University of Southern Denmark

INTRODUCTION TO AI E20 - GROUP 30

The Sokoban Robot

Martin Duong Dong Nguyen
mangu17@student.sdu.dk

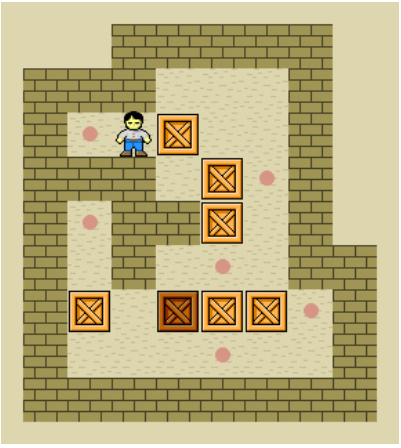
Jakob Grøftehauge Rasmussen
jakra17@student.sdu.dk

Handover Date: 18-12-2020

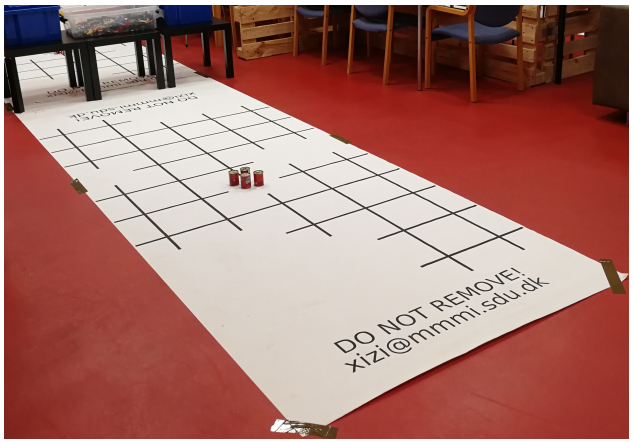
Contents

1 Introduction

Sokoban is a classic video game where a virtual player has to push crates around a warehouse. The virtual player can only move up, down, left and right and is only able to push a crate the forward direction. Furthermore the player is only capable of pushing one crate at a time. As seen in figure 1a the warehouse is confined by a number of walls, meaning the player and the crates cannot be pushed freely around further complicating the process of moving crates. In order to complete the game, the player has to move the crates to predefined storage locations. In figure 1a the storage zones is illustrated by the red dots [1].



(a) The classic Sokonban game [1].



(b) The Sokoban course to solve.

Figure 1: Illustrations of the Sokoban games.

The aim of this project is to create a robot cabable of solving a real life Sokoban course, similar to the one shown in figure 1b, in the least amount of time. The intersections of lines represents a grid cell in a classic Sokoban map, the lines connecting the intersections can be utilised to navigate between different intersections. The project involves developing an artificial intelligence which is able to find a series of actions completing a given sokoban map. Furthermore a robot which can navigate the environment has to be designed. For this to be achieved a number of behaviours has to be designed. The behaviour has to make the robot capable of navigating the environment and manipulating objects in the same manner as the virtual player in the original sokoban game.

2 Sokoban Solver

As mentioned in section 1 an artificial intelligence capable of determining a series of actions accomplishing the map has to be developed. To solve this problem a graph or tree search algorithm can be used. For this project it has been decided to apply a breath first search algorithm on a tree structure. Before the tree can be constructed and a searching algorithm can be applied, a way of representing the state of a Sokoban environment has to be designed. Moreover a framework for calculating the subsequent states of a state has to be developed in order to make the construction of the tree possible.

It has been decided to search for the solution in the classic Sokoban game and afterwards translating this solution to the behaviours of the robot. This is done in an effort to separate the development of the solver and the robot. The only parameters which can change between states is the position of the player and the position of the crates, why it has naturally been decided to only include this information in the representation of a state. This means that from the state itself it is not possible to determine neither valid subsequent states or to check if the given state completes the map. Instead the developed Sokoban framework will be responsible for handling this functionality. The framework holds information about the storage zones as well as the map itself. Using this information a method for calculating the next state and its validity has been implemented. The method needs information about the current states, the map layout as well as the action taken to get to the next subsequent state.

