

Realisierung eines Vier-Gewinnt Roboters

ggf. Untertitel mit ergänzenden Hinweisen

Studienarbeit T3_3200

Studiengang Elektrotechnik

Studienrichtung Automation

Duale Hochschule Baden-Württemberg Ravensburg, Campus Friedrichshafen

von

Simon Gschell / Patrik Peters

| | |
|-----------------------|------------------------------|
| Abgabedatum: | 10. Juli 2025 |
| Bearbeitungszeitraum: | 01.01.2025 - 31.06.2025 |
| Matrikelnummer: | 123 456 |
| Kurs: | TEA22 |
| Betreuer: | Prof. Dr. ing Thorsten Kever |

Erklärung

gemäß Ziffer 1.1.14 der Anlage 1 zu §§ 3, 4 und 5 der Studien- und Prüfungsordnung für die Bachelorstudiengänge im Studienbereich Technik der Dualen Hochschule Baden-Württemberg vom 29.09.2017 in der Fassung vom 24.07.2023.

Ich versichere hiermit, dass ich meine Studienarbeit T3_3200 mit dem Thema:

Realisierung eines Vier-Gewinnt Roboters

selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Ich versichere zudem, dass die eingereichte elektronische Fassung mit der gedruckten Fassung übereinstimmt.

Friedrichshafen, den 10. Juli 2025

Simon Gschell / Patrik Peters

Kurzfassung

Diese Studienarbeit wurde im Rahmen der sechsten Akademiephase angefertigt. Im ersten Teil der Arbeit wurden verschiedene Spieltheorien miteinander verglichen. Aufbauend auf den Erkenntnissen der ersten Studienarbeit, lag der Schwerpunkt diesmal in der praktischen Umsetzung eines Vier-Gewinnt-Roboters. Das Ziel dieser Arbeit bestand darin, einen Roboter zu entwickeln, der eigenständig Spielzüge beim Spiel „Vier gewinnt“ ausführen kann und somit in der Lage ist, gegen einen anderen Roboter anzutreten.

Für die Realisierung des Projekts wurde ein LEGO Spike Prime Set verwendet. Die Konstruktion des Roboters besteht aus verschiedenen zusammengesetzten LEGO-Bauelementen. Vereinzelt wurden auch Elemente selbst entworfen und mittels 3D-Drucker angefertigt. Die Steuerung und Überwachung der Aktoren, wie zum Beispiel der Motoren und Sensoren, erfolgt mithilfe eines LEGO Spike Hub. Dieser Mikrocontroller verarbeitet die eingehenden Sensordaten und steuert die Bewegungen des Roboters entsprechend der programmierten Logik.

Die Programmierung erfolgte in der Sprache MircoPython in der LEGO Spike App. Das Kernstück des Programms ist der Alpha-Beta-Algorithmus, mit ihm wird der nächste optimale Zug berechnet. Durch die Kombination aus mechanischer Konstruktion und programmierter Software entstand ein funktionsfähiger Prototyp, der die gestellten Anforderungen erfüllt und einen Spielzug eigenständig ausführen kann.

Abstract

This student research project was carried out during the sixth academy phase. In the first part of the project, various game theories were examined and compared. Building on the insights from that initial work, the focus this time was on the practical development of a Four-in-a-Row robot. The goal was to create a robot capable of making its own moves in the game "Connect Four, allowing it to compete against another robot.

The project was implemented using a LEGO Spike Prime set. The robot itself was built from a range of LEGO components, with some parts specially designed and produced using a 3D printer. The motors and sensors are managed by a LEGO Spike Hub, which acts as the robot's microcontroller. This hub processes sensor data and directs the robot's movements according to the programmed logic.

Programming was done in MicroPython using the LEGO Spike app. At the heart of the software is the alpha-beta algorithm, which determines the best possible move at each turn. By combining mechanical design with custom software, the project resulted in a working prototype that meets the requirements and can play the game autonomously.

Inhaltsverzeichnis

| | | |
|----------|--|-----------|
| 1 | Einleitung | 1 |
| 2 | Grundlagen | 3 |
| 2.1 | Vier-Gewinnt | 3 |
| 2.2 | Alpha-Beta-Pruning-Algorithmus | 3 |
| 2.3 | Mikropython | 4 |
| 2.4 | LEGO Spike Komponenten | 5 |
| 2.4.1 | Hub | 5 |
| 2.4.2 | LEGO Spike Kraft- oder Touchsensor | 6 |
| 2.4.3 | LEGO Spike Farbsensor | 7 |
| 2.4.4 | LEGO Spike Winkelmotor | 8 |
| 3 | Vorgehen | 9 |
| 3.1 | Aufgabenpräzisierung | 9 |
| 3.2 | Anforderungen an den Roboter | 10 |
| 3.3 | Konzept | 10 |
| 3.4 | Zeitplan | 12 |
| 4 | Umsetzung | 13 |
| 4.1 | Mechanischer Aufbau | 13 |
| 4.1.1 | Einbindung von Aktorik | 13 |
| 4.1.2 | Einbindung von Sensorik | 14 |
| 4.2 | Software | 16 |
| 4.2.1 | Ablauf der Software | 16 |
| 4.3 | Programmlogik | 18 |
| 4.3.1 | Algorithmus | 19 |

| | | |
|----------|---|-----------|
| 4.4 | Hauptprogramm | 24 |
| 4.5 | Ressourcenschonende Implementierung der Spielsteuerung und Entscheidungslogik | 27 |
| 4.5.1 | Begrenzte Hardware-Ressourcen | 27 |
| 4.5.2 | Zeitbasierte Steuerung mit <code>time.sleep()</code> | 28 |
| 4.5.3 | Spielfeldvergleich zur Erkennung neuer Spielzüge | 28 |
| 4.5.4 | Spaltenweises Scannen statt Vollscan | 29 |
| 4.5.5 | Dynamische Suchtiefe im Algorithmus | 29 |
| 4.5.6 | Speicherung bewerteter Zustände (Transposition Table) | 30 |
| 4.5.7 | Minimale Boarddarstellung mit Ganzzahlen | 30 |
| 4.6 | Test und Versuchsauswertung | 31 |
| 4.6.1 | Durchführung und Ergebnisse der Testreihe | 31 |
| 4.6.2 | Analyse der Unentschieden und Niederlagen | 32 |
| | 4.6.2.1 Unentschieden durch Blockaden im Endspiel | 32 |
| | 4.6.2.2 Niederlagen durch fehlende Mehrzugererkennung | 32 |
| 5 | Zusammenfassung | 34 |
| | Literaturverzeichnis | 36 |
| | Abbildungsverzeichnis | 38 |
| | Tabellenverzeichnis | 39 |
| A | Komplettes Python-Programm | 40 |
| B | Nutzung von Künstliche Intelligenz basierten Werkzeugen | 49 |

1 Einleitung

Das allseits bekannte klassische Gesellschaftsspiel „Vier Gewinnt“ erfreut sich seit sehr vielen Jahren großer Beliebtheit bei Alt und Jung. Es ist ein strategisches Zweipersonenspiel mit sehr einfachen Regeln, aber mit einer erstaunlichen Spieltiefe. Im Zeitalter der Industrie 4.0 und Digitalisierung sollen immer mehr Abläufe automatisiert werden. Dadurch stellt sich die Frage, wie sich das Spiel „Vier Gewinnt“ mithilfe von Robotik und Algorithmus automatisieren lässt.

Im ersten Teil der Studienarbeit T3_3100 wurden zunächst die spieltheoretischen Grundlagen untersucht. Hierbei wurden zentrale Konzepte wie dominante Strategien, Nash-Gleichgewichte und das Minimax-Prinzip näher analysiert. Ein besonderes Augenmerk wurde auf die Entwicklung von Algorithmen gelegt. Der Fokus lag hierbei auf dem Alpha-Beta-Algorithmus.

Aufbauend auf der ersten Arbeit geht es in diesem Teil der Studienarbeit um den Entwurf und um die Realisierung eines Roboters, der eigenständig Spielzüge im Spiel „Vier Gewinnt“ ausführen kann. Ziel dieser Arbeit ist es, einen Roboter mithilfe von LEGO Spike Prime zu konstruieren und zu programmieren, der in der Lage ist, selbstständig das Spielfeld abzufahren und gegnerische Spielsteine zu erkennen. Auf diesen Fähigkeiten aufbauend soll der Roboter in der Lage sein, mithilfe eines Algorithmus den optimalen Spielzug zu berechnen und diesen eigenständig auszuführen.

Diese Arbeit beinhaltet sowohl mechanische und elektrische Entwicklung als auch das Entwerfen der Software. Ein besonderes wichtig hierbei ist das Zusammenspiel zwischen Hardware und Software, die es dem Roboter ermöglichen, flexibel und zuverlässig auf verschiedene Spielsituationen reagieren zu können.

Die Studienarbeit zielt auch darauf ab, eine Verbindung zwischen theoretischen Konzepten und der praktischen Anwendung in der Robotik zu demonstrieren. Sie ist dabei in folgende Kapitel unterteilt.

- **Grundlagen:** Im Kapitel „Grundlagen“ werden die zentralen Begriffe und Methoden vorgestellt.
- **Vorgehen:** Im zweiten Kapitel „Vorgehen“ wird auf die Anforderung an das System eingegangen. Darüber hinaus werden in diesem Abschnitt auch das Konzept und die Planung erläutert.
- **Umsetzung:** Im letzten Kapitel „Umsetzung“ geht es um die praktische Realisierung des Roboters. Hierbei wird auf den mechanischen Aufbau, sowie die Funktion der Software eingegangen.

2 Grundlagen

In diesem Kapitel werden die zentralen Begriffe und Methoden ausführlich vorgestellt. So wird das notwendige Grundlagenwissen vermittelt, auf dem die weitere Ausarbeitung aufbaut.

