Technische Hochschule Brandenburg
University of Applied Sciences

Dokumentation Projekt Künstliche Intelligenz

Jan Philipp Seeland

Technische Hochschule Brandenburg

Studiengang Informatik

KI-Projekt im WS 2020/21

Thema:
Schiebepuzzle

Vorgelegt von:

Jan Philipp Seeland
Fachsemester: 5
Matrikelnummer: 20213364

Bearbeitungszeitraum:

9. Oktober 2020 – 23. Januar 2021

Projektbetreuer:

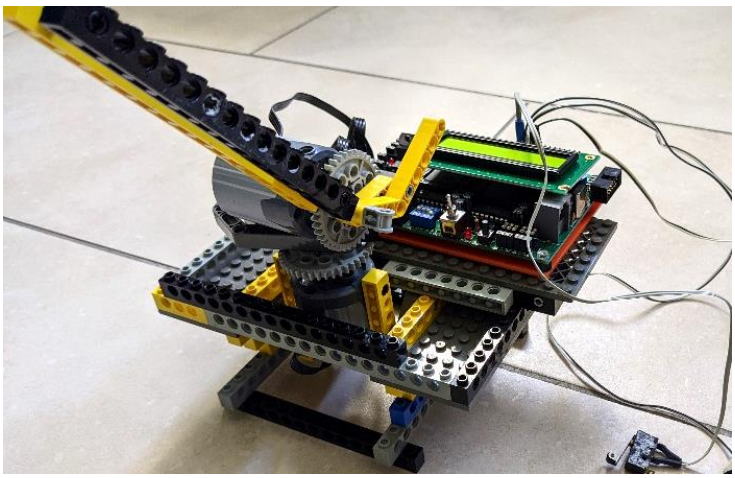
Dipl.-Inform. Ingo Boersch
Prof. Dr. Jochen Heinsohn

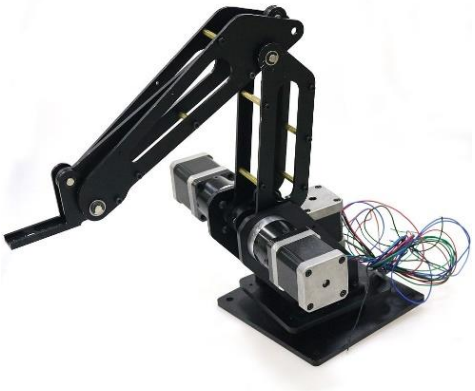
Inhaltsverzeichnis

1. Aufgabenstellung

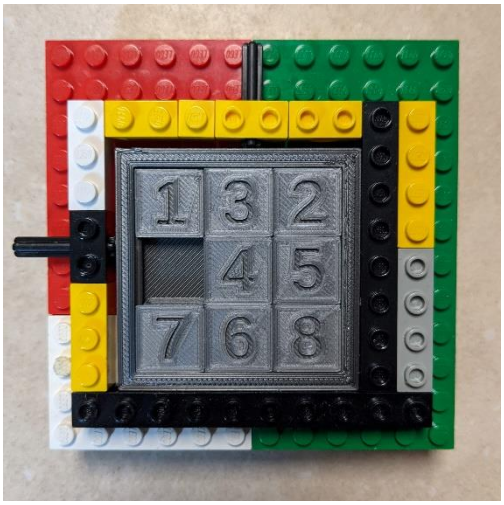
Die Aufgabe im diesjährigen KI-Projekt war es, mit dem AKSEN-Board einen Roboter so zu konstruieren, dass dieser autonom ein 3x3 Schiebepuzzle löst.
Der Roboter muss das Puzzle in akzeptabler Zeit lösen und eine mechanische Visualisierung des Puzzles schieben können. Die Anfangs- und Zielposition steht dem Roboter als Datei auf dem AKSEN Board zur Verfügung, damit keine Puzzleteil Erkennung durchgeführt werden muss.
Am Ende des Projekts treten die einzelnen Roboter im Wettkampf gegeneinander an. Bewertet wird, ob das Puzzle korrekt gelöst wurde, wie schnell das Puzzle in den Zielzustand geschoben wurde und wie schnell der Algorithmus die Planungsaufgabe gelöst hat.

2. Variante Gelenkarm





Der erste Ansatz war von dreiachsigen Roboterarmen inspiriert, um vollständige Bewegungsfreiheit zu haben. Diese Ansätze musste ich dann jedoch verwerfen, da das Ganze zu instabil und ungenau war um das kleine 3D-gedruckte Puzzle zu verschieben.

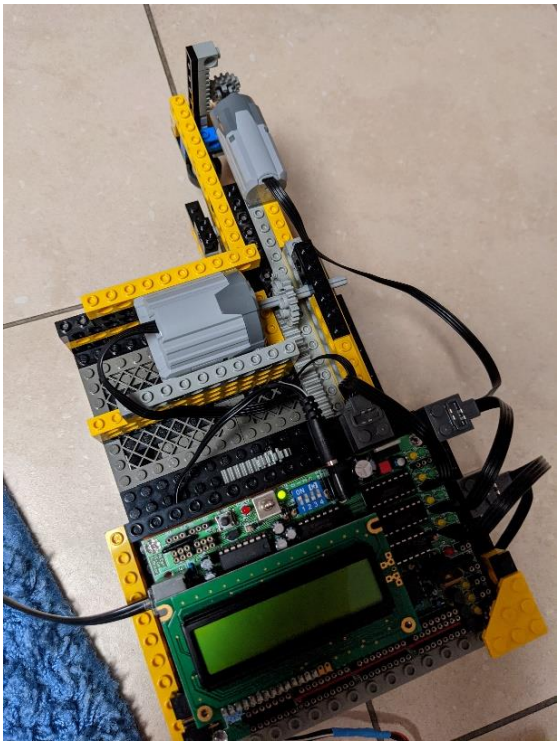


Das Puzzle ist auf einer kleinen Plattform befestigt und von links und rechts eingespannt, da es nicht ganz den Proportionen der LEGO-Einheit entsprach.

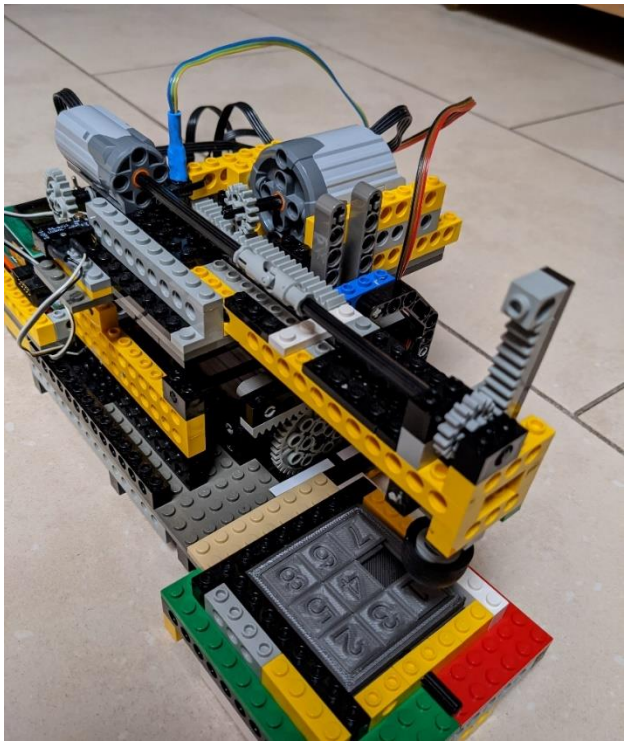
3. Variante Schienenführung

Um die Stabilität zu erhöhen, wurden einige Designs durchgespielt und am Ende erwies sich das Design die Achsen auf Schienen fahren zu lassen als akzeptabel. Dadurch, dass alle Motoren übereinandergestapelt sind, ist der Roboter kompakt und auch vom Gewicht ausbalanciert.

