Technische Hochschule
Brandenburg
University of
Applied Sciences

Dokumentation des Projektes Künstliche Intelligenz - Schiebepuzzleroboter

Projektdokumentation
für das Projekt „KI“ im 5. Semester
des Fachbereichs Informatik und Medien der
Technischen Hochschule Brandenburg

vorgelegt von:
Laurin Jonientz
Camillo Dobrovsky

Betreuer:
Ingo Boersch
Prof. Dr. Jochen Heinsohn

Brandenburg an der Havel, 25. Januar 2023

Inhaltsverzeichnis

1 Einleitung

Im Projekt „KI“ sollte in diesem Jahr ein Roboter gebaut werden, welcher ein mechanisches 8er-Schiebepuzzle (3x3 Feld mit Zahlen von 1-8) selbstständig löst. Hierzu konnten Sensoren, Motoren, Legosteine, sonstige mechanische Bauteile und ein AKSEN-Board verwendet werden.
Sowohl die Mechanik als auch im Optimalfall die Planung sollte hierbei von einem AKSEN-Board gesteuert bzw. ausgeführt werden. Der Fokus des Projektes lag nicht auf der Erkennung des Startzustandes aus dem mechanischen Puzzle, deswegen wurden Startzustand und Zielzustand im Code vorgegeben.
Als Projektabschluss sind die Roboter in einem Wettbewerb gegeneinander angetreten. Wertungskriterien waren hierbei unter anderem mechanische Funktionalität, mechanische Lösungsgeschwindigkeit und Planungsgeschwindigkeit.
Diese Dokumentation beschreibt sowohl den verwendeten Lösungsansatz des mechanischen Puzzles als auch einen verworfenen Ansatz, sowie den Aufbau des Roboters, verwendete Sensoren und zeigt einige Erklärungen zum Programmcode auf.

2 Lösungsansätze

Im Laufe der Projektarbeit standen mehrere Lösungsansätze zur Debatte. In diesem Kapitel soll ein nicht verwendeter Lösungsansatz und der letztendlich verwendete Lösungsansatz vorgestellt werden. Hierbei werden auch mögliche Vorteile und Probleme des jeweiligen Ansatzes aufgezeigt. Beide Lösungsansätze verwenden hierbei Bälle (z.B. Tischtennisbälle) als mechanische Repräsentation des Puzzles.

2.1 Lösungsansatz 1

Ein möglicher Lösungsansatz ist die Verschiebung der Bälle durch Neigen zu erreichen. Zu jedem Zeitpunkt ist nur genau eine Position des Puzzles frei. Das ermöglicht durch Kippen des Feldes in eine bestimmte Richtung, den korrekten Ball in diese freie Position zu bewegen. Diese Variante hat den Vorteil, dass sie sehr zeiteffizient ist, weil nur die Bewegung eines Motors pro Verschiebung nötig ist.
Gleichzeitig tritt hierbei aber ein Problem auf. Durch das Verschieben des korrekten Balles aus einer beliebigen Mittelposition (horizontale oder vertikale Achse) in die freie Position entsteht eine neue freie Position auf einer der Mittelachsen. In diese freie Position rollt der daneben liegende Ball. Das bedeutet, dass bei diesem Ansatz zusätzlich zur Kippbewegung auch eine Fixierung des Balles notwendig ist, welcher nicht „hinterherrollen“ soll.
Eine Möglichkeit dieses Problem zu lösen, ist das Blockieren des zu fixierenden Balles mit Hilfe von Stäben, welche sich von unten in den Weg des Balles schieben. Nachfolgende Skizze (Abbildung 1) verdeutlicht diesen Ansatz noch einmal.
Dieser Lösungsansatz wurde aber verworfen, weil die Umsetzung, durch die herausfahrenden Stäbe, deutlich komplizierter wurde. So hätten die Motoren, welche die Ebene kippen sollten, nicht nur die Ebene mit den Bällen, sondern zusätzlich noch die Motoren mit den Stäben kippen müssen.