2.1 Vier-Gewinnt

Das Spiel Vier-Gewinnt wird auf einem Spielfeldraster mit sechs Zeilen und sieben Spalten gespielt. Zum Spielbeginn erhält jeder Spieler 21 Spielchips, entweder Rote oder Gelbe. Ziel des Spiels ist es, möglichst schnell vier Chips der eigenen Farbe in eine Reihe zu bringen – waagrecht, senkrecht oder diagonal. Die Spieler werfen abwechselnd ihre Chips in das Spielfeld, bis entweder ein Spieler das Ziel erreicht oder alle 42 Felder belegt sind [Has20].

2.2 Alpha-Beta-Pruning-Algorithmus

Alpha-Beta-Pruning ist ein Verfahren, das bei Spielen wie Schach, Dame oder Vier Gewinn eingesetzt wird, um den optimalen nächsten Zug zu bestimmen. Ziel des Algorithmus ist es, die Suche im Spielbaum effizienter zu machen. Im Unterschied zum Minimax-Algorithmus werden beim Alpha-Beta-Pruning gezielt Teilbäume weggelassen, die für das Endergebnis keine Rolle spielen. Während der Tiefensuche durch

den Spielbaum arbeitet der Algorithmus mit zwei Schrankenwerten, dem Alpha (α) und dem Beta (β). Zu Beginn wird Alpha auf $-\infty$ und Beta auf $+\infty$ gesetzt. An jedem MAX-Knoten wird das Maximum aus dem bisherigen Alpha und den Werten der Nachfolgeknoten ausgewählt. Das bedeutet, Alpha wird immer dann erhöht, wenn ein nachfolgender Knoten einen höheren Wert liefert als das aktuelle Alpha. Am MIN-Knoten hingegen wird Beta jeweils auf das Minimum aus dem bisherigen Beta und den Werten der Nachfolgeknoten gesetzt. Sobald an einem Knoten die Bedingung $\alpha \geq \beta$ erfüllt ist, wird der restliche Teilbaum nicht weiter betrachtet. In diesem Fall hat der MIN bereits eine bessere Alternative gefunden, sodass MAX diesen Zweig des Baums nicht mehr wählen würde [Ado09].

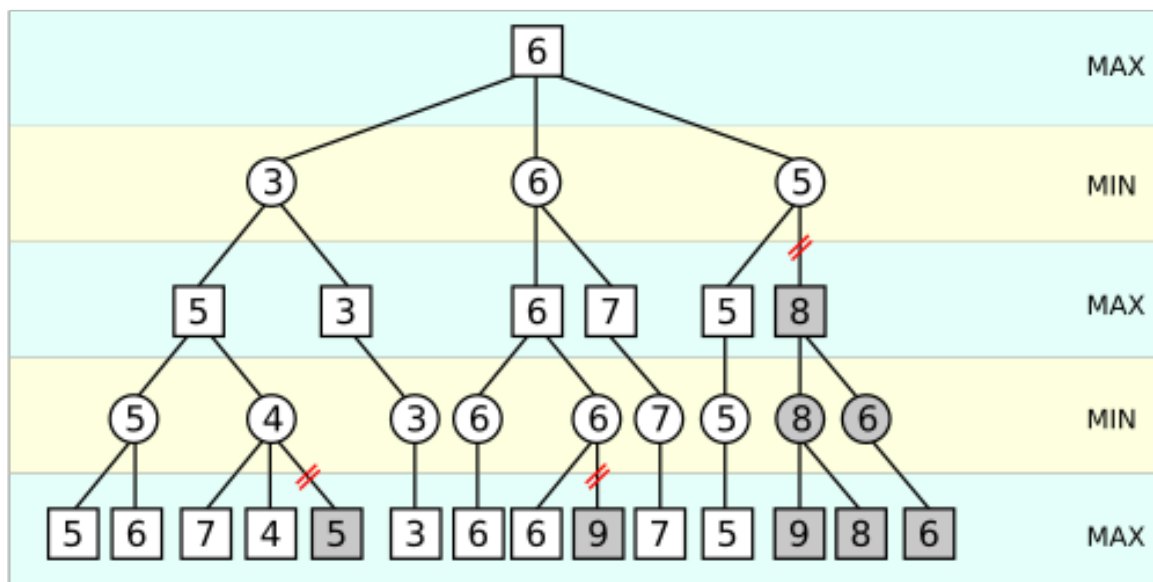


Abbildung 2.1: Alpha-Beta Spielbaum

2.3 Mikropython

MicroPython ist eine speziell für Mikrocontroller angepasste Version der Programmiersprache Python. Im Gegensatz zur Desktop-Variante lässt sich MicroPython-Code direkt auf Hardware mit begrenzten Ressourcen ausführen[SS23][PLJ23]. Im Unterschied zu Standard-Python 3 bringt MicroPython allerdings nur einen Teil der

gewohnten Python-Standardbibliotheken mit.

Wodurch es weniger Speicherplatz benötigt. Zudem besitzt MicroPython einen eigenen Interpreter, der direkt auf einem Mikrokontroller ausgeführt werden kann. Dadurch eignet sich MicroPython besonders gut für die Programmierung des LEGO Spike Hubs[Bel24].

2.4 LEGO Spike Komponenten

In diesem Kapitel werden die zentralen LEGO Spike-Komponenten beschrieben, die beim Bau des Roboters verwendet wurden.

2.4.1 Hub

Der LEGO Spike Hub ist das Herzstück des LEGO Spike Prime Sets. Er dient als programmierbare Steuereinheit mit sechs LPF2 input/output ports, an die alle LEGO-Sensoren und -Motoren angeschlossen werden können. Im Inneren arbeitet ein eigener Prozessor (100 MHz ARM Cortex-M4), unterstützt von 320 KB RAM und 1 MB Flash-Speicher. Die Programmierung des LEGO Spike Hubs erfolgt in der Sprache MicroPython. LEGO stellt dafür eine eigene Entwicklungsumgebung (IDE) bereit, mit dieser der Hub einfach programmiert werden kann. Hierfür kann der Hub über USB oder via Bluetooth mit dem Computer verbunden werden. Die Steuereinheit wird über einen wiederaufladbarer Lithium-Ionen-Akku mit Strom und Spannung versorgt [LEG20b].

Weitere technische Merkmale des LEGO Spike Hubs sind:

- Individuell anpassbaren Lichtmatrix (5x5)
- Lautsprecher

- Taster mit integrierter Statusleuchte
- Tasten für die Navigation und Steuerung durch Menüs am Hub
- Lautsprecher
- sechssachsiger Gyrosensor

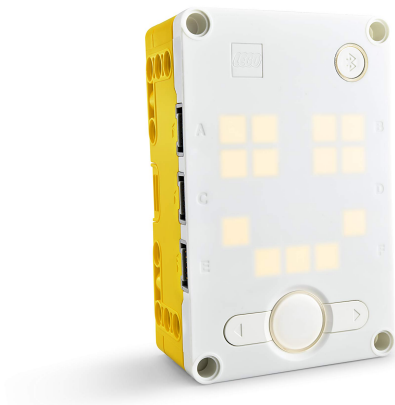


Abbildung 2.2: LEGO Spike Hub

2.4.2 LEGO Spike Kraft- oder Touchsensor

Dieser Sensor erkennt, ob er gedrückt wurde, und misst dabei gleichzeitig die auf ihn ausgeübte Kraft. Mit einer Abtastrate von 100 Hz erfasst er Kräfte im Bereich von 2,5 bis 10 Newton und arbeitet dabei mit einer Genauigkeit von $\pm 0,65$ Newton. Der gemessene Wert wird als Prozentwert ausgegeben, wobei 100% einem Tastendruck von 10 Newton entsprechen. Typischerweise wird der Sensor zum Erkennen von Hindernissen oder als Start- bzw. Stopptaste eines Roboters eingesetzt. Der Kraft- oder Touchsensor wird direkt am LEGO Spike Hub angeschlossen [LEG20c].



Abbildung 2.3: LEGO Spike Kraft- oder Touchsensor

2.4.3 LEGO Spike Farbsensor

Dieser elektronische Farbsensor wurde speziell für LEGO Spike entwickelt. Seine Abtastrate beträgt 1 kHz und er kann direkt am Hub angeschlossen werden. Der Sensor kann bis zu acht verschiedene Farben erkennen, darunter Schwarz, Blau, Rot, Weiß, Braun, Gelb, Pink und Grün. Außerdem misst er sowohl die Intensität des reflektierten Lichts als auch die des Umgebungslichts [LEG20a][LEG20c].

Für die Farberkennung erfasst der Sensor die Farbwerte sowohl im RGB- (Rot, Grün, Blau) als auch im HSV-Farbraum (hue = Farbton, saturation = Sättigung, value = Helligkeit). Die Messergebnisse werden als Ganzzahlen ausgegeben [LEG20a].

Zur Reflexionsmessung sendet der Sensor weißes Licht auf eine Oberfläche und misst das zurückgeworfene Licht. Diese Funktion wird häufig für Linienführung eingesetzt [Bet25][LEG20a].



Abbildung 2.4: LEGO Technic Farbsensor

2.4.4 LEGO Spike Winkelmotor

Der LEGO Spike Winkelmotor ist nicht nur ein einfacher Elektromotor, aufgrund eines integrierten Drehsensors kann er nicht nur die Drehrichtung, sondern auch die relative und absolute Position (in Grad) sowie die Drehgeschwindigkeit erfassen. Eine vollständige Umdrehung wird dabei in 360 einzelne Zählimpulse unterteilt, wobei die Genauigkeit des Motors bei ± 3 Grad liegt. Muss der Motor ein Drehmoment von mehr als 5 Ncm aufbringen, blockiert er. Mit einer Abtastrate von 100 Hz erfasst der Motor die Position sowohl beim automatischen als auch im manuellen Betrieb. Der LEGO Spike Winkelmotor eignet sich somit nicht nur für Bewegungsaufgaben, sondern auch zur Positionsbestimmung [LEG20c].



Abbildung 2.5: LEGO Spike Winkelmotor

3 Vorgehen

Im folgendem Kapitel wird auf die Planung des Vier-Gewinnt-Roboters eingegangen. Zunächst werden die Anforderungen definiert. Anschließend werden verschiedene Möglichkeiten genauer betrachtet und miteinander verglichen.

