# Sokoban Robot

Semester Project: Introduction to Artificial Intelligence

MSc in Engineering - Robot Systems
(Advanced Robotics Technology/Drones and Autonomous Systems)

The University of Southern Denmark


Written by:

Nathan Durocher
489942
nadur20@student.sdu.dk

Jan-Ruben Schmid
491531
jschm20@student.sdu.dk


Group number 12

15.12.2020

# Contents

# 1 Introduction

This report is based on an assignment during the AI01 course at the University of Southern Denmark. The assignment given, is to create a robot out of Lego Mindstorms which is able to autonomously execute a Sokoban quiz. This includes a solver for the quiz, a design of the physical structure of the robot and to give it the ability to follow lines, push objects and turn on a given map. Furthermore this report contains a listing of the tools used, a detailed description of the robots behaviours, the search algorithms used and performance testing of the described behaviours and features. The conclusion summarizes the result of the report and the performance on the competition day, as well as ideas for improvements.

# 2 Tools

In this section all tools are described which were used during the development process.
The Solver is an independent project, written in C#. It is an object oriented, high performance, high level programming language and part of the .NET framework. The outcome of the solver, which stores the entire path to solve the competition is stored in a text file, which contains each step as a global direction (left, right, up, down).
The ev3 dev [1] image is used on a LEGO® MINDSTORMS® EV3 intelligent brick to create the controller. It is a Linux image with a preinstalled python library and brickrun, a command line tool for launching ev3dev programs. The connection between computer and robot is established via Bluetooth, which is used to send the text file of directions and the executable Python files. As development environment Visual Studio Code is used with the LEGO® MINDSTORMS® EV3 MicroPython extension. This extension changes Visual Studio Code (VS Code) into an Integrated Development Environment (IDE) for the robot, which gave the ability to Build, Download and Run the source code from VS Code [2].
Before the robot starts execute the behaviours, the output of the solver (global direction) has to be converted to behaviours. This step get executed before each run where the global step description get read in and converted to behaviours before being stored in an array. This design has the advantage to minimize the execution time between each behaviour while the robot is moving by eliminating the need to open and close a text file at each step.

# 3 Robot Construction

A clean and robust robot design is essential to solve the path repeatably. As inspiration, a previously proven and tested design, provided by Lego was used [3]. The design starts from the bottom up with a chassis built primarily from the two large servo motors. The two are connected by multiple cross pieces to ensure structural integrity. From there, a large wheel is attached to each motors with a third "wheel" in the form of a ball bearing is connected at the rear to keep the vehicle level. The tires at the front originally had an extra gear to achieve higher speeds but were removed as they were found to be unnecessary and added error into the control of the robot. Moving up from the chassis, the front of the robot supports a cantilever structure to suspend two light sensors side by side and a plow mounted even further out. The decision to place the light sensors side by side in the front of the vehicle was made with the line following behaviour in mind. Placing the sensors at the front allows the robot to have feedback from its environment in the direction of travel. Furthermore having the sensors so close together provided a narrow field of view to keep the robot as close to the line as possible. Although this reduced its recovery ability, this was found to be a worthy trade off to have the best possibility to keep the vehicle straight on the line. The plow is large v-shaped wall placed in front of the sensors. The size and height of the plow were chosen to ensure smooth movement of the cans and to extent the "reach" of the robot if cans are misaligned on the cross lines. Originally the plow was a single pair of straight pieces extending at roughly a 45°angle but the higher point of contact when pushing resulted in cans tipping over. This was fixed by increasing the surface area of the plow lowering the point of contact. Finally, to power and control the robot, the intelligent brick was attached to the chassis and the cantilever. The description above can be seen in figure 1, where a gyro is also present on top of the intelligent brick. The gyro is a remnant from an earlier design and is not used other than for aesthetics.

