Advanced Robotics - Report 1

Balázs Tóth (bato@itu.dk), Boris Karavasilev (boka@itu.dk), Michele Imbriani (miim@itu.dk)

September 28, 2022

Contents

1 Week 1 - Designing the LEGO robot + basic control

This week's goal was to build a robot that is able to go from an initial position and orientation to a target position and orientation as precise as possible. We were provided with a set of LEGO Mindstorms EV3 components which included a controller, two motors, one light sensor, one colour sensor and a push sensor. We also had access to several boxes of LEGO Technic parts which we could use.

1.1 Designing the LEGO robot

To reach the target position and orientation our robot had to be able to move straight, turn a certain amount of degrees and stop at the target. We paid attention to keeping our design extensible because knew that we had to attach sensors to the robot later on. The sensor layout is further discussed later in the report.

First, we made a base for our robot that allowed us to easily attach and detach the Intelligent Brick holding the batteries. This design decision saved us from having to partially reconstruct our robot every time we had to replace the batteries.

We decided to make a three-wheeled robot. Two of its wheels are motorised, and the third caster wheel is added for stability and low friction. We aimed for precision over speed, so we chose the smallest available wheels and placed them fairly far from each other. The final version of the robot can be seen in Figure 1.

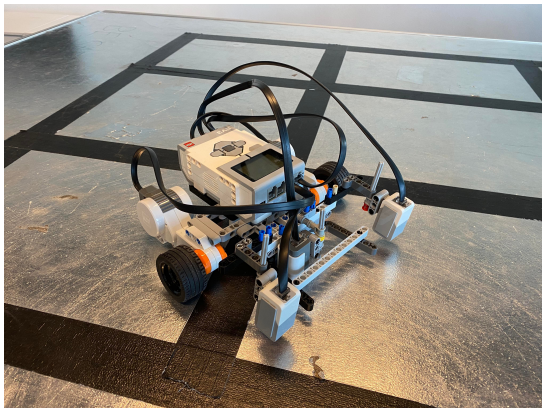


Figure 1: Final version of the robot

The grabber is placed on the front of the robot, as it can be seen in Figure 1 . It is designed to be able to push cans and keep them in front of the robot throughout the movement. We designed the grabber to be in line with the 2 sensors which are sensing the cross sections, so the cans be precisely delivered.

1.2 Programming of the movements

We decided to use the `DriveBase` class from the `pybricks.robotics` module to not have to implement simple functionality such as making our robot turn a specific number of degrees.

To initialize the `DriveBase` class we provide it with the following arguments: the ports of the motors (Port A and B), the diameter of the wheels ($42\,mm$), and the length of the imaginary axle ($190\,mm$). We

also set the speed($1500\,mm/s$), acceleration($500\,mm/s^2$), turn rate ($100\,deg/s$) and angular acceleration ($100\,deg/s^2$)

We used two methods to reach the target. The `driveStraight(distance in mm)` makes the robot drive straight for a given distance, and the `rotate(deg)` makes the robot turn a specified angle.

1.2.1 Solving the first challenge

The first challenge for our robot was to be able to reach a specific coordinate from a starting point. We got 3 different targets, and the final one was revealed at the beginning of the competition. There are 2 approaches to solving this problem. The first one is to turn the robot at the starting point and then drive straight to the goal. The second one is to first drive the robot straight, make a 90 degrees turn to face the target and drive straight again to land on the goal. Since we had difficulties measuring angles precisely, and a small deviation could result in a massive miss of the target, we decided to use the second approach. That way we were able to minimise the error introduced by an imprecise measurement of the angle between the starting and target position.

2 Week 2 - Behaviour-based robotics and experimental methodology

The goal of this week was to randomly navigate a 4x4 grid. The grid was marked with 5 cm wide black tape on a reflective surface. We were provided with 3 colour sensors able to detect discrete colours or measure the amount of reflected light. We decided to use the sensors in the second regime.

2.1 Experiments

We performed a series of experiments where we collected data about the readings from the sensors at various distances from the board. The readings of the sensors were measured above the reflective surface as well as above the black tape at the same distances. The collected data is presented in table 1 and plotted in figure 2.

Experiment number	1	2	3	4	5	6	7
Distance (mm)	50	40	30	20	10	5	1
Sensor value (no tape)	6	8	13	29	95	100	34
Sensor value (tape)	0	0	0	2	4	10	4

Table 1: Sensor data collected in experiments

We found out that the distance of the sensors from the ground significantly influences the readings. In order to be able to adjust the height of each sensor to be within the ideal range from 10 mm to 5 mm we attached each sensor by a rod with adjustable length as seen in figure 3. We define the ideal range as the one with the biggest difference between the maximum and minimum values, therefore with the highest precision.