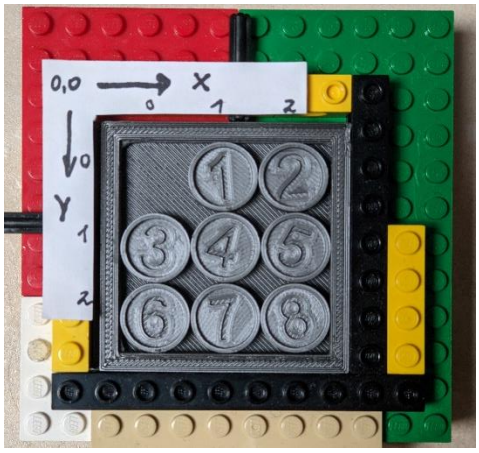
Motor vorne: Gewicht unausgeglichen



Motor hinten: Gewicht ausbalanciert



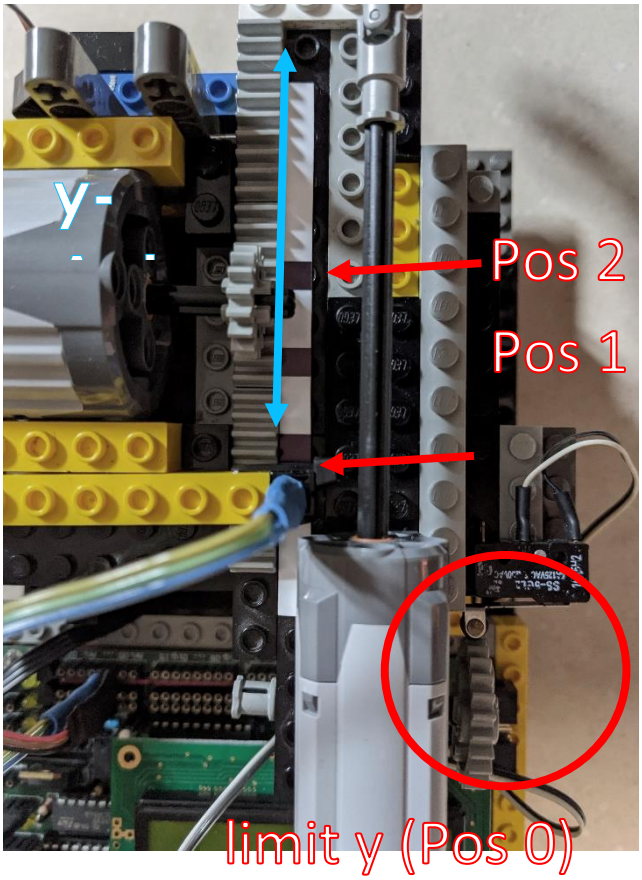
Videobeispiel: Funktionstest - schieben der Bausteine des Puzzles:
https://1drv.ms/v/s!ArFBNeFXk1O2hO9lAgvaDQNAF9dGHQ?e=zfPvEo



Das Design der Puzzleteile wurde ebenfalls geändert. Da die viereckigen, kantigen Teile immer exakt geschoben werden mussten, um nicht zu verhaken, hat dies zu

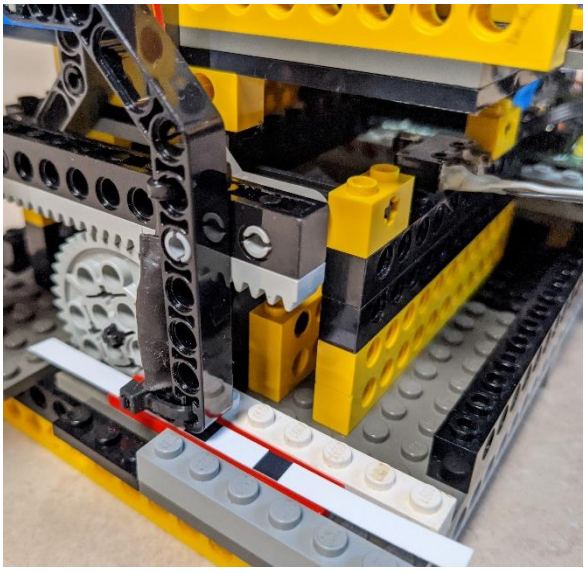
einer hohen Fehleranfälligkeit geführt. Mit komplett runden Teilen konnten sich die Teile „selbst korrigieren“, wenn sie nicht exakt an die Position geschoben wurden.

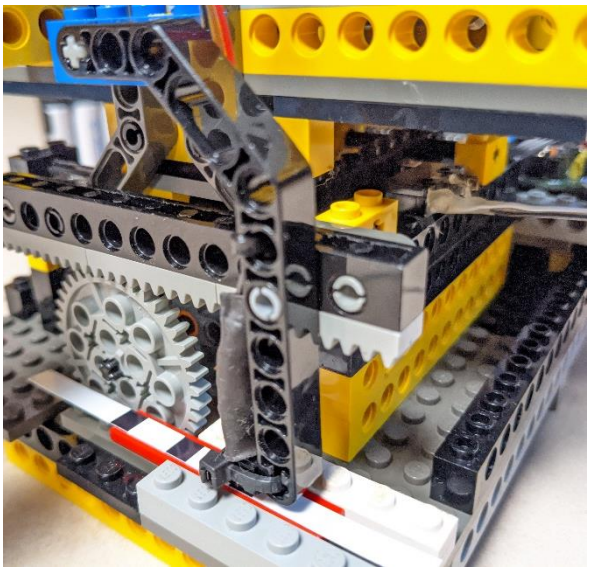
4. Sensorik



Ein Ziel war es, mit so wenig wie möglich Sensoren auszukommen und die verbaute Hardware so effizient wie möglich zu nutzen.
Daher war die Idee, die Position durch Optokoppler zu bestimmen. Das ermöglicht auch, die Positionen im Nachhinein anpassen zu können, indem man die Abstände zwischen den schwarzen Markern erhöht oder verringert.

In der Testphase trat jedoch das Problem auf, die Optokoppler dazu zu bringen, die Marker zuverlässig zu erkennen. Häufig fluktuierenden die Werte der Optokoppler, wodurch es schwer war, das System und die dazugehörige Software zuverlässig zu gestalten.
Nach mehreren Tagen Materialforschung haben sich gedruckte Streifen (Druckertinte statt Edding!) auf mattem Papier als zuverlässig erwiesen. Ein geringer, aber nicht zu geringer Abstand sowie ein genauer 90° Winkel waren für die zuverlässige Erkennung auch wichtig.





