hochschule mannheim

Applying Monte Carlo Search and Monte Carlo Tree Search on Embedded Systems to Play Connect Four with a Robotic Arm

Moritz Duarte Pinheiro-Torres Vogt

Bachelor Thesis

for the acquisition of the academic degree Bachelor of Science (B.Sc.)

Course of Studies: Mechatronics

Department of Computer Science

University of Applied Sciences Mannheim

29.02.2020

Supervised by

Prof. Dr. Jörn Fischer, University of Applied Sciences Mannheim

Prof. Dr. Thomas Ihme, University of Applied Sciences Mannheim

Statement of authorship

I, Moritz Duarte Pinheiro-Torres Vogt, hereby declare that I am the sole author of this thesis and that I have not used any sources other than those listed in the bibliography and identified as references.

I agree that this work will be published and thus electronically saved, converted to other formats and provided on the servers of the Hochschule Mannheim for public accessibility, whereby it may be distributed over the internet.

Mannheim, 29.02.20

....
Moritz Duarte Pinheiro-Torres Vogt

Preface

The presented bachelor thesis represents my acquisition of the academic degree Bachelor of Science for the course of studies Mechatronics. It is accomplished in the Department of Computer Science at the University of Applied Sciences Mannheim.

At this point, I would like to express my sincere gratitude to my supervisors Prof. Dr. Jörn Fischer. Thank you for your motivation for this work, your enthusiastic effort to realize the 3D printings, your continuous support for any problem and the inspiring discussions - I have learned a lot.
I would also like to thank Prof. Dr. Thomas Ihme for supporting me with ingenious suggestions and well-considered advice.
Gratitude is also owed to my dear friend Frederic. Thank you for your aid in learning how to write scientific works and for your extensive effort in English language.
Lastly, I would like to thank you, Adina, not only for your aid in English language, but also for your mental care. You give me energy when mine is low and support me immensely thereby. Also your culinary dedication relieves me a lot.
Thank you.

Abstract

Monte Carlo based methods have brought a remarkable improvement in performance of artificial intelligence in the realm of games in recent years, whereby human champions could be beaten in several board games of high complexity. In this work, two Monte Carlo based approaches, the Monte Carlo Search and the Monte Carlo Tree Search, are applied to the game of *Connect Four*. In order to test and compare these two approaches with regard to performance, games are simulated where they play against each other with equal limitations. The better approach is then further placed against human opponents using on the one hand a website (https://carloconnect.com/) to evaluate the algorithm's strength and on the other hand a 3D printed game board and robotic arm to play with physical components. All tests are conducted on a Raspberry Pi 3 Model B to investigate the approaches' performance in an environment with limited available computing power and memory. The experiments demonstrate that under these conditions the Monte Carlo Tree Search clearly outperforms the Monte Carlo Search for every tested computing time limit. Hence, the approach that is based on the Monte Carlo Tree Search is the better solution for this application and is called CarloConnect. Against human opponents, CarloConnect proves itself in winning most of the games, thus playing at human level even while having only low computing resources.

Zusammenfassung

In den letzten Jahren hat sich die Effizienz von künstlichen Intelligenzen deutlich verbessert. Grund dafür ist unter anderem die Anwendung von Monte Carlo Methoden, welche erlauben, Weltmeister in verschiedenen hochkomplexen Spielen zu schlagen. In dieser Arbeit werden zwei Monte Carlo basierte Algorithmen, der Monte Carlo Search und der Monte Carlo Tree Search, für das Spiel Vier Gewinnt angewandt. Diese werden miteinander verglichen, indem Spiele mittels der beiden Algorithmen simuliert werden. Die Algorithmen spielen dafür unter den gleichen Voraussetzungen. Gegen den stärkeren Algorithmus treten dann menschliche Gegner, Studenten, an. Zum einen konnten Testspiele auf einer Website (https://carloconnect.com/) durchgeführt werden, die das Spielfeld darstellt und der Gegner mittels der stärkeren KI agiert; zum anderen wurde ein Spielfeld modelliert, ein Roboterarm ausgewählt und mit einem 3D Drucker beides hergestellt, wobei der Roboterarm die Züge des Algorithmus ausführt. Alle Tests werden auf einem Raspberry Pi 3 Model B ausgeführt, um bei begrenzten Kapazitäten die Algorithmen auf ihre Stärken untersuchen zu können. Die Ergebnisse der Tests zeigen, dass der Monte Carlo Tree Search dem Monte Carlo Search für jede getestete Zeitbegrenzung bei der Suche nach dem besten Zug stark überlegen ist. Folglich wird der Monte Carlo Tree Search basierte Algorithmus CarloConnect genannt. Gegen menschliche Gegner erweist sich CarloConnect als ein gleichrangiger Gegner, der nicht nur auf menschlichem Niveau spielt, sondern gar die meisten Partien gewinnt, trotz der geringen zur Verfügung stehenden Rechenkapazitäten.

Contents

Chapter 1

Introduction

In the last decades, artificial intelligence (AI) has become increasingly important in a multitude of domains. Tasks, that are difficult or impossible to program by hand, such as face recognition [1] or stock market prediction [2] can be tackled utilizing machine learning. In the domain of games, too, AI has been applied successfully in recent years to enhance the performance of virtual players significantly – especially in games with a high degree of complexity.

Implementing such a search algorithm for games with perfect information, usually requires an evaluation function based on heuristic knowledge, which determines the value of a game state. In the context of the game theory, perfect information means that each player is always perfectly informed about all the events that have previously occurred, including the initial conditions of all players [3]. The more complex the game is, meaning the more possible actions there are in a state, the more difficult and time-consuming the defining of an evaluation function becomes. Depending on the complexity of the game, there may be a very large number of possible game states – specially in *Go* where about 10^{172} possible sequences of moves exist [4]. A search in a search tree that contains all these states is therefore infeasible, due to enormous requirements regarding computing power and memory [5].

There are several Monte Carlo based approaches which neither require an evaluation function, making them applicable to any game of finite length, nor need to carry out exhaustive searches through a complete search tree of states. These approaches develop game-specific knowledge that does not need to be complete in order to evaluate the states and choose the best action [6]. These search algorithms are applied to many games like *Poker* [7] or lately to *Go*, where the search algorithm *AlphaGo* defeated the European *Go* champion by 5 games to 0 [8].

In this study, two Monte Carlo based search algorithms, the Monte Carlo Search (MCS) [9] and the Monte Carlo Tree Search (MCTS) [10], are applied to the game of *Connect Four* to investigate their performance in games with a comparable degree of complexity and the impact of different limitation factors for these approaches. The better solution for this application is called CarloConnect. While Google is utilizing computers with extremely high computing power, it is becoming more and more relevant to implement AIs on embedded systems with limited computing power and memory, which is why all experiments of this work are run on a single-board computer. For further evaluation and applicability, CarloConnect is then tested by carrying out games against human opponents. Therefore, a website is hosted on the single-board computer where the game can be played online. Furthermore, for playing with physical components, a game board for the game of *Connect Four* and a robotic arm, which is utilized to perform the moves from CarloConnect, are constructed and 3D printed. With different switches the moves of the human opponent are registered, the game can be restarted and different levels of difficulty can be set.

Chapter 2

Theoretical fundamentals

In this chapter theoretical fundamentals are presented to provide all information that are necessary to understand the approaches carried out and analysed in subsequent chapters.

2.1 The game of Connect Four

Connect Four is a game with perfect information in which two players can drop discs into a vertical board with 42 squares distributed in 6 rows and 7 columns. Each square has a window, so that discs are visible from both sides of the game board. The players make their moves in turn, where the configuration of the board at a given turn is referred to as the state of the game, which allows for a specific set of valid actions for a player. If a disc is inserted, it falls straight down to the lowest unoccupied square within the selected column. The first player to get four in a row wins, whereby orientation can be either horizontal, vertical or diagonal.

The game is solved by *Allis* [11] showing that in case of perfect playing of both players, the first player always wins if he chooses the middle column as his first move and loses if he selects the four outer most columns. The selection of the other two columns leads to a draw. Furthermore, *Edelkamp and Kissmann* [12] calculated the total amount of possible states of over $4.5 \cdot 10^{12}$ which would require nearly 43.8 terabytes of memory if the value of a position is stored in 2 bits.

Figure 2.1 illustrates the board of the game and a possible configuration of the board.

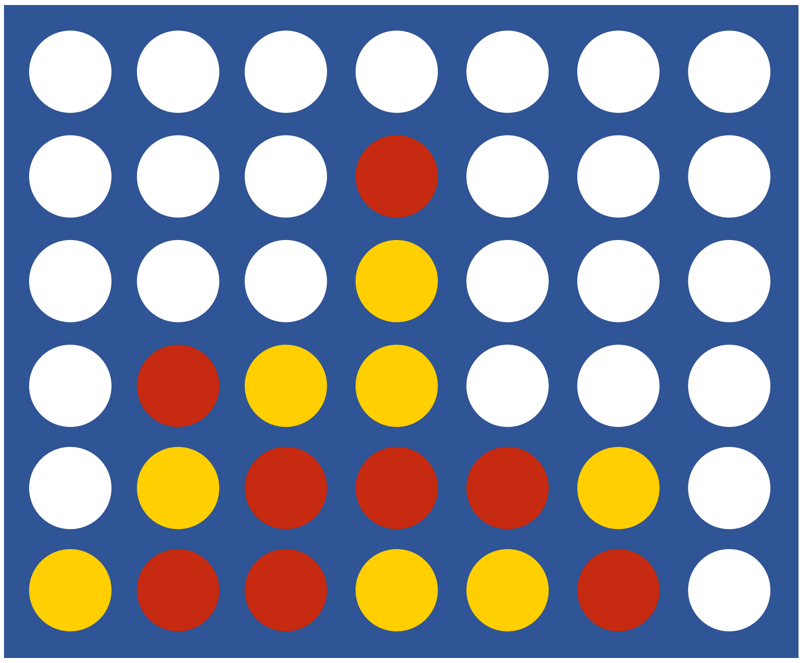

Figure 2.1 | The board of the game *Connect Four*. In this state of the game yellow wins due to a diagonal sequence of four discs.

2.2 Monte Carlo based search algorithms

A search algorithm is an important component to solve a search problem in AI, or more generally, in computer science and engineering [13]. An example for such a search problem is the *multi-armed bandit problem*, where originally a set of one-armed bandit machines must be played successively in an order that maximizes the gain [14]. Therefore, heuristic techniques are commonly used which are utilizing limited knowledge to receive probable solutions within short time of computation [15]. Many heuristic approaches that are used these days, for example by Google [8], rely on the Monte Carlo methods.

The idea of Monte Carlo methods is to repeat an experiment many times to receive a large amount of results, which are statistical in nature. Due to the law of large numbers, the average of all results can be assumed to stabilize around the expected value for a large amount of experiments. With these experiments, a temporary heuristic game-specific knowledge for a given state can be obtained. “Sometimes, in spite of the random character of the *answer*, it is the most accurate *answer* that can be obtained for a given investment of computer time.” [16]. In the context of this work, the *answer* is an approximation for a best action to undertake for a given player in a given state of the game, resulting from the computational outcome.

Two general Monte Carlo based approaches are presented in the following.

2.2.1 Monte Carlo Search

The Monte Carlo Search (MCS) algorithm, as first described by *Tesauro and Galperin* [9], is a straightforward technique where for each of the available actions of a current state, a large number of simulations is run, limited either by a specified amount of simulations or by computing time. In each simulation, a complete play through is performed. In this work, the term play through shall imply that from a selected action the game is played virtually by selecting actions for each player at random until a final game state is reached and the winning player is returned.

In two-player zero-sum games, such as *Connect Four*, each player is assigned a win count [17]. If the winner of a simulation is the same as the player for whom the best action is searched for, the player's win count is increased by 1. Analogously, in the case of a loss, the win count is decreased by 1. When a draw occurs, the win count is not modified. As soon as all simulations are concluded or the time limit is reached, a win rate is calculated using the win count $wins_{action}$ and the amount of simulations num_{sim}, as in *Equation 2.1.*

$$\text{winRate} = \frac{wins_{action}}{num_{sim}} \tag{2.1}$$

This procedure is applied to all the available actions of a current game state, eventually returning the action with the best win rate.

2.2.2 Monte Carlo Tree Search

As the MCS simulations are random and thus unguided, the MCS has a lack of tactical insight. In order to compensate for this disadvantage, *Coulum* [10] combined the Monte Carlo Search with a search tree, thereby creating the so-called Monte Carlo Tree Search (MCTS) algorithm. This algorithm iteratively builds a search tree of possible future game states, node by node, where each node represents a state at a given player's turn and its estimated value. When a limiting factor, i.e. time or memory, is reached, the search iterations stop and the most promising action is returned. The larger the tree grows the higher the probability becomes to increase the value of the returned move, thus getting closer to the best possible move. The way the tree is built depends on how nodes in the tree are evaluated and selected. It should be noted that each node is an instance that requires memory, which can be limited by the embedded system that is utilized for the computing.

According to *Chaslot et al.*, [18] an individual search iteration can be split into four phases:

1. *Selection*: Starting from the root node, the tree is traversed by selecting child nodes until a node is reached that has unexpanded child nodes left and is thus expandable. Therefore, a selection function is needed that in addition has to control the balance between exploitation and exploration – where exploitation means to further utilize the best nodes to reinforce their statistical significance, and exploration means to weight nodes higher if they are visited less – to evaluate undetected moves. This problem of balance is similar to the *multi-armed bandit problem.* A general approach for this issue is the Upper Confidence Bounds 1 (UCB1) policy [19]. For application with search trees, *Kocsis and Szepesvári* [20] proposed the UCB1 applied to trees (UCT) strategy (see *Equation 2.2*),

   $$\mathrm{UCT} = \frac{w_i}{v_i} + c \cdot \sqrt{\frac{\ln(v_r)}{v_i}} \tag{2.2}$$

   where w_i is the win count of node i, v_i is the visit count of node i, v_r is the root node's visit count that correlates with the total amount of simulations and c is a constant that favors exploitation if low and exploration if high.

2. *Expansion*: One of the unexpanded child nodes is randomly selected and added as a new node to the tree. That way, the tree is expanded by one node with each iteration. Optionally, the tree can be expanded by multiple nodes, i.e. by all available actions, but it is more memory-efficient to create just one node per iteration.

3. *Simulation*: From the new node's state a play through is performed that returns a result, i.e. the victorious player of the simulated scenario.

4. *Backpropagation*: All nodes that are traversed by the selection are updated. Thereby, their visit count is increased by 1. The result of the simulation is subsequently backpropagated depending on the game type and its evaluation function. In the case of a two-player zero-sum game, similarly to the evaluation strategy presented in MCS, those nodes that represent the player that wins according to the simulation have their win count increased by 1. Analogously, the other nodes represent the player that loses and have their win count decreased by 1. In case of a draw, the win counts are not modified.

Figure 2.2 illustrates this iteration process where the players, which are represented by the levels in the tree, are identified by different colors – here shown for a two-player game.

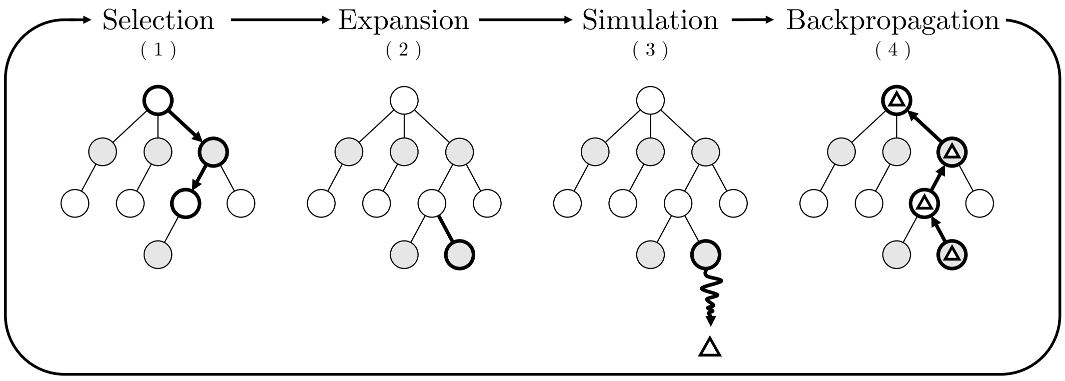

Figure 2.2 | The four phases of an iteration of the Monte Carlo Tree Search (MCTS) algorithm, colored for two-player games where each colour represents a player. The MCTS is an iterating search algorithm that builds a search tree from scratch rooted at the current state to find a best action. An iteration can be split into four phases. In the first phase the tree is traversed using a tree policy until a node with unexpanded child nodes is reached (1). Thereupon, one of the unexpanded child nodes is created (2) and a game is simulated by selecting actions for each player at random until a final game state is reached, returning a result Δ (3). This result is then used to update all nodes that are traversed during the selection phase including the new expanded node (4). Figure adapted from [6].

