policy $\hat{\pi}_{i-1}$, and D_i (Line 6) is a *dataset* that maps states in S_i to actions selected by the *expert policy* π^*, this *dataset* is then used by the *apprentice* to approximate the *expert policy*.

Algorithm 6.1 – *ExIt pseudo-code* [Anthony et al., 2017]

```
function ExIt
    $\hat{\pi}_0 \leftarrow$ InitialPolicy
    $\pi_0^* \leftarrow$ BuildExpert($\hat{\pi}_0$)
    for $i = 1; i \leq maxIterations; i++$ do
        $S_i \leftarrow$ SampleSelfPlay($\hat{\pi}_{i-1}$)
        $D_i \leftarrow \{$(s, ImmitationLearningTarget($\pi_{i-1}^*(s)$)) $|s \in S_i\}$
        $\hat{\pi}_i \leftarrow$ TrainPolicy($D_i$)
        $\pi_i^* \leftarrow$ BuildExpert($\hat{\pi}_i$)
```

Since the expert system is a planning algorithm (MCTS), which depends on the environment model for sampling states forward in the future, ExIt requires that the agent knows a perfect model for the environment. This contrast with our approach that tries to learn the rules of the games without using the real simulator. Thus, our approach does not intend to compete in gameplay performance with such works, but presents a more general approach to play board games in a way that requires limited interaction to learn the rules of the game (instead of having an accurate simulation), and use the resulting approximate model with traditional planning algorithms that account for the expected inaccuracies of the learned model.

6.3 MuZero

MuZero is a state-of-the-art algorithm for GGP that is a direct evolution of the Alpha Zero algorithm that, much like our own research, can play various games without any previous knowledge of the game's rules or access to a game simulator [Schrittwieser et al., 2019]. However, like Alpha Zero, MuZero learns better strategies over time via self-play, and can be viewed as a model-free ExIt algorithm. This algorithm was tested in 3 classical board games (Chess, Shogi and Go), and in a variety of Atari games — specifically in Atari games, MuZero set a new gameplay performance record for all games it was tested, achieving much better results than that of previous work. In order to approximate the game's rules and learn better strategies for it, MuZero trains a single model with three distinct components:

- Representation: from an actual observation o^t, the model generates a hidden state s^k through function *h*.
- Dynamics: a function *g* that produces an immediate reward r^k and a new hidden state s^k from hidden state s^{k-1} and action a^k.

- Prediction: a function f that computes a policy p^k and value function v^k from a hidden state s^k.

Figure 6.2 illustrates MuZero's model usage and training, and how each of its components work in both scenarios. During usage, the model receives an observation o^t, and transforms it into a hidden state s^0. From this hidden state, the algorithm performs a traditional MCTS search by using the results from the prediction function f (p and v) to select trajectories, and the dynamics function g to simulate them. After each episode, the episode's trajectory (the chain of observations, actions, and rewards) is saved in a replay buffer.

The model is then trained by sampling *real* trajectories from the replay buffer and *mimicking* them in hidden states: from an initial observation o^t, the model generates a hidden state s^0 and selects the same actions that were selected during the sampled trajectory, using the dynamics function g to simulate the outcome of each action. Then, the parameters of the representation (h), dynamics (g), and prediction (f) are jointly trained over three objectives: minimize the error between the trajectory policy Tp_t and predicted policy p_t^k; minimize the error between the trajectory value target Tz_t (the sum of discounted observed rewards Tu_t^k) and the predicted value function v_t^k; minimize the error between the observed reward Tu_t^k and the predicted reward k_t^k.