Die Anschläge wurden mit Schaltern realisiert. Diese gewährleisten, dass der Roboterarm immer exakt auf Position (0,0) fahren kann. Diese Position wird auch zur Kalibrierung benutzt.

5. Software

Für die Wegfindung wurde der Iterative deepening A* (IDA*) Algorithmus in C implementiert. Der IDA* Algorithmus erweitert die Iterative Tiefensuche um eine Heuristik. Bei informierter Suche wird immer der optimale Weg gefunden. Die Heuristik erhöht dabei die Geschwindigkeit bei der Vertiefung.
Da der Algorithmus nach Erreichen einer bestimmten Tiefe von vorne anfängt, wird wenig Speicher benötigt, jedoch mehr Zeit da Knoten oft mehrmals exploriert werden. [1]

4.1 Baumstruktur und Agenda

```c
//define struct node
typedef struct node{
    //depth
    unsigned char g;
    //heuristic
    unsigned char h;
    //puzzle
    unsigned char puzzle[3][3];
    //move to be made
    unsigned char move;
}node_t;
```

Selbstdefinierter Datentyp für einen Knoten (node) im Suchbaum.

```c
void init_Node(node_t *node_ptr, unsigned char g, unsigned char h, unsigned char const *puzzle_ptr, unsigned char move){
    unsigned char i, j;
    node_ptr->g = g;
    node_ptr->h = h;
    for (i = 0; i < 3; i++) {
        for (j = 0; j < 3; j++) {
            node_ptr->puzzle[i][j] = *(puzzle_ptr + (i*3) + j);
        }
    }
    node_ptr->move=move;
}
```

init_node() initialisiert die Knoten im Suchbaum. Da die Knoten am speicherhungrigsten sind, ist es wichtig, nicht zu viele gleichzeitig speichern zu müssen. Auch hier spielt uns der IDA* Algorithmus in die Hände:

In der Agenda werden nur zu besuchende Knoten gespeichert. Bereits besuchte Knoten oder Knoten, welche die maximalen Kosten überschreiten, werden aus der Agenda gelöscht.

Von den insgesamt 8kB RAM auf dem AKSEN-Board fallen rund 0,8 kB weg und lassen 7,2 kB für globale Variablen übrig. Ein Node braucht 12 Byte Speicher. Die Agenda ist ein Array vom Typ node_t und ist für die Speicherung der Knoten verantwortlich. Mit 255 als größte Zahl, die sich mit einem unsigned char darstellen lässt, könnten maximal 255 Knoten adressiert werden. Damit werden maximal 3,06 kB für den Suchbaum verwendet werden. Für globale Variablen bleiben damit 4,1 kB übrig,

mehr als genug für die restliche Programmlogik. In Tests wurde festgestellt, dass die Agenda in den meisten Fällen lediglich mit
30 - 40 Knoten gefüllt ist.
Die Website MovingAI [2] hat mir auch sehr geholfen, den IDA* Algorithmus zu verstehen.

4.2 Heuristik

Als Heuristik wurde die Summe der Distanzen der Puzzleteile zu ihrer Zielposition genommen (Manhattan Distance). Wie auch im Kurs „Grundlagen der Wissensverarbeitung“ gelehrt, differenziert diese Heuristik sehr stark und gibt damit eine gute Schätzung der Kosten zum Ziel ab.

```
Index arrInd, endInd;
int dist(int *arr_ptr, int *end_arr_ptr){
    int i, dist_temp = 0;
    for (i = 1; i < 9; ++i) {
        getIndex(&arrInd, arr_ptr,     i);
        getIndex(&endInd, end_arr_ptr, i);
        dist_temp += my_abs((arrInd.x) - (endInd.x)) + my_abs((arrInd.y) - (endInd.y));
    }
    return dist_temp;
}
```

4.3 Move-Befehle und Bewegungsinterpreter

Die Speicherung der moves, die zum Lösen des Puzzles benötigt werden, erscheint auf den ersten Blick unintuitiv, erspart aber einiges an Arbeit.
Anstatt den Weg im Suchbaum nachzuvollziehen, indem man den Baum vom Lösungsknoten zum Wurzelknoten traversiert, wird jeder Baumtiefe ein move-Befehl zugeordnet:
Wenn während der Suche, Knoten in einer neuen Tiefe expandiert werden, wird der move Befehl des expandierten Vaterknotens im moves-Array an der Position der Baumtiefe gespeichert.
Am Ende wird nur der optimale Pfad zum Lösungsknoten expandiert. Daraus folgt, dass auch alle Vaterknoten optimal sind. Damit stehen am Ende die move-Befehle in korrekter Reihenfolge im moves-Array. Alle nicht optimalen Expandierungen, die während der Suche des Lösungsknotens ausgeführt wurden, werden überschrieben.
Die Befehle werden dann an den Bewegungsinterpreter weitergegeben.

Der Roboter kann die Bewegungsbefehle „u“ - up, „d“ – down, „l“ – left und „r“ – right verstehen.
Es wird das Leerfeld in die jeweilige Richtung geschoben.

4.4 Dateistruktur

Der Code ist nach C-Konvention in .c und dazugehörige .h Header Files aufgeteilt. In der „main.c“ Datei sind alle Dateien inkludiert, dessen Funktionen aufgerufen werden. Die Main-Datei hält auch Start- und Endpositionen der Puzzle.

main.c ← benutzt die Funktionen der beiden „Bibliotheken“ motor_driver und puzzle_logic

main.ihx

makefile Kompilierte Datei, welche auf das AKSEN-Board

motor_driver.c
motor_driver.h } Verantwortlich für Bewegung des Roboters und Interpretation der Bewegungsbefehle

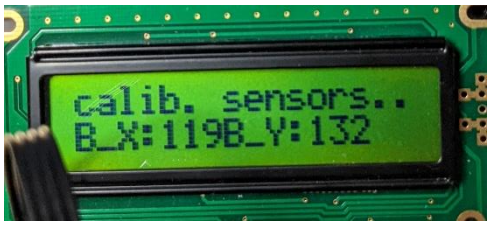
puzzle_logic.c
puzzle_logic.h } Verantwortlich für Lösen des Puzzles, enthält alle nötigen Funktionen zur Lösungsfindung

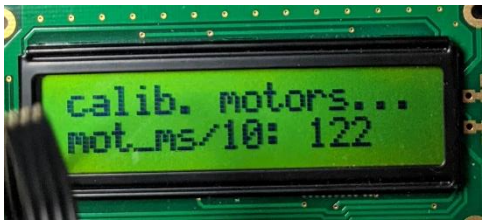
6. Optimierungen

5.1 Kalibrierung

Seit dem 29.11.2020 gibt es eine Kalibrierung!
Die Optokoppler werden am Anfang auf eine schwarze Fläche gefahren und lesen einen Messwert. Anhand dieser Messung wird der Grenzwert für den Schwarzwert gesetzt.

Auch die Motoren sollen eine Kalibrierung bekommen. Die Idee ist, den zeitlichen Abstand zwischen Position 0 und 1 zu messen. Dies legt das Maximum für die Zeit fest, in der die Optokoppler inaktiv sind. Damit soll verhindert werden, dass die aktuelle weiße Markierung als die nächste Markierung interpretiert wird).





