Learning to Play Monopoly with Monte Carlo Tree Search

Siim Sammul

4th Year Project Report
Computer Science
School of Informatics
University of Edinburgh

2018

Abstract

Monte Carlo Tree Search is a common algorithm for learning optimal policies in a variety of decision and game problems. It is especially suited for tasks with vast state and action spaces, since in these scenarios exact inference methods can become intractable. However, one major issue of Monte Carlo Tree Search often occurs in games where the cardinality of different types of actions is not balanced: the types of actions with fewer possible options, including actions which are essential to advance the game (like ending one's turn) do not get sampled often enough. This results in long and potentially cyclic simulations and the performance of the algorithm will degrade. Dobre et al. (2017) introduced a way to address this issue for the game of Settlers of Catan. Their idea is to sample actions in two stages: first the action type and then the specific action itself from actions of that type.

This project aims to investigate whether the approach they used for learning Settlers of Catan can successfully be applied to other gaming environments: here, I examine the performance of this approach on learning the board game Monopoly. I train two agents: one that samples actions directly and one that samples actions in two steps - first the type of the action and then the specific action from actions of that type. I show that encoding domain knowledge by sub-grouping possible actions to different types and then sampling in two layers is also beneficial in the game of Monopoly and speeds up learning.

Contents

Chapter 1

Introduction

The field of reinforcement learning deals with learning optimal policies. It combines planning and learning - finding the sequence of actions that maximises the expected reward of a known model and learning the model from interaction with the environment (Sutton et al., 1998). This can be viewed as an agent learning which actions it needs to take to maximise its expected reward over time. The reward can be different for various problems: it could be some accumulated score, negative cost, or even just whether the agent won or lost.

Reinforcement learning is an exact inference method, and consequently tractable only for domains of a limited size. One popular approximate inference alternative, which exploits sampling, is Monte Carlo Tree Search (MCTS) (Coulom, 2006). MCTS is a popular method for learning optimal policies "online": in other words, one attempts to compute the best next action in a given state only when one experiences that state, rather than learning the entire policy function, for all possible states, in advance of acting in the world.

While the basic MCTS algorithm works well in many scenarios, in some complex tasks various challenges can arise. One issue that can occur in many domains is the curse of history - the problem scales exponentially as the length of the planning horizon increases. A second challenge is the curse of dimensionality - the simulation trees have a huge branching factor, making the simulation massive (Pineau et al., 2006).

In many games, there is a single action like ending a turn that is necessary to progress but which does not offer a noticeable immediate benefit. This feature, combined with the large cardinality of the other types of actions, can introduce both of the previously mentioned problems for the Monte Carlo Tree Search algorithm as the learner will not sample the turn ending move often enough. This can create very long simulations, reducing the efficiency and feasibility of the learning process. Games like Monopoly (Darrow, 1935) or Settlers of Catan (Teuber, 2007), which allow the players to build and trade in a multitude of ways are such games. For example, the rules of Monopoly allow a player to keep offering trades without ever ending their turn. And as trade offers are on average 94% of the legal actions (while ending the turn is only 0.007%), uniform sampling of moves will result in the agent constantly trading.

In some cases it is possible to utilise knowledge of the task and its characteristics to group actions in a meaningful way, effectively creating sub-goals and converting the problem to a hierarchical one. However, such grouping cannot be arbitrary - the moves in a group need to be correlated (Childs et al., 2008a) and thus it is hard to find a general method for categorising the actions. In the domain of games, it is common that there are well-defined rules, which specify the kinds of moves that are possible in the game - inherently grouping the possible actions by design. Dobre et al. (2017) made use of this property for the game of Catan and turned the sampling process into a hierarchical one: they group actions by types inherent to the rules of the game, sample an action type from those available, and only then sample the specific action. This reduces the number of possible actions and helps avoid cyclic behaviour - thus addressing both the dimensionality problem and the simulation length problem.

This project focuses on using Monte Carlo Tree Search to learn the classic board game Monopoly. I take the approach used for Settlers of Catan by Dobre et al. (2017) and apply the same ideas to Monopoly, which also has clear action types defined by its rules. On average, there are around 150 legal actions in Monopoly, approximately 140 of which are trading actions. The amount of moves one can perform per turn is not limited, yet there is only one action that ends a players turn. Thus, if one were sample according to a uniform distribution over the actions, they would hardly explore the consequences of ending their turn. This makes Monopoly an excellent candidate for trying typed sampling.

I use MCTS to train two agents: one that samples actions without looking at the types, and one that picks moves in two layers - first the type of the action and then the specific action itself. By comparing these two agents I can show that the typed MCTS approach is applicable and beneficial not just in the game of Settlers but also in Monopoly - effectively showing that this approach of grouping actions by types inherent to the environment rules or descriptions can generalise to different problems and improve learning efficiency.

1.1 Contributions

A summary of my contributions:

- I implemented a fully functional version of the game of Monopoly in Java from scratch. It supports the evaluation of agents via a testing environment where relative performance is measured by simulating several thousand games to counteract the significant part that chance plays in Monopoly (Section 3.1.1).
- I designed and implemented four hand-crafted heuristics-based baseline agents: a random agent, an aggressive agent, a cautious agent, and an agent described by Bailis et al. (2014) (Section 3.4).
- I integrated the Monte Carlo Tree Search code (designed for Settlers of Catan) provided by Dobre et al. (2017) to work with my implementation of Monopoly (Section 3.1.2).

- I trained and evaluated two agents with MCTS against the baseline agents and each other: one which samples actions in two steps - first type and then action, and one that samples moves uniformly (Section 3.5).
- I analysed and compared the performance, speed, and effectiveness of typed and uniform sampling (Chapter 4 and Section 5.1.3).

1.2 Overview

The report is structured as follows:

- Chapter 2 of the report presents a brief background and literature review of the covered topics: the game of Monopoly, relevant reinforcement learning algorithms, and a summary of related work.
- Chapter 3 explains the setup of the system - game and learning environment, the design of baseline agents, and the methodology used to prepare and carry out experiments.
- Chapter 4 details the results of the various experiments performed.
- Chapter 5 features a discussion of those results and findings, a review of the successes and shortfalls of the methodology, and some ideas for future work.
- Chapter 6 draws conclusions from the experiments and the project in general.

Chapter 2

Background

2.1 Monopoly

Monopoly [1] is a board game where players pose as landowners to buy and develop a set of properties. The goal of the game is to be the last player remaining by forcing every other player to go bankrupt and thus achieving a monopoly in the real estate market. It is one of the most played board games in the world with over 500 million players worldwide (Guiness World Records, 1999). Monopoly has a vast state space, and the players need to make numerous decisions at every turn. It also has some randomness as the players roll dice and occasionally take cards that have different effects. Thus it is an exciting game to study.

Slight variations exist in the rules of the game in different countries, and in reality, most people do not play according to the official rules - they either do not follow every detail or just prefer to make their own adjustments (BoardGameGeek, n.d.). Thus some confusion can arise when different players expect different rules, and it is entirely possible that gameplay will be affected. To avoid any confusion, my project is based on the UK version of the game and the corresponding rules (Waddingtons Games, 1996) with some explicitly stated simplifications (Section 3.2.1).

2.1.1 Board

The game board (Figure 2.1) consists of 40 squares, each belonging to one of 7 categories: start square, tax square, property square, card square, jail square, go to jail square, and free parking square. There are 28 properties that players can buy and trade, two tax squares which have the player pay a fee, and six card squares that have the player take either a community chest or a chance card depending on the specific square.

[1] The official rules of Monopoly: `http://www.tmk.edu.ee/~creature/monopoly/download/official_rules_gathering/instructions.pdf`

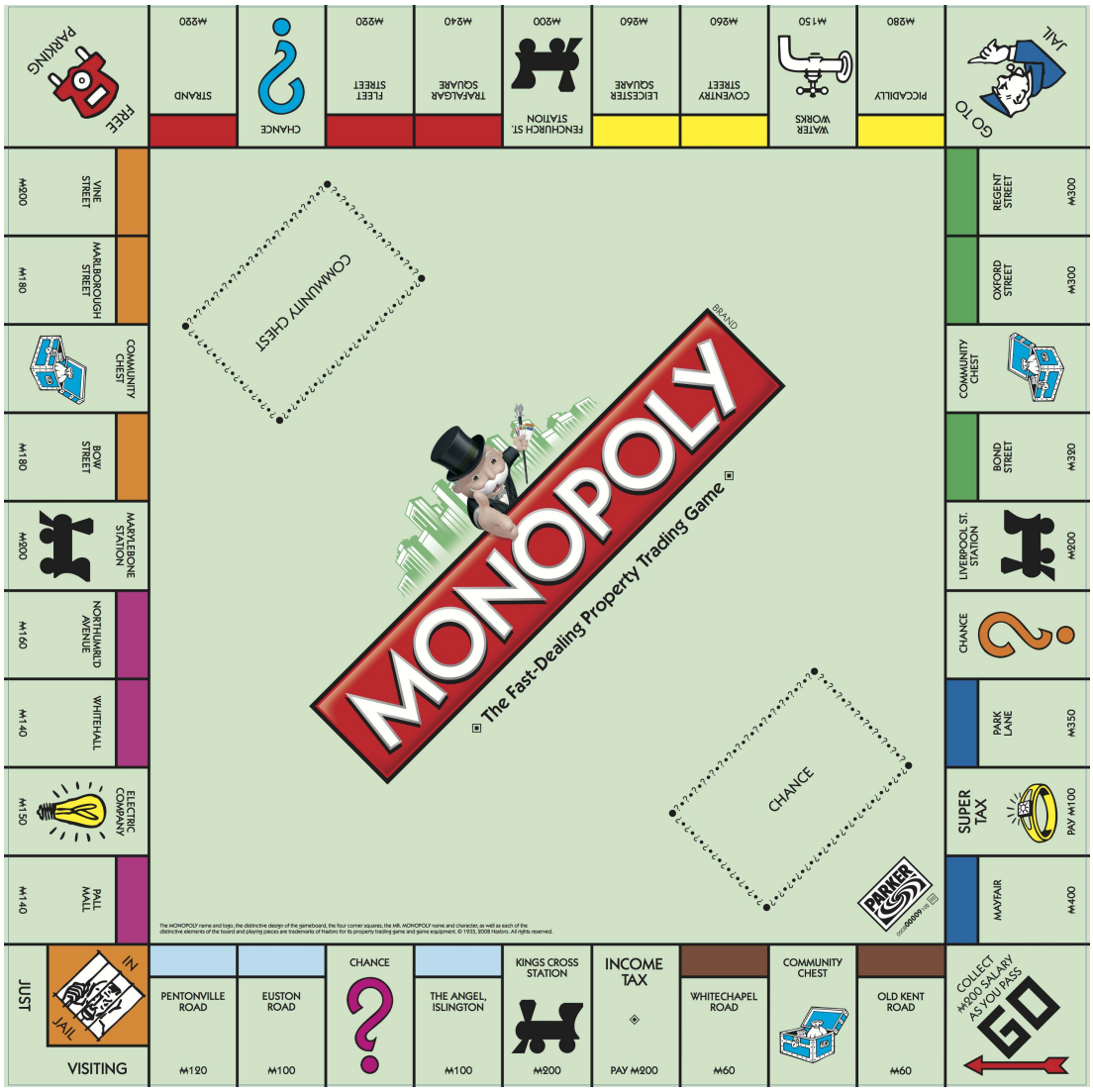

Figure 2.1: Monopoly board (Daniel Sheppardl, 2015b)

2.1.2 Gameplay

Monopoly is a turn-based game, where players move by rolling two six-sided dice and then act according to the square they land on. Stepping on a property that is owned by the bank gives the player the right to buy that property. Arriving on a property owned by another player requires the turn player to pay rent to the owner of the property. Players can build houses on the properties of which they hold a full set (all the properties of that colour) and thus increase the rent on the property by a significant factor. Therefore it is beneficial to obtain sets of properties, which players can do by either getting lucky with dice rolls and making the right purchase decisions or by trading properties with the other players.