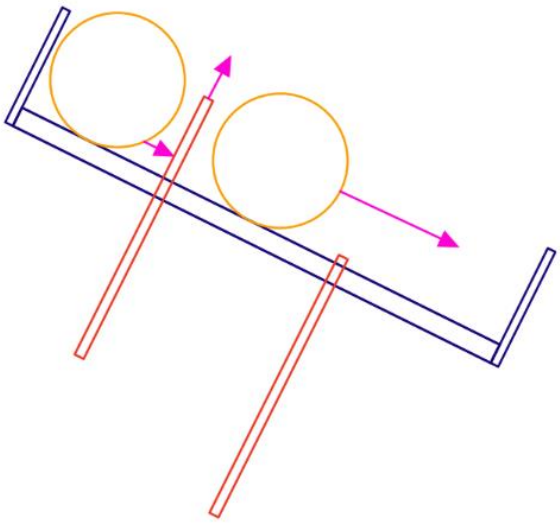

Abbildung 1: Lösungsansatz 1 mit Stäben

2.2 Lösungsansatz 2

Ein anderer Lösungsansatz ist das Bewegen des zu verschiebenden Balles mit einem von unten nach oben fahrendem Stab. Hierbei ist das Feld so begrenzt, dass die Bälle nicht nach außen gestoßen werden können. Auf den Stab, welcher den Ball auf die freie Position schiebt, ist eine Art Rampe befestigt, welche den Ball in die korrekte Richtung drängt.
Dieser Lösungsansatz wurde umgesetzt. Weitere Details zum Aufbau und Arbeitsweise des Puzzles befindet sich in Abschnitt 3.1.

3 Hardware

3.1 Arbeitsweise

Die grundlegende Funktionsweise ist das Bewegen von Bällen mithilfe des Hochdrückens eines Stabes. Die Besonderheit hierbei ist, dass nicht der Stab, sondern das Feld bewegt wird.
Hierfür befindet sich das Feld in einer Schiene und wird von einem Motor mithilfe eines Zahnrades bewegt. Zum Bewegen sind auf dem Feld Zahnstangen befestigt. Motor und Schiene befinden sich auf eine Plattform, welche innerhalb in einer weiteren Schiene durch einen Motor bewegt werden kann. So ist eine Bewegung in alle 4 Richtungen möglich. Für eine Balance zwischen Geschwindigkeit und Präzision wurden hierfür jeweils Zahnräder mit 24 Zähnen eingesetzt.

Abbildung 2: Bewegungsmotoren des Feldes

Um den Stab hochzustoßen, befindet er sich in einer kleinen Box, an welcher sich Zahnstangen befinden. Zudem ist die Box wie das Feld in einer Schiene. Die Bewegung wird wieder durch einen Motor mit einem z24 Zahnrad gesteuert.