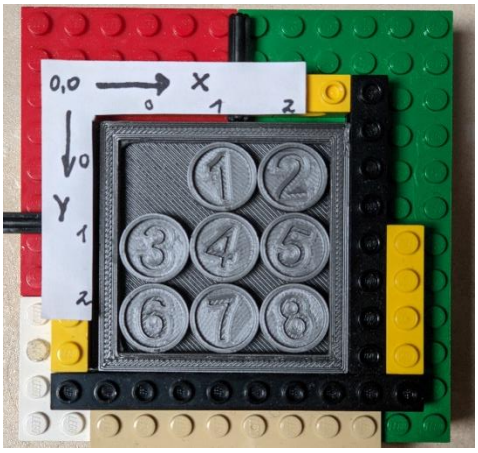
5.2 Motorsteuerung

Die Motoren werden nicht nur an- und ausgeschaltet, sondern auch gestoppt. Dafür wird ganz simpel, eine kurze Gegenrotation angelegt, um den Rotor zu bremsen.

5.3 Puzzleoptimierung

Das 3D-gedruckte Puzzle hatte den entscheidenden Nachteil, dass die Ecken spitz waren und so gut wie keine Fehlertoleranz zugelassen haben, da schon bei einer kleinen Ungenauigkeit beim Schieben sich die Puzzleteile verkantet haben. Die erste Idee war, mit einem Dremel die Ecken der Puzzleteile abzuschleifen:

In der Theorie hätte das geklappt, nur hätten perfekte, komplett identische Rundungen geschliffen werden müssen.
Da das offensichtlich nur maschinell möglich ist, fiel die Entscheidung, das 3D-Modell neu, mit runden Ecken zu drucken.

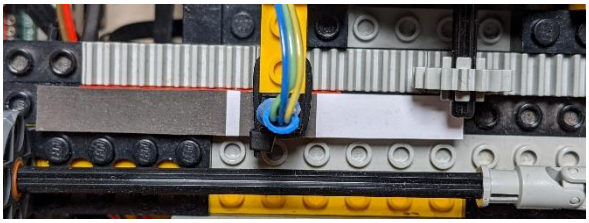


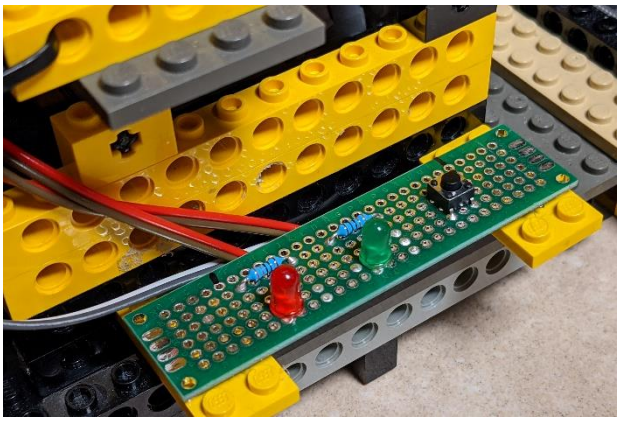
Das neu designte Puzzle, welches komplett runde Bausteine hat, bestand alle Zuverlässigkeitstests, u.a. 20 min. fehlerfrei zu laufen.

Zeitraffer 20min. fehlerfrei: https://1drv.ms/v/s!ArFBNeFXk1O2hPodavc4jW5yisjLtg

5.4 Hardwareoptimierungen

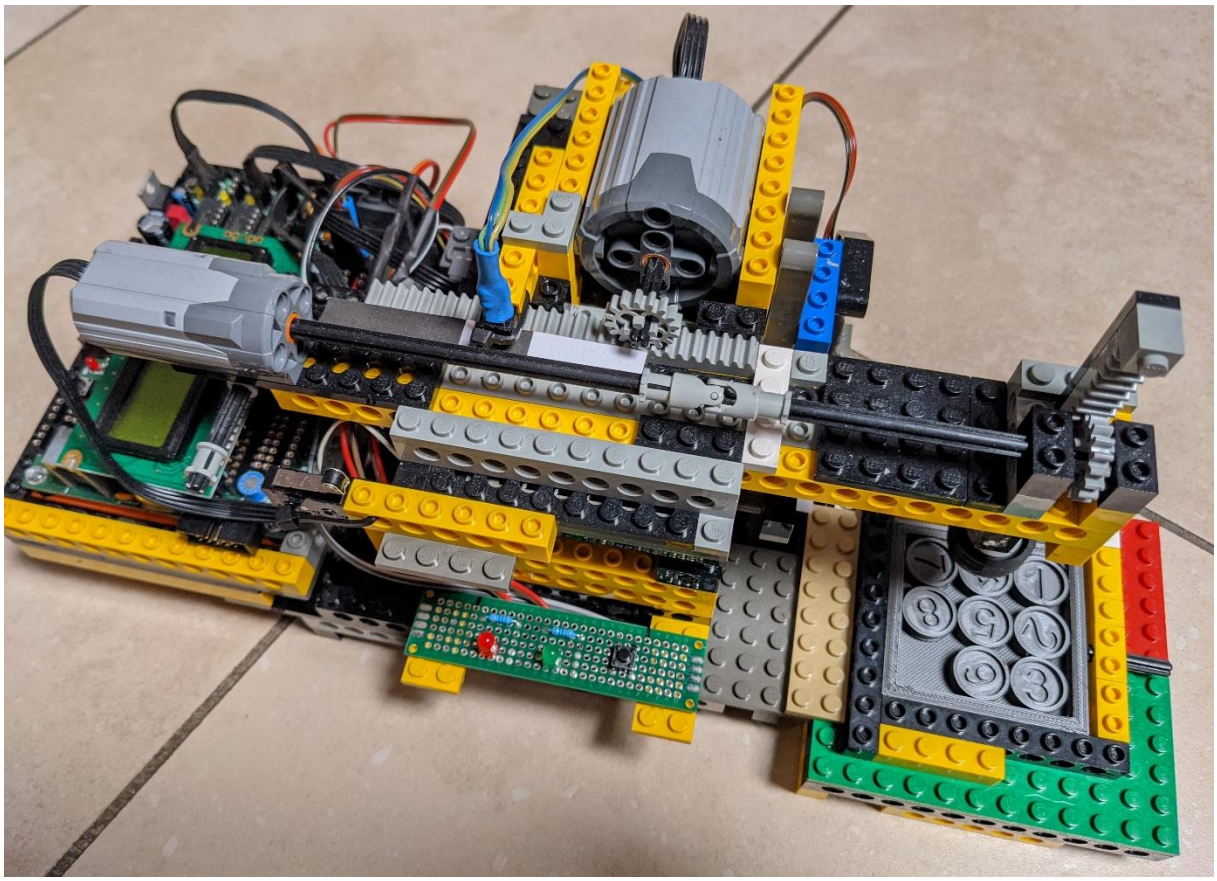
Am grundlegenden Design hat sich seit Version 2 nicht viel geändert.
Die Streifen, an denen sich die Optokoppler orientieren, wurden invertiert, da die Sensoren den Übergang von Schwarz zu weiß besser erkannt haben.