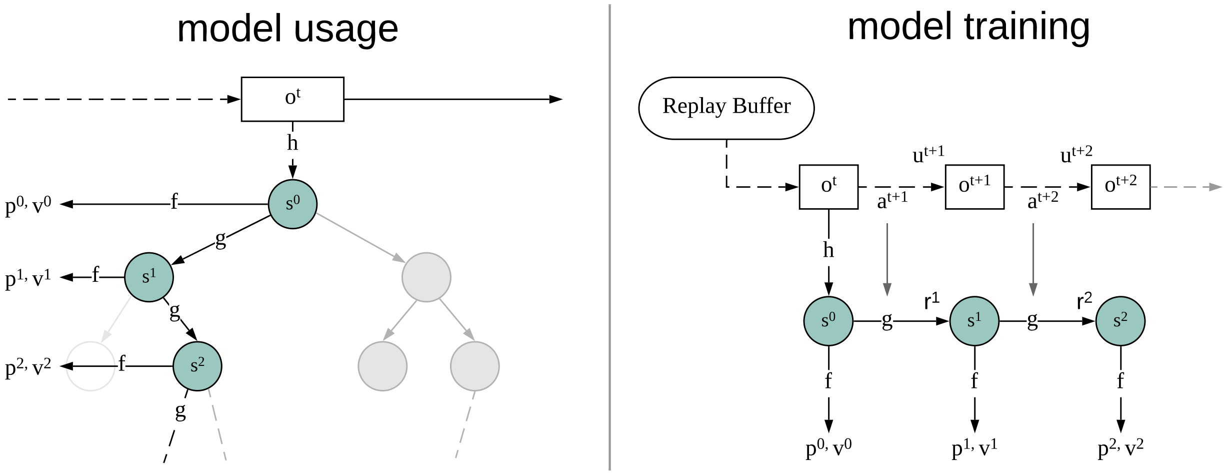

Figure 6.2 – an illustration of how the MuZero algorithm utilizes its approximated model for planning and how this model is trained [Schrittwieser et al., 2019].

The main difference from our approach is that MuZero relies on a much more complicated network structure and, consequently, requires much more computational *power* in order to be trained — especially considering its complete network architecture and settings, which are not totally clear yet, like Alpha Zero before it. Moreover, the algorithm's original implementation, like Alpha Zero, is not publicly available, and we could not find any third-party implementation of it at the time of writing this work. As such, we did not compare our results with that of MuZero, but we do believe that MuZero should be far superior than our approach, albeit it requires more computational *power*. Even with all dissimilarities between this work and our own, it provides more evidence of the importance and rising interest in the objective of our research: the development of AIs that are more generally applicable, and can perform without a model, i.e. without previous knowledge of the game’s rules.

7. CONCLUSION

We developed two methods for dealing with model imperfections during a UCT search with an approximate simulator: the tree pruning strategy of the TP-UCT agent, that bias the expansion of UCT's search tree towards valid state-transitions, and the selective rollout strategy of the SR-UCT agent, which selects actions at each rollout step in order to keep the rollout trajectory from reaching invalid state-transitions. The selective rollout strategy of the SR-UCT agent provides substantial improvements over a regular UCT agent that samples states through an approximate simulator in the game of Hex and Checkers, while the pruning strategy of TP-UCT provides similar improvements in the game of Othello.

Our empirical results show that, for environments where running the simulator is not possible or practical, RND agents can substantially outperform regular UCT for learning winning policies while coping with the errors from approximate models. As part of our analysis, we evaluate the impact of various levels of error (imperfections) in the model to UCT-based search. As shown in Table 5.2, the complexity of a game's rules had great impact in our agents performance, they obtained better results in Hex, followed by Othello, and performed poorly in Checkers. For Checkers, small model inaccuracies can have major impact in gameplay, and we believe this is due to that game's *sink* states, as discussed in Chapter 5.

While experimenting with different network architectures for each game, before defining the architecture presented in Chapter 4, we also found that this percentage of invalid rollouts differs between different networks, even if their accuracy is the same. Because of this result, networks should be trained with two objectives: minimize state-transition inaccuracy and minimize invalid rollout percentage. We intend to further investigate how to train better approximate models using such training objectives in future work. We argue that improving the performance of the model learning algorithm is an orthogonal objective to that of our work, which is to compensate for the inevitable inaccuracies of the learned model by reasoning about the nature of the transitions generated by it.

Although we did not find a general solution for dealing with imperfect models for all of the proposed games, our work contributes to a better understanding of the challenges of planning with such models. Furthermore, we present some conclusions that can guide future research on this subject: models should be trained with rollout accuracy as a main objective; state-transition accuracy is also important for keeping meaningful information between state-transitions, even if they are invalid; game characteristics have a major impact over search strategies; an agent's search can be improved with a state-transition classifier; and the RND network is not only a practical state-transition classifier — since it does not depend on labeled transitions — it is also much easier to train than a state-transition predictor.