The tree is made up of nodes and constructs itself dynamically while it is being searched. A node in the tree consist of an object containing the Sokoban state as well as connections to the parent node and the children nodes. To keep track of which nodes should be expanded next a simple queue is utilised. The state of the Sokoban environment do not depend on previous actions meaning the subsequent states will be the same no matter how the state was entered. Therefore if all valid action in the tree is being expanded a lot of redundant search will be conducted. To eliminate this problem and thereby reducing the search space, it has been ensured that a state can only be expanded once. This is being done by generating a unique key using the player position and the position of the crates. A hash table is used to check if the node already has been constructed. If the key already exists in the table the node will not be generated. If it does not exist in the table the key is stored in the table and the node is generated. When generating the key the position of the crates is sorted according to their x and y position. This ensures that it does not have an influence which crate number is being placed where, and thereby further reducing the search space further.

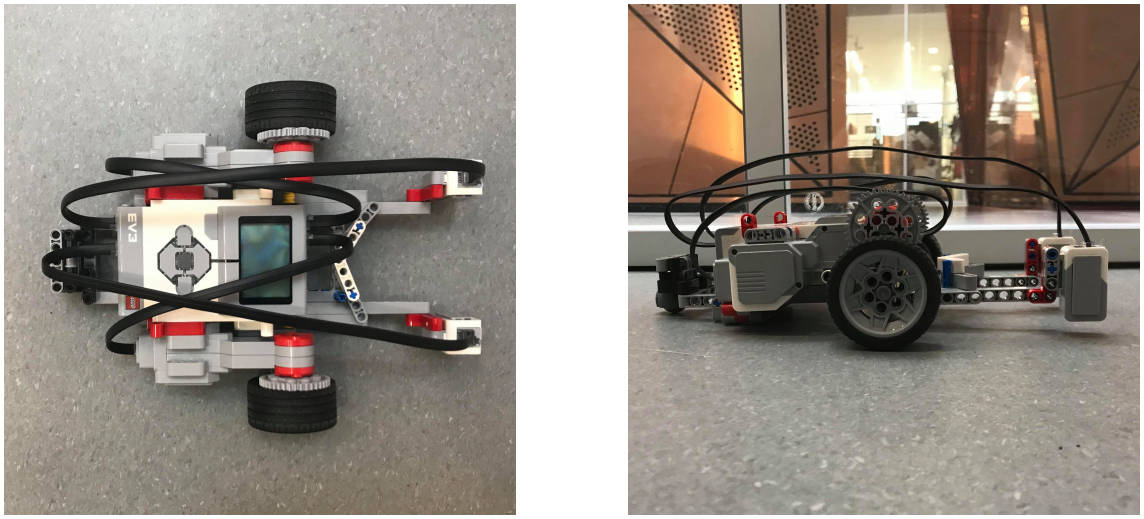
When the completion state is encountered the first time the search is stopped. It does not make sense to do additional search as the breath first search algorithm guarantees that the optimal path is found the first time the completion state is encountered. From the last node the tree is backtracked until the root node is encountered. For each node-pair the two states are compared in order to determine which action lead to the subsequent state. The determined actions is then stored in string, which will be supplied to the robot. The robot does not possess any information about the environment, why it is also indicated if an action manipulate one of the crates. This is done by comparing the position of the crates between two states, and adding a "." to the action manipulating a crate.

3 Robot

3.1 Physical setup and design choices

The robot will be build using a provided LEGO Mindstorms kit. When designing the robot, it is important to consider how the design will impact the overall performance of the robot. Initially the robot design was mainly focused on speed. Therefore a robot with a low center of gravity and

gearing on the motors was created. The initial design is depicted in figure 2. The light sensors located at the front of the robot was to be used for detecting lines perpendicular to the driving direction. The idea of having two sensors was to use the time between the detections of the sensors to correct the orientation of the robot. To keep the design simple and light a static passive U-shaped gripper was designed to enable control of the can without any active components, meaning the controller of the can is physically embedded into the design of the gripper. During early development of the initial design it became evident that the robot was not able to do corrections precise enough which resulted in the robot drifting out of course. It is expected that this was a result of a poor sampling rate on the sensor and a poor accuracy and repeatability introduced by the gearing.