Um beim Wettbewerb gegen die anderen Roboter eine Indikator Anzeige zu haben, wann der Roboter plant (rote LED blinkt), fertig mit der Planung ist (grüne LED an) und die Planung oder Ausführung starten soll (Knopf drücken), habe ich ein kleines „breakout board“ gebaut, um alles schick an einem Platz zu haben.

Hier nochmal ein Bild vom finalen Roboter im Ganzen:



7. Schlusswort

Die Aufgabe, ein Autonomes System zum Lösen eines Schiebepuzzles zu konstruieren, war ohne Frage eine Herausforderung. Viele Probleme und Unklarheiten mussten gelöst werden. Dazu zählt der Bau eines zuverlässigen Systems, welches mechanisch die Puzzleteile schieben konnte und der Algorithmus zur Wegfindung auf einem Mikrocontroller Board mit begrenzten Hardwareressourcen. Dadurch dass man ständig dazu angehalten war, effiziente Lösungen für die Probleme zu finden, war die Aufgabe äußerst lehrreich und sehr interessant. Das Ziel am Ende beim Wettkampf einen guten Platz zu belegen hat die Motivation ebenfalls gesteigert.

Da der Wettbewerb und die Vorstellung der Projekte durch Corona online stattgefunden haben, hatte ich für die Gruppe das Abschlussvideo zusammengeschnitten, welches unter diesem Link abrufbar ist:

https://1drv.ms/v/s!ArFBNeFXk1O2hZUZ_Mqw5_LAaJ3wiQ?e=T8VDn4

```c
//AKSEN lib
#include <stub.h>

//custom includes
#include "motor_driver.h"
#include "puzzle_logic.h"

//var's
unsigned char i, d, pwm;
#define USER_BTN digital_in(8)

//init puzzles to solve
unsigned char puzzle_0[3][3] = {{0,1,2}, {3,4,5}, {6,7,8}};
unsigned char puzzle_1[3][3] = {{1,2,0}, {3,4,5}, {6,7,8}};
unsigned char puzzle_2[3][3] = {{1,2,3}, {0,4,5}, {6,7,8}};
unsigned char puzzle_3[3][3] = {{1,2,3}, {4,5,6}, {0,7,8}};
unsigned char puzzle_4[3][3] = {{0,1,2}, {3,4,5}, {6,7,8}};

void waitingForUserInput(){
    //use a PWM signal to dimm the LED
    while(USER_BTN != 0){
        if (pwm==1) pwm=0; else pwm=1;
        led(1,pwm);
        sleep(5);
    }
    led(1,0);
}

//solve routine
void solveRoutine(unsigned char *s_pos,unsigned char *s_end){
    Index hole_pos;
    lcd_cls();
    lcd_setxy(0,0);
    lcd_puts("solving...");
    solve(s_pos, s_end);
    //if solving routine finishes, a route was found
    lcd_setxy(0,0);
    lcd_puts("route gefunden!: ");
    //display route, befor doing any moves
    i = 1;
    lcd_setxy(1,0);
    d = getDepth();
    while(i <= d){
        lcd_putchar(getMove(i));
        i++;
    }
    waitingForUserInput();
    //beginning of moves sequence starts over 0 -> position arm there
    getIndex(&hole_pos, s_pos, 0);
    moveTo_index(hole_pos.x, hole_pos.y);
    i = 1;
    //execute moves
    while(i <= d){
```

```
54          move(getMove(i));
55          i++;
56      }
57      lcd_cls();
58      lcd_setxy(1,0);
59      lcd_puts("Puzzle solved!!");
60      //task finished, goto idle
61      lcd_setxy(0,0);
62      lcd_puts("await user input");
63      waitingForUserInput();
64  }
65
66  void AksenMain(void) {
67      //green LED on while ready for input
68      lcd_setxy(0,0);
69      lcd_puts("await user input");
70      waitingForUserInput();
71      //calibrate
72      calibrate();
73      lcd_setxy(0,0);
74      lcd_puts("await user input");
75      waitingForUserInput();
76      //solve first puzzle and wait for next round
77      solveRoutine(&puzzle_0[0][0], &puzzle_1[0][0]);
78      //solve 2nd puzzle and wait for next round
79      solveRoutine(&puzzle_1[0][0], &puzzle_2[0][0]);
80      //solve 3rd puzzle and wait for next round
81      solveRoutine(&puzzle_2[0][0], &puzzle_3[0][0]);
82      //solve final puzzle and wait for input
83      solveRoutine(&puzzle_3[0][0], &puzzle_4[0][0]);
84      while(1);
85  }
86
```

```
1  //
2  // Created by JP_Seeland on 07.10.2020.
3  //
4
5  #ifndef MOTOR_DRIVER_H
6  #define MOTOR_DRIVER_H
7
8  //Sensor port defines
9  #define OPTO_X 2
10 #define OPTO_Y 3
11
12 //Motor port defines
13 #define MOT_X 0
14 #define MOT_Y 2
15 #define MOT_Z 3
16
17 //Timings
18 #define ZAXIS 280
19 #define OPTO_SLEEP 200
20 #define C_SPIN 15   //counter spin to stop motor
21
22 //Limits
23 #define LIMIT_X 0
24 #define LIMIT_Y 1
25
26 //Motor speed
27 #define X_SPEED 5
28 #define Y_SPEED 6
29 #define HEAD_SPEED 5
30
31 int MIN_BLACK_X = 150;
32 int MIN_BLACK_Y = 150;
33 /**
34  * Execute in init phase to save a relative black value
35  *
36  */
37 void calibrate();
38 /**
39  * Operates robot arm to move the empty space in puzzle.
40  * Movement end's over position of new empty space.
41  * @param direction : u, d, l, r
42  */
43 void move(unsigned char direction);
44
45 /**
46  *  reset's head to 0,0 then moves to specific index
47  *  @param int x: 0 - 2
48  *  @param int y: 0 - 2
49  */
50 void moveTo_index(int x, int y);
51
52 /**
53  * Smooth motor motion
```

```
54   * @param mot: choose motor 0-3
55   * @param speed : 0 - 10
56   * @param dir : Direction 0 or 1
57   * @param ms : 0 - MAX_INT
58   */
59  void easy_ease(unsigned char mot, unsigned char speed, unsigned char dir,
       int ms);
60
61  /******HELPER FUNCTIONS*******/
62  void h_up();
63  void h_down();
64  void h_left();
65  void h_right();
66  void h_grab();
67  void h_release();
68
69
70  #endif //PUZZLE_SOLVER_MOTOR_DRIVER_H
71
```

```c
//
// Created by JP_Se on 07.10.2020.
//
#include <stub.h>
#include "motor_driver.h"

//global var's
unsigned char move_mot = 0;
unsigned int limit_y, limit_x = 0;
unsigned int opto_x, opto_y;
int head_pos_x, head_pos_y = 0;

/****internal functions******/
// I/O
void read_opto(unsigned int *opto_x, unsigned int *opto_y) {
    *opto_x = analog(OPTO_X);
    *opto_y = analog(OPTO_Y);
}
// I/O
void read_limiter(unsigned int *limit_x, unsigned int *limit_y) {
    *limit_x = digital_in(LIMIT_X);
    *limit_y = digital_in(LIMIT_Y);
}

int white_x(){
    if(analog(OPTO_X) > MIN_BLACK_X){
        return 0; //black
    }else{
        return 1; //white
    }
}
int white_y(){
    if(analog(OPTO_Y) > MIN_BLACK_Y){
        return 0; //black
    }else{
        return 1; //white
    }
}

void calibrate(){
    int cal_i = 0;
    //move to limit's -> ensures opto on black
    moveTo_index(0,0);
    //read opto
    read_opto(&opto_x, &opto_y);
    //set MIN_BLACK_X threshold
    MIN_BLACK_X = opto_x - 50;
    MIN_BLACK_Y = opto_y - 50;
    lcd_setxy(0,0);
    lcd_puts("calib. sensors...");
    lcd_setxy(1,0);
    lcd_puts("B_X:");
    lcd_int(MIN_BLACK_X);
```

```
54      lcd_puts("B_Y:");
55      lcd_int(MIN_BLACK_Y);
56      sleep(2000);
57      //calibrate motors
58      lcd_setxy(0,0);
59      lcd_puts("calib. motors...");
60      //move y till white:
61      motor_richtung(MOT_Y, 0);
62      motor_pwm(MOT_Y, 3);
63      while(!white_y()){
64          sleep(50);
65          cal_i++;
66      }
67      motor_pwm(MOT_Y, 0);
68      lcd_setxy(1,0);
69      lcd_puts("mot_ms/10: ");
70      lcd_int(cal_i);
71      //wenn höher als 600, dann motoren langsamer einstellen
72      if(cal_i > 600){
73          lcd_setxy(1,0);
74          lcd_puts("mot speed reduced!");
75      }
76
77      //reset robot
78      moveTo_index(0,0);
79  }
80
81  /******external functions******/
82  void move(unsigned char direction){
83      switch(direction){
84          case 'u':
85              //UP, GRAB, DOWN, RELEASE, UP
86              h_up();
87              h_grab();
88              h_down();
89              h_release();
90              h_up();
91              break;
92          case 'd':
93              //DOWN, GRAB, UP, RELEASE, DOWN
94              h_down();
95              h_grab();
96              h_up();
97              h_release();
98              h_down();
99              break;
100         case 'l':
101             //LEFT, GRAB, RIGHT, RELEASE, LEFT
102             h_left();
103             h_grab();
104             h_right();
105             h_release();
106             h_left();
```

```
107             break;
108         case 'r':
109             //RIGHT, GRAB, LEFT, RELEASE, RIGHT
110             h_right();
111             h_grab();
112             h_left();
113             h_release();
114             h_right();
115             break;
116         default:
117             break;
118     }
119 }
120 
121 void moveTo_index(int x, int y){
122     //go to limiter (pos 0,0)
123     unsigned int limit_x, limit_y;
124     h_release();
125     head_pos_x = 0;
126     head_pos_y = 0;
127     read_limiter(&limit_x, &limit_y);
128     motor_richtung(MOT_X, 1);
129     motor_pwm(MOT_X, X_SPEED);
130     do{
131         read_limiter(&limit_x, &limit_y);
132     }while(limit_x != 0);
133     motor_pwm(MOT_X, 0);
134 
135     motor_richtung(MOT_Y, 1);
136     motor_pwm(MOT_Y, Y_SPEED);
137     do {
138         read_limiter(&limit_x, &limit_y);
139     } while(limit_y != 0);
140     motor_pwm(MOT_Y, 0);
141     //go to index
142     while(x != head_pos_x){
143         h_right();
144     }
145     while(y != head_pos_y){
146         h_down();
147     }
148 }
149 
150 void easy_ease(unsigned char mot, unsigned char speed, unsigned char dir,  ↵
      int ms){
151     if(ms != 0){
152         int ease_t = ms/10;
153         motor_richtung(mot, dir);
154         motor_pwm(mot, speed/2);
