# Tichu explored with Monte Carlo Simulation Methods

Pestalozzi Lukas

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Semester Project, 4th Master

Supervised by

**Igor Kulev and Boy Faltings**

Summer 2017

## Abstract

Tichu is a multiplayer card game with hidden information and both cooperative and competitive elements. Those properties makes it an interesting game from the point of view of Artificial Intelligence. The goal of the project was to create an AI-agent able to play Tichu without adding any special knowledge about the game. To this end the project mainly explores and compares various Monte Carlo Tree Search (MCTS) methods. It shows that MCTS handles the difficulties posed by Tichu reasonable well but there is a lot of potential for improvements.

# Contents

# List of Figures

# List of Tables

# Introduction

I found computer programs that are able to play games always fascinating. So I took the opportunity offered by the semester project to create one myself.
Instead of concentrating on pure playing strength of the agent, I wanted to explore methods that don't use in depth knowledge about the game and its tactics.
As for the game, Tichu is a popular card game in the german part of Switzerland and I thought it poses a good challenge. During the research for algorithms able to tackle the challenge, I encountered Monte Carlo Tree Search methods and decided to concentrate mainly on those.

In the following report, first the rules of the game are presented followed by an overview of important features of Tichu with regard to AI. The next chapter gives a short insight in the implementation of the game-framework and some difficulties encountered during that process. The main chapter follows. It contains an introduction to Monte Carlo Tree Search as well as methods to deal with the various difficulties posed by Tichu. Then, the agents implemented over the course of the project are described in detail. In the last part, the agents are evaluated against each other and the results are discussed.

# The game Tichu

Figure 2.1: Pictures of the game and cards

Tichu is a 4-player gambling card game which is marketed by Fata-Morgana [1]. It falls into the category of ladder games, where players must play a "higher" ranked set of cards on their turn or pass. The four players build 2 teams which compete against each other for points. Tichu contains elements from some Chinese card-games such as *Dou Di Zhu* or *Zheng Fen* and is very popular in the german part of Switzerland.

## 2.1 Rules

More detailed rules can be found in [1, 2].
The game is played in successive rounds until one team reaches a predefined amount of points.

**Cards**
There are 4 suits (Sword, Pagoda, Star, Jade) of 13 cards each corresponding in rank to the standard Bridge cards (2, 3, ... ,10, J, Q, K, A). In addition there are 4 special cards without any suit: *Mah-Jong, Dog, Phoenix* and *Dragon*. Hence there are 56 cards in total.

**Preparation**
The winner of the previous round shuffles and cuts the deck. When all cards are distributed, each player gives one card to each of the 3 other players. This phase is called card trading.

**The Game**
A round is started by the player who has the *Mah-Jong* card. He may lay down any one of the following combinations:

**Single** A single card

**Pair** Two cards of the same rank

**Triple** Three cards of the same rank

**Square** Four cards of the same rank

**Fullhouse** A triple and pair together

**Straight** At least five cards of consecutive ranks

**Pair Step** Two or more consecutive Pairs (ie, J, J, Q, Q, A, A)

The next player (on the right) has two options, either to play a combination of the same type but with a higher rank, or to pass (not play at all). That means, for example, a pair can only be beaten by another pair of strictly higher rank and a straight only by another, higher straight of the same length.
The play then proceeds to the next player. If all players pass consecutively the trick ends and the player who laid the last combination takes the trick and leads a new one. If he has no more hand-cards, he retires from the game and the next not-retired player may start the trick.

**Bombs**
Bombs are either a straight where all cards have the same suit (”straight flush”) or four cards of the same rank (”square”). A bomb beats all other combinations and a higher bomb beats a lower one. Any straight bomb is higher than a square bomb and a longer straight bomb beats a shorter one. Bombs can be played at any time during a trick but only after the first play was made.

**The Special Cards**
*Mah-Jong:* Whoever has this cards makes the first lead. The *Mah-Jong* has a rank of 1 and can be used in a straight (eg. 1,2,3,4,5,6). When a player plays the *Mah-Jong*, he can wish any specific card rank (not including special cards). The next player who has a card with the wished rank and can play it *in accordance with the rules of the game*, must play it. This condition remains in force until somebody plays such a card.

*Dog:* The dog has no rank and can only be played as single card and only as *first-play* (first card of a new trick). It immediately gives the lead to the partner of the player who played the dog (it can’t be beaten by a bomb). If the partner already finished (has no handcards left) then the lead passes to the player on the right.

*Dragon:* The dragon is the highest single card and can also only be played as a single card. If the Dragon wins the trick, the player must give the entire trick to any player of the opposing team.

*Phoenix:* The phoenix can be used as a joker in any combination but it can’t be used to create a bomb. It also can’t replace any other special card. When played as a single card it has a value half a point above the last played card but it can’t beat the dragon. If it is played first it has a value of 1.5.

**Calling Tichu**
Before playing his first card, each player has the right to announce a ”Tichu”. If he then wins the round (finishes first), his team receives 100 additional points - otherwise the team loses 100 points. A player can also announce a ”grand Tichu” before getting the $9^{th}$ card from the dealer. This gives 200 additional points (in case of success) or a penalty of -200 otherwise.

**Goal of the game**
The round is played until three of the four players have played all of their cards. The remaining player gives his remaining handcards to the opposing team and all the tricks he won to the player who finished first. Then the cards of each team are counted as follows:

- Kings and 10's are worth 10 points each
- 5's are 5 points each
- The Dragon is worth 25 points
- The Phoenix costs 25 points (is worth -25)

However, if both players of the same team have a doublewin (both finished before any of the opposing team), then that team gets 200 points and the card points are not counted.

When one team reaches a predefined number of points (usually 1000) they are the winners of the game.

## 2.2 Tactics

Beginners are often told to concentrate on finishing first and not to pay attention on at the points of the cards during the game. This is good advice and even advanced players don't pay too much attention to the actual point-values of the cards when considering to play a combination.
Getting rid of cards in ascending order is already a good strategy as having low card at the end makes it quite hard to finish. That said, it is seldom advantageous to split combinations apart.

Team-play is key. Points won by announcing Tichu and from doublewins often dominate points won with tricks during the game. It is therefore extremely important to support your partner when she announced a Tichu and an enemy Tichu or doublewin should be prevented at all cost.

This is not a comprehensive list of tactics by any mean, but it helps to get a better idea of the game.

## 2.3 Tichu and Artificial Intelligence

Tichu is quite different from "classical" games such as chess or Connect Four.

First of all, it is a game with hidden information since the handcards of the other players are unknown (with the exception of the traded cards at the beginning). This introduces several problems discussed in the section *Hidden Information* below.
The second big difference to the classical games is that the game is both, a cooperative and competitive at the same time. In a team the players must play cooperatively and help each other while competing with the other team for points.
Tichu is a deterministic game in the sense that the effect of each action is known with certainty. However, it is not possible to determine precisely what the possible actions of any of the other players may be since their handcards are not known. This makes is necessary to model Tichu as a stochastic game from the viewpoint of a player.
The game is played over several, in essence independent, rounds which makes it an episodic game. Each round consists of several tricks being played, so each round has also an episodic nature. However, unlike the rounds, the tricks of the same round are not independent. The actions in a trick depend on actions played in previous ones.
Tichu is a discrete, turn based game. The fact that bombs can be played at any time can be modeled with a 'fast round': After each played combination all players have to decide whether they want to play a bomb or not. Then, the next player can play a 'normal' combination.

Finally, Tichu is a zero-sum game even though the points of a round typically don't sum up to zero. At the end exactly one team wins, the other looses. It is possible to center the points after each round around zero to make them sum up to zero without impacting the gameplay at all.

### 2.3.1 Difficulties

Tichu is in several aspects a "difficult" game for AI. In this section some of the difficulties are presented.

**Hidden Information** The problem with games with hidden information is that a player does not have all information necessary to find the optimal action. This leads to uncertainty about "how good" each action actually is since it depends, at least partially, on the hidden information (or else the game is probably not fun to play). In most such games it is a big advantage to learn or at least approximate the hidden information, Tichu is no exception.
A player knowing exactly what the other players handcards are can determine the best action with 100% confidence. However, due to the random dealing of the cards, knowing everything doesn't guarantee a win. It may happen that the enemy just has better cards. This implies that perfect information might not be as important in Tichu as in other games.
There are roughly two different approaches to deal with hidden information. Infer the information, for example from the way a player plays, or deal with it by finding out what actions are good in a lot of possible cases. This project explores both approaches to a certain degree.

**Big branching factor** Even without the hidden information Tichu would be challenging, not least because of the large branching factor of the gametree[1].

The game-tree of a typical Tichu game has two different node-types with regard to the branching factor. The *first-play* actions (a player can play any combination-type) have a branching factor that is typically between 10 and 30, but can reach up to 450 in rare cases. Especially in the beginning of the game, when all players still have most cards, the branching factors are relatively big. The nodes corresponding to the remaining plays in the trick have a small branching factor (seldom more than 3) since the player has to play the same combination-type as played in the *first-play.*
Most games go through 8-15 tricks, each consists of roughly 10-20 actions. Thus, a game contains roughly 100 actions in total. (The longest possible game lasts for 220 actions[2]) Therefore, a typical game-tree is one that branches out rather fast near the root (at the beginning of the game), is around 100 deep and contains 8-15 *first-plays.* Leading to a tree with roughly $10^{20}$ nodes. From those numbers it is clear that an exhaustive search for even one particular deal is practically unfeasible.

Unlike in chess, where the starting position is always the same, we start with a different deal in each Tichu game. This excludes the possibility of creating the Tichu equivalent of an opening-moves database. Therefore, each player has to compute the best action online.

**Multiagent, cooperative and competitive** The simultaneous cooperative and competitive nature of the game gives raise to other interesting challenges. However they are not addressed in this project.

[1]for gametree see 4.1

[2]The leading player always plays a single card and the other players always pass such that the loosing player has 1 card left at the end, $4 * 14 * 3 + 4 * 13 = 220$.

# Implementation

As I could not find any open source implementation of the game Tichu that could be used out of the box without major modification[1], and the owners of *Brettspielwelt.de*[2] did not respond to my inquiries, I decided to write my own framework. As programming language I choose Python (v3.6) because it allows fast developing and has dynamic typing which I find to be very convenient. Despite that, the implementation of the framework and the different agents took up most of the time for this project.

It follows a brief overview of the architecture of the framework and some difficulties I encountered during the implementation.

## 3.1 Architecture

The framework is implemented to be compatible with the environments of *OpenAI-gym*[3]. Besides providing a standardized interface, it allows the use of third party libraries which are written to solve the *OpenAI-gym* challenges, such as *Keras-rl* [17] (a reinforcement learning library).

The heart of the implementation is the representation of the gamestate. It took some time to come up with a compact and fast way to represent a state. I ended up making the class TichuState immutable[4] and created the functions *possible_actions()* and *next_state(Action)*.
Calling *possible_actions()* on a TichuState instance returns all legal actions for the state and calling *next_state(Action)* returns the resulting state when applying the given action to the state.

The Agent interface is minimal and consists of the single function *action(State)*, that, given a gamestate returns the action the agent wants to play. Note that the *State* is the perfect-information state of the game. This allows the agent to "cheat" (ie. look at the handcards of the other players).

An advantage of implementing my own framework is that the agents can utilize the same code as the game-implementation itself and no conversion between different state representations have to be made.

## 3.2 Difficulties

The first challenge was to find a good architecture for the whole framework. The in the previous section presented architecture was preceded by two other approaches which were not nearly as compact and convenient to use. Another "software architecture" challenge was, how to implement the many different agents (described in the next chapter) without duplicating code everywhere. Luckily, Python has a very convenient multi-class-inheritance scheme, which allows a kind of "code injection" by inheriting from multiple parent classes[5].