When the limiting factor is reached and consequently the iterations are stopped, the most promising child node of the root node, representing an action, is returned. *Coulum* [10] presents four operations for this final action selection, which are extended by *Chaslot et al.* [21] with an additional one:

1. *Mean child*: Select the child node with the best win rate.
2. *Max child*: Select the child node with the highest win count.
3. *Robust Max child*: Select the child node with the highest visit count.
4. *Mix child*: Select the child node in dependency of both the visit counts and the win rate. In case the child node with the most visits is not the same as the child node with the best win rate, the search is continued until there is a child node that fulfills both conditions.
5. *Secure child*: Select the child node that maximizes a lower confidence bound.

Experiments from *Coulum* [10] evince that the best choice to make is to utilize the *Mix child* since it produces the smallest mean error of those strategies. Furthermore, the results show that the *Mean child* under-estimates the node value, whereas the *Max child* over-estimates it. With the *Robust Max child*, in turn, results with similarly small mean error can be achieved. The consideration of the fact that the child node with the most visits mostly is the one with the best win rate, too, confirms this similarity. *Chaslot et al.* [21] did not measure any significant difference in performance between those final selection methods and eventually used the *Robust Max Child* for his *Go* program *MANGO*.

2.3 Computing platform

Most of the electronic projects nowadays require a certain digital computer at its heart. There are many systems that can carry out the computation [22][23]. A simple solution is a microcontroller. A microcontroller, or microcontroller unit (MCU), is a small controller on an integrated circuit (IC) chip, that can be instructed to carry out sequences of arithmetic or logical operations. It can be combined with other components of a computer, for example the central processing unit (CPU) or memory. If all components are combined on a single IC, the system is called a system on a chip (SoC). Alternatively, the system can be integrated on a single printed circuit board (PCB), then it is called a system on module (SOM) or computer on module (COM). A SOM/COM may additionally integrate digital and analog functions. If a SOM/COM is extended by the standard connectors for any input/output peripherals to be attached directly to the board, it is called a single board computer (SBC). An overview of these computing platforms is demonstrated in *Figure 2.3*.

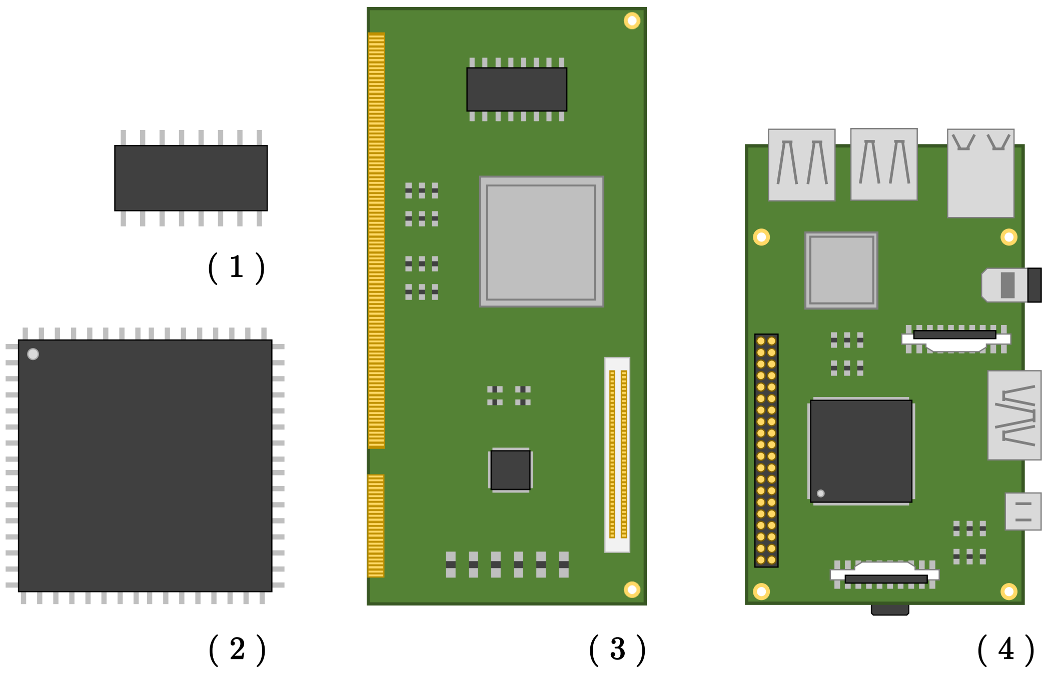

Figure 2.3 | Overview of different computing platforms. Four different computing platforms that can be used for different purposes are presented. (1) a microcontroller unit (MCU), (2) a system on a chip (SoC), (3) a system on a module (SOM), or computer on a module (COM), and (4) a single board computer (SBC).

Depending on the purpose the computing platform is required for, it may be called an embedded system [24]. Embedded systems are computing platforms of all kind built into a device developed for a particular purpose as control, automation, communication or many other. The connectivity of the embedded systems is often specially designed therefore.

2.4 Electronic components

In the field of electronics, there are many components beyond those that regulate the currency and voltage. For autonomously operating robots the influence of the environment has to be digitized [25]. Therefore, sensors can be used to transform non-electrical signals, such as light, temperature or others, to electrical signals. In this sense, electrical switches can be seen as sensors, too. In order to react to the environment, a robot has to do the transformation the other way around – converting the signal's energy into mechanical motion. Components, that carry out the mechanical motions, are called actuators, whereby the motion can be e.g. pneumatic, hydraulic or electromechanic (from motors).

Hereinafter, both sensors and actuators that are chosen for completion of this work are described more precisely concerning their functionality.

2.4.1 Switch

Electrical switches are utilized to connect or disconnect the conducting path of an electrical circuit [25]. There are many different types of switches such as slide switches, pushbutton switches, toggle switches, rocker switches, micro switches or many more. The purpose of all of them is to vary a signal that for instance turns on a device. Basically, when the switch is pressed, a metal contact is opening an electrical circuit and closing another one causing that an electrical signal is transferred or interrupted.

Three types of switches are presented in the following.

Pushbutton switch

Pushbutton switches exist in many sizes, from tiny (on-board restart button of SOMs) to the size of a hand (emergency buttons in buildings), or even bigger. A pushbutton switch contains at least two contacts, which close or open when the button is pressed [25]. When releasing, the button returns to its original position, mostly due to a spring. Pushbutton switches are often utilized for one-time signals, for example to (re)start systems or to activate an emergency state. This can be attributed to the primitive usability – the button can be pressed only in one way and one direction and returns thereupon to its original position.

Figure 2.4 shows a pushbutton switch with a scheme of its internal construction.

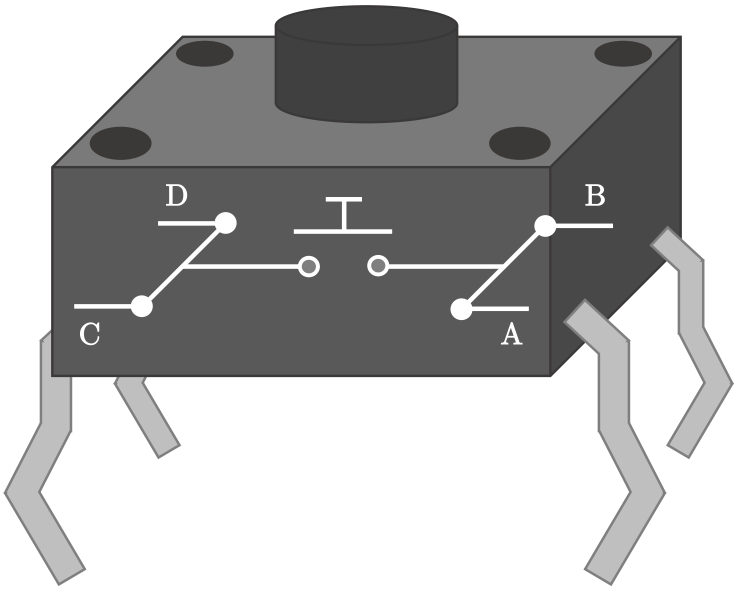

Figure 2.4 | A pushbutton switch with a scheme of its internal construction. Pushbutton switches are mostly utilized to send a signal that activates a system of any kind. The pins, or legs, (A) and (B) are always connected, so are the pins (C) and (D). When the pushbutton is pressed, a circuit that connects both sides is closed and by that all of the four pins are connected. Thereby, a signal can be transferred from the pins (A) or (B) to the pins (C) and (D) – or the other way around.

Rocker switch

Rocker switches are usually not very sensitive, which is why they are used to set steady states. There are different kinds of rocker switches – some provide two different selectable states, some three and others even more. Rocker switches are designed for push-insertion into a sized rectangular hole in the panel of the switch [25]. They are commonly utilized as light switches or power switches, as once they are pushed in one state, they require a push in the opposite direction to return in the previous state. Hence, they are simple to use and secure at the same time.

Figure 2.5 shows a rocker switch with a scheme of its internal construction and a typical pin-out.

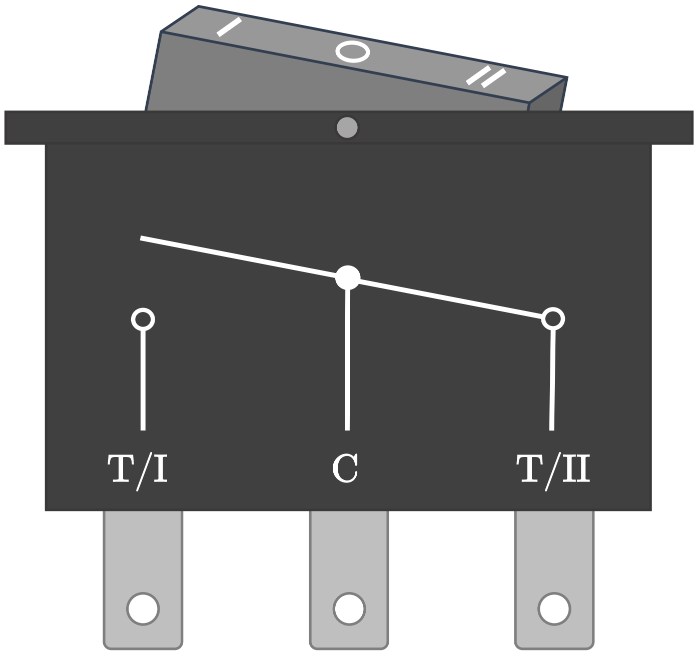

Figure 2.5 | A rocker switch and its sketched schematics. Rocker switches are mostly utilized to turn devices on or off. The presented rocker switch provides two terminals (T/I and T/II) that can either be selected by the common channel (C). Besides those states of the two terminal, there is a third so-called "none" state, where no signal is transferred.

Micro Switch

Micro switches have an advantage as they are small in size, like indicated by the name. The small size of them is very valuable in the field of robotics where limited space and the need of many sensors are common conditions. Micro switches are very sensitive compared to rocker switches and thus used to recognize small physical forces. Usually, they have three pins, a common terminal (C) and two individual channels, a normally closed (N/C) channel and a normally opened (N/O) channel [26].

Figure 2.6 shows a micro switch with a scheme of its internal construction and a typical pin-out.

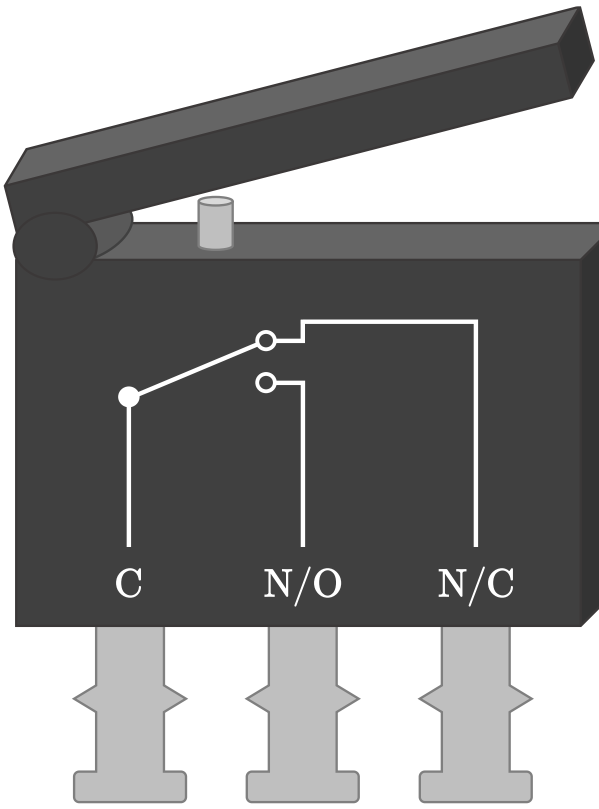

Figure 2.6 | A micro switch with a scheme of its internal construction. Micro switches are utilized for little forces. The common terminal (C) can either connect one of the two other channels, the normally closed (N/C) or the normally opened (N/O) one.

2.4.2 Servo motor

The radio control servo motor (RC servo motor), short servo motor, is a combination of an electric motor, which is mostly direct current (DC), a reduction gearing, control electronics and a potentiometer for position determination [26]. All these components are suited to fit in a compact plastic case. Due to the small size and the high precision, servo motors are much used in the field of robotics and model making.

Servo motors are not designed for continuous rotations as conventional motors, but rather for precise control of angular positions. Usually, positions in the range of 0 and about 180 ° can be set. Some models supply a range that is slightly wider, but it is always noticeably smaller than 360 °. Generally, servo motors have three wires: one for the power supply, one for the ground and one for a pulse width modulation (PWM) signal which controls the rotation of the servo motor [25].

Figure 2.7 shows a servo motor with its internal construction and a typical connectivity.

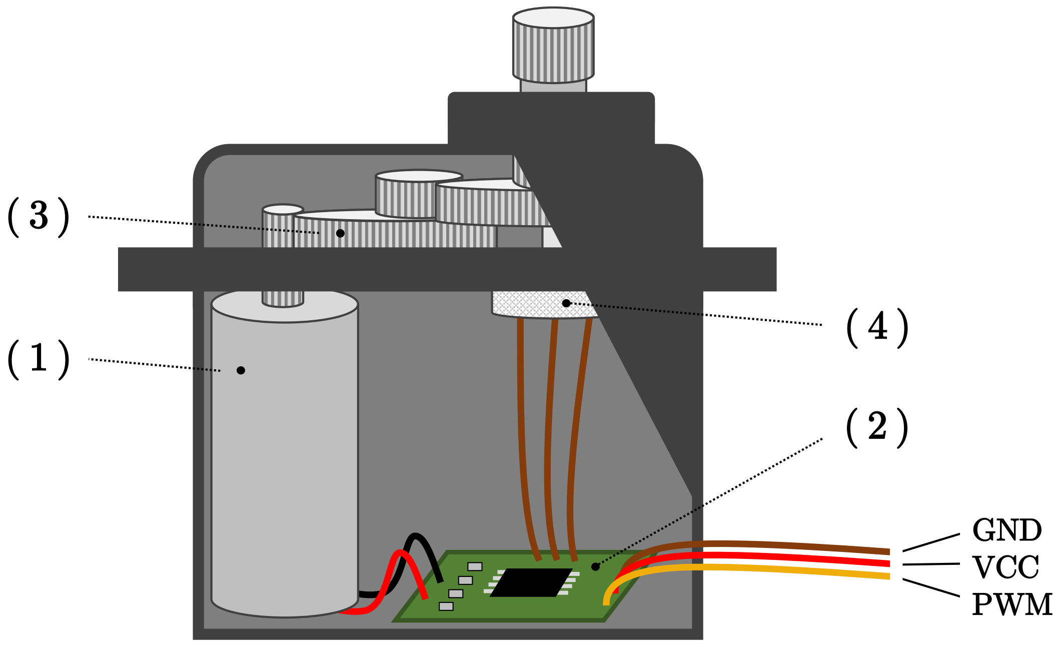

Figure 2.7 | **A servo motor with its internal construction.** Servo motors are designed for precise control of angular positions. They are composed of an electric motor (1), control electronics (2), a reduction gearing (3) and a potentiometer for position determination (4). A servo motor needs three external signals to operate, a power supply (VCC), a ground (GND) and a pulse width modulation (PWM) signal to set the desired angular positions.

A PWM signal is a square pulse wave signal [27]. The width of the pulses can be modulated to send specific signals or to lower the voltage of the output averagely. It is determined by a frequency and a duty cycle which defines the relation as a percentage between the pulse width and the period, that depends on the set frequency.

In *Figure 2.8* a square pulse wave on a PWM signal is demonstrated.

