Monte Carlo Tree Search for Carcassonne

Bachelor's Thesis

Max Jappert
max.jappert@unibas.ch
18-052-647

Examiner: Prof. Dr. Malte Helmert
Supervisor: Dr. Silvan Sievers

Faculty of Science at the University of Basel
Department of Mathematics and Computer Science
Artificial Intelligence
https://ai.dmi.unibas.ch/

12.06.2022

Abstract

Carcassonne is a tile-based board game with a large state space and a high branching factor and therefore poses a challenge to artificial intelligence. In the past, Monte Carlo Tree Search (MCTS), a search algorithm for sequential decision-making processes, has been shown to find good solutions in large state spaces. MCTS works by iteratively building a game tree according to a tree policy. The profitability of paths within that tree is evaluated using a default policy, which influences in what directions the game tree is expanded. The functionality of these two policies, as well as other factors, can be implemented in many different ways. In consequence, many different variants of MCTS exist. In this thesis, we applied MCTS to the domain of two-player Carcassonne and evaluated different variants in regard to their performance and runtime. We found significant differences in performance for various variable aspects of MCTS and could thereby evaluate a configuration which performs best on the domain of Carcassonne. This variant consistently outperformed an average human player with a feasible runtime.

Contents

List of Tables

List of Figures

1 Introduction

Carcassonne is a tile-based board game for between two and five players (Wrede, 2005) which poses an interesting challenge to artificial intelligence. The board is iteratively built by the players, who try to shape it to their own advantage. In consequence, a significant branching factor emerges via the possibilities of expanding the board over the course of 72 rounds of play, leading to more than 10^{40} possible board configurations (Heyden, 2009). In addition, the different possibilities of placing meeples (see Section 2.1) add to the branching. Consequently, there are roughly 10^{192} possible states the game can be in, as shown by Heyden (2009).

A comparable domain in that regard is the game Go, for which, despite decades of research, a truly strong computer player wasn't developed for a long time due to its large branching factor and consequently large state space (Bouzy and Cazenave, 2001). When AlphaGo beat Fan Hui back in 2015, it was the first time a program had won against a professional Go player (Silver *et al.*, 2016), nearly 20 years after DeepBlue beat Garry Kasparov, the world's leading Chess player at the time.

AlphaGo used the Monte Carlo Tree Search (MCTS) framework for handling Go's large state space (Silver *et al.*, 2016). In the past few years, the MCTS framework has been shown to handle other domains with a large state space well (Browne *et al.*, 2012), e.g. Hex (Arneson *et al.*, 2010), Lines of Action (Winands *et al.*, 2010), Settlers of Catan (Szita *et al.*, 2009) and self-driving cars (Lenz *et al.*, 2016).

In regard to previous research on computers playing Carcassonne, Heyden (2009) concluded that the Star2.5 algorithm, an adaptation of Minimax search with α-β pruning, managed to outperform both an MCTS variant, as well as advanced human players in Carcassonne. However, she only tested one MCTS variant, and fails to describe it in any great detail. Additionally, she does not report on the runtimes of her implementations. Ameneyro *et al.* (2020), on the other hand, report to have found a variant of MCTS which "consistently outperformed the Star2.5 algorithm" (p. 2343), attributing this to the ability of MCTS to "find long-term strategies" (p. 2349). Their MCTS variant is also not described in much detail. Both papers only mention that they used UCT as a tree policy (see Section 3.3.3).

Ameneyro *et al.* (2020) also considered MCTS-RAVE, which uses the *all-moves-as-first* (AMAF) heuristic. MCTS-RAVE assumes that there will be a similar outcome from an action regardless of when it was performed. It does this by assuming an AMAF-value for each action regardless of the state it was played at. MCTS-RAVE proved to be profitable for Go (Gelly and Silver, 2011), yet Ameneyro *et al.* (2020) concluded that MCTS performed better without the AMAF heuristic.

In this thesis, we will be considering and optimising Monte Carlo Tree Search on the domain of two-player Carcassonne. We will be testing and comparing different variants in regard to their performance and runtime, in order to approximate an optimal MCTS variant.

In the next chapter, we will be introducing Carcassonne and the basic idea of MCTS in an effort to make the thesis self-contained. Thereafter we will be introducing relevant variable aspects of MCTS. In Section 4 we will cover the detailed results of our experiments, in order to reach a conclusion in Section 5.

2 Background

2.1 Carcassonne

Carcassonne is a tile-based board game for between 2 and 5 players which involves placing a total of 72 tiles to create a landscape consisting of roads, cities, fields and monasteries. The players aim to claim these in order to obtain points, which is done by placing meeples. It is a perfect information game despite the randomness of drawing the tile due to the fact that all players share their information (Osborne and Rubinstein, 1994). For this thesis, we will only be considering Carcassonne for 2 players and will therefore introduce it as such.

Each player is dealt 7 meeples. The players take turns in executing the following sequence of actions:

1. Drawing a tile from the deck.
2. Placing a tile.
3. Optionally placing a meeple on the placed tile.

The following sections will cover the rules involved in the three steps.

2.1.1 Placing a tile

Each tile has a distinct pattern, which can include monasteries, intersections and/or parts of cities, roads and fields. The tiles can only be placed such that the roads, fields and cities are continued. In the rare case that this shouldn't be possible, the card is discarded and the player has to draw a new tile.

2.1.2 Placing a follower

The player may place a meeple on and thereby occupy either a monastery, a field, or on an unfinished road or city on the card they have placed. The following rules must be adhered to:

- Only a single meeple may be placed at once.
- The meeple may only be placed on the card which has been drawn in the previous step.
- The meeple can take the role of a thief, a knight, a monk or a farmer by placing them on a road, city, monastery or field respectively.
- On a single connected road, city, field or monastery only a single meeple of any player can be placed at any given time, regardless of their distance.[1]

[1] Despite this rule, it is possible to have a city, road or field occupied by multiple players, if two unfinished roads, cities or fields are both occupied and later connected.

2.1.3 Counting the points

Points are generated by completing parts of the map while having followers placed on the completed part. There are three ways to obtain points during the game:

1. A road is completed if both ends reach a border, which can either be an intersection or a city. The player whose follower is placed on the completed road earns **1 point** per tile which the road consists of. If there are multiple followers placed on the road, the player with the most followers gets the points. If both players have the same amount of followers on the road, both get the points.

2. A city is completed once it's completely surrounded by walls, without gaps. The player who occupies the city earns **2 points** for each tile the city consists of, as well as an extra **2 points** per pennant, which can occur on tiles with cities on them. If there are multiple followers placed on a city, the player with the most followers gets the points. If both players have the same amount of followers on a city, then both get the points.

3. A monastery is completed if it is fully surrounded by 8 tiles. The player who occupies the monastery then earns **9 points**.

At the end of the game, the points for occupying fields are distributed. Fields are separated from each other by roads and cities. The field is occupied by whichever player has placed the most followers on a given field. The occupant of a field gains **3 points** per completed city which borders on the field. If both players have the same amount of followers on a field, then both get the points.

Additionally, at the end of the game each occupied, yet non-completed road, city and monastery the occupying player gains **1 point** per involved tile.

2.2 Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) is an uninformed search algorithm, used for sequential decision making processes. It combines the Monte Carlo method, with reinforcement learning in order to approximate an optimal strategy for traversing a state space in form of a tree (Browne *et al.*, 2012). We define a state space as the set of all possible states which the game can be in. The state space can be represented as a game tree, whereby the nodes represent the states and the edges represent actions which lead from one state to the next. Thereby for a parent node v_p with a child v_c, the parent-child relation represent the fact that the state represented by v_c can be reached by executing an action at the state represented by v_p. The Monte Carlo method is hereby defined as a type of algorithm which uses random sampling as a means of obtaining numerical results (Metropolis and Ulam, 1949). Reinforcement learning has been defined by Kaelbling *et al.* (1996) as "the problem faced by an agent that must learn behavior through trial-and-error interactions with a dynamic environment" (p. 237).

MCTS manages to deal with large state spaces (i.e., those represented by a game tree with a large branching factor) by iteratively expanding the game

tree rather than trying to navigate the entire state space. It does this by approximating which directions are profitable to explore further and consequently ignores those which it considers not to be.

Each node v stores the values $Q(v) \in \mathbb{R}_0^+$ and $N(v) \in \mathbb{N}_0$. $Q(v)$ denotes the approximated payoff which can be expected when the game is in the state represented by node v. $N(v)$ denotes how often v was visited during training (Browne *et al.*, 2012). The payoff is a measure of the quality of an action or a path of actions. In the case of Carcassonne we will be using the score as the payoff.

2.2.1 Training Steps

The training works by incrementally and asymmetrically building a game tree. In the beginning, the tree only consists of the root node v_0 denoting the start state s_0. MCTS trains by executing $n \gg 1$ training iterations (Browne *et al.*, 2012). Each training iteration of MCTS consists of the following four steps:

1. **Selection:** Starting from the root node v_0, the game tree is traversed according to the *tree policy* until a leaf node is reached. The tree policy is a non-deterministic function mapping each node v to one of its child nodes v_j or to itself if it does not have any children. This function is recursively called on each child node it returns, until a leaf node is reached. The goal of a tree policy is to explore the state space by guiding the traversal of the game tree. Thereby it needs to balance the exploitation of previously profitable choices with the need for exploring new parts of the state space.

2. **Expansion:** If the reached leaf node v does not represent the end of the game, i.e., it does not represent a terminal state, and has successor states which aren't yet represented by child nodes, then the node is expanded. The expansion consists of adding either one or many child nodes representing legal actions which can be played at the state s represented by the leaf node v. One of the created child nodes is then visited.