[1] There are plenty halfhearted implementation on github.

[2] A platform providing the possibility to play games online, including Tichu.

[3] A platform providing environments to train AI-agents in predefined environments ranging from the "mountain-car" problem to the classical Atari games. (https://gym.openai.com/)

[4] immutable meaning that the values of attributes of an instance never change.

[5] Watch Raymond Hettinger's Pycon2015 talk "Super Considered Super" for more details: https://www.youtube.com/watch?v=EiOglTERPEo

Many challenges revolved around performance. Python is not designed to be fast, which became a problem when doing Monte Carlo Simulations. Most notably, finding all playable *different* combinations in a set of cards had to be optimized considerably. An important observation was that for similar combinations it does not matter which one is played and therefore only one of them has to be considered when searching for the best action to play. As an example, a set of cards containing the three kings $K\heartsuit, K\diamondsuit, K\clubsuit$, also contains three different *pairs* of Kings ($K\heartsuit, K\diamondsuit$ and $K\heartsuit, K\clubsuit$ and $K\diamondsuit, K\clubsuit$). However, it does not matter at all which two kings are played since the suit has no relevance in Tichu. In this case only one out of the three possible pairs has to be considered as a possible action. There is of course one exception to this observation: when the set also contains a *straight-flush* containing one of the Kings. Then it does matter which King (suit) is played in order not to 'destroy' the *straight-flush*. But this case is easy enough to detect and seldom enough not to impact performance. So the algorithm just returns all different combinations in this case.

# Agents

In this chapter the different agents are introduced and discussed.

## Simplifications

To limit the scope of the project and to be able to concentrate on the 'card play' aspect of Tichu, I decided to simplify the agents as follows:

**No Trading** The agents all trade random cards. This has the same effect as omitting the trading phase.

**No Tichu announcement** As mentioned in section 2.2, points won with Tichus often dominate points won by wining tricks. This diverts from the 'card play' aspect. Therefore all Agents are implemented such that they never announce any Tichu.

**Bombs not anytime** For simplicity and ease of simulation, a player can only play when it's his turn. That means bombs can not be played out of turn. This does not have a big impact on the game play overall, since it is rare to gain an advantage in playing a bomb out of turn.

**Random Wish** There are different strategies to determine which cards to wish. To not introduce domain knowledge, all agents wish for a random card.

## 4.1 Background

This section contains a short overview of the main algorithms used by the different agents.

An important notion is the **gametree**, which is a tree containing as nodes all possible gamestates and as edges the (game)actions leading from a parent state to a child state. The root is the initial state of the game, and the leafs are states in which the game ended (one player or team has won).

### 4.1.1 Minimax Search

Given a gametree, the optimal strategy can be determined from the minimax value of each node. This value denotes how 'good' a gamestate is for the searching player and is computed as follows:

$$minimaxval(s) = \begin{cases} Reward(s), & \text{if s is terminal} \\ \max_{a \in Actions(s)} minimaxval(next(s,a)), & \text{if player(s) = searching player} \\ \min_{a \in Actions(s)} minimaxval(next(s,a)), & \text{otherwise} \end{cases}$$

where $s$ is a gamestate, $Reward(s)$ denotes the reward for the searching player in $s$, $Actions(s)$ is the set of legal actions in $s$, $player(s)$ denotes the player able to play in $s$ and $next(s,a)$ is the resulting state when playing action $a$ in $s$.
This algorithm performs a complete depth-first search of the gametree, which is computationally unfeasible in most games, but it is a good theoretical baseline for other algorithms.

Despite this, it is possible to use minimax in larger games by stopping at non terminal states and evaluating them with an *utility* function. Minimax has been successfully used in games where it is possible to create a good *utility* function, most prominently in chess.

Furthermore, it is possible to prune away large parts of the searchtree without loosing accuracy using **alpha-beta pruning**. This method makes use of the fact that some subtrees don't influence the final result since the players never play suboptimal actions. For more details see [9, chapter 5, p. 170+] or any textbook about artificial intelligence.

### 4.1.2 Monte Carlo Tree Search (MCTS)

Similar to minimax search, MCTS also searches the gametree, but uses simulated games to evaluate nonterminal states and thus does not need an utility function. This is especially useful for games where it is hard to determine whether a state is 'good' or 'bad' for a player, for example in games with hidden information, but also in perfect information games such as GO.

The algorithm iteratively builds a subtree of the gametree where the root corresponds to the current state of the game. Each iteration consists of the 4 phases shown in Fig. 4.1:
**Tree Selection** The algorithm traverses the (in previous iterations built up) gametree until it reaches a leaf-node. The decision which actions to follow during the traversal is determined by the *tree policy*. The tree policy attempts to balance considerations of exploration (look in areas that have not been well sampled yet) and exploitation (look in areas which appear to be promising). An often used tree policy is the UCT (Upper Confidence bound for Trees) formula proposed by [5] (see section 4.4.1).
**Expansion** Once arrived at a leaf state, one more action is chosen and a node for the resulting gamestate is added to the tree. Form this state a *simulation* (or *rollout*) is performed.
**Simulation** A simulation is the run from a gamestate to a terminal state of the game. During simulation the moves are made according to the *default policy*, which typically chooses an action uniformly at random.
**Backpropagation** The final state is evaluated and the reward for each player is calculated. The reward is then used to update the statistics of the nodes visited during tree selection, which in turn is used for the *tree policy* in future iterations.

The search can be terminated at any point which makes MCTS especially suitable for games with time or computational resource constraints.

### 4.1.3 Determinization and Perfect Information MCTS (PIMCTS)

**Determinization** is the process of taking a gamestate with hidden information and creating ('determining') a perfect information state that is consistent with the hidden information. In essence, determinization chooses one out of the many possible perfect information gamestates for a given hidden information gamestate.

To deal with hidden information, PIMCTS performs several instances of MCTS, each on a different determinization. The best action is then chosen based on the results of the individual MCTS. This approach was successfully used for the game Bridge in [6], but has two main drawbacks, also described in [6] and further explored in [3] and [7]:
**Strategy fusion:** An agent can only make one decision in each hidden information state, but in different determinizations different decisions can be made based on the perfect information. This lets the agent 'believe' that he can distinguish between determinizations of the same state, which is not the case. Strategy fusion can cause the agent to choose a bad over a winning action.

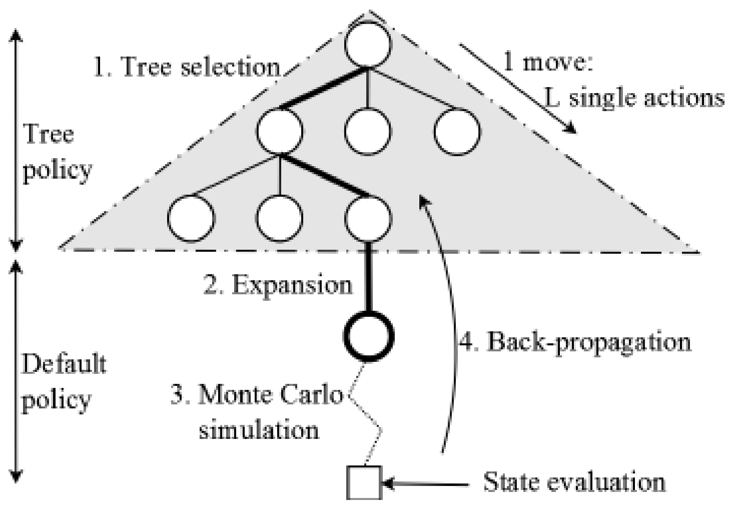


Figure 4.1: The basic steps of MCTS. Source [8]

**Nonlocality:** Some determinizations may be vanishingly unlikely (rendering their solutions irrelevant to the overall decision process) due to the other players' abilities to direct play away from the corresponding states.

### 4.1.4 Information Set MCTS (ISMCTS)

To tackle the problem of strategy fusion, *Cowling et al.* propose *Information Set MCTS* in [7]. Instead of building a different searchtree for each determinization, ISMCTS builds one search tree with the information from all determinizations using *Information Sets* instead of gamestates as the nodes in the tree. An information set is basically the set containing all indistinguishable gamestates from the the point of view of the player. This allows the sharing of information over different determinizations, which in addition to reducing strategy fusion, provides a basic algorithm with many possibilities for adaptions and improvements. Some of them are implemented in the agents described in the following sections.

## 4.2 Simple and Random Agent

To have a baseline, I created the *simple* and the *random* agent. The *simple* agent always passes whenever possible and plays a random single card when passing is not allowed. This is the simplest agent able to play the game, but it is also one of the worst possible agents.

The *random* agent always plays an action chosen uniformly at random from the possible actions.

## 4.3 Minimax Agent

To see how well minimax performs on Tichu, despite the big branching-factor, I implemented an agent using the minimax algorithm (with alpha-beta pruning). The maximal searchdepth that runs in acceptable time is 10, which corresponds not even to one trick. It takes more than a minute on my machine to complete one search.

To circumvent having to deal with hidden information, the agent has the possibility to 'cheat', that is, to observe the handcards of the other players, and therefore does not have to search several determinizations.

The simplest *utility* function for the agent is just *14 - "the number of handcards the player has"* since the goal is to get rid of all cards. However this leads the agent always to play the longest combination possible which turns out not to be a good strategy.

The fact that creating a good *utility* function needs a lot of *domain knowledge* and that minimax can't deal efficiently enough with the hidden information let me to quickly search for more suitable algorithms.

This minimax agent beats the random agent easily, but is in turn easily beaten by the MCTS agent (results in section 5.1)

Side-note: *David Pfander* ([4]) built an agent for Tichu with a carefully crafted *utility* function (requiring a lot of *domain knowledge*) and managed to achieve 'average human level' play.

## 4.4 Monte Carlo Tree Search Agents

MCTS nicely fits the requirements of "using little *domain knowledge*" and is able to deal with hidden information. That makes it an ideal algorithm for this project, and after reading [10, 7, 11] and their application of ISMCTS on the game *Dou Di Zhu* (which has many similarities to Tichu), I decided to concentrate, for the remainder of the project on agents using ISMCTS as base algorithm. In particular the following parts of ISMCTS have to be implemented and/or explored:

- Explore possibilities to make better *determinizations* without using *domain knowledge*
- Reducing or managing the branching factor
- Adding improvements over the ISMCTS and compare them
- Finding a better *default policy*(in the simulation phase) than the random strategy

In the remainder of this section, I discuss different enhancements for the ISMCTS-Tichu agents related to one of the 4 parts. Each of those enhancements are then evaluated in the *Experiments* section.

### 4.4.1 Default ISMCTS Agent

The *default ISMCTS agent* builds the basis for the agents improving on the various parts of the algorithm. In this section the *tree policy*, the *state evaluation* and the *selecting of the best action* are determined and used for all other ISMCTS agents, except when stated otherwise.

#### Tree selection

As hinted at in the *Background* section, the simplest and most often used *tree policy* uses the UCT (Upper Confidence bound for Trees) formula proposed by [5]. It is derived from viewing the selection problem as a multiarmed bandit problem where each action is an arm and the reward is the result of doing a Monte Carlo simulation after choosing that action.

To be able to use the UCT formula, each node ($n$) keeps track of the number of times it has been visited ($v_n$) and the sum of the rewards the player got after visiting the node ($r_n$).