Servo motors respond to pulses with the width of 0.5 to 2.5 ms, independently of the value of the duty cycle. The range of the pulses can vary with different servo motor models, but it is mostly within this spectrum [26]. With the width of the pulse of the sent PWM signal the rotation and thus the desired angular position can be set.

For implementation, several values may be necessary in order to configure the PWM signal properly [28]. Some of those values, the frequency f_{servo}, the servo motor operates with, the clock f_{osc} of the oscillator, that is utilized for the signal, and its minimum pulse width t_{min} are given in the datasheet or documentation of the used hardware. It should be noted, that the clock f_{osc} can either be software or hardware based. This means that for the clock either the physical oscillator of the system or the clock of the CPU is utilized to

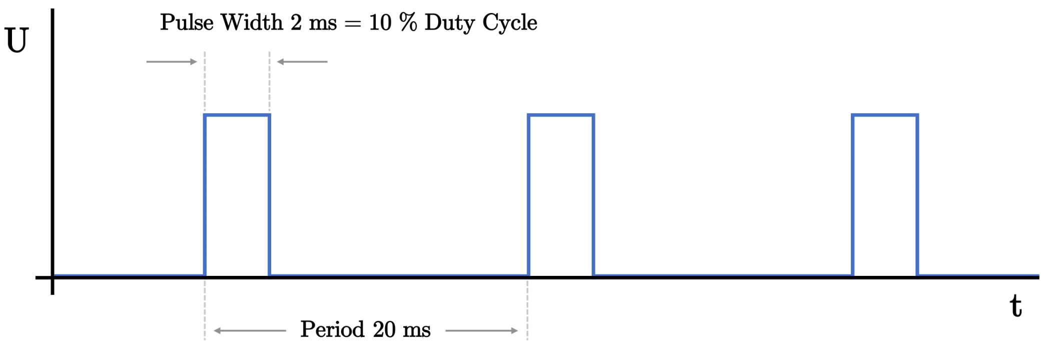

Figure 2.8 | A pulse width modulation (PWM) signal. Servo motors are controlled with PWM signals to set certain angular positions. The width of a pulse can me modulated. Servo motors regularly respond to pulse width of 0.5 to 2.5 ms – in this image a width of 2 ms is shown. The relation between the pulse width and the period is defined as duty cycle.

derive the frequency of a signal. However, both clock rates can be found in the documentation. The remaining values need to be calculated. First the period T of the cycle has to be determined. It is the reciprocal of the frequency f_{servo} (see *Equation 2.3*).

$$T = \frac{1}{f_{servo}} \tag{2.3}$$

The PWM range PWM_R, which is the period of the PWM signal expressed in the scale of the minimum pulse width of the clock, can be calculated with the period T and the minimum pulse width t_{min} (see *Equation 2.4*).

$$PWM_R = \frac{T}{t_{min}} \tag{2.4}$$

The last variable that may be necessary is the clock divider PWM_{CD}. It configures the prescaler register, which divides the oscillator's frequency by the set clock divider. This way, the frequency can be decreased to any desired value. Together with all given and calculated values the clock divider PWM_{CD} can be calculated as in *Equation 2.5*.

$$PWM_{CD} = \frac{f_{osc}}{f_{servo} \cdot PWM_R} \tag{2.5}$$

There are libraries offering functions for configuration of the prescale register. Still, in many cases, the parameters of the calculations above must be passed as arguments.

2.5 Kinematic chains

In today's industry, robots are very powerful components to operate precisely a wide range of tasks [29]. A robotic arm is a chain of multiple bodies, called links, connected by joints, where each joint can move its outward neighbouring link with respect to its inward neighbour. One end of the chain is the base which is fixed. The other end is free to move in space and holds the tool or the end-effector [30]. Each joint of the chain can either be translational (a prismatic joint) or rotational (a revolute joint) and has one degree of freedom. In kinematics, these chains are called kinematic chains and can be described mathematically and used to determine positions or angles of particular links. Kinematics, in general, describes the relation and motion of points, links and coordinate systems [31]. The links' masses or the acting forces are not relevant.

With the so-called forward kinematics, the position of the end-effector in a three-dimensional space can be calculated using the joints' angles. It is possible to do a determination the other way around, too. In this case, with the inverse kinematics, from a given position of the end-effector the angles of the joints can be calculated. Both kinematics are provided in this section. In *Figure 2.9* the working directions of these two kinematics are demonstrated.

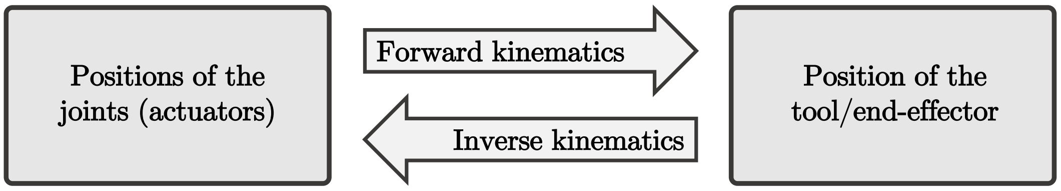
Positions of the joints (actuators)
Forward kinematics
Inverse kinematics
Position of the tool/end-effector

Figure 2.9 | Working directions of the forward and the inverse kinematics. With the forward kinematics the given positions of the joints of a kinematic chain are used to determine the position of the end-effector in three-dimensional space. The inverse kinematics is utilized for the opposite – in dependency of the position of the end-effector, the (possible) positions for each joint of the kinematic chain can be calculated.

2.5.1 Forward kinematics

The objective of forward kinematics is finding the position and orientation of a robotic arm's end-effector as a function of its joint angles. There are two ways for building forward kinematics – it can either be derived from its space with a geometric approach or determined using the Denavit-Hartenberg (DH) convention [29].

In the geometric approach, the investigated robotic arm or system, usually described in a 3D Cartesian coordinate system, is parsed in 2D Cartesian coordinate systems and then solved using trigonometry. For complex kinematic chains, this task can become very complex and thus may be difficult for humans to solve. The DH convention, on the contrary, simplifies the calculations by assigning four parameters, the DH parameters, to each link [32]. Thereby, the position of each link to its respective neighbours can be sufficiently described. In *Figure 2.10* a kinematic chain, consisting of three segments, and the DH parameters of one segment are presented.

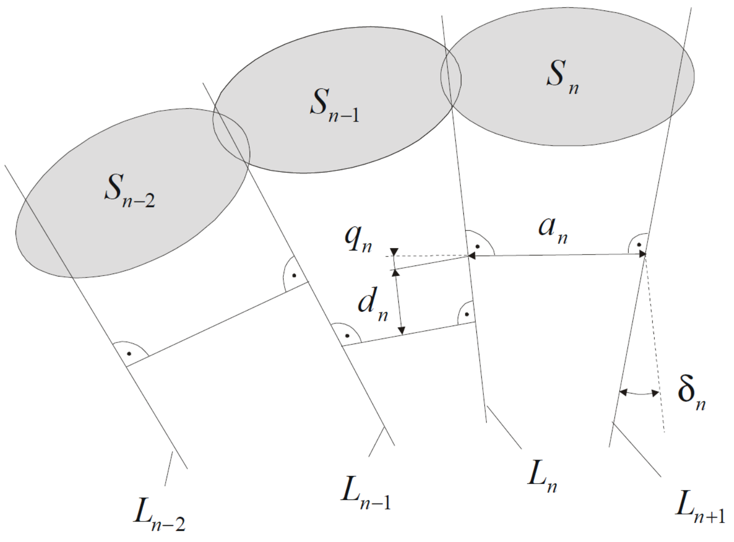

Figure 2.10 | Denavit-Hartenberg (DH) parameters for kinematic chains. With the DH parameters, kinematic chains can be described more easily in order to determine the position of the end-effector of a kinematic chain. The presented kinematic chain consists of three segments, S_{n-2}, S_{n-1} and S_n. They are connected by the axes L_{n-2} to L_{n+1}. The constellation of the segments allows the calculation of the parameters a_n, d_n, δ_n and q_n required for the determination. Figure taken from [33].

The four parameters of the link S_n are determined as follows:

1. Link length a_n is the the perpendicular distance between the axis L_n and the axis L_{n+1}.
2. Link offset d_n is the distance between the normal of the axis L_n and the axis L_{n+1} and the normal of the axis L_{n-1} and the axis L_n, measured along the axis L_n.
3. Link twist δ_n is the angle between the axis L_n and the axis L_{n+1} about the normal of S_n.

4. Joint angle q_n is the angle between the normal of the axis L_n and the axis L_{n+1} and the normal of the axis L_{n-1} and the axis L_n about the axis L_n.

Normally, three out of the four parameters of a link are constant. The fourth parameter, which is q_n for revolute joints (rotational) and a_n for prismatic joints (translational), varies.

Using the axis L_n and the normal a_n, a coordinate systems for the segment (joint) S_n can be built. Accordingly, a respective coordinate system can be built for each of the joints (see *Figure 2.11*).

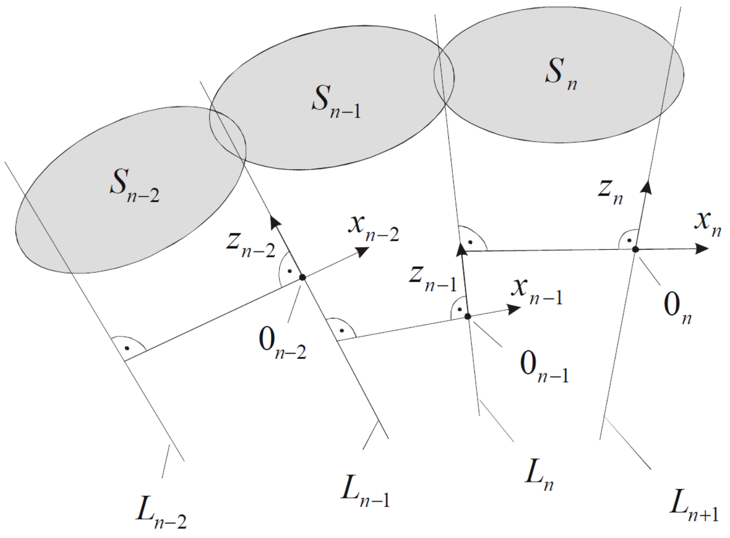

Figure 2.11 | Joint coordinate systems according to the Denavit-Hartenberg (DH) convention. For the DH convention the frame of each segment of a kinematic chain must be described in respective coordinate systems. These rely on the constellation of the segments of the chain, further presented in *Figure 2.10*. Figure taken from [33].

With the DH parameters and the according generalized joint coordinate systems, a homogeneous transformation matrix T can be created for every link [33]. This transformation matrix transfers a joint coordinate system $\{n-1\}$ to a joint coordinate system $\{n\}$ as in *Equation 2.6*.

$$ {}^{n-1}_{n}T = \text{Rot}(z_{n-1}, q_n) \cdot \text{Trans}(0,0,d_n) \cdot \text{Trans}(a_n,0,0) \cdot \text{Rot}(x_n, \delta_n) \tag{2.6} $$

The multiplication using the homogeneous rotation and translation matrix leads to *Equation 2.7* [29].

$$
{}^{n-1}_{n}T = \begin{bmatrix} \cos(q_n) & -\cos(\delta_n)\sin(q_n) & \sin(\delta_n)\sin(q_n) & a_n\cos(q_n) \\ \sin(q_n) & \cos(\delta_n)\cos(q_n) & -\sin(\delta_n)\sin(q_n) & a_n\sin(q_n) \\ 0 & \sin(\delta_n) & \cos(\delta_n) & d_n \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.7}
$$

This resulting matrix can be summarized to a matrix with two submatrixes, the 3x3 submatrix R that describes the rotation and the 3x1 submatrix T that describes the translation, as in *Equation 2.8* [34].

$$
{}^{n-1}_{n}T = \left[\begin{array}{ccc|c} & R & & T \\ \hline 0 & 0 & 0 & 1 \end{array}\right] \tag{2.8}
$$

The transformation of the kinematic chain is the product of the homogeneous transformation matrix of each link (see *Equation 2.9*).

$$
{}^{0}_{m}T = {}^{0}_{1}T \cdot {}^{1}_{2}T \cdot {}^{2}_{3}T \ldots \cdot {}^{m-1}_{m}T \tag{2.9}
$$

According to *Equation 2.8*, *Equation 2.9* can be further summarized which yields a matrix with a rotational and a translational component, as presented in *Equation 2.9*,

$$
{}^{0}_{m}T = \begin{bmatrix} {}^{0}_{m}R & {}^{0}\vec{r}_m \\ 0 & 1 \end{bmatrix} \tag{2.10}
$$

where ${}^{0}_{m}R$ is the rotation matrix, that describes the orientation of the two coordinate systems of two ends of the kinematic chain (the fixed base and the end-effector), and ${}^{0}\vec{r}_m$ is the position vector, that describes the origin of the end-effector relatively to the first segment of the chain.

2.5.2 Inverse kinematics

The aim of the inverse kinematics is to determine the positions and angles of the joint angles of the investigated kinematic chain with a given position of the end-effector. The required equations for the inverse kinematics are nonlinear, meaning there is no linear relation between the end-effector and the joints of the kinematic chain. Therefore, no general approach exists [32]. Still, there are three approaches to solve these equations, an analytic approach, a numeric approach and a geometric approach.

As for the forward kinematics, in the geometric approach, the investigated robotic arm or system is parsed in 2D Cartesian coordinate systems and then solved. For complex kinematic chains, this task can become very complex. With the numeric approach, on the contrary, kinematic chains with rotational and translational axes and six degrees of freedom can be solved mathematically. The analytic approach can only be applied if certain conditions are given [32].

In general, for serial robots, the forward kinematics has an explicit outcome – the inverse kinematics, in turn, can have infinite outcomes [33]. This can be attributed to the variety of different possible joint positions that would lead to the same position of the end-effector. Some of the outcomes are not valid though, as due to the mechanical constellation, many positions are not reachable.

Chapter 3

State of the art

Monte Carlo methods are these days mostly utilized in combination with a search tree, as MCTS. MCTS has great success in the difficult problem of a *Go* computer, but also proves to be beneficial in a range of other domains. The domains include games of different kind, optimisation, satisfaction, scheduling, planning, et cetera [6].

In the domain of games, the MCTS is applied to many classic board games, as two-player games with perfect information – mainly with success. Examples therefore are the game of *Go* [35], *Chinese Dark Chess* [36] or *Hex* [37]. For games of this kind, the MCTS is sometimes further combined with opening books, that provide highly valued starting moves and strategies. Modern board and card games, that are mostly discrete in information and may be on a random basis, often have a variable initial setup that makes it impossible to use opening books [18]. As the MCTS does not require any analytic information, unlike other AI approaches, it can be applied to such games – to modern games like *Poker* [38], *Hearthstone* [39] or *Settlers of Catan* [40]. Similarly, it is suitable for real-time video games. Real-time games are not turn-based and mostly multi-player games where the environment and the tasks are complex. The MCTS is applied to several games of this complexity, for example to *Ms. Pac-Man* [41] and to some games for the Atari [42], but only partially with success. A special kind of real-times video games are real-time strategy (RTS) games. In order to consider the problem of tactical planning of real-time games of high complexity, the MCTS is therefore applied, too. *Balla and Fern* use the game of *Wargus* [43] for this purpose. Furthermore, *Naveed et al.* utilized the Open Real Time Strategy (ORTS) [44] Game Engine for the same reason of investigation.

Many approaches for many real-time games can be applied successfully, but not all in real-time yet. The games are often slowed down for the MCTS. For classic or modern board or card games, in turn, the MCTS based AIs can be applied with no abnormal restrictions, like slowing down the game. Due to this applicability, the Monte Carlo based approaches are used in this work to build a reliable *Connect Four* program.

There are several Connect Four programs, the major part is based on knowledge. The first published program is *VICTOR*. It is based on the work from *Allis* [11] and implemented by him. A very similar approach is *Velena* from *Bertoletti* [45]. It plays perfect with the help of an opening book and eight mathematical rules, which are described by *Allis* [11]. Another approach uses *Velena* as supervisor to learn from it during simulations [46]. Beyond these approaches, there are approaches that play against them selves to learn. *Faußer* and *Schwenker* [47] use a neural network and a Monte Carlo policy evaluation method to train their program. *Thill et al.* [48] utilizes temporal difference (TD) learning methods, which are similar to Monte Carlo methods, but make additional early predictions to match more accurate predictions about the outcome of a simulation before it is known [49]. Moreover, *Young*[1] presents an online *Connect Four* program that is trained with a neuronal network based on *AlphaZero* [35], an algorithm developed by Google as a generalization and optimization of *AlphaGo* [8]. All these approaches use a sort of database, that can either be given or created with training. The training and so the database require time, computing power, memory and disk space.