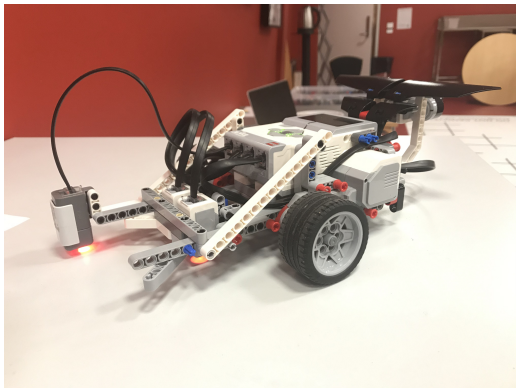


(a) Top View

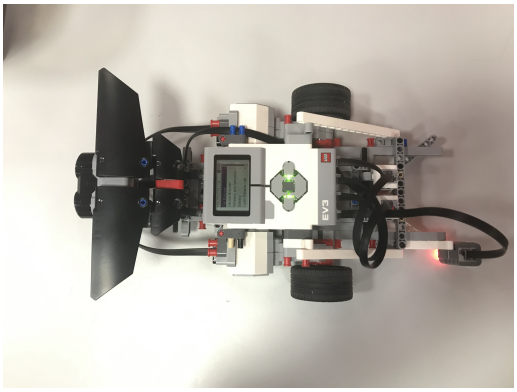
(b) Side View

Figure 2: The constructed robot platform

A new design with focus on rigidity and repeatability was created. The new design is inspired by the intial design, though the gearing of the motor is removed. Furthermore it was decided to develop a new line following method. As seen from figure 3, the two sensors used for detecting perpendicular lines is moved towards the center of the robot and are placed closer together. This is done to make it possible to follow the line parallel to the driving direction of the robot using the sensors. This setup makes it possible to do corrections more often, than only when a new grid cell is entered. An additional light sensor was added to the robot. This is to be used for detecting perpendicular lines, so the robot will know when a grid cell ends. A rear spoiler and lights are added for aesthetics.



(a) Corner View



(b) Top View

Figure 3: The final robot platform

3.2 Behaviours

In order to navigate around the course the robot needs to have some navigation behaviours. The player in the classic Sokoban game has 4 behaviours; up, down, left and right. As the robot is not equipped with omnidirectional wheels these behaviours cannot directly be used for the robot. Considering the design of the robot the minimal behaviours the robots should be equipped are: A behaviour for following a straight line one grid square, turn left 90° and turn right 90°. With these three behaviours it is possible to construct a chain of actions for navigating the robot between any two connected grid squares in the environment.

Some special physical properties exists in the Sokoban environment. The tomato cans can only be manipulated in a forward direction, meaning that the robot is not allowed to turn while holding the can. Therefore a fourth behaviour making it possible to separate the robot and the can will be designed. For simplicity it has been chosen to use a reverse motion to do the trick. Meaning a behaviour for reversing one grid square has to be designed.

For the line following behaviour a PID-controller is utilised. As described in section 3.1 the robot platform is equipped with two light sensors which is used for the line following behaviour. The input to the PID-controller is the difference between the raw values of the two sensors. The gains used for the controller are determined by trail and error. The behaviour is stopped when the stop sensor detects a line. When the behaviour stops both motors halts. To determine when a current sensor readout should be interpreted as a line a threshold has to established. This is done by letting the robot drive straight through five grids while recording the sensor values. The result of this can be seen in figure 4. From the figure it can be seen that the robot detects four lines and stops at the fifth and that each time the sensor values drops from approximately 54 to 25. From this the threshold for detecting the grid lines are empirically set to 45. Furthermore it is important to notice that the sensor values are quite stable and only drops significantly when a black line is crossed. The implementation of the line following behaviour is inspired by the provided PID line following algorithm from the EV3DEV2 library [2]. The EV3DEV2 library only supports line following using a single light sensor. Furthermore it is only possible to run the function for a specified amount of time, which is why it had to be modified for this project.