3.1 Aufgabenpräzisierung

Für das Projekt soll mit LEGO Spike Prime ein Roboter entwickelt werden, der Spielzüge im Spiel Vier-Gewinnt vollkommen eigenständig gegen einen menschlichen Gegner oder einen anderen Roboter spielen kann. Der Roboter muss dazu in der Lage sein, die auf dem Spielfeld platzierten gelben und roten Spielsteine zuverlässig zu erkennen und deren Positionen zu erfassen, um das gesamte Spielfeld systematisch auswerten zu können.

Nach Abschluss des Scanvorgangs berechnet der Roboter mithilfe eines passenden Algorithmus die nächste optimale Position. An der ermittelten Stelle platziert er anschließend eigenständig den nächsten Spielstein. Es ist wichtig, dass der Roboter flexibel ist und sowohl mit gelben als auch mit roten Steinen spielen kann. Außerdem muss es möglich sein, dass der Roboter das Spiel entweder eröffnet oder als zweiter Spieler startet.

Für jeden Spielzug gilt eine maximale Zeitvorgabe von 90 Sekunden, die nicht überschritten werden darf. Der Roboter zieht sich nach jedem Zug komplett vom Spielfeld zurück und wartet darauf, dass der Gegner seinen Spielzug komplett abgeschlossen hat. Erst danach startet der Roboter erneut mit dem Scannen des Spielfelds und der darauf folgenden Berechnung des nächsten Zuges.

3.2 Anforderungen an den Roboter

In der Tabelle 3.1 sind die Anforderungen an den Vier-Gewinnt-Roboter in einer Anforderungsliste zusammengetragen. Dabei wird zwischen Forderungen und Wünschen unterschieden. Anforderungen, die mit **F** gekennzeichnet sind, müssen unbedingt umgesetzt werden. Während hingegen Anforderungen, die mit **W** markiert sind, als Wünsche zu verstehen sind und nicht zwingend im System realisiert werden müssen.


Tabelle 3.1: Anforderungstabelle für einen Vier-Gewinnt-Roboter

| Nr. | Anforderung an das System | F/W |
|-----|--|-----|
| 1 | komplettes Spielfeld scannen | F |
| 2 | das Ende des Spiels erkennen | F |
| 3 | autonom fahren | F |
| 4 | Begrenzungen des Spielfelds erkennen | F |
| 5 | Steine selber platzieren | F |
| 6 | immer nur ein Stein pro Spielzug | F |
| 7 | Abwarten, bis der Gegner seinen Zug beendet hat | F |
| 8 | nach jedem Zug rechts vom Spielfeld wegfahren | F |
| 9 | maximal 90 Sekunden pro Spielzug | F |
| 10 | optimalen Spielzug berechnen | F |
| 11 | es sollte möglich sein sowohl mit Gelb als auch Rot zu spielen | F |
| 12 | sowohl als Erster als auch als Zweiter zu starten | F |
| 13 | während des gegnerischen Spielzugs warten | F |
| 14 | Scannen, nur bis ein neuer gegnerischer Stein erkannt wurde | W |
| 15 | volle Spalten überspringen | W |
| 16 | wenn ein leerer Platz erkannt wurde, zur nächsten Spalte | W |

Legende: **F** = Forderung, **W** = Wunsch,

3.3 Konzept

Nachdem die Anforderungsliste erstellt wurde, folgt im nächsten Schritt die Ausarbeitung eines Konzepts. Um die vielen verschiedenen Möglichkeiten übersichtlich und strukturiert darzustellen, wurde hierfür der Ansatz eines morphologischer Kas-

ten gewählt. Dieser ist in der Tabelle 3.2 dargestellt. Durch diese Herangehensweise können unterschiedliche Kombinationen von Lösungsansätzen untersucht und miteinander verglichen werden. Dadurch kann die beste Lösung für die Realisierung gefunden werden. In der Tabelle 3.2 ist diese durch eine rote Linie dargestellt .

| Merkmale | Variante 1 | Variante 2 | Variante 3 |
|-------------------------|------------------|-------------------------|-----------------------|
| Berechnung | auf dem Computer | auf dem Controller | |
| Rückmeldung | Computer | LED-Matrix | akustisches Signal |
| Programmiersprache | MicroPython | Scratch | |
| Algorithmus | Zufall | Minimax/Alpha-Beta | Vorprogrammierte Züge |
| Verbindung zum PC | Bluetooth | USB-Kabel | |
| Art der Startbetätigung | vom Computer | Kraftsensor | Taste am Hub |
| Spielfeld-Erkennung | Farbsensor | manuelle Eingabe | |
| Steinplatzierung | Greifarm | Fallmechanismus/Rutsche | Förderband |

Tabelle 3.2: Morphologischer Kasten

3.4 Zeitplan

Im Zeitplan werden die einzelnen Arbeitsschritte strukturiert und organisiert grafisch dargestellt. Hierfür wird für jede einzelnen Arbeitsaufgaben ein Zeitraum festgelegt. Dadurch kann der Fortschritt besser verfolgt werden. Ein Zeitplan ist bei einer Projektarbeit ein unerlässliches Werkzeug, um die Arbeit effizient und zielgerecht durchführen zu können.

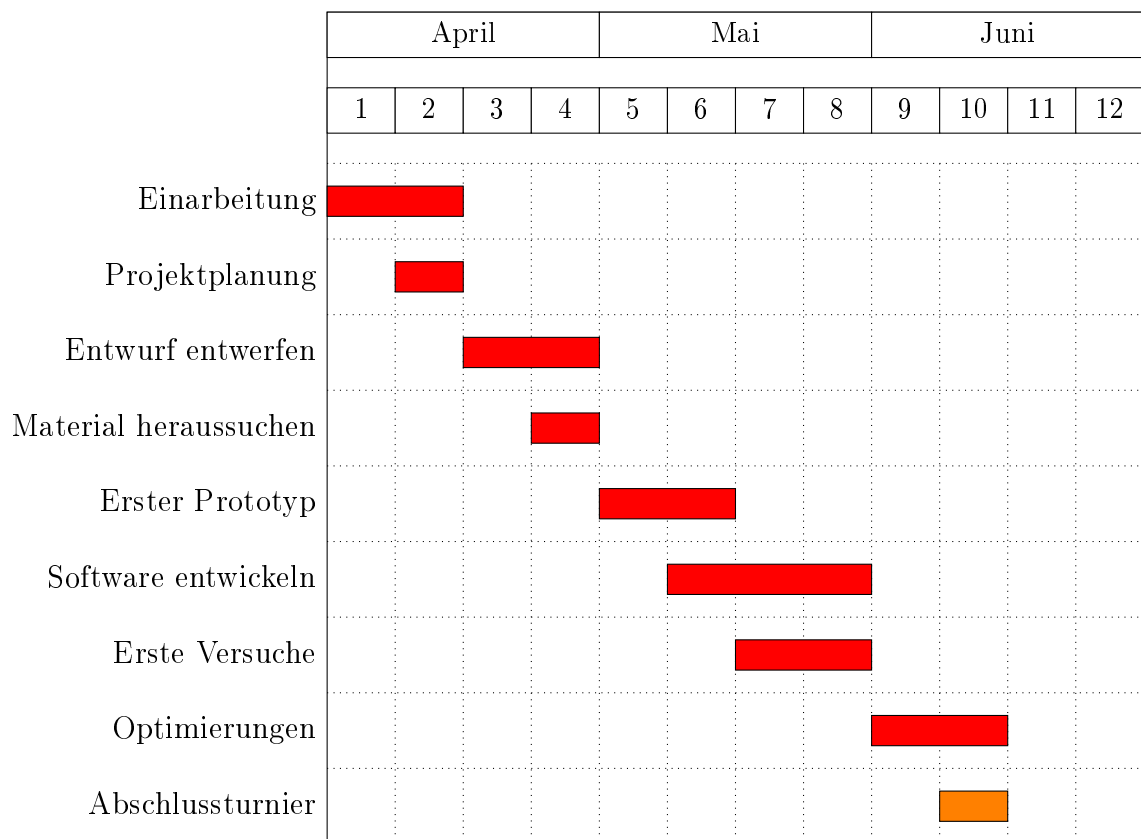


Tabelle 3.3: Zeitplan

4 Umsetzung

In diesem Kapitel steht die praktische Umsetzung des Roboters im Fokus. Dabei wird auf den Aufbau eingegangen, ebenso wie auf die Funktionsweise der Software.

4.1 Mechanischer Aufbau

Für die Umsetzung des 4-Gewinnt-Roboters wurde eine mechanische Konstruktion gewählt, die es erlaubt, das Spielfeld zu scannen sowie Chips gezielt in eine Spalte einzuwerfen. Der Aufbau umfasst drei Winkelmotoren, einen Farbsensor und einen Drucktaster. Im Folgenden werden die einzelnen Komponenten detailliert beschrieben. Dabei beziehen sich die Ziffern der Aufzählung der einzelnen Sensoren und Aktoren auf die Abbildung 4.1 und Abbildung 4.2.

Die mechanische Konstruktion basiert auf die Idee einem kartesischen Koordinatensystem, bei dem der Farbsensor durch die Kombination aus horizontaler und vertikaler Bewegung jede Spielfeldposition präzise anfahren kann. Das System erlaubt eine vollautomatische Spielweise.

4.1.1 Einbindung von Aktorik

1. Horizontalantrieb

Der horizontale Antrieb des Farbsensors erfolgt über einen Winkelmotor. Dieser

ist dafür zuständig, die Spielfeldspalten nacheinander anzufahren. Der Motor ist mit einer Achse verbunden, welche zwei Räder antreiben. Die Bewegung erfolgt in gleichmäßigen Schritten: Eine Drehung um exakt 72 Grad bewegt den Roboter um eine Spalte weiter. Diese Schrittweite wurde so gewählt, dass sie der Breite einer Spalte im Spielfeld entspricht. Dadurch ist eine exakte Positionierung des Sensors über jeder Spalte möglich, ohne dass zusätzliche Sensoren zur Positionsbestimmung notwendig sind.