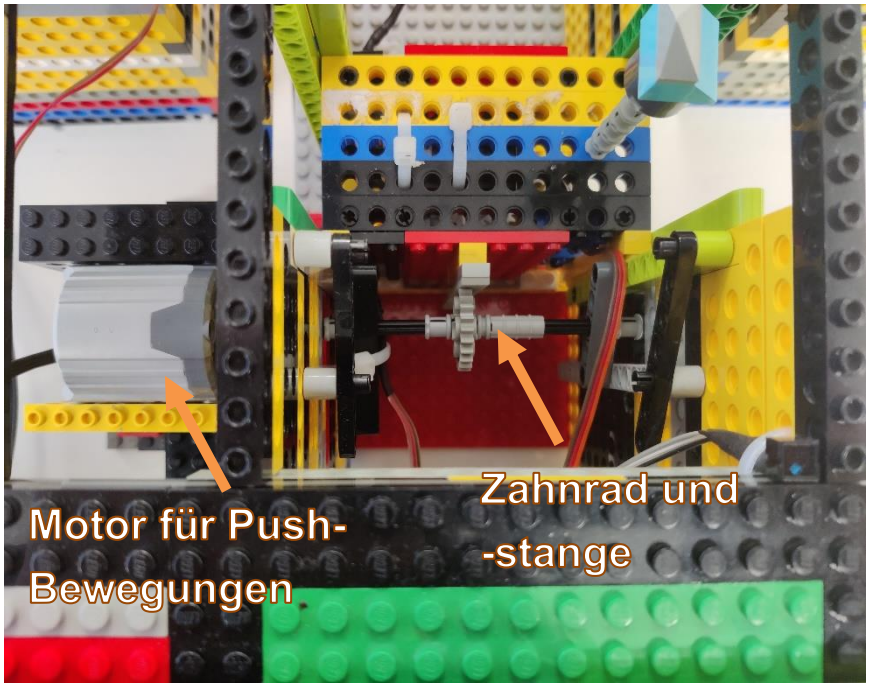

Abbildung 3: Push-Mechanismus des Stabes

Auf dem Stab befindet sich eine kleine Rampe, welche immer in die Richtung zeigen soll, in die der Ball rollen soll. Dazu muss der Stab gedreht werden können. Um dies zu bewerkstelligen, wurde ein Servo eingesetzt. Da der Servo nur eine maximale Drehweite von 180° besitzt, aber für die Ausrichtung in alle 4 Richtungen eine Drehweite von mindestens 270° benötigt wird, wurde mit einer Übersetzung gearbeitet. Hierbei dreht der Servo ein z40 Zahnrad, welches auf ein z24 Zahnrad Übersetzung, auf dem der Stab sitzt. Dadurch wurde die Drehweite auf 300° gesteigert.

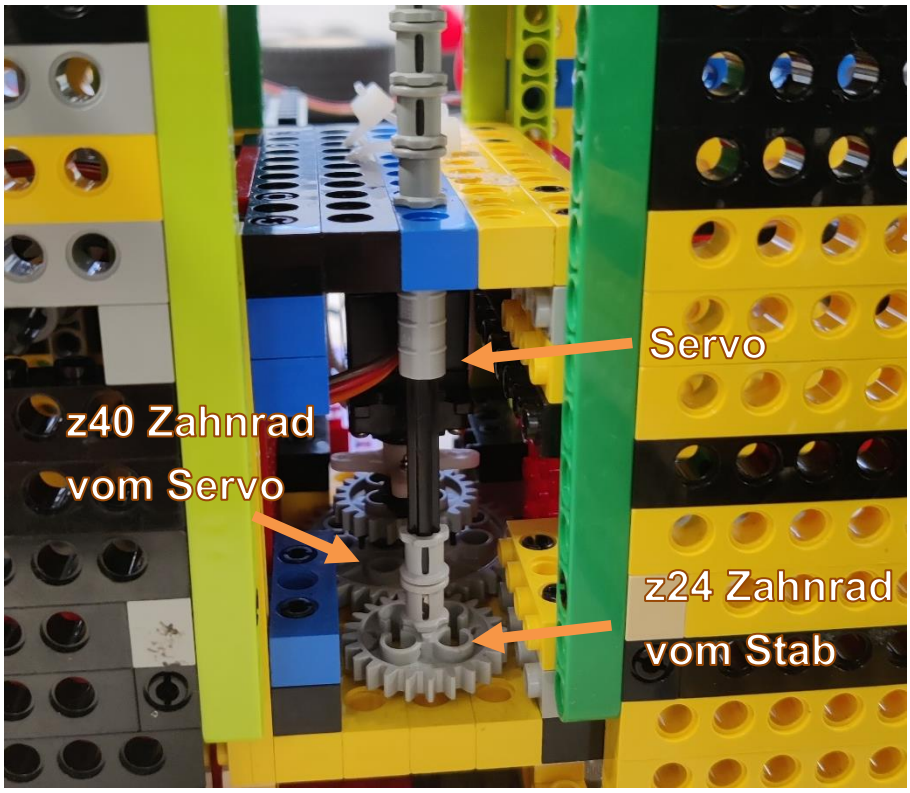

Abbildung 4: Drehmechanismus des Stabes

3.2 Sensorik

Um den Roboter möglichst zuverlässig funktionieren zu lassen und möglichst wenig von der Verlässlichkeit der Motoren abhängig zu sein, wurden zwei Optokoppler und eine Lichtschranke verwendet.