As future work, we aim to apply our methods in more complicated environments and to expand its capabilities. First, we aim to evaluate our agents that use approximate models in board games with more complex gameplay rules. Second, we will evaluate the effect of learned models that include not only the gameplay rules, but also the preferences of opposing players [Silva et al., 2019]. Specifically, this is one of the key cases where using an actual simulator with the opposing player preferences is infeasible, so an approximate model trained with the history of past matches is an efficient replacement for actual play against the opponent. Third, we intend to investigate how imperfect models impact the performance of agents that can learn better strategies over time, such as HexIt, and how it performs with a similar pruning strategy of TP-UCT and the selective rollout strategy of SR-UCT. Fourth, we will evaluate our solution with other environments, such as Atari games and environments with continuous states. Finally, we will develop agents that learn not only the state-transition function, but also the ACTIONS and TERMINALTEST functions, and tries to plan using an approximated model for all of them.

During the course of this work, we submitted a paper presenting our results in the game of Hex, and — although it is not yet accepted, at the time of this writing — it was met with interest and valuable critic response. Specifically, we submitted our paper "General Game Playing with Learned Rules" in AAMAS 2020 (International Conference on Autonomous Agents and Multiagent Systems), and later, a new revision titled "Boardgame Playing with Learned Rules" to IJCAI 2020 (International Joint Conferences on Artificial Intelligence). To summarize: we conclude that our work is relevant, gives valuable insights on the challenges of achieving a general solution for learning rules and planning over them, and draw directions for future research on this subject.

REFERENCES

[Anthony et al., 2017] Anthony, T. Tian, Z. and Barber, D. (2017). Thinking Fast and Slow with Deep Learning and Tree Search. *CoRR*, abs/1705.08439.

[Browne et al., 2012] Browne, C. B. Powley, E. Whitehouse, D. Lucas, S. M. Cowling, P. I. Rohlfshagen, P. Tavener, S. Perez, D. Samothrakis, S. and Colton, S. (2012). A Survey of Monte Carlo Tree Search Methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(1):1–43.

[Burda et al., 2018] Burda, Y. Edwards, H. Storkey, A. J. and Klimov, O. (2018). Exploration by Random Network Distillation. *CoRR*, abs/1810.12894.

[Campbell et al., 2002] Campbell, M. Hoane, Jr., A. J. and Hsu, F.-h. (2002). Deep blue. *Artif. Intell.*, 134(1-2):57–83.

[Chaslot et al., 2008a] Chaslot, G. Bakkes, S. Szita, I. and Spronck, P. (2008a). Monte-carlo Tree Search: A New Framework for Game AI. In: *Proceedings of the Fourth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, AIIDE'08, pages 216–217. AAAI Press.

[Chaslot et al., 2008b] Chaslot, G. M. Winands, M. H. and Herik, H. J. (2008b). Parallel Monte-Carlo Tree Search. In: *Proceedings of the 6th International Conference on Computers and Games*, CG '08, pages 60–71, Berlin, Heidelberg. Springer-Verlag.

[de Lima et al., 2017] de Lima, G. d. A. R. Paz, B. F. and Meneguzzi, F. R. (2017). Optimizing uct for settlers of catan. In: *2017 16th Brazilian Symposium on Computer Games and Digital Entertainment (SBGames)*, pages 164–172. IEEE.

[Finnsson and Björnsson, 2008] Finnsson, H. and Björnsson, Y. (2008). Simulation-based approach to general game playing. In: *Proceedings of the 23rd National Conference on Artificial Intelligence - Volume 1*, AAAI'08, pages 259–264. AAAI Press.

[Genesereth et al., 2005] Genesereth, M. Love, N. and Pell, B. (2005). General game playing: Overview of the AAAI competition. *AI magazine*, 26(2):62.

[Ha and Schmidhuber, 2018] Ha, D. and Schmidhuber, J. (2018). World Models. *CoRR*, abs/1803.10122.

[Kaelbling et al., 1996] Kaelbling, L. P. Littman, M. L. and Moore, A. P. (1996). Reinforcement Learning: A Survey. *Journal of Artificial Intelligence Research*, 4:237–285.