3. **Simulation:** In the Simulation step, the payoff which reaching the given leaf node will produce is approximated using the *default policy*. This can either be done directly using a heuristic function or by performing a playout, which makes use of the Monte Carlo method. In a playout, starting from the node visited in the Expansion step, the game is played until the end. The most common implementation of the default policy is a playout, where a random legal action is played at every state (Browne *et al.*, 2012).

4. **Backpropagation:** The payoff yielded by the Simulation step (denoted by Δ) is propagated back up the tree, along the path chosen in the Selection step. Thereby each node's Q-value is updated as $Q(v) \leftarrow \frac{Q(v)+\Delta}{N(v)}$ for all visited nodes v. The visit count $N(v)$ is also updated as $N(v) \leftarrow N(v) + 1$.

Consequently, for each node v, $Q(v)$ takes on the value defined in Equation (1).

$$Q(v_j) = \frac{1}{N(v_j)} \sum_{i=1}^{N(v)} \mathbb{I}_i(v_j) z_i \tag{1}$$

Thereby v_j denotes a child node of v. $\mathbb{I}_i$ is an indicator function returning 1 if v_j was selected at the i-th visit of v, and z_i is the payoff that was achieved in the playout after v's i-th visit.

MCTS repeatedly playing games against itself and updating its strategy according to the outcome of each game denotes the reinforcement learning aspect of the algorithm. This makes MCTS flexible, since it does not necessarily require domain-specific knowledge to produce good results (Browne *et al.*, 2012), thereby making it applicable to any domain which can be modelled using a tree. Despite this, domain-specific knowledge can be implemented to potentially improve or change performance.

Figure 1 provides a visualisation of the training process of MCTS, consisting of the four steps described above.

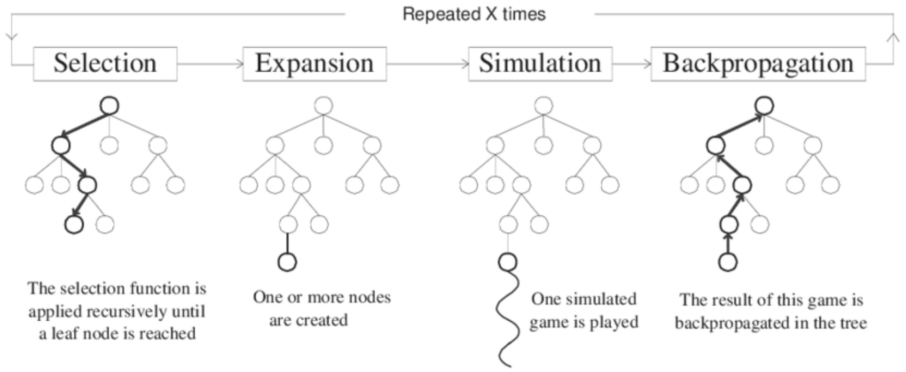


Figure 1: A visual representation of the training process with a playout as a default policy, whereby X denotes the number of training iterations (Chaslot *et al.*, 2008, p. 216).

2.2.2 Asymptotic Optimality

A MCTS variant is asymptotically optimal if the probability of MCTS deciding on a sub-optimal action converges to 0. This implies that, given infinite time and memory, the Q-values of all nodes converge to their optimal value Q^*. Consequently, an asymptotically optimal MCTS variant allows the game tree to converge to the Minimax tree (Browne *et al.*, 2012).

3 Implementation

The previous section offers a high-level introduction to how MCTS works. This introduction barely reveals any concrete details on how the individual steps of the algorithm work, since they are highly dependant on how various aspects of MCTS are implemented. For example, the choice of tree policy and default policy are not predefined and significantly influence the algorithm's performance (Browne *et al.*, 2012). Additionally, there are many aspects which can be tweaked when implementing a MCTS variant. In this section, we will be describing the variable aspects of MCTS which we will be considering in the evaluation in Section 4 or find noteworthy regarding some other aspect. The

variants this section will introduce are not complete implementations, but rather different ways of implementing various aspects of the algorithm. These aspects regard the form of the game tree, the heuristic function, as well as different ways of implementing each of the four steps described in Section 2.2. In the evaluation in Section 4 we will be assembling variants composed of these aspects and testing their performance.

3.1 Game Tree

Referring back to Section 2.1, each action in Carcassonne consists of three separate steps:

1. **Chance Step:** The player draws a tile randomly from the deck.
2. **Placement Step:** Given the drawn tile, the player decides on where and with which rotation to place it.
3. **Meeple Step:** Given the placement on the board, the player has the option to place a meeple on the tile.

As such, we model the game tree in a similar fashion, whereby every action consists of three layers of nodes which are independently expanded. Each layer's nodes represent one of the three action types. The placement nodes denote the action of deciding on a placement of the tile, the meeple nodes denote the action of deciding on a meeple placement and the chance nodes denote the random action of drawing a tile.

If each action were to be modelled on a single layer, we would have an average branching factor of 55 (Heyden, 2009). In our implementation, we have a branching factor between 4 and 30, which is much more manageable.

We implemented two ways for dealing with the random aspect of drawing tiles as part of the game tree. We will discuss these in the following subsections.

3.1.1 Single Game Tree

In most literature MCTS uses a single game tree and as such it stands to reason that modelling the random aspect of Carcassonne as part of a single game tree makes sense. We did this by adding a placement node for each possible tile which can be drawn at a given chance node. When the tree policy reaches a chance node, it does not pick the child node according to the tree policy, but rather picks a random child. Thereby the random nature of the chance node is considered during the Selection step. During the Expansion step, we expand all three layers, such that each Expansion step expands one action.

Figure 2 is a visualisation of a partially expanded game tree that our implementation generated during the training process after 50 training iterations.[2] The red nodes denote the placement nodes, the green nodes are the meeple nodes and the blue nodes are the chance nodes.

[2]The trees in Figures 2 and 3 were generated using Graphviz (`https://www.graphviz.org/`).

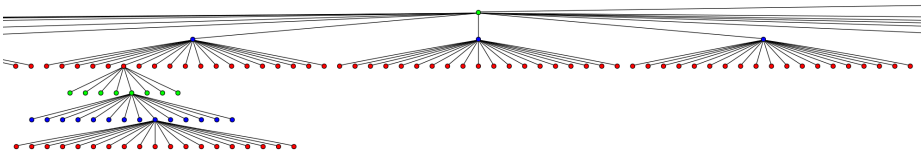


Figure 2: An excerpt of a game tree generated by our implementation.

This method assumes that drawing a tile corresponds to a uniform distribution over the remaining tiles in the deck. In other words, the algorithm assumes that the probability of drawing a tile of which four are present in the deck is identical to drawing a tile of which only one is left. We implemented it as such to keep the size of the game tree within manageable limits. It may be interesting, for future work, to explore the possibility of adding a node for each individual tile remaining in the deck, such that MCTS takes into account the probability of drawing a particular tile.

3.1.2 Ensemble MCTS

The drawback of using a single game tree is that a significant part of the game tree (roughly a third) is occupied by nodes which are only randomly visited and do not add to the knowledge on the state space. This is due to the fact that their Q-values and visit counts N are insignificant (because they get selected randomly). The proportion of such placement nodes when using a single game tree can be observed in Figure 2, where they are denoted by the red nodes.

As an alternative, we will consider Ensemble MCTS, as described by Sievers and Helmert (2015) and Mirsoleimani *et al.* (2015). Thereby the tree assumes a fixed permutation of the deck and thereby constructs the game tree as if there were no randomness. To counteract overfitting, multiple game trees assuming different permutations are constructed, whose results are merged in order to make a decision on the move to pick. This merge takes on the form of a vote, whereby each tree votes for one action. MCTS then chooses the action with the most votes, or one of the actions randomly if no action has a majority.

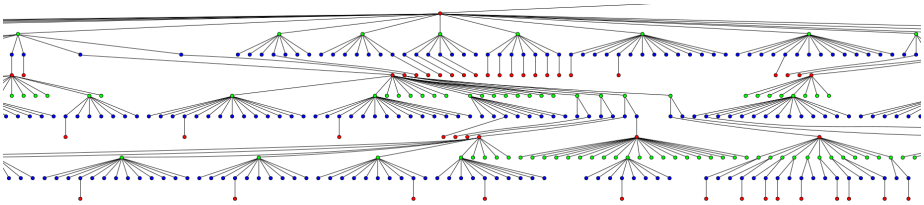


Figure 3: An excerpt of a game tree generated by our implementation using Ensemble MCTS.

Our implementation remains the same in regard to the differentiation between the three types of nodes. What changes is that each chance node generates exactly one placement node corresponding to the tile it draws from the fixed deck permutation it assumes, instead of generating a placement node for each possible tile it could draw. This can be seen in Figure 3. As in Figure 2, the

red nodes denote the placement nodes, the green nodes are the meeple nodes and the blue nodes are the chance nodes.

3.2 Heuristic Function

We implemented two heuristic functions, $h_P : S \times A \to \mathbb{R}_0^+$ for the set of states S and the set of legal placements of a given tile A and $h_M : S \times A \times M \to \mathbb{R}_0^+$ for the set of legal meeple placements M. The functions h_P and h_M value state-action pairs by considering actions which immediately generate points as more valuable than actions which do not or generate less.

3.3 Tree Policies

As previously mentioned, MCTS does not have a pre-defined tree policy. As such, the choice of tree policy is not fixed and will therefore influence the performance of the algorithm. In this chapter we will be introducing the tree policies which we have implemented and will evaluate in regard to their performance in Section 4.

