University of Southern Denmark

Introduction to AI 7th semester - Autumn 2019

Sokoban Game

Mathias Emil Slettemark-Nielsen
2302-1995
masle16@student.sdu.dk

Mikkel Larsen
0808-1996
milar16@student.sdu.dk

Group: 11

Supervisors: John Hallam and Xiaofeng Xiong

Project period: 02/09/2019 - 18/12/2019

Contents

1 Introduction

The Sokoban game is a challenging single-player game - for both man and machine. Sokoban is a computer puzzle game in which the player pushes jewels around a maze to place them in designated locations. This report describes how to solve the Sokoban game with a minimalistic robot. The LEGO Mindstorms EV3 set [1] is used to create the robot.

In this report the Sokoban problem is solved via a hybrid system which is illustrated in Figure 1. The solution of the Sokoban map is found separately on a computer and via a "Glue" is the solution translated so the controller on the robot understands what to do.

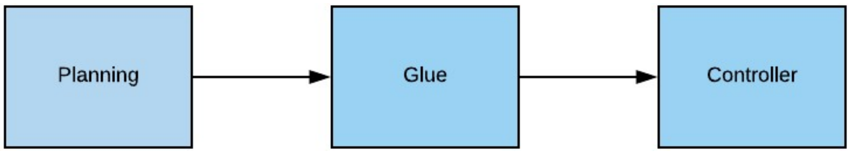


Figure 1: System overview of the system.

The code for the project can be found at [2].

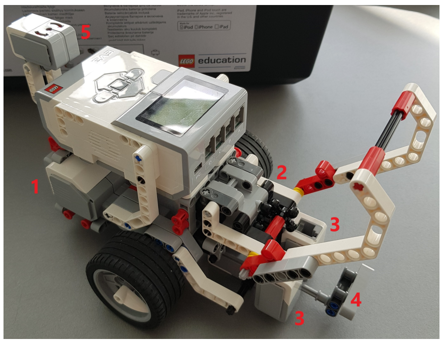
2 Design of the LEGO Mindstorm Robot

In this section, the design of the LEGO Mindstorm Robot will be described which consists of both the physical design choices and controller design choices. Firstly, the physical design choices are described in Section 2.1. Secondly, the controller design choices are described in Section 2.2.

2.1 Physical Robot design

The physical design of the LEGO Mindstorm robot will be described in this section.

The robot design is based on the LEGO building instruction for the Robot Educator [3]. Different modules are built separately and can be individually put on the robot, thus giving the robot a dynamic design. The first and second design iteration of the LEGO robot is illustrated in Figure 2.



(a) First design iteration of robot

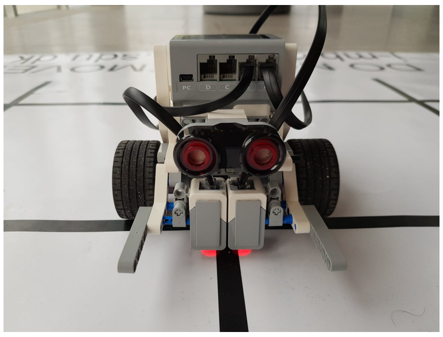
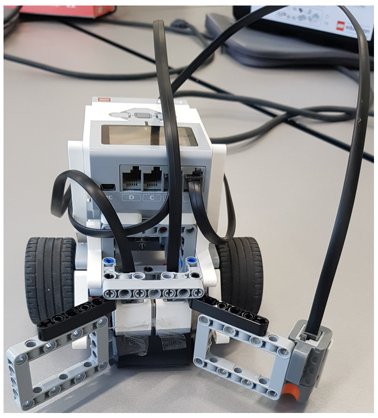

(b) Second design iteration of robot

Figure 2: First and second design iteration of the robot

In the first design iteration of the robot illustrated in Figure 2a it was quickly discovered many of the modules was not necessary, thus the second design iteration illustrated in Figure 2b was only built with 2 color sensors, 2 motors, and an ultrasonic sensor. This design was used until the robot had to push the jewels. It was discovered that the jewels could slide to the sides when pushing, thus make the robot unable to catch the jewels when pushed again. Furthermore, the ultrasonic sensor was deemed unnecessary. The final design iteration of the robot is shown in Figure 3


(a) Final robot design front.

