Introduction to Artificial Intelligence - Final Report

Group 51: Askarbek Pazylbekov, Tamás Túri

December 18, 2019

Summary

During the assignment based on the published assignment [1], we created:

- a Lego Mindstorm robot, which would be able to execute tasks in the physical world, and moving objects along lines on a map placed on the floor.
- a solver running on personal computer, which would resolve the classical Sokoban game by informed or uninformed search algorithms and creates the set of operation for the robot

We publish and summarize our build and achieved results in the present document, and draw conclusion on the design decision we made.

Contents

1 Design concept and physical build of the Robot

1.1 Analysis of Selected Behaviours

To successfully perform the Sokoban Robot Challenge, it is crucial to determine the set of behaviors that will control the robot. For this challenge, we have defined four main features (listed from basic to more advanced):

1. **Calibration (Initialization)**: In this behavior, the robot is taking measurements from its environment (i.e. battery level, reference sensor values). These measurements are stored in variables and used in different modules of the robot. This behavior is required to "adapt" to different outside conditions, e.g. light intensity, dirt on the floor etc.
2. **Line-following and rotations**: This is a basic function of the robot to utilize the map and the information provided from the environment. Based on sensor values, the robot will alter the actuators, which are two Lego large motors on the side. The robot is using these sensor values and have no other information from the outside world.
3. **Sense of intersection (Cross-finder)**: When the robot passes through an intersection it shall be able to sens it and setup a counter for navigation. These intersections are crucial for the robot to navigate on the map and give signal which can be used to control loops in the program.
4. **Push forward objects**: This function is essential to complete the task since the robot needs to be able to push forward a given size physical object (size of a tomato can) through the map.

We have planned to use the below sensors and actuators to achieve the above goals.

Sensors (Inputs)	Actuators (Outputs)	Actions in Physical World
Color Sensor 1 Color Sensor 2	Left Motor Right Motor	Go Forward Turn Left Turn Right

Table 1: Sensors, actuators and action of the Sokoban robot

Each behavior is highly dependent on sensing information that sensors will receive, and the concept of operation of the robot. Therefore the type of sensors and their placement is a strategic decision to be made in the early state of the robot design.

1.2 Robot structure and concept of operation

By analyzing the rules, we identified two possible concepts for operation.

- Concept 1: In this concept, we utilize the ability to place a robot and can on the same spot. It has been clarified that this does not violate the rules of the challenge [1].
- Concept 2: Here the robot cannot occupy the same place as the can, therefore it needs to go around to move the target in a diagonal direction.

Since it is not allowed to grip or drag backward the can by ***Concept 2***, the selected concept for this robot could drastically affect the speed of the execution, also the rules to follow inside the solver.

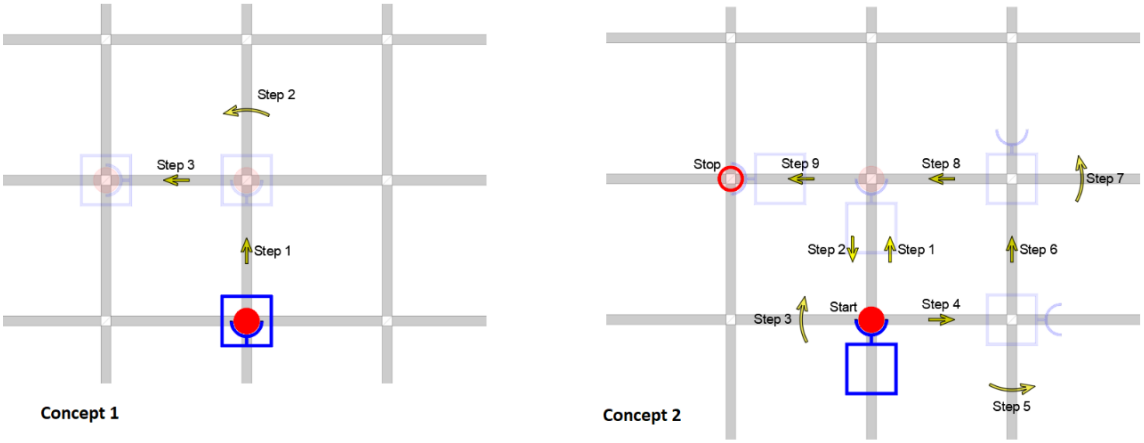


Figure 1: Dependence of the physical structure on the speed of task execution. Concept 1 - chosen physical structure, concept 2 - conventional structure