Die Optokoppler wurde für das Bewegen des Feldes genutzt. So befinden sich an der Seite des Feldes sowie unterhalb der bewegbaren Plattform jeweils ein weißer Streifen mit schwarzen Balken. Ein schwarzer Balken steht hierbei für eine Feldposition. Sobald der Optokoppler einen Balken erkennt, weiß das Programm, dass sich der Stab genau unter einer Feldposition befindet. Der mittlere schwarze Balken ist so breit, dass der Stab immer an der gleichen Position zum Stehen kommt, egal in welche Richtung das Feld gerade bewegt wurde. Die Balken links und rechts sind für die Initialisierung des Feldes breiter, damit das Programm weiß an welcher Position des Feldes sich der Stab befindet. Weitere Details dazu werden in Abschnitt 4.4 erklärt.

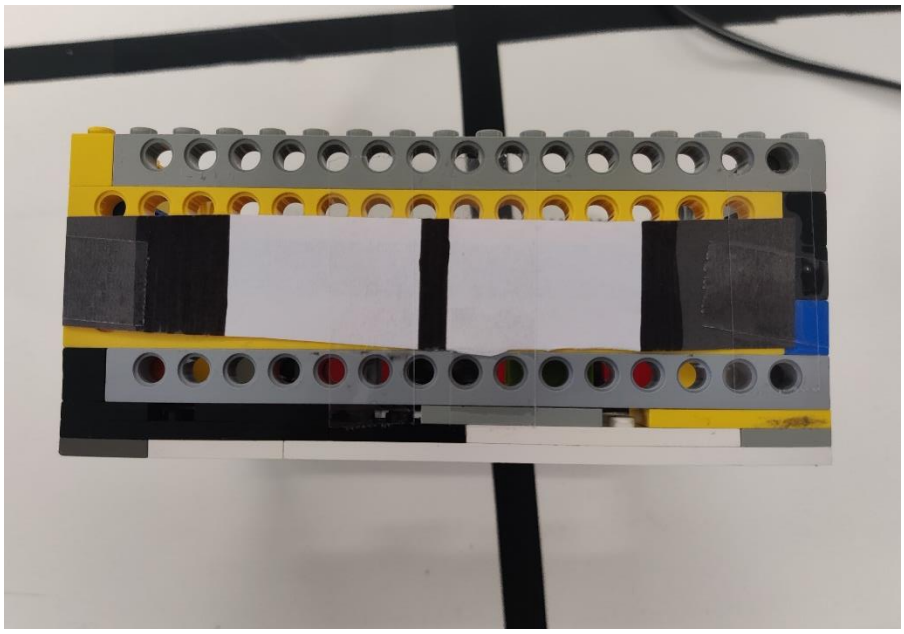

Abbildung 5: Steifen am Feld

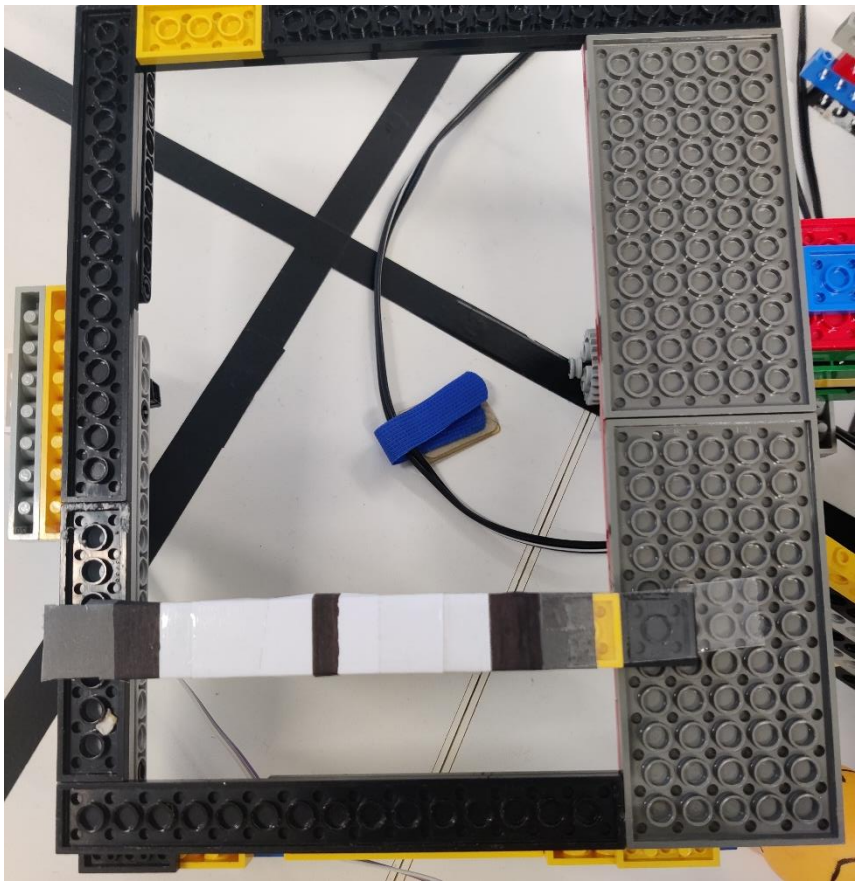

Abbildung 6: Streifen unterhalb der beweglichen Plattform

Die Lichtschranke ist für das Hoch- und Runterbewegen des Stabes zuständig. Sie besteht aus einem Infrarot-Sender und einem Infrarot-Empfänger. Hierbei wird der Stab so lange hochgefahren, bis sich seine Box oberhalb des Senders befindet und

die Lichtschranke sich schließt. Daraufhin wird so lange heruntergefahren, bis der die Lichtschranke wieder unterbrochen wird. Das reicht aus, damit die Box wieder auf dem Boden ankommt.