Rolling doubles (two of the same number) gives the player an extra consecutive turn. However, if a player rolls doubles three times in a row they are immediately moved to

jail. The jail square cannot be left unless the player pays a fee of £50, uses a ”get out of jail free” card, or rolls doubles at their next turn. This does not apply if the player is not actually jailed, and is rather just visiting the jail, which happens if they only stepped on the jail square.

Money is paid out from the bank every time a player passes or steps on the start square. Additionally, some chance and community chest cards award the player some currency. However, the cards can also make the player pay various fees. Should a player not have enough money to pay a fee, they have to either sell some houses that they own or mortgage one of their properties. If the right amount of money has not been found even after selling all houses and mortgaging all properties, the player goes bankrupt and will be removed from the game.

2.1.3 Related Games

The game of Monopoly has some defining features that it is known for and which are the reasons people keep playing it: the basic rules are simple and rely partially on luck due to dice rolls and chance cards, however, players still have a lot of decisions to make, thus making the game somewhat but not entirely skill based. Players need to be sensible in the use of money so they could buy and develop properties and afford to pay rent. Finally, the ability to trade provides almost endless options to improve one’s position should they manage to negotiate a deal with the other players. From a game theory standpoint, Monopoly is non-deterministic as it has chance events - players need to roll dice to progress. It is also a game of imperfect information - the order of the chance and community chest cards is unknown, and incomplete information - the opponent strategies are not known to the player.

Many games share some of those characteristics with Monopoly and thus are in some ways very similar. The aforementioned Settlers of Catan requires players to roll dice to get resources, which they can trade between players to build roads and cities. Genoa (Dorn, 2001) and Bohnanza (Rosenberg, 1997) also have a heavy focus on trading and negotiation, Stone Age (Tummelhofer, 2008) and Kingsburg (Iennaco, 2007) rely on dice rolls and require the players to cleverly allocate resources to progress.

There are two main reasons I am covering Monopoly in this project and not some other game. Firstly, the defining attributes of Monopoly are quite close to those of Settlers of Catan and precisely those that are of interest for studying the effects of the typed sampling approach of Monte Carlo Tree Search. Secondly, Monopoly is a very known game, and thus most people reading this report will have either played or at least heard of the game, making it more intuitive and interesting to discuss the various aspects of learning to play the game.

2.2 Reinforcement Learning

Reinforcement learning is an area of machine learning that focuses on learning optimal policies in sequential decision-making problems (Sutton et al., 1998). In a typical setting, an agent earns rewards in response to acting in an environment. The rewards could be associated with particular states or actions or a combination of both. At each step, the agent will get an observation of the environment and will, in turn, choose and execute an action based on its policy. The observation could be the actual state, in which case the problem is fully observable. If part of the state is hidden from the agent, the problem is said to be partially observable.

There are multiple ways to learn a good policy. It is possible to learn a value function - an estimate of the expected reward from each state (s). The Bellman optimality equation (Equation 2.1) desribes the optimal value function V^* (Bellman, 1957). $P^a_{ss'}$ is the probability of action a taken in state s, resulting in state s'. $R^a_{ss'}$ is the resulting reward and γ is the discount factor for future reward.

$$V^*(s) = \max_a \sum_{s'} P^a_{ss'}[R^a_{ss'} + \gamma V^*(s')] \tag{2.1}$$

This value function could then be used to find the best policy, should the transition functions be known: for each state choose the action that results in the state of the highest value. It is also possible to learn a policy directly: the first step is to choose an arbitrary policy (π), then that policy should be evaluated by finding the expected reward from each state (Equation 2.2). Now for each state, the new best action should be chosen. This method is called Policy Iteration (Bertsekas, 1995).

$$V^\pi(s) = \sum_a \pi(s,a) \sum_{s'} P^a_{ss'}[R^a_{ss'} + \gamma V^\pi(s')] \tag{2.2}$$

Q-learning (Watkins et al., 1992) and various other reinforcement learning methods estimate the action-value function (Q-function) instead. The Q-function (Equation 2.3) is similar to the state-value function except it indicates the expected value of taking the specific action a from the state s. Explicitly indicating the action is beneficial as it removes the need to know the transition function of the environment. However, it comes at the cost of increased dimensionality of the value function (Sutton et al., 1998).

$$Q^\pi(s,a) = \sum_{s'} P^a_{ss'}[R^a_{ss'} + \gamma V^\pi(s')] \tag{2.3}$$

An essential factor in reinforcement learning is the balance of exploration and exploitation: it is important to visit all states of the environment to get a good overview of the world, but it is also important to do well and earn rewards. One way of doing this is to use an off-policy method, where at the training phase the agent prioritises exploring

the world, but then in the playing stage uses a greedy policy. In contrast, with an on-policy method the agent would use a single policy to achieve both goals. For example, epsilon-greedy - at each step the agent takes a random action with a certain probability and otherwise chooses the best action (Sutton et al., 1998). It is also possible to decay epsilon as learning proceeds so that in the later stages of learning the agent focuses more on performing actions it deems optimal rather than exploring the domain.

2.2.1 Markov Decision Process

Markov Decision Process (MDP) (Puterman, 1994) is a stochastic process with states and actions that affect the environment. It satisfies the Markov Property - the outcome (the resulting state and reward) of an action depends only on the current state and action and not the history of previous states or actions (Equation 2.4). This is useful for reinforcement learning, as decisions can be based on the current state, which is a lot more feasible than remembering the whole sequence of gameplay.

$$P(s_{t+1}, r_{t+1}|s_0, a_0, s_1, a_1, ..., s_t, a_t) = P(s_{t+1}, r_{t+1}|s_t, a_t) \tag{2.4}$$

MDPs are very popular for describing learning and optimisation problems. If the transition function is known it is possible to do model-based learning - the environment definition can be used to calculate the useful policies without actually executing actions in the system. In model-free learning, the state transitions or their probabilities are unknown and therefore cannot be used. In these problems, it is effective to use methods like Monte Carlo, which do not assume specific knowledge of the domain and learn from their own past decisions.

2.2.2 Monte Carlo Tree Search

I will use Monte Carlo Tree Search (Coulom, 2006) to learn to play Monopoly. MCTS exploits sampling by applying Monte Carlo estimation (Equation 2.5: V_s is the value of the state, w_s is the number of wins from that state, and n_s is the number of visits to that state) to tree search - a number of games are simulated, and for each visited state the end result of that game is stored. Then win rates from each state - the approximate values of those states - can be calculated.

$$V_s = \frac{w_s}{n_s} \tag{2.5}$$

MCTS is used widely in games with perfect information, most notably AlphaGo (DeepMind, 2017) which beat the world Go champion Lee Sedol in 2016. However, variations of the method like Imperfect Information Monte Carlo (IIMC) (Furtak et al., 2013) can be used to solve imperfect information games effectively. Monte Carlo Tree Search is a compelling choice for games and problems with vast state spaces as it does not have to do a full search of the game tree. A new simulation can be run for each new

choice, and the simulation can be stopped at any point, be that after a specified number of rollouts or a certain time. MCTS is also useful for stochastic games as getting several thousand or even tens of thousands of samples will yield relatively accurate estimates of the true value function.

2.2.3 Hierarchical Planning

Hierarchical planning is the act of dividing a complex task into smaller ones by defining sub-goals, each of which builds up to the bigger goal. The first step should be to abstract the task, and then solve that abstract version of the problem (Yang, 1997). The solution can be used to identify and complete the next sub-goal. By addressing the situation piece-by-piece, one can progress and eventually finish the whole task in a manner where each step is of a manageable size. Such abstractions have been successfully used in variations of the card game Poker (Shi et al., 2000).

In Monte Carlo Tree Search, hierarchical planning can be used to combat the curse of dimensionality - deal with the large branching factor. The technique of grouping the possible actions the agent can take into several categories so that actions in each group are correlated is called Move Groups (Childs et al., 2008b). This approach allows for actions to be sampled in two stages in MCTS - first the group is selected and then the specific move from that group. Since the actions in each group are correlated, it is likely that they have similar effects. Therefore, sampling in two stages reduces branching, as the impact of the second choice can be "predicted" from the first choice, and there are less possible choices in both stages of the process.

2.3 Related Work

2.3.1 Reinforcement Learning

Although reinforcement learning has been a topic of interest since Samuel (1959), a lot of the more interesting success stories have happened in the more recent years. In the gaming field, DeepMind used deep reinforcement learning to learn to play ATARI games, beating human experts in several of the classic games (Mnih et al., 2013). They were also immensely successful in the game of Go, first beating the world champion Lee Sedol (DeepMind, 2017) with a mixture of supervised learning from human experts and reinforcement learning from self-play (Silver et al., 2016), and then improving even more by entirely learning from self-play to create AlphaGo Zero (Silver et al., 2017). OpenAI (2017) just recently beat a professional player at Dota 2 (Valve Corporation, 2013), which is currently one of the most popular real-time competitive eSports. However, this attempt was limited to 1v1 gameplay, which is not common for Dota - the game is usually played in a 5v5 setting, adding a magnitude of more complexity and the need for cooperation. Now Facebook, DeepMind and others are trying to develop agents that could learn to play StarCraft (Blizzard Entertainment, 1998) and

StarCraft II (Blizzard Entertainment, 2010), which are more complex due to the imperfect information and the real-time nature of the games, in addition to the massive state and action spaces, similar in size to Go (Vinyals et al., 2017). These attempts have not yet managed to reach levels competitive for humans, but tournaments are held where the various bots play against each other (Simonite, 2017). However, in the game of Super Smash Bros Melee (HAL Laboratory, 2001), which is also challenging due to the game being real-time and only partially observable, deep reinforcement learning agents have already beat the experts (Firoiu et al., 2017).

Additionally, in the field of robotics, Ng et al. (2006) and Abbeel et al. (2010) taught a model helicopter extremely complicated manoeuvres, including inverted flight and various flips. Several frameworks have been proposed (Sallab et al., 2017) and attempts have been made to apply reinforcement learning to autonomous driving (Shalev-Shwartz et al., 2016; You et al., 2017), which has become a controversial topic as companies like Tesla, Uber and Alphabet compete to produce the first road-worthy fully autonomous vehicle. In finance, reinforcement learning has been used for portfolio management (Jiang et al., 2017) and automated trading (Bertoluzzo et al., 2014). In neuroscience, neurotransmitters have been researched and modelled using reinforcement learning methods (Balasubramani et al., 2014). In medicine, reinforcement learning can help segment medical images (Sahba et al., 2006) and find optimal treatment regimens (Zhao et al., 2011).