Besides these knowledge based programs, there is one more published approach, that is based on temporary heuristic knowledge gained with simulations at a given state. *Baier* and *Winands* [50] present three MCTS-minimax hybrids in order to refine the MCTS. Minimax is a decision rule that can cut of whole branches of a search tree. For the three hybrids, the minimax is integrated in different phases of the MCTS: in the backpropagation phase, in the selection and the expansion phase, solely in the selection phase (for details about the phases see *Subsection 2.2.2*). The results show that for *Connect Four* this approach does not lead to significantly better results than a regular MCTS approach.

None of these approaches is designed or tested on computing platforms with limited computing power, memory and disk space, but rather the opposite – Google's *AlphaGo* is distributed over many machines and uses 48 Tensor Processing Unit (TPU) packages [35]. A TPU is an AI accelerating application-specific integrated circuit (ASIC) developed by Google specially for neural network machine learning [51]. For *AlphaGo* the first generation of TPUs is utilized, where each TPU provides a clock speed of 700 MHz and is combined with 8 GB of weight memory.

[1]Website: https://azfour.com [05.02.2020]

Chapter 4

Materials and methods

For the experiments, the two Monte Carlo based search algorithms described in *Section 2.2*, MCS and MCTS, are applied to the game of *Connect Four*. In order to obtain reliable results and to ensure consistent behavior of both algorithms, their limiting condition has to be the equal. Therefore, a fixed amount of computing time to calculate a best move is used for both approaches. It is notable that the computing time a simulation takes varies depending on the game's progress. The more progressed a game is, the less moves and thus less time is needed to reach a final game state. This results in a decrease of the computing time of a simulation and hence in an increase of the algorithms' performance – they become stronger in the late game. Additionally, time is an overarching variable that can be set independently of the used embedded system, which is why it is utilized as the limiting factor for the two approaches.

Both search algorithms are tested by carrying out games where they play against each other. The strength of both algorithms can thereby be evaluated by calculating the win rate of the respective search algorithms. Therefore, similarly to *Equation 2.1*, the win rate is calculated by dividing the amount of wins by the amount of performed games. Furthermore, the simulation speeds of both search algorithms, representing how many simulations of a search for the first move of a game are conducted within an equal given time, are compared.

Therefore, for the tests, the defined first player (the search algorithm making the first move) as well as the limitation factor (a computing time between 100 and 5000 ms) are varied. Each of the in total 12 tests (see *Table 4.1*) consists of 100 simulated games and is run five times in order to quantify the fluctuations of the performance. The stronger algorithm, CarloConnect, is then placed against human opponents to further evaluate the AI's strength and investigate the humanity of its behavior, wherefore a short survey is made.

Table 4.1 | Overview of the tests performed to compare the MCS with the MCTS. For each of the six different computing times, tests where the MCS makes the first move and tests where the MCTS makes the first move are performed. Thereby, both algorithms can be evaluated equally.

Computing time (ms)	Test n°, first player MCS	Test n°, first player MCTS
100	1_a	1_b
200	2_a	2_b
500	3_a	3_b
1000	4_a	4_b
2000	5_a	5_b
5000	6_a	6_b

In order to realize the playing against human opponents and evaluate CarloConnect, a website is set up to play online. Furthermore, to play with physical components, a 3D printed game board is designed, whereby a 3D printed robotic arm is utilized to carry out the moves calculated from CarloConnect. For the robotic arm, a printable open source 3D model is selected, the EEZYbotARM MK1[2]. The full control of the robotic arm requires four servo motors, wherefore the MG90S[3] providing metal gear for additional strength and durability is used. In order to digitize the player's move, a small micro switch, the SMKW-01[4], is integrated in each of the seven slots for the game pieces of the game board. For different levels of difficulty a rocker switch of the kind DA101[5] is implemented. Thereby, three different levels of difficulty, meaning different time limits for CarloConnect can be selected. To restart the game or start a new game, a pushbutton switch, the MS-100630[6], is integrated in the system, too.

All components of the game board and the robotic arm are 3D printed with a Creality Ender 3 Pro 3D printer[7], using a profile precision of 0.2 mm with a 0.4 mm nozzle.

To run the search algorithms, control the servo motors and process the signals, a computing platform is required. SOMs and SBCs provide all requirements to fulfill these necessary functionalities. In addition, they have the advantage of being very compact, low-energy consuming, portable and mostly inexpensive. The market offers several products in this realm. Two very conventional

[2]Website: http://eezyrobots.it/eba_mk1.html [10.01.2020]
[3]Datasheet: https://engineering.tamu.edu/media/4247823/ds-servo-mg90s.pdf [10.01.2020]
[4]Datasheet: https://workupload.com/pdf/cy9fNTrt [11.01.2020]
[5]Datasheet: https://mouser.com/datasheet/2/60/rocker-1109491.pdf [11.01.2020]
[6]Datasheet: https://mouser.com/datasheet/2/209/MS-100630-1172392.pdf [11.01.2020]
[7]Website: https://creality3dofficial.com/products/creality-ender-3-pro-3d-printer [17.01.2020]

options are the Arduino[8] and the Raspberry Pi[9]. Both computing platforms are available in several variations, differing in performance and connectivity, which makes them applicable for projects comparable to the one of this work.

Investigating the available variants of the Arduino exhibits that most of them come with 2 to 32 kilobytes of memory. In consideration of the MCTS based algorithm, having sufficient memory is an important aspect. Saying that an object that represents a node in the search tree requires roughly 40 bytes would result in a maximum tree size and thus an maximum amount of simulation of 51 to 819, if no memory is required for the rest of the algorithm, depending on the Arduino variant. These amounts of simulations do not suffice for a good approximation of the best move of the given game state. A restriction in memory like for the Arduinos may not affect the MCS based algorithm but it is crucial for the approach that uses the MCTS. The search of the MCTS would be thereby limited, besides the computing time, additionally by the available memory.

Raspberry Pis have a memory of at least 512 to 1024 megabytes – not restricting the algorithms in this regard. While Arduinos can only compile Arduino programming language, which is based on C++, Raspberry Pis can compile any programming language as Linux or Windows based operating systems can be installed on them. This makes Raspberry Pis more complex but flexible, too.

Due to the limited memory of the Arduinos, a Raspberry Pi is the better choice for this work. All tests are performed on a Raspberry Pi 3 Model B with a 64-bit 1.2 GHz quad-core processor and 1 gigabyte of memory, running on Ubuntu 16.04.6 LTS.

[8]Website: https://arduino.cc/en/products/compare [12.01.2020]
[9]Website: https://raspberrypi.org/products [12.01.2020]

Chapter 5

Construction

The physical components presented in this chapter are individual and must be constructed therefore. The circuit board for the power supply is soldered. All remaining parts are for the purpose of replicability and customization 3D printed. In order to ensure that the components can be connected properly, a tolerance of 0.3 mm for each interstice is used for connecting parts.

5.1 Robotic arm

The EEZYbotARM MK1 robotic arm is 3D printed. The models of the arm are provided online on the website mentioned above. The given instructions help to compose all components including the servo motors.

For the purpose of this work, the models of the gripper are modified to facilitate and stabilize the gripping of balls (see *Figure 5.1*).

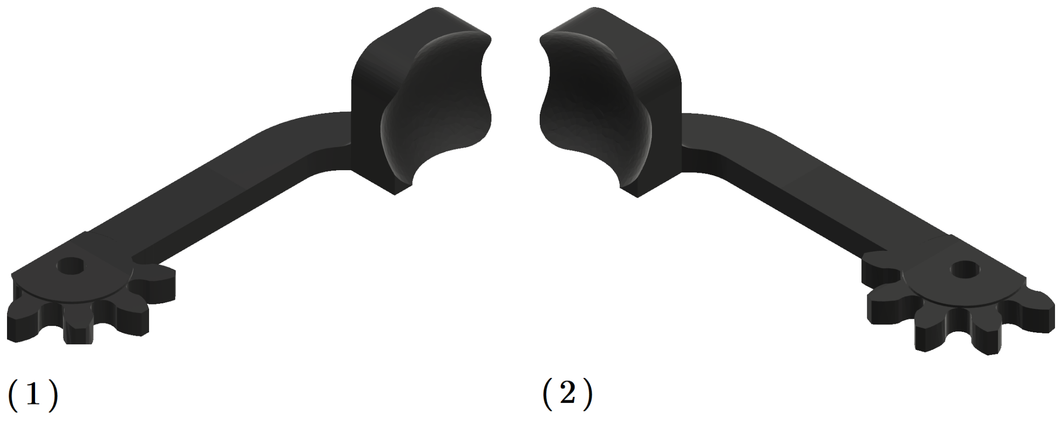

Figure 5.1 | Gripper of the robotic arm. This 3D model is designed for the EEZYbotARM MK1 robotic arm in order to facilitate and stabilize the gripping of balls. From the perspective of the robot, (1) is the left part of the gripper and (2) is the right part of it.

Therefore, the palms of the gripper are adjusted stabilizing the balls from all sides when grabbed. Furthermore, the length is extended to ensure sufficient space between the grabbed ball and the robot arm's body.

In *Figure 5.2* the complete robot with the new gripper models and the four required servo motors is presented. The four servo motors are used to rotate the whole robot about the vertical axis of the ground where the robot arm is attached to, to move the arm forwards and backwards, to move it in vertical direction and to open and close the gripper.

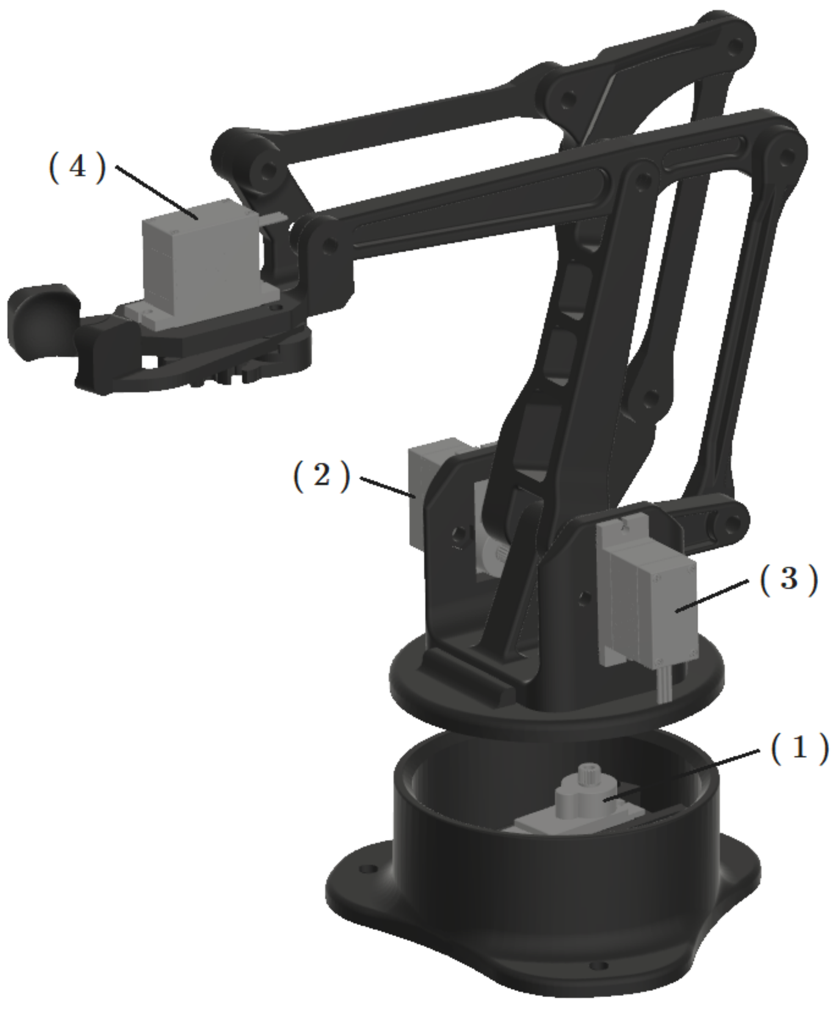

Figure 5.2 | Fully composed robotic arm. The robotic arm is composed of its original open source 3D components from the EEZYbotARM MK1 robot arm, a for this work designed gripper for the robot arm and four servo motors. (1) rotates the robot arm about the vertical axis of its ground plate, (2) moves it forwards and backwards, (3) moves it in vertical direction, (4) opens and closes the gripper.

5.2 Game board and periphery

The game board is designed to use balls instead of discs for the game, as they can be grabbed and inserted easier. Hence, a set of seven connected individual tubes is printed to form the game board (see *Figure 5.3*).

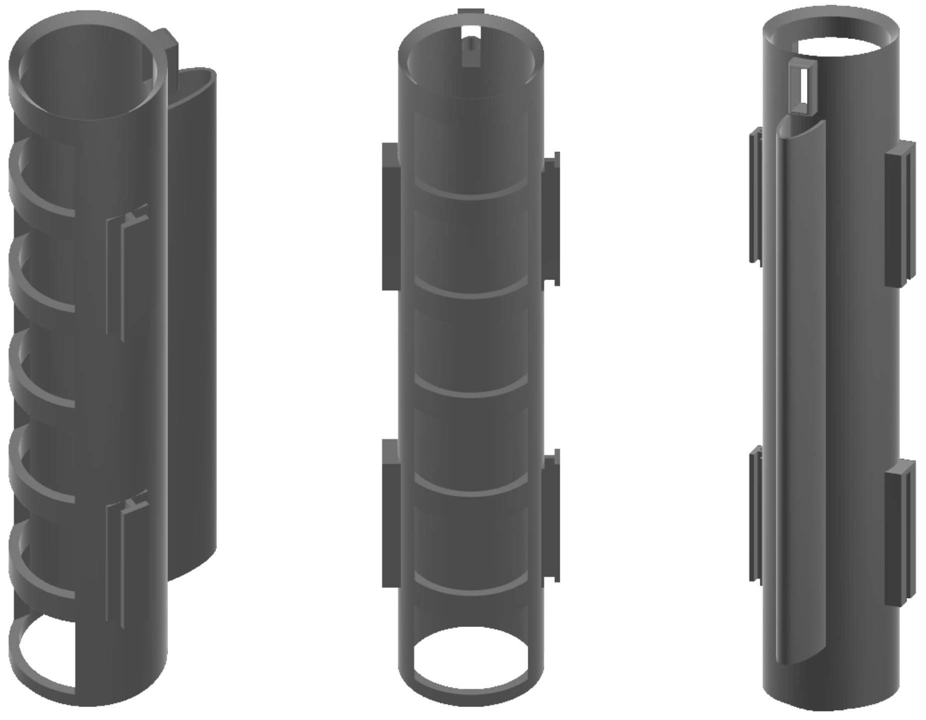
Figure 5.3 | Tube of the game board. Three perspectives of a tube are shown. Seven tubes are connected to build the game board. The balls are visible through the windows of the tubes, the slot and cable duct on the back side are for a micro switch.

A tube provides six windows on front side – the human player faced side – to see all inserted balls. On the back side, a cable duct and a small slot above the cable duct are added. These are designed to integrate a micro switch at the top of the tube that is used to recognize inserted balls. For the connection to the other tubes or stands, t-profiled guide rails are applied on both sides.

The first and the last tube are connected with their free faces to the stands of the game board, shown in *Figure 5.4*. The stands stabilize the set of tubes and lift it up to provide space for releasing the balls from the tubes. Holes on the bottom plate of the stands provide that the game board can be mounted with screws in a fixed position.

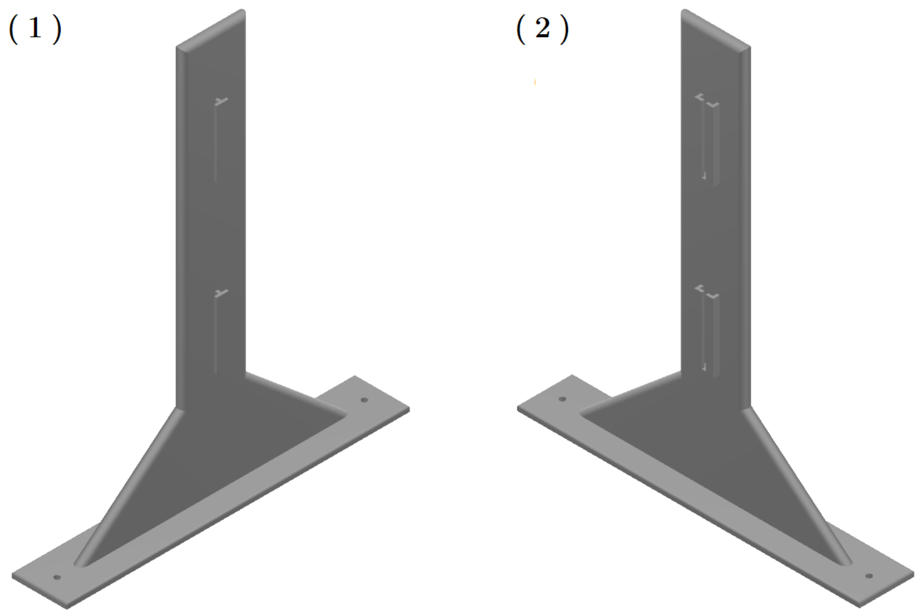

Figure 5.4 | Stands of the game board. For both sides of the game board a stand is required. Each stand has the according connecting part of the t-profiled guide rails, where (1) is the stand for the left hand side from the perspective of the human player and (2) is the stand for the right hand side. The stands stabilize and lift the tubes so that the balls can be released through the bottom side of the tubes.