The *tree policy* is then: For a given node $p$, select the childnode $c$ that maximizes following formula (ties are broken arbitrarily):

$$UCT = \begin{cases} \frac{r_c}{v_c} + C\sqrt{\frac{2\ln v_p}{v_c}}, & \text{if } v_c > 0 \\ FMU, & \text{otherwise} \end{cases}$$

where $r_c$ is the total reward of the child, $v_p$ is the number of times the current (parent) node has been visited, $v_c$ the number for the child node and $C > 0$ is a constant balancing exploration and exploitation ($\sqrt{2}$ is recommended by [7]).
It is generally recommended that $FMU = \infty$ (First Move Urgency), so that not yet visited child nodes will be visited first before any of them are expanded. However $FMU$ can be set to any value, allowing 'good' children to be expanded before some of their siblings are visited for the first time.

I didn't tune neither $C$ nor $FMU$ and kept them at $\sqrt{2}$ and $\infty$ respectively.

**Evaluation**

At the end of a rollout, the final state has to be evaluated and the reward $r$ is backpropagated trough the gametree. The evaluation thus plays a central role on determining the UCT value of nodes.

I considered following different evaluation strategies (remember that the players from the same team get the same amount of points at the end of a round):

**Absolute Points,** Each team gets the unaltered points as reward.

**Absolute Normalized Points,** The absolute points normalized between -1 and 1

**Relative Points,** Each team gets the difference of points to the other team as the reward. For example, the round ended $70 : 30$, then the first team gets $70 - 30 = 40$ reward, the second $30 - 70 = -40$ reward.

**Relative Normalized Points,** The relative points normalized between -1 and 1

**Ranking Based,** This evaluation rewards the teams based on the ranking at the end of the round:
- $+1$ for a doublewin
- $-1$ for a double loss (enemy has doublewin)
- 0.5 for a player of the team finishing first (but no doublewin)
- 0 when an enemy player finished first (but no doublewin)

Note that those evaluation strategies suffer from the *credit- assignment problem.* Both agents get the same reward, regardless of the order they finished. Thus a single agent can't tell whether he or the teammate played well. In the case of MCTS this does not matter since the agent does not 'learn' from rewards. It may however influence reinforcement learning based agents.

It turns out that the *ranking based* evaluation produces better results than any of the other strategies (experiments in section 5.3). Therefore this evaluation was used for all agents.

### Selecting the best action at the end

After the search completed, the best action from the initial search-state has to be determined. The recommended strategy is to select the action corresponding to the most visited node. And indeed that is what I ended up implementing after trying out the two other obvious strategies: taking the node with *highest UCB value* and taking the node with *highest average reward*.

The experiments confirming the recommendation can be found in section 5.9.

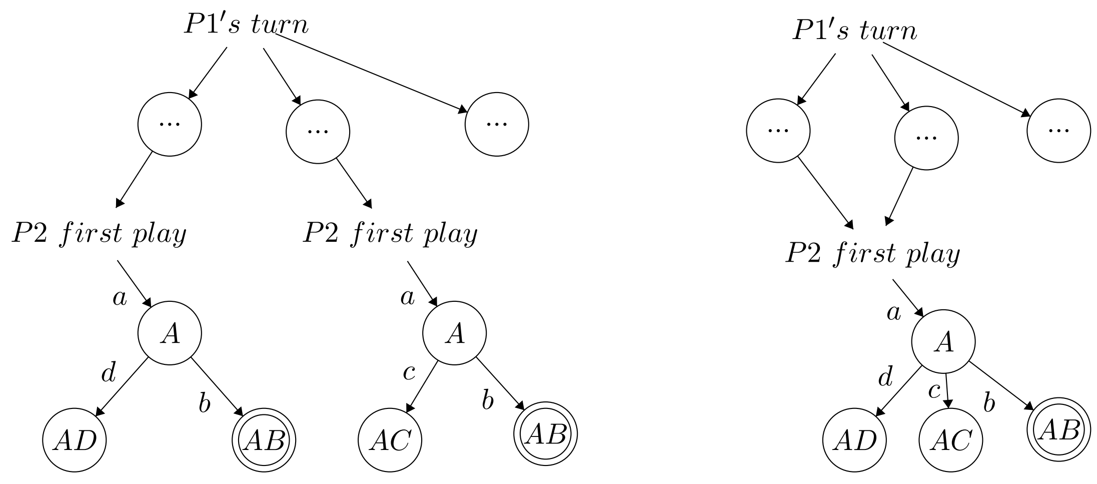
## 4.4.2 Managing the branching factor

In [7] it is shown that ISMCTS does not perform significantly better than PIMCTS in *Dou Di Zhu*. However, they suggest this is due to the high branching factor of the game and that in each determinization new possible actions are discovered. Reducing those two factors would increase the advantages of ISMCTS over PIMCTS.

A big reduction of the branching factor comes from the observation that many possible combinations in a given hand are logically identical and thus only one of them has to be considered (see the *Implementation* section).

Other reductions are achieved by the following two enhancements to ISMCTS.

### Episodic Information Capture and Reuse (EPIC)



(a) An example of the *position in epic*. The two circled states have the same *position in epic* but are in a different part of the gametree. Both are reached by action $a$ followed by $b$ from a state where player 2 can play first. However state $AD$ and $AC$ have different *position in epic*. The nodes labeled with $A$ and *P2 first play* also have the same *position in epic*

(b) The EPIC-gamegraph corresponding to the gametree in (a). The two nodes labeled $AB$ are merged as well as the states labeled $A$ and the state where Player 2 can play first.

Figure 4.2: EPIC example

EPIC is an enhancement proposed by [11] and aims to take advantage of the episodic nature of certain games. The idea is that instead of discovering the same strategy for very similar situations (episodes) in different parts of the game tree, EPIC puts them together such that those similar

situations are the same and thus only have to be discovered once.
It yielded good improvements in *Dou Di Zhu* and it stands to reason that it does also help in Tichu. *Dou Di Zhu* has the same trick system as Tichu, but it is played with only 3 players and the playable combinations are slightly different. But overall the two games are similar very to each other.

In *Dou Di Zhu* and Tichu, an episode is naturally defined by a trick (from first play until all players pass consecutively). The *position in episode* of a node is the path from the node upwards until the beginning of the trick is reached. An example is shown in Fig. 4.2a. Similar on how all nodes belonging to the same *information set* were put together for ISMCTS, for EPIC, all states with the same *position in epic* are merged into one node creating an EPIC-gamegraph. The result for the example is shown in Fig. 4.2b. For Tichu, the gamegraph contains 4 ”root nodes” corresponding to the 4 states where a particular player can play first.

The search is then performed on this gamegraph according to ISMCTS. To implement EPIC very little change to ISMCT is required. Only the way how gamestates are mapped to nodes has to be modified. The rest of the algorithm is identical.

EPIC reduces the branching factor dramatically since many different nodes are merged. The ordering of different tricks in the game is basically removed, which, of course, introduces some new difficulties and is probably the main reason why it did not perform well on Tichu.

**Move groups**

Another way to reduce the branching factor and possibly improve UCT decisions are *move-groups*. The idea is to divide the actions into (not necessarily disjoint) groups. Those groups are then inserted as an additional layer after each node in the gametree. The decision in the *tree policy* then becomes a two-step process. First select a group and then an action belonging to that group (UCT can be used in both phases). Move groups are introduced in [12] and further examined in [13].

This enhancement reduces the effective branching factor and has been shown to help with move selection when similar valued moves appear. However, it is not trivial to find good groups. [13] found that random groups did neither improve nor worsen the simulation efficiency, but increased the speed. Grouping the best nodes together seems to yield the best result, but finding the best nodes is exactly the problem.

I decided on putting all actions of the same *combination-type* into the same group. With the idea that they most of the time lead to similar results and thus the chance that the best actions are in the same group are increased.

Surprisingly, ISMCTS with *move-gropus* performed very poorly against the default ISMCTS agent (more in section 5.10).

### 4.4.3 Determinization

*Whitehouse* shows in [11, p. 54+] that for the game *Dou Di Zhu*, more than 20 determinizations per search yield diminishing returns, and the more simulations per determinization are made, the better the agent plays. Assuming this is value is similar in Tichu I decided to generate 30 different determinizations per search, just to be on the save side (Tichu has one player more).

The problem of creating determinizations in Tichu boils down to infer the remaining handcards of the other players. Even for human players this is hard and involves a lot of guesswork and knowledge about the game and the other player. However, it turns out that the extend of the advantage of good determinizations it is not exactly clear. For more see section 5.2 and [7].

To avoid using *domain knowledge*, I decided to use data of existing human games. *Brettspielwelt.de* is an online platform that provides the possibility to play different games online against other other players. One to the offered games is Tichu. The game-logs of played Tichu games can be found at *log.tichumania.de* [15], where about 500'000 games of 5'000 different players are stored. Sadly, the *Brettspielwelt* team did not respond to my inquiry for the entire dataset displayed there, and I had to write a scraper to obtain the gamelogs. In the following sections the obtained data is visualized and different strategies to exploit the pattern in the data are discussed.

### Random

The easiest determinization strategy is to distribute the unknown cards uniformly at random amongst the other players. Note that the observer player knows his own handcards and which cards already have been played. Therefore it can deduce exactly which cards are among the handcards of the other players, as well as how many cards each player has.

### The Data

The goal is to determine the prior probability of a given set of cards of size $n$ (a player has $n$ handcards) to contain a particular combination. That is, the answer to the question: "what is the probability that a player, having 6 cards left, possesses a pair of kings"?

First I have to introduce the notion of a **General Combination**. A "normal" combination is a set of cards, each with its rank and suit. A *General Combination* however is suit agnostic. So the combinations $(K\heartsuit, K\diamondsuit)$ and $(K\diamondsuit, K\clubsuit)$ are two distinct combinations, but belong to the same *General Combination*, which is $Pair(King, 1)$
More formal: A *General Combination* is the triple (*combination-type*, *height*, *length*), where *combination-type* is one of [Single, Pair, Triple, Square, Fullhouse, Straight, Pairstep, SquareBomb, StraightBomb], *height* is the value of the combination and *length* denotes how many cards belong to the combination (this is used to distinguish straights and pairsteps).

There are 255 different *general combinations*, about 100 of them are extremely rare (a straight of length 14 practically never appears for example).

To approximate the probability for each of those *general combinations* appearing in different sized handcards, I scraped around 2 million different handcards sets[1] from the *tichumania* website and counted which *general combinations* appear how many times. In total there are about 12 million *general combinations* in those 2 million handcards, giving an average of about 6 different *general combinations* per hand. Some of the results are shown in Fig. 4.3, the remaining plots can be found in the appendix 6.2.

Figure 4.3a shows the distribution when a player has only one single card left. With one card, only the *Single combination-type* is possible.
It is clearly visible that the player is more likely to have either a low card (2, 3 or 4) or a high card (King, Ace) than to have a 7 or 8. The special cards have an overall lower probability of occurring

[1]In a game, each time a player plays some combination, the remaining handcards are stored (as well as the initial cards at the beginning of the game.)

because there is only one of each and not four. Nevertheless, The Mahjong almost never is the last card[2], while the Dog is almost as likely to appear as a 7. It makes sense that either low or high cards "survive" longer than medium ranked cards. High cards are advantageous at the end of the game because it is easier to finish with them, so players take care to keep them, where as low cards are harder to get rid of, and therefore they persist until the end. Another reason why low cards are that common, is that an often seen strategy is to play one high card as second last to win the trick and then to play the low card to finish. The high probabilities for high cards in Figure 4.3b confirms this.

Figure 4.3b shows the probabilities for 2 remaining handcards. It is very likely that at least one of the two cards is an Ace, followed by the King and Queen. Interestingly, the dragon and phoenix are also rather likely. With 2 cards, it is possible to have a pair, however they seem to be distributed uniformly and only very small trends can be inferred (for example an Ace-pair is lower, while the 2-pair is slightly higher than all others).