3.3.1 Greedy

A naive implementation of the tree policy would involve mapping the child node v_j with the highest Q-value to the parent node node v (Gelly and Silver, 2011). We will call this tree policy Greedy, since it approaches the problem of deciding on how to expand the game tree by exclusively considering the locally optimal choice. MCTS with a Greedy tree policy does not give any incentive to explore unknown parts of the game tree, thereby failing to consider that the local optimum usually does not correspond to the global optimum.

3.3.2 ε-Greedy

An ε-Greedy tree policy expands on the Greedy tree policy by allowing for exploration of the game tree. This is achieved by returning a random child node with a probability of $\varepsilon \in [0, 1]$ and selecting a node greedily with a probability of $1-\varepsilon$. The probability ε can either be stated as a scalar or take on a gradually decaying value, such as $\varepsilon = 1/t$ for iteration t of the training process. We will call this decaying variant Decaying ε-Greedy.

3.3.3 Upper Confidence Bounds for Trees (UCT)

Upper Confidence Bound for Trees (UCT) was introduced by Kocsis and Szepesvári (2006) and expands on the upper confidence bounds policy UCB1 for multi-armed bandit problems (Auer *et al.*, 2002) by extending it for the use on trees, i.e., on sequential decision making processes. A K-armed bandit problem is defined by random variables $X_{i,n}$ for $1 \leq i \leq K$ and $n \geq 1$, whereby i denotes which of the K decisions can be made and n denotes the amount of times the decision will have been made. $X_{i,1}, ..., X_{i,n}$ are independently and identically distributed, whereby the distribution is not known. The approximated mean μ_j of X_j for state s_j corresponds to the Q-value of the node representing the given state, such that $\bar{X}_j = Q(v_j)$ holds. This implies that $Q(v_j)$ corresponds to the expected payoff when reaching the state represented by node v_j.

For each node v, UCT returns the child node v_j according to equation (2) (Browne *et al.*, 2012).

$$\mathrm{UCT}(v) = \arg\max_{v_j} Q(v_j) + 2c\sqrt{\frac{2\ln N(v)}{N(v_j)}} \tag{2}$$

N is defined as in Equation (1) and c is an exploration constant, a hyperparameter which can be adjusted in order to specify the degree of exploration. If $N(v_j) = 0$ holds, then $\mathrm{UCT}(v) = v_j$ holds, such as to guarantee that each node gets visited at least once. The UCT function encourages exploitation as a measure of the node's value, i.e., ensuring that a node is more likely to be chosen if it has previously yielded a high payoff in the Simulation steps. The exploration term $2c\sqrt{\frac{2\ln N(v)}{N(v_j)}}$ encourages exploration by being large if the node hasn't been visited often. Thereby nodes which haven't been visited and therefore evaluated extensively continue to be considered and aren't forgotten. The value of the exploration term c influences the degree of exploration in the Selection step.

UCT has been a very prominent tree policy. This is likely due to its early success in computer Go (e.g., Gelly and Silver, 2011). Additionally, Kocsis and Szepesvári (2006) proved that using UCT as a tree policy allows for asymptotically optimal MCTS variants.

3.3.4 UCT-Tuned

Auer *et al.* (2002) suggest UCB1-Tuned as an enhancement for UCB1. It replaces the exploration term of UCT, such that the UCT-Tuned tree policy is defined as in Equation (3).

$$\text{UCT-Tuned}(v) = \arg\max_{v_j} Q(v_j) + 2c\sqrt{\frac{\ln N(v)}{N(v_j)}\min\left\{\frac{1}{4}, V_j\right\}} \tag{3}$$

Thereby V_j is defined as in Equation (4).

$$V_j = \left(\frac{1}{N(v_j)}\sum_{\tau=1}^{N(v_j)} z_{j,\tau}^2\right) - \bar{X}_j^2 + \sqrt{\frac{2\ln t}{N(v_j)}} \tag{4}$$

$z_{j,\tau}$ denotes the payoff in the τ-th visit of s_j during the selection phase. The integer t denotes the number of training iterations at that point. Thus, the tree policy using UCB1-Tuned explores increasingly less, corresponding to the fact that the state space becomes increasingly known.

3.3.5 Boltzmann Exploration

Boltzmann exploration returns a child node proportionally to the Gibbs-Boltzmann distribution, such that the probability of selecting child node v_j is defined as in Equation (5) (Riveret *et al.*, 2014).

$$\mathbb{P}(v) \propto e^{\frac{\bar{X}_j}{\tau}} \tag{5}$$

The Gibbs-Boltzmann distribution has its main application in statistical mechanics, used for determining the probability of a system being in a state s_j

given its energy $\bar{X}_j$ and temperature τ. In regard to using it in a tree policy for MCTS, $\bar{X}_j$ is defined as for UCT and τ is simply a constant to avoid the term growing past the limits of computational feasibility (Riveret *et al.*, 2014).

3.4 Heuristic Tree Policy

It is also possible to use the heuristic functions as a tree policy. Thereby the choice of meeple nodes is guided by h_P and the choice of chance nodes is guided by h_M, as defined in Section 3.2.

3.5 Default Policies

The default policy determines how the simulation is performed, i.e., how the payoff Δ is approximated for the leaf nodes reached during the Selection step. In this section we will be describing the default policies that we will be considering during the evaluation in Section 4.2.4, as well as one default policy we find noteworthy in regard to potential future research.

3.5.1 Random Playout

In its most basic form, MCTS uses a random playout as a default policy. Thereby the game is simulated to the end by sampling each action according to a uniform distribution over the set of all legal actions at every decision point. Implemented as such, MCTS is an uninformed algorithm which works without requiring domain-specific knowledge and is applicable in the same form for any state space which can be modelled as a tree. This is usually the fastest way of performing a playout as part of the Simulation step, since it requires relatively little computational power to uniformly sample an action given a set of actions (Browne *et al.*, 2012).

While it is most common for a random playout to perform a single playout per training iteration (Browne *et al.*, 2012), we will later consider how using multiple playouts during the Simulation step influences performance.

3.5.2 Heuristic Playout

On the other hand, as has been mentioned extensively (e.g., by Silver *et al.* (2017a)), adding domain-specific knowledge can lead to better results. If the default policy consists of informed moves, then the generated payoff much more accurately reflects the true payoff of a state, since the playout more closely reflects optimal play. This is achieved by using a heuristic function and playing the game such that for every round the move is picked which maximises the heuristic function.

A heuristic playout could also be implemented to only perform a limited amount of actions per playout, instead of simulating the game until the end, in order to reduce the runtime. We didn't test this, but mention it for potential future research.

Additionally, it has been shown that using a shallow α-β search as a default policy can increase the performance for certain domains, such as for Lines of Action (Winands and Björnsson, 2009). This is impractical for the domain of Carcassonne, due to the large branching factor. We observed that a Minimax

playout for Carcassonne with a depth of $d = 2$ requires, on average, around 30 seconds to compute each move, which is infeasibly long for use as a playout for the default policy. As such, we will not be considering a shallow α-β search as a default policy.

3.5.3 Direct Heuristic

An alternative to executing an actual playout (in the sense of simulating entire games per training iterations) is to use the heuristic function to directly evaluate a state and propagate the resulting value back up the tree. This is cheaper than performing a playout, since a playout consumes significantly more resources than the heuristic function needs to evaluate a state.

3.5.4 Neural Networks

It would also be possible to use a neural network as a tree policy. Therefore it would need to be trained such as to evaluate the expected payoff when at a given state. AlphaGo, the first algorithm to reach a superhuman level of play in the game of Go, used two neural networks[3] as part of the default policy in order to achieve this unprecedented feat (Silver *et al.*, 2017b).

We unfortunately haven't found a way of efficiently representing the iteratively growing board with the different types of tiles. There are 72×72 possible board dimensions, which would therefore have to be the size of the input layer. Regarding the encoding, 19 different tiles which can each have one of 12 meeple placements and can have one of four rotations would have to be represented numerically, which would be infeasible and exceed the scope of this paper. Additionally, in regard to training the network, the team behind AlphaGo could utilise a database with millions of previously played games (Silver *et al.*, 2016). For us this would be impossible, since no such database exists for Carcassonne.

As an outlook, if an efficient representation of the board, as well as an efficient way of training a network on Carcassonne should be found in some later work, it is likely that using neural networks in a similar way would further improve our implementation.

3.6 Additional Tweaks

Apart from the tree policy and default policy, there are some further tweaks to MCTS which we will be evaluating in regard to their performance on Carcassonne.

3.6.1 Decaying Exploration Constant

Many tree policies encourage exploration in the early stages of training, when the game tree is still small, and increasingly discourage exploration as the state space is expanded and thus more information on the quality of states is available, e.g. Decaying ε-Greedy with $\varepsilon = 1/t$ (Section 3.3.2) and UCT-Tuned (Section 3.3.4). This effect can be simulated for any tree policy by using an exploration constant $c' = c/t$, where c represents a fixed constant and t denotes the current

[3] One network outputs the probability of winning given the current state, the other network outputs a probability distribution over the possible actions in regard to winning when choosing each action (Silver *et al.*, 2016).

training iteration. Thereby exploration is increasingly discouraged, such that after extensively exploring during the early stages of training, the later stages concern themselves with exploiting the generated knowledge.

3.6.2 Dynamic Backpropagation Weight

Intuitively, later playouts generate more accurate approximations of the value of a game state, since the state space is better known. Consequently, one could argue that the outcomes of those decisions should be weighted higher, since they are more representative of the strategy which preceded the decision. This was observed by Xie and Liu (2009), who claim to have achieved a performance increase on the domain of Go when weighting later decisions more highly using an exponentially growing weight.
We suggest the term $2^{\frac{t}{k}}$ as a weight to Δ during the Backpropagation step, whereby t denotes the current iteration and k is a constant. For higher values of k, the term grows slower and for lower values of k it grows faster.