To release the balls from the tubes, a tank is designed that can carry all balls (see *Figure 5.5*). It is positioned under the tubes so that these are closed from the bottom side. The tank is flexible, if moved in direction of the robot, the bottom sides of the tubes are opened. This way, at the end of a game, the balls can be removed from the tubes without dismantling the whole game board. The tank cannot be pulled too far in direction of the human player due to a guide rail on its tank backside.

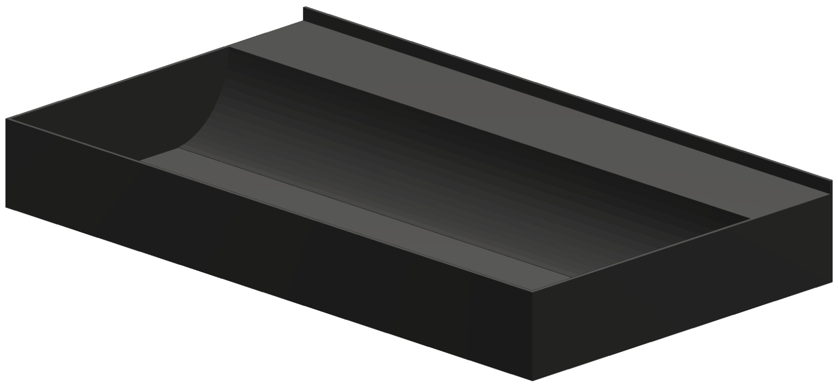

Figure 5.5 | Tank for the bottom of the tubes. This tank is positioned at the bottom of the tubes of the game board. It keeps the balls inside the tubes. In order to open the tubes and release the balls, it can be moved toward the robot.

In *Figure 5.6* all of the above described components are set together to build the final game board.

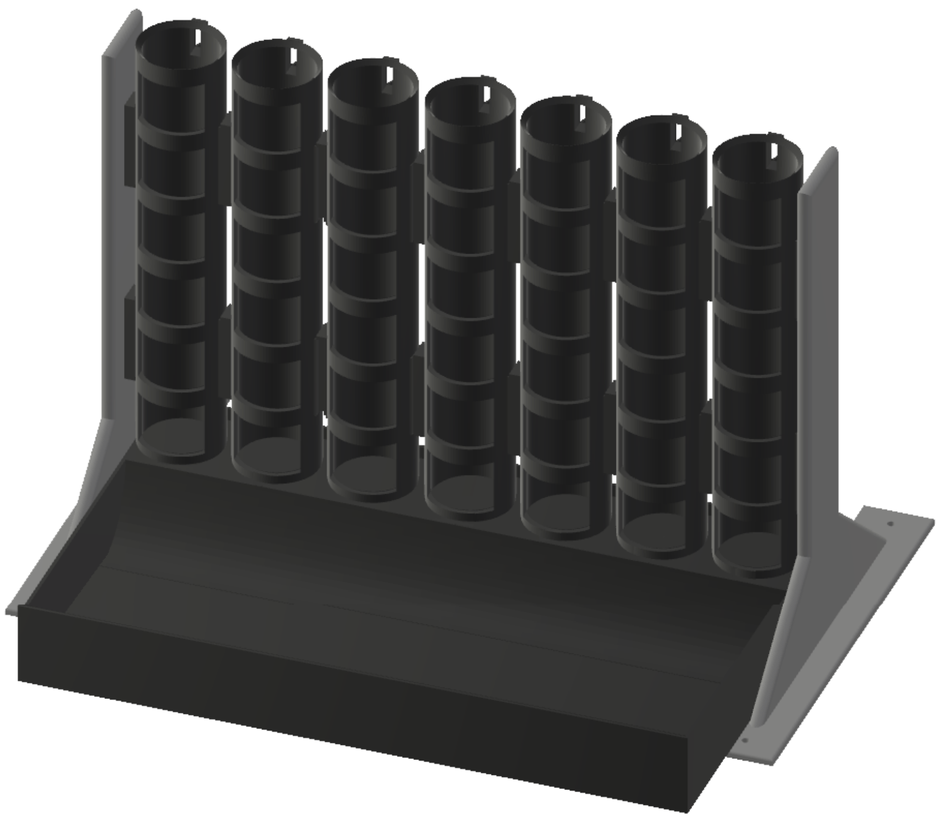

Figure 5.6 | Assembled game board for Connect Four. Seven tubes, two stands and a tank build together the game board. The tubes and the stands are connected with t-profiled guide rails. The tank is placed below the tubes.

The balls, shown in *Figure 5.7*, are adjusted in size and weight ensuring that the robot can grab and insert these playing pieces properly and that the micro switch is activated when a ball is dropped into a tube.

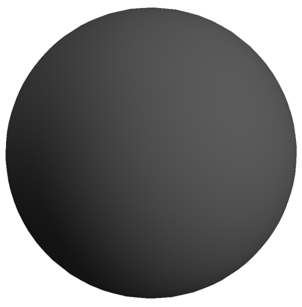

Figure 5.7 | Balls as playing pieces. To ensure that the playing pieces can be grabbed and inserted properly from the robot, balls are used instead of discs. By printing these with a 3D printer, the size and weight can be modulated easily.

In order to receive the balls from a constant position, which is required for a steered robot, a ramp (see *Figure 5.8*) that stores all balls for the robot and forward always a single ball to the constant position is designed. In terms of the fixed position, as well as the stands of the tubes, holes are placed on the bottom plate of the ramp to provide that the ramp can be mounted with screws.

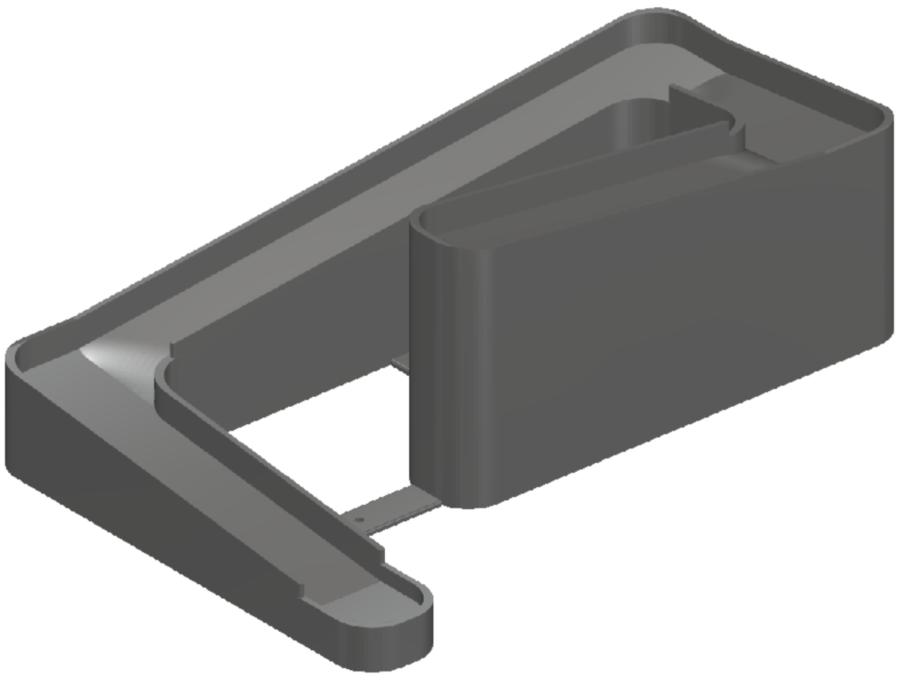

Figure 5.8 | Ramp to pass the balls. All balls of a player for a game are stored in the ramp. It forwards the balls individually to the end of the ramp. Thereby, the robot can pick it up from a constant position.

A ramp is manufactured for the human player, too, which is the exact same as the one for the robot.

The reset pushbutton switch, the rocker switch that determines the level of difficulty of CarloConnect and the power supply circuit board, presented later in this section, are brought together in a compact case. This case is rectangular, providing slots on the top for the switches, a gap on the front side for the cables of the switches and a recess, as well on the front side, to put in the power supply circuit board. There are holes on the bottom of the inner side of the case in order to mount it with screws. The case is presented in *Figure 5.9*.

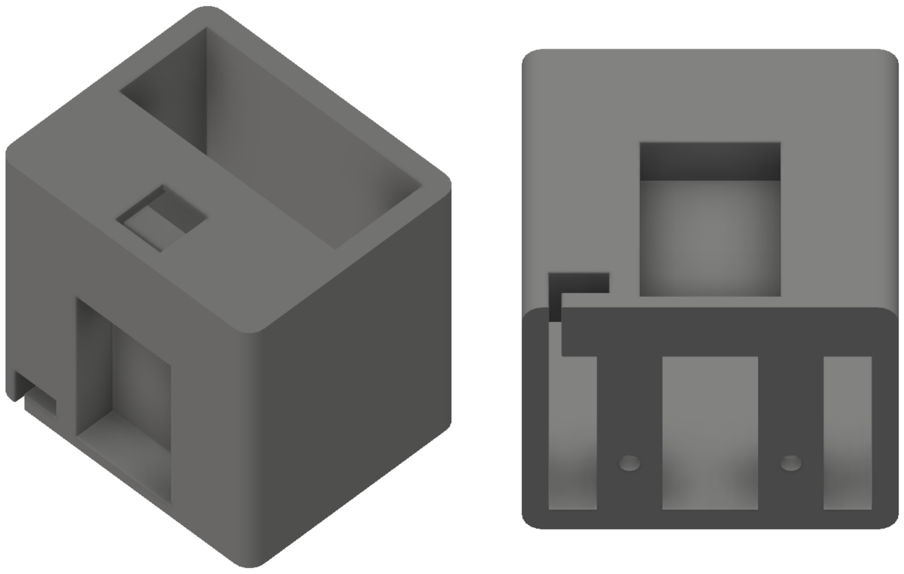

Figure 5.9 | Case for switches and the power supply circuit board. Two perspectives of the case are demonstrated. This case provides slots and recesses to place the rocker switch, the pushbutton switch and the power supply circuit board, which are used in this work.

5.3 Power supply circuit board

As the servo motors are only driven one at a time, the power supply of the utilized Raspberry Pi suffice. Still, all servo motors require a connection to the according power output of the computing platform. Since there are only two 5.5 V outputs, which is the necessary operating voltage, the access has to be extended. The same applies to the ground connection. A circuit board for the complete power supply is soldered therefore.

Many servo motors have a 3-pin female connector. For this reason, it is useful to use male pin headers on the circuit board as output for the servo motor connections. Besides the voltage and the ground, the servo motors require an individual PWM signal. This port can be accessed via a solder connection between the according male pin header of the connector of the servo motor and a female pin header. The female pin headers are connected to output pins of the Raspberry Pi. In order to distinguish the pin headers generally, for output signals from the Raspberry Pi only female pin headers are used. Accordingly, the male pin headers are only used for the output from the circuit board, suitably for the connectors of the servo motors.

Each of the seven micro switches, the rocker switch and the pushbutton switch need a ground connection, too. For this purpose, another nine male pin headers are implemented on the circuit board. *Figure 5.10* demonstrates an illustration of the surface of the circuit board.

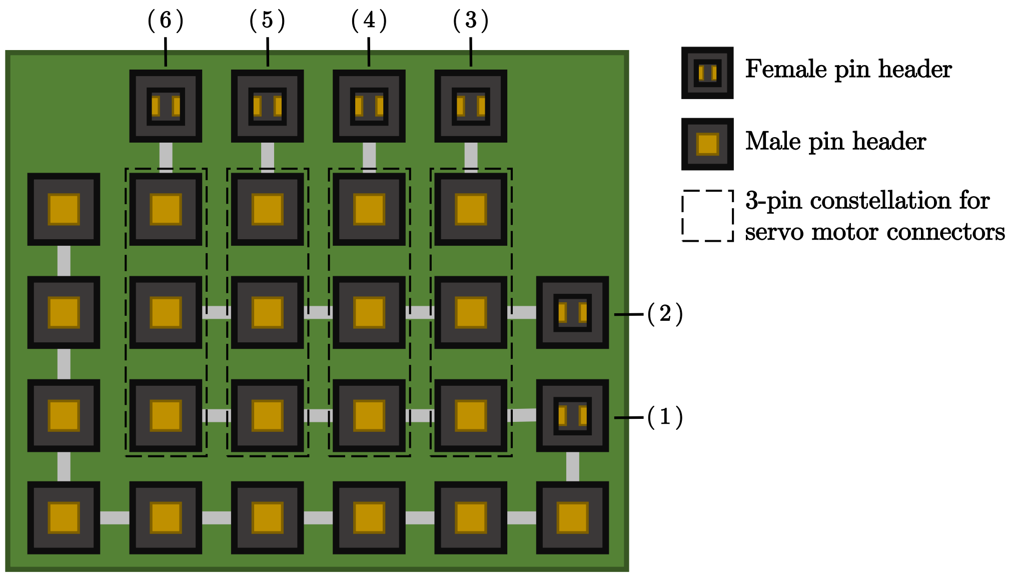

Figure 5.10 | Power supply circuit board. Via the female pin headers, that are connected to the Raspberry Pi, this board provides a ground connection (1) for the four servo motors, the seven micro switches, the rocker switch and the pushbutton. In addition it supplies voltage (2) for the four servo motors. The female pin headers (3) - (6) deliver individual pulse width modulation (PWM) signals for each of the four servo motors.

5.4 Assembly

This section presents briefly the composed game board including all components of this chapter and the Raspberry Pi 3 Model B (see *Figure 5.11*).

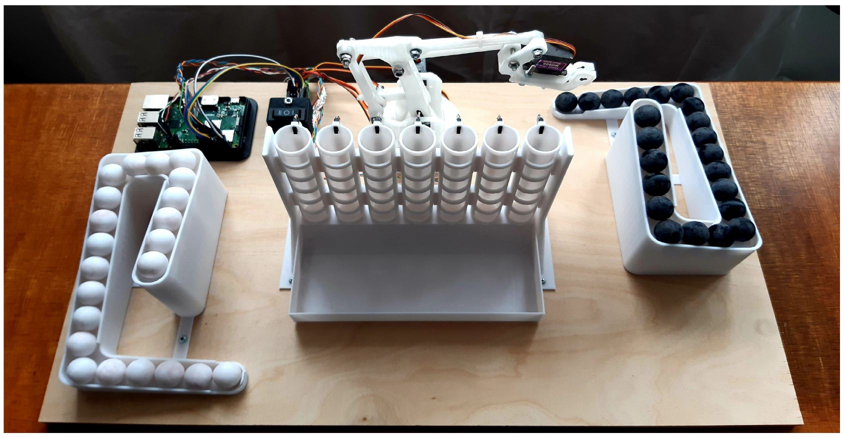

Figure 5.11 | Final mounted game board. All components described in this chapter are mounted on a wooden plate to build the final game board which can be used to play *Connect Four* with physical components. Balls of the color black are for the robotic arm, balls of the color white are for the human player.

Chapter 6

Implementation

A general overview of how the implementations of the game of *Connect Four*, the MCS, the MCTS, the control of the servo motors, the read of the signals from the micro switches and the website is presented in this chapter. Everything, except the website, is implemented in C++. For the robotic arm, the final program is run on startup via systemd as service, daemon. A daemon is a process that continues running until the system is shut down [52]. Systemd is the first process that is started during booting of the computer system. It is a daemon process that manages the system including all user processes.

6.1 Connect Four

The game board of the game is realized as a two dimensional array. When the main function is run, the game board is initialized. Then, in a loop, from the player whose turn it is the move is requested. When the move is set, the game state is checked to see if a final game state is reached. In case the player's move is a winning move, the loop breaks. Otherwise, the actual player is switched so that it is the other player's turn. At that the loop is repeated.

The function that checks if a final game state is reached is optimized by a reduction of the amount of necessary comparison iterations. From initial 69 iterations (a check of every row, column and diagonal possibility), a reduction of up to 88.4 % of the performed iterations is possible. For this reduction, the exact position of the row and column of the new set move is used. Thereby, only 8 to 13 iterations, depending on the position of the new set move, need to be carried out.