2. Vertikalantrieb

Um das Spielfeld auch in vertikaler Richtung abfahren zu können, ist der Farbsensor an einer Kette montiert. Diese Kette wird durch einen Winkelmotor angetrieben. Der Sensor ist an einem mittleren Segment der Kette befestigt und fährt beim Drehen der Kette entsprechend auf und ab. Ein Schritt des Motors um 95 Grad bewegt den Sensor um genau eine Spielfeldhöhe weiter. Auf diese Weise können sämtliche sechs Reihen der aktuellen Spalte nacheinander abgescannt werden. Die Rückwärtsbewegung der Kette erlaubt es, den Sensor wieder nach unten zu fahren.

3. Chipauswerfer

Das Einwerfen des eigenen Spielsteins erfolgt ebenfalls über einen Winkelmotor. An diesem Motor ist eine Stange montiert, die bei einer vollständigen Umdrehung einen Spielchip aus dem Vorratsmagazin (mit der Software Fusion360 konstruiert und 3D-gedruckt) in die gewünschte Spalte stößt. Nach der Auslösung kann ein neuer Chip in die Abschussposition nachrutschen. In der Software ist eine Wartezeit nach dem Auslösen eingebaut, damit der Chip sicher im Spielfeld ankommt, bevor die nächste Aktion beginnt.

4.1.2 Einbindung von Sensorik

4. Startsignal

Um dem Roboter mitzuteilen, dass er den nächsten Zug starten kann wurde ein Kraftsensor angebracht. Dieser befindet sich an der Vorderseite des Roboters.

Sobald der Spieler den Sensor leicht berührt, wird ein Signal ausgelöst und der Prozess startet.

5. Spielfeldscan

Für die Farberkennung des Spielfeldes wurde ein LEGO-Farbsensor verwendet, der über die oben beschriebene Kettenkonstruktion vertikal verfahrbar ist. Die Farbmessung erfolgt jeweils in der Mitte eines Spielfeldes. Der Sensor erkennt RGB-Werte (in diesem Projekt benutzt: Rot, Gelb oder Leer). Der Abstand zwischen Sensor und Spielfeld beträgt etwa 7 mm. Dieser Wert hat sich als optimal für zuverlässige Farbmessung erwiesen.

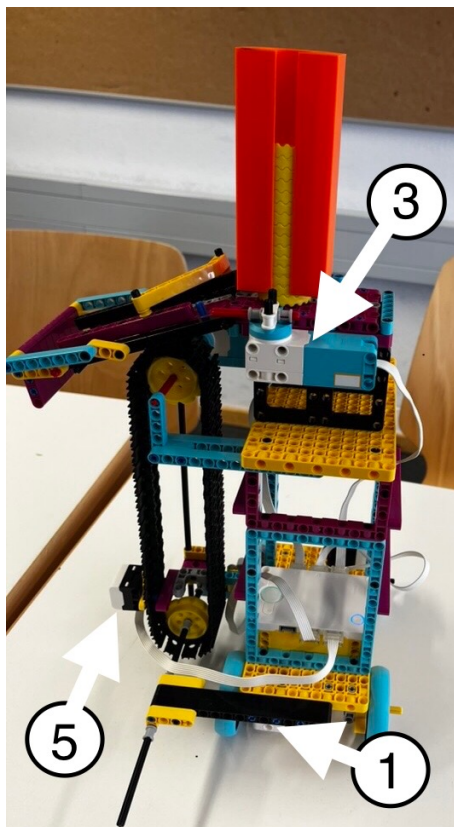


Abbildung 4.1: Seitenansicht links

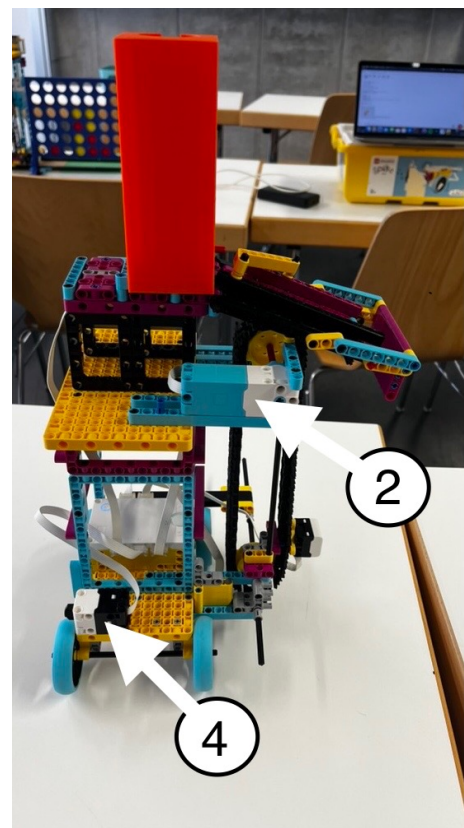


Abbildung 4.2: Seitenansicht rechts

Legende:

- 1 = Motor für horizontale Bewegung,
- 2 = Motor für vertikale Bewegung,
- 3 = Motor für Steineinwurf,
- 4 = Kraft-/Touchsensor 5 = Farbsensor,

4.2 Software

Das Kapitel Software beschreibt ausführlicher, wie der Vier-Gewinnt-Roboter programmiert wird. Die Software ist das Herzstück des Roboters und spielt eine entscheidende Rolle für seine autonome Teilnahme am Spiel.

4.2.1 Ablauf der Software

Das in der Abbildung 4.3 dargestellte Flussdiagramm zeigt grafisch den groben Ablauf der Software.

Zu Beginn befindet sich das System im Wartezustand und wartet auf die Betätigung des Touch-/Kraftsensors. Dieser signalisiert den Start eines neuen Spielzugs. Sobald der Sensor aktiviert wurde, fährt der Roboter zur ersten Spalte des Spielfelds. Anschließend scannt das System die einzelnen Spalten des Spielfelds ab, um den neuen gegnerischen Stein zu entdecken. Dabei wird das interne Spielfeld aktualisiert, bis ein neuer gegnerischer Stein erkannt wurde oder alle 7 Spalten gescannt wurden.

Im nächsten Schritt prüft die Software das interne Spielfeld und berechnet mithilfe des Alpha-Beta-Algorithmus den optimalen Spielzug. Dabei ist es nicht von Bedeutung, ob ein gegnerischer Stein erkannt wurde oder nicht. Da zu Beginn des Spiels noch gar keine Steine im Feld vorhanden sind. Der Roboter platziert nach der Berechnung den eigenen Spielstein an der berechneten optimalen Position.

Im Anschluss prüft die Software, ob das Spiel gewonnen wurde. Ist das der Fall, so wird eine Glückwunschkmeldung und ein Ton ausgegeben. Falls kein Gewinn vorliegt, wird einfach das aktuelle Spielfeld auf der Konsole in der LEGO Spike App ausgegeben und der Prozess beginnt von vorne in der Warteposition mit dem Warten auf die nächste Sensoraktivierung.

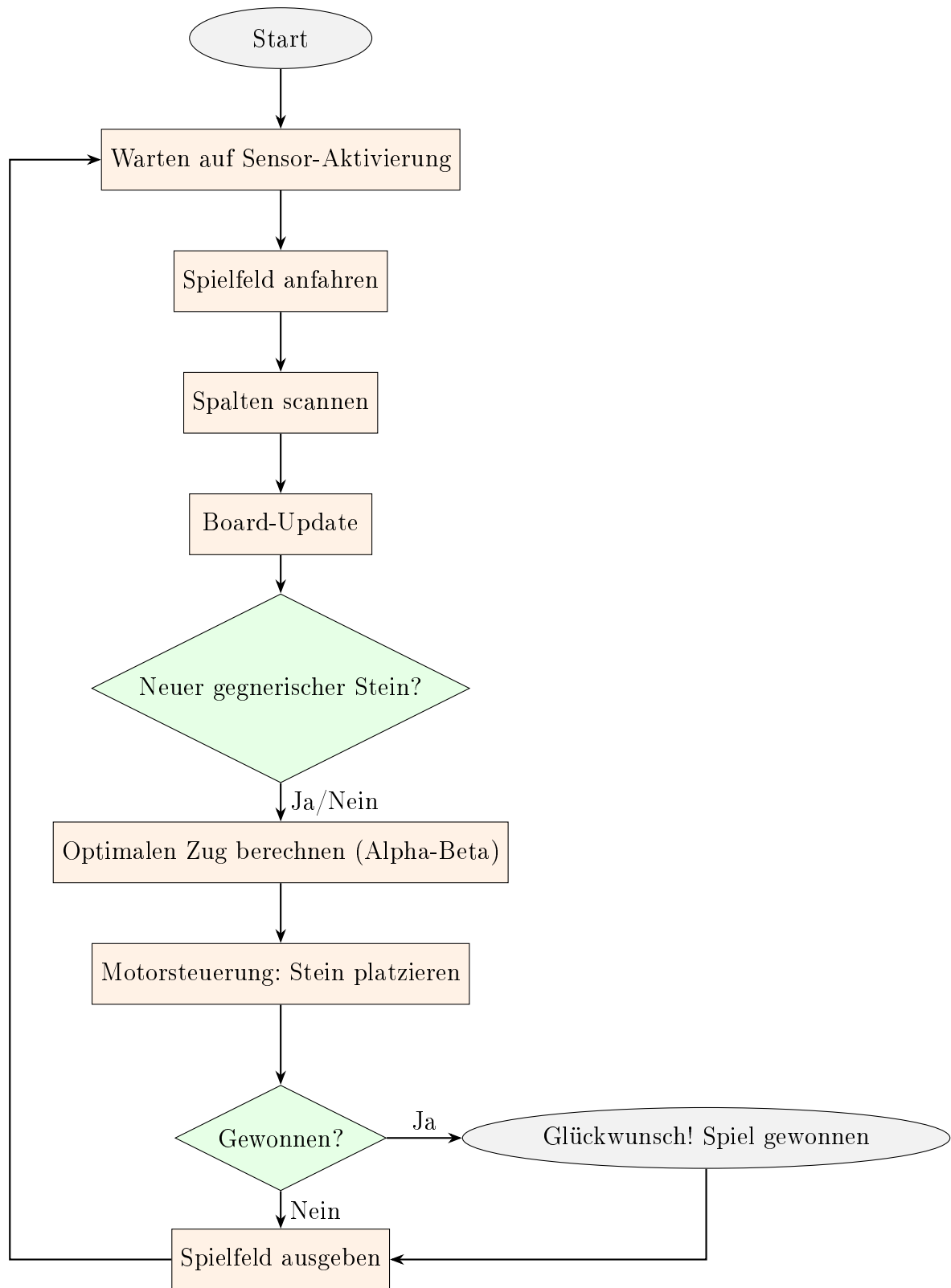


Abbildung 4.3: Flussdiagramm der Software

4.3 Programmlogik

In diesem Kapitel wird die Softwarestruktur des 4-Gewinnt-Roboters systematisch beschrieben. Da der vollständige Quellcode eine Vielzahl an Funktionen, Hilfsroutinen und technischen Details umfasst, wird in diesem Kapitel aus Gründen der Übersichtlichkeit nicht jede einzelne Codezeile dargestellt und erläutert. Stattdessen liegt der Fokus auf den wesentlichen Programmabschnitten, die das Spielverhalten maßgeblich bestimmen. Der vollständige Code kann aus dem Anhang entnommen werden.