4 Evaluation

4.1 Setup

We conducted all mentioned experiments at the sciCORE[4] scientific computing core facility at the University of Basel using the generic experimentation package Lab (Seipp *et al.*, 2017). We thereby used Intel Xeon E5-2660 CPUs running on a clock speed of 2.2 GHz. We didn't impose a time limit. Our implementation was written in Java 11.

In these experiments, if not stated otherwise, we evaluated the performance of a single parameter per experiment. We tested each parameter using the same MCTS benchmark player. The MCTS benchmark player for a given tree policy uses a random playout as a default policy with the corresponding meeple placement probability evaluated in Section 4.2.5, the corresponding exploration constant evaluated in Section 4.2.1, trains for 1000 iterations and has no further tweaks. The amount of training iterations was chosen due to the fact that it is the minimum number of iterations which allows for the maximum performance from the benchmark players (see Figure 5).

Each tested variant had the same preconditions, i.e., the same deck permutation and random seeds for all random decisions being made. Additionally, all the generated scores are averages over at least 10 games, whereby both implementations switched sides after half of the games in all test scenarios. Thereby each implementation gets tested both as player 1 and player 2 for each deck permutation.

Whenever we henceforth consider a Random player, it implies a player who picks a random legal move at every turn. Whenever we henceforth consider a Heuristic player, it implies a player who picks the legal move which maximises the heuristic function at every move. The Heuristic player is not to be confused with the Heuristic MCTS player, which uses MCTS with the heuristic function as a tree policy, as introduced in Section 3.4.

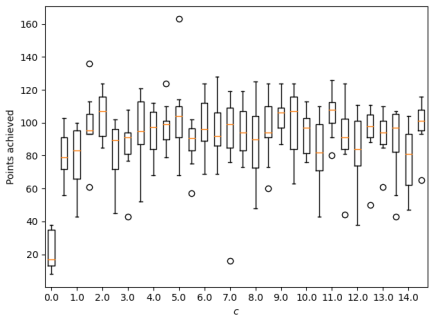
[4] `http://scicore.unibas.ch/`

We visualised the results of many experiments using box plots. Each entry in a box plot consists of a box with a horizontal line, as well as two vertical lines extending out of the box (i.e., the *whiskers*). The lower boundary of the box denotes the first quantile Q_1 (the median value of the lower half of the values), the upper boundary denotes the third quantile Q_3 (the median value of the upper half of the values), the horizontal line inside the box denotes the median value of all values. The whiskers extend towards the smallest and the largest value overall. The circles denote statistical outliers.

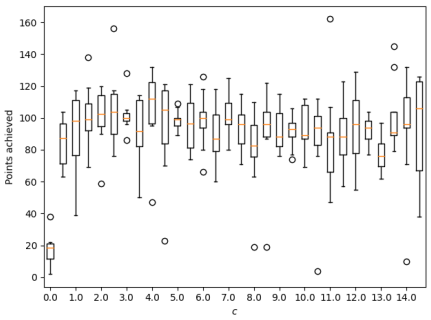
4.2 Experiments

4.2.1 Exploration Constant

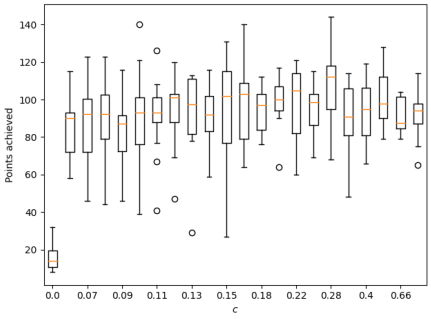
In order to evaluate a good exploration constant, we had benchmark variants using UCT, UCT-Tuned and ε-Greedy as tree policies with different exploration constants play against a Random player in order to evaluate which exploration constants allowed the tree policies to score the most points. The results have been visualised in Figure 4.



(a) UCT-Tuned.



(b) UCT.



(c) ε-Greedy.

Figure 4: Performance of tree policies with different exploration constants against a Random player.

These results suggest that the choice of exploration constant does not seem to make a significant difference in performance for the tested tree policies. Nonetheless, we can observe that $\varepsilon = 0.3$ for ε-Greedy, $c = 4$ for UCT and $c = 2$ for UCT-Tuned lead to the highest mean payoff by a slight margin. These

results demonstrate clearly that the performance drops very significantly as the constant takes on a value of 0. This corresponds to the algorithm seizing to explore, which leads to drastically worse results than if it explores.

The presented results correspond to the implementations playing against a single type of opponent. One must consider that optimal exploration terms can vary depending on the opponent and their strategy. Nonetheless, due to the fact that we observed such an insignificant performance difference for all constants larger than 0, we will henceforth be using the values which yielded the largest mean payoff for their respective experiment.

4.2.2 Tree Policies

For comparing the performance of the tree policies, we let a MCTS benchmark variant for each of them compete against a MCTS benchmark variant of every other tree policy. Each pair played 20 games. Table 1 shows the average score of each match together with the standard deviation (denoted by the number following the ± symbol). Table 2 shows the corresponding win percentages. For each pair, the higher average score and the higher win percentage are printed in **bold face**, in order to make the outcome more clearly visible.

Measured \ Opponent	UCT	UCT-Tuned	Boltzmann	ε-Greedy	Dec. ε-G.	Heur. MCTS	Heuristic	Random
UCT	–	100.2 ± 16.7	**113.7** ± 15.7	98.2 ± 14.8	**93.2** ± 18.9	**76.5** ± 19.5	**71.5** ± 11.1	**94.5** ± 17.6
UCT-Tuned	**109.0** ± 14.9	–	**110.8** ± 20.4	**103.5** ± 20.8	**94.8** ± 11.4	**74.3** ± 13.8	**79.9** ± 23.7	**91.3** ± 23.7
Boltzmann	83.0 ± 31.7	89.8 ± 20.8	–	75.0 ± 33.0	**84.1** ± 7.6	**70.3** ± 23.5	**68.7** ± 14.8	**76.4** ± 9.8
ε-Greedy	**101.2** ± 16.1	90.2 ± 31.8	**97.6** ± 17.9	–	**99.9** ± 20.2	**73.9** ± 13.3	**75.7** ± **21.3**	**103.0** ± 10.8
Decaying ε-Greedy	62.6 ± 21.9	62.0 ± 19.9	59.2 ± 23.4	56.8 ± 25.0	–	38.1 ± 17.8	**38.9** ± 12.0	**46.0** ± 9.8
Heuristic MCTS	53.6 ± 25.3	57.5 ± 21.2	56.5 ± 14.6	46.8 ± 20.5	**41.1** ± 13.5	–	39.8 ± 15.2	**44.1** ± 15.7
Heuristic	54.9 ± 24.9	51.9 ± 22.7	61.1 ± 33.7	48.9 ± 21.2	34.7 ± 12.7	**52.4** ± 23.5	–	**36.2** ± 30.3
Random	27.6 ± 14.1	23.5 ± 15.0	25.0 ± 14.9	24.6 ± 18.8	12.6 ± 11.2	11.1 ± 4.9	10.7 ± 5.0	–

Table 1: Average score and standard deviation in the matches between tree policies.

Measured \ Opponent	UCT	UCT-Tuned	Boltzmann	ε-Greedy	Dec. ε-G.	Heur. MCTS	Heuristic	Random
UCT	–	38.2%	**63.9%**	44.4%	**72.2%**	**83.3%**	**81.1%**	**100.0%**
UCT-Tuned	**61.8%**	–	**70.3%**	**51.4%**	**80.6%**	**86.1%**	**77.8%**	**100.0%**
Boltzmann	36.1%	29.7%	–	27.8%	**81.1%**	**63.9%**	**62.9%**	**100.0%**
ε-Greedy	**55.6%**	48.6%	**72.2%**	–	**86.5%**	**83.8%**	**78.4%**	**100.0%**
Decaying ε-Greedy	27.8%	19.4%	18.9%	13.5%	–	31.4%	**51.4%**	**89.2%**
Heuristic MCTS	16.7%	13.9%	36.1%	16.2%	**68.6%**	–	**59.5%**	**100.0%**
Heuristic	18.9%	22.2%	37.1%	21.6%	48.6%	40.5%	–	**100.0%**
Random	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	8.6%	–

Table 2: Win percentage of the matches between tree policies.

These results suggest that UCT-Tuned, Boltzmann and ε-Greedy are the most powerful tree policies, with UCT coming in close behind. With the exception of Decaying ε-Greedy beating UCT-Tuned, none of the three tree policies lost against any other player. Decaying ε-Greedy was noticeably outperformed by all other MCTS variants. We assume this is due to the exploration constant ε converging towards 0 at a fast rate, such that for the 1000 training iterations, at iterations 500 ε already takes on a value of $\varepsilon = 1/500 = 0.002$. Considering how unprofitable the results in Section 4.2.1 turned out to be when exploration constants took on a value of 0, this seems to correlate quite strongly and further demonstrates the importance of the exploration constant.

Heuristic MCTS was noticeably outperformed by all other MCTS variants, with the exception of Decaying ε-Greedy. This demonstrates how important a good tree policy is for MCTS to perform well, since, with the exception of the tree policy, all other aspects of the compared MCTS variants are identical. Other than the mentioned exception, the Random player and the Heuristic player were remarkably outperformed by all the MCTS variants, losing against all other opponents on average. Thereby the MCTS variants all needed, on average, around 10 seconds to compute each move, depending on how computationally expensive the corresponding tree policy function is. The Heuristic player, on the other hand, needed around 0.02 seconds, while the Random player required around 10^{-5} seconds.