Durch diese Sensoren kann der Roboter ohne Probleme Lösungswege der Größe 31 oder Endlosschleifen über 10 Minuten lang durchführen ohne Fehlfunktionen zu erleiden.

3.3 Verbesserungsmöglichkeiten

Die Arbeit des Roboters ist sehr zuverlässig. Nach Fertigstellung der Hardware und Software wurden keinerlei Fehler (in Planung und Ausführung) festgestellt. Insbesondere die Planung (siehe Kapitel 4 Software) läuft schnell und findet einen optimalen Plan. Auch bei der Ausführung des Planes konnten wir zu keinem Zeitpunkt eine falsche Bewegung oder einen Fehler des Roboters feststellen.
Nichtsdestotrotz besteht in Hinblick auf die Ausführungsgeschwindigkeit noch Verbesserungspotential. Im Rahmen des Wettbewerbs haben Roboter teilgenommen, welche beim Verschieben von Puzzleteilen etwas schneller (aber auch deutlich unzuverlässiger) waren.

4 Software

Aufgrund der beschränkten Rechenkapazität und des geringen Speichers des AKSEN-Boards ist eine gute Codeoptimierung notwendig. Das folgende Kapitel beschreibt die Theorie des verwendeten Algorithmus, Details zu dem Programm und einige vorgenommenen Optimierungen. Der vollständige (kommentierte) Code ist dieser Dokumentation außerdem als extra Datei beigefügt.