2.3.2 Monte Carlo Tree Search

Monte Carlo Tree Search is a popular reinforcement learning algorithm due to its simplicity and the ability to handle large search spaces and incomplete information. It has seen a lot of application in games, most notably in the game of Go, where numerous attempts have been made using MCTS (Chaslot et al., 2008; Enzenberger et al., 2010; Gelly et al., 2011), the most successful so far is the aforementioned AlphaGo, which used a combination of Monte Carlo Tree Search and deep neural networks. Other board games tackled with MCTS are Backgammon (Van Lishout et al., 2007), Connect6 (Yen et al., 2011), Havannah (Lorentz, 2010), and Hex (Arneson et al., 2010). The algorithm has also been relatively successful in various card games: the common casino game Poker (Ponsen et al., 2010), one of the most common Chinese card games Dou Di Zhu (Whitehouse et al., 2011), and the long-popular collectible card game Magic: The Gathering (Cowling et al., 2012). Video game applications are also plentiful: from learning to avoid ghosts in the classic Ms. Pac-Man (Pepels et al., 2014), to bringing MCTS to AAA titles by using Monte Carlo Tree Search to choose the campaign AI actions in the successful strategy game Total War: Rome II (Champandard, 2014; The Creative Assembly, 2013).

However, the use of the MCTS algorithm is not limited to games; it is successful in other fields as well. It has been used to optimize bus wait times (Cazenave et al., 2009) and print job scheduling (Matsumoto et al., 2010). The Travelling Salesman Problem has been addressed using Monte Carlo Tree Search with Time Windows (Rimmel et al., 2011) and macro-actions (Powley et al., 2012). And even attempts at using MCTS

to evaluate security have been made (Tanabe et al., 2009).

2.3.3 Monopoly

Rather surprisingly not much work has been done in using machine learning methods to play Monopoly. There have, however, been attempts at modelling the game as a Markov Decision Process (Ash et al., 1972) and doing statistical analysis of the game states (Daniel Sheppardl, 2015a). These mathematical approaches give some insight into the progression of the game and the potential characteristics or features that decisions can be based on, which could be used to simplify or add structure to the machine learning process. However, using machine learning to play the game is still a different problem and might have to be approached differently.

Bailis et al. (2014) tackled a simplified version of Monopoly with Q-Learning. Their approach was to come up with an equation based on the number of players and their current monetary and asset values and use it to evaluate game states by estimating the reward. They then trained a model against random agents and a baseline of their design. However, their approach, while modelled as an MDP, is restricted to a very simplified version of the game, in which no trading is allowed. Not supporting trading removes a major chunk of possible actions and greatly reduces the complexity of the game. Furthermore, estimating the reward with a simple equation is not entirely accurate, as there are various factors which make up a state and how good it is. I suspect that Q-learning in an MDP would not scale effectively to the full version of Monopoly, although confirming this is an open empirical matter.

As the goals of participating parties do not usually align when trading in Monopoly, studying strategic negotiation can be beneficial to gain an advantage over opponents. For the game of Settlers of Catan, Guhe et al. (2012) have studied conversation strategies for effective trading of resources. Additionally, some work has been done in trying to learn such interactive trading between players with reinforcement learning in various toy problems and games (Hiraoka et al., 2015). While the game of study has not been Monopoly, the concept of negotiating in trades definitely applies to Monopoly as well. Thus making use of the knowledge learned from other games could improve agents that play Monopoly, as proposing smart offers and counteroffers to potential trades will most likely lead to more successful exchanges of properties. However, as I state later, in this project haggling in trades will not be allowed as it further complicates the game and extended dialogues do not offer much concerning the research question. Instead, trades will work on a proposal basis: the turn player has a chance to propose a trade which the other player can either accept or reject (they can propose another trade after getting a response).

Chapter 3

Experimental Design

3.1 Resources

3.1.1 Gaming Environment

There was no suitable open source game environment for Monopoly, so I developed one myself. As stated previously, I based my implementation on the official rules of the UK version of the game. I used Java for the game to be compatible with the MCTS model code (Section 3.1.2). The codebase design follows an object-oriented pattern to make it easily configurable and extensible. The structure of the game implementation can be seen from Figure 3.1: the main game class creates a board (which holds data about all of the squares and properties) and the players, each of which has an associated agent specified by the game configuration. Since the design is object-oriented, each type of property (train stations, company squares) is merely a subclass of the main property class and no special treatment is needed for the different types. Similarly, the player strategies can easily be changed by merely swapping the agents. Thus, it is straightforward to test the performance of one player type against another.

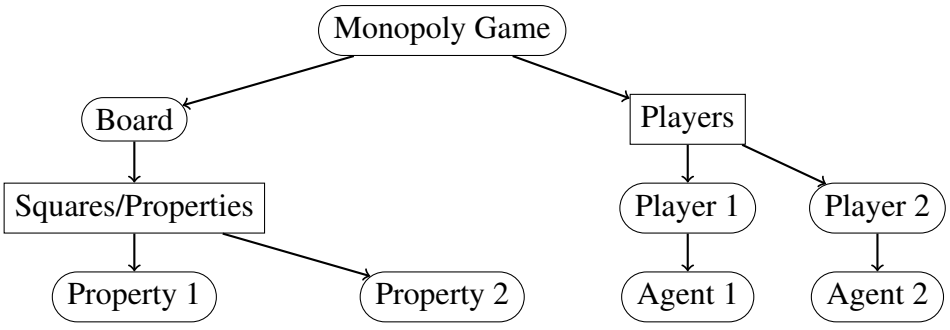

Figure 3.1: Monopoly implementation structure

The gaming environment can support gameplay between any combination and number of agents, however, in this project, 4 agents are used in all experiments. Technically even humans could play, but since there is no graphical user interface, it would be extremely cumbersome. The environment supports simulating games of Monopoly, measuring win rates, and calculating other statistics over several thousand games, as required for evaluation (see Section 3.3 for details). The summary statistics are printed at the end of the simulation and (among other things) show the average and the maximum number of turns per game, decisions per game, amount of actions executed (per type), and the amount of trade offers made and accepted. The gaming environment is multi-threaded and supports parallel execution of games. It is achieved by using a thread pool, where each thread runs one game; should a game finish, the now free thread will start another.

Configuring simulations can be done via command-line flags. Modifiable characteristics include the number of games and threads, the choice of agents, maximum game length, and whether trading is allowed. Furthermore, should any MCTS agents be used, it is possible to modify the number of rollouts and specify whether sampling is typed or uniform. Various levels of logging can be enabled: at the WARN level, only summary statistics and game initialisations and winners are printed (a "print step" value controls how often these messages are logged, as when thousands of games are played, these messages are simply an indication of progress). At the INFO level, each bankruptcy and the time it took for each MCTS decision are logged in addition to everything printed for the WARN level. At the lowest logging level, DEBUG, complete information about every choice of every agent and every event in the game is logged.

3.1.2 MCTS Model

My supervisor Alex Lascarides and her PhD student Mihai Dobre provided the code to learn Settlers of Catan with MCTS (Dobre et al., 2017). The codebase is written in Java and implements the Settlers game and the code needed for an agent to learn to play the game using Monte Carlo Tree Search (MCTS learner). Settlers of Catan and Monopoly have multiple similar characteristics: both games have large action spaces, randomness, and a heavy emphasis on trading. While these properties make it interesting to tackle the two games with reinforcement learning and compare the results, the actual rules and gameplay are different enough that sharing game code was not feasible. However, as I applied the same learning process as was used for Settlers of Catan by Dobre et al. (2017) to Monopoly, I could make use of the MCTS learner and integrate it into my Monopoly implementation.

The MCTS learner implements parallel Monte Carlo Tree Search. The learner allows for selection of the number of rollouts, maximum tree size and depth, and the number of threads to use. Furthermore, action selection policy can be changed: MCTS learner comes with implementations of Upper Confidence Bounds for trees (UCT) (Kocsis et al., 2006), Rapid Action Value Estimation UCT (UCT-RAVE) (Gelly et al., 2007), and Polynomial UCT (PUCT) (Auger et al., 2013). However, for this project, only UCT was used (this was found to be the metric that gave best performance on Settlers of

Catan). To find a best possible action in a given state of the game, the MCTS learner works as follows: It creates an instance of the game with that state and adds it to the tree of known states as a node. Each node is either a standard node - the outcome of any actions in that state are deterministic, or a chance node - the consequences are not deterministic, for example, dice rolls or stochastic opponent policies. The learner will then get a list of the possible actions in that state, and if the state is a chance node, sample one of the valid actions according to the games transition model. If the state is a standard node, it will expand the node: select an action based on the selection policy and simulate a game. The states encountered will be added to the tree as nodes, and each of them will get updated to reflect the chance of winning when the game is in this state. These actions will be selected and games simulated until the specified number of rollouts is reached. The selection policy is important, because it has to balance exploration and exploitation - it is necessary to explore the state space, but it is also important to play well to get an accurate estimate of the usefulness of the available actions. UCT does this by choosing actions based on their win rate and how much they are visited: it selects the action that maximizes Equation 3.1 at the node in question (Browne et al., 2012). In Equation 3.1, n is the number of visits to the parent and n_j to the child, X_j is the win rate after the j-th move, and C is a constant indicating indicating the trade-off between exploration and exploitation: higher C indicates more exploration.

$$UCT = \overline{X_j} + C\sqrt{\frac{2\ln n}{n_j}} \tag{3.1}$$

Some work was needed to allow Monopoly to be integrated with the MCTS learner. It was aimed at learning Settlers of Catan, but did provide an interface that a new game could implement to work with the tree search. The interface requires the game to provide a representation of state and the current node (standard or chance), list the currently available actions, perform a chosen action, and clone the game. As my implementation of Monopoly is object-oriented, implementing the interface was not trivial, as not all of the necessary info was visible to the main Monopoly game class or presented in the same format. For example, in my implementation, each player has an associated agent that makes the decisions on what action to take, and the main game class does not access this info (Figure 3.1). Therefore, I wrote a wrapper that implements the interface required by the MCTS learner and instantiates my Monopoly class based on the state information. The Monopoly class then exposes all the necessary information to the wrapper, which will, in turn, provide them to the MCTS learner in the required format. Additionally, in the MCTS simulations, the actions of the MCTS agent are not decided by the agent as when playing the game, instead, the available moves are sent to the wrapper so that the MCTS learner could sample the action.

3.2 Game setup

3.2.1 Simplifications

To make the game computationally more feasible, I decided to make some simplifications to the official rules of Monopoly.

Firstly, I decided not to allow bidding on properties that a player chose not to purchase. So if a player lands on a property square but does not buy it, it will not be auctioned off to the highest bidder. Instead, it will remain the property of the bank and can be purchased by the next player who lands on the square.

Secondly, the official rules allow mortgaging property, unmortgaging property, building and selling houses at any point in the game. However, as that leads to a chaotic structure of the game, and complicates things by increasing the number of choices the players have to make by a significant factor. Thus I decided to only allow these actions on the player's own turn, before they roll the dice.