Another interesting conclusion from these results is that worse players, i.e., the Heuristic MCTS-, Heuristic- and Random players, achieved better scores against better players. For instance, the Random player achieved an average of 24 points against the best four policies, yet only an average of 11 against the other players. This goes against the notion of Minimax, which operates under the assumption that a high quality strategy minimises the opponent's score, while maximising ones own. We assume this comes down to the fact that in Carcassonne a high quality strategy usually involves constructing a large amount of cities. Consequently, the opponent's field meeples will generate points for additionally built cities. As such, their score will increase if their opponent plays with a profitable strategy.

A strange observation is the surprisingly high standard deviation in the experiments. We assume that the high deviation appears due to the fact that small decisions can lead to large differences in the score. For example, placing a single meeple on a field can easily lead to an increase of 21 points if it borders on 7 cities. The results also suggest that the deviation increases proportionally to profitability, which makes sense considering the higher absolute numbers involved.

4.2.3 Training Iterations

We tested the relation between the amount of training iterations and performance by having multiple MCTS benchmark variants play against a Random player with different numbers of training iterations. The results have been plotted in Figure 5. The intuitive notion in regard to the amount of training iterations is that an increase in the number of training iterations leads to an increase in performance. Our results suggest that this holds up to a certain point.

Once again we observed that Decaying ε-Greedy performed worst by far. Interestingly, it does not follow the growth of the other three tree policies, which all behave very similarly in relation to the amount of training iterations. Its relative performance didn't grow as significantly with more training iterations than it was the case with the other tree policies. We will therefore only be considering UCT, UCT-Tuned, ε-Greedy and Boltzmann in this section.

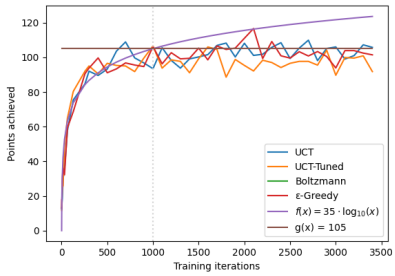


Figure 5: Mean performance of different tree policies against a Random player over the number of training iterations.

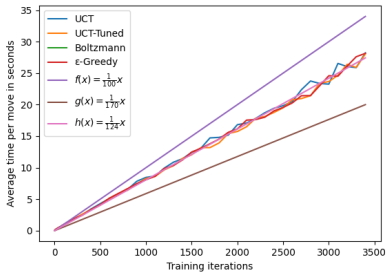


Figure 6: The runtimes of the different tree policies over the number of training iterations.

In Figure 5 we plotted the number of points each amount of training iterations achieved against a Random player on average. We observed that while the number of training iterations increases the performance, the performance gain decreases with a higher number of training iterations. For $n \leq 1000$ training iterations, the amount of points achieved very closely follows the function $f(n) = 35 \log_{10}(n)$. It follows that for $n \leq 1000$ the performance grows logarithmically over the number of training iterations. For $n > 1000$ training iterations, the payoff follows the function $g(n) = 105$. It follows that the performance of our benchmark implementation reaches its maximum performance when training for $n \geq 1000$ iterations. We plotted $f(n)$ and $g(n)$ next to the mean amounts of points scored for various tree policies in Figure 5.

In regard to Figure 5 it must be considered that these increases were evaluated playing against a Random player. This was done such that the algorithms have comparable opponents, yet a Random player is an atypical opponent to play against, since its performance is terrible. Nonetheless, we present these results under the assumption that the tendency in regard to the relation between the number of training iterations and performance remains the same when playing against more advanced players.

In addition, during these experiments, we observed that our implementation has a linear time complexity of $\Theta(n)$, as defined by Knuth (1976) in Definition 1.

Definition 1. *$\Theta(f(n))$ denotes the set of all $g(n)$ such that there exist positive constants C, C' and n_0 with $Cf(n) \leq g(n) \leq C'f(n)$ for all $n \geq n_0$.*

In Figure 6 we can see that the runtimes over the amount of training iterations closely correspond to the function $g(n) = \frac{1}{124}n$. For constants $C = \frac{1}{170}$, $C' = \frac{1}{100}$, $n_0 = 500$ and function $f(n) = n$ we can observe that $Cf(n) \leq g(n) \leq C'f(n)$ holds for all $n \geq n_0$ in Figure 6. We therefore conclude by Definition 1 that $g(n) \in \Theta(n)$ holds. Since $g(n)$ corresponds very closely to the runtime of our implementation over the number of training iterations, it follows that the runtime of our implementation is in $\Theta(n)$, for n training iterations.

4.2.4 Default Policies

As previously mentioned, the default policy defines how the leaf nodes are valued during the Simulation step. We mentioned in Section 3.5 that the default policy mostly consists of a playout (i.e., a simulation of the game until the end) whereby random moves are selected. We also mentioned that it can be improved by guiding the selection using a heuristic function, or using either a heuristic function or a neural network to directly value the leaf nodes, without performing a playout.

We tested the profitability of using a playout guided by the heuristic function by having two MCTS benchmark implementations, whereby one of them played using a heuristic playout as a default policy, play against each other. We found that for UCT and UCT-Tuned, using a heuristic playout lead to a noticeably improved performance. This is demonstrated in Figure 7.

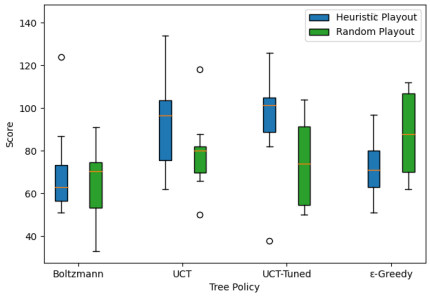


Figure 7: The results of benchmark implementations playing against their corresponding counterparts using a heuristic playout and 100 training iterations.

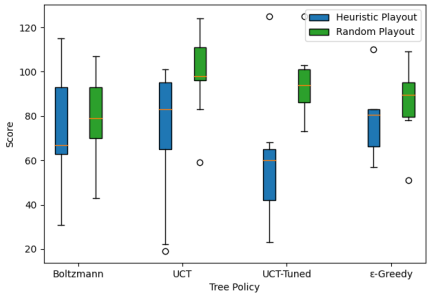


Figure 8: The results of a match between MCTS benchmark variants using 3000 training iterations against a corresponding implementation using a heuristic playout and 100 training iterations.

From Figure 7, we conclude that the performance difference invoked by using a heuristic playout is significant enough for there to exist a correlation between the heuristic playout and the performance increase.

It is important to thereby note that the runtime of the heuristic players was significantly higher, by a factor of 30. We were interested in testing the heuristic playout against an opponent with a similar runtime. We did this by having implementations with 100 training iterations and a heuristic default policy compete against implementations with 3000 iterations and a random default policy. This leads to comparable runtimes. The resulting average scores can be seen in Figure 8.

Our results suggest that an implementation using a random default policy with 3000 training iterations noticeably outperforms an implementation using a heuristic default policy using 100 iterations for all tree policies, whereby both implementations have a similar runtime. From the premises established in Section 4.2.3, we can deduce that an implementation using 1000 training iterations and a random playout as a default policy would perform very similarly as the first implementation, i.e., beating the implementation using the heuristic default policy, with a third of the runtime.

As such, we conclude that it is more beneficial to increase the number of training iterations with a random default policy instead of using a heuristic default policy, in order to maximise performance under consideration of a feasible runtime.

We achieved slightly worse results using a direct heuristic default policy (see Section 3.5.3). This approach reduces the runtime by a factor of 30 when compared to a random playout given the same amount of training iterations. But even when accounting for the difference in runtime by allowing the variant with the direct heuristic default policy thirty times as many training iterations, using a random playout proves to be more effective. This has been visualised in Figure 9.

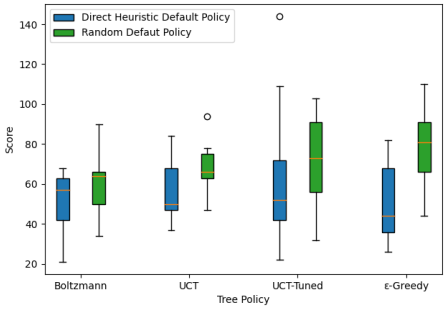


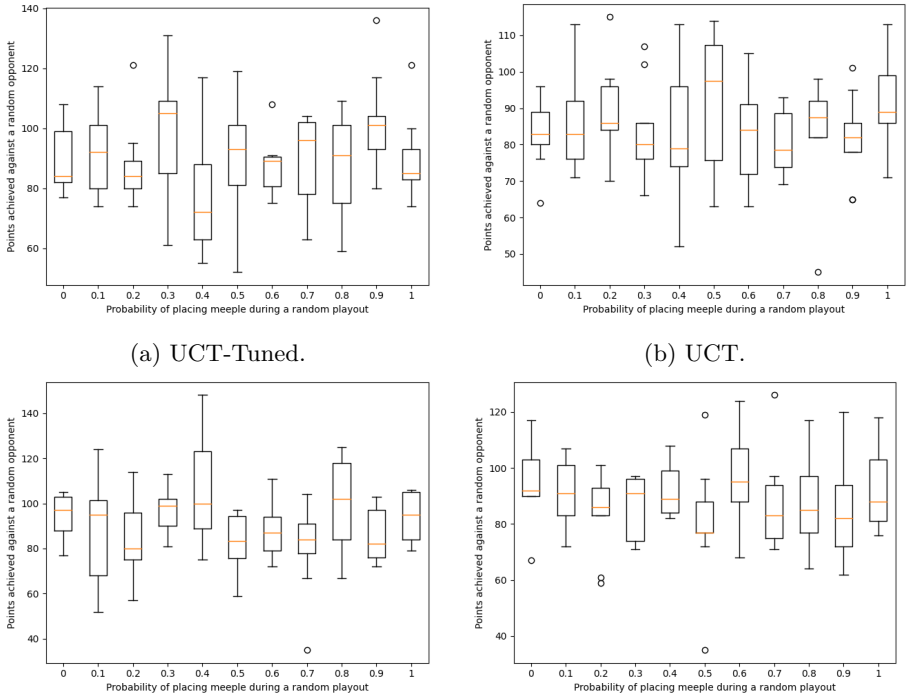
Figure 9: The results of a direct heuristic default policy using 3000 training iterations facing a random playout using 100 training iterations.