For instance, if the task is to move the can to left diagonal intersection, the robot with ***Concept 1*** have to execute the sequence in only 3 steps: *Go Forward* → *Turn Left* → *Go Forward* (See Figure 1)

If we consider conventional structure ***Concept 2***, the robot need to perform 9 actions: *Go Forward*→ *Go Backward* → *Turn Right* → *Go Forward* → *Turn Left* → *Go Forward* → *Turn Left* → *Go Forward* → *Go Forward*.

Before building the actual robot, we developed the concept of the physical structure to understand where to place sensors and actuators presented in Figure 2. When the robot reaches can, the can will be placed in the axis of rotation of the robot.

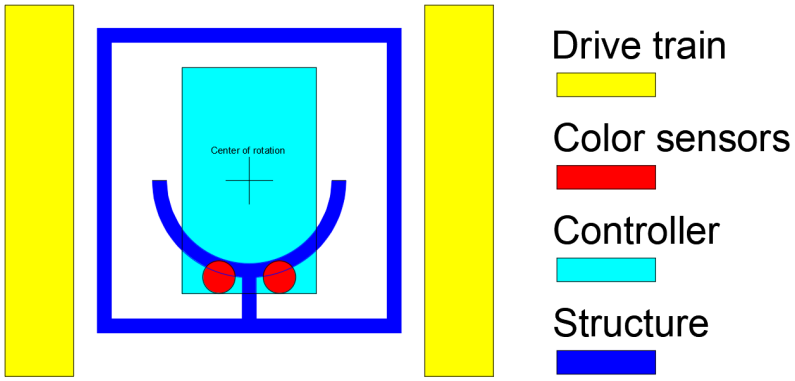


Figure 2: Structure of the robot with allocated sensors and actuators. Bottom view

As can be seen from the physical structure specifications (Figure 3) our robot has a tank chassis that allows rotating around one point in the world. We utilized this feature and left the space for the can underneath, such that the robot can easily rotate around the can.

After two different trials with other concepts of chassis, it becomes evident that the task cannot

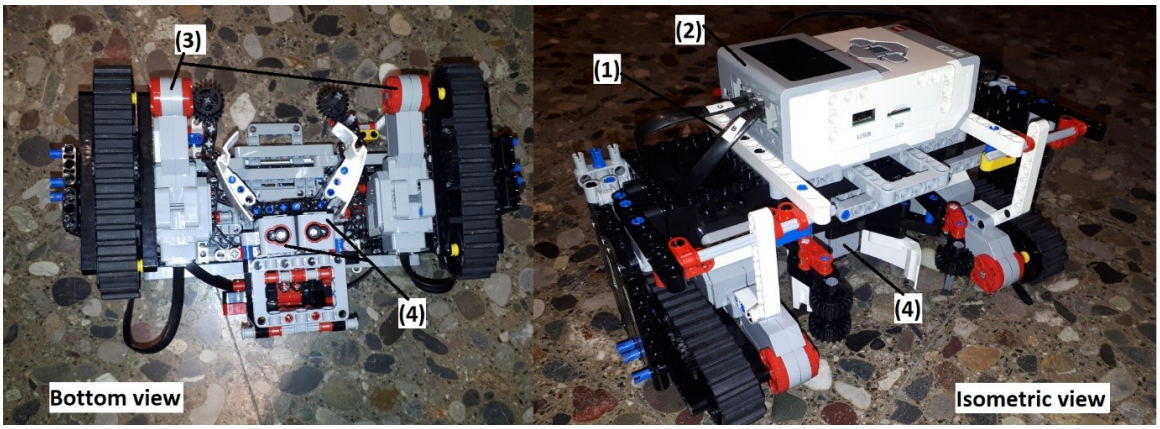


Figure 3: Bottom and Isometric views of the proposed robot. (1) - Physical Structure (Skeleton of the robot), (2) - EV3 controller, (3) - Motors, (4) - Color Sensors

be solved by ***four*** big wheels due to its inability to turn without excessive torque on the drive drain. The wheels shall be possible to steer itself, but it would make driving the robot over complicated. Therefore the selected drivetrain design is to use Lego tracks.