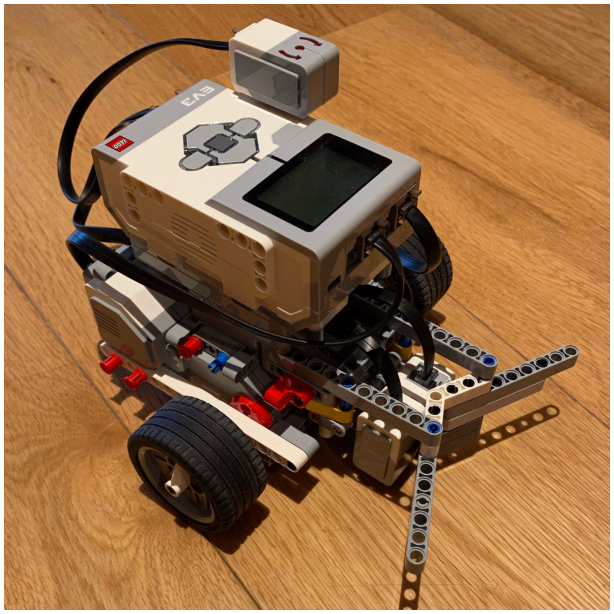

Figure 1: Final construction

# 4 Problem description

This section describes how the "Sokoban Quiz" get solved. This includes the map representation and the algorithm of the sokoban solver.

## 4.1 Map representation

With the target map being provided as image, it was made machine readable by manually conversion to a text file. The text file follows this representation:

- '#': wall
- ' ': free space
- '$': box
- '.': goal
- '*': box placed on goal
- '@': Sokoban
- '+': Sokoban on goal

The representation of the 2020 Map is shown in listing 2. The map get loaded into an array to be accessed by the solver.

## 4.2 Path planning

To solve the game, it has to be considered that the robot is moving in an optimal way. The optimal way can be interpreted as "shortest way" or the way with the lowest cost. The lowest cost means in this case, that each behaviour of the robot takes a different amount of time, which can be implemented into the algorithm. This means, the shortest path is not necessarily the fastest path. The Breadth-first search (BFS) algorithm is a complete, but undirected search and returns always the shortest path. The A* search algorithm returns a complete and optimal result. Both algorithm are using a hash keys to save each discovered state in the list. The solver returns a list of directions. The directions are always

in referenced in the direction of the top of the map. For the last push of a can, a capital letter is used, because the action of the last push of a can differs from a normal move. The first approach was the implementation of the BFS algorithm. After the final map were handed out, it was found that the result contained unnecessary turns, see figure 2. To remove this behaviour, the BFS algorithm was extended to the A*. In the following sections both algorithm are discussed in detail.

#### 4.2.1 Breadth-first search

The BFS is an uninformed search, which searches the graph in width for an element by expanding the single levels of the graph starting from the start node. At the beginning, a start node get selected, from which each edge get considered. If the new node is not discovered, it get added to a waiting queue and further processed in the next step. By repeating this procedure, the tree will get expanded until a solution is found. For the implementation of the algorithm 2 lists are required, an open list and a closed list, where the open list contains all undiscovered nodes and the closed list all discovered nodes.

#### 4.2.2 A*

The A* algorithm is similar to the already discussed BFS. Additionally, it calculates the cost for each step. If a node is in the closed list, it checks if the new path cost is less than the previous, if so, the node get overwritten by the path with less costs. With this approach, the knowledge of the Robot can be implemented into the algorithm to find the fastest possible route as defined by the cost function.

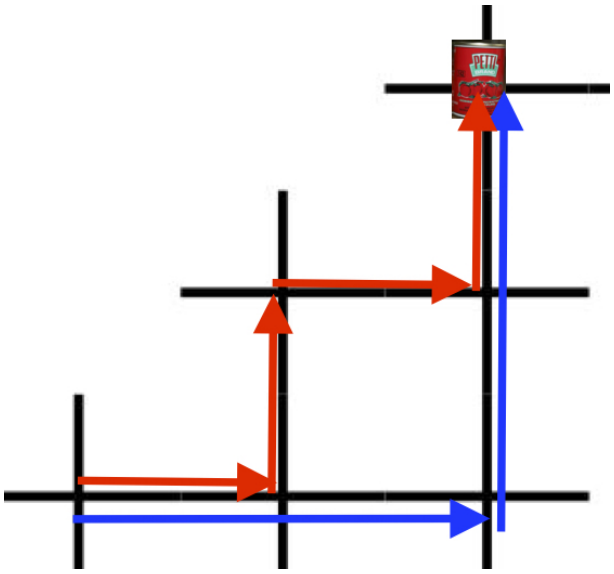

Figure 2: An illustration of the advantage of A* (Blue) versus BFS (Red), the A* path requires less turns providing a fast solution

### 4.3 Performance test

In a first approach, the solver was implemented in Python with the use of BFS. This resulted in an unreasonable high execution time of a few hours. Therefore, the code was ported to C#, where the execution time of the 2019 map was found to be 20 seconds. When increasing the complexity of the map to 6 cans, the execution time would get up to 120 seconds, which is still a reasonable amount of time compared to the increased complexity and the original implementation. Both algorithms, A* and BFS have the same complexity and therefore a similar execution time.