Trading is another complicated part of the game, and thus I made multiple simplifications to it. Unlike Bailis et al. (2014), I do not completely remove trading from the game. However, I do not allow haggling in trading, instead, trades are posed as offers "I want to trade my x for your y" that can either be accepted or rejected, counteroffers cannot be made. Instead, if a player wishes to make a "counteroffer" they have to wait for their turn and propose a new offer then. Trading is restricted to property for property. Money and "get out of jail free" cards cannot be traded.

In the actual rules, the player receiving a mortgaged property from a trade must immediately unmortgage it or pay a 10% fee for keeping it mortgaged. This rule is seldom used when playing Monopoly. Thus, to avoid adding an extra choice to the game, I decided to simplify the rule and take the decision factor out of it. One way would be to make it a requirement to always pay the 10% fee, but generating a list of possible trades would be quite complicated, as it would still be necessary to check whether the player who is offered a trade has enough money to pay that fee. The idea which I decided to use was that mortgaged properties could be traded normally - just as regular properties. The player receiving the property would then have a mortgaged property they can unmortgage at any time (by paying the bank). Another alternative solution would be that only properties that are not mortgaged are allowed to be traded, however, this approach would be more restrictive.

Furthermore, as the number of possible trades the official rules allow could be huge, I limit the number of properties that can be traded in one interaction. A similar limit was introduced by Dobre et al. (2017) for Settlers of Catan - they limited trading of resources to a maximum of one resource card for two (and two for one). To find the number of properties that is feasible to trade at a time, I calculated the theoretical limits on the number of possible trades: even when only one for one (1:1) trades are allowed there are theoretically up to 196 possible trades. Including two for one (and one for two) would result in an extra 2548 possible combinations and 2:2 would add 8281 potential trades, so anything more than 1:1 is most likely not feasible. However, as

these numbers are just the upper bounds on the number of trade options, it is possible that these situations do not rise often in actual gameplay and handling the average number of options is more manageable. Therefore, I simulated four random agents and recorded the available trade options in every turn. On average, more than half of the theoretical maximum number of actions were available: one for one trading allows around 140 possible trades, two for one around 1900, and two for two over 6000 (table 3.1). I chose to limit trading to one property for one property, as 1900 trade actions each turn seemed too much considering the computing power I had available.

As the amount of available trades at any given moment is relatively high, it is possible that the vast number options could potentially still pose an issue. As a remedy, I decided to run the same simulation recording the number of trade options with the alternative trading simplification, where mortgaged properties are not allowed to be traded at all. As it turns out the majority of potential trades included mortgaged property, so while removing that option is restrictive, it cuts down the number of available trades by more than a factor of 5 in one for one trading and nearly a factor of 15 in two for two trading (table 3.1). Thus, removing the ability to trade mortgaged property could be a viable method for restricting the number of available actions should it be necessary. However, at this point I decided not to limit trading of mortgaged property further.

Type of Trade	Allow Mortgaged	Average Available	Maximum Available
1 for 1	Yes	142.40	196
2 for 1	Yes	1900.91	2744
2 for 2	Yes	6472.91	11025
1 for 1	No	24.56	196
2 for 1	No	177.75	2744
2 for 2	No	377.11	11025

Table 3.1: The number of available trade options for each restriction on trading (maximum number of properties in trade and whether mortgaged properties can be traded). Collected from simulating 4 random agents by running 10,000 games with a maximum of 10,000 turns per game.

In case a player goes bankrupt, all properties they own will go to the bank and get unmortgaged. In case the fee that they were unable to pay was to another player, that player would only receive as much money as the now-bankrupt player had after selling all houses and mortgaging all property. The bank does not pay out anything for receiving the mortgaged property.

Finally, I modelled chance and community chest cards as “draw, replace, and shuffle” instead of keeping the card if it is a “get out of jail free” card and putting it at the bottom of the pile otherwise. Thus I do not have to keep track of the order of the card piles, which would increase the state space and make it more difficult for the MCTS agent to learn the game.

3.2.2 Modelling the Game

As stated previously, the unknown order of chance and community chest cards make Monopoly a game of imperfect information. However, by modelling the chance and community chest card piles as "draw, replace, and shuffle", they become chance events and the draw history is not important. Therefore, my implementation of Monopoly can be considered a perfect information game (with some non-deterministic actions) - the history of all turns and the entire current state of the game is known to each player (Chen, 2012). Perfect information makes playing the game and learning it easier as the exact current state of the game is known and does not have to be estimated. However, as there are chance events in the game, Monopoly is a non-deterministic game, and it is not possible to precisely predict the outcome of every action, even with complete knowledge of the state of the game.

Monopoly is usually played with open hands - the title deeds have to be kept face up in front of the player (Waddingtons Games, 1996), but in most cases, money is also kept out in the open. Thus, each player has complete knowledge of the possessions of the other players. However, as opponent strategies are unknown (and I do not model player types), it is still a game of incomplete information. Combining this with the massive state space, it is most likely not possible to analytically solve the game. This is one of the reasons I am using a simulation-based approach to learning the game, which will estimate the outcomes of actions from rollouts of the game.

As I explore the effect of sampling action types and actual actions separately (and in that order), it is necessary to define what the action types are. Luckily, the rules of Monopoly define these types quite clearly; and as there are not too many of these, I can just use the definitions without modification. Therefore, the action types I use in the main phase of the game are: build house, sell house, unmortgage property, mortgage property, trade, and roll dice. Additional actions are sometimes available in other stages of the game: responding to a trade, the option to buy a property after landing on the corresponding square, and choosing to pay a fine or take a chance card (if the corresponding community chest card is taken). The only compulsory action in the main phase of the turn is the dice roll, which effectively ends the players turn (if the player rolls doubles they get to go again, but that is considered the next turn). However, players are not limited as to how many of the other actions they perform - their property and money mostly restrict it. This grouping also illustrates the action type cardinality problem: there is always one "roll dice" action, but the amount of other actions combined is on average around 150, and the number of possible trades can go as high as 196 (table 3.2), even when restricted to only 1 for 1 trading.

When trading is not allowed, or an agent simply decides not to cooperate with the other players, it is possible that the game never ends - if no player can complete a full set of properties, the rents will be too low compared to the 200 pounds that the players receive every time they pass the start square, thus effectively paying each player more than they can spend and making it impossible for anyone to go bankrupt. As I cannot run infinite simulations, I will need to terminate these games at some point. My approach is to call the game a draw should the game exceed a certain amount of turns. An alternative would be to liquidate all property that the players own and call the player with the

ACTION	PROBABILITY	AVERAGE	MAXIMUM
BUILD HOUSE	13.59%	0.28	15
SELL HOUSE	12.86%	0.23	10
UNMORTGAGE	90.28%	4.91	23
MORTGAGE	91.38%	3.44	21
TRADE (1:1)	99.86%	142.40	196
ROLL DICE	100%	1	1

Table 3.2: Probabilities of different action types being available and the average and the maximum number of available actions for each action type. Collected from 10,000 games with four typed random agents and a maximum of 10,000 turns per game.

most monetary value the winner. However, the monetary value is not always a perfect indicator of a players position in the game. Furthermore, this approach could change the strategies of the agents as it penalises investments in the later parts of the game (liquidating assets does not yield as much money as was invested). And as agents that are open to trades will most likely not get into this situation (as can be seen from the next section), calling it a draw seems like a simpler, less disruptive solution.

3.3 Method of Evaluation

The main metric I use to evaluate agents is their win rate - the percentage of games each agent won. As the game of Monopoly is stochastic by nature, many trials are needed in order to get an accurate estimate. When I compare heuristic agents, I can run 10,000 games with each set of players. However, once I introduce agents that use Monte Carlo Tree Search to decide their moves, the computational cost of simulating games increases by a large factor. Therefore, with the limited computational capacity I have access to, I can run 100 games for each of those experiments. While I would have ideally conducted a larger number of trials, 100 is still reasonable, as it results in accuracy of one percentage point and allows me to get statistically significant results (Chapter 4).

All of the games and simulations I run have four agents. Furthermore, as the starting agent is always random, I can state the null hypothesis that if all four agents are of equal strength, each of them wins 25% of the games (and any observed deviations are due to noise) (Tijms, 2012). Since every agent either wins or loses, I can calculate the binomial confidence intervals and standard deviations for the win rates (Hayter, 2012). With 10,000 trials the binomial 99% confidence interval indicates that any win rate lower than 23.9% or higher than 26.1% is significantly different from the null hypothesis. For 100 games, I used the 90% confidence interval which shows that any win rate between 17.86% and 32.14% is statistically not significant compared to the null hypothesis.

The number of games (or trials) should not be confused with the number of rollouts.

The rollout phase is the stage in Monte Carlo Tree Search, where a predefined number of games is simulated. These simulations are used to estimate the value of game states, which are the basis of the MCTS agents decisions. As the number of rollouts increases, the model will become more accurate and the agent will be able to make smarter decisions. This comes at the cost of compute time - each additional rollout is a simulation of a whole Monopoly game. At a certain point the performance of the agent will start to level out and any further simulations only have a tiny effect. Thus, a balance needs to be found between the performance of the agents and the computational cost.

To compare the performance of typed-MCTS and uniform-MCTS, I first play two sets of games against the best agent among the heuristic baselines I designed (details in Section 3.4): a set with typed sampling and another with uniform sampling. I then compare the relative performance of the two MCTS agents to see which is more effective for the game of Monopoly. However, as this relative win rate only shows that one of the methods of sampling is better against this specific opponent, I also play a typed-MCTS agent against three uniform-MCTS agents to get a head-to-head comparison of the sampling methods.

I play the MCTS agent against three opponents of the same kind, as it is the standard practise for evaluating the quality of an agent in a 4 person stochastic game of extended form (Bailis et al. (2014) used the same method for Monopoly and Dobre et al. (2017) for Settlers of Catan). Furthermore, my model does not attempt to classify the opponent types it plays against and only learns a single strategy. Therefore, I can also sample against three opponents of the same kind in the rollout phase.

3.4 Baseline Agents

Monte Carlo Tree Search learns a policy by sampling actions and evaluating them by simulating the game. So, to train an agent with MCTS I needed heuristic agents to sample opponent actions from. I started with an extremely simple agent - a random agent, that samples actions uniformly at random. The random agent is also “typed” - it samples an action type and then samples the actual move, both uniformly at random. This agent mainly served as a baseline for evaluating the smarter heuristic agents.

To have something I could compare my results against, I implemented the agent described by Bailis et al. (2014) (from now on Bailis agent) which they used as a baseline for a Q-learning approach to Monopoly. That agent sells houses and mortgages property whenever it has less than 150 pounds, buys and unmortgages property and houses if it has more than 350 pounds and does not trade. Against three random agents, this baseline achieved a win rate of 91.1% and draw rate of 7.8% as can be seen from Table 3.4.

As the process to design the Bailis agent was not explained in much detail, I decided to try and design my own heuristic baseline to see if it can beat the performance of the Bailis agent and have alternative agents to train and evaluate against. I wrote a configurable agent that can be tuned to a variety of play styles. It is a heuristic agent that samples actions in the typed manner discussed earlier: in the first stage it filters

out action types that meet the criteria specified by the supplied parameters and then chooses a random acceptable action type. In the second phase, it chooses a random move of that chosen type to be executed. Five parameters can be configured to choose which actions are taken:

- Buying base value (B_b) - a constant value indicating the minimum amount of money needed for the buying actions to be available
- Buying relative value (B_m) - a fraction of the current highest possible cost (C_{max}) added to the purchase base value
- Selling base value (S_b) - a constant value indicating the amount of money under which the selling actions become available
- Selling relative value (S_m) - a fraction of the current highest possible cost (C_{max}) added to the sell base value
- Trades boolean value - controls whether the agent accepts and makes trade offers or declines everything.