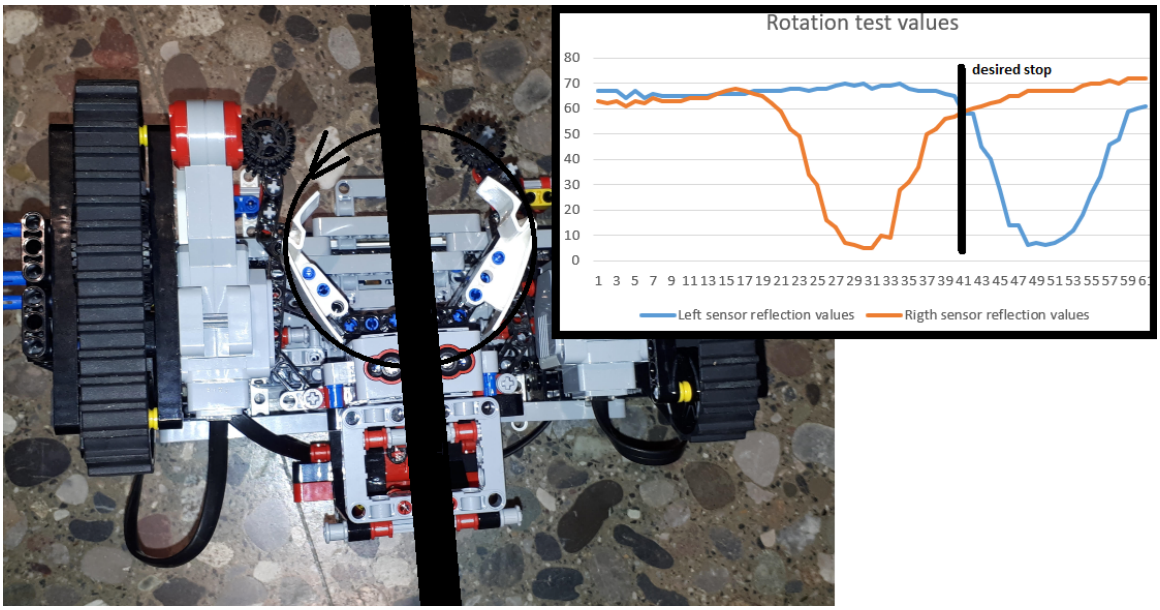


Figure 4: Bottom view of the robot with an imaginary black line that the robot needs to follow. The graph represents values from the right and left sensors

1.3 Behaviour planning of the robot

The script written in microPython based on the Lego EV3 EV3 MicroPython embedded controller platform [4] was designed to execute the below tasks (for the full code, please see the code on github repository [3])

- Behavior 1 - class Sokoban → def startup(): Changing indication light to red showing the robot is now in execution mode. The program is also checking the battery status and give error message in case the battery voltage is dropped below 7000 $[mV]$.
- Behavior 2 - class Sokoban → def calibration(): Placing the robot perpendicular to the line. The robot shall do calibration before execution the map. The purpose is to avoid variables which are not controlled during the execution and being able to measure and setup reference values which might change from the test site to the competition site (i.e. ligthning, dirtiness of the map etc.)
- Behavior 2.1 Robot find the line: The robot shall use left and right motor (motor_left, motor_right) at constant speed (speed_initializing) until its sensors (sensor_left, sensor_right) sense the line. Sensing means it has a sudden change in brightness values (i.e.: $>$ 50 if 0 is dark, 100 is full saturation of the image sensor pixels)
- Behavior 2.2 Sensor reference values: The robot moves back and forth 5 times and modify the reference values based on average sensor values measured, and return reference black and white values (black_ref, white_ref).