Skipping some steps, figure 4.3c shows the probabilities for 5 remaining handcards. The single cards are still clearly distributed, and also the pairs seem to diverge from a uniform distribution. All other combinations however are too infrequent to have a clear distribution.

Figure 4.3d shows the distribution for 12 cards and it is clear, the more cards, the closer to uniform each combination distribution gets. This makes sense since in the beginning (with 14 cards) they should be uniformly distributed due to the random deals.

This analysis shows that especially towards the end of the game, this prior data can help to create better (more accurate) determinizations compared to the random determinization.

In the next two sections, two strategies to create determinizations using this data are described.

### Combination Determinization

This strategy selects entire combinations and gives them to the player which is most likely to have it.

First, all possible combinations that might appear in any of the handcards of the players are generated. For each of those combinations three probabilities are calculated and summed together (one for appearing in each players handcards). Each possible combination thus has a score approximately proportional to the probability of actually appearing in any of the handcards. Then combinations are repeatedly sampled (weighted by their score) and "given" to the player most likely to have them, until all cards are distributed. There is of course some bookkeeping to make sure all players get the right amount of cards and that no card is lost or created during the process. But this is an implementation detail.

The problem with this strategy is that combinations that are very unlikely are selected too often because the differences between the probabilities are not especially high. And indeed, the experiments show that it is even worse than the random determinization (section 5.4).

[2]It is customary (and smart) to play the Mah-Jong in the first trick, which explains the low probability.

#### Single Determinization

The data shows that single cards are the only *combination-type* other than the pair, which is not distributed uniformly at random. This strategy aims to exploit that. Instead of generating all possible combinations, it looks at each player in ascending order of their number of handcards and gives them one card at the time, sampled from the remaining cards.
So players with less cards left, get their cards first and the determinization is more likely to "hit" correct cards for them. From experience it is more important to know the enemies cards when they have only a few left, since I want to prevent them from finishing.

In the experiments, this strategy performed at least as well as the random determinization (section 5.4).

#### Machine Learning approach

I tried to train 14 different models (one for each size of handcards $N$). Given all the unknown cards and all cards already played by a player, the model should predict which $N$ cards the player has in his hands. The idea being that the history of actions and the remaining cards are correlated. Together with the possible cards, it might be possible to predict something. However, I did not manage to create a model that converges. The data is probably too noisy and too sparse to infer anything. In addition, "near misses" are also counted as wrong. The evaluation should have a concept of "closeness" of the cards. For example giving a 10 instead of a 9 is much better than giving a King. The models often get one or two cards right, but that is not better than random. Sadly there was no more time left to explore more in this direction.

### 4.4.4 Rollout

As a reminder: The rollout phase simulates a game from a given gamestate to a terminal one, following a (fast) action selection strategy (*default policy*). The goal is to estimate an initial value for the given gamestate.

The rollout phase is an important part for the overall playing strength of the agent. The closer the *default policy* imitates the strategy of the enemy, the better the initial estimation of each gamestate gets, and with it the *tree policy* can pick the next action more accurately, which in turn yield an better final decision.
Funnily enough, this implies that the best *default policy* when playing against the *random* agent, a random *default policy* would be the best.

#### Random Rollout

The random rollout *default policy* selects one action uniformly at random out of all legal actions. It is very fast and enables many rollouts to be performed. However, it suffers from multiple problems. One of which (in my opinion the most problematic) is that it, in some sense, only approximates the number of paths that lead to a win (or loss). It does not account for "obviously good/bad moves" being played more or less often and thus simulates games that will never happen, resulting in quite random rewards.
Despite this, it still works because in most cases, good states more often than not lead to a win, and bad states have more paths to a loss.

In the endgame of Tichu, when a big part of the gametree can be explored during a search, a random *default policy* might even be sufficient. Especially when only 2 players remain, there is no hidden information anymore (all cards can be inferred) and a rollout is quickly done (the end of the game is near).

### Last Good Response With Forgetting (LGRF)

LGRF aims to be an enhancement to the random *default policy*. Actions are still selected uniformly at random, with one exception: When a previous stored action is encountered, the response to that action is always selected. During a rollout, all pairs of actions are recorded as *(action→ response)*-tuples. After the rollout, the responses of the players that won are stored and the responses of the players that lost are deleted (forgotten) from the storage.[11]

The idea is to keep the good actions, and forget the bad ones. This works good for games where the same action always has a similar 'value' independent of the circumstances or actions often depend on the previous one. Tichu does not exactly fulfill this criteria. Many *(action→ response)*-tuples make sense, for example, "play single 2 → play single 3" is a perfectly good reaction. However, for a player to win he plays a *King* on a 2. Then the "play single 2 → play single *King*" is stored, and in the next rollout the player always plays a *King* on a 2, which is probably bad.

Then there is the problem with the *pass* action. It is by far the most used action and seldom played as a response on another action. Furthermore, combinations are played on other combinations, independent of whether a player passed in between or not. Thus pass actions are ignored with respect to LGRF.

Sadly, but not surprisingly, LGRF decreased the playing strength of an agent compared to a purely random rollout.

### No Rollout

There is the possibility of doing no rollout at all. That is, all actions are added to the gametree and chosen according to UCT. Note that this still mostly ends up in randomly selecting actions since UCT selects randomly among, as of yet, unvisited nodes. However, it makes sense together with EPIC. Since the EPIC-gamegraph contains loops, the algorithm will reach nodes that are already (at least) partially explored and there the additional UCT information might improve the search.

### Neural network Rollout

*AlphaGo* uses MCTS, with a deep neural network (NN), trained on human expert games, as *default policy* [16]. They use another NN as *tree policy*, which is trained by self-play with reinforcement learning methods. So I thought whether it is possible to train a NN for Tichu and use it as *default policy.* See section 4.5 for the description of the agents.

## 4.5 Neural Network Agents

As an enhancement to the MCTS agents, several Deep-Q-learning (DQN) agents were implemented. The agents only differ in the network architecture. To train the network, the Deep-Q-learning algorithm implemented in the Keras-rl package[17] was used.

The goal was to use the neural network agents as the *default policy* in the MCTS agent.

### 4.5.1 Architectures

The input to the network consists of the handcards of all players, the combination on the table and some information about the current ranking. I tested two different 'encodings' for a set of cards:

1. Boolean Vector of length 56: Each card has a different number ranging from 0 to 55. If a card with number $n$ is contained in the set, then the bit at position $n$ in the vector is set to 1 (or $True$).
2. Integer Vector of length 17: Each rank has a different number ranging from 0 to 16. This includes special cards (Dog=0, Mahjong=1, Two=2, ..., King=13, Ace=14, Dragon=15, Phoenix=16). The vector then represents how many cards of each rank appear in the set.

Both representations have their advantages and disadvantages. 1) leads to a rather long but boolean input, while 2) discards information about the suits of the card in favor to a shorter, but numeric representation.

I tested two different architectures. Both have 2 hidden, fully connected layers, with 'elu'[3] as activation function.

1. Together: Connects the input vector directly to the 2 hidden layers.
2. Separate: Takes the input vector and splits it into the 6 separate components (4 players handcards, the combination on the table and the ranking-information). Each component then connects to a 'private' hidden layer before concatenating them to the second 'shared' hidden layer.

The output layer has 258 neurons, which corresponds to the number of possible actions at any time in the game.

A visualization of the architecture can be found in Fig. 4.4

A particular challenge was to make sure the agent plays only legal moves. This was solved by adding a last layer which subtracts 500 from the output neurons representing a illegal action. The Q-values (output neurons) of illegal actions are thus lower than the others, and the agent chooses an legal action. To do so, a vector encoding the legal actions had to be given as another input to the network.

The essential information about the current ranking of the players is encoded in 2 bits. The first bit indicates whether the teammate finished first, the second bit whether an enemy finished first. Note

[3]see https://en.wikipedia.org/wiki/Activation_function

that at most one of the two bits is set to 1 since only one player can finish first. In the *Together* architecture, the bits are contained in the input vector, while in the *Separate* architecture they also are a separate input.

### 4.5.2 Training

To train the network, the Deep-Q-learning algorithm implemented in the *Keras-rl* library[17] was used.

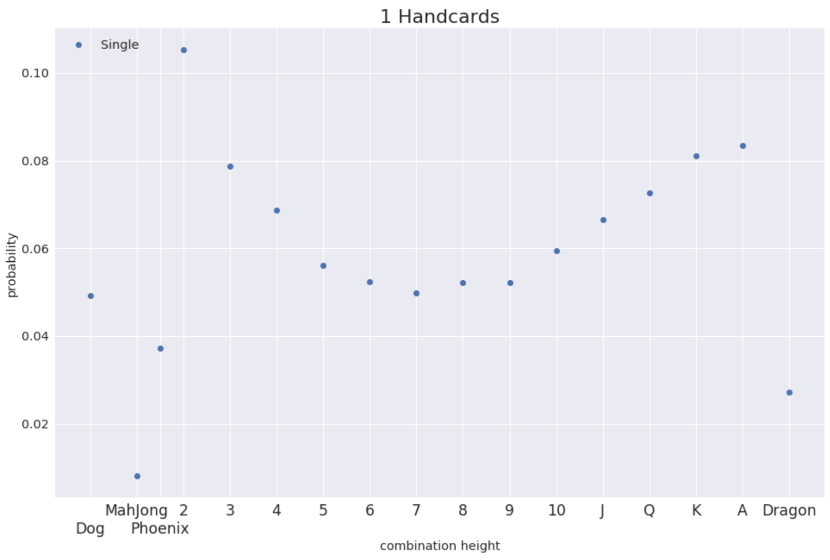
As training policy for the DQN-agent I used the Bolzman-policy with linearly decreasing $\tau$. With the Bolzman-policy, each action $a$ has following probability of being selected:

$$p(q_a) = \frac{\exp(\frac{q_a}{\tau})}{\sum_{k \in Actions} \exp(\frac{q_k}{\tau})}$$

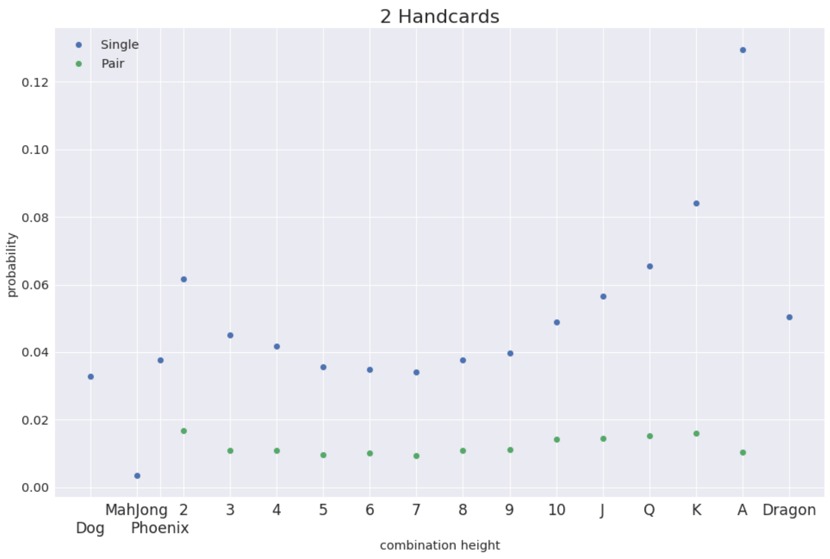
where $q_a$ denotes the value of the output neuron corresponding to action $a$ (the q-value of $a$). $\tau$ determines the exploration/exploitation ratio. The higher $\tau$, the more exploration is done.
This function selects the highest q-value with very high probability when $\tau$ is small (exploitation). However, when $\tau$ is big, all actions have the same probability of being selected (exploration).