4.1 Algorithmus

Für die Planung des optimalen Lösungsweges wird der IDA* Algorithmus verwendet. Bei dem IDA* Suchverfahren liegt als grundlegender Ansatz die Tiefensuche vor. Da es bei dem Schiebepuzzle jedoch keinen vorgegebenen Suchbaum gibt, welche der Algorithmus durchsuchen soll, und die Aufstellung eines Suchbaumes zu einer unendlichen Tiefe führen kann, eignet sich IDA* in diesem Fall besonders.
Der Algorithmus bezieht in die Suche nicht nur die bisher angewendeten Operationen (=Tiefe), sondern auch die voraussichtlich zurückzulegenden Operationen (= Heuristik) mit ein. Die Kosten eines Knotens setzen sich hierbei also aus Tiefe + Heuristik zusammen. Deswegen nennt man IDA* informierte Suche.
Bei der Suche werden lediglich Knoten expandiert, welche die zuvor festgelegten maximalen Kosten nicht überschreiten. Zu Beginn der Suche werden die maximalen Kosten auf den Wert der Heuristik des Startzustandes gesetzt. Nach Expandieren aller Knoten können die neuen maximalen Kosten anhand der gefundenen, aber nicht expandierten Knotenkosten ermittelt werden. Die neuen maximalen Kosten werden hierbei auf die kleinsten dieser Knotenkosten gesetzt. So garantiert der IDA* einen optimalen Lösungsweg zu finden.

4.2 Heuristik

Die verwendete Heuristik richtet sich nach der in Grundlagen der Wissensverarbeitung kennengelernten Heuristik. Sie wird aus der Summe der Entfernungen (x und y Richtung) jedes einzelnes Puzzleteils ermittelt.
Lediglich das leere Feld wird hierbei nicht beachtet. Dies ist wichtig damit die Heuristik auch bei einfachen Problemen gültig ist. Bei jeder einzelnen Operation

wird das leere Feld mit genau einem Puzzleteil getauscht. Wenn das leere Feld mit in die Heuristik einfließen würde, würde also nicht nur die korrekte Verschiebung des Puzzleteils die Heuristik verkleinern, sondern auch die Verschiebung des leeren Feldes. Dadurch würde diese Heuristik also die Situation in manchen Fällen überschätzen.

4.3 Optimierung bei Implementierung und Agenda

Zur weiteren Optimierung des IDA* werden folgende zwei Aspekte beachtet:

1) Die maximale Länge einer optimalen Lösung beträgt 31. Wenn die Kosten eines nicht Zielknotens größer gleich 31 sind, wird dieser nicht expandiert. Wenn das Puzzle lösbar ist, gibt es in diesem Fall eine kürzere Lösung.
2) Wenn keine Knoten mehr expandiert werden, welche eine kürzere Lösung als 31 Schritte, bricht der Algorithmus ab. In diesem Fall ist das Puzzle unlösbar.

Die größte Optimierung bringt jedoch die Agenda mit sich. Diese speichert folgende Daten zu jedem Knoten:

[char Operation, char Heuristik, char Tiefe]

Die Speicherung des aktuellen Feldes des Knotens wurde, zum Sparen von Speicher, bewusst vermieden. Später stellte sich heraus, dass dies auch einen deutlichen Geschwindigkeitsvorteil mit sich bringt. Das führt jedoch dazu, dass während des Suchens immer das Feld des aktuell betrachteten Knotens rekonstruiert werden muss. Dies geschieht mit Hilfe von 3 gespeicherten „Variablen“:

1) char-Array der Größe 9 für das Feld des aktuellen Knotens (curr_field)
2) char-Array der Größe 33 für die angewendeten Operationen (current_path)
3) char, welches die Tiefe des aktuellen Feldes speichert (curr_field_d)

Zu Beginn wird in curr_field das Startfeld kopiert und curr_field_d auf -1 gesetzt. Sobald ein Knoten aus der Agenda geholt wird, wird die Heuristik und die Tiefe dieses Knotens (= „aktueller Knoten“) gespeichert. Nun wird geprüft, ob der aktuelle Knoten das Ziel erreicht oder ob der aktuelle Knoten aufgrund seiner Kosten nicht expandiert werden muss.