# 5 Behaviours

This section presents the details of the robot's behaviours. All behaviours were implemented with the ev3-dev python library [2] [4]. To perform the various movements to complete the game, the robot was required to move in straight lines and have the ability to turn in both directions. This led to 5 behaviours across the two classes being developed, as seen in figure 3.

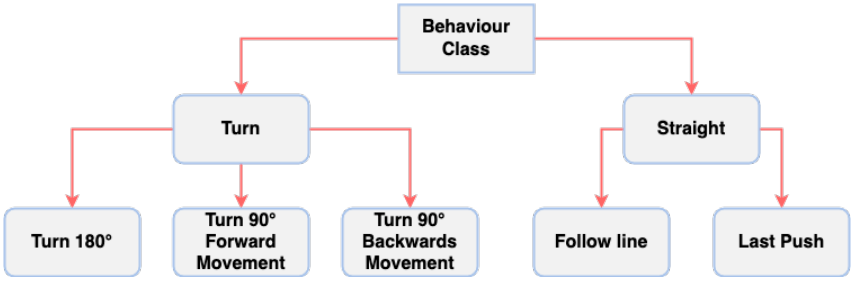


Figure 3: Behaviour class tree

The behaviour decision pipeline is included in figure 4. This image illustrates the workflow used to interpret the moves list from the solver. This process is critical to the integration of the two systems to change the game from a character list to a executable instructions for the robot.