The combination of these parameters and the current highest possible cost that can occur to

(C_{max}) is used to calculate which actions the player can perform: should the player have more money than B they can buy, unmortgage properties, and build houses; should they have less than S money, they can sell houses and mortgage properties (Equation 3.2).

$$\begin{aligned} B &= B_b + B_m * C_{max} \\ S &= S_b + S_m * C_{max} \end{aligned} \tag{3.2}$$

To tune the configurable agent, I ran a grid search with various settings of the parameters to see which settings of the parameters performed best against three random agents. I chose to highlight two configurations and use those as my agents. One of them keeps more money and plays safer; thus I call this the cautious agent. The cautious agent bases its purchase decisions on the current highest cost that can occur - it calculates the maximum of all possible chance cards and rents on the board and uses that number as a basis. Specifically, it sells houses and mortgages property whenever it has less than 0.75x the current highest cost of money, buys and unmortgages property and houses if it has more than 3x the current highest cost of money. The other agent I chose keeps very little money and takes more risks, so I call it the aggressive agent. The aggressive agent is quite simple: it buys and unmortgages property and houses if it has more than 50 pounds and only sells houses and mortgages property if it does not have enough money pay a fee that has occurred. For ease of comparison, the configurations of the agents are also presented in Table 3.3.

To compare the performance of the Bailis agent against my baselines, I played each agent (separately) against three random agents for 10,000 games. Table 3.4 presents the win and draw rates of each different agent, both with trading and without. I used the null hypothesis that each agent would win 25% of the games if they were of equal

AGENT	BUY, BUILD, AND UNMORTGAGE	SELL AND MORTGAGE
BAILIS	£350	£150
CAUTIOUS	$3 * C_{max}$	$0.75 * C_{max}$
AGGRESSIVE	£50	£0

Table 3.3: Configurations of the different baseline agents. The "Buy, Build, and Unmortgage" field shows the minimum amount of money the agent needs to have to do buy property, build houses, and unmortgage properties. If the agent has less money than what is listed in the "Sell and Mortgage" field, it will sell a house or mortgage a property. C_{max} is the current highest possible cost the player could have.

strength to the random agent with trading capability. I found the significance thresholds by calculating the binomial 99% confidence interval (Hayter, 2012): as I simulated 10,000 games, any win rate lower than 23.9% or higher than 26.1% is significantly different from the null hypothesis. This is backed up in Table 3.4 as the random agent that trades wins 24.83%. As every result in Table 3.4 is significant, I will not present the standard deviations or z-scores.

Comparing the heuristic baselines, it turned out that the aggressive agent did, in fact, perform better against random agents than the Bailis agent, reaching a win rate of 92.16%, thus beating the Bailis agent by around one percentage point. The performance of the cautious agent and the Bailis agent was nearly identical, with both at around 91% win rate. Regardless, all of the baseline agents outperform the random agents by a significant margin.

AGENT	TRADING	WIN RATE	DRAW RATE
RANDOM	NO	28.22%	4.86%
BAILIS	NO	91.10%	7.80%
CAUTIOUS	NO	91.00%	7.80%
AGGRESSIVE	NO	92.16%	6.73%

Table 3.4: Comparison of agents that do not trade evaluated against 3 random agents that trade. 10,000 games with a maximum of 10,000 turns per game.

Trading is necessary to achieve high performance in the game as neglecting this opportunity handicaps the player, leaving them unable to obtain any new properties once everything on the board is bought. However, as can be seen from Table 3.4, trading is useful only if trades are done intelligently - a random agent that does not trade actually wins 28.22% of the games, 3% more than a random agent that is allowed to trade (which would win 25% against 3 random agents that trade as per the null hypothesis). Therefore, the baseline agents also need intelligent trading strategies.

I designed the following trading policy for the cautious agent: it never gives away any properties from a completed set and accepts any trade which completes a set. Unless the trade completes a set, the agent will never trade an unmortgaged property for a

mortgaged one. If the trade does not complete a set nor break one, the agent accepts the deal in case it is closer to finished set as a result and the property that it receives is worth more than the one it gives away. The agent accepts any trade that matches these criteria and declines all other proposals, however, when it decides to propose a deal, it finds all acceptable trades and offers one of them uniformly at random. This is to avoid a repeating situation where the agent keeps proposing what it deems the best trade, whereas if the other player declined the offer before, its highly likely they would refuse it again.

Ideally, multiple trading strategies would be considered, as the machine learning model might find a good policy that works against that specific baseline but completely fails to generalise. However, as this is speculation and coming up with alternative strategies takes time (potentially unnecessarily), it seemed best to start by using this successful strategy for the other two agents (Bailis agent and aggressive agent) and only come back to designing alternative strategies should they be necessary. Unsurprisingly, trading boosted the performance of the agents, improving their win rates by around 5% (Table 3.5). Using the same trading strategy resulted in similar performances in all three agents that made use of it, and did not change the fact that the aggressive agent had the highest win rate.

AGENT	TRADING	WIN RATE	DRAW RATE
RANDOM	YES	24.83%	0.12%
BAILIS	YES	96.06%	0%
CAUTIOUS	YES	96.18%	0%
AGGRESSIVE	YES	96.86%	0%

Table 3.5: Comparison of agents that trade evaluated against 3 random agents that also trade. 10,000 games with a maximum of 10,000 turns per game.

Although the aggressive agent did perform best against random agents, it is still necessary to evaluate the performance of the different agents against different and smarter opponents to get an estimate of whether the strategy generalises. To do this, I played all four of the baseline agents (random, Bailis, aggressive and cautious) against each other. The result confirmed my suspicions and showed that the three strategies are not as equal as they seem: the aggressive agent will still win the most games, but the previously pretty good cautious agent will lose to both Bailis and aggressive agents by a significant margin (Table 3.6). Since the aggressive agent wins more than half of the games and beats the win rates of all others by over 20%, it is a natural choice for the agent to use for the opponents in MCTS simulations.

3.5 Evaluating the MCTS Agents

Before any experiments with MCTS agents could be run, I had to decide all of the parameters of the simulations. I already chose the aggressive agent as the baseline

Winning Agent	With Trading	Without Trading
Random	0.06%	0.03%
Bailis	32.06%	17.0%
Aggressive	55.01%	17.69%
Cautious	12.87%	13.69%
No winner	0%	51.59%

Table 3.6: Comparison of win rates of all 4 baseline agents evaluated against each other. 10,000 rollouts with a maximum of 10,000 turns per game.

agent to sample against in rollouts as it was vastly superior to the other heuristic agents. Since in some situations, games can get very long or not finish at all (I call this a draw), I also needed to find a reasonable maximum length of the game. Up to this point I had used 10,000 turns, but this limit could be unnecessarily long and slow down the simulations. Finally, I had to decide on the number of rollouts - I needed to find a balance between the performance of MCTS and the cost of compute time. While a bigger number of rollouts would be preferred, I had to make sure that it would still be possible to run enough games with the slower uniform sampling, as my computational resources were very limited.

I will now explain the relevant data and details about these choices.

3.5.1 Branching and Run Time

The two main problems typed-MCTS is supposed to address are the curse of history and the curse of dimensionality: the simulations get exponentially more complex as the depth (the number of decisions made in a game) and the branching (the average number of available actions per turn) of the problem increase.

To see if using typed sampling in Monopoly reduces the length and dimensionality of games, I needed to get estimates of the branching and depth that can be expected from the game with both uniform and typed sampling. To do this, I simulated 2,000 games against random agents playing first a typed random agent (first samples action type randomly and then the specific action randomly) and then a uniform random agent against three typed random agents. I collected the branching and depth of the game from the chosen agents point of view. As can be seen from Table 3.7, using typed sampling reduced the depth of the game more than four times and the branching almost six times. It also has a huge effect on the run time of the game: games that had one uniform random agent took on average five times longer than the ones that only used typed agents.

However, as the MCTS agent would not be trained against random agents, but rather the previously designed heuristic baselines, namely aggressive agents, it was more accurate to estimate the dimensionality and length of the games playing against those baselines. Table 3.8 presents the results of games against 3 aggressive agents. Due

AGENT	BRANCHING	DEPTH	RUN TIME
UNIFORM RANDOM	118.77	22515.08	221.31
TYPED RANDOM	20.17	5028.49	43.3

Table 3.7: Comparison of average branching, depth and run time (in milliseconds) of the simulations between a uniform random agent and a typed random agent. Collected from 2,000 games against typed random agents. Simulations are run on a a single 2.4GHz core of a laptop.

to the smarter decisions from aggressive agents, the games are a lot shorter and run much faster. In the uniform case, game depth is more than three times smaller, and branching is reduced by around 43%. In the typed case, the effect is even bigger: game depth drops 16 times and branching is halved. The effect on run time is also clear: for uniform games run time is reduced four times and for typed over seven times.

AGENT	BRANCHING	DEPTH	RUN TIME
UNIFORM RANDOM	67.17	6479.50	52.84
TYPED RANDOM	9.07	304.75	6.00

Table 3.8: Comparison of average branching, depth and run time (in milliseconds) of the simulations between a uniform random agent and a typed random agent. Collected from 2,000 games played against 3 of the aggressive baseline agents. Simulations are run on a a single 2.4GHz core of a laptop.

As uniform sampling results in games of vastly bigger depth and branching than typed sampling, the uniform-MCTS simulations are expected to take longer than the typed ones. MCTS scales exponentially with the depth of the game: at each new decision a new tree search needs to be run which also scales with the depth of the game. Additionally, the games with uniform agents are already very slow: around 9 times slower than those with the typed agent. Therefore, some additional limits had to be placed on the game, as otherwise it would have been impossible to run any reasonable number of simulations with the computational resources I had available.

The main source of available actions is trading (as shown in Section 3.2.2), thus trade-actions get sampled a lot - on average 55 times in a turn and occasionally up to 350 times in a single turn (collected from simulating 2,000 games against aggressive agents). Therefore, setting a cap on the number of allowed trades per turn seemed to be an effective way to limit the run time of the game. Dobre et al. (2017) introduced a similar restriction for Settlers of Catan - they limited the number of consecutive trade actions allowed to 5. This would also be a reasonable rule for human players, as it would be incredibly annoying if one player decided to offer 350 (not necessarily different) trades in a single turn and almost 6000 in an average game. Thus, I experimented with a couple of settings on the trade limit to see how it impacts the branching, depth, and run time of the games.

As can be seen from Table 3.9, setting a limit on the allowed trades per turn is very effective in reducing the length and number of available actions when doing uniform sampling - limiting the trades per turn to 3 reduces branching by more than 3 times and the depth by 13 times. However, typed sampling is hardly affected - both branching and depth are reduced by around 4%. Therefore, in order to still allow a reasonable number of trade proposals, but also keep the game size under control, I decided to set the maximum number of trade offers per turn to 3. Furthermore, as the uniform games now took around 300 turns on average, I decided to limit the length of the game to 1000 turns - long enough to allow longer games to finish, but short enough to not affect performance should the game result in a draw eventually anyway.