Wenn der aktuelle Knoten expandiert werden muss, werden vorher folgende Schritte ausgeführt:

1) curr_field zum parent_field des aktuellen Knotens machen. In den meisten Fällen ist dies bereits gegeben, da es sich um die Tiefensuche handelt (also „nach unten gesucht wird“).
   Sollte dies nicht der Fall sein, weil bei einem vorherigen Knoten die maximalen Kosten erreicht wurden, kann mit Hilfe der curr_field_d, der Tiefe des aktuellen Knotens und des gespeicherten Pfades diese Bedingung hergestellt werden: Solange die curr_field_d größer gleich der Tiefe des aktuellen Knotens ist, wird die Umkehroperation der in current_path[curr_field_d] gespeicherten Operation (siehe Schritt 2) auf das curr_field angewendet und curr_field_d verkleinert.
2) Die Operation des aktuellen Knotens in current_path[stack_de] speichern.
3) Diese Operation auf das curr_field anwenden und curr_field_d auf die Tiefe des aktuellen Knotens setzen.

Dieses Vorgehen wirkt komplizierter als das einfache Speichern des aktuellen Feldes in der Agende. Betrachtet man aber die mindestens notwendigen Speicheroperationen zum Speichern des gesamten Feldes (je 9 Lese- und Schreibzugriffe) mit den der Wiederherstellung des Feldes (je 3) fällt hier die Optimierung auf.

4.4 Bewegungssteuerung

Für die direkte Steuerung der Hardware gibt es verschiedene Methoden.
Zum Bewegen des Feldes gibt es die Methoden *void leftRight(char schritte)* und *void upDown(char schritte)*. Ein positiver Wert bewegt das Feld die geforderte Anzahl an Schritten/ Feldpositionen nach rechts bzw. oben und ein negativer Wert bewegt das Feld nach links bzw. unten.
Wichtig hierbei zu erwähnen ist, dass beim Bewegen des Feldes die Richtung, in welche der Stab relativ zum Feld bewegt wird, angesteuert wird und nicht die Richtung, in die sich das Feld bewegt. Wenn also *leftRight(1)* aufgerufen wird, bewegt sich das Feld nach links, während der Stab relativ zum Feld nach rechts bewegt wird.

Mit der Funktion *void rotate(char ausrichtung)*, wird der Stab gedreht, wobei eine Zahl von 1 bis 4 für die jeweilige Rotation mitangegeben wird, und mit *void push()*, wird er hoch- und wieder runterfahren.
Damit der vom Planer gefundene Lösungsweg ausgeführt werden kann, gibt er einen String mit der maximalen Größe von 32 Zeichen zurück. Dieser String kann zum Beispiel „6UULDDL“ sein. Das erste Zeichen beschreibt immer die Anfangsposition des Loches. Dieses wird allerdings aufgrund der initialen Ausrichtung des Feldes anders beschrieben als üblich. So befindet sich Feld 0 oben rechts und Feld 8 unten links.