Of all the tested tree policies, Boltzmann performed best with a direct heuristic default policy. But even in that match it only won 4 out of 10 games with one draw. In summary, we conclude that using a random playout is the most beneficial default policy out of all the variants we tested in this section.

4.2.5 Meeple Placement Probability during Simulation

A random playout such as the default policy implies random moves being selected. Those random moves include placing a meeple. Yet placing a meeple during the first 7 turns and then having none left is not necessarily the only approach to playing the game randomly. As such, we tested different probabilities of placing meeples during a random playout.

Figure 10 shows visualisations of the performance with different probabilities against a Random player.



(a) UCT-Tuned.

(b) UCT.

(c) ε-Greedy.

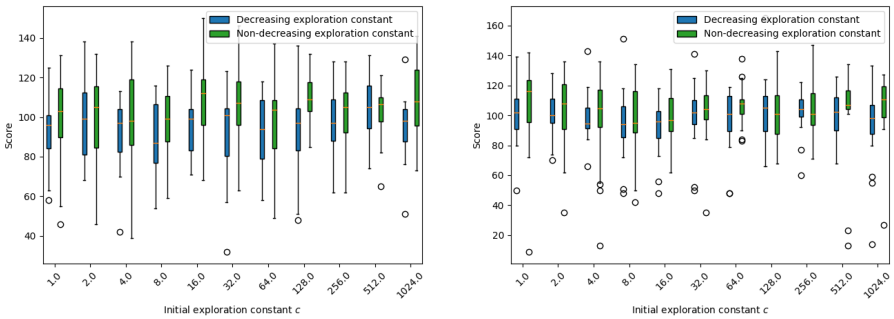
(d) Boltzmann.

Figure 10: Average amount of points achieved by benchmark implementations playing against a Random player using different meeple placement probabilities during the playout.

These results suggest that the probability of placing a meeple during a random playout slightly influences the performance. We do observe that a placement probability of around 0.6 for Boltzmann, 0.8 for ε-Greedy, 0.5 for UCT and 0.3 for UCT-Tuned seem to have performed best in our experiments.

4.2.6 Decaying Exploration Constant

For testing the profitability of iteratively decreasing the exploration constant (and thereby increasing exploitation) as the training progresses, we had MCTS benchmark variants play against each other, whereby one variant had a decreasing exploration constant as defined in Section 3.6.1 (i.e., $c' = c/t$ for a fixed constant c and training iteration t). We visualised the results in Figure 11. The different fixed constants c we tested are denoted on the x-axis of Figure 11.



(a) UCT-Tuned.

(b) UCT.

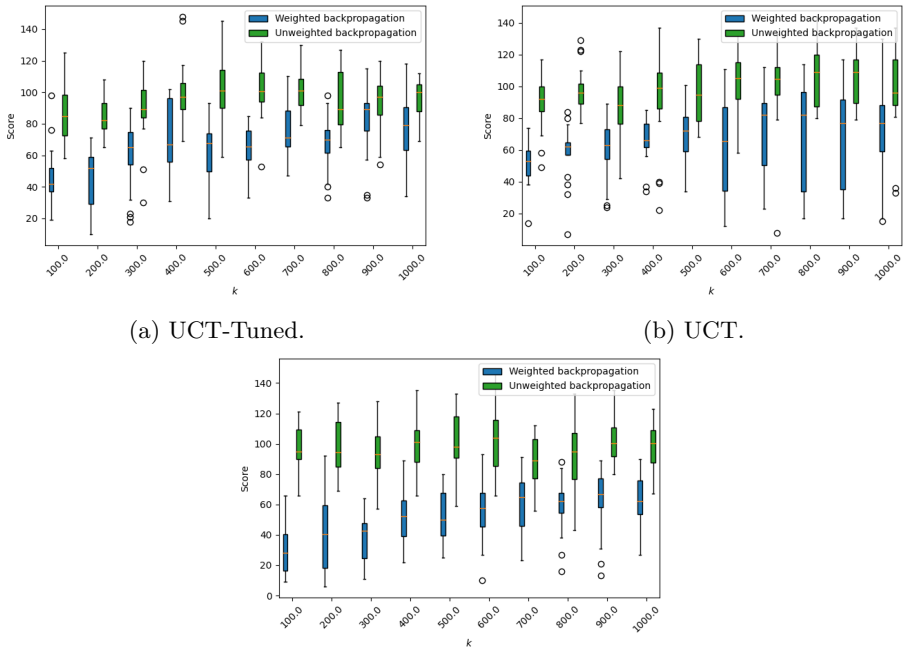
Figure 11: Average results of benchmark implementations with a decreasing exploration constant against a corresponding benchmark implementation.

For UCT, we conclude that iteratively decreasing the exploration constant leads to slightly worse results for all values c except for $c = 128$ and $c = 256$. For $c = 128$ UCT with a decaying exploration constant won 55% of the games, while for $c = 256$ it won 50% of the games. Additionally, our results suggest that for UCT, the standard deviation tends to be smaller when using a decreasing exploration constant.

For UCT-Tuned, we observed that while when considering the mean score a decreasing constant produced worse results, a decreasing constant with an initial value of $c = 512$ won 53% of the games.

4.2.7 Dynamic Backpropagation Weight

We evaluated the profitability of using a backpropagation weight of $2^{\frac{t}{k}}$ as defined in Section 3.6.2. The setup was similar to other results, i.e., two benchmark variants playing multiple games against each other, whereby one of the variants has a decreasing backpropagation weight for different values of k. Figure 12 shows the results of these experiments.



(a) UCT-Tuned.

(b) UCT.

(c) ε-Greedy.

Figure 12: Results of benchmark variants playing against corresponding variants using a weighted backpropagation.

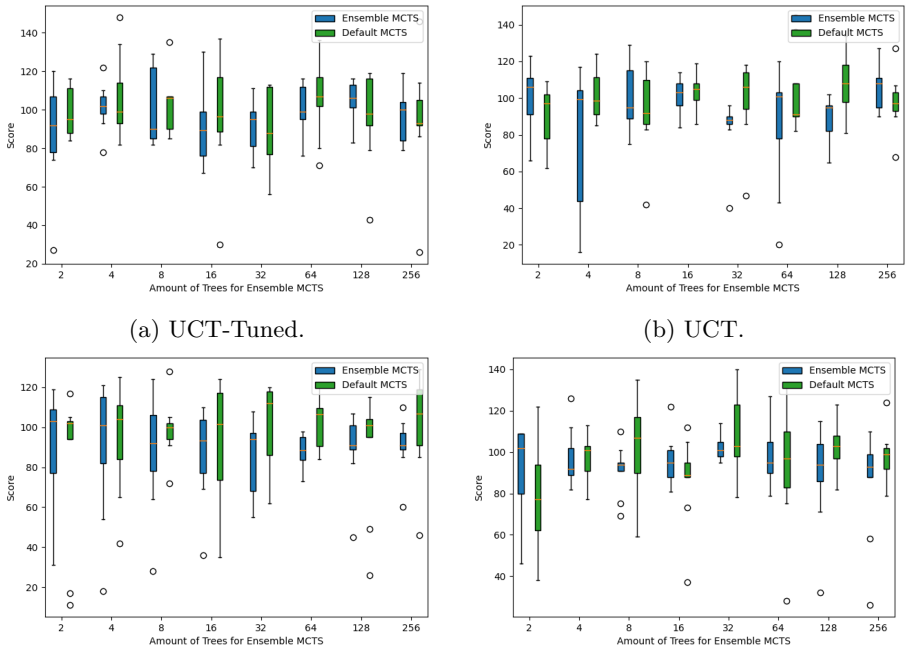
Our results suggest that it is not profitable to iteratively increase the backpropagation weight for all tested values of k. The results also suggest that a variant's performance grows roughly in proportion to the value of k. This further suggests that a growing backpropagation weight does not benefit performance, since $\lim_{k\to\infty} e^{\frac{t}{k}} = 1$ holds, whereby having no weight corresponds to having a constant weight of 1.

The results couldn't be tested for Boltzmann exploration, due to the fact that for a linear increase of $\bar{X}$, the term $e^{\frac{\bar{X}}{\tau}}$ increases exponentially. E.g., for $\bar{X} = 180$ and $\tau = 7$, the term takes on the value $e^{\frac{\bar{X}}{\tau}} = 147086357068$. If the increasing backpropagation-weight ends up increasing the Q-Value by a factor of only 2, we end up with $\bar{X} = 360$ and $e^{\frac{\bar{X}}{\tau}} \approx 2.16 \cdot 10^{22}$, which is several orders of magnitude larger. Considering that the maximum value a variable of the type Double can take on in Java[5] is approximately $9.22 \cdot 10^{18}$, we can see that it becomes a challenge to compute the probabilities of the Boltzmann distribution for larger Q-values than normal. It would be possible to adapt the value of τ, yet considering the similar results of an increasing backpropagation weight for the other tree policies, we do not expect a significantly different result to emerge for Boltzmann.