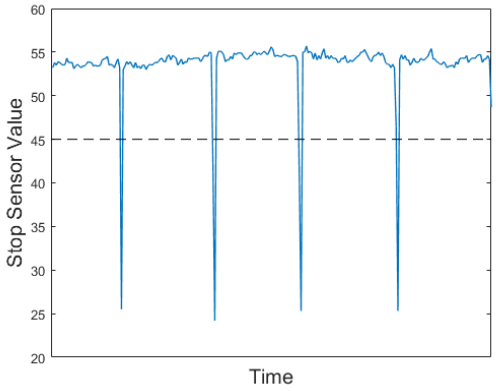


Figure 4: Stop sensor readout

For the turning behaviours a method is available through the EV3DEV2 [2] library to make the robot rotate a given angle. However since the robot are rotating around the wheel axle center the robot should align the with the grid cross. This means that whenever the robot should make a turn it should first have to drive forward until the grid cross and the wheel axle center are aligned. This aligning is done by driving a fixed number of wheel rotations before any turn. By implementing the turn behaviours as written a 180 degree turn will not be possible by for instance making two left turns after each other since the robot will drive a bit forward after the first turn and thus be misplaced. Therefore a third turning behaviour is implemented for doing the 180 degree turns.

The reverse behaviour is only used when a can has been placed, meaning that multiple successive reverse behaviour will never be used. For simplicity the reverse behaviour has therefore been implemented as a fixed number of wheel rotations, meaning no line following behaviour is applied while reversing. This means that the reverse behaviour rely heavily on the orientation prior to reversing in order to prevent the robot from drifting of course.

3.3 Decoder

As mentioned in section 2 the output from the solver is a string of actions. Each letter in the string represents an action. If the character "." precedes the letter it is indication that the can is being manipulated. The possible actions in the Sokoban environment does not directly translate to the behaviours of the robot. Therefore a python class for fetching each individual actions and translating it into number of behaviours has been developed. A string of identical actions is expanded to a turn behaviour followed by a number of forward behaviors equal to the number of identical and consecutive actions. The turn behaviours are found by comparing the initial direction and the wanted direction and utilising a switch case to get the correct turn behaviour. A string of identical actions manipulating the can is expanded as described above, though at the end of the string a forward and backwards behaviour are appended to account for the fact that the robot and can is located at the same grid cell when it is manipulated. This is not possible in the original Sokoban environment which is why the behaviours need to be added.

4 Evaluation of Performance

4.1 Sokoban solver

The developed solver has been tested on the practice map provided on Blackboard [3]. It has not been possible to verify that the found solution actually is the optimal path, though it should be noted that it is better than the solution provided on blackboard. The found solution consist of 108 steps, while the provided solution consist of 134 steps. The time to calculate the solution for the practice map is around 8-9 seconds using a standard consumer grade laptop running Ubuntu. This deemed tolerable given the fact that it is allowed to calculate the solution offline.

4.2 Robot

In order to determine the performance of the robot useful tests should be designed. Firstly it is interesting to do isolated test for the robot and its hardware. For this three test are designed, a line following test, turning test and reversing test. Since the goal of this project is to place tomato cans at desired locations, it is therefore also interesting to test the robot's ability to place a tomato can at a given point. Furthermore a test for speed and navigation on the grid is designed, as it is a performance metric of the Sokoban game. At last the performance of the robot will be tested on the provided map based on its completion time.

The line following behaviour itself will be tested on its ability to correct orientation for one grid square. This test will performed by placing the robot with a predetermined orientation different from zero degrees with respect to the grid. Then letting the robot do line following for one grid square and measure the orientation of the robot in order to determine the performance of the line following. This test will be conducted for respectively 5, 10, 20 and 30 degrees in orientation and will be repeated five times each.