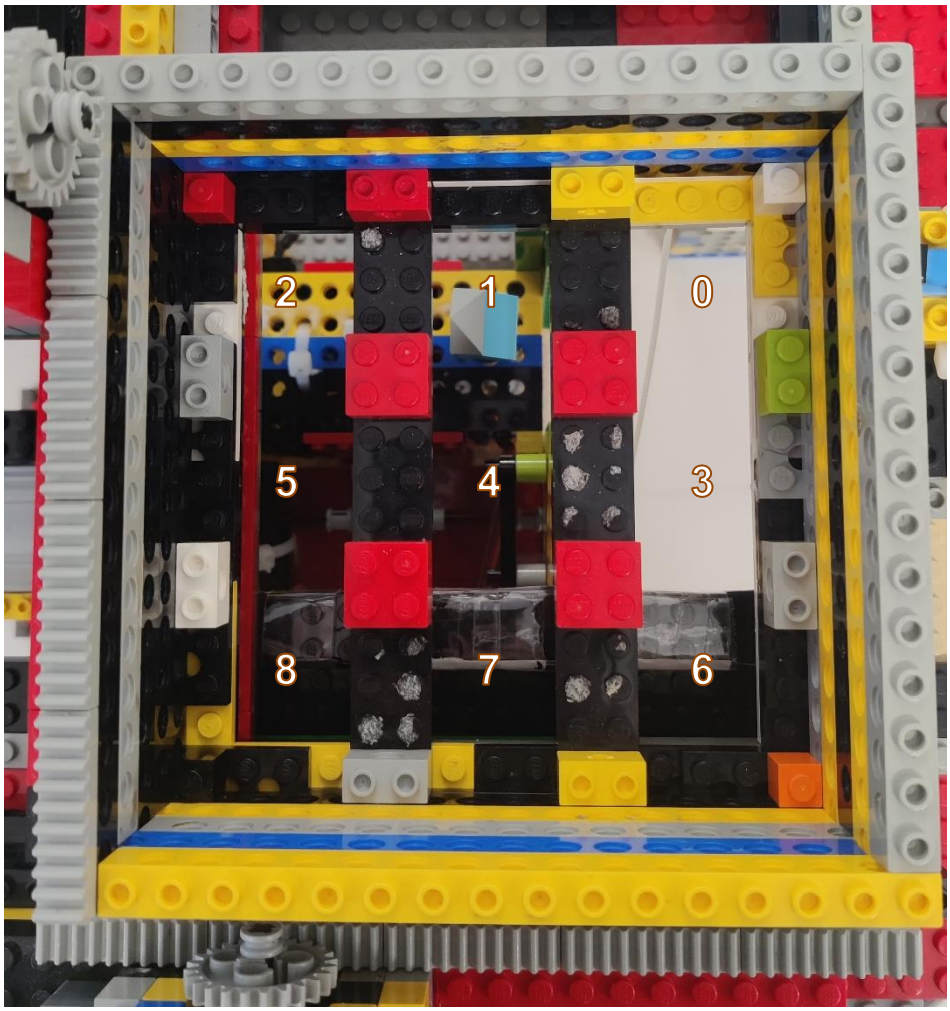

Abbildung 7: Anfangsposition bei der Ausrichtung

Mit dieser Zahl wird dann der Befehl *void init(char position)* aufgerufen. Dieses sorgt für das Einschalten der Lichtschranke und das Herunterfahren des Stabes bei Bedarf. Zudem ruft es *void ausrichten()* auf, welches die Feldposition 0 über den Stab bringt.
Für die Ausrichtung von Up/Down fährt das Feld so lange nach unten bis der Optokoppler für eine längere vordefinierte Zeit die Farbe schwarz sieht. Damit weiß

das Programm, dass es sich am Rand des Feldes befindet und nicht gerade den mittleren Balken erkennt. Danach fährt er wieder zu weiß zurück und nutzt die Methode *upDown(-1)* um auf der vorgesehen Feldposition zu landen. Dasselbe passiert für die Ausrichtung für Left/Right.
Danach fährt *init()* das Feld mithilfe von *UpDown()* und *leftRight()* an die richtige Position.
Der restliche String wird *void move(char* plan)* übergeben. Dieses durchläuft den String und führt je nach Richtung bzw. Zeichen vorbestimmte Befehle aus. Dabei steht „L“ für links, „R“ für rechts, „U“ für oben und „D“ für Down. Dies bezeichnet die Richtung, in die sich das Loch bewegen soll. Ist das Loch zum Beispiel auf der Position 6 und der nächste Befehl lautet „U“, so bewegt sich das Loch auf die Position 3. Die Abfolge der Befehle für das Bewegen des Loches lautet immer *rotate(rotation)*, *upDown(schritte)*/*leftRight(schritte)* und am Ende *push()*.
Nachdem der String vollständig durchlaufen wurde, wird mittels Displays signalisiert, dass der Roboter fertig ist.