6.2 Search algorithm MCS

When a move is searched with the MCS, the current game state and the time limit is passed. The available time is divided by the amount of possible actions. Thereby, each of the available actions has the same amount of time. When the amount of available actions shrinks with the game progress, the time is still split equally. For every available action of the current game state as many simulations are performed as possible within the proportionate time limit.

In accordance with the description of the MCS in the theoretical fundamentals, each simulation returns a result representing which player wins in the respective simulation. This result is added to the win count, +1 if for a win and -1 for a loss. When the time is up, a win ratio is calculated with the win count and the amount of performed simulations. Thereupon the win count and the amount of simulations is reset and simulations for the next possible action are carried out. When for all possible actions the time limit is over, the action with the highest win ratio is returned. *Algorithm 1* demonstrates the iterative process as pseudo code.

Algorithm 1 | The Monte Carlo Search approach.

```
1: procedure RunMCS($state, limit$)
2:     $limit \leftarrow limit/amountPossibleActions$
3:     for every possible $action$ do
4:         $simulations \leftarrow 0$
5:         $wins \leftarrow 0$
6:         while $limit$ is not reached do
7:             $result \leftarrow$ Simulate($state, action$)
8:             $wins \leftarrow wins + result$
9:             $simulations \leftarrow simulations + 1$
10:        $ratio[action] \leftarrow wins/simulations$
11:    return $action$ with the highest $ratio$
```

6.3 Search algorithm MCTS

For a MCTS search, besides the passed variable for the MCS search, the time limit and the current game state, the current player is passed, too. First, a root node on the basis of the current game state is created and copied to another new node. In terms of the iterative process, the current game state and the current player are copied to variables. All of these three copies are marked as temporary. The child nodes of a node are implemented as an array of pointers to nodes. The parent node of a node is referenced similarly with a pointer to a node. Thereby, full access and referencing is possible.

In the loop, the amount of child nodes of the temporary node are checked. If there are child nodes that are not expanded yet, randomly one of the left child nodes is expanded by adding it to the list of child nodes from the temporary node. As a child node represents a move of a given state, the corresponding move is used for a simulation. The simulation returns the winner of it that equals a score. This score is updated to all nodes that are traversed to get to this child node. Additionally, the temporary variables, node, game state and player, are reset to the initial values utilizing the root node and the passed variables. Then the loop is repeated.

If all child nodes of the temporary node are expanded, the state of the node is checked to ensure that it is no final game state. If it is a final game state, all traversed nodes are updated and the temporary values reset. Otherwise, the best of all child nodes of the temporary node is selected using the UCT strategy for selection. The representing move of the selected child node is then set in the temporary game state, the current player is switched, stored in the temporary player variable, and the loop is repeated.

When the time limit is reached, the loop breaks and the best child node of the root node is selected. Therefore, as final selection criterion, in accordance with the studies of *Chaslot* [21], the *Robust Max child* is utilized. The move that represents the child node that is selected in doing so, is returned. In *Algorithm 2* the principle of the algorithm is presented as pseudo code.

Algorithm 2 | The Monte Carlo Tree Search approach.

```
1: procedure RunMCTS($state, player, limit$)
2:     new $node_{root}(state)$
3:     new $node_{temp} \leftarrow node_{root}$
4:     $state_{temp} \leftarrow state$
5:     $player_{temp} \leftarrow player$
6:     while $limit$ is not reached do
7:         if unexpanded child node of $node_{temp}$ left then
8:             new $childNode \leftarrow$ RandomChildNode($nod_{temp}e$)
9:             add $childNode$ to $listChildNodes$ of $node_{temp}$
10:            $result \leftarrow$ Simulate($state_{temp}, childNode$)
11:            Update($result$)
12:            $node_{temp} \leftarrow node_{root}$
13:            $state_{temp} \leftarrow state$
14:            $player_{temp} \leftarrow player$
15:        else
16:            if $state_{temp}$ is a final game state then
17:                $result \leftarrow$ CheckWin($state_{temp}$)
18:                Update($result$)
19:                $node_{temp} \leftarrow node_{root}$
20:                $state_{temp} \leftarrow state$
21:                $player_{temp} \leftarrow player$
22:            else
23:                $node_{temp} \leftarrow$ UctSelect($node_{temp}$)
24:                SetMove($state_{temp}, node_{temp}$)
25:                SwitchPlayer($player_{temp}$)
26:    $action \leftarrow$ RobustMaxChild($note_{root}$)
27:    return $action$
```

6.4 Controlling of servo motors

The Raspberry Pi 3 Model B only provides two hardware based PWM signals. Hence, in order to control four servo motors, software based PWM signals are required. Even though all of the four signals could be software based, still, the two hardware based PWM signals are used for this work, too, as these are more accurate than the software based PWM signals. In order to facilitate the implementation, a library providing functions for pin controlling, wiringPi[10], is applied. It supports the BCM2837[11] SoC, which is used by the Raspberry Pi Model 3 B. The BCM2837 provides a crystal oscillator that generates clock pulses of 19.2 MHz with a minimum pulse width of 100 μs.

When setting up hardware based PWM signals, the according pin must be configured as PWM output. There are two kinds of PWM output signals that can be produced with this library – the Balanced mode and the Mark-Space mode. The Balance mode sends a combination of the clock pulses which results in an average pulse height that is sent for during the whole period. In Mark-Space mode, the output is set HIGH during a pulse and LOW for the rest of the period. For the purpose of sending PWM signals for angular control of the servo motors, the Mark-Space mode is used.

The frequency of the PWM signal can be configured by modifying the PWM range and the clock divider. According to the formulas explained in the theoretical fundamentals (see *Subsection 2.4.2*), both the PWM range and the clock divider can be calculated with the crystal oscillator's clock and minimum pulse width. This way, any desired frequency for the PWM signal can be set.

The MG90S servo motors are designed to be updated with a frequency of 50 Hz. To achieve this frequency, first the period has to be calculated using *Equation 2.3.*

$$T = \frac{1}{50\,Hz} = 20\,ms \tag{6.1}$$

According to *Equation 2.4*, the PWM range can be calculated taking this period further together with the minimum pulse width.

$$PWM_R = \frac{20\,ms}{100\,\mu s} = 200 \tag{6.2}$$

Last, the clock divider can be determined as in *Equation 2.5.*

$$PWM_{CD} = \frac{19.2\,MHz}{50\,Hz \cdot 200} = 1920 \tag{6.3}$$

With these values the configuration can be carried out. Having all these preparations done, a PWM signal can be sent to set the angles for the servo motors. Using the values above, angles from 0 to 180 ° can be set by passing values

[10]Website: http://wiringpi.com [24.01.20]

[11]Datasheet: https://cs140e.sergio.bz/docs/BCM2837-ARM-Peripherals.pdf [24.01.20]

from 5 to 23 to the PWM function of the wiringPi library. Hence, there are 18 different adjustable positions – steps of 10 ° each.

In order to ensure that all positions of the tubes can be reached, the resolution has to be refined. This can be achieved by increasing the PWM range. To maintain the frequency of the PWM signal, the clock divider has to be decreased proportionally. Therefore, a coefficient of 2,6 is used resulting in a PWM range of 520 and a clock divider of 739. With this resolution of the PWM signal, values from 15 to 60 can be used for the PWM signal. Thus, 45 different adjustable positions are possible – steps of 4 ° each.

The servo motor that rotates the robot arm about the vertical axis of its ground plate and the servo motor that moves it forwards and backwards are controlled with the hardware based PWM signals. The remaining two servo motors are consequently controlled with software based PWM signals (see *Figure 5.2*).

A software based PWM signal can be created by defining a pin and the desired PWM range within the same function. With a PWM range of 200, the resulting PWM signal has a frequency of 50 Hz. By modifying the PWM range, only the frequency can be configured. The resolution of a software based PWM signal cannot be configured with this library. Hence, only values from 5 to 23 can be used for the PWM signal to set angles from 0 to 180 ° – again in steps of 10 ° each. These should suffice for the servo motor that moves the robot arm in vertical direction and the servo motor that opens and closes the gripper.

To make sure that the software based PWM signal is sufficient accurate, a digital waveform viewer, piscope[12], is used to investigate the outputs of the utilized pins. A screenshot of three different simultaneously tested signals is presented in *Figure 6.1*, showing that the signals are very accurate if no other task is run at the same time. Thus, for the purpose of controlling servo motors, the signals are accurate enough and can be used for robot arm.

For smooth movements of the servo motors, a function is implemented that increases or decreases the position of the servo motors iteratively. Therefore, the old and the new positions of the servo motors must be passed to the function.

Due to the low resolution of the software PWM signal (the inaccuracy about the according angles), the backlash between the joints and the links and the limitation of the mechanical movements of the robotic arm, the positions (values) of the servo motors for the positions required for the game play are not determined with the inverse kinematics, but with testings instead.

[12]Website: http://abyz.me.uk/rpi/pigpio/piscope.html [25.01.20]

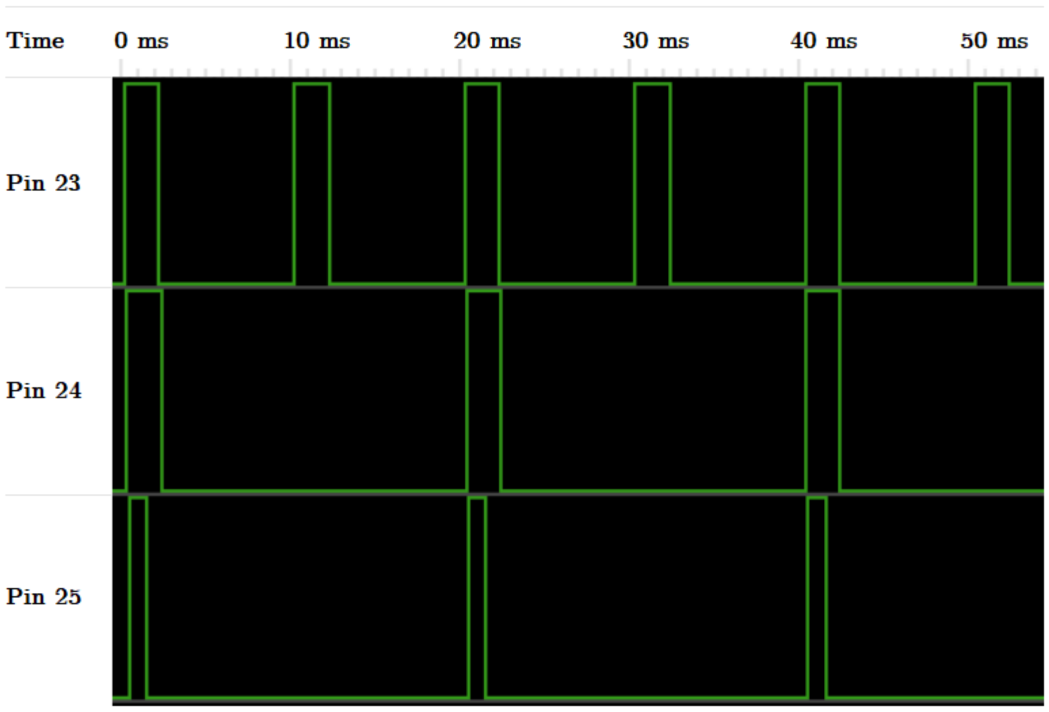

Figure 6.1 | Scope of a software based pulse width modulation (PWM) signal. Three different software based PWM signals on three different pins are produced and observed with this scope in terms of the accuracy. Pin 23 has a frequency of 100 Hz and a pulse width of 2 ms. Pin 24 has the same pulse width but half the frequency, 50 Hz. Pin 25 has half of both values compared to Pin 23, a frequency of 50 Hz and a pulse width of 1 ms.

6.5 Signal processing with switches

When the human player inserts a ball into one of the seven tubes of the game board, the selected tube has to be registered by the program. Therefore the micro switches are integrated. These are placed at the top of each tube, whereby they are activated when a ball is inserted, regardless of the game state.

To receive a signal when a micro switch is activated, a signal has to be passed to the micro switches, that is forwarded when a micro switch is in the activated, pressed, state. At the same time, when the micro switches that are not pressed, a signal unequal to the activation signal must be forwarded. As activation signal a LOW signal, the ground, is used.

Out of the three available pins from the micro switches (see *Subsection 2.4.1*), only two are utilized. The normally opened channels are connected to the ground of the Raspberry Pi, the common terminals are connected to individual pins of the Raspberry Pi that are initialized as input pins. Applying the same library as for the servo motors, the wiringPi library, this initialization can be done with a single function respectively. In order to ensure that while the micro switches are not pressed the input signal is HIGH, not LOW, the input pins must be initialized further by activating on-board pull-up resistors for each input pin. Therefore, another function can be used. This is necessary, as the normally closed channels are not wired and thus the input signal can be arbitrary – HIGH or LOW.

The rocker switch for the adjusting the level of difficulty of CarloConnect is implemented similarly. Therefore, again the wiringPi library is applied. The common terminal of the rocker switch is connected to the ground, the two terminals are individually connected to pins, which are initialized as input pins with a pull-up resistor activated. This way, three states can be defined: (1) terminal one pulled-up & terminal two pulled-up, (2) terminal one LOW & terminal two pulled-up and (3) terminal one pulled-up & terminal two LOW. For (1) a time limit of 100 ms is set, for (2) 1000 ms and for (3) 5000 ms.

Moreover, to restart the game, a pushbutton switch is integrated into the game board. With an implementation equal to the other switches, a LOW signal is transferred when the pushbutton switch is pressed. In order to ensure that the game can be restarted at any time, an interrupt function (from wiringPi) can be initialized. It registers a desired function to received interrupts on a specified pin. Within the interrupt function, the interrupt can be further configured to be either triggered at falling edges, at rising edges or at both kinds of edges. As a ground signal is transferred when the pushbutton switch is activated, in this case falling edges are defined in the interrupt function to trigger the interrupt. When it is triggered, the desired function is called – in this case, the game is reinitialized, meaning that the board is cleared and the player to move is reset.

To further ensure that the interrupt is triggered only once per activation, a timer based debounce is implemented for the pushbutton switch.

6.6 Parallelization

The Raspberry Pi 3 Model B has a quad-core processor and thus four cores. By default, a program runs on a single thread – on a single core. In order to utilize more than one of the cores for the running code, certain parts of the code, in particular loops, can be spread over multiple threads. Open Multi-Processing (OpenMP)[13] is an application programming interface (API) that supports multiprocessing programming, inter alia, in C++ on many platforms and operating systems. It consists of directives that influence run-time behavior. Thereby, in a for loop for example, the task (iterations) is divided by the amount of desired threads to be used and partially distributed of them. With software profiler, compute-intensive loops can be found that may be worth to parallelize. The to be parallelized loops must be independent without loop-

[13]Website: https://www.openmp.org [07.02.20]

carried dependencies to ensure that the threads can safely execute in any order, that can be arbitrary.

For the purpose of this work, only few operations can be parallelized, as most of the loops carry dependencies that require a sequential iteration. For further parallelization, multiple MCTS instances could help, to find reliable and good decisions. Such adaptations are complex and would go beyond the scope of this work.

6.7 Website – *https://carloconnect.com/*

In accordance with the rest of this work, a Raspberry Pi 3 Model B is used to host a web server providing a Hypertext Transfer Protocol (HTTP) website[14]. On this website, a description about the purpose of the website, a short survey, a *Connect Four* game board and buttons to restart the game and to set different computing time limits for the MCTS - *Easy* (15 ms), *Moderate* (60 ms) and *Hard* (1500 ms) - are given.

The web server is set up with Node.js, a cross-platform JavaScript runtime environment. It is designed to execute server-side JavaScript code, producing dynamic web page contents before the page is sent to the client's web browser. The website is designed with HTML, CSS, JavaScript and jQuery.

As CarloConnect is considered to be strong (due to preliminary testings), the human player always starts first, providing better win chances for him (see *Section 2.1*). When the human player selects a column, the lowest unoccupied slot of the selected column is marked with the human player's color. Thereupon, the game state and the selected time limit for the MCTS are sent as a HTTP POST request to the server, the Raspberry Pi. With a HTTP POST request, the web server accepts data enclosed in the body of the request message. This data can thereby be stored or computed on the server [53]. This website uses, as is common, JavaScript Object Notation (JSON) in order to transfer data. The game state and the time limit are therefore joined to a single JSON object, which is then sent via a HTTP POST request to the server where they are parsed to their original object types.

In order to operate with the received game board and the time limit, all values are saved in a single text file, whereby each value has its own line. Line 1 to 42 are the frames of the game board and line 43 is the time limit. To ensure that each request is saved in an individual text file, the text files are tagged with time stamps including milliseconds. The MCTS script is launched server-side using the unique time stamp as an argument, which is used to read in the according text file at the beginning of the search to initialize the game board

[14]Link: https://carloconnect.com/

and set the time limit. When the time limit is reached, the MCTS script returns the selected move.