155         sleep(ease_t);
156         motor_pwm(mot, speed);
157         sleep(ease_t*8);
158         motor_pwm(mot, speed/2);
```

```
159         sleep(ease_t);
160         motor_pwm(mot, 0);
161     }else{
162         motor_richtung(mot, dir);
163         motor_pwm(mot, speed);
164     }
165 }
166 
167 /***helper functions****/
168 void h_up(){
169     if(head_pos_y == 1){
170         //check limit's
171         motor_richtung(MOT_Y, 1);
172         motor_pwm(MOT_Y, Y_SPEED);
173         while(digital_in(LIMIT_Y) != 0);
174         motor_pwm(MOT_Y, 0);
175     }
176     if(head_pos_y == 2){
177         //check opto's
178         motor_richtung(MOT_Y, 1);
179         motor_pwm(MOT_Y, Y_SPEED);
180         sleep(OPTO_SLEEP);
181         while(!white_y());
182         motor_richtung(MOT_Y, 0);
183         sleep(C_SPIN);
184         motor_pwm(MOT_Y, 0);
185     }
186     if(head_pos_y == 0){
187         //Kopf am Anschlag, nicht bewegen (y: 0)
188         lcd_setxy(0,0);
189         lcd_puts("illegal (h_up)");
190     }
191     head_pos_y--;
192 }
193 
194 void h_down(){
195     if(head_pos_y < 2){
196         motor_richtung(MOT_Y, 0);
197         motor_pwm(MOT_Y, Y_SPEED);
198         sleep(OPTO_SLEEP);
199         while(!white_y());
200         motor_richtung(MOT_Y, 1);
201         sleep(C_SPIN);
202         motor_pwm(MOT_Y, 0);
203     }else{
204         //Kopf am Anschlag, nicht bewegen (y: 2)
205         lcd_setxy(0,0);
206         lcd_puts("illegal (h_down)");
207     }
208     head_pos_y++;
209 }
210 
211 void h_left(){
```

```
212     if(head_pos_x == 1){
213         //check limit's
214         motor_richtung(MOT_X, 1);
215         motor_pwm(MOT_X, X_SPEED);
216         while(digital_in(LIMIT_X) != 0);
217         motor_pwm(MOT_X, 0);
218     }
219     if(head_pos_x == 2){
220         //check opto's
221         motor_richtung(MOT_X, 1);
222         motor_pwm(MOT_X, X_SPEED);
223         sleep(OPTO_SLEEP);
224         while(!white_x());
225         motor_richtung(MOT_X, 0);
226         sleep(C_SPIN);
227         motor_pwm(MOT_X, 0);
228     }
229     if(head_pos_x == 0){
230         //Kopf am Anschlag, nicht bewegen (y: 0)
231         lcd_setxy(0,0);
232         lcd_puts("illegal (h_left)");
233     }
234     head_pos_x--;
235 }
236 
237 void h_right(){
238     if(head_pos_x < 2){
239         motor_richtung(MOT_X, 0);
240         motor_pwm(MOT_X, X_SPEED);
241         sleep(OPTO_SLEEP);
242         while(!white_x());
243         motor_richtung(MOT_X, 1);
244         sleep(C_SPIN);
245         motor_pwm(MOT_X, 0);
246     }else{
247         //Kopf am Anschlag, nicht bewegen (y: 2)
248         lcd_setxy(0,0);
249         lcd_puts("illegal (h_right)");
250     }
251     head_pos_x++;
252 }
253 
254 void h_grab(){
255     easy_ease(MOT_Z, HEAD_SPEED, 1, ZAXIS);
256 }
257 
258 void h_release(){
259     easy_ease(MOT_Z, HEAD_SPEED, 0, ZAXIS);
260 }
261 
262 
```

```
1  //
2  // Created by JP_Seeland on 11.11.2020.
3  //
4
5  #ifndef PUZZLE_LOGIC_H
6  #define PUZZLE_LOGIC_H
7
8  //define struct node
9  typedef struct node{
10     //depth
11     unsigned char g;
12     //heuristic
13     unsigned char h;
14     //puzzle
15     unsigned char puzzle[3][3];
16     //move to be made
17     unsigned char move;
18 }node_t;
19
20 void init_Node(node_t *node_ptr, unsigned char g, unsigned char h, unsigned
      char const *puzzle_ptr, unsigned char move);
21
22 typedef struct{
23     int x;
24     int y;
25 }Index;
26
27 /**
28  * calculates position of number in puzzle
29  * @param Index &index: pointer of Index
30  * @param int &puzzle_ptr: ptr from puzzle array
31  * @param int num: number of puzzle
32  */
33 void getIndex(Index *index, unsigned char const *puzzle_ptr, unsigned char
     e);
34
35 /**
36  * distance of puzzle-piece to destination
37  * @param *arr_ptr: pointer of current puzzle state
38  * @param *end_arr_ptr: pointer of final puzzle state
39  * @return int dist: returns distance as integer
40  */
41 int dist(unsigned char *arr_ptr, unsigned char *end_arr_ptr);
42
43 /**
44  * absolute values function
45  * @param numb: signed int
46  * @return: unsigned positive int
47  */
48 int my_abs(int numb);
49
50 /**
51  * deep copy of current puzzle to new puzzle. temp new puzzle required for
```

```
      heuristic recalculation
52    * @param current_pzl_ptr: parent puzzle
53    * @param new_pzl_ptr: child puzzle
54    * @param move: slide direction hole
55    */
56   void update_pzl(unsigned char const *current_pzl_ptr, unsigned char
       *new_pzl_ptr, unsigned char move);
57
58   /**
59    * deep copy of src_node. Makes an exact copy
60    * @param dest_node: pointer of start of the new node
61    * @param src_node: pointer of start of the old node
62    */
63   void cpy_node(node_t *dest_node, node_t *src_node);
64
65   /**TODO: too many copy operations! immense speed penalty!  -> push/pop
       elements by in-/decrementing counter!
66    * implements a push function with the option to sort the new element
67    * @param node_ptr: pointer to node that should be pushed
68    * @param agenda_ptr: pointer to start of agenda the node should be pushed
       to
69    * @param sort_active: 0 = new node pushed to position 0 of agenda, 1 =
       node sorted in agenda
70    */
71   void sort_agenda(node_t *node_ptr, node_t *agenda_ptr, unsigned char
       sort_active);
72
73   /**
74    * main funcion that implements the IDA* algorithm
75    * @param s_pos: pointer of the puzzle in initial state
76    * @param s_end: pointer of the puzzle in solved state
77    */
78   void solve(unsigned char *s_pos, unsigned char *s_end);
79
80   /**
81    * returns the depth the solution is at in the search tree
82    * -> should only be called after running solve function
83    * @return: unsigned char with number of tree depth
84    */
85   unsigned char getDepth(void);
86
87   /**
88    * returns the move at certain index (index corresponds to move at tree
       depth)
89    * -> should only be called after running solve function
90    * @param: i = unsigned char, 0 = root node (move 's'), from 1 = index of
       moves
91    * @return: unsigned char which is the move at index i ('l','r','u','d')
92    */
93   unsigned char getMove(unsigned char i);
94
95   #endif //PUZZLE_SOLVER_PUZZLE_LOGIC_H
96
```

```c
//
// Created by JP_Seeland on 11.11.2020.
//
#include <stub.h>
//self defined includes
#include "puzzle_logic.h"

void init_Node(node_t *node_ptr, unsigned char g, unsigned char h,
  unsigned char const *puzzle_ptr, unsigned char move){
    unsigned char i, j;
    node_ptr->g = g;
    node_ptr->h = h;
    for (i = 0; i < 3; i++) {
        for (j = 0; j < 3; j++) {
            node_ptr->puzzle[i][j] = *(puzzle_ptr + (i*3) + j);
        }
    }
    node_ptr->move=move;
}

void getIndex(Index *index, unsigned char const *puzzle_ptr, unsigned char
   e){
    unsigned char i, j;
    for (i = 0; i < 3; i++) {
        for (j = 0; j < 3; j++) {
            if (*(puzzle_ptr + (i*3) + j) == e) {
                index->x = j;
                index->y = i;
            }
        }
    }
}

Index arrInd, endInd;
int dist(unsigned char *arr_ptr, unsigned char *end_arr_ptr){
    int i, dist_temp = 0;
    for (i = 1; i < 9; i++) {
        getIndex(&arrInd, arr_ptr,     i);
        getIndex(&endInd, end_arr_ptr, i);
        dist_temp += my_abs((arrInd.x) - (endInd.x)) + my_abs((arrInd.y) -
           (endInd.y));
    }
    return dist_temp;
}

//effizienter mit abs() funktion oder mit eigener Schreibweise??
int my_abs(int numb){
    if(numb < 0){
        return numb*(-1);
    }else{
        return numb;
    }
}
```

```
51
52  void update_pzl(unsigned char const *current_pzl_ptr, unsigned char
      *new_pzl_ptr, unsigned char move){
53      Index hole;
54      unsigned char x, y, temp;
55      unsigned char i, j;
56      for (i = 0; i < 3; i++) {
57          for (j = 0; j < 3; j++) {