Orientation [°]	5	10	15
Mean [°]	2.4 ± 1.1402	3.8 ± 0.4472	7.6 ± 0.8944

Table 1: Results from the line following test. Drive speed = 40, P = 0.2 and D = 0.1

Table 1 shows the results from the line following test. It is worth noting that the mean for each experiment is less than the respectively starting orientation even when taking the standard deviation into account. This confirms that the line following is working as intended. It is unsure why the standard deviations do not have any tendency, however it should be noted that only ten measures per experiment was taken.

The purpose of the turn test is to determine the repeatability and accuracy at which the robot can turn. This will be done by letting the robot making 90°turns both ways and then measure the deviation in orientation between the grid and the robot. By repeating this test ten times for each turn a statistical outcome can be made which will be used to determine the actual performance.

Turn Type	Left	Right	180°
Mean [°]	-0.8 ± 0.862	-2.5 ± 0.990	0.067 ± 0.884

Table 2: turn test all

The results from the turn test is shown in table 2. Based on the mean both the left and right turns seem to be turning less than desired. Additionally the right turns performs worse than the left turns. However this accuracy is deemed to be sufficient for the robot to navigate around the map. For the 180°turns, the mean deviation is better than both the left and the right turns. Additionally the turn accuracy is tested for different turn speeds. However this is only tested for the left turn exclusively to get a picture of how the mean and variations differs with increased turn speeds. Based on the results from table 3 it is noticeable that the mean increases as a function of the turn speed. Moreover it is interesting that the standard deviation does not seem to depend on the turn speed. This means that by manually adjusting the turns a higher turn speed can be ulitized while still maintaining more or less the same variation.

Turn speed [rpm]	50	60	80	100	170
Mean [°]	89 ± 0.9944	89.3 ± 0.9487	90.9 ± 0.8756	91 ± 0.9428	93.2 ± 0.9189

Table 3: Results from the turn test at different speeds

The reversing behaviour will be tested on its ability to reverse for one grid square. Since this is done without any line following it is interesting to observe the consistency and accuracy. This test will simply be performed by letting the robot reverse for one grid square and measure the deviation in orientation and distance from the center line. Repeating this process 10 times it is possible to examine the consistency and accuracy. The results for the reversing test is depicted in figure 5 where each observations are shown with its position relative to the desired position and an orientation relative to the grid. Based on the results it is clear that the position and orientation does not change much after reversing one grid. With these small deviations is it deemed acceptable for the robot's application.

	Time [min]
Initial	5.12,56
Optimized	4.16,94

Table 4: Course completion time.

Base Speed [%]	40	45	50
Execution Time [s]	24.99	23.14	21.54
Success Rate	5/5	5/5	5/7

Table 5: Results Changing base speed

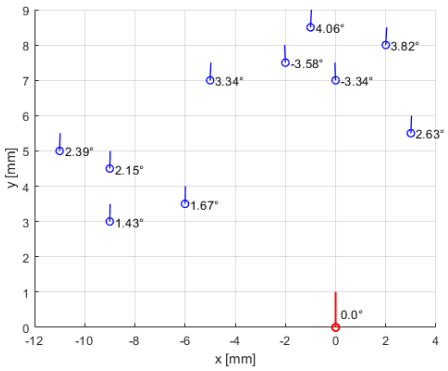


Figure 5: Position and orientation of robot center after reversing one grid square. The red marker the depict the wanted position and orirentation

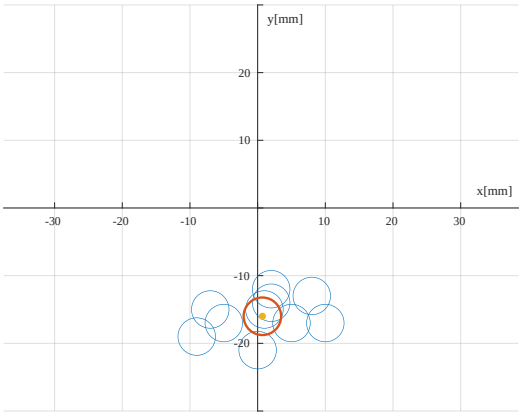


Figure 6: Position of cans after placement. The red circle depict the average position.