During training, the agent plays multiple games with three other agents. Those other agents are either 3 random agents, 3 other (already trained) DQN agents or 3 MCTS agents. Each agent is trained for 1 million steps, which corresponds approximately to 300'000 games.

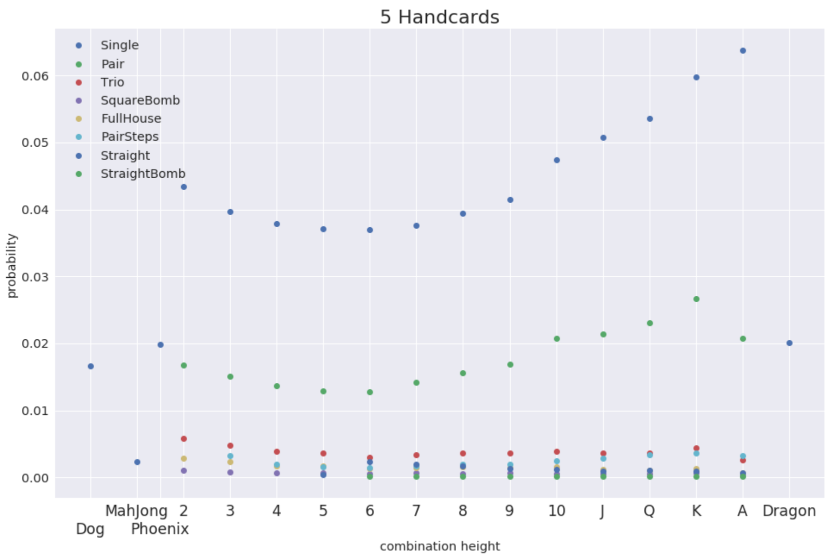
Some trainings are visualized in Fig. 4.5



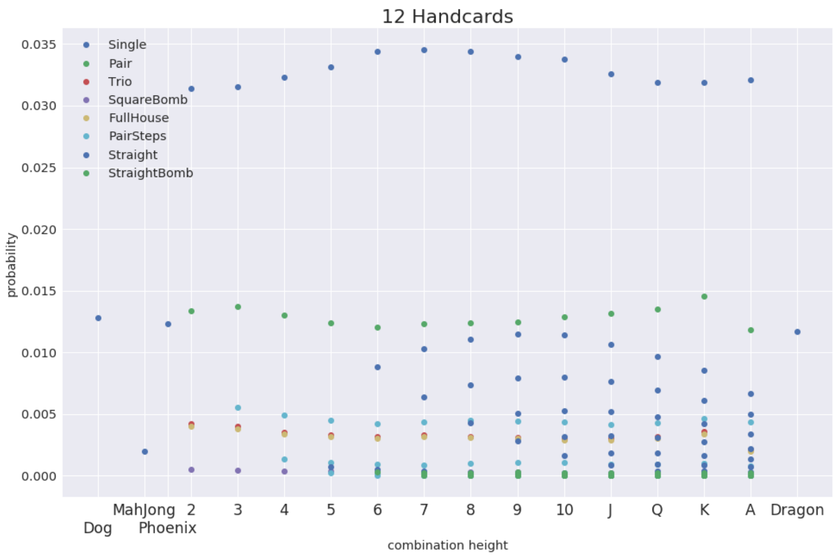
(a) 1 card: Only singles are possible with one card left. Low and high cards are more likely to appear than middle ranked cards. The Mah-Jong is seldom the last card, but the Dog appears more often than expected.



(b) 2 cards: Single cards are similar but not the same as in (a). Higher cards are more now more likely than lower cards. Pairs might appear, but there is no apparent significant trend.

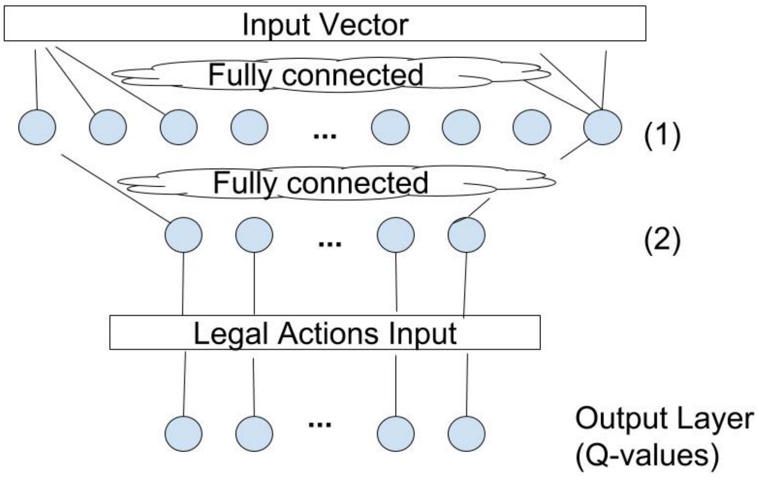


(c) 5 cards: Single cards are very similar to (b), and high Pairs are slightly more likely than lower ones. All other combinations seem to be distributed uniformly at random.

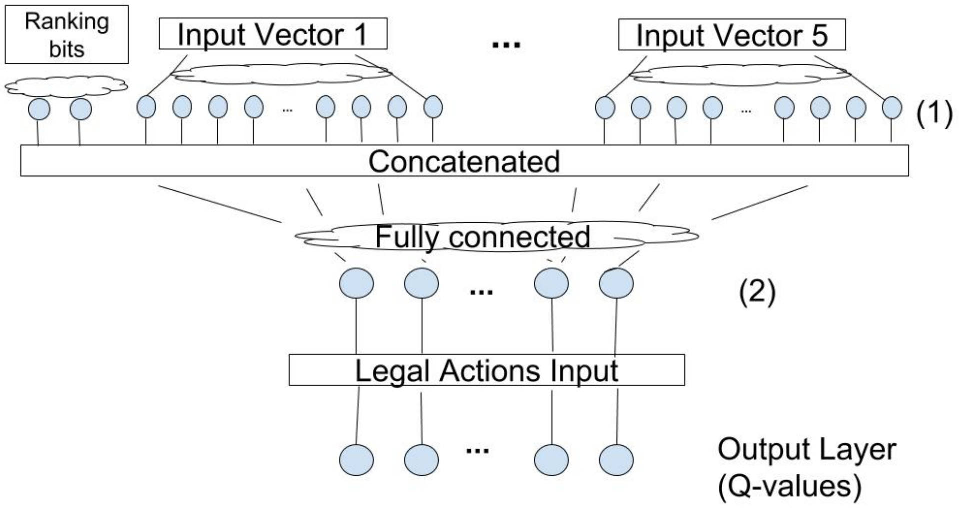


(d) 12 cards: No obvious trend is visible. This is not surprising as only 2 cards have been plaid.

Figure 4.3: The Plots for the Handcard Data

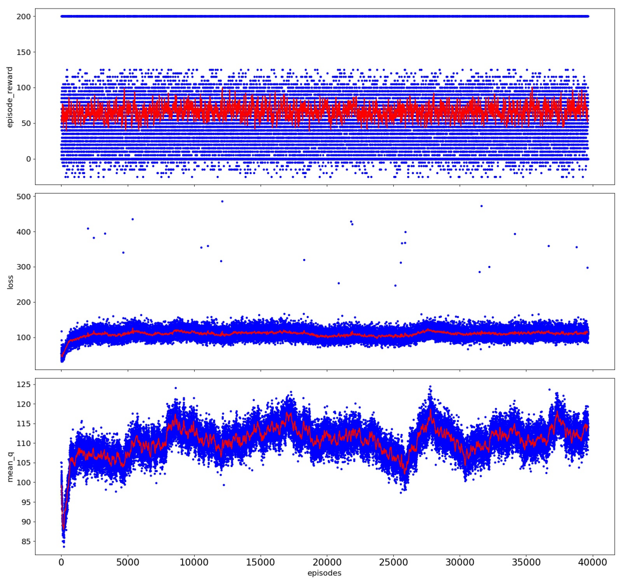


(a) Together NN architecture. Layer 1) has the same number of neurons as the input vector and is fully connected to the second layer which has 258 neurons. The *legal Actions input* reduces the q-values of the illegal actions by 500 in order to prevent the agent from selecting them.

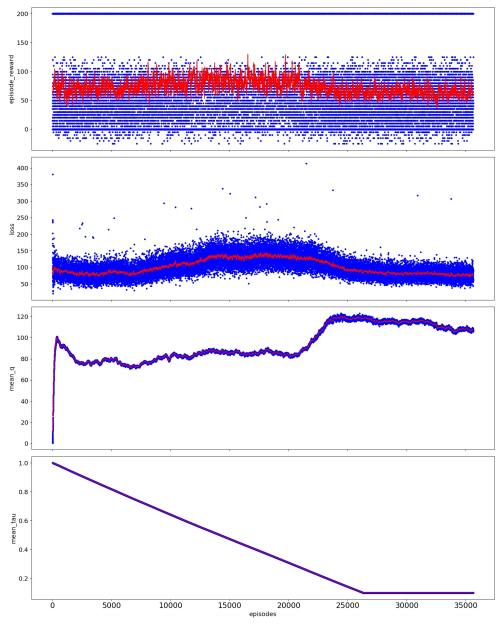


(b) Separate NN architecture. Each input is fully connected to a "private" set of neurons (1). Those neurons are then concatenated to a single layer which is in turn fully connected to the second hidden layer (2) of length 258.*legal Actions input* is the same as in (a).

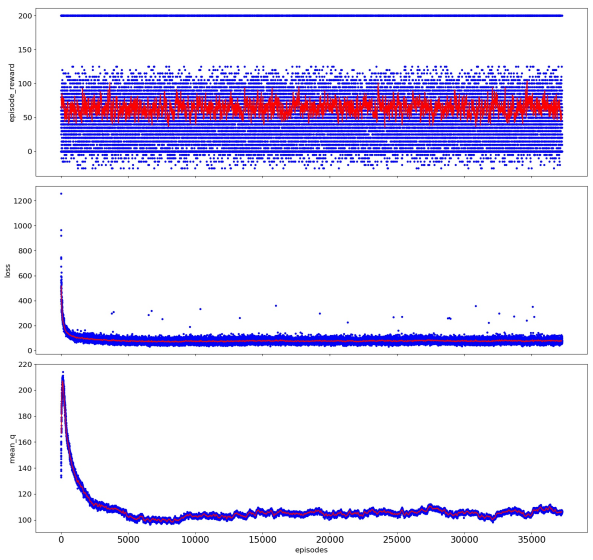
Figure 4.4: The two Neural Network Architectures.



(a) DQN with the *Seperate* Architecture and the 56-boolean input, trained for 1 million steps against 3 random agents. There is no real improvement during the training. The mean-q values approximate the episode reward somewhat.



(b) The same agent as in (a) but trained against itself. No improvement can be seen in the episode reward, but the q-values increase when $\tau$ reaches 0.2.



(c) A second training period for the agent in (b), but with a constant $\tau$ of 0.2. All values converge really quickly and no change happens afterwards.

Figure 4.5: Two trainings on different environments. b) and c) are two training periods of the same agent.

# Experiments

In this section the tournaments done with the agents are described and the results are shortly discussed.

To evaluate two agents against each other, a full Tichu game is played with two instances of each agent forming a team. Depending on how close the agents are and how long it takes them to play a game, the targeted points may range from 1'000 up to 100'000 points. The difference of the final points is then the measure on how much better one agent is to another. There is of course a rather high variance in each game result (depending mostly on the dealt cards and luck) which is why longer games give a better impression on the relative strength of the agents.