TRADE LIMIT	AGENT	BRANCHING	DEPTH	RUN TIME
3	TYPED	8.70	292.43	6.29
5	TYPED	8.90	296.14	6.15
10	TYPED	8.92	301.04	7.41
NONE	TYPED	9.07	304.75	6.00
3	UNIFORM	21.38	468.85	7.40
5	UNIFORM	27.94	672.23	9.95
10	UNIFORM	38.07	1176.86	13.53
NONE	UNIFORM	67.17	6479.50	52.84

Table 3.9: Comparison of average branching, depth and run time (in milliseconds) of the simulations between a uniform random agent and a typed random agent with varying trade limits (max trades per turn). Collected from 2,000 games played against 3 aggressive agents. Simulations are run on a a single 2.4GHz core of a laptop.

3.5.2 Number of Rollouts

To find a balance between performance of the MCTS agent and the time it takes to learn a decent strategy, I experimented with a varying number of rollouts and estimated the learning curve of the typed-MCTS agent. Table 3.10 presents the win rates of a typed-MCTS agent played against 3 aggressive agents dependent on the number of rollouts the typed-MCTS agent performs. As can be seen, the typed-MCTS agent plays very strong at a higher number of rollouts, reaching a win rate of 53% at 20,000 rollouts, thus winning more than half of the four player games. Due to the lack of computational resources, these values are estimated from 100 games. Therefore, more games should be run in the future, to confirm that these rates are accurate.

As an agent of equal strength to the baselines would win 25% of the games, using the binomial 90% confidence interval shows that any win rate between 17.86% and 32.14% is statistically not significant compared to the null hypothesis (Hayter, 2012). While this means that typed-MCTS with 5,000 rollouts might not actually outperform the aggressive agent, an agent with 10,000 or more rollouts would. Judging from the data I have, the performance of the typed agent could still increase with more rollouts. However, in the interest of saving computation time, I decided to limit any

further experiments to 10,000 rollouts. This choice would also make it feasible for the agents to be played against humans, as the decision times of the typed-MCTS agent are usually less than 20 seconds at 10,000 rollouts. Furthermore, 10,000 rollouts was also chosen by Dobre et al. (2017) when they applied this method to Settlers of Catan.

ROLLOUTS	WIN RATE	SD
1K	14%	3.47%
2K	21%	4.07%
5K	30%	4.58%
10K	46%	4.98%
20K	53%	4.99%

Table 3.10: Comparison of win rates of the typed-MCTS agent dependent on the number of rollouts performed. Evaluated against 3 aggressive agents, 100 games with each configuration. Standard deviations are calculated according to a binomial distribution.

Running 100 games of 3 aggressive agents and one typed-MCTS agent with 10,000 rollouts took around 40 hours on the student compute cluster I had available. This cluster has 40 cores running at 2GHz, however, it is shared among students, so not all of it was available to me at all times. As shown earlier, uniform sampling is significantly slower: running 100 games of 3 aggressive agents and one uniform-MCTS agent took over 90 hours. Therefore, while some sacrifices in the number of games had to be made, running a somewhat reasonable amount of games with both of the agents was still feasible and the number of rollouts did not have to be reduced.

Chapter 4

Results

4.1 Games Against Heuristic Baselines

I evaluated both the typed-MCTS and uniform-MCTS agents against 3 of the agressive agents. As can be seen from Table 4.1, with the same number of rollouts typed sampling is far more effective than uniform sampling - typed-MCTS reaches a win rate of 46%, outperforming the heuristic agents by a significant margin, whereas uniform-MCTS does not even come close to beating the best heuristic baseline and only has a win rate of 8%. Furthermore, uniform sampling is slower to run - running 100 games with typed-MCTS takes around 40 hours (on the student compute cluster), but nearly 100 with uniform sampling. In terms of win rate, uniform sampling is comparable to typed sampling with somewhere around 20 times fewer rollouts: typed-MCTS with 500 rollouts reaches a win rate of 8% - identical to uniform-MCTS at 10,000 rollouts. However, using typed sampling with 500 rollouts is far cheaper - 100 games only take around 2 hours to run - 50 times less than those with the uniform agent.

AGENT	ROLLOUTS	WIN RATE	SD
UNIFORM-MCTS	10K	8%	2.71%
TYPED-MCTS	500	8%	2.71%
TYPED-MCTS	10K	46%	4.98%

Table 4.1: Comparison of win rates of the typed-MCTS agent and the uniform-MCTS agent. Evaluated against 3 aggressive agents. Standard deviations are calculated according to a binomial distribution.

Since in the previous experiment the MCTS agents both play against aggressive agents and sample against aggressive agents in MCTS rollouts, they have an unfair advantage - the ability to simulate games against the opponent's strategy. This could result in the MCTS agent only learning a strategy that works against that specific opponent and fails to win against anyone else. To see if the typed agent learned a good general strategy, I evaluated it against all of the heuristic baselines that I had. In the rollouts, games

are still simulated against three aggressive agents, however, the actual games would be played against 3 distinct agents: one Bailis agent, one aggressive agent, and one cautious agent. Table 4.2 presents the results from 100 such games. As can be seen, the typed-MCTS agent does even better than when playing against three aggressive agents - it now reaches a win rate of 63%, far higher than any of the heuristic agents.

AGENT	WIN RATE	SD
CAUTIOUS	5%	2.18%
BAILIS	9%	2.86%
AGGRESSIVE	23%	4.21%
TYPED-MCTS	63%	4.83%

Table 4.2: Comparison of win rates of the typed-MCTS and baseline agents when played against each other (all 4 agents at the same time). Evaluated from 100 games. Typed-MCTS samples against 3 aggressive agents and does 10,000 rollouts. Standard deviations are calculated according to a binomial distribution.

4.2 Typed Against Uniform

To conclusively show that typed sampling is better than uniform sampling in the game of Monopoly, I played a typed-MCTS agent against three uniform-MCTS agents. As each of the four agents now needed to run simulations for their decisions, these games are inherently around four times slower than only playing one uniform agent. Additionally, the average length of these games was longer (average length of 315 turns compared to 186 in the heuristic case). Therefore, with the limited amount of computing power I had available, I had to limit the number of rollouts further. I decided to run this experiment with 2,000 rollouts for each agent. As before, I restricted the game length to 1,000 turns, after which I call it a draw.

The typed-MCTS agent won 47% of the 100 games (standard deviation of 4.99%), whereas on average, each of the three uniform-MCTS agents won 17.33% (standard deviation of 3.79%). Therefore, typed sampling is also far more effective when playing head-to-head against agents that sample their moves uniformly. The uniform-MCTS agent proved to be an easier opponent than the aggressive agents (typed sampling with 2,000 rollouts only got a win rate of 21% against aggressive agents). However, in some instances the games were quite even and lasted for a long time: one of the games even resulted in a draw. In comparison, no draws occurred when playing either of the Monte Carlo Tree Search agents against heuristic agents at any number of rollouts. That said, the win rates show that sampling actions in two steps was the key to winning.

Chapter 5

Discussion

5.1 Methodology

5.1.1 Gaming Environment

The first half of the project consisted mostly of designing and implementing the gaming environment and the baseline agents. However, in the second half, after the system had all the necessary functionality (the ability to run a set number of games, collect statistics, easily modify experiment parameters, and swap out the playing agents), experimentation took priority. I based the design of the gaming environment and the Monopoly implementation on object-oriented principles. However, the interface that was needed for the MCTS learner did not quite fit my pattern and I had to make some adjustments that would not be considered good practice. Thus, in some ways, functionality became more important than code health.

Looking back, sacrificing some design principles (like making the implementation object-oriented) right from the start could have made it easier to integrate Monopoly with the Monte Carlo Tree Search code. However, I believe my design choices were worth it for the eventual clarity of the code, and the shortcomings could be fixed with a reasonable amount of work. Restoring the proper abstractions I made at the beginning of the project and doing a general code clean-up would improve the quality of the codebase, while still allowing for the MCTS learner to function correctly.

While seemingly simple, Monopoly has some rules and moves that make it computationally expensive. Availability of some actions like building houses and mortgaging properties depends on whether the player owns a full set of the properties in question. However, checking all properties of a specific colour, and whether there are houses on them gets expensive if it is done too often. I dealt with this by separately keeping count on these conditions and updating them every time a property changed owner, or a house was built or sold. Updating it after each event was significantly faster than doing these checks every time a list of possible actions was requested.

The primary factor slowing down the games was trading - finding a list of possible

trades was not as effective to cache as the other moves, and deciding on trade offers was also slower. Thus I placed some restrictions on what can be traded and how trading is handled. I allowed mortgaged properties to be exchanged without a fee, restricted trading to only one property for another property, and limited the number of trades allowed per turn. These simplifications made it possible to conduct the experiments in a reasonable time frame. With more processing power, some of the simplifications I made could be loosened or removed. Although, experimenting on the game with the complete original rule set would likely still be intractable, as the number of possible trades gets colossal when any combination of any number of properties and any amount of money is allowed. However, lifting some of the other restrictions seems to be a lot more feasible, for example, trading could be expanded, so that each player can offer up to two properties per trade.

The Monopoly implementation could use some improvement, should future experiments be planned. Currently, the gaming environment is completely functional and easily configurable, has extensive logging and I have manually tested it. However, it would be reasonable to write automatic unit and integration tests, especially if the code needs to be modified in the future. Automated tests could drastically reduce the number of new issues and simplify the debugging process.

5.1.2 Baseline Agents

The main function of the baseline agents was to act as opponents in the Monte Carlo Tree Search rollouts phase. Using the heuristic agents allowed MCTS to quickly simulate games against reasonably intelligent opponents and thus evaluate the current possible moves. I designed two heuristic agents myself and implemented what I call the Bailis agent according to the description of a Q-learning paper on Monopoly. Unlike the previous work by Bailis et al. (2014), all the agents could trade as it is what makes Monopoly complicated and interesting to learn. Having three distinct agents with varying play styles and strategies was beneficial, as it helped me choose a strong agent to use for the MCTS rollouts.

The agents were reasonably challenging - while typed sampling beat all baselines by a significant margin, it still required a decent number of rollouts to do so. However, uniform sampling with 10,000 rollouts did not even come close to the performance of the heuristic agents. Thus for these experiments, the baseline agents were perfect as they challenged the MCTS agents and illustrated the differences between the two methods of sampling. In the future, alternative, potentially more complex agents could be considered to see if the added intelligence helps with MCTS sampling and boosts the performance even more.

5.1.3 Typed Sampling

Using typed sampling with Monte Carlo Tree Search proved to be highly effective. As shown in Section 3.5.2, by doing 10,000 or more rollouts, typed-MCTS wins around

half of the games against three aggressive agents. In comparison, uniform-MCTS only achieved a win rate of 8% with the same number of simulations. When played head-to-head against each other (one typed-MCTS agent and three uniform-MCTS agents), the agent using typed sampling won 47% of the games, whereas the ones using uniform sampling only won 17% of the games on average. This result is similar to what Dobre et al. (2017) observed in Settlers of Catan - in their case typed-MCTS won 54% of the games against 3 uniform agents. The difference could be due to the game, or the number of rollouts - they managed to do 10,000 rollouts for each agent, whereas I had to settle for 2,000 due to lack of computing power. In either case, sampling actions in two steps was substantially better than sampling from all actions in one go.