- Behavior 3 - class Sokoban → def rotate(direction): The left and right motor (motor_left, motor_right) rotates in opposite direction while the robot is not rotated to a minimum of around 40 degree. This is to disregard signal values at the beginning of the cycle. The way to acquire break signal is that extracting the sensor values from each other (i.e. (sensor_left-sensor_left_correction)-sensor_right-sensor_right_correction)) and setup a counter to find the approximate middle of the line to break movement. The method takes an argument of direction "left" and "rigth" and manipulate the drivetrain accordingly.

- Behavior 4 - class Sokoban → def gostraigth(direction): The robot follow the line based on proportional–integral–derivative algorithm using difference in sensor values $(error = self.sensor_left.reflection() - self.sensor_rigth.reflection())$, and creates a steering value as an output $(steering = ((error * self.Kp) + (integral * self.Ki) + (derivative * self.Kd)))$. The steering value then used proportionally, to reduce one or the another track speed by a maximum of 40% of the execution speed of the drivetrain.

- Behavior 5 - execution - Overall - The robot looping through the hard coded solution one by one (U: Up without can / D: Down without can / L: Left without can / R: Right without can / u: Up with can / d: Down with can / l: Left with can / r: Right with can) by using the methods gostraigth, and rotate(direction). For the demonstrative code finally uploaded into the robot, we did not make a difference between moving together with a can (small letter in the solution), or without the can (capital letter in solution). Obviously it shall be distinguished in a working concept to make sure precise movement of the can, dependent on the sensors would be utilized.

1.4 Discussion and conclusion for the robot build

The selected concept of physical structure limited our choice of placing the sensors "behind" the center of the robot. We knew that based on internet search and consultation, that is not recommended, but trusted that a "smart" code or an extra sensor would be able to solve these problems. Due to the color sensors are placed behind the center point of the robot, the implemented proportional-integral-derivative (PID) controller algorithm with the current sensor setup, it was not able to compensate for this design drawback.
Our team spent long hours on testing and trying to resolve that problem, however without success. One of the discussed possible solution would be a radical change in the sensor setup by utilizing computer vision, this way the signal (error in line following control algorithm) could be acquired from the front of the robot, rather then from the back. This was not possible to make from current Lego blocks straightforward, and would require radical change in the controller (i.e.: possibly utize even another type of controller like Arduino etc.)
As a very last attempt, one night before the competition day, we had changed the design of the robot and the controller code such that one extra color sensor are placed in the front side, ahead of the robot center. This design would be in conflict with the chosen concept of operation, since now could not be position the robot over the can in the axis of robot rotation. Therefore we realized we could only demonstrate the potential of the concept with this reduced number of operation, but not fully achieve the final goal to navigate through the map and being in competition with the other concepts.
Nevertheless, the designed ***Concept 1*** had major drawbacks which affected the performance and inability to finish the competition task successfully. This demonstrative robot concept and the code in operation can be seen in the uploaded video see []. The demonstration is far from the expected performance, and even it was crashed on the morning of the competition day.

2 Design of Sokoban solver algorithms implemented

For this competition, because of chosen physical configuration, our team had to develop own solvers. Therefore, three solvers were created, each utilizing three different algorithms:

- Depth-first search
- Breadth-first search
- A-star

Details of how these "classic" algorithms works are excluded from the report based on the content requirements posted on SDU blackboard, however the full code of Breadth first search, and A-star algorithm implementations are uploaded into github [3].
They are ready to run and test on a personal computer with Python interpreter installed, also short descriptions are provided on their function and demonstrative maps for each file.