Several agents can play a tournament, where each agent plays one game against each other agent.

The Default MCTS Agent used in most experiments is the one from 4.4.1 with random determinization and random rollout policy.

## 5.1 Minimax, Random, default MCTS Tournament

### Setup

**Target Points:** 2000
**Agents:**

- MinimaxAgent (from 4.3) with search-depth 9
- RandomAgent (from 4.2)
- Default MCTS Agent with 100 iterations.

### Results

| **Team1** | **Result** | **Team2** |
|---|---|---|
| **Minimax** | 2070 : 730 | Random |
| Minimax | 145 : 2055 | **MCTS** |
| Random | 390 : 2110 | **MCTS** |

Table 5.1: Result of the Minimax vs. Random vs. default MCTS Tournament.
The MCTS agent clearly beats the other two, while Minimax also clearly beats the Random algorithm.

### Discussion

In this tournament a baseline between the MCTS and the random agent is established. And it is confirmed that MCTS easily beats Minimax, which easily beats the random agent. This result is not surprising not least because the utility function for minimax is very basic.

A small surprise is that MCTS wins higher against Minimax than against the random agent, but this may just be a chance event.

## 5.2 Cheating Tournament

A tournament to get an idea how useful it is to know the cards of the other players.

### Setup

**Target Points:** 7000
**Agents:** All agents are Default MCTS Agents with 100 iterations and varying degrees of cheating. Agents cheat by looking at a given proportion of the enemies cards and fix them in each determinization, giving the agent an advantage.

- NoCheat Agent: Does not cheat
- Cheat2 Agent: Looks at 20% of the enemy cards
- Cheat6 Agent: Looks at 60% of the enemy cards
- Cheat8 Agent: Looks at 80% of the enemy cards
- FullCheat Agent: Looks at 100% of the enemy cards

### Results

| **Team1** | **Result** | **Team2** |
|---|---|---|
| NoCheat | 5585 : 7015 | **Cheat2** |
| NoCheat | 5210 : 7010 | **Cheat6** |
| NoCheat | 6490 : 7070 | **Cheat8** |
| NoCheat | 5000 : 7090 | **FullCheat** |
| Cheat2 | 6945 : 7085 | **Cheat6** |
| Cheat2 | 4795 : 7105 | **Cheat8** |
| Cheat2 | 5520, 7040 | **FullCheat** |
| **Cheat6** | 7025 : 6475 | Cheat8 |
| Cheat6 | 6350: 7050 | **FullCheat** |
| Cheat8 | 6770: 7030 | **FullCheat** |

Table 5.2: The agents cheating more win their matches, with the exception that Cheat6 beat Cheat8.

### Discussion

The results are quite variable. The more cheating agents almost always win their matches against more honest agents, but not as decisive as may be expected. In 5 out of the 9 matches, the loosing player gets more than 6000 points.
Cheating seems to improve the agents chance to win, but not by a lot.

In *Dou Di Zhu*, [7] found that knowing the handcards of the other players is only a significant advantage in 1/3 of all the deals. This may be similar in Tichu and would explain the results. For clearer results more experiments have to be made.

## 5.3 Reward Tournament

To determine the best evaluation strategy of terminal states. The different strategies are described in section 4.4.1.

### Setup

**Target Points:** 5000
**Agents:** All agents are Default MCTS Agents with 100 iterations but rollouts generate the rewards differently. The names are taken from section 4.4.1

- Absolute Points
- Absolute Normalized Points
- Relative Points
- Relative Normalized Points
- Ranking Based

### Results

| **Team1** | **Result** | **Team2** |
|---|---|---|
| **Ranking** | 5125 : 2275 | Absolute |
| **Ranking** | 5050 : 4650 | Relative |
| **Ranking** | 5165 : 1035 | Relative Normalized |
| **Ranking** | 5000 : 600 | Absolute Normalized |
| Absolute | 3635 : 5065 | **Relative** |
| **Absolute** | 5135 : 865 | Relative Normalized |
| **Absolute** | 5060 : 840 | Absolute Normalized |
| **Relative** | 5105 : 1195 | Relative Normalized |
| **Relative** | 5075 : 825 | Absolute Normalized |
| **Relative Normalized** | 5030 : 2860 | Absolute Normalized |

Table 5.3: *Ranking* wins all its matches. *Relative* wins all but one, *Absolute* wins 2 and *Relative Normalized* wins only against *Absolute Normalized*, which looses all its matches.

### Discussion

The ranking and relative reward system seem to be clearly better than the others. Ranking also beat Relative, but not by a lot.

## 5.4 Determinization Tournament

To evaluate the three determinization strategies described in section 4.4.3.

### Setup

**Target Points:** 10000 (30000 for the match RandomDet vs SingleDet because the result is very close).
**Agents:** All agents are Default MCTS Agents with 10 seconds time to determine the next move. Each utilizes a different determinization strategy.

- RandomDet: Random determinization
- CombinationDet: Distributes entire combinations at once based on prior probabilities
- SingleDet: Distributes a card after another based on prior probabilities

### Results

| **Team1** | **Result** | **Team2** |
|---|---|---|
| **RandomDet** | 10030 : 2670 | CombinationDet |
| RandomDet | 27565 : 30095 | **SingleDet** |
| CombinationDet | 4090 : 10010 | **SingleDet** |

Table 5.4: *SingleDet* its two matches, *RandomDet* wins one

### Discussion

The Combination Determinization performs very poorly, but the Single Determinization is not worse than the Random Determinization. It wins the match against the Random Determinization but not by a huge margin. It seems that either the prior probabilities could not be exploited enough, or the accuracy of the determinizations is not as important as expected.

## 5.5 EPIC Tournament

A tournament to evaluate the strength of the EPIC enhancement to ISMCTS. The match MCTS vs Random was omitted since it is not relevant in this context. **Target Points:** 5000
**Agents:**

- EPIC: Default MCTS Agent with the EPIC enhancement
- MCTS: Default MCTS Agent (without EPIC)
- Random Agent

### Results

| **Team1** | **Result** | **Team2** |
|---|---|---|
| EPIC | 3140, 5160 | **MCTS** |
| **EPIC** | 5090, 1810 | Random |

Table 5.5: MCTS wins against EPIC but EPIC still clearly beat Random

### Discussion

Sadly the EPIC enhancement seems not to improve the playing strength of the Agent. This may have many reasons. I guess EPIC suffers a lot from the loss of trick ordering during a simulation. It can't distinguish between playing a combination at the beginning and playing it later on. But in reality it is important for example playing an Ace when the Dragon was not yet played is risky, while playing it after the Dragon almost guarantees the win of the trick.

## 5.6 Deep-Q learning Tournament

To compare the different network architectures.

### Setup

**Target Points:** 10000
**Agents:**

- Dqn56
- Dqn56-separate
- Dqn17
- Dqn17-separate
- Random Agent

### Results

### Discussion

Disappointingly, only Dqn56-separate manages to beat the Random agent. Both DQN17 agents were beaten rather close. It shows that creating an neural network that works is not that easy. It might just be that the agents need to be trained much longer to reach a higher playing strength. But it might as well be that the network architecture has to be different. In any case, more experiments are needed.

| **Team1** | **Result** | **Team2** |
|---|---|---|
| **Random** | 10185, 2015 | Dqn56 |
| Random | 7690, 10010 | **Dqn56-separate** |
| **Random** | 10170, 8830 | Dqn17 |
| **Random** | 10070, 9130 | Dqn17-separate |
| Dqn56 | 6390, 10110 | **Dqn56-separate** |
| Dqn56 | 4570, 10030 | **Dqn17** |
| Dqn56 | 1525, 10175 | **Dqn17-separate** |
| **Dqn56-separate** | 10165, 4835 | Dqn17 |
| **Dqn56-separate** | 10000, 2000 | Dqn17-separate |
| Dqn17 | 9735, 10165 | **Dqn17-separate** |

Table 5.6: Dqn56-separate wins all of its games, while the random wins all except against Dqn56-separate

## 5.7 Rollout Tournament

The tournament comparing the rollout strategies.

### Setup

**Target Points:** 20000
**Agents:**

- NNRollout: Default MCTS agent but with a DQN-agent as default policy.
- Default MCTS Agent (with random rollout policy)
- LGRF: Default MCTS agent with the LGRF rollout policy.
- Random Agent

### Results

| **Team1** | **Result** | **Team2** |
|---|---|---|
| **NNRollout** | 20095 : 4035 | Random |
| NNRollout | 7930 : 20010 | **DefaultMCTS** |
| **NNRollout** | 20035 : 17480 | LGRF |
| LGRF | 4010 : 20020 | **DefaultMCTS** |
| **LGRF** | 20090 : 8910 | Random |

Table 5.7: DefaultMCTS beats both NNRollout and LGRF decisively. And NNRollout beats LGRF. The Random agent looses all matches

### Discussion

Neither NNRollout nor LGRF manage to beat the power of randomness. Apparently, the random rollout is quite a passable strategy. LGRF suffers from some problems already discussed in section 4.4.4 so this is not further surprising. The DQN-agent already performed badly against the random algorithm in 5.6, thus the result is not that surprising.

## 5.8 Split Tournament

To determine how efficient the MCTS agent is at the beginning of the game, the *Split Agent* was created. The Split Agent contains two other agents and depending on how many handcards the player has, the first or the second agent determine the next action. The number of handcards when the agents switch is called the *switch length*. The first agent decides until the number of handcards reach the *switch length*, from then on the second agent decides.

### Setup

**Target Points:** 10000
**Agents:** All agents are Split Agents with a Default MCTS Agents with 100 iterations as first agent and a Random as second agent.

- Sw13: *switch length* at 13
- Sw12: *switch length* at 12
- Sw11: *switch length* at 11
- Sw10: *switch length* at 10
- Sw9: *switch length* at 9
- Sw7: *switch length* at 7
- Sw5: *switch length* at 5
- Sw3: *switch length* at 3
- Random Agent

### Results

| **Team1** | **Result** | **Team2** |
|---|---|---|
| Sw13 | 9025 : 10175 | **Sw12** |
| Sw13 | 7565 : 10035 | **Sw11** |
| Sw13 | 9085 : 10015 | **Sw10** |
| Sw13 | 8935 : 10065 | **Sw9** |
| Sw13 | 9150 : 10050 | **Random** |
| Sw12 | 9160 : 10040 | **Sw11** |
| Sw12 | 7980 : 10020 | **Sw10** |
| Sw12 | 9065 : 10035 | **Sw9** |
| **Sw12** | 10080 : 9720 | Random |
| **Sw11** | 10005 : 9695 | Sw10 |
| **Sw11** | 10060 : 9140 | Sw9 |
| **Sw11** | 10065 : 7535 | Random |
| Sw10 | 8075 : 10025 | **Sw9** |
| **Sw10** | 10050 : 5450 | Random |
| **Sw9** | 10010 : 6790 | Random |
| Sw9 | 8100 : 10100 | **Sw7** |
| Sw9 | 3690 : 10010 | **Sw5** |
| Sw9 | 3730 : 10170 | **Sw3** |
| **Sw9** | 10045 : 5055 | Random |
| Sw7 | 7700 : 10000 | **Sw5** |
| Sw7 | 5275 : 10125 | **Sw3** |
| **Sw7** | 10020 : 4980 | Random |
| Sw5 | 7030 : 10170 | **Sw3** |
| **Sw5** | 10065 : 3135 | Random |
| **Sw3** | 10065 : 2335 | Random |

Table 5.8: The lower switch agents win most of their matches against a higher one.

### Discussion

The lower switch agents win 22 out of 25 matches. Only the Sw13 loosing against the random agent and Sw11 winning two matches against Sw10 and Sw9. This indicates that MCTS helps even when only used at the very beginning of the game.