[Kaiser et al., 2019] Kaiser, L. Babaeizadeh, M. Milos, P. Osinski, B. Campbell, R. H. Czechowski, K. Erhan, D. Finn, C. Kozakowski, P. Levine, S. Sepassi, R. Tucker, G. and Michalewski, H. (2019). Model-based reinforcement learning for atari.

[Katz et al., 2017] Katz, M. Lipovetzky, N. Moshkovich, D. and Tuisov, A. (2017). Adapting Novelty to Classical Planning as Heuristic Search. In: *Proceedings of the Twenty-Seventh International Conference on Automated Planning and Scheduling, ICAPS 2017*, pages 172–180.

[Moravcík et al., 2017] Moravcík, M. Schmid, M. Burch, N. Lisý, V. Morrill, D. Bard, N. Davis, T. Waugh, K. Johanson, M. and Bowling, M. H. (2017). DeepStack: Expert-Level Artificial Intelligence in No-Limit Poker. *CoRR*, abs/1701.01724.

[Ramanujan et al., 2011] Ramanujan, R. Sabharwal, A. and Selman, B. (2011). On the Behavior of UCT in Synthetic Search Spaces. In: *Proc. 21st Int. Conf. Automat. Plan. Sched., Freiburg, Germany*.

[Ross et al., 2010] Ross, S. Gordon, G. J. and Bagnell, J. A. (2010). No-Regret Reductions for Imitation Learning and Structured Prediction. *CoRR*, abs/1011.0686.

[Russell and Norvig, 2010] Russell, S. J. and Norvig, P. (2010). *Artificial Intelligence: A Modern Approach (Third Edition)*. Prentice Hall, Upper Saddle River, New Jersey, USA.

[Schrittwieser et al., 2019] Schrittwieser, J. Antonoglou, I. Hubert, T. Simonyan, K. Sifre, L. Schmitt, S. Guez, A. Lockhart, E. Hassabis, D. Graepel, T. Lillicrap, T. P. and Silver, D. (2019). Mastering atari, go, chess and shogi by planning with a learned model. *ArXiv*, abs/1911.08265.

[Silva et al., 2019] Silva, F. L. D. Costa, A. H. R. and Stone, P. (2019). Building self-play curricula online by playing with expert agents in adversarial games. In: *Proceedings of the 8th Brazilian Conference on Intelligent Systems (BRACIS)*.

[Silver et al., 2016] Silver, D. Huang, A. Maddison, C. J. Guez, A. Sifre, L. Van Den Driessche, G. Schrittwieser, J. Antonoglou, I. Panneershelvam, V. Lanctot, M. et al. (2016). Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587):484.

[Silver et al., 2017] Silver, D. Hubert, T. Schrittwieser, J. Antonoglou, I. Lai, M. Guez, A. Lanctot, M. Sifre, L. Kumaran, D. Graepel, T. Lillicrap, T. P. Simonyan, K. and Hassabis, D. (2017). Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm. *CoRR*, abs/1712.01815.

[St B. T. Evans, 2011] St B. T. Evans, J. (2011). Heuristic and Analytic Processes in Reasoning. *British Journal of Psychology*, 75:451 – 468.

[Stadie et al., 2015] Stadie, B. C. Levine, S. and Abbeel, P. (2015). Incentivizing Exploration In Reinforcement Learning With Deep Predictive Models. *CoRR*, abs/1507.00814.

[Sutton and Barto, 2018] Sutton, R. S. and Barto, A. G. (2018). *Reinforcement Learning: An Introduction*. The MIT Press, second edition.

[Tan et al., 2005] Tan, P.-N. Steinbach, M. and Kumar, V. (2005). *Introduction to Data Mining*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, first edition.

[Weber et al., 2017] Weber, T. Racanière, S. Reichert, D. P. Buesing, L. Guez, A. Rezende, D. J. Badia, A. P. Vinyals, O. Heess, N. Li, Y. Pascanu, R. Battaglia, P. Silver, D. and Wierstra, D. (2017). Imagination-augmented agents for deep reinforcement learning. *CoRR*, abs/1707.06203.