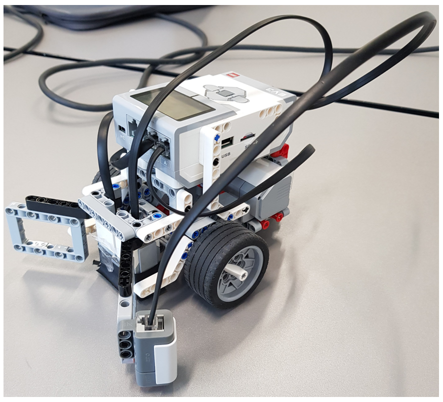

(b) Final robot design side.

Figure 3: The final robot design.

To counteract the jewels sliding to the sides when pushing, the gripper was re-engineered to the design shown in Figure 3. Furthermore, a light sensor was added to the gripper to get more robust intersection detection, than using both color sensors. The final design of the robot consist of the following modules.

1. Driving base module with EV3 microcontroller and 2 large motors
2. Grabber module
3. Color sensor module with 2 color sensors
4. Light sensor module with 1 light sensor

2.2 Controlling the Robot

The design of the control system for the robot will be described in this section.

A behavior-based control system divides the overall problem into behaviors, where each behavior is responsible to solve the task correctly. In contrast to classical robot control where a central controller controls the whole system and normally is advanced and difficult to modify, behavior-based systems are easy to modify and have a fast prototype pipeline. Thus, the control of the LEGO Mindstorm robot is done as a behavior-based system.

2.2.1 Behaviors

The behaviors in behavior-based systems can be interconnected with each other to either overrule one behavior from another or to give inputs to a behavior to modify it. All the behaviors used to solve the Sokoban puzzle is shown in table 1.

Number	**Name**	**Description**
1	Follow	Follow a line.
2	Intersection	Detect an intersection.
3	Forward	Run forward for a certain period.
4	Backwards	Run backwards for a certain period.
5	Turn left	Rotate left for a certain period.
6	Turn right	Rotate right for a certain period.
7	Turn 180	Rotate 180 degrees.

Table 1: Berhaviours of the robot to solve the sokoban puzzle.

Follow: The following behavior is used to keep the robot onto the line of the Sokoban map. The behavior is designed as a reactive controller, by taking the two color sensors outputs and using the outputs to set speed on the wheels. The left color sensor output is used to control the left motor and the right color sensor output is used to control the right motor. Thus, making a controller that is "scared" of black lines. The color sensor's values are scaled with a constant to control the speed of the robot.

Intersection: The intersection behavior controls whether an intersection has been detected or not. This is done by using the light sensor output. When the light intensity is lower than a specific threshold an intersection is present.

Forward: The forward behavior sets both wheels to a constant positive velocity in a specific time interval. The behavior is used to drive over intersections.

Backwards: The Backwards behavior sets both wheels to a constant negative velocity in a specific time interval. The behavior is used to reverse the robot away from a jewel after placing it.

Turn left: The turn left behavior starts by driving forward over the intersection in order to make a 90 degree left turn afterward.

Turn right: The turn right behavior starts by driving forward over the intersection in order to make a 90 degree right turn afterward.

Turn 180: The turn 180 behavior is used to turn 180 degrees after reversing away from the jewel, in order to position the robot correctly after a jewel push.

2.2.2 Decision rules for switching behavior

For the robot to solve the real-life Sokoban puzzle using the behaviors described above, decision rules are designed in order to execute the correct behavior. The decision rules implemented on the robot decide which behavior to execute based on a current action. The current action is from a sequence of actions generated beforehand by the path planner, which is described in section 3. A flow chart of the decision rules is shown in Figure 4.