[5]According to the official documentation: `https://docs.oracle.com/javase/specs/jls/se7/html/jls-4.html#jls-4.2.3`

4.2.8 Ensemble MCTS

We tested the performance of Ensemble MCTS (Section 3.1.2) by letting MCTS benchmark variants play against each other, whereby one of the variants used Ensemble MCTS. The variant using Ensemble MCTS was then tested using different amounts of game trees. In the first experiment, the variant using Ensemble MCTS trained each of the k trees with $1000/k$ training iterations. With this approach, both variants had a similar runtime of around 10 seconds per move. These results have been visualised in Figure 13.



(a) UCT-Tuned.

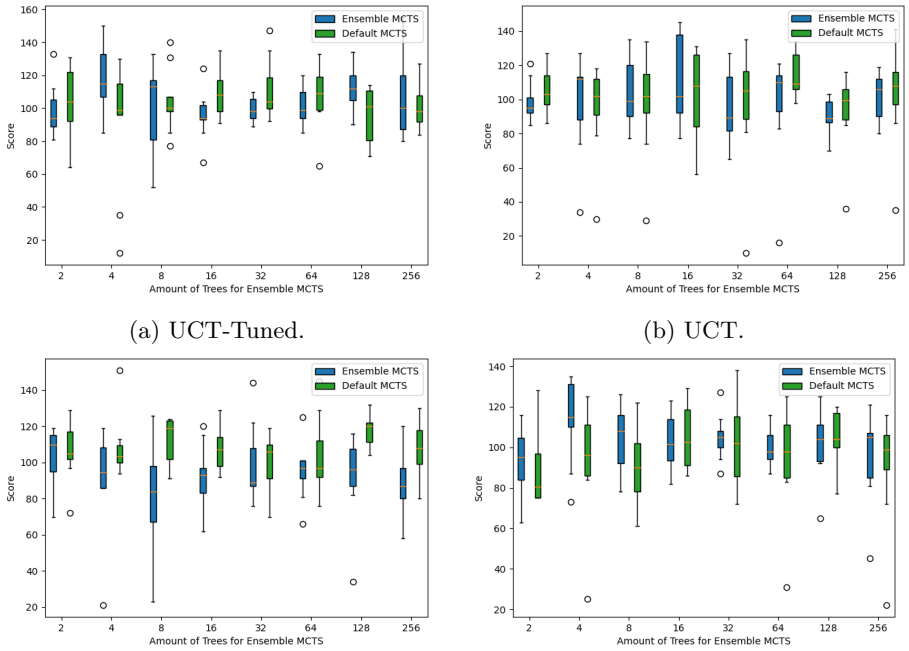
(b) UCT.

(c) ε-Greedy.

(d) Boltzmann.

Figure 13: Results of benchmark variants playing against their corresponding Ensemble MCTS variants. Ensemble MCTS played with $1000/k$ training iterations for each of its k trees, resulting in a similar runtime for both players.

The results in Figure 13 suggest that Ensemble MCTS can match the payoff of a MCTS variant using a single game tree. Since we previously observed that we reach a performance limit at 1000 training iteration for the strongest variants using a single game tree, we decided to test if variants using Ensemble MCTS with $3000/k$ training iteration for each of the k trees would outperform the respective MCTS benchmark implementation. This entails a higher runtime for the variant using Ensemble MCTS. The results of this experiment have been visualised in Figure 14.



(a) UCT-Tuned.

(b) UCT.

(c) ε-Greedy.

(d) Boltzmann.

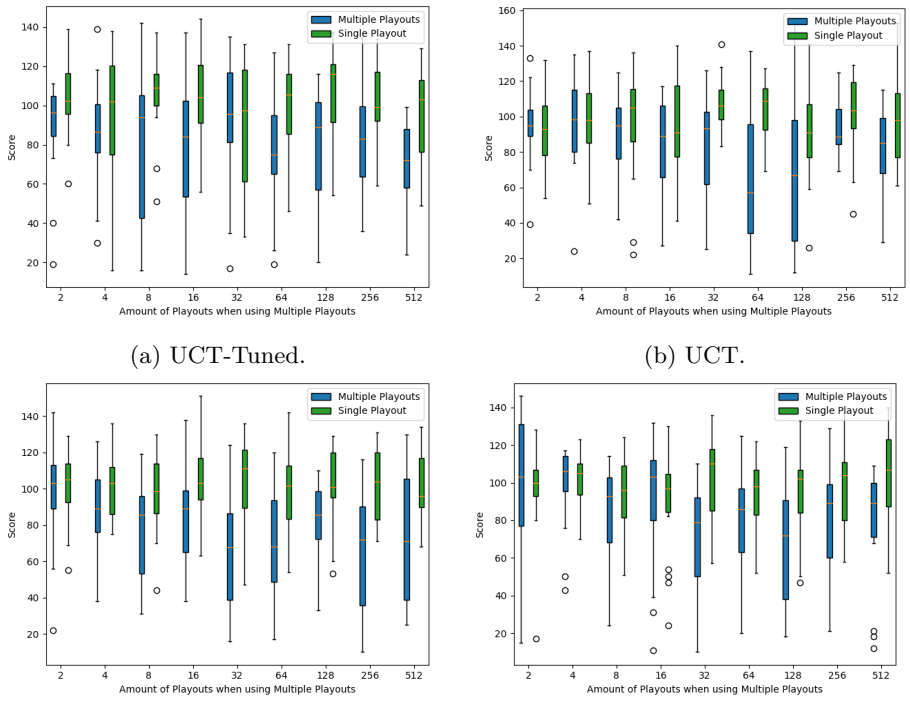
Figure 14: Results of benchmark variants playing against their corresponding Ensemble MCTS variants. Ensemble MCTS played with $3000/k$ training iterations for each of its k trees, resulting in a runtime three times higher for Ensemble MCTS.

The results in Figure 14 suggest that for UCT-Tuned, using Ensemble MCTS with $k = 4$ game trees, each training with 750 iterations, outperforms our MCTS benchmark implementation. It won 90% of the games it played against its corresponding MCTS benchmark variant. As we have noted, this comes at the cost of a higher runtime. The variant using Ensemble MCTS with $3000/k$ training iterations needs, on average, around 25 seconds to compute each move, compared to around 10 seconds for our MCTS benchmark implementations. Nonetheless, we consider 25 seconds to be within the limits of a feasible runtime, since a human can be expected to contemplate a move for 25 seconds.

4.2.9 Multiple Playouts

We tested the profitability of performing multiple random playouts per round of training by evaluating the results of two MCTS benchmark variants playing against each other, whereby one variant used multiple playouts per training iteration during the Simulation step. Both implementations played with the same number of training iterations (i.e., 1000, as stated in Section 4.1). For a small number of playouts the runtime difference is negligible. For example, a variant using 16 random playouts during the Simulation step only requires around 2 seconds more to compute each move on average compared to a corresponding MCTS benchmark variant. For larger numbers of playouts, the difference becomes more noticeable. For example, a variant using 512 playouts needed more

than a minute to compute each move, compared to the 10 seconds of its opponent. This must be considered for the results, which have been plotted in Figure 15.



(a) UCT-Tuned.

(b) UCT.

(c) ε-Greedy.

(d) Boltzmann.

Figure 15: Performance of different tree policies against benchmark implementations using different amounts of playouts for each training iteration.

The results in Figure 15 suggest that it is not profitable to increase the number of random playouts past a single playout per training iteration. Especially for a larger number of playouts the performance dropped significantly, while the increase standard deviation suggests that those variants performed less consistently. This may be the result of overfitting, since the explored parts of the tree get evaluated much more extensively.

An alternative approach, instead of simply adding more sampling, is to dynamically increase the number of playouts during the training process. This corresponds to the previously mentioned assumption that later decisions during the training process should be weighted higher, since more information regarding the state space is available. We tested this by having two corresponding benchmark variants play against each other, whereby one of them performed t playouts for training iteration t. The results are visualised in Figure 16.

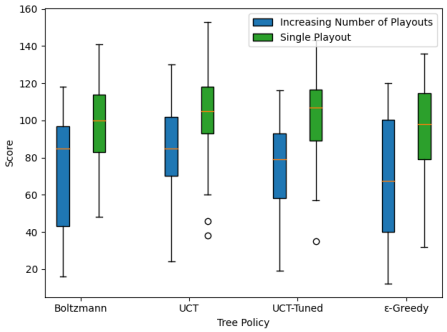


Figure 16: The results of a benchmark implementation executing t playouts for iteration t of the training process against a corresponding benchmark implementation.

Our results suggest that increasing the number of playouts linearly during the training process is not more profitable than consistently executing a single random playout.

4.2.10 Bonus: What happens if MCTS knows the deck configuration?

In our final evaluation, we tested how MCTS would perform if it had perfect knowledge of the game state, i.e., would know the deck configuration and therefore cheat. We used an implementation with only one placement node per chance node, whereby the placement node corresponds to the tile which MCTS now knows it will draw when reaching the given point in the game. The results of these experiments have been visualised in Figure 17.

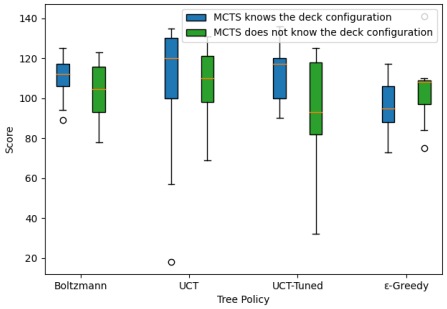


Figure 17: The results of matches between two MCTS benchmark variants, whereby one implementation knows the state of the deck.