A HTTP POST request is not only capable to receive data, but also to send data as response to the request. With this functionality the returned move is sent back as response to the client's browser. According to the received response, the corresponding slot of the browser's game board is then marked with CarloConnect's color. After each move from the human opponent and CarloConnect the game board is checked for whether a final game state is reached. In the case of a final game state, another HTTP POST request is sent and the human player's input is deactivated until the game is restarted via the restart button. The HTTP POST request saves the winner in a text file, that corresponds to the set time limit, on the server. All HTTP POST requests are implemented with Asynchronous JavaScript and XML (AJAX) calls. This way, the client's browser can send and retrieve data from a server asynchronously in the background.

When multiple clients are playing at the same time, multiple HTTP POST requests to get a move from CarloConnect and thereby multiple searches may be launched at the same time. For time limits of 15 ms and 60 ms (difficulty *Easy* and *Moderate*), temporal overlap is unlikely. For the time limit of difficulty *Hard*, 1500 ms, temporal overlaps become probable. As a search runs as a single thread on a single core and the Raspberry Pi 3 Model B offers four cores, the threads are distributed automatically by the system. Still, a search with a low time limit may be assigned to a core where a move of the difficulty *Hard* is launched. In order to avoid this overlap, one of the four cores of the Raspberry Pi is programmatically isolated from system assignments and only used for searches of the difficulty *Hard*. Hence, the three other cores are available for searches of small time limits reducing the impact of temporal overlaps and thus the potential of lag for games of low difficulty. To further ensure that in the case of a lag enough simulations are conducted for the selected difficulty (computing time limit), a minimum amount of simulations is set as second limit condition for the searches. Only if the time limit and the minimum amount of simulations are reached, the search ends. The minimum amount of simulations is the average amount of simulations CarloConnect conducts within the respective time limits when there is only one search running, determined in preliminary testings (175 simulations at *Easy*, 840 at *Moderate* and 31500 at *Hard*).

The survey provided on the website contains two questions, one about the game feeling and one about the age of the client. The answers of these questions are saved analogously to the final game state using a HTTP POST request and saving the content in a text file server-side.

Chapter 7

Results and discussion

To evaluate the two search algorithms, they are tested in performance by conducting games where they play against each other. This way the stronger AI can be determined. The stronger AI is then further evaluated by being placed against human opponents. The results of these tests are presented in this chapter.

7.1 Comparison of the two approaches

The respective strength of the search algorithms is assessed by the MCTS' win rate and further investigated by measuring the amount of simulations that are performed to find the first move.

7.1.1 The win rate of the MCTS

The results, shown in *Figure 7.1*, prove that the MCTS clearly outperforms the MCS. The more computing time is given, the stronger the MCTS becomes in comparison with the MCS until it wins all of the games. This shows that due to the lack of tactical insight of the MCS algorithm, the opponent's behavior does not receive sufficient consideration, which is why the potential of MCS's performance is limited. With a rising degree of complexity of a game, the importance of the tactical insight grows. Therefore the MCS does not suffice anymore and the MCTS begins to lead instead. This applicability is also described in a work [8] about mastering the game *Go*.

In addition, the results show that for each of the differently set computing times the win rate of the MCTS is consistently lower if the MCTS starts first, even when the standard error is taken into account. Investigations of the first moves of the search algorithms prove that in all of the tests, both algorithms always start with a move in the middle column, which provides the potential of a safe win (see *Section 2.1*). Yet, the quantitatively stronger

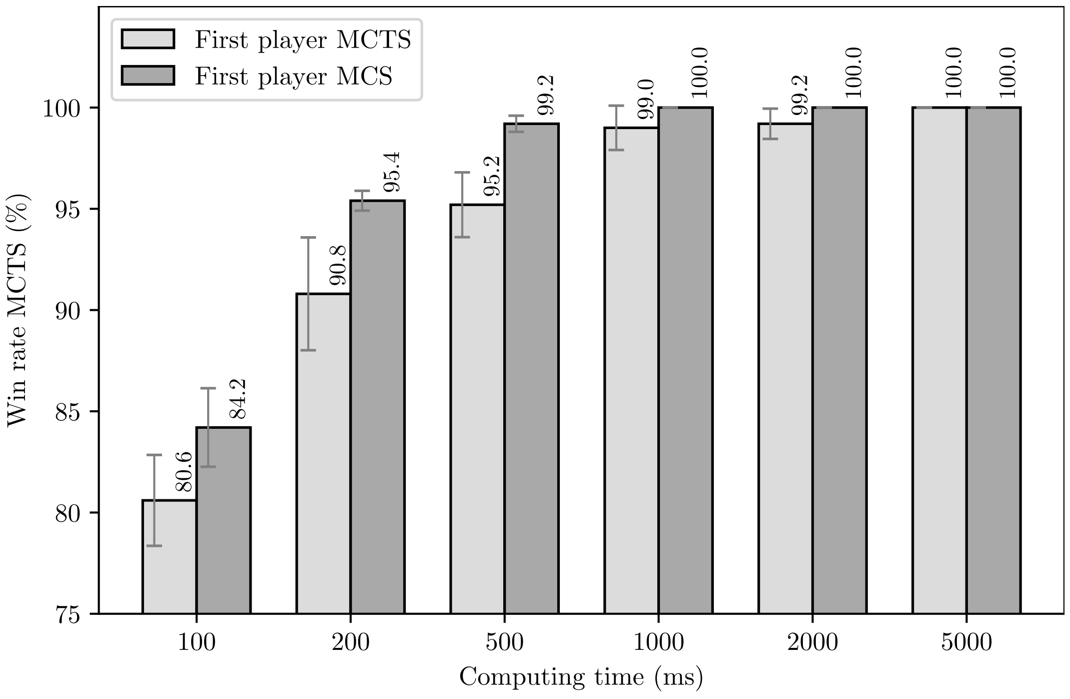

Figure 7.1 | Results of performance tests of the Monte Carlo Tree Search (MCTS) algorithm playing *Connect Four* against the Monte Carlo Search (MCS) algorithm. Tests with different conditions are conducted where the starting player (MCTS or MCS) and the amount of available computing time as limitation for the algorithms are varied. Five times 100 games are performed for each test. The average win rates of the MCTS are represented as bars in the chart. Their corresponding standard deviation is plotted as horizontal lines.

algorithm, the MCTS, wins less if it starts, which contradicts the conclusion *Allis* [11] that the win chances are the highest with a start in the middle column. According to the results, the opposite occurs as the MCTS wins more often as second player after the MCS made its first move selecting the middle column. One might expect that while the algorithms do not play perfectly, as required in the theorem, the approach that overall performs better still should tend to perform better especially when it gets the first move, which is not the case. This behavior could be caused by the impact of the early and the late game and the performances of the approaches during these phases or by their imperfect choice of moves and must be investigated more precisely for further analyses. Moreover, the standard error of the tests where the MCTS starts first is significantly higher than the standard error of those where the MCS makes the first move, which could be related to the behavior mentioned above.

7.1.2 Simulation Speed

In order to compare the tests concerning the simulation speed in consideration of the different set computing time limits, the simulation speed is specified in the unit *simulations per second*. This unit is calculated by dividing the set computing time limit by its respective achieved amount of simulations (see *Figure 7.2*).

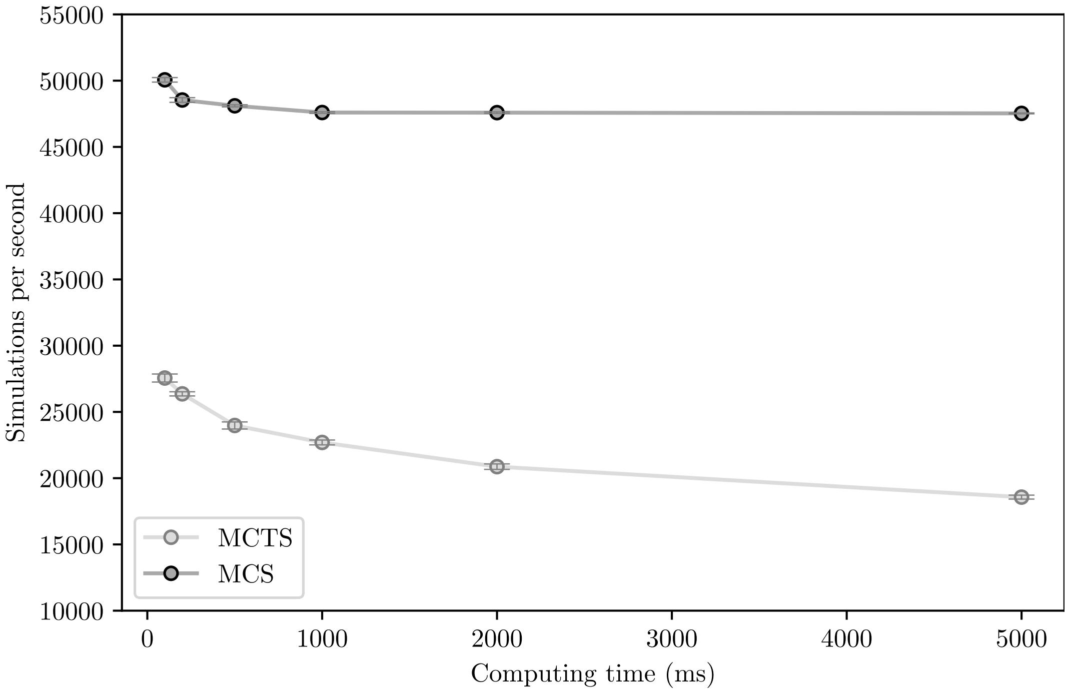

Figure 7.2 | Comparison of the amount of simulations the Monte Carlo Tree Search (MCTS) and the Monte Carlo Search (MCS) conducted per second during performance tests. The analysis of performance tests that differ in a computing time limitation show that there is a significant difference in the amount of simulations of the two tested search algorithms. For evaluation the amount of simulations conducted for a first move of each algorithm is used and transformed into *simulations per second*. With the values of each test (represented as dots in the chart) a characteristic curve is built which is presented in the graph. The corresponding standard deviation of the values is plotted as horizontal lines.

This resultant graph shows that the MCTS performs approximately two times less simulations than the MCS, which is due to its iterative tree building process. The decrease of the amount of simulations the MCTS can perform per second with increasing computing time can be explained by the fact that the longer the MCTS algorithm computes, the bigger the search tree grows. Thus, the branches get longer and so does the time that it takes to reach a leaf node where a simulation is conducted. The reason why the rate of decrease drops with higher computing time is the finite nature of *Connect Four*. When a branch reaches its maximum length, which is defined through the maximum amount of turns – 42 in this case – it cannot be expanded any further as a final game state is reached at this point. Consequently, the time it takes to traverse the tree up to a leaf node cannot exceed the time for a full traverse, which is why the slope of the MCTS's simulation speed seems to be monotonously increasing and converging to 0 for increasing computing time.

On the other hand, the simulation speed's decrease in the range of small computing times for the MCS algorithm can be attributed to overhead operations that are run once or at least only few times during a search, such as initialization or calculations. These operations have an impact on the average number of simulations per second when the computing time is much less than one second. When the computing time is higher and thus also the amount of simulations, the impact of the above mentioned operations approaches zero, which explains the nearly constant value of the MCS's curve for a computing time greater than one second.

7.2 CarloConnect versus humans

In this second evaluation phase, CarloConnect is placed against human opponents to see whether CarloConnect is capable to play on a human level. Three different difficulties are thereby tested to investigate a wide spectrum of the AI's strength. Therefore, the win rates of the human player and CarloConnect as well as the tie rate are calculated dividing the amount of the respective event by the total amount of games played for the set computing time – as done for the results of *Section 7.1.* The results including these rates of the games that are played on the website are presented in *Table 7.1.*

Table 7.1 | Win and tie rates of CarloConnect playing against human opponents. For three different computing times, games are played via the website (see *Section 6.7*), whereby in every played game, as an advantage for the human player, the human player starts first. The resulting win and tie rates are presented in this table.

Computing time (ms)	Total games played	Win rate (%), human player	Win rate (%), CarloConnect	Tie (%)
15	505	21.1	78.9	0.0
60	235	10.9	89.1	0.0
1500	536	5.2	94.4	0.4

With a computing time of only 15 ms (difficulty *Easy*), CarloConnect already wins about 79 % of the games. At the difficulty *Moderate*, which provides 60 ms for CarloConnect to search for a move, the win rate for the human players almost halves from 21.1 % (difficulty *Easy*) to 10.9 % –

CarloConnect wins approximately 10 % more games at this level of difficulty. From the difficulty *Moderate* to *Hard* the human players' win rate is slightly more than halved, again. Human players only win 5.2 % of the games at this stage, though rarely managing to cause a tie with a rate of 0.4 %. These results demonstrate that CarloConnect can compete against human players and even outperform most of them.

As for the tests against MCS, the more time is given, the stronger the MCTS based AI becomes. The time limit steps of the games of the website are big though, from *Easy* to *Moderate* a factor of 4 is applied, from *Moderate* to *Hard* a factor of 25. As a result, the human players' win rate always shrinks by half for each step whereby CarloConnect's win rate reaches 94.4 %. The factor that would be required to half the human players' win rate again can be expected to be large, thus leading to move searches that may take minutes.

In total 1276 games are played, whereby most of the games are played at the difficulty *Easy* and *Hard*. The high amount of played games at the first level (difficulty *Easy*) may be attributed to human players that do not defeat CarloConnect at this level and thus try it many times. Those who win, experienced players, try the next level and mostly, after some tries, manage to win there, too, going to the last level. At the last level, the difficulty is high enough to thwart those experienced players which hence have to play many games at this level to accomplish a defeat of CarloConnect.

The results of the additionally provided survey about the age of the players and their game feeling are shown in *Table 7.2*.

Table 7.2 | Short survey for human players that played against CarloConnect. In addition to the game of connect four, a short survey is provided on the website (see *Section 6.7*). The player's age and opinion about the game feeling is thereby listed.

Age	< 18	18-24	25-39	40-65	> 65	Total
Amount players	1	36	27	8	0	72
Normal game feeling	0	33	22	7	0	62
Artificial game feeling	1	3	5	1	0	10

Altogether 72 players submitted the survey, whereby most of them are of the age 18 to 39. Ten out of the 72 human opponents, about 14 %, noticed a strange behavior of the AI's playing. This may be caused by an unusual behavior of CarloConnect in the late game. In a progressed game state, where only few moves and thus options are left, when CarloConnect cannot win anymore and is sooner or later forced to make his move in a column that leads to a win for the human opponent, CarloConnect chooses to rather give up by making his next move in the column that opens a win opportunity for the human player instead of playing a few more defensive moves before being forced to make the game losing move. In *Figure 7.3* such a scenario is demonstrated.

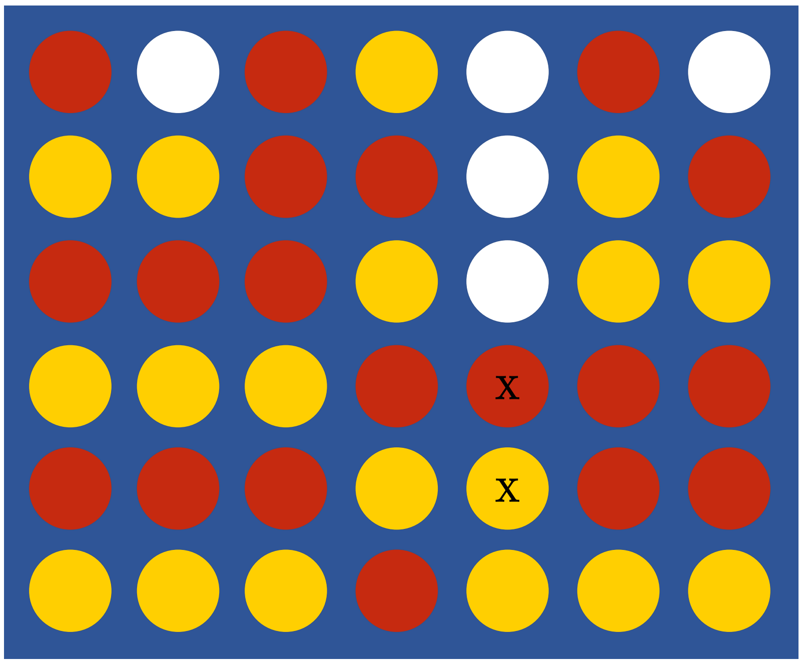

Figure 7.3 | Unexpected behavior of CarloConnect in progressed games. In the presented scenario *red* (the human player) wins with a horizontal combination. The last two moves are marked with an *X*. Instead of making a defense move either in the free slot on the top left side or on the top right side, *yellow* (CarloConnect) chooses to make its move in the column that opens a win opportunity for *red* instead.