2.1 Depth-first search implementation

For the reason of firstly selected physical design, it was required to rethink the conventional approaches for Sokoban solver widely presented online. Considering the fact that the robot could rotate around the can, the potential blocks that may occur in the corners of the map are taken into consideration as a valid step.
The working principle of the solver presented in this section is that it is not needed to define heuristics for the map, and it can solve any kind of map of four objects and four goal position (however, map should be defined in the following way: 'X' - stands for wall, 'G'- stands for goal position, 'J' - stands for object, can etc, 'M' - for robot, '.' - for clear path), since it is based on Depth First Search. The working principle is the following: because of four cans and four positions the whole task is divided into eight subtasks:

- Robot Initial Position to Can 1 $\Longrightarrow$ Can 1 to Goal 1$\Longrightarrow$ ... $\Longrightarrow$ Goal 3 to Can 4 $\Longrightarrow$ Can 4 to Goal 4

The working principle is very similar how a Depth-first search algorithms solve maze problems by navigating between a start and a goal position. This solver does that eigth times to pick up, and to move cans to a desired position.

2.1.1 Details of the non-classic depth-first search implementation

Firstly, the map which is stored in .txt file is divided into array of chars, such that it is easier to manipulate across the map using row-column system. For that special *coordinate* class is created, which stores row and column. Consequently, three main objects are searched and defined: robot position, goal and can, and they respective row and column values are assigned to them.

Then, the initial map is copied to the reserve map, where visited steps are indicated. The map is changed only after subtask is executed in a way such that start position is indicated as '.' and goal position is indicated as 'X' if robot with a can. The reserve map is copied after each subtask execution to clear the reserve map from visited steps of previous subtask and update the data of which can is already on the goal postion.

The solver function has three main arguments: start position, end position, and value that indicates if the robot with the can or not. Initially, the queue with appended start position is

declared. Then, the point is poped from the queue, stored to the another queue2, and if the point is equal to the goal position the function prints the array of chars (which consequently converted to the string); otherwise, the neighbours are explored using special function and visited coordinates are marked as 'T'. Once goal is found the coordinates are printed and appended to the final list of moves.
Resulted output for DFS solver:

- llllllRRRRRURULBdldllluRDRRURUBldllllluRRDRRURBdllllluRRRDRRUB
- Final list length is 60
- The mean elapsed time for the solver for 4 goal map is **0.087 seconds**
- Explored nodes: 125

We have modified the original map by placing the cans and goal in different coordinates. Our team believes that this high speed of execution is reached for two main reasons: the utilizing of second map instead of dictionaries (as it is implemented in different online solutions) significantly reduces time for searching around the visited map; additionally, since the solver chooses the first found solution it is also significantly reduces execution time. Moreover, the speed may be reduced by previously indicating the sequence of which can to which goal might be placed.

For 5 goal map the average time of execution is **0.12 seconds**. However, this number is not should be justified by real 5 jewel maps, we have checked the solver by placing fifth jewel and the fifth goal in the random place.

Listing 1: Example of main map after third can is placed

```
['X', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X']
['X', 'X', '.', '.', '.', 'X', '.', '.', '.', '.', '.', 'X']
['X', 'X', '.', '.', '.', 'X', '.', 'X', 'X', '.', '.', 'X']
['X', 'X', 'J', '.', '.', '.', 'X', 'M', 'X', 'X', 'X', 'X']
['X', '.', '.', '.', '.', '.', '.', '.', 'X', 'X', 'X', 'X']
['X', '.', '.', '.', 'X', '.', '.', '.', 'X', 'X', 'X', 'X']
['X', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X']
```

Listing 2: Example of "Visited" map after third can is placed

```
['X', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X']
['X', 'X', '.', '.', '.', 'X', '.', '.', '.', '.', '.', 'X']
['X', 'X', 'T', 'T', 'T', 'X', '.', 'X', 'X', '.', '.', 'X']
['X', 'X', 'T', 'T', 'T', 'T', 'X', 'T', 'X', 'X', 'X', 'X']
['X', '.', 'T', 'T', 'T', 'T', 'T', 'T', 'X', 'X', 'X', 'X']
['X', '.', '.', '.', 'X', 'T', 'T', 'T', 'X', 'X', 'X', 'X']
['X', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X']
```