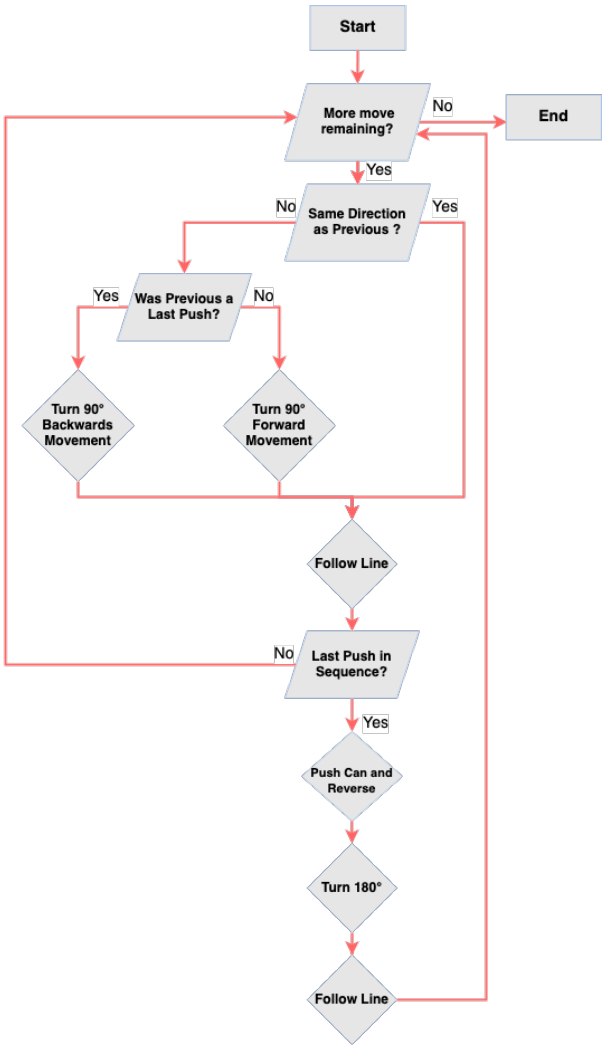


Figure 4: Behaviour pipeline

## 5.1 Linear motion

The most used behaviour and therefore, the main requirement of the robot was the ability to follow a line on the map. This behaviour uses the light sensors mounted on the front of the robot to detect the black lines against the white map background. The following function is designed as a proportional control with the motors speeds being adjusted based on the sensor readings. To begin both of the motors are started at 70% of the maximum speed, this was found to be the limit before the robot began to skip lines as the sensor's sampling frequency is not longer large enough to provide accurate information. Then, if either of the sensors reads above a specified white light threshold, the motor on the opposite side has its speed reduced by an amount proportional to the error. In addition the motor on the same side as the sensor has its speed increased by the same amount. The difference in speed causes the robot to turn slightly back towards the line. While adjusting the motor speeds the system is also checking for the cross lines on the map. To check for cross lines the system takes reading from both sensors and compares them to a threshold. If the combined value of the two sensors is greater, the robot will put the motors in a coast mode otherwise it will continue. Allowing the motors to coast instead of being stopped allows for smoother transitions between behaviours. The other linear motion behaviour is the last can push. This is the sequence required to position the can on a cross point followed by a return to the adjacent point as per the rules of the game. This behaviour is executed following the last push of the can in a single direction. Once the robot crosses the last cross line, it then moves forward with both motors a fixed distance, which was found through trial and error. Following that, it reverses with both motors another fixed distance to position its self to make a turn for the next move.

## 5.2 Turning

In a first approach the gyro sensor was used to determine how much the robot has turned. It was determined that the gyro sensor limited the overall turning performance in terms of speed and precision due to a insufficient sample rate. In a second approach the turning behaviour was implemented with a fixed amount for which the motors were to spin. This allows for the motors to be spun at full speed during a turn while having a higher precision. For this behaviour the function of listing 1 of the ev3 dev library is used.

```
motor.run_to_rel_pos(position_sp=degrees, speed_sp=1000, stop_action=coast)
```

Listing 1: Turning behaviour, implemented with ev3dev library [2]

As in figure 3, under the turn class the robot was designed to perform three different styles of turns. While all three turns are made at the maximum motor speed, the first two are very similar in execution and the third is an extension. The forward and backwards movement 90° turns are both designed in the same way with each actuating the motors by a fixed amount in opposing directions. During these movements one of the motors is driven forward while the other is driven backward. The counteracting motion allows the robot to rotate with minimal translation to stay on the map grid. The requirement of the second turn style was found during testing after implementing the preliminary forward turn behaviour. The forward turn was designed with a parameter to change the amount each motor turns which affects the translation during the spin. After optimizing this parameter going forward, it was found that if the robot attempted a turn following a reversing action or a dead stop, the translation was no longer acceptable. This led to the addition of the backward movement turn style that required its own tunable parameter, which was optimized separately. The root cause of the two different turn seems to be the "coast" action, since the previous motor speed has a significant impact on the turn behaviour. The final turn style is the 180° turn around, required only after a last push of the can. Following the reverse sequence of the last push behaviour if the robot needs to continue along a path directly behind it, it will perform a complete a 180° turn. This sequence is seen at the bottom of figure 4. The turn is designed similar to the forward motion 90° turn, but with a larger fixed total rotation of the motors. Again a tunable parameter is used to provide a means to control the translation during rotation, which was also optimized through repeated testing.

## 5.3 Tuning parameters

After the implementation of the different behaviours, it is critical to adjust different tuning parameters to get the best result. Because a change of the behaviour does not come with a full stop of the robot, the parameter depend on each other why a different set of tuning parameters is required, as well as a overall strategy.

### 5.3.1 Line following

The line following function has three different parameter, speed, gain and light threshold.

**Light threshold** compares the combined brightness of both light sensor with the given threshold. If the value is lower than the threshold, a line was detected and the line following section ends. This parameter is adjusted by measuring the overall light through the track.

**Speed** has the highest impact on the execution time, because it set's the speed of the robot while following a line. Changing the parameter has an impact on all turning parameters and the line follower gain. The maximum speed is limited by the reading time of the line sensors. It turned out that pushing the parameter to over 80 percent of the max speed results in a failing line following behaviour.

**Gain** adjustment of the the p-controller has an impact on how agile the line following works. The target is to get the robot as fast as possible back to the center of the line after a turn without getting a swinging system. This parameter has to be adjusted for forward and backwards moving of the robot.

### 5.3.2 Turning

As described in section 5.2, multiple turning styles are used which follow the same principle which is described in detail in this paragraph. Each turn has the independent parameters "totalDegrees" and "turnFactor".

**Turn speed** is fix with the maximum speed of the motors.

**TotalTurningDegrees** defines the total turning distance. This parameter has to be tuned to turn the robot as closed as possible to the target turning degrees.

**TurnRatio** is a factor which splits up the total turning degrees on both wheels. Adjusting this parameter changes the offset of the robot to the line.