The motivation for typed sampling came from its effect on branching: if an action type is chosen, it is no longer necessary to consider moves from any other category. Therefore, the number of choices will never be as extensive, and each of the options can be evaluated properly. In addition to the intelligent behaviour, the two-step process was significantly faster to run. While the speed-up is partially due to reduced branching, the games are also considerably shorter. Therefore, typed sampling allows the agent to make more influential decisions instead of stalling and spending time on useless moves. Smarter choices are likely a consequence of how the actions are grouped - since the moves in each category are correlated, they have similar effects and achieve matching goals. Thus, deciding on the type of the action plays a prominent role in the result of that action. And since there are only a few possible categories of moves, the problem becomes a lot less complicated.

5.1.4 Experiments

Monte Carlo Tree Search for games of this complexity can require a lot of processing power. The model is not pre-trained, instead, new simulations need to be run for each decision. Moves can still be made in a reasonable time - with typed sampling (at 10,000 rollouts) most decisions were made in 10-20 seconds, thus probably faster than a human would. While the continuous simulation reduces preparation time, evaluating the method is expensive, as a considerable number of games needs to be run. Unfortunately, the lack of computing power was an issue throughout the project. I was still able to perform the experiments I intended to and run enough games to get clear results, but I would have liked to do more.

All of the experimental results with Monte Carlo Tree Search were collected from 100 games per configuration. Although these 100 games were enough in this case for all the comparisons I made, the standard deviations of win rates were in worst cases nearly 5%. With such big error bars, it would have been difficult to make conclusive observations, had the difference in performance of the evaluated methods been smaller. Around 2,000 games with each configuration would be needed to get the standard deviation of the win rates of the agents down to 1%. This is as expected, as the game has highly stochastic elements: dice rolls, chance and community chest cards. Altogether, the experiments I presented in the project took around 500 hours of compute time on the student compute cluster. Therefore, to get very precise results, around a year worth

of compute time would be necessary with such a small cluster.

5.2 Future Work

While I was able to achieve the original goals of the project, I came up with further ideas throughout the process. I already mentioned a few improvements and adjustments, but there are a couple of additional things that could be explored which I will detail below.

5.2.1 Public Codebase

As there was no good Java-based open-source Monopoly implementation available, it could be beneficial to make my implementation of Monopoly open-source. This would encourage more people to try and use reinforcement learning to play this game, as they would not have to implement their own version of it. Furthermore, by using my codebase, it would be easy for anyone to use the same simplifications of the game as I did, thus making their results comparable to those presented in this project.

Some additional work would be necessary before my code could be used for future projects. With the integration of the MCTS learner, I had to expose several fields and methods in the Monopoly implementation that best practises would keep private. The code could use a cleanup, as there is still functionality that is specific to Monte Carlo Tree Search, and thus should be abstracted. The wrapper for Monopoly could be modified, so that it not only complies with the interface needed for MCTS, but would also make it easy to use another learning algorithm. Furthermore, in order to actually release the codebase, thorough documentation would be needed. While I have commented my code, I would have to double check, as they could be unclear to a third party. In order for it to be useful for someone to use or adapt this implementation for their needs, automated tests would be reasonable, and a high-level description of the system along with a user manual should be written.

5.2.2 Human Play

As the typed-MCTS agent can usually make decisions in around 10-20 seconds (with 10,000 rollouts), it could be tested against people in human trials. While this would currently be tedious, as there is no visual representation of the game, adding a minimalistic user interface would be relatively easy. It would be interesting to see how the agent fares against seasoned Monopoly players and see whether artificial intelligence can beat humans in this game.

Additionally, it would be beneficial to record the decisions made by humans, as the it could then be used to further improve the MCTS agent. Silver et al. (2016) used gameplay data from human experts to train a supervised model, which predicts expert actions in the game of Go. Such models are then used in Monte Carlo Tree Search to

improve value function estimation. It could be interesting to see if a similar approach could be combined with typed sampling and applied to Monopoly.

5.2.3 Comparison to Other Methods

Whereas the focus of this project is in comparing typed and uniform sampling, it would also be interesting to see how the typed-MCTS algorithm performs against other reinforcement learning approaches. Unfortunately, there are not many papers on playing Monopoly, but as previously mentioned, Bailis et al. (2014) used Q-learning to play Monopoly. They considered a simplified version of the game, where trading was not allowed.

Trading is a big part of the game and one of the main reasons for the large branching that makes Monopoly tricky to learn. As the primary benefit of typed sampling is due to reducing branching, Monopoly without trading might not benefit as much from it. Nevertheless, MCTS and Q-learning could be compared to see which works better for this game. Although I do not have access to the aforementioned Q-learning agent to play it against a typed-MCTS agent directly, I did implement the baseline they trained and evaluated their agent against (Bailis agent). Thus, it could be possible to use that baseline for MCTS simulations and evaluate the performance of typed-MCTS that way. However, they also made modifications to the game, some of which are different than the adjustments I made. Thus, even though the differences are not major, the results reported in Bailis et al. (2014) are not directly comparable to any results I would be able to get.

Should it be attempted to use MCTS for playing Monopoly without trading, run time has to be considered. As can be seen from Table 5.1, games where trading is disabled require nearly 7 times more decisions than those that allow trading. However, run time is only around three times longer without trading, which is due to there being less possible choices of actions at each decision - average branching without trading is more than three times smaller than when trades have to be decided. While the reduced branching is good for performance, the extreme length of the games might pose a problem for MCTS or will at least require a lot of computational power, which is another reason why I did not attempt such experiments in this project. Should in the future extra resources become available, MCTS could be compared to Q-learning, but to use the results from Bailis et al. (2014), game rules should be modified to match between the two approaches.

Trading	Branching	Depth	Run Time	Draw Rate
Enabled	9.03	336.14	8.37	0.00%
Disabled	2.69	2257.18	26.47	51.7%

Table 5.1: Comparison of average branching, depth and run time (in milliseconds) of games with trading disabled and enabled. Collected from 2,000 games of a typed random agent playing against 3 Bailis baseline agents. Draw is called after 10,000 turns without a winner. Simulated on a a single 2.4GHz core of a laptop.

Chapter 6

Conclusion

I set out to explore the effects of two-stage action sampling in Monte Carlo Tree Search on the game of Monopoly. The project was a success: I was able to implement a gaming environment that allowed me to simulate games of Monopoly and collect statistics, I designed three different heuristic baseline agents, and despite the limited computing power, managed to conduct experiments that compared typed and uniform sampling in Monte Carlo Tree Search. Unlike previous work, I did not remove trading from the game, making it more complicated to learn and play. While some simplifications and restrictions were required to make the simulations computationally feasible, the majority of those would be perfectly reasonable in real life gameplay.

This work demonstrates that typed sampling is superior - not only can it get better results in games, it is also significantly faster and does not require as many restrictions on the game as uniform sampling. As typed-MCTS was also highly effective in Settlers of Catan, it is likely that other games and problems with similar characteristics (imbalanced cardinalities of action types, large state and action spaces) would also benefit from action categorisation. It is especially useful in games, as the action types are inherent in the rules of the game and thus no extra work is needed to group the moves. Hence, Monte Carlo Tree Search with two-step sampling should be applied to more games, so its full potential could be utilised.

Bibliography

[1] Pieter Abbeel, Adam Coates, and Andrew Y Ng. "Autonomous helicopter aerobatics through apprenticeship learning". In: *The International Journal of Robotics Research* 29.13 (2010), pp. 1608–1639.

[2] Broderick Arneson, Ryan B Hayward, and Philip Henderson. "Monte Carlo tree search in Hex". In: *IEEE Transactions on Computational Intelligence and AI in Games* 2.4 (2010), pp. 251–258.

[3] Robert B Ash and Richard L Bishop. "Monopoly as a Markov Process". In: *Mathematics Magazine* 45.1 (1972), pp. 26–29.

[4] David Auger, Adrien Couetoux, and Olivier Teytaud. "Continuous upper confidence trees with polynomial exploration–consistency". In: *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*. Springer. 2013, pp. 194–209.

[5] P Bailis, Anestis Fachantidis, and I Vlahavas. "Learning to play monopoly: A Reinforcement learning approach". In: (Jan. 2014).

[6] Pragathi P. Balasubramani, V. Srinivasa Chakravarthy, Balaraman Ravindran, and Ahmed A. Moustafa. "An extended reinforcement learning model of basal ganglia to understand the contributions of serotonin and dopamine in risk-based decision making, reward prediction, and punishment learning". In: *Frontiers in Computational Neuroscience* 8 (2014), p. 47. ISSN: 1662-5188. DOI: 10.3389/fncom.2014.00047. URL: https://www.frontiersin.org/article/10.3389/fncom.2014.00047.

[7] Richard Bellman. *Dynamic Programming*. 1st ed. Princeton, NJ, USA: Princeton University Press, 1957.

[8] Francesco Bertoluzzo and Marco Corazza. "Reinforcement Learning for Automated Financial Trading: Basics and Applications". In: *Recent Advances of Neural Network Models and Applications: Proceedings of the 23rd Workshop of the Italian Neural Networks Society (SIREN), May 23-25, Vietri sul Mare, Salerno, Italy*. Ed. by Simone Bassis, Anna Esposito, and Francesco Carlo Morabito. Cham: Springer International Publishing, 2014, pp. 197–213. ISBN: 978-3-319-04129-2. DOI: 10.1007/978-3-319-04129-2_20. URL: https://doi.org/10.1007/978-3-319-04129-2_20.

[9] Dimitri P Bertsekas. *Dynamic programming and optimal control*. Vol. 1. 2. 1995.

[10] Blizzard Entertainment. *StarCraft*. 1998.

[11] Blizzard Entertainment. *StarCraft II: Wings of Liberty*. 2010.

[12] BoardGameGeek. *Monopoly (1933)*. n.d. URL: https://boardgamegeek.com/boardgame/1406/monopoly (visited on 12/21/2017).

[13] Cameron B Browne et al. "A survey of monte carlo tree search methods". In: *IEEE Transactions on Computational Intelligence and AI in games* 4.1 (2012), pp. 1–43.

[14] Tristan Cazenave, Flavien Balbo, and Suzanne Pinson. "Using a monte-carlo approach for bus regulation". In: *Intelligent Transportation Systems, 2009. ITSC'09. 12th International IEEE Conference on*. IEEE. 2009, pp. 1–6.

[15] Alex J. Champandard. *Monte-Carlo Tree Search in TOTAL WAR: ROME IIs Campaign AI*. 2014. URL: http://aigamedev.com/open/coverage/mcts-rome-ii/ (visited on 01/14/2018).

[16] Guillaume Chaslot, Mark HM Winands, and H Jaap van Den Herik. "Parallel monte-carlo tree search". In: *Computers and Games* 5131 (2008), pp. 60–71.

[17] Yiling Chen. *Games with Perfect Information*. Sept. 2012. URL: http://www.eecs.harvard.edu/cs286r/courses/fall12/presentations/lecture2.pdf (visited on 12/23/2017).