Zur besseren Nachvollziehbarkeit der Funktionsweise wird die Darstellung in zwei Teile gegliedert:

- Zunächst wird ausschließlich der **Algorithmus** zur Spielentscheidung detailliert erläutert. Dabei handelt es sich um den Minimax-Algorithmus mit Alpha-Beta-Pruning.
- Im Anschluss wird das **Hauptprogramm** vorgestellt, das alle Bestandteile miteinander verknüpft. Es steuert den Ablauf über das Spielfeld-Scanning bis zum Berechnen und Ausführen des eigenen Spielzugs. Dabei werden sowohl Sensoren als auch Motoren angesprochen und der Entscheidungsalgorithmus eingebunden.

Diese Zweiteilung erlaubt es, sowohl die algorithmische Tiefe als auch die technische Umsetzung getrennt zu betrachten und anschließend im Gesamtkontext zu verstehen.

Desweiteren wird im Unterkapitel 4.7 die konkrete Umsetzung der Spielsteuerung und Entscheidungslogik für den 4-Gewinnt-Roboter beschrieben. Der Fokus liegt auf der softwareseitigen Realisierung unter Berücksichtigung der limitierten Hardware-Ressourcen des LEGO Spike Prime Hubs. Es werden hierbei zentrale Entwurfsentscheidungen erläutert.

4.3.1 Algorithmus

Ein zentraler Bestandteil des 4-Gewinnt-Roboters ist die Entscheidungsfindung durch einen algorithmischen Spielbaum. Dieser wird mit dem bekannten Minimax-Algorithmus unter Verwendung von Alpha-Beta-Pruning realisiert. Ziel ist es, basierend auf dem aktuellen Spielfeldzustand den optimalen Zug für den Roboter zu berechnen.

Der Algorithmus bewertet mögliche Züge bis zu einer bestimmten Tiefe im Spielbaum und trifft Entscheidungen, die langfristig zum Sieg führen können oder gegnerische Gewinnzüge verhindern.

Spielfeld und Spielerdefinition:

Im Vorfeld des Algorithmus ist festgelegt, welches Symbol der Algorithmus (Roboter) spielt:

```
1 my_piece = 1 # 1 = YELLOW (AI player), -1 = RED
2 opponent_piece = -my_piece
```

Dabei entspricht 1 dem gelben Spielstein (Roboter), -1 dem roten Spielstein (Gegner). Diese numerische Darstellung vereinfacht die Bewertung und das Vergleichen der Felder im Spielfeld.

Bewertungsfunktion:

Der Algorithmus benötigt eine Bewertungsfunktion, die die Qualität eines Spielzustands abschätzt. Dies geschieht durch eine Heuristik, die mögliche Gewinnlinien zählt und bewertet. Die Bewertungsfunktion basiert auf der Idee, sogenannte „Fenster“ (Ausschnitte aus 4 Feldern) im Spiel zu analysieren und zu beurteilen, wie viele Steine der Roboter bzw. des Gegners in diesen Fenstern enthalten sind:

```
1  def evaluate_window(window, player):
2      opp_player = opponent_piece
3      if player == my_piece else my_piece
4          score = 0
5      if window.count(player) == 4:
6          score += 100
7      elif window.count(player) == 3 and window.count(0) == 1:
8          score += 5
9      elif window.count(player) == 2 and window.count(0) == 2:
10         score += 2
11     if window.count(opp_player) == 3 and window.count(0) == 1:
12         score -= 4
13     return score
```

Diese Funktion bewertet sowohl offensive als auch defensive Situationen. Ein Fenster mit drei eigenen Steinen und einem leeren Feld wird positiv bewertet, ein Fenster mit drei gegnerischen Steinen und einem leeren Feld hingegen negativ, um Bedrohungen abzuwehren.

Die Hauptfunktion zur Bewertung des gesamten Spielfeldes aggregiert alle horizontalen, vertikalen und diagonalen Fenster:

```
1  def evaluate(board):
2      score = 0
3      center_array = [board[r][field_width//2]
4      for r in range(field_height)]
5          center_count = center_array.count(my_piece)
6          score += center_count * 3
```

Zunächst werden die mittleren Spalten stärker gewichtet, da sie strategisch wichtiger sind.

Anschließend werden alle Zeilen, Spalten und Diagonalen analysiert:

```
1     for r in range(field_height):
2         row_array = [board[r][c] for c in range(field_width)]
3     for c in range(field_width - 3):
4         window = row_array[c:c+4]
5         score += evaluate_window(window, my_piece)
6         score -= evaluate_window(window, opponent_piece)
```

Diese Schleifen bilden das heuristische Fundament für die spätere Entscheidungsfindung.

Minimax mit Alpha-Beta-Pruning:

Die Hauptentscheidung trifft der Minimax-Algorithmus. Dabei wird rekursiv der Spielbaum aufgebaut, wobei sich der Algorithmus abwechselnd in die Rolle des Roboters („maximizing player“) und des Gegners („minimizing player“) versetzt. Um die Effizienz zu steigern, wird Alpha-Beta-Pruning genutzt. Dabei werden Äste im Spielbaum verworfen, wenn sie nachweislich zu schlechteren Ergebnissen führen.

Der Einstiegspunkt ist:

```
1     def alpha_beta(board, depth, alpha, beta, maximizing_player)
```

Zuerst wird geprüft, ob der aktuelle Zustand bereits im Transposition Tabelle gespeichert ist. Das ist ein Cache zur Vermeidung redundanter Berechnungen:

```
1         key = (board_hash(board, maximizing_player), depth)
2         if key in transposition_table:
3             return transposition_table[key]
```

Anschließend erfolgt eine Prüfung: Ist der Zug eine Gewinnsituation, oder wurde die maximale Tiefe erreicht?

```
1     valid_locations = [col for col in range(field_width)
2         if is_valid_location(board, col)]
3         terminal = winning_move(board, my_piece) or winning_move
4         (board, opponent_piece) or len(valid_locations) == 0
5     if depth == 0 or terminal:
6         ...
```

Falls ja, gibt die Funktion eine Bewertung zurück. Andernfalls wird der Spielbaum weiter durchlaufen.

Maximierender Spieler (Roboter):

Hier versucht der Algorithmus, die maximal erreichbare Bewertung zu finden und prüft regelmäßig, ob das aktuelle Ergebnis besser ist als die bisherige beste Option. Wenn $\alpha \geq \beta$, wird der restliche Baum abgeschnitten (Pruning).

```
1     if maximizing_player:
2         value = -float('inf')
3         for col in valid_locations:
4             ...
5         new_score = alpha_beta(..., False)[1]
6         if new_score > value:
7             value = new_score
8             best_col = col
9             alpha = max(alpha, value)
10    if alpha >= beta:
11        break
```

Minimierender Spieler (Gegner):

Analog erfolgt das Vorgehen für den Gegner:

```
1     else:
2         value = float('inf')
3         for col in valid_locations:
4             ...
5         new_score = alpha_beta(..., True)[1]
6         if new_score < value:
7             value = new_score
8             best_col = col
9             beta = min(beta, value)
10        if beta <= alpha:
11            break
```

Am Ende wird das Ergebnis in der Transpositionstabelle gespeichert und zurückgegeben:

```
1     transposition_table[key] = result
2     return result
```

Dynamische Suchtiefe:

Je nach Spielphase kann es sinnvoll sein, tiefer oder flacher zu suchen. Zu Beginn reicht eine niedrige Tiefe, da viele Züge möglich sind. In späteren Phasen erhöht sich die Tiefe:

```
1     def get_dynamic_depth(board):
2         empty = sum(row.count(0) for row in board)
3         if empty > 30:
4             return 3
5         else:
6             return 4
```

Diese dynamische Anpassung balanciert Spielstärke und Rechenzeit optimal.

4.4 Hauptprogramm

Nachdem der Algorithmus erläutert wurde und in Kapitel 4.1 die Konstruktion und Ansteuerung des Roboters aufgezeigt wurde, beschreibt dieses Kapitel den Gesamtlauf des Programms. Dabei steht im Fokus, wie die Spielfeldererkennung, Algorithmus und Ausführungsschritte zu einem vollständigen Spielzug kombiniert werden.