58              *(new_pzl_ptr + (i*3) + j) = *(current_pzl_ptr + (i*3) + j);
59          }
60      }
61      getIndex(&hole, new_pzl_ptr, 0);
62      x = hole.x;
63      y = hole.y;
64
65      if(move == 'l'){
66          temp = *(new_pzl_ptr + (y*3) + x-1);
67          *(new_pzl_ptr + (y*3) + x-1) = 0;
68          *(new_pzl_ptr + (y*3) + x) = temp;
69      }
70      if(move == 'r'){
71          temp = *(new_pzl_ptr + (y*3) + x+1);
72          *(new_pzl_ptr + (y*3) + x+1) = 0;
73          *(new_pzl_ptr + (y*3) + x) = temp;
74      }
75      if(move == 'u'){
76          temp = *(new_pzl_ptr + ((y-1)*3) + x);
77          *(new_pzl_ptr + ((y-1)*3) + x) = 0;
78          *(new_pzl_ptr + (y*3) + x) = temp;
79      }
80      if(move == 'd'){
81          temp = *(new_pzl_ptr + ((y+1)*3) + x);
82          *(new_pzl_ptr + ((y+1)*3) + x) = 0;
83          *(new_pzl_ptr + (y*3) + x) = temp;
84      }
85  }
86
87  void cpy_node(node_t *dest_node, node_t *src_node){
88      unsigned char i,j;
89      dest_node->g = src_node->g;
90      dest_node->h = src_node->h;
91      for (i = 0; i < 3; i++) {
92          for (j = 0; j < 3; j++) {
93              dest_node->puzzle[i][j] = src_node->puzzle[i][j];
94          }
95      }
96      dest_node->move = src_node->move;
97  }
98
99
100 //agenda
101 node_t agenda[255]; //31 = max pzl depth + puffer
102 void sort_agenda(node_t *node_ptr, node_t *agenda_ptr, unsigned char
```

```
      sort_active){
103     unsigned char i = 0;
104     node_t temp1;
105     node_t temp2;
106     if(sort_active == 1) {
107         //search for closest cost in agenda or find the end
108         while (((node_ptr->g) + (node_ptr->h) < (agenda_ptr[i].g) +
              (agenda_ptr[i].h)) && agenda_ptr[i].move != 0 && i <= 2) {
109             i++;
110         }
111     }
112     //save current agenda value
113     if(i%2){
114         //ungerader index
115         cpy_node(&temp1, &agenda_ptr[i]);
116     }else{
117         //gerader index
118         cpy_node(&temp2, &agenda_ptr[i]);
119     }
120     //save new node in place and save old value in temp
121     cpy_node(&agenda_ptr[i], node_ptr);
122     i++;
123     while(agenda[i].move != 0){
124         if(i%2){
125             cpy_node(&temp1, &agenda_ptr[i]);
126             cpy_node(&agenda_ptr[i], &temp2);
127         }else{
128             cpy_node(&temp2, &agenda_ptr[i]);
129             cpy_node(&agenda_ptr[i], &temp1);
130         }
131         i++;
132     }
133     //save last temp value in last free agenda slot
134     if(i%2){
135         cpy_node(&agenda_ptr[i], &temp2);
136     }else{
137         cpy_node(&agenda_ptr[i], &temp1);
138     }
139 }
140
141 void pop_first(node_t *agenda_ptr){
142     unsigned char i = 0;
143     while(agenda_ptr[i].move != 0) {
144             cpy_node(&agenda_ptr[i], &agenda_ptr[i + 1]);
145             i++;
146         }
147     }
148
149 Index hole_pos;
150 unsigned char head_x, head_y, depth = 0;
151 unsigned char temp_pzl[3][3];
152 unsigned char depth_move[48] = {0};
153 unsigned char led_switch = 1;
```

```
154 node_t temp_node;
155 node_t temp_first;
156
157 void solve(unsigned char *s_pos_ptr, unsigned char *s_end_ptr){
158 #define SORT 0  //no sorting, bc. of too many memory operations (faster w/ ↵
      o sorting)
159     unsigned char temp_dist;
160     unsigned char solved = 0, cost_bound = 0, new_max_cost = 0;
161     unsigned char last_move;
162     unsigned char i = 0;
163     //root node
164     init_Node(&agenda[0], depth, dist(s_pos_ptr, s_end_ptr), s_pos_ptr,    ↵
        's');
165     while(solved == 0){
166         //save movement at depth from expanding node
167         depth_move[agenda[0].g] = agenda[0].move;
168         if((agenda[0].g + agenda[0].h) <= cost_bound){ //make children
169             //set depth
170             depth = agenda[0].g;
171             depth++;
172             //toggle LED to monitor status
173             led(0,led_switch);
174             led_switch = (1-led_switch);
175             //save last move, exclude opposite move from tree (would       ↵
                result in a cycle otherwise)
176             last_move = (agenda[0].move);
177             //get hole pos
178             getIndex(&hole_pos, &agenda[0].puzzle[0][0], 0);
179             head_x = hole_pos.x;
180             head_y = hole_pos.y;
181             //copy node so it can be removed before children are pushed to ↵
                agenda
182             cpy_node(&temp_first, &agenda[0]);
183             pop_first(&agenda[0]);
184
185             if(head_y > 0 && last_move != 'd'){
186                 //legal up move
187                 update_pzl(&temp_first.puzzle[0][0], &temp_pzl[0][0],      ↵
                    'u');
188                 temp_dist = dist(&temp_pzl[0][0], s_end_ptr);
189                 if(temp_dist == 0){
190                     depth_move[depth] = 'u';
191                     solved = 1;
192                 }
193                 init_Node(&temp_node, depth, temp_dist, &temp_pzl[0][0],   ↵
                    'u');
194                 sort_agenda(&temp_node, &agenda[0],SORT);
195             }
196
197             if(head_x < 2 && last_move != 'l'){
198                 //legal right move
199                 update_pzl(&temp_first.puzzle[0][0], &temp_pzl[0][0],      ↵
                    'r');
```

```
200                 temp_dist = dist(&temp_pzl[0][0], s_end_ptr);
201                 if(temp_dist == 0){
202                     depth_move[depth] = 'r';
203                     solved = 1;
204                 }
205                 init_Node(&temp_node, depth, temp_dist, &temp_pzl[0][0],
                      'r');
206                 sort_agenda(&temp_node, &agenda[0],SORT);
207             }
208             if(head_x > 0 && last_move != 'r'){
209                 //legal left move
210                 update_pzl(&temp_first.puzzle[0][0], &temp_pzl[0][0],
                      'l');
211                 temp_dist = dist(&temp_pzl[0][0], s_end_ptr);
212                 if(temp_dist == 0){
213                     depth_move[depth] = 'l';
214                     solved = 1;
215                 }
216                 init_Node(&temp_node, depth, temp_dist, &temp_pzl[0][0],
                      'l');
217                 sort_agenda(&temp_node, &agenda[0],SORT);
218             }
219             if(head_y < 2 && last_move != 'u'){
220                 //legal down move
221                 update_pzl(&temp_first.puzzle[0][0], &temp_pzl[0][0],
                      'd');
222                 temp_dist = dist(&temp_pzl[0][0], s_end_ptr);
223                 if(temp_dist == 0){
224                     depth_move[depth] = 'd';
225                     solved = 1;
226                 }
227                 init_Node(&temp_node, depth, temp_dist, &temp_pzl[0][0],
                      'd');
228                 sort_agenda(&temp_node, &agenda[0],SORT);
229             }
230 
231         }else{
232             //max_depth reached -> don't go deeper, save node cost as new
                  max_cost
233             if(new_max_cost < (agenda[0].g + agenda[0].h)) {
234                 new_max_cost = (agenda[0].g + agenda[0].h);
235             }
236             //pop node of agenda
237             pop_first(&agenda[0]);
238             //continue with next element in agenda...
239         }
240         //if root of tree reached, set new max cost
241         if(agenda[0].move == 0){
242             cost_bound = new_max_cost;
243             //setup new root node to reiterate tree with new max_cost
244             init_Node(&agenda[0], 0, dist(s_pos_ptr, s_end_ptr),s_pos_ptr,
                    's');
245         }
```

```
246
247     }/* end while solved == 0 */
248     led(0,0);
249 }/* end solve */
250
251 unsigned char getDepth(void){
252     return depth;
253 }
254
255 unsigned char getMove(unsigned char i){
256     return depth_move[i];
257 }
258
259
```