## 5.9 Best Action Tournament

To determine how the best action should be selected after the search (see section 4.4.1).

### Setup

**Target Points:** 5000
**Agents:**

- MostVisited

- MaxUCT value
- Average Reward

### Results

| Team1 | Result | Team2 |
|---|---|---|
| **MostVisited** | 3190, 1310 | MaxUCT |
| **MostVisited** | 3050, 2250 | Average Reward |
| MaxUCT | 2345, 3055 | **Average Reward** |

Table 5.9: MostVisited wins all its matches, and AverageReward places second.

### Discussion

MostVisited clearly is the best way to choose the best action. This confirms the suggestion made in [7, 11].

## 5.10 Move Groups Match

A match between a MCTS agent using movegroups against one without them.

### Setup

**Target Points:** 10000
**Agents:**

- Movegroup: An default MCTS agent with move group selection
- NoMovegroup: An default MCTS agent without move group selection

### Results

| Team1 | Result | Team2 |
|---|---|---|
| Movegroup | 4535 : 10065 | **NoMovegroup** |

Table 5.10

### Discussion

The Movegroup Agent gets defeated quite clearly, but there is not enough time to conduct more experiments and to find the reason. I guess the definition of the groups might be the problem and another approach would yield better results.

## 5.11 Playing against a Human

During the semester I played every once in a while against some version of the ISMCTS agents. Here are some points I noticed:

**Overall Play** Against a non-cheating agent even a beginner may win most matches against the agent. The agent makes especially in the beginning quite a lot of suboptimal moves, in particular, it quickly plays its highest cards at the earliest opportunity. The Dragon typically is played in the first 2-3 tricks. However, towards the end of the game the agent plays better, and if you make a mistake towards the end, it is hard to win. The possibilities towards the end of the game are much smaller, and the agent can much better 'guess' the remaining cards of the enemies.

**Cheating Agent** The cheating agent, unsurprisingly, plays stronger than the non-cheating ones.

**Teamplay** Towards the end, it seems that the teammate agent actually helps. For example, the teammate tends to pass on a trick where I'm leading, even though it could have played.

**Phoenix** The agents seem not to deal well with the phoenix. They seldom use it in a 'smart' way, and it seems that agents with the phoenix make bad decisions in simple situations. This is probably because the Phoenix drastically increases the amount of possible actions for the agent and thus it can't search deeply enough and the actions become more random.

# Summary

In this project I implemented a framework to play Tichu and explored the effectiveness of various Monte Carlo Tree Search methods and enhancements. I demonstrate that MCTS handles the many difficulties of Tichu reasonable well but also that more improvements can be made. Based on the experiments, the best default MCTS agent uses the single determinization strategy and a random default policy. The EPIC enhancement as well as the Movegroups couldn't improve over the default agent. However, I did not yet explored all aspects of the Tichu game and some experiments would need to be analyzed more in detail. Doing this, it should be possible to create an agent that can play on human level.

## 6.1 Future work

I'd like to continue this project. The first thing to do however, is to port the entire framework to Java or C++. A big problem was, that playing tournaments with MCTS took a lot of time. Even when using as few as 100 iterations, a single game took several hours to complete. A compiled language may improve the speed drastically and with it the possibility to do more iterations.

The results for the DQN-rollout strategy were rather disappointing. I's like to improve on that. Eventually it would be fun to evaluate my agents against other people's agents.

## 6.2 Lessons Learned

Towards the end of the project I still had many ideas and improvements in my mind but the time lacked to implement them all. It would have helped a lot to clearly define what I wanted to achieve in the scope of the project and then stick to that. Also, my time management could still be improved.

On the plus side, I learned a lot about MCTS methods and their applications. And since a big part of the project was spent implementing the framework I can say that I now know python quite well.

# Acknowledgments

I would like to thank Igor Kulev and Boi Faltings for their assistance. Igor always provided helpful advice and I really enjoyed working on this project.

# Author's Declaration

I declare that this project was composed by myself, that the work contained herein is my own except where explicitly stated otherwise.

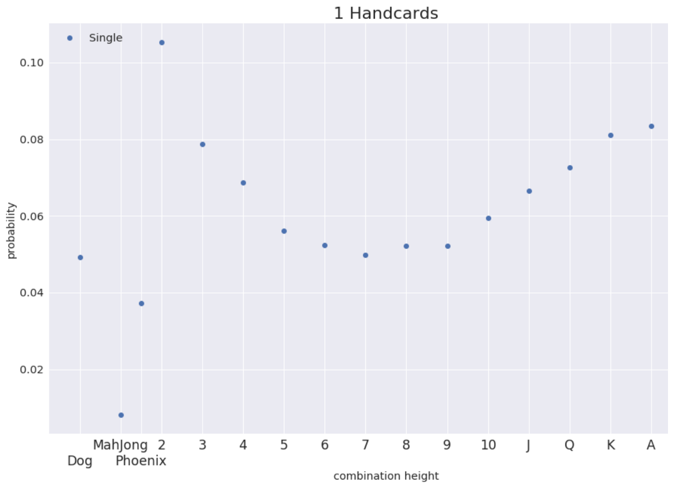
Friday 9 June 2017,

[illegible]