Initialisierung:

Zu Beginn wird das Spielfeld als leere Matrix angelegt. Zusätzlich wird eine Kopie gespeichert, um Änderungen im Vergleich zur vorherigen Runde erkennen zu können.

```
1 board = [[0 for _ in range(field_width)]
2 for _ in range(field_height)]
3 last_board = [row[:] for row in board]
```

Warten auf Eingabe durch den Spieler:

Bevor der Roboter mit dem Scannen des Spielfeldes beginnt, wartet er auf eine Aktivierung des Drucksensors am Port C.

```
1 print("Waiting for sensor at port C...")
2 while not sensor_activated():
3     time.sleep(0.1)
```

Positionierung an der Startspalte:

Der Roboter fährt an die rechte Spielfeldseite (Spalte 0), um von dort den Scan zu beginnen.

```
1 motor.run_for_degrees(port.D, 198, 170)
2 time.sleep(1.5)
```

Scannen des Spielfelds:

Von rechts nach links wird jede Spalte analysiert. Dabei wird der Farbsensor in die erste freie Zeile der Spalte bewegt:

```
1     motor.run_for_degrees(port.E, move_distance_e * (free_row),
    speed_E)
2     detected_color = color_sensor.color(port.B)
3     update_board(free_row, matrix_col, detected_color)
4     motor.run_for_degrees(port.E, -move_distance_e * free_row,
    speed_E)
```

Wird ein neuer gegnerischer Spielstein erkannt, wird seine Position gespeichert und der Scan abgebrochen:

```
1     if (last_board[free_row][matrix_col] == 0 and
2         board[free_row][matrix_col] == opponent_piece):
3         opponent_piece_found = True
4         opponent_col = col
```

Berechnung des Spielzugs:

Die Tiefe der Suche wird dynamisch abhängig vom Spielstand gewählt. Anschließend wird der beste Spielzug mit Minimax und Alpha-Beta-Pruning berechnet:

```
1     dynamic_depth = get_dynamic_depth(board_numeric)
2     best_col, _ = alpha_beta(
3         board_numeric,
4         depth=dynamic_depth,
5         alpha=-float('inf'),
6         beta=float('inf'),
7         maximizing_player=(my_piece == 1)
8     )
```


Ausführen des Spielzugs:

Zuerst wird die physische Zielspalte berechnet und der Roboter dorthin bewegt:

```
1     physical_target_col = field_width - 1 - best_col
2     motor.run_for_degrees(port.D, -move_distance_d *
    physical_target_col, speed_D)
```

Danach wird ein Spielstein mithilfe des Motors A ausgeworfen:

```
1     motor.run_for_degrees(port.A, -360, speed_A)
2     time.sleep(3)
```

Das Spielfeld wird nach dem Wurf aktualisiert:

```
1     board[best_row][best_col] = my_piece
2     last_board = [row[:] for row in board]
```

Anschließend erfolgt eine Prüfung auf einen möglichen Spielsieg:

```
1     if winning_move(board, my_piece):
2         print(" Congratulations! The robot has WON the game!")
3         print_board(board)
4         sound.beep(440, 1000000, 100)
5         break
```

Zurückfahren in Ausgangsposition:

Unabhängig vom Spieldesign kehrt der Roboter an seine Startposition zurück:

```
1 motor.run_for_degrees(port.D, -199, speed_D)
```

Warten auf die nächste Runde:

Abschließend wird auf das Loslassen des Drucksensors gewartet, bevor ein neuer Zyklus beginnt:

```
1 while sensor_activated():  
2     time.sleep(0.1)
```

4.5 Ressourcenschonende Implementierung der Spielsteuerung und Entscheidungslogik

Die Implementierung des Spielablaufs und der Entscheidungslogik wurde unter besonderer Berücksichtigung der eingeschränkten Ressourcen des LEGO Spike Prime Hub entwickelt. Durch gezielte Reduktion von unnötigen Berechnungen, Verwendung eines Gedächtnisses für das Spielfeld, dynamische Anpassung der Suchtiefe und einfache Ablaufsteuerung konnte ein vollständiger Spielzyklus umgesetzt werden, der sowohl strategisch leistungsfähig als auch technisch robust ist. Im Folgenden werden die zentralen Entwurfsentscheidungen aufgezeigt.

4.5.1 Begrenzte Hardware-Ressourcen

Der LEGO Spike Hub besitzt mit seinem 100MHz ARM Cortex-M4 Prozessor, 320 KB RAM und 1 MB Flash-Speicher eine stark begrenzte Hardwareausstattung [LEG20b]. Diese Ressourcen reichen für einfache Steuerungsaufgaben, setzen aber dem Einsatz komplexer Algorithmen wie Minimax enge Grenzen. Diese Rahmenbedingungen erfordern eine möglichst effiziente und ressourcenschonende Programmstruktur. Daher

wurde bewusst auf eine komplexe Multithread-Struktur verzichtet und stattdessen ein sequenzieller, wartender Ablauf gewählt.

4.5.2 Zeitbasierte Steuerung mit `time.sleep()`

Zur Koordination zwischen Sensorik, Motorik und internen Berechnungen wurde `time.sleep()` gezielt eingesetzt. Es erfüllt mehrere Aufgaben:

- **Sicherstellung der mechanischen Stabilität:** Nach jeder Bewegung oder Farberkennung sorgt eine kurze Pause dafür, dass der Sensor sich mechanisch beruhigen kann und stabile Werte liefert.
- **Hardware-Synchronisierung:** Vorgänge, wie etwa das Einwerfen eines Chips, benötigen eine kurze Wartezeit, die hardwareseitig nicht automatisch rückgemeldet wird. Durch gezielte Pausen wird so ein zuverlässiger Ablauf ohne ungewollte Überschneidungen erreicht.
- **Einfachheit:** In Abwesenheit von Interrupts auf dem Hub ist `time.sleep()` eine praktikable Lösung zur Ablaufsteuerung.

4.5.3 Spielfeldvergleich zur Erkennung neuer Spielzüge

Ein wesentlicher Optimierungsschritt liegt in der Verwendung eines “Gedächtnisses” über das vorherige Spielfeld. Zu Beginn jeder Spielrunde wird die aktuelle Matrix `board` mit dem gespeicherten Zustand `last_board` verglichen. Ziel ist es, festzustellen, wo genau ein neuer gegnerischer Spielstein hinzugekommen ist, ohne jedes einzelne Feld vollständig neu scannen zu müssen:

```
1  if last_board[zeile][spalte] == 0 and board[zeile][spalte]  
    == opponent_piece:
```

Dieser Vergleich erlaubt es, gezielt den neuen Zug des Gegners zu erkennen und den Scanvorgang direkt danach abubrechen. Dies reduziert die benötigte Zeit pro Runde drastisch. Insbesondere im späteren Spielverlauf, wenn viele Felder bereits belegt sind.

4.5.4 Spaltenweises Scannen statt Vollscan

Anstatt das gesamte Spielfeld (6 Zeilen \times 7 Spalten) vollständig zu scannen, wird nur von rechts nach links spaltenweise geprüft. Sobald ein neuer Spielstein entdeckt wurde, wird der Scan abgebrochen:

```
1     if opponent_piece_found:
2         break
```

Diese Strategie basiert auf der Annahme, dass pro Runde exakt ein neuer gegnerischer Stein erscheint. Dadurch kann der Großteil des Spielfelds übersprungen werden, sobald der neue gegnerische Zug erkannt wurde. Dies reduziert die Anzahl der Motorbewegungen und gewinnt somit an Zeit.

4.5.5 Dynamische Suchtiefe im Algorithmus

Der Minimax-Algorithmus mit Alpha-Beta-Pruning wird verwendet, um den optimalen eigenen Spielzug zu berechnen. Um die Rechenlast dabei zu steuern, wird die maximale Suchtiefe dynamisch an die Spielsituation angepasst:

```
1     def get_dynamic_depth(board):
2         empty = sum(row.count(0) for row in board)
3         return 3 if empty > 30 else 4
```

In der Anfangsphase sind noch viele Züge möglich, was den Suchbaum exponentiell wachsen lässt. Eine flachere Suchtiefe (z. B. 3) ist hier sinnvoll, da es ohnehin viele gleichwertige Optionen gibt. Im Endspiel hingegen sind nur noch wenige Felder frei, wodurch eine tiefere Suche (z. B. 4) möglich und auch sinnvoll wird. Diese dynamische Anpassung balanciert Rechenzeit und Spielqualität optimal.

4.5.6 Speicherung bewerteter Zustände (Transposition Table)

Zur weiteren Reduktion der Rechenlast wird eine sogenannte Transposition Table eingesetzt. Diese speichert bereits bewertete Spielzustände in einer Hash-Tabelle, sodass doppelt auftretende Konstellationen nicht erneut berechnet werden müssen:

```
1     key = (board_hash(board, maximizing_player), depth)
2     if key in transposition_table:
3         return transposition_table[key]
```

Diese Technik ist besonders im mittleren Spielverlauf effektiv, da viele unterschiedliche Zugfolgen zu identischen Spielzuständen führen können.

4.5.7 Minimale Boarddarstellung mit Ganzzahlen

Das Spielfeld wird intern als Liste von Ganzzahlen (-1, 0, 1) dargestellt. Diese Codierung ist speicherarm, ermöglicht arithmetische Operationen (z. B. Summieren zur Bewertung) und reduziert die Komplexität beim Kopieren und Vergleichen des Boards.

4.6 Test und Versuchsauswertung

Zur Überprüfung der Funktionalität und Spielstärke des entwickelten 4-Gewinnt-Roboters wurde eine umfangreiche Testreihe mit menschlichen Mitspielern durchgeführt. Ziel dieser Versuche war es, das Verhalten des Roboters in realen Spielsituationen zu beobachten, die Zuverlässigkeit der Spielfeldererkennung zu bewerten und die Qualität des Entscheidungsalgorithmus praktisch zu prüfen.

4.6.1 Durchführung und Ergebnisse der Testreihe

Der Roboter wurde in der finalen Version gegen vier unterschiedliche Spieler getestet. Es wurden insgesamt 20 vollständige Partien gespielt. Die menschlichen Spieler handelten eigenständig und spielten mit realem Gewinninteresse.

Die Resultate der Testreihe lauten wie folgt:

- **14 Siege des Roboters**
- **4 Unentschieden**
- **2 Niederlagen gegen menschliche Spieler**

Damit konnte der Roboter in 70% der Partien gewinnen und blieb in 90% der Fälle ungeschlagen.

4.6.2 Analyse der Unentschieden und Niederlagen

Insgesamt sechs Partien wurden nicht gewonnen. Diese lassen sich in vier Unentschieden und zwei Niederlagen unterteilen. Beide Fälle sind technisch nachvollziehbar und lassen sich mit den Eigenschaften des Algorithmus sowie den physikalischen Begrenzungen des Systems erklären.