For the tomato can placing test a tomato can is placed one grid ahead of the robot where it then will be pushed one grid further by the robot. The distance between the tomato can center and the desired grid location is measured. This will be repeated 10 times to determine the performance of the tomato can placing. Figure 6 depicts the results for the can placement test, where the blue circles are the observations and the red circle with an orange dot indicates the mean position of the can. Based in this, it can be seen that the robot always places the can at a different position. However the variation around the mean is quite small and therefore negligible.

It it wanted to test how high a base speed the robot can use when driving straight, while still maintaining a high repeatability. The test is conducted by creating a small Sokoban course and have the robot complete the course 5 times. The above is repeated for three different base speeds. The result can be seen in table 5 which shows that the optimal based speed is 45 as it appears not to decrease the repeatability of the robot.

A performance evaluation of the optimised robot setup will be conducted on a full size map, giving an indication of how fast the robot can complete the competition map. The map on which the test is conducted will be the map provided on blackboard. The solution to the map will be calculated using the developed Sokoban solver. The test will be conducted 5 times, to give a accurate measure of the execution time and to check if the result is repeatable

5 Discussion

5.1 Optimizing the solver

In this project the emphasis has been put on developing and tuning the robot instead of optimizing the solver. Though our solver is capable of finding a solution relatively fast. It should be noted that it can be optimized in terms of time as well as number of nodes expanded.

A way of optimizing the solver could be by deploying another search algorithm. It would possibly be advantageous to deploy the A* algorithm. The A* algorithm is used for graph search, and uses a heuristic to determine which nodes in the graph should be expanded next. In order for the algorithm to work a heuristic has to be designed, but for the Sokoban environment it is not as simple as one might think. Close to the completion state it is relative easy to determine if an action results in a state closer to the completion state, though as the state come further away it becomes harder to determine. Furthermore the required book keeping of visited noted will increased. As the algorithm is used to searching graphs instead of trees. The graph introduced the possibility of cycles which needs to be taken care of in order for the algorithm to work properly. Furthermore the steps from a node to the starting state needs to be kept track of and updated as well. This increases the complexity of expanding a new node, though it is expected that the number of nodes expanded will decrease, so it is still expected that the overall execution time would decrease.

Furthermore the solver could be optimized even more by reducing the search space. The developed framework does not detect states where it easily can be determined that the state can never lead to the completion state. This could for example be that a crate has ended up i an location where it cannot be moved from. With the current framework the state can still be expanded and possibly resulting in a lot of irrelevant search. Though it has not been verified how large a part of the complete search space could be eliminated if this functionality was implemented. Though it is expected to have a positive impact in terms of the total number of expanded nodes and execution time.

5.2 Optimizing the Robot

The robot has only been tested and tuned in a single environment. It is known from previous experience that the sensor reading can be influenced significant by the amount of ambient lighting. Therefore a test of how different lighting conditions will affect the performance would have been preferable. Though this has not been possible as equipment for measuring the amount and composition of light in different environments has not been available. A simple solution to the problem could be to shield the sensor form ambient noise with the use of tape eliminating the gap between the ground and the light sensor.

6 Conclusion

A solver capable of determining a solution to any classic Sokoban map has been developed. The solver has not been significantly optimised in regards to speed, though it should theoretically be

able to lways determining the shortest path, this has however not been verified. Furthermore a robot which is able of navigating the Sokoban map has been developed. The robot is capable of translating a solution to a classic Sokoban map into a chain of behaviours which it can execute to complete a Sokoban course.

References

[1] (2020, oct) Sokoban. [Online]. Available: https://en.wikipedia.org/wiki/Sokoban

[2] (2020, oct) Ev3dev2 api reference. [Online]. Available: https://ev3dev-lang.readthedocs.io/projects/python-ev3dev/en/latest/spec.html

[3] X. Xiong, "2019_initial_positions_in_map.pdf," BlackBoard - Introduction to Artificial Intelligence, (E20).