Having the opportunity of selecting two other columns instead of the one that leads to a loss when it is the opponent's turn, CarloConnect, *yellow* in *Figure 7.3*, still selects the column that opens a win chance for the human player, *red* in *Figure 7.3*. This decision can be attributed to the possibilities that are given if the column that opens a win opportunity for *red* is selected first: even though *red* can win within the next move, *red* could also chose two other columns, thus there are three possible outcomes for the subsequent move. In the scenario of playing defensive first, two moves later *yellow* would end up in the same slot, having filled the other possible columns first. This leads to a single possible unavoidable outcome where *red* selects the column that leads to a win, as it is the only column left. In this respect, not playing defensive can be considered as better, but seems unusual and thus strange for a human player.

Chapter 8

Conclusion and outlook

In this study experiments are conducted to compare two Monte Carlo based approaches, MCS and MCTS, in order to investigate their performance in the game of *Connect Four*. The experiments evince that when it comes to a direct comparison, the MCTS outperforms the MCS due to a lack of tactical insight of the MCS, even though the total amount of simulations is comparatively low.

Concerning the behavior in performance of the MCTS, as expected the accuracy increases with the available computing time. Still, it remains unclear why the MCTS wins more often if its weaker opponent MCS starts first as this should provide a lower win chance for the MCTS according to the game solution found in the literature. In the future, further investigations can be carried out to find out whether this phenomenon is hard- or software based or if the win expectations for the second player do not depend on the first player's move in the case of imperfect playing.

Regarding the fact that the MCTS needs an amount of memory that depends on the number of nodes that are created during a search, the embedded system must fulfill the requirement of sufficient memory. In this work a Raspberry Pi 3 Model B is utilized which supplies 1 gigabyte of memory, which is why the MCTS can be used with no doubts and is thus the better approach to select. Common small computer platforms (like most of the Arduinos) often only provide little memory from 2 to 32 kilobytes, which would limit the MCTS in expanding the search tree, as each node of the tree is an instance that requires memory. This leads to a small possible amount of simulations which would result in a low accuracy of the search's result, wherefore the MCS would be the better choice.

Nevertheless, considering the strength of the MCTS, a computing platform that fulfils the requirement of the MCTS would be chosen, which is why the final program CarloConnect uses the MCTS based approach.

Using the website, many human players can play against CarloConnect simultaneously from anywhere. Thereby, many game results can be collectively gained within short time. Using the robotic arm, the same results can be achieved, but this approach takes longer as every game has to be accomplished one by one. Overall, the games against human opponents show that CarloConnect performs strong enough to reach a win rate of 94.4 % against human players having 1500 ms computing time. The required time in order to achieve a win rate of 100 % on a Raspberry Pi 3 Model B remains to be investigated.

The behavior of CarloConnect is mostly described as normal. 14 % of the players note a strange behavior that can be attributed to a different treatment of late game situations, where the loss of CarloConnect is clear. CarloConnect stops playing defensive at some point referring to probabilities and thus picking a move that leads to a loss, even if there are other columns available for selection. For future work, the behavior of CarloConnect could be further analysed by conducting games where CarloConnect plays against professional *Connect Four* players, whereby tactics could reveal other anomalies.

List of Figures

List of Tables

List of Abbreviations

AJAX	Asynchronous JavaScript and XML
API	Application programming interface
ASIC	Application-specific integrated circuit
AI	Artificial intelligence
COM	Computer on module
CPU	Central processing unit
DC	Direct current
DH	Denavit-Hartenberg
HTTP	Hypertext Transfer Protocol
IC	Integrated circuit
JSON	JavaScript Object Notation
MCS	Monte Carlo Search
MCTS	Monte Carlo Tree Search
MCU	Microcontroller unit
OpenMP	Open Multi-Processing
ORTS	Open Real Time Strategy
PCB	Printed circuit board
PWM	Pulse width modulation
RC servo motor	Radio control servo motor
RTS	Real-time strategy
SBC	Single board computer
SoC	System on a chip
SOM	System on module
TD	Temporal difference
TPU	Tensor Processing Unit

UCB1	Upper Confidence Bounds 1
UCT	UCB1 applied to trees

References

[1] S. Lawrence *et al.*, "Face recognition: a convolutional neural-network approach," *IEEE Transactions on Neural Networks*, vol. 8, no. 1, pp. 98–113, 1997. DOI: 10.1109/72.554195.

[2] J. L. Ticknor, "A Bayesian regularized artificial neural network for stock market forecasting," *Expert Systems with Applications*, vol. 40, no. 14, pp. 5501–5506, 2013. DOI: 10.1016/j.eswa.2013.04.013.

[3] J. Mycielski, "Chapter 3 Games with perfect information," in *Handbook of Game Theory with Economic Applications*, vol. 1, pp. 41–70, Elsevier, 1992. DOI: 10.1016/S1574-0005(05)80006-2.

[4] L. V. Allis, "Searching for Solutions in Games and Artificial Intelligence," *PhD theses, Univ. Limburg, Maastricht, The Netherlands*, 1994. ISBN: 90-9007488-0.

[5] J. Schaeffer, "The games computers (and people) play," in *Advances in Computers*, vol. 52, pp. 189–266, Elsevier, 2000. DOI: 10.1016/S0065-2458(00)80019-4.

[6] C. B. Browne *et al.*, "A Survey of Monte Carlo Tree Search Methods," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 4, no. 1, pp. 1–43, 2012. DOI: 10.1109/TCIAIG.2012.2186810.

[7] J. Rubin and I. Watson, "Computer poker: A review," *Artificial Intelligence*, vol. 175, no. 5-6, pp. 958–987, 2011. DOI: 10.1016/j.artint.2010.12.005.

[8] D. Silver *et al.*, "Mastering the game of Go with deep neural networks and tree search," *Nature*, vol. 529, no. 7587, pp. 484–489, 2016. DOI: 10.1038/nature16961.

[9] G. Tesauro and G. R. Galperin, "On-line Policy Improvement using Monte-Carlo Search," *The MIT Press: Advances in Neural Information Processing*, pp. 1068–1074, 1996. ISBN: 978-0262100656.

[10] R. Coulom, "Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search," in *Computers and Games*, vol. 4630, pp. 72–83, Berlin, Heidelberg: Springer Berlin Heidelberg, 2007. DOI: 10.1007/978-3-540-75538-8_7.

[11] V. Allis, "A Knowledge-Based Approach of Connect-Four: The Game Is Solved: White Wins," *ICGA Journal*, vol. 11, no. 4, p. 165, 1988. DOI: 10.3233/ICG-1988-11410.

[12] S. Edelkamp and P. Kissmann, "Symbolic Classification of General Two-Player Games," in *KI 2008: Advances in Artificial Intelligence*, vol. 5243, pp. 185–192, Berlin, Heidelberg: Springer Berlin Heidelberg, 2008. DOI: 10.1007/978-3-540-85845-4_23.

[13] L. Kanal and V. Kumar, *Search in Artificial Intelligence.* New York, NY: Springer New York, 1988. DOI: 10.1007/978-1-4613-8788-6.

[14] M. N. Katehakis and A. F. Veinott, "The Multi-Armed Bandit Problem: Decomposition and Computation," *Mathematics of Operations Research*, vol. 12, no. 2, pp. 262–268, 1987. DOI: 10.1287/moor.12.2.262.

[15] G. Gigerenzer and P. M. Todd, *Simple heuristics that make us smart.* Evolution and cognition, Oxford: Oxford Univ. Press, 1. issued as an oxford univ. press paperback ed., 1999. ISBN: 978-0-19-512156-8.

[16] M. H. Kalos and P. A. Whitlock, *Monte Carlo methods.* Weinheim: WILEY-VCH, 2., rev. and enl. ed ed., 2008. ISBN: 978-3-527-40760-6.

[17] K. G. Binmore, *Playing for real: a text on game theory.* Oxford, New York: Oxford University Press, 2007. ISBN: 978-0-19-530057-4.

[18] G. Chaslot *et al.*, "Monte-Carlo Tree Search: A New Framework for Game AI," in *Proc. Artif. Intell. Interact. Digital Entert. Conf.*, pp. 216–217, California: Stanford Univ., 2008. ISBN: 978-1-57735-391-1.

[19] P. Auer *et al.*, "Finite-time Analysis of the Multiarmed Bandit Problem," *Kluwer Academic Publishers*, vol. Mach. Learn., vol. 47, no. 2, pp. 235–256, 2002. DOI: 10.1023/A:1013689704352.

[20] L. Kocsis and C. Szepesvári, "Bandit Based Monte-Carlo Planning," in *Machine Learning: ECML 2006*, vol. 4212, pp. 282–293, Berlin, Heidelberg: Springer Berlin Heidelberg, 2006. DOI: 10.1007/11871842_29.

[21] G. M. J.-B. Chaslot *et al.*, "Progressive Strategies For Monte-Carlo Tree Search," *World Scientific Publishing Co.: New Mathematics and Natural Computation*, vol. 4, no. 3, pp. 343–357, 2008. DOI: 10.1142/S1793005708001094.

[22] M. J. Flynn and W. Luk, *Computer system design: system-on-chip.* Hoboken, N.J: Wiley, 2011. ISBN: 978-0-470-64336-5.

[23] R. Bermbach, *Embedded Controller: eine Einführung in Hard- und Software.* München: Hanser, 2001. ISBN: 978-3-446-19434-2.

[24] D. J. Russell, *Introduction to embedded systems: using ANSI C and the Arduino development evironment.* No. 30 in Synthesis lectures on digital circuits and systems, San Rafael, Calif.: Morgan & Claypool, 2010. ISBN:

978-1-60845-499-0.

[25] C. Platt, *Encyclopedia of electronic components.* Sebastopol, CA: Maker Media, 2013. ISBN: 978-1-4493-3418-5.

[26] N. Cameron, *Arduino Applied: Comprehensive Projects for Everyday Electronics.* Edinburgh, UK: Apress, 2019. DOI: 10.1007/978-1-4842-3960-5.

[27] R. Firoozian, *Servo motors and industrial control theory.* Heidelberg, New York, London: Springer, 2014. ISBN: 978-3-319-07275-3.

[28] S. P. Gimenez, *8051 microcontrollers: fundamental concepts, hardware, software and applications in electronics.* Springer, 2019. ISBN: 978-3-319-76439-9.

[29] A. Beni, "Kinematics of AdeptThree Robot Arm," in *Robot Arms*, InTech, 2011. DOI: 10.5772/17732.

[30] P. Corke, *Robot Arm Kinematics*, pp. 137–170. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011. DOI: 10.1007/978-3-642-20144-8_7.

[31] T. Dietz, *Industrielle Robotersysteme: Entscheiderwissen für die Planung und Umsetzung wirtschaftlicher Roboterlösungen.* S.l.: Morgan Kaufmann, 2019. ISBN: 978-3-658-25344-8.

[32] D. W. Wloka, *Robotersysteme 1.* Berlin, Heidelberg: Springer Berlin Heidelberg, 1992. DOI: 10.1007/978-3-642-93509-1.

[33] T. Ihme, "Dissertation: Steuerung von sechsbeinigen Laufrobotern unter dem Aspekt technischer Anwendungen," *Otto-von-Guericke-Universität Magdeburg*, p. 200, 2002. URL: https://d-nb.info/965010236/34 [04.02.2020].

[34] G. Legnani *et al.*, "A homogeneous matrix approach to 3D kinematics and dynamics — I. Theory," *Mechanism and Machine Theory*, vol. 31, no. 5, pp. 573–587, 1996. DOI: 10.1016/0094-114X(95)00100-D.

[35] D. Silver *et al.*, "Mastering the game of Go without human knowledge," *Nature*, vol. 550, no. 7676, pp. 354–359, 2017. DOI: 10.1038/nature24270.

[36] N. Jouandeau and T. Cazenave, "Small and Large MCTS Playouts Applied to Chinese Dark Chess Stochastic Game," in *Computer Games*, vol. 504, pp. 78–89, Cham: Springer International Publishing, 2014. DOI: 10.1007/978-3-319-14923-3_6.

[37] B. Arneson *et al.*, "Monte Carlo Tree Search in Hex," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 2, no. 4, pp. 251–258, 2010. DOI: 10.1109/TCIAIG.2010.2067212.

[38] G. Van den Broeck *et al.*, "Monte-Carlo Tree Search in Poker Using Expected Reward Distributions," in *Advances in Machine Learning*, vol. 5828, pp. 367–381, Berlin, Heidelberg: Springer Berlin Heidelberg,

2009. DOI: 10.1007/978-3-642-05224-8_28.

[39] A. Santos *et al.*, “Monte Carlo tree search experiments in hearthstone,” in *2017 IEEE Conference on Computational Intelligence and Games (CIG)*, (New York, NY, USA), pp. 272–279, IEEE, 2017. DOI: 10.1109/CIG.2017.8080446.

[40] I. Szita *et al.*, “Monte-Carlo Tree Search in Settlers of Catan,” in *Advances in Computer Games*, vol. 6048, pp. 21–32, Berlin, Heidelberg: Springer Berlin Heidelberg, 2010. DOI: 10.1007/978-3-642-12993-3_3.

[41] T. Pepels *et al.*, “Real-Time Monte Carlo Tree Search in Ms Pac-Man,” *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 6, no. 3, pp. 245–257, 2014. DOI: 10.1109/TCIAIG.2013.2291577.

[42] X. Guo *et al.*, “Deep Learning for Real-Time Atari Game Play Using Offline Monte-Carlo Tree Search Planning,” in *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2*, NIPS’14, (Cambridge, MA, USA), p. 3338–3346, MIT Press, 2014. URL: https://papers.nips.cc/book/advances-in-neural-information-processing-systems-27-2014 [26.01.2020].

[43] R.-K. Balla and A. Fern, “UCT for Tactical Assault Planning in Real-Time Strategy Games,” in *IJCAI 2009, Proceedings of the 21st International Joint Conference on Artificial Intelligence, Pasadena, California, USA, July 11-17, 2009*, pp. 40–45, 2009. ISBN: 978-1-57735-429-1.

[44] M. Naveed, D. Kitchin, and A. Crampton, “Monte-Carlo Planning for Pathfinding in Real-Time Strategy Games,” *PlanSig*, p. 8, 2010. ISSN: 1368-5708.

[45] G. Bertoletti, “Velena: A shannon c-type program which plays connect four perfectly,” 1997. URL: http://www.ce.unipr.it/%7Egbe/velsrc.html [26.01.2020].

[46] T. P. Runarsson and S. M. Lucas, “On imitating Connect-4 game trajectories using an approximate n-tuple evaluation function,” in *2015 IEEE Conference on Computational Intelligence and Games (CIG)*, (Tainan, Taiwan), pp. 208–213, IEEE, 2015. DOI: 10.1109/CIG.2015.7317961.

[47] S. Faußer and F. Schwenker, “Neural approximation of monte carlo policy evaluation deployed in connect four,” in *Artificial Neural Networks in Pattern Recognition*, (Berlin, Heidelberg), pp. 90–100, Springer Berlin Heidelberg, 2008. DOI: 10.1007/978-3-540-69939-2_9.

[48] M. Thill, S. Bagheri, P. Koch, and W. Konen, “Temporal difference learning with eligibility traces for the game connect four,” in *2014 IEEE Conference on Computational Intelligence and Games*, (Dortmund, Germany), pp. 1–8, IEEE, 2014. DOI: 10.1109/CIG.2014.6932870.

[49] R. S. Sutton, "Learning to predict by the methods of temporal differences," *Machine Learning*, vol. 3, no. 1, pp. 9–44, 1988. DOI: 10.1007/BF00115009.

[50] H. Baier and M. H. M. Winands, "MCTS-Minimax Hybrids," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 7, no. 2, pp. 167–179, 2015. DOI: 10.1109/TCIAIG.2014.2366555.

[51] N. P. Jouppi *et al.*, "In-Datacenter Performance Analysis of a Tensor Processing Unit," in *Proceedings of the 44th Annual International Symposium on Computer Architecture - ISCA '17*, (Toronto, ON, Canada), pp. 1–12, ACM Press, 2017. DOI: 10.1145/3079856.3080246.

[52] H. Jurzik, *Debian GNU/Linux: das umfassende Handbuch.* Rheinwerk Computing, Bonn: Rheinwerk Verlag GmbH, 6., aktualisierte auflage ed., 2015. ISBN: 978-3-8362-3764-2.

[53] D. Flanagan and L. Schulten, *JavaScript: das umfassende Referenzwerk ; [behandelt ECMAScript 5 & HTML5].* Köln: O'Reilly, 6. aufl ed., 2012. ISBN: 978-3-86899-135-2.