4.6.2.1 Unentschieden durch Blockaden im Endspiel

In vier Partien endete das Spiel unentschieden: Beide Spieler konnten keine vier Spielsteine mehr in eine Reihe bringen, das Spielfeld war komplett gefüllt. Die Ursache lag dabei nicht in einem technischen Fehler, sondern im Spielverhalten des Roboters. Er verhinderte konsequent alle potenziellen Gewinnchancen des Gegners, agierte aber gleichzeitig zu passiv, um selbst eine entscheidende Reihe aufzubauen.

Der Algorithmus spielte in diesen Situationen eher defensiv und wählte im Mittelspiel häufig sichere, ausgeglichene Positionen, anstatt gezielt einen eigenen Angriff vorzubereiten. Dadurch entstand ein blockiertes Spiel, in dem letztlich keiner der beiden Spieler gewinnen konnte.

4.6.2.2 Niederlagen durch fehlende Mehrzugererkennung

Zwei Partien wurden verloren, weil der Algorithmus eine mehrstufige Kombination des Gegners nicht rechtzeitig erkannte. Ursache ist die eingeschränkte Suchtiefe zu Beginn des Spiels:

```
1  def get_dynamic_depth(board):  
2      empty = sum(row.count(0) for row in board)  
3      return 3 if empty > 30 else 4
```

In frühen Spielphasen prüft der Algorithmus nur drei Züge voraus, um Rechenzeit zu sparen. Dadurch übersieht er unter Umständen mehrphasige Angriffsstrategien vom Gegner. Ein Spieler nutzte diese Gelegenheit und platzierte seine Spielsteine so, dass der Roboter einen drohenden Vierer erst bemerkte, als keine Abwehr mehr möglich war.

5 Zusammenfassung

Im Rahmen dieser Arbeit wurde ein autonomer 4-Gewinnt-Roboter entwickelt, der in der Lage ist, selbstständig gegen menschliche Spieler oder einen anderen Roboter anzutreten. Ziel war es, ein System zu entwerfen, das sowohl spieltheoretisch fundierte Entscheidungen trifft als auch physisch in der Lage ist, Spielzüge eigenständig umzusetzen.

Als Grundlage diente die in einem ersten Schritt entwickelte Spieltheorie, auf der ein Minimax-Algorithmus mit Alpha-Beta-Pruning aufgebaut wurde. Dieser bewertet mögliche Züge im Voraus und trifft Entscheidungen, die langfristig zum Sieg führen oder gegnerische Gewinnchancen verhindern. Der Algorithmus wurde so angepasst, dass er auf die begrenzten Ressourcen des LEGO Spike Prime Hubs abgestimmt ist. Dazu zählt unter anderem eine dynamische Anpassung der Rechentiefe abhängig vom Spielverlauf, um eine stabile und reaktionsschnelle Umsetzung zu ermöglichen.

Ergänzt wurde der Algorithmus durch eine passende mechanische Konstruktion. Der Roboter kann mithilfe eines Farbsensors das Spielfeld erkennen, neue gegnerische Spielsteine lokalisieren und über Motoren präzise den eigenen Spielstein platzieren. Ein Kraftsensor dient als einfacher, intuitiver Startmechanismus für den nächsten Spielzug.

In einem abschließenden Test wurde das System in 20 Partien gegen verschiedene menschliche Spieler geprüft. Dabei konnte der Roboter 14 Spiele gewinnen, 4-mal ein Unentschieden erzielen und wurde nur 2-mal geschlagen. Diese Ergebnisse zeigen, dass das System weitgehend erfolgreich funktioniert und auch in realen Spielsituationen zuverlässig arbeitet. Die Niederlagen waren dabei vor allem auf die bewusst begrenzte

Rechentiefe im frühen Spielverlauf zurückzuführen.

Besonders positiv ist hervorzuheben, dass der entwickelte Roboter auch gegen andere studentische Projekte antreten konnte und dabei ebenfalls als Sieger hervorging. Das zeigt, dass sowohl die theoretische als auch die technische Umsetzung konkurrenzfähig und durchdacht war.

Insgesamt lässt sich festhalten, dass das gesteckte Ziel erreicht wurde. Der Roboter verbindet erfolgreich die theoretischen Grundlagen mit praktischer Anwendung. Als Ausblick bieten sich weiterführende Ansätze wie eine noch tiefere Spielfeldauswertung, verbesserte Heuristiken oder auch ein lernfähiger Algorithmus an, um die Spielstärke künftig weiter zu erhöhen.

Literaturverzeichnis

- [Ado09] Julius Adorf. *Adversariale Suche für optimales Spiel: Der Minimax-Algorithmus und die Alpha-Beta-Suche*. Proseminararbeit. Betreuer: Lars Kunze, Dominik Jain. Abgabetermin: 2. Dezember 2009. München: Technische Universität München, Fakultät für Informatik, Forschungs- und Lehrereinheit Informatik IX, 2009. URL: <https://www.juliusadorf.com/pub/alphabeta-seminar-paper.pdf>.
- [Bel24] Charles A. Bell. *MicroPython for the Internet of Things: A Beginner's Guide to Programming with Python on Microcontrollers*. English. 2nd. Berkeley, CA: Apress, 2024. ISBN: 9781484298619.
- [Bet25] Betzold GmbH. *LEGO Education SPIKE Technic Farbsensor*. Zugriff am 04.07.2025. 2025. URL: https://www.betzold.de/prod/E_761144/.
- [Has20] SA Hasbro. *Das Originale 4Gewinnt Anleitung*. Hasbro Gaming, 2020.
- [LEG20a] LEGO Education. *Technical Specifications: Technic Color Sensor*. https://assets.education.lego.com/v3/assets/blt293eea581807678a/blt62a78c227edef070/5f8801b9a302dc0d859a732b/techspecs_techniccolorsensor.pdf?locale=en-us. Accessed: 2025-07-04. 2020.
- [LEG20b] LEGO Education. *Technical Specifications: Technic Large Hub*. https://assets.education.lego.com/v3/assets/blt293eea581807678a/bltf512a371e82f6420/5f8801baf4f4cf0fa39d2feb/techspecs_techniclargehub.pdf?locale=en-us. Zugriff am 03.07.2025. 2020.
- [LEG20c] LEGO Education Community. *Exploring SPIKE™ Prime Sensors*. Accessed: 2025-07-04. 2020. URL: <https://community.legoeducation.com/blogs/31/220>.

- [PLJ23] Ignas Plauska, Agnius Liutkevičius und Audronė Janavičiūtė. „Performance Evaluation of C/C++, MicroPython, Rust and TinyGo Programming Languages on ESP32 Microcontroller“. In: *Electronics* 12.1 (2023). Open Access Article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license, S. 143. DOI: 10.3390/electronics12010143. URL: <https://www.mdpi.com/2079-9292/12/1/143>.
- [SS23] Albin Samefors und Felix Sundman. *Investiating Energy Consumption and Responsiveness of low power modes in MicroPython for STM32WB55*. Bachelor's thesis. 15 hp (first-cycle education). Jönköping, Sweden, Juni 2023. URL: <https://www.diva-portal.org/smash/get/diva2:1778426/FULLTEXT01.pdf>.

Abbildungsverzeichnis

| | | |
|-----|--|----|
| 2.1 | Alpha-Beta Spielbaum Quelle: [wikimediaABpruning] | 4 |
| 2.2 | LEGO Spike Hub Quelle:[LEG20c] | 6 |
| 2.3 | LEGO Spike Kraft- oder Touchsensor Quelle:[LEG20c] | 7 |
| 2.4 | LEGO Technic Farbsensor Quelle:[LEG20c] | 7 |
| 2.5 | LEGO Spike Winkelmotor Quelle:[LEG20c] | 8 |
| 4.1 | Seitenansicht links | 15 |
| 4.2 | Seitenansicht rechts | 15 |
| 4.3 | Flussdiagramm | 17 |

Tabellenverzeichnis

| | | |
|-----|--|----|
| 3.1 | Anforderungstabelle für einen Vier-Gewinnt-Roboter | 10 |
| 3.2 | Morphologischer Kasten | 11 |
| 3.3 | Zeitplan | 12 |
| 2.1 | Liste der verwendeten Künstliche Intelligenz basierten Werkzeuge . . . | 49 |