**Push can** The can push defines the overall push and backup distance of the robot. Since two turning function have to be used anyway, both distances are separated to be capable of turning the robot before the line sensors detect the corner after backup. This reduces the overall distance and therefore execution time.

## 5.4 Performance testing

A comparative test for the speed at which a turn could be completed using the two approaches is described in table 1. To tests the performance of the final behaviours several debugging maps were used. These maps were combinations of a few behaviours in a row to observe the robot's execution before adjusting parameters to further optimize the system. To evaluate the turning behaviour a figure-eight map was made requiring four left turns and four turns in a row. Through repeated testing the behaviour was tuned until the robot was able to complete the figure-eight track 20 times in a row without fail. Similarly the turn around behaviour was tested by performing a spin followed by a line follow 10 times.

This setup mimicked the movement seen while attempting to solve the sokoban games. Additionally, to prove the observed poor performance of the gyro a timed test was conducted. As seen in figure 1 the turn behaviour is significantly slower when relying on measurements from the gyro.

| Time to turn based on Gyro (s) | Time to turn based on motor encoder (s) |
|---|---|
| 1.13 | 0.56 |
| 1.35 | 0.45 |
| 1.02 | 0.51 |
| 1.11 | 0.51 |
| 1.10 | 0.50 |
| 1.09 | 0.51 |
| 1.00 | 0.55 |
| 1.11 | 0.51 |
| 1.08 | 0.52 |
| 1.02 | 0.49 |
| Avg: 1.10 | Avg: 0.51 |

Table 1: Turning behaviour timing test results

# 6 Conclusion

As mentioned above, most of the initial design decisions were well thought out and as a result made the final product. The behaviour based design allowed a straight forward implementation of new behaviours, as well as adopting existing behaviours. Through the parameter tuning process, smaller adaptions and improvements were made, such as replacing the gyro with fixed turning degrees, and the removal of the 1:1 gear. These improvements resulted in a stable, robust and fast design which was only able to complete the course but won the competition by a significant margin, see the appended video [5]. While the robot performed outstandingly, there were a few areas that could have received more time and effort. For example, the space requirement of the algorithm can be significant reduced by saving only the positions of each moving item (can, Sokoban) instead of the entire map. Furthermore additional constrains could be implemented to reduce the required steps, for example a "dead corner" detection. The execution time could also be improved by a mechanism which pushes the can to the next line instead of driving there.

# 7 Appendix

```
XXXXXXXXXXXX
XX   X    @X
XX    $ $$ X
XX.. XX$  XX
X .. X  XXXX
X       XXXX
XXXXXXXXXXXX
```

Listing 2: 2020 Map representation

```
openList = []
closedList=[]
openList.add(startNode)
while(openList not empty){
    node = openList.dequeue();
    if(node == goalNode)
        return node;
    for (direction in possibleDirections){
        newPosition = Sokoban.move(node, direction);
        if(!closedList.contains(newPosition)){
            openList.add(newPosition)
        }
    }
}
return null
```

Listing 3: Pseudo code BFS

```
openList = []
closedList=[]
openList.add(startNode, 0)
while(openList not empty){
    node = openList.dequeue();
    if(node == goalNode)
        return node;
    for (direction in possibleDirections){
        newPosition, cost = Sokoban.move(node, direction);
        if(!closedList.contains(newPosition)){
            openList.add(newPosition, cost)
        else if(closedList[newPosition].Cost>cost)
            closedList[newPosition]=(newPosition, cost)
        }
    }
}
return null
```

Listing 4: Pseudo code for A*

# Acronyms

**BFS** Breadth-first search

**IDE** Integrated Development Environment

**VS Code** Visual Studio Code

# References

[1] *Ev3dev downloads*. [Online]. Available: https://www.ev3dev.org/downloads/.

[2] *Ev3dev-lang*. [Online]. Available: https://github.com/ev3dev/ev3dev-lang.

[3] *Ev3-rem-color-sensor-down-driving-base.pdf*. [Online]. Available: https://education.lego.com/v3/assets/blt293eea581807678a/blt8b300493e30608e9/5f8801dfb8b59a77a945d13c/ev3-rem-color-sensor-down-driving-base.pdf.

[4] *Ev3dev language bindings*. [Online]. Available: https://ev3dev-lang.readthedocs.io/en/latest/.

[5] *Full video of the robot at the competition day*. [Online]. Available: https://nextcloud.sdu.dk/index.php/s/bksip8BiFCMLmEm.