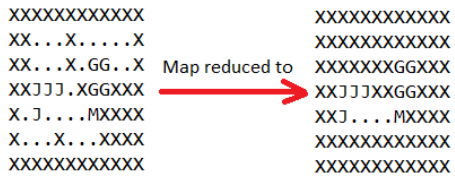
2.2 Breadth-first search implementation

This is a classic implementation of the Breadth-first search algorithm, however we were not able to obtain the solution in the final map due to execution time.
The root cause of that the implementation of such a heavy calculation can be problematic in Python which is an interpreted scripting language, and we could expect a 100 times faster execution in case the implementation is done in a compiled language like C++. The algorithm was however tested on smaller maps, and it was possible to obtain the final solution for our concept by closing down the nodes which would visit the area marked with red in the below picture. The full code is available to run, along with test maps on github [3] (see *Sokoban_BFS_Reduced_map.py*)

The Resulted output for solver is:

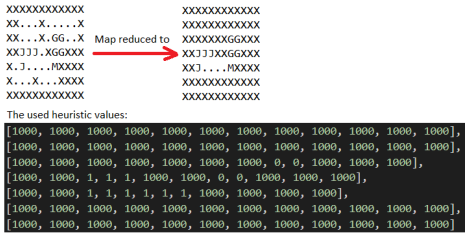
- LLLUdrrruuDDLLLULdULrrdrrruruDLDLLLLrrrrurLDLLLLLrrrrru
- Final list length is 55
- Elapsed time for solving 372.4 seconds
- Explored nodes: 30681080

This solution consist less steps then obtained through depth-first search, and it was used to hard code the exucution of the map into the robot.



2.3 A-star implementation

We could conclude that similarly to the breadth-first search algorithm that Python implementation is not perfect for this algorithm. However the number of visited nodes are reduced significantly compared to breadth-first search on the same map (see side by side comparison in the next section). Even if the number of nodes visited were less, due to implementation of sorting the list of unexplored nodes based on their weight this advantage has been reduced as the number of unexplored but obtained nodes increased, the sorting operation reduced the performance of the code. So even by employing reduced maps and heuristic as showed in the picture below, no complete solution was achieved.



Algorithm	Average execution time	Number of explored nodes	Comment
Depth-first search	0.087 [s]	125	
Breadth-first search	0.05 [s]	688	$Sokoban\_BFS\_Quick\_test\_map.py$
Breadth-first search	372,4	30681080 [s]	$Sokoban\_BFS\_Reduced\_map.py$
Breadth-first search	over 3 days it was killed	over160 million	$Sokoban\_BFS\_Competition\_map.py$
A-star	0.0001 [s]	56	$Sokoban\_A-star\_Quick\_test\_map.py$
A-star	0.047 [s]	3208	$Sokoban\_A-star\_Competition\_map(1box).py$
A-star	185.4 [s]	249008	$Sokoban\_A-star\_Competition\_map(2box).py$
A-star	over 3 days it was killed	not known	$Sokoban\_A-star\_Reduced\_map.py$
A-star	over 3 days it was killed	not known	$Sokoban\_A-star\_Competition\_map.py$

2.4 Conclusion and comparison of the implemented solvers

In the below table we summarized the performance of the three separate implementation of the solvers for comparison reason.

It is visible that the fastest implementation was achieved with the depth-first search solver, mostly because it was the most simplistic solution for the problem by drastically reducing the number of possibilities due to reducing the task into maze solutions. The implemented A-star and Breadth-first search solvers was

References

[1] *Introduction to Lego Kits: Building an Embodied AI Mobile Platform.* SDU - Introduction to Artificial Intelligence, 2019.

[2] *Video taken on 2019-12-02 with the new demonstrative concept*[Online]. Available: https://www.youtube.com/watch?v=DleCN5NC1Oc

[3] GitHub repository, *Introduction to Artificial Intelligence - Sokoban solver*[Online]. Available: https://github.com/tturi/SDU_IntroToAI

[4] Lego EV3 MicroPython, *Program in Python with EV3* . Lego, 2019[Online]. Available: https://education.lego.com/en-us/support/mindstorms-ev3/python-for-ev3