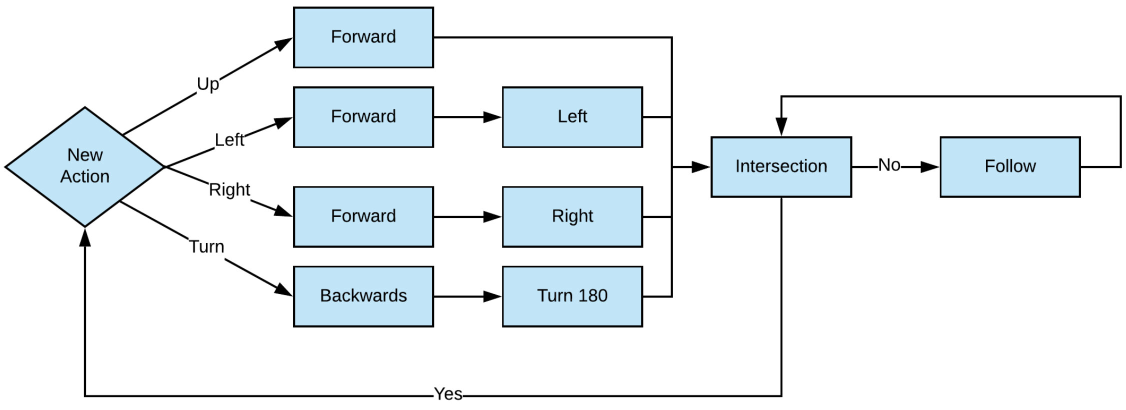


Figure 4: Flow chart of decision rules for executing behaviors. Diamond is function, Square is behavior. If there is no text on an arrow the condition is: behavior done executing.

2.3 Performance Evaluation of the Final Robot Design

In this section the performance of the final robot design is tested. Each test was conducted with three different color sensor scaling factors 1, 1.25, 1.5. The lower the color sensor scale the higher the speed of the robot. The following tests are conducted.

180 degree turns in a straight line: The robot was set to drive forward one time followed by a backward and 180 degree turn. This sequence was repeated 10 times, counting as one try. The robot did 10 tries.

Left and Right turns in a circle: The robot was set to drive around in a circle of 10 laps. The test was conducted 10 times each for right and left turns.

Intersection detection: The robot was set to drive up to the nearest intersection with the behavior follow. When detecting an intersection it drove backward and then repeated the sequence. This was done 10 times counting as one try. The robot did 10 tries.

Final map: The last test conducted was the robot running the optimal solution for the final map. The final map is illustrated in Figure 6e. The solution was found by the algorithm described in section 3. The robot only ran the solution once for every color sensor scaling.

All of the test described above is illustrated as GIFS and can be seen on [2] in the README.md file.

Results:
From the results of all the turning test in Table 2, it can be seen that the robot is robust when turning right or left and when detecting intersections because zero failure were observed. However, when looking at the 180 degree turn test it can be seen that the color sensor scaling has a big impact on repeatability. When the 180 degree turn test was conducted it was observed that the primary reason for failure was because the robot was located skew relative to an intersection right before a turn. This meant the robot overshot when turning because the turning is a hardcoded value.

	180 Degree turn	**Right turn**	**Left turn**	**Intersection detections**
Colorsensor Scaling	Mean number of rounds before failure	Mean number of rounds before failure	Mean number of rounds before failure	Mean number of rounds before failure
1	4	10	10	10
1.25	5.3	10	10	10
1.5	9	10	10	10

Table 2: Results for all the turning tests and the intersection test.

The results of the final map test is shown in Table 3. From the results, it can be seen that a lesser color sensor scaling value corresponds to a quicker time. However, when conducting the test it was observed that the robot was more unstable running at a lower color sensor scaling. Thus, if the test was to be run multiple times, the failure rate of the robot may yield to be higher with a lower color sensor scaling.

Color sensor scaling	**Time [Minuts]**
1	5
1.25	6.1
1.5	6.42

Table 3: The total time for completing the final map with corresponding color sensor scaling.

3 Design of the Sokoban Puzzle Path Planner

In this section, an algorithm for solving a sokoban puzzle is outlined. To solve a sokoban puzzle a graph search algorithm is used, the search creates a directed graph of states, thus the game must be represented as a search graph.

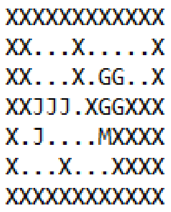
The Sokoban map is a two dimensional area that contains walls, jewels, goals, a player and free space. The walls indicate the constraint of the map. The jewels can be pushed around by the player. The goals are the desired placement of the jewels. The player can only perform the actions: up, down, right and left.

How the Sokoban puzzle is represented as a search graph and the algorithm to solve the sokoban puzzle will be outlined in 3.1. Lastly, the developed Sokoban puzzle path planner will be evaluated in 3.2.