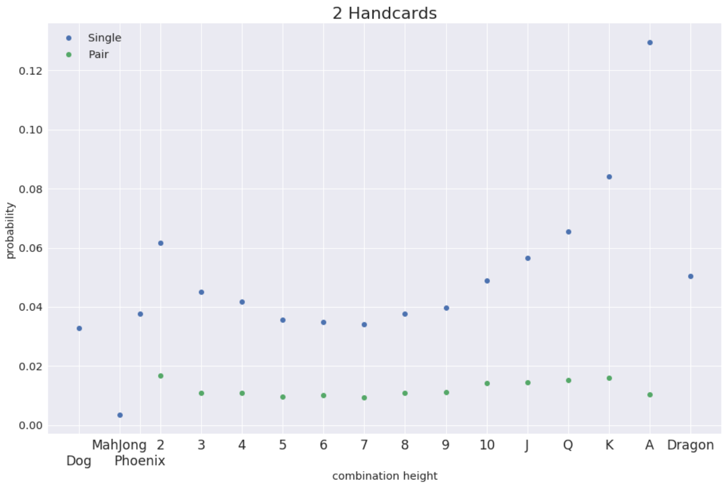
Lukas Pestalozzi

# Appendix

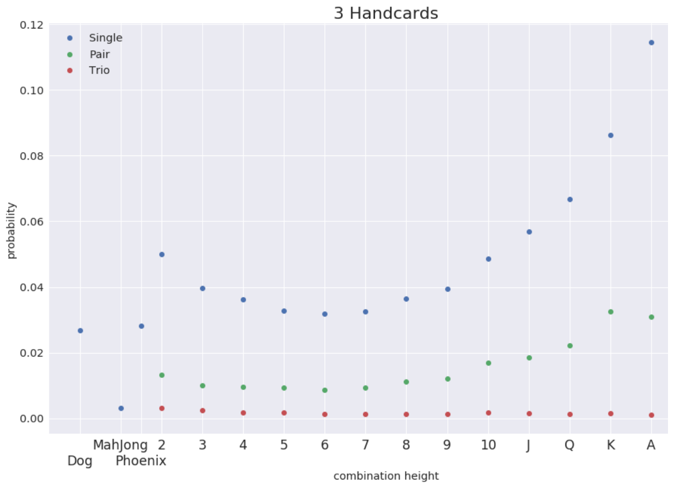
## Complete Handcard data plots



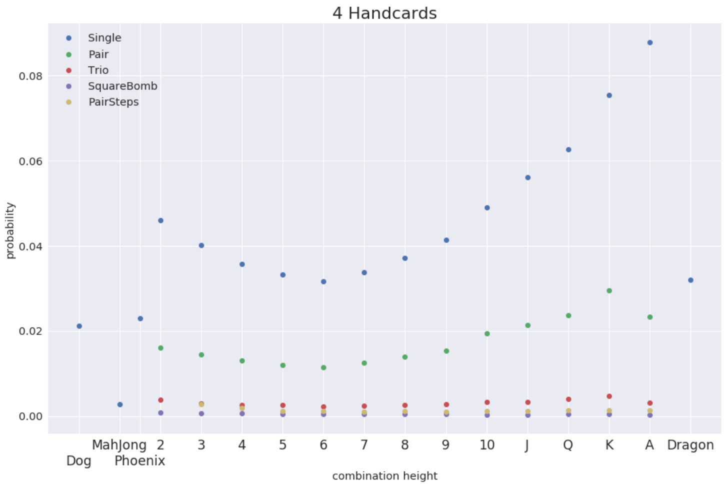
(a) Handcards of length 1



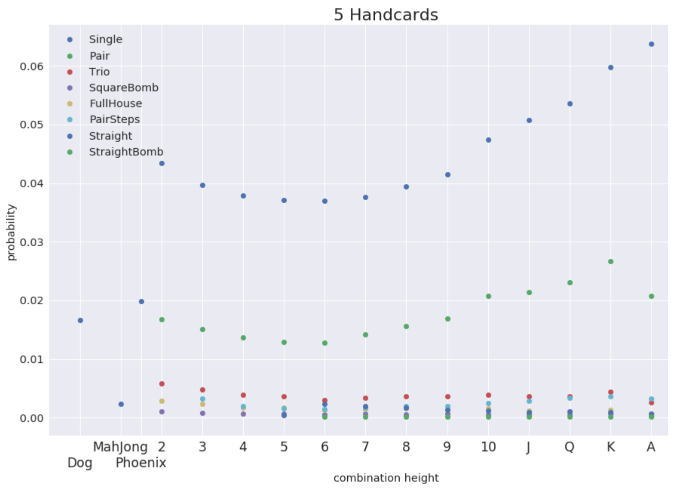
(b) Handcards of length 2



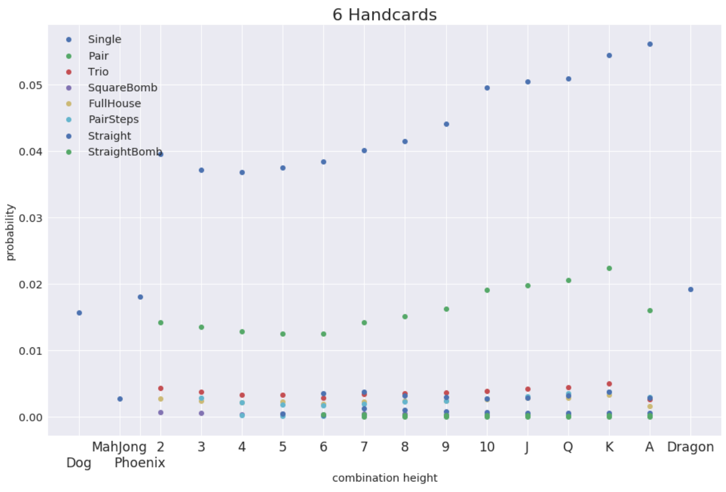
(c) Handcards of length 3



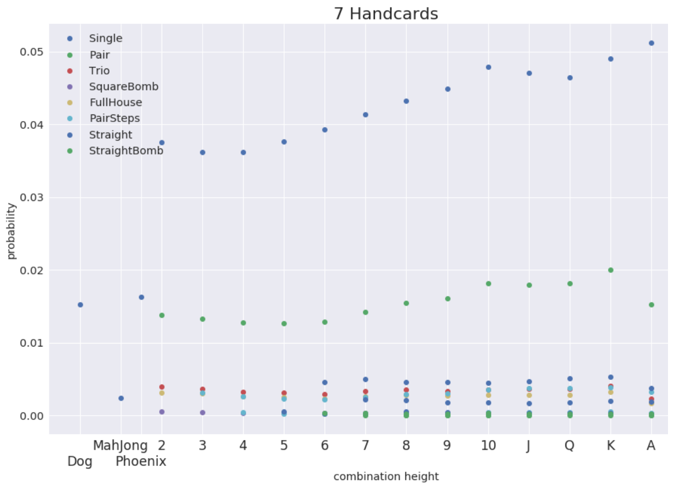
(d) Handcards of length 4



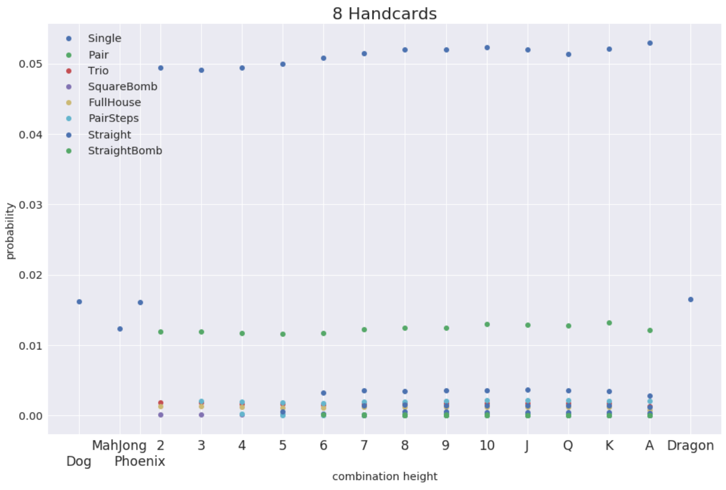
(e) Handcards of length 5



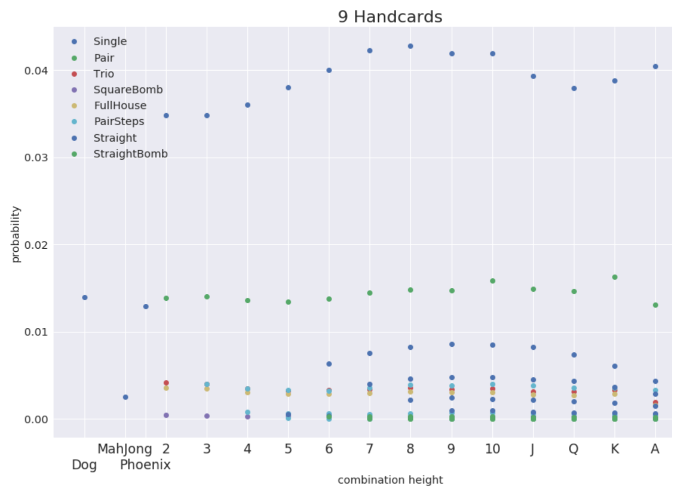
(f) Handcards of length 6



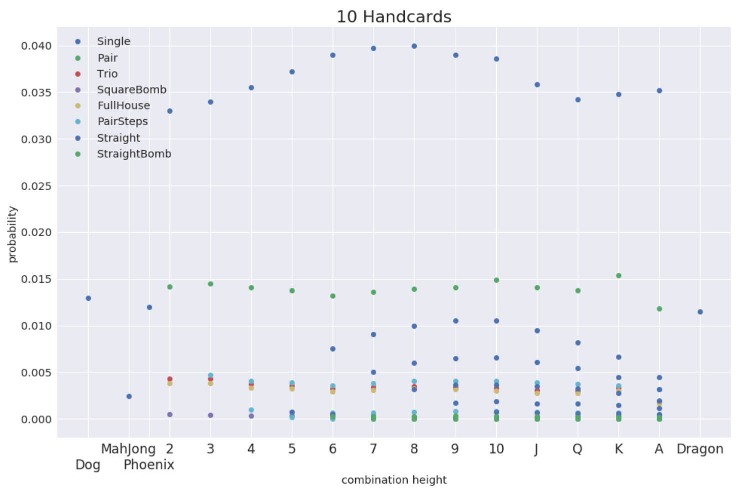
(g) Handcards of length 7



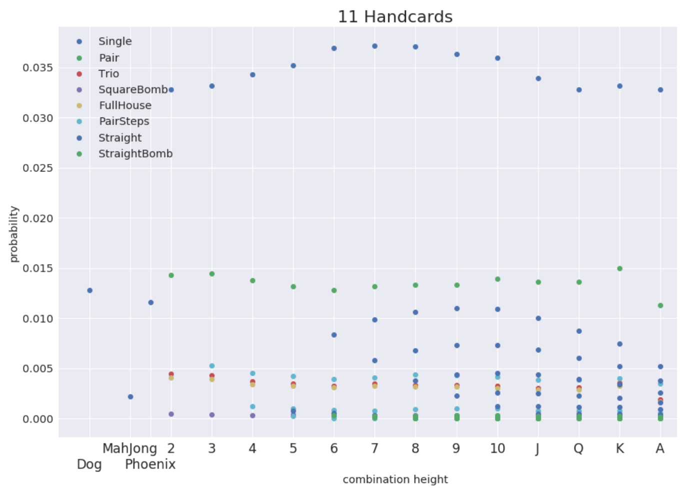
(h) Handcards of length 8



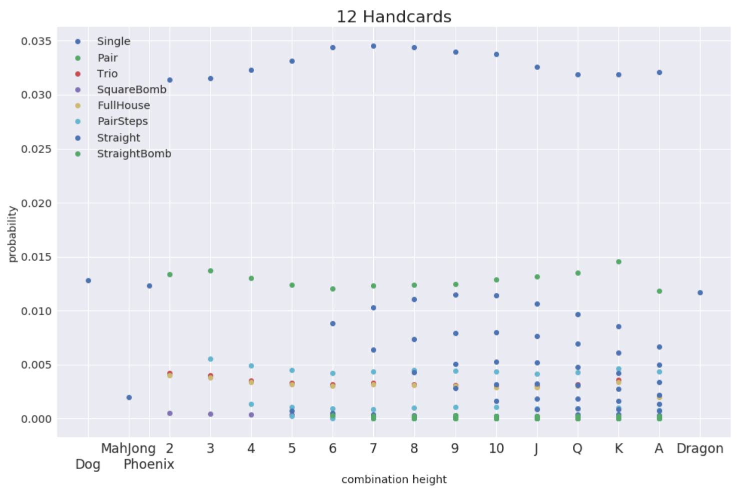
(i) Handcards of length 9



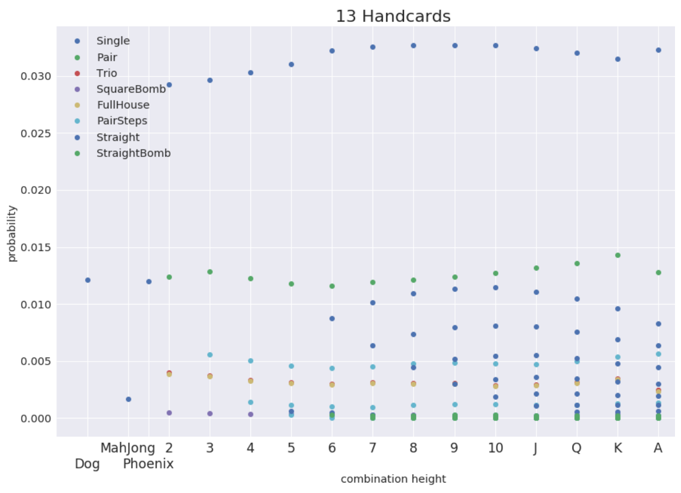
(j) Handcards of length 10



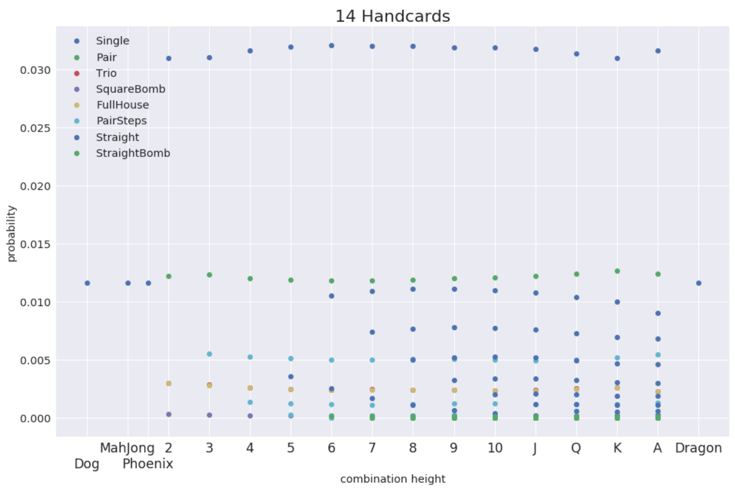
(k) Handcards of length 11



(l) Handcards of length 12



(m) Handcards of length 13



(n) Handcards of length 14

# Bibliography

[1] Fata Morgana: *http://www.fatamorgana.ch/tichu/tichu.asp*

[2] Tichu Rules and strategies: *http://scv.bu.edu/∼aarondf/Games/Tichu/*

[3] Jeffrey Long, Nathan R. Sturtevant, Michael Buro, Timothy Furtak, **Understanding the Success of Perfect Information Monte Carlo Sampling in Game Tree Search**, Department of Computing Science, University of Alberta Edmonton, Alberta, Canada in Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence, 2010

[4] David Pfander, **Eine künstliche Intelligenz für das Kartenspiel Tichu**, Universität Stuttgart, 2013.

[5] L. Kocsis and C. Szepesvári, **Bandit based Monte-Carlo Planning** in Euro. Conf. Mach. Learn. Berlin, Germany: Springer, 2006, pp. 282–293.

[6] I. Frank and D. Basin, **Search in games with incomplete information: A case study using Bridge card play** Artif. Intell., vol. 100, no. 1–2, pp. 87–123, 1998.

[7] Peter I. Cowling, Edward J. Powley, Daniel Whitehouse, **Information Set Monte Carlo Tree Search**, IEEE TRANSACTIONS ON COMPUTATIONAL INTELLIGENCE AND AI IN GAMES, VOL. 4, NO. 2, JUNE 2012

[8] Diego Perez, Spyridon Samotharakis, Simon M.Lucas, **Rolling Horizon Evolution versus Tree Search for Navigation in Single-Player Real-Time Games**, Juli 2013

[9] Stuart Russel, Peter Norvig **Artificial Intelligence, a modern approach**, third edition, 2014.

[10] Cameron Browne, Edward Powley, Daniel Whitehouse, Simon 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, VOL. 4, NO. 1, MARCH 2012

[11] Daniel Whitehouse, **Monte Carlo Tree Search for games with Hidden Information and Uncertainty**, University of York, August 2014

[12] Benjamin E. Childs, James H. Brodeur, **Transpositions and Move Groups in Monte Carlo Tree Search**,

[13] Gabriel Van Eyck, Martin Müller, **Revisiting Move Groups in Monte Carlo Tree Search**, University of Alberta, Edmonton, Canada, January 2012

[14] D. Robilliard, C. Fonlupt, and F. Teytaud **Monte-Carlo Tree Search for the Game of "7 Wonders"**, LISIC, ULCO, Univ Lille–Nord de France, FRANCE

[15] Tichu game-logs provided by *http://www.brettspielwelt.de/*: *http://log.tichumania.de*

[16] Silver and Huang et al, **Mastering the game of Go with deep neural networks and tree search**, Google DeepMind, nature vol 529, p484+, January 2016

[17] Github page of keras-rl python package: **https://github.com/matthiasplappert/keras-rl**