Our results suggest that MCTS benchmark variants using Boltzmann, UCT and UCT-Tuned profit from this added knowledge slightly, while the MCTS benchmark variant using ε-Greedy seems to have not profited at all. Although this does not correspond to intuition, we suggest a possible explanation. Considering the large branching factor of the game tree, our experiments, run with 1000

training iterations each, only generate a tree which considers around 3 tiles to be drawn from the deck. As such, MCTS cannot fully profit from the increased knowledge, except in the Simulation step, where it does not seem to make a big difference.

4.3 Qualitative Evaluation of how MCTS plays Carcassonne

We played six games, switching sides after three games, against the most profitable MCTS variant we found in order to evaluate if it performs better than an average human player. We used an implementation with UCT-Tuned as a tree policy and a random playout as a default policy with a decaying exploration constant of $c = 512$ and a meeple placement probability of 30%. It uses Ensemble MCTS with 4 game trees, each training on 750 training iterations.

The algorithm often played such as to force us to play moves which were to its own advantage. Examples include placing a meeple on a road which we would necessarily need to complete if we wanted finish our monastery, or building its monasteries next to our monasteries, such that it would necessarily benefit if we were to complete our monastery.

It also tended to steal cities, i.e., build cities next to ours, such that we had to attach our city to its newly built city in order to finish it. It also often placed tiles such that it manoeuvred us into awkward positions which only a single type of tile could solve.

It was also noticeable that after a few rounds, it would never have more than one meeple remaining. It would try to place them as broadly as possible, such that, at times, every city was occupied by one of its meeples.

In general, it felt as if the algorithm managed to anticipate our moves frequently, which suggests a capability of not only considering its opponent's short term moves, but also their long term strategy. We often felt as though all our decisions could only lead to greedily harvesting points from small roads and cities, while the algorithm managed to harvest a huge amount of points from field meeples. Indeed, this observation underpins the conclusion of Ameneyro *et al.* (2020) that MCTS is capable of developing a profitable long term strategy for Carcassonne.

The algorithm won 83% of the games it played against us. The average score was 95.3 ± 18.1 to 88.3 ± 22.1 in favour of the algorithm. Under the assumption that we are average human players, we can conclude that a good implementation of the algorithm is capable of consistently beating an average human player at Carcassonne.

5 Conclusion

We have shown that that certain implementations of MCTS achieve good results on Carcassonne, whereby the quality of the results and the runtime to achieve them are highly dependant on how MCTS is implemented. Profitable variants using a good tree policy (such as UCT-Tuned) consistently beat both other benchmark players and human players on average.

We found that implementations using UCT-Tuned with $c \neq 0$ as a tree policy produced the best results on average, while implementations using UCT with

$c \neq 0$, Boltzmann exploration and ε-Greedy with $\varepsilon = 0.3$ also performed well. A surprising result was that for UCT and UCT-Tuned with a non-decaying exploration constant the value of the constant didn't influence the runtime significantly, with the exception of $c = 0$, which caused a significant drop in performance as MCTS stopped exploring during the Selection step.

While a heuristic playout as a default policy was shown to produce better results compared to a random default policy for the same amount of training iterations, the increased runtime of the heuristic playout can be compensated with a random playout using more training iterations, which then produces better results. Additionally, the heuristic playout increases runtime by a factor of 30 compared to using a random playout, which renders the runtime infeasible when increasing the amount of training iterations to the degree where performance is maximised.

Using the heuristic function directly showed promising results and decreased the runtime by a factor of 30 compared to using a random playout. Despite this, we managed to achieve better results using a random playout instead of a direct heuristic default policy. Using a backpropagation weight proved to be highly unprofitable, while a decreasing exploration constant over the course of the training process proved to slightly increase performance for certain constants, such as $c = 512$ for UCT-Tuned. Performing multiple playouts didn't improve the performance.

Ensemble MCTS produced similar results to using a single game tree given a similar runtime, yet it emerged as a method of increasing the performance past the performance cap of 1000 training iterations per tree. As such, multiple well-trained trees "vote" on the best move, which produced better results than when using a single game tree.

These results have confirmed the pre-established notion that MCTS is capable of handling large state spaces and traversing the corresponding game trees in order to find good solutions. We managed to implement MCTS such that it consistently outperforms an average human player. Thereby the strongest implementation within the time limitation of playing against a human has shown to use Ensemble MCTS with four trees, each using 750 training iterations with UCT-Tuned as a tree policy, whereby each training iteration t uses the exploration constant $c' = 512/t$. The default policy is thereby a random playout with a meeple placement probability of 30%.

Overall, the most interesting part of these results was what turned out to be the best variant of MCTS; rather unexpectedly, the best variant ended up being an uninformed variant, i.e., a variant which does not utilise domain specific knowledge. Simply through reinforcement learning it can acquire the skills to beat the people who implemented it.

With this conclusion we suggest that improvements of our implementation, e.g., by incorporating neural networks into the Simulation step, would be an interesting area of future research. Other improvements could concern themselves with more accurately modelling the randomness as part of the game tree, e.g., by adding a placement node for each individual tile in the deck, or with the performance of a bounded heuristic playout. Developing a more advanced heuristic function may also increase the performance of the direct heuristic default policy.

Bibliography

Fred Valdez Ameneyro, Edgar Galván, and Ángel Fernando Kuri Morales. Playing carcassonne with monte carlo tree search. In *2020 IEEE Symposium Series on Computational Intelligence (SSCI)*, pages 2343–2350, 2020.

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

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

Bruno Bouzy and Tristan Cazenave. Computer go: an AI oriented survey. *Artificial Intelligence*, 132(1):39–103, 2001.

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

Guillaume Chaslot, Sander Bakkes, Istvan Szita, and Pieter Spronck. Monte-carlo tree search: A new framework for game AI. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, volume 4, pages 216–217, 2008.

Sylvain Gelly and David Silver. Monte-carlo tree search and rapid action value estimation in computer go. *Artificial Intelligence*, 175(11):1856–1875, 2011.

Cathleen Heyden. Implementing a computer player for carcassonne. Master's thesis, Maastricht University, Department of Knowledge Engineering, 2009.

Leslie Pack Kaelbling, Michael L Littman, and Andrew W Moore. Reinforcement learning: A survey. *Journal of artificial intelligence research*, 4:237–285, 1996.

Donald E Knuth. Big omicron and big omega and big theta. *ACM Sigact News*, 8(2):18–24, 1976.

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

David Lenz, Tobias Kessler, and Alois Knoll. Tactical cooperative planning for autonomous highway driving using monte-carlo tree search. In *2016 IEEE Intelligent Vehicles Symposium (IV)*, pages 447–453. IEEE, 2016.

Nicholas Metropolis and Stanislaw Ulam. The monte carlo method. *Journal of the American statistical association*, 44(247):335–341, 1949.

S Ali Mirsoleimani, Aske Plaat, and Jaap van den Herik. Ensemble UCT needs high exploitation. *arXiv preprint arXiv:1509.08434*, 2015.

Martin J Osborne and Ariel Rubinstein. *A course in game theory*, chapter 6. MIT press, 1994.

Regis Riveret, Cameron Browne, Didac Busquets, and Jeremy Pitt. A monte-carlo tree search in argumentation. In *Proceedings of the Eleventh International Workshop on Argumentation in Multi-Agent Systems (ArgMAS 2014)*, pages 1–10. Workshop on Argumentation in Multi-Agent Systems, 2014.

Jendrik Seipp, Florian Pommerening, Silvan Sievers, and Malte Helmert. Downward Lab. `https://doi.org/10.5281/zenodo.790461`, 2017.

Silvan Sievers and Malte Helmert. A Doppelkopf player based on UCT. In *Joint German/Austrian Conference on Artificial Intelligence (Künstliche Intelligenz)*, pages 151–165. Springer, 2015.

David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.

David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *arXiv preprint arXiv:1712.01815*, 2017.

David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. Mastering the game of go without human knowledge. *Nature*, 550(7676):354–359, 2017.

István Szita, Guillaume Chaslot, and Pieter Spronck. Monte-carlo tree search in settlers of catan. In *Advances in Computer Games*, pages 21–32. Springer, 2009.

Mark HM Winands and Yngvi Björnsson. Evaluation function based monte-carlo loa. In *Advances in Computer Games*, pages 33–44. Springer, 2009.

Mark HM Winands, Yngvi Bjornsson, and Jahn-Takeshi Saito. Monte carlo tree search in lines of action. *IEEE Transactions on Computational Intelligence and AI in Games*, 2(4):239–250, 2010.

Klaus-Jürgen Wrede. *Carcassonne*. Hans im Glück, Munich, 2005. Board Game.

Fan Xie and Zhiqing Liu. Backpropagation modification in monte-carlo game tree search. In *2009 Third International Symposium on Intelligent Information Technology Application*, volume 2, pages 125–128. IEEE, 2009.

University of Basel

Faculty of Science

Declaration on Scientific Integrity

(including a Declaration on Plagiarism and Fraud)
Translation from German original

Title of Thesis: Monte Carlo Tree Search for Carcassonne

Name Assessor: Prof. Dr. Malte Helmert

Name Student: Max Jappert

Matriculation No.: 18-052-647

With my signature I declare that this submission is my own work and that I have fully acknowledged the assistance received in completing this work and that it contains no material that has not been formally acknowledged. I have mentioned all source materials used and have cited these in accordance with recognised scientific rules.

Place, Date: Basel, 12.06.2022 Student: 

Will this work be published?

- ☐ No
- ☐ Yes. With my signature I confirm that I agree to a publication of the work (print/digital) in the library, on the research database of the University of Basel and/or on the document server of the department. Likewise, I agree to the bibliographic reference in the catalog SLSP (Swiss Library Service Platform). (cross out as applicable)

Publication as of: _____

Place, Date: _____ Student: _____

Place, Date: _____ Assessor: _____

Please enclose a completed and signed copy of this declaration in your Bachelor's or Master's thesis .