3.1 The Algorithm to Solve a Sokoban Puzzle

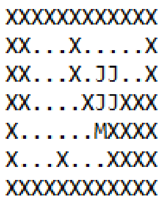
The object of the sokoban puzzle is to reach a specified goal state, such that all the jewels are placed on the goals. The formulations are as follows:

- **States:** A state description specifies the location of the player and the jewels.
- **Initial state:** Any state can be designated as the initial state.
- **Actions:** The simplest formulation defines the actions as movements of the free space: left, right, up or down.
- **Transition model:** Given a state and action, this returns the resulting state. For example, in Figure 5a if the action left is applied, the resulting state would have the M moved one to the left.
- **Goal test:** This checks if the state matches the goal state shown in Figure 5b.
- **Path cost:** Each step cost one, so the path cost is the number of steps in the path.



```
XXXXXXXXXXXX
XX...X.....X
XX...X.GG..X
XXJJJ.XGGXXX
X.J....MXXXX
X...X...XXXX
XXXXXXXXXXXX
```

(a) Initial state of the sokoban puzzle.



```
XXXXXXXXXXXX
XX...X.....X
XX...X.JJ..X
XX....XJJXXX
X......MXXXX
X...X...XXXX
XXXXXXXXXXXX
```

(b) Goal state of the sokoban puzzle.

Figure 5: Illustration of inital state and goal state of a sokoban puzzle. The "X" represents a wall, "J" represents a jewel, "G" represents a goal, "." represents free space and "M" represents the player.

When working with search graphs the amount of memory in each node must be considered, because the number of nodes can become very high when working with complex problems. Thus, it is desired to only store the necessary data in each of the reached states at each node.

It was chosen to store the previous state, the jewels positions, the player position and the action of the current state as stated above. Furthermore, it was chosen to use a breadth-first search algorithm to solve the sokoban puzzle. The information is stored in an object, where each object points at the previous object. Thus, by backtracking through the objects, from the goal state, a solution sequence of actions to the sokoban puzzle is found. This sokoban solver is referred to as solver 1 in Section 3.2.

However, solver 1 was not deemed fast enough thus a second algorithm was implemented and tested. The second algorithm has inspiration from [4]. Each node in the second algorithm stores the player position, wall, jewels and free space as a string. Furthermore, the sequence from the initial state to the current state is stored for each node, thus no backtracking is needed. Moreover, some optimized functions in python are also used. This sokoban solver is referred to as solver 2 in Section 3.2.

3.2 Performance Evaluation of the Implemented Solver

To evaluate the implemented sokoban puzzle solvers five different maps are used. The maps used are illustrated in Figure 6. The "X" character represents a wall, the "G" character represents a goal, the "J" character represents a jewel and the "M" character represents the player.

```
XXXXXXX
XG....X
X.....X
X.X...X
XXXX.XX
X.J...X
X.....X
X..XXXX
X....MX
X.....X
X...XXX
XXXXXXX
```

(a) Map 1.

```
XXXXXXX
XG....X
X.....X
XGX...X
XXXX.XX
X.J...X
X.J...X
X..XXXX
X....MX
X.....X
X...XXX
XXXXXXX
```

(b) Map 2.

```
XXXXXXX
XG....X
X.....X
XGXG..X
XXXX.XX
X.J...X
X.....X
X.JXXXX
X.J..MX
X.....X
X...XXX
XXXXXXX
```

(c) Map 3.

```
XXXXXXX
XG....X
XG....X
XGXG..X
XXXX.XX
X.J...X
X.....X
X.JXXXX
X.J.JMX
X.....X
X...XXX
XXXXXXX
```

(d) Map 4.

```
XXXXXXXXXXXX
XX...X.....X
XX...X.GG..X
XXJJJ.XGGXXX
X.J....MXXXX
X...X...XXXX
XXXXXXXXXXXX
```

(e) Map 5.

Figure 6: Maps used for performance evaluation of the implemented solvers.

In Figure 6, the five different maps has different number of jewels, thus the complexity of the maps rises from Figure 6a to Figure 6e. Each solver was stopped if it exceeded 15 minutes, as this was deemed the maximum acceptable solving time.