[18] Benjamin E Childs, James H Brodeur, and Levente Kocsis. "Transpositions and move groups in Monte Carlo tree search". In: *Computational Intelligence and Games, 2008. CIG'08. IEEE Symposium On*. IEEE. 2008, pp. 389–395.

[19] Benjamin E Childs, James H Brodeur, and Levente Kocsis. "Transpositions and move groups in Monte Carlo tree search". In: *Computational Intelligence and Games, 2008. CIG'08. IEEE Symposium On*. IEEE. 2008, pp. 389–395.

[20] Rémi Coulom. "Efficient selectivity and backup operators in Monte-Carlo tree search". In: *International conference on computers and games*. Springer. 2006, pp. 72–83.

[21] Peter I Cowling, Colin D Ward, and Edward J Powley. "Ensemble determinization in monte carlo tree search for the imperfect information card game magic: The gathering". In: *IEEE Transactions on Computational Intelligence and AI in Games* 4.4 (2012), pp. 241–257.

[22] [kinch2002] Daniel Sheppardl. *Monopoly What Can Happen on the First Turn?* Sept. 2015. URL: http://www.kinch2002.com/monopoly-what-can-happen-on-the-first-turn/.

[23] [kinch2002] Daniel Sheppardl. *Monopoly Board*. Image: Board.jpg. Sept. 2015. URL: http://www.kinch2002.com/wp-content/uploads/2015/09/Board.jpg.

[24] Charles Darrow. *Monopoly: The Fast-Dealing Property Trading Game*. 1935.

[25] DeepMind. *The story of AlphaGo so far*. 2017. URL: https://deepmind.com/research/alphago/ (visited on 12/23/2017).

[26] Mihai Dobre and Alex Lascarides. "Exploiting Action Categories in Learning Complex Games". In: *IEEE Technically Sponsored Intelligent Systems Conference (IntelliSys 2017)*. Jan. 2017.

[27] Rudiger Dorn. *Genoa*. 2001.

[28] Markus Enzenberger, Martin Muller, Broderick Arneson, and Richard Segal. "Fuegoan open-source framework for board games and Go engine based on Monte Carlo tree search". In: *IEEE Transactions on Computational Intelligence and AI in Games* 2.4 (2010), pp. 259–270.

[29] Vlad Firoiu, William F. Whitney, and Joshua B. Tenenbaum. "Beating the World's Best at Super Smash Bros. with Deep Reinforcement Learning". In: *CoRR* abs/1702.06230 (2017). arXiv: 1702.06230. URL: http://arxiv.org/abs/1702.06230.

[30] Timothy Furtak and Michael Buro. "Recursive Monte Carlo search for imperfect information games". In: *Computational Intelligence in Games (CIG), 2013 IEEE Conference on*. IEEE. 2013, pp. 1–8.

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

[32] Sylvain Gelly and David Silver. "Monte-Carlo tree search and rapid action value estimation in computer Go". In: *Artificial Intelligence* 175.11 (2011), pp. 1856–1875.

[33] Markus Guhe and Alex Lascarides. "Trading in a multiplayer board game: Towards an analysis of non-cooperative dialogue". In: *Proceedings of the Cognitive Science Society*. Vol. 34. 34. 2012.

[34] Guiness World Records. *Most popular board game*. 1999. URL: http://www.guinnessworldrecords.com/search/applicationrecordsearch?term=most+popular+board+game&contentType=record (visited on 12/21/2017).

[35] HAL Laboratory. *Super Smash Bros Melee*. 2001.

[36] A.J. Hayter. *Probability and Statistics for Engineers and Scientists*. Cengage Learning, 2012. ISBN: 9781111827045. URL: https://books.google.co.uk/books?id=Z3lr7UHceYEC.

[37] Takuya Hiraoka, Kallirroi Georgila, Elnaz Nouri, David R Traum, and Satoshi Nakamura. "Reinforcement Learning in Multi-Party Trading Dialog." In: 2015.

[38] Luca Iennaco. *Kingsburg*. 2007.

[39] Zhengyao Jiang, Dixing Xu, and Jinjun Liang. "A Deep Reinforcement Learning Framework for the Financial Portfolio Management Problem". In: (June 2017).

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

[41] Richard J Lorentz. "Improving monte–carlo tree search in havannah". In: *International Conference on Computers and Games*. Springer. 2010, pp. 105–115.

[42] Shimpei Matsumoto, Noriaki Hirosue, Kyohei Itonaga, Kazuma Yokoo, and Hisatomo Futahashi. "Evaluation of simulation strategy on single-player Monte-Carlo tree search and its discussion for a practical scheduling problem". In: *Proceedings of the International MultiConference of Engineers and Computer Scientists*. Vol. 3. 2010, pp. 2086–2091.

[43] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. "Playing atari with deep reinforcement learning". In: *arXiv preprint arXiv:1312.5602* (2013).

[44] Andrew Y Ng, Adam Coates, Mark Diel, Varun Ganapathi, Jamie Schulte, Ben Tse, Eric Berger, and Eric Liang. "Autonomous inverted helicopter flight via reinforcement learning". In: *Experimental Robotics IX*. Springer, 2006, pp. 363–372.

[45] OpenAI. *Dota 2*. 2017. URL: https://blog.openai.com/dota-2/ (visited on 01/12/2018).

[46] Tom Pepels, Mark HM Winands, and Marc Lanctot. “Real-time monte carlo tree search in ms pac-man”. In: *IEEE Transactions on Computational Intelligence and AI in games* 6.3 (2014), pp. 245–257.

[47] Joelle Pineau, Geoffrey Gordon, and Sebastian Thrun. “Anytime point-based approximations for large POMDPs”. In: *Journal of Artificial Intelligence Research* 27 (2006), pp. 335–380.

[48] Marc JV Ponsen, Geert Gerritsen, and Guillaume Chaslot. “Integrating Opponent Models with Monte-Carlo Tree Search in Poker.” In: 2010.

[49] Edward J Powley, Daniel Whitehouse, and Peter I Cowling. “Monte carlo tree search with macro-actions and heuristic route planning for the physical travelling salesman problem”. In: *Computational Intelligence and Games (CIG), 2012 IEEE Conference on*. IEEE. 2012, pp. 234–241.

[50] Martin L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. 1st. New York, NY, USA: John Wiley & Sons, Inc., 1994. ISBN: 0471619779.

[51] Arpad Rimmel, Fabien Teytaud, and Tristan Cazenave. “Optimization of the nested monte-carlo algorithm on the traveling salesman problem with time windows”. In: *Applications of Evolutionary Computation* (2011), pp. 501–510.

[52] Uwe Rosenberg. *Bohnanza*. 1997.

[53] Farhang Sahba, Hamid R Tizhoosh, and Magdy MA Salama. “A reinforcement learning framework for medical image segmentation”. In: *Neural Networks, 2006. IJCNN’06. International Joint Conference on*. IEEE. 2006, pp. 511–517.

[54] Ahmad EL Sallab, Mohammed Abdou, Etienne Perot, and Senthil Yogamani. “Deep reinforcement learning framework for autonomous driving”. In: *Electronic Imaging* 2017.19 (2017), pp. 70–76.

[55] A. L. Samuel. “Some Studies in Machine Learning Using the Game of Checkers”. In: *IBM Journal of Research and Development* 3.3 (July 1959), pp. 210–229. ISSN: 0018-8646. DOI: 10.1147/rd.33.0210.

[56] Shai Shalev-Shwartz, Shaked Shammah, and Amnon Shashua. “Safe, Multi-Agent, Reinforcement Learning for Autonomous Driving”. In: *CoRR* abs/1610.03295 (2016). arXiv: 1610.03295. URL: http://arxiv.org/abs/1610.03295.

[57] Jiefu Shi and Michael L Littman. “Abstraction methods for game theoretic poker”. In: *International Conference on Computers and Games*. Springer. 2000, pp. 333–345.

[58] David Silver et al. “Mastering the game of Go with deep neural networks and tree search”. In: *Nature* 529.7587 (2016), pp. 484–489.

[59] David Silver et al. “Mastering the game of Go without human knowledge”. In: 550 (Oct. 2017), pp. 354–359.

[60] Tom Simonite. *Facebook Quietly Enters StarCraft War for AI Bots, and Loses*. 2017. URL: https://www.wired.com/story/facebook-quietly-enters-starcraft-war-for-ai-bots-and-loses/ (visited on 01/12/2018).

[61] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. Vol. 1. 1. 1998.

[62] Yasuhiro Tanabe, Kazuki Yoshizoe, and Hideki Imai. “A study on security evaluation methodology for image-based biometrics authentication systems”. In:

Biometrics: Theory, Applications, and Systems, 2009. BTAS'09. IEEE 3rd International Conference on. IEEE. 2009, pp. 1–6.

[63] Klaus Teuber. *The settlers of Catan : award-winning game of discovery, settlement & trade*. Skokie, IL, 2007.

[64] The Creative Assembly. *Total War: Rome II*. 2013.

[65] Henk Tijms. *Understanding probability*. Cambridge University Press, 2012.

[66] Michael Tummelhofer. *Stone Age*. 2008.

[67] Valve Corporation. *Dota 2*. 2013.

[68] François Van Lishout, Guillaume Chaslot, and Jos WHM Uiterwijk. "Monte-Carlo tree search in backgammon". In: (2007).

[69] Oriol Vinyals et al. "StarCraft II: A New Challenge for Reinforcement Learning". In: *CoRR* abs/1708.04782 (2017). arXiv: 1708.04782. URL: http://arxiv.org/abs/1708.04782.

[70] Waddingtons Games. *The Property Trading Board Game Rules*. 1996. URL: http://www.tmk.edu.ee/~creature/monopoly/download/official_rules_gathering/instructions.pdf.

[71] Christopher JCH Watkins and Peter Dayan. "Q-learning". In: *Machine learning* 8.3-4 (1992), pp. 279–292.

[72] Daniel Whitehouse, Edward Jack Powley, and Peter I Cowling. "Determinization and information set Monte Carlo tree search for the card game Dou Di Zhu". In: *Computational Intelligence and Games (CIG), 2011 IEEE Conference on*. IEEE. 2011, pp. 87–94.

[73] Qiang Yang. "Hierarchical Planning". In: *Intelligent Planning: A Decomposition and Abstraction Based Approach*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1997, pp. 163–188. ISBN: 978-3-642-60618-2. DOI: 10.1007/978-3-642-60618-2_10. URL: https://doi.org/10.1007/978-3-642-60618-2_10.

[74] Shi-Jim Yen and Jung-Kuei Yang. "Two-stage Monte Carlo tree search for Connect6". In: *IEEE Transactions on Computational Intelligence and AI in Games* 3.2 (2011), pp. 100–118.

[75] Yurong You, Xinlei Pan, Ziyan Wang, and Cewu Lu. "Virtual to Real Reinforcement Learning for Autonomous Driving". In: *CoRR* abs/1704.03952 (2017). arXiv: 1704.03952. URL: http://arxiv.org/abs/1704.03952.

[76] Yufan Zhao, Donglin Zeng, Mark A Socinski, and Michael R Kosorok. "Reinforcement learning strategies for clinical trials in nonsmall cell lung cancer". In: *Biometrics* 67.4 (2011), pp. 1422–1433.