A Komplettes Python-Programm

```
1  import motor
2  import color_sensor
3  import color
4  import time
5  import force_sensor
6  from hub import port, sound
7
8  # — CONFIGURATION —
9  my_piece = 1 # -1 = RED, 1 = YELLOW (AI player)
10 opponent_piece = -my_piece
11
12 speed_D = 230
13 speed_A = 500
14 speed_E = 600
15 move_distance_e = 97 # Step motor E per row, calibrate if necessary!
16 move_distance_d = 73
17 break_motor_e = 1
18 break_motor_e_back = 2
19 break_motor_d = 1
20 field_width = 7
21 field_height = 6
22 waiting_line = 1 # Waiting time after reaching the line (in seconds)
23
24 # — HELPER FUNCTIONS —
25 def sensor_activated():
26     try:
27         return force_sensor.force(port.C) > 30
28     except:
29         return False
30
```

```
31 def update_board(row, col, detected_color):
32     if detected_color in (color.RED, color.PURPLE, color.MAGENTA):
33         board[row][col] = -1
34     elif detected_color in (color.YELLOW, color.WHITE, color.GREEN):
35         board[row][col] = 1
36     else:
37         board[row][col] = 0
38
39 def print_board(board):
40     symbol_map = {-1: "R", 1: "Y", 0: "B", None: "B"}
41     print("\nCurrent board (bottom right = [0][0]):")
42     for row_idx in reversed(range(field_height)):
43         row = board[row_idx]
44         print(" ".join(symbol_map.get(cell, "?") for cell in row))
45
46 def is_valid_location(board, col):
47     return board[field_height - 1][col] == 0
48
49 def get_next_open_row(board, col):
50     for r in range(field_height):
51         if board[r][col] == 0:
52             return r
53     return None
54
55 def winning_move(board, piece):
56     for c in range(field_width - 3):
57         for r in range(field_height):
58             if board[r][c] == piece and board[r][c + 1] == piece and board[r][
59 c + 2] == piece and board[r][c + 3] == piece:
60                 return True
61     for c in range(field_width):
62         for r in range(field_height - 3):
63             if board[r][c] == piece and board[r + 1][c] == piece and board[r
64 + 2][c] == piece and board[r + 3][c] == piece:
65                 return True
66     for c in range(field_width - 3):
67         for r in range(field_height - 3):
68             if board[r][c] == piece and board[r + 1][c + 1] == piece and board[r
69 + 2][c + 2] == piece and board[r + 3][c + 3] == piece:
70                 return True
```



```
68     for r in range(3, field_height):
69         if board[r][c] == piece and board[r-1][c+1] == piece and board[r
-2][c+2] == piece and board[r-3][c+3] == piece:
70             return True
71     return False
72
73 def evaluate_window(window, player):
74     opp_player = opponent_piece if player == my_piece else my_piece
75     score = 0
76     if window.count(player) == 4:
77         score += 100
78     elif window.count(player) == 3 and window.count(0) == 1:
79         score += 5
80     elif window.count(player) == 2 and window.count(0) == 2:
81         score += 2
82     if window.count(opp_player) == 3 and window.count(0) == 1:
83         score -= 4
84     return score
85
86 def evaluate(board):
87     score = 0
88     center_array = [board[r][field_width//2] for r in range(field_height
)]
89     center_count = center_array.count(my_piece)
90     score += center_count * 3
91     for r in range(field_height):
92         row_array = [board[r][c] for c in range(field_width)]
93         for c in range(field_width - 3):
94             window = row_array[c:c+4]
95             score += evaluate_window(window, my_piece)
96             score -= evaluate_window(window, opponent_piece)
97     for c in range(field_width):
98         col_array = [board[r][c] for r in range(field_height)]
99         for r in range(field_height - 3):
100             window = col_array[r:r+4]
101             score += evaluate_window(window, my_piece)
102             score -= evaluate_window(window, opponent_piece)
103     for r in range(field_height - 3):
104         for c in range(field_width - 3):
105             window = [board[r+i][c+i] for i in range(4)]
```

```
106         score += evaluate_window(window, my_piece)
107         score -= evaluate_window(window, opponent_piece)
108     for r in range(3, field_height):
109         for c in range(field_width - 3):
110             window = [board[r-i][c+i] for i in range(4)]
111             score += evaluate_window(window, my_piece)
112             score -= evaluate_window(window, opponent_piece)
113     return score
114
115 transposition_table = {}
116
117 def board_hash(board, maximizing_player):
118     return (tuple([item for row in board for item in row]),
119             maximizing_player)
120
121 def get_dynamic_depth(board):
122     # Count empty fields
123     empty = sum(row.count(0) for row in board)
124     if empty > 30:
125         return 3 # Beginning: fast, low depth
126     else:
127         return 4 # End: high depth for best moves
128
129 def alpha_beta(board, depth, alpha, beta, maximizing_player):
130     key = (board_hash(board, maximizing_player), depth)
131     if key in transposition_table:
132         return transposition_table[key]
133     valid_locations = [col for col in range(field_width) if
134                       is_valid_location(board, col)]
135     valid_locations.sort(key=lambda c: abs(c - field_width // 2))
136     terminal = winning_move(board, my_piece) or winning_move(board,
137                       opponent_piece) or len(valid_locations) == 0
138     if depth == 0 or terminal:
139         if terminal:
140             if winning_move(board, my_piece): return (None, 1000000)
141             elif winning_move(board, opponent_piece): return (None,
142                               -1000000)
143         else: return (None, 0)
144     else:
145         return (None, evaluate(board))
```

```
142     if maximizing_player:
143         value = -float('inf')
144         best_col = valid_locations[0]
145         for col in valid_locations:
146             row = get_next_open_row(board, col)
147             if row is None:
148                 continue
149             board_copy = [r[:] for r in board]
150             board_copy[row][col] = my_piece
151             new_score = alpha_beta(board_copy, depth-1, alpha, beta, False)
[1]
152             if new_score > value:
153                 value = new_score
154                 best_col = col
155                 alpha = max(alpha, value)
156                 if alpha >= beta:
157                     break
158             result = (best_col, value)
159     else:
160         value = float('inf')
161         best_col = valid_locations[0]
162         for col in valid_locations:
163             row = get_next_open_row(board, col)
164             if row is None:
165                 continue
166             board_copy = [r[:] for r in board]
167             board_copy[row][col] = opponent_piece
168             new_score = alpha_beta(board_copy, depth-1, alpha, beta, True)
[1]
169             if new_score < value:
170                 value = new_score
171                 best_col = col
172                 beta = min(beta, value)
173                 if beta <= alpha:
174                     break
175             result = (best_col, value)
176     transposition_table[key] = result
177     return result
178
179
```

```
180 # ——— MAIN PROGRAM ———
181 board = [[0 for _ in range(field_width)] for _ in range(field_height)]
182 last_board = [row[:] for row in board]
183
184 def move_motor_e_to_zero():
185     pass
186
187 while True:
188     print("Waiting for sensor at port C...")
189     while not sensor_activated():
190         time.sleep(0.1)
191     print("Sensor detected! Starting move...")
192     time.sleep(1)
193
194     # Move to field
195     motor.run_for_degrees(port.D, 198, 170)
196     print("Field reached...")
197     time.sleep(1.5)
198
199     transposition_table.clear()
200
201     opponent_piece_found = False
202     opponent_col = None
203     move_motor_e_to_zero()
204
205     for col in range(field_width - 1, -1, -1):
206         matrix_col = field_width - 1 - col
207         if last_board[field_height - 1][matrix_col] != 0:
208             if col > 0:
209                 motor.run_for_degrees(port.D, move_distance_d, 170)
210                 time.sleep(break_motor_d)
211             continue
212
213         free_row = None
214         for row in range(field_height):
215             if last_board[row][matrix_col] == 0:
216                 free_row = row
217                 break
218         if free_row is None:
219             continue
```

```
220
221     motor.run_for_degrees(port.E, move_distance_e * (free_row), speed_E)
222     time.sleep(break_motor_e)
223     time.sleep(waiting_line)
224
225     detected_color = color_sensor.color(port.B)
226     update_board(free_row, matrix_col, detected_color)
227     print("Matrix entry: Row {}, Col {}: {}".format(
228         free_row, matrix_col,
229         "RED" if board[free_row][matrix_col] == -1
230         else "YELLOW" if board[free_row][matrix_col] == 1
231         else "NONE"))
232
233     if (last_board[free_row][matrix_col] == 0 and
234         board[free_row][matrix_col] == opponent_piece):
235         print("New opponent piece in column {}, row {}".format(matrix_col,
236             free_row))
237         opponent_piece_found = True
238         opponent_col = col
239         time.sleep(1)
240
241     motor.run_for_degrees(port.E, -move_distance_e * free_row, speed_E)
242     time.sleep(break_motor_e_back)
243
244     if opponent_piece_found:
245         break
246
247     if col > 0:
248         motor.run_for_degrees(port.D, move_distance_d, speed_D)
249         time.sleep(break_motor_d)
250
251     if opponent_piece_found and opponent_col is not None and
252     opponent_col != 0:
253         steps_right = opponent_col
254         motor.run_for_degrees(port.D, move_distance_d * steps_right,
255             speed_D)
256         time.sleep(4)
257
258     board_numeric = [[0 if x is None else x for x in row] for row in
259         board]
```

```
256     dynamic_depth = get_dynamic_depth(board_numeric)
257     best_col, _ = alpha_beta(
258         board_numeric,
259         depth=dynamic_depth,
260         alpha=float('inf'),
261         beta=float('inf'),
262         maximizing_player=(my_piece == 1)
263     )
264
265     best_row = get_next_open_row(board_numeric, best_col)
266     color_str = "RED" if my_piece == -1 else "YELLOW"
267     print("\nOptimal position for {}".format(color_str))
268     print("Column: {}, Row: {}".format(best_col, best_row))
269
270     if best_col is not None and best_row is not None:
271         physical_target_col = field_width - 1 - best_col
272         motor.run_for_degrees(port.D, -move_distance_d *
physical_target_col, speed_D)
273         time.sleep(break_motor_d * 4)
274         motor.run_for_degrees(port.A, -360, speed_A)
275         time.sleep(3)
276         board[best_row][best_col] = my_piece
277         last_board = [row[:] for row in board]
278
279         if winning_move(board, my_piece):
280             print(" Congratulations! The robot has WON the game!")
281             print_board(board)
282             sound.beep(440, 1000000, 100)
283             motor.run_for_degrees(port.D, -move_distance_d * best_col,
speed_D)
284             time.sleep_ms(2500)
285             motor.run_for_degrees(port.D, -move_distance_d * 3, speed_D)
286             sound.beep(0, 1000000, 100)
287             break
288
289         motor.run_for_degrees(port.D, -move_distance_d * best_col, speed_D
)
290
291     print_board(board)
292     time.sleep_ms(2500)
```

```
293     motor.run_for_degrees(port.D, -199, speed_D)
294
295     time.sleep(break_motor_d)
296
297     while sensor_activated():
298         time.sleep(0.1)
299     print("Move completed. Waiting for next activation...")
```

B Nutzung von Künstliche Intelligenz basierten Werkzeugen

Im Rahmen dieser Arbeit wurden Künstliche Intelligenz (KI) basierte Werkzeuge benutzt. Tabelle 2.1 gibt eine Übersicht über die verwendeten Werkzeuge und den jeweiligen Einsatzzweck.

Tabelle 2.1: Liste der verwendeten KI basierten Werkzeuge

| Werkzeug | Beschreibung der Nutzung |
|--------------|--|
| ChatGPT | <ul style="list-style-type: none">• Grundlagenrecherche zu Spieltheorie |
| Perplexity | <ul style="list-style-type: none">• Grundlagenrecherche zu Spieltheorie• Formulierungshilfe |
| DeepL | <ul style="list-style-type: none">• Unterstützung beim Übersetzung von Abstract |
| Languagetool | <ul style="list-style-type: none">• Formulierungshilfe• Rechtschreibkorrektur |