Map	Solver 1			Solver 2		
	Time [s]	Memory usage [MB]	Solved?	Time [s]	Memory usage [MB]	Solved?
Map 1	0.491	29.282	True	0.0101	11.788	True
Map 2	94.665	104.821	True	0.343	14.852	True
Map 3	900.343	327.049	False	4.700	28.484	True
Map 4	900.499	479.531	False	42.808	110.637	True
Map 5	900.451	496.239	False	18.781	99.533	True

Table 4: Results of performance evaluation of the implemented solvers on the maps shown in Figure 6.

From Table 4, solver 2 is faster than solver 1 and it uses less memory. For example, in map 1 solver 2 has a 97.9 % decrease in time compared to solver 1. Furthermore, solver

2 has a 59.7 % decrease in memory usage compared to solver 1 on map 1. On map 2 the decrease in time, from solver 2 to solver 1, is 99.6 % and the decrease in memory is 85.8 %. However, on maps 3, 4 and 5 solver 1 could not find a solution in under 15 minutes. Based on the results solver 2 is used.

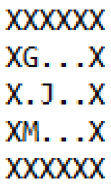
4 Combining the Path Planner with the Robot controller

In this section, the glue to combine the path planner to the robot controller is outlined. The action sequence from the path planner described in Section 3, which is a sequence of action choices to get from the initial state to the goal state, must be converted to a sequence the robot can understand. Two main problems are covered. One for positioning the robot correctly after a push action, and one for orientating the robot correctly on the real world Sokoban map in contrast when moving the player when finding the solution.

Positioning the robot: The sequence from the path planner is converted to the behaviors described in Table 1. This is done to replace the big letters, which indicates a push in the path planner, to the corresponding action and a turn action. For example, if the current sequence action is lowercase and the next action is uppercase, then a "Turn 180" action is inserted between the current action and the next action.

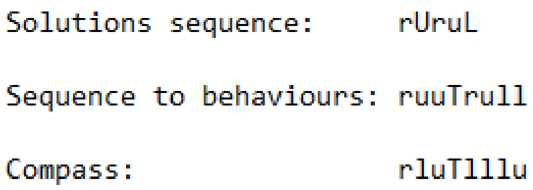
Orientating the robot: The behavior sequence is converted to corporate with the orientation of the robot, by implementing a compass which keeps track of the robot orientation on the map. At the first action in the sequence the compass is: "up", "right", "down" and "left", then depending on the next action the compass will rotate. For example, if the next action is "Turn 180" the compass will rotate 2, thus the resulting compass would be: "down", "left", "up" and "right". Thereby, keeping the right orientation of the robot throughout the whole map.

Figure 7 demonstrates the solutions sequence conversion from the path planner to the final sequence which the robot understands.



```
XXXXXX
XG...X
X.J..X
XM...X
XXXXXX
```

(a) Sokoban puzzle



```
Solutions sequence:     rUruL

Sequence to behaviours: ruuTrull

Compass:                rluTlllu
```

(b) The sequences to solve the sokoban puzzle shown in Figure 7a.

Figure 7: A sokoban puzzle with a sequences of action to solve it. The "Solutions sequence" is the sequence from the path planner. The "Sequence to behaviours" is the path planner sequence converted to behaviours. The "Compass" is the behaviours sequence adjusted to the robots movement.

5 Discussion

Informed Search Algorithm

In section 3, the used search algorithm was breadth first search which is an exhaustive search, thus a smarter search algorithm could have been used. For example, A*, which introduces a heuristic cost by computing the cost to reach the node, $g(n)$, and the cost to get from the node to the goal, $h(n)$. By combining $g(n)$ and $h(n)$ the cost of the cheapest solution through n, $f(n)$, is estimated. Thereby, when the goal is to find the cheapest solution, A* tries the node with the lowest value of $f(n)$. However, the difficult thing with A* is to calculate $f(n)$. A solution to estimate the heuristic could be to used breadth-first search, which was described in section 3.

Fail with Behavior Forward

In subsection 2.3 it was described that a large percentile of the failures were caused by the behavior forward. This was due to the robot being located skew relative to an intersection and after an intersection detection, running the behavior forward would result in the robot going off the map. A solution could be a lower speed and time length in the behavior forward when running over an intersection. However, when the speed and time length was lowered the robot would sometimes get stuck at intersections when pushing jewels, due to the behavior follow would activate before running over an intersection.

Potential Divider

When testing the robot it was observed that the robot could make large direction changes when trying to follow a line due to low color sensor scaling. This made the robot unstable and slower when trying to drive in a straight line. A solution to this problem is to make a potential divider [5] between the readings of the color sensors, resulting in less difference between the readings. This solution also makes the robot able to drive straight over an intersection without using the behavior forward.

Faster ev3dev Library

It was observed that the official library [6] sometimes caused inconsistent sensor readings. Research showed that a lighter third party library [7] was programmed to get higher pulling rates from the sensors. The results of the source [8] showed an improvement up to 11x faster pulling-rate on the color sensor if using the third party library. Thus, the whole third party library should be implemented on the robot. For even further optimization the programming language NXC [9] could have been used. NXC showed faster loop time when doing a line follower benchmark in [7], than python on ev3dev and python on ev3dev with ev3fast.

6 Conclusion

Throughout the report, a LEGO Mindstorm EV3 robot and a path planner algorithm were designed to solve a Sokoban puzzle.

The LEGO Mindstorm robot's final design was with two color sensors to follow a line and one light intensity sensor to detect intersections. Moreover, a set of behaviors was conducted making the robot able to move on a Sokoban map. The final design was able to perform right turns, left turns and detect intersections with no failure at any color sensor scaling tested. However, when performing a 180 degree turn the robot's robustness was only deemed acceptable at a color sensor scaling of 1.5. The robot is able to solve the final map, shown in Figure 6e, with a color sensor scaling value of 1 in 5 minutes.

The final path planner stores in each state the player position, walls, jewels, and free space, and uses a breadth first search algorithm. The final path planner is able to create a sequence of actions that solves the final map, shown in figure 6e, in 18.781 seconds with a memory usage of 99.533 MegaBytes.

For binding the hybrid system together, a "glue" was designed to translate the sequence from the path planner to a sequence the robot control system could understand. The two main problems the "glue" solved was the position of the robot after pushing a jewel, and the orientation of the robot when navigating on the Sokoban map. After the translation, the robot could successfully understand the sequence generated from the path planner algorithm.

References

[1] MV-Nordic. *LEGO® MINDSTORMS®Education EV3 Programmering er fremtidens håndværk.* 2017. URL: `https://www.mv-nordic.com/wp-content/uploads/2017/11/LEGO-Education-EV3-dk-1.pdf` (visited on 12/14/2019).

[2] Mathias Emil Nielsen and Mikkel Larsen. *LEGO sokoban game.* 2019. URL: `https://github.com/mikkellars/LEGO_Sokoban_game` (visited on 12/14/2019).

[3] LEGO-Group. *Building Instructions for Robot Educator.* 2019. URL: `https://education.lego.com/en-us/support/mindstorms-ev3/building-instructions?fbclid=IwAR2L2TbIbW93HWbv34NpbbeiZq03DsMRMBlJG3yRqeRasNe-l9xnZS0FY6A` (visited on 12/14/2019).

[4] *Sokoban.* 2019. URL: `https://rosettacode.org/wiki/Sokoban#Python` (visited on 12/14/2019).

[5] the free encyclopedia Wikipedia. *Voltage divider.* 2019. URL: `https://en.wikipedia.org/wiki/Voltage_divider` (visited on 12/17/2019).

[6] ev3dev. *Python language bindings for ev3dev.* 2019. URL: `https://github.com/ev3dev/ev3dev-lang-python` (visited on 12/14/2019).

[7] QuirkyCort. *Python language bindings for ev3dev.* 2019. URL: `https://github.com/QuirkyCort/ev3dev-lang-python-fast` (visited on 12/14/2019).

[8] Yoni Garbourg and Cort. *ev3fast – A Python module for faster ev3 interface.* 2019. URL: `https://www.aposteriori.com.sg/ev3fast-a-python-module-for-faster-ev3-interface/` (visited on 12/14/2019).

[9] John Hansen. *NXC Programmer's Guide.* 2013. URL: `http://bricxcc.sourceforge.net/nbc/nxcdoc/nxcapi/index.html` (visited on 12/17/2019).