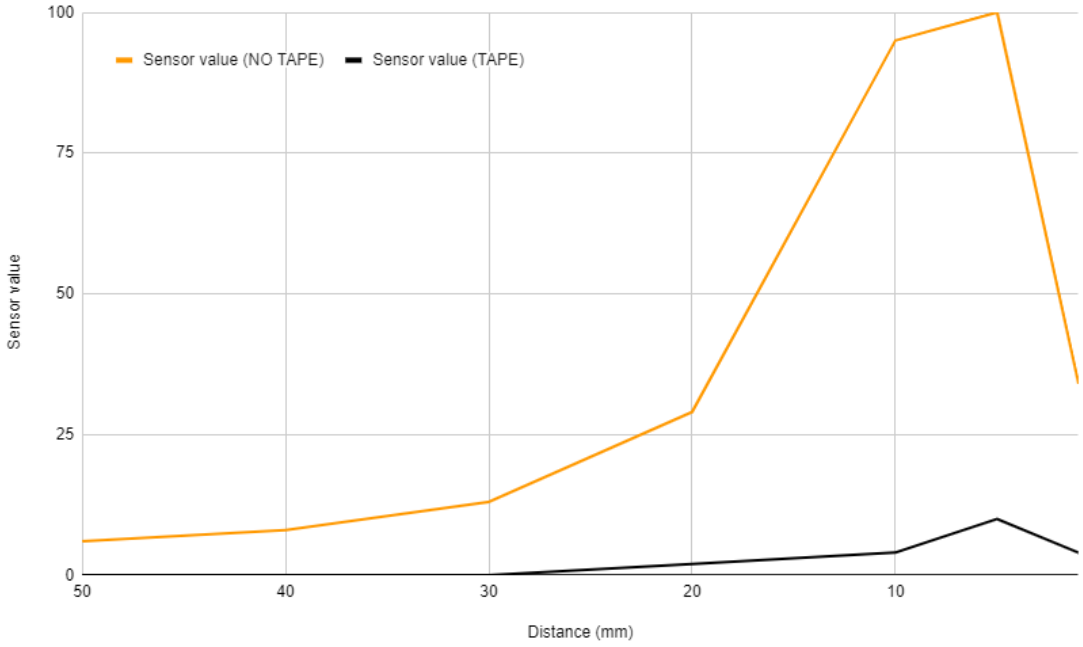


Figure 2: Sensor values depending on the distance from the ground

2.2 Layout of the sensors

In order to detect all possible types of intersections and precisely follow the grid lines we considered various layouts of the colour sensors and settled on the layout depicted in figure 4.

Our sensor layout enables our robot to both precisely follow the edge of the tape and is able to detect intersections. Beyond detecting intersections our robot is able to detect the following types of intersections: left turn, right turn and t-junction. Unfortunately, we were not able to identify a junction with four exits. Because the classification of all types of intersections is unnecessary for random navigation of a maze or execution of predefined plans our solution is sufficient.

2.3 Implementation of a behaviour-based controller

Our implementation divides the robot's behaviour into the state of line following and intersection crossing. The robot follows the edge of the tape until it reaches an intersection. Afterwards, the robot takes a decision either randomly or based on the predefined route if it shall turn and how many degrees it shall turn. In the case where it shall go straight, it blindly drives forward a fixed distance disregarding the inputs from the sensors. After the intersection has been crossed the robot returns to the line following state.

2.4 Line following implementation

We use one of the sensors to follow the right edge of the line. We chose the edge as a target as it sits approximately in the middle of the spectrum of the sensor's readings. When the sensor is entirely on top

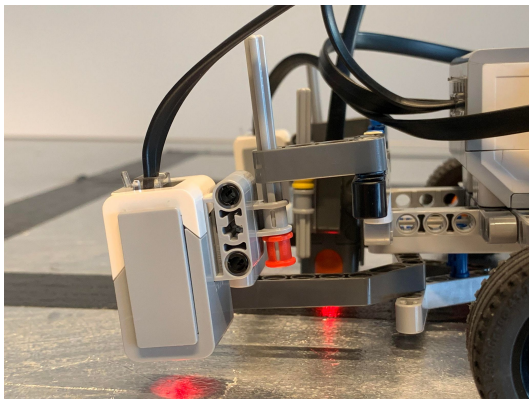


Figure 3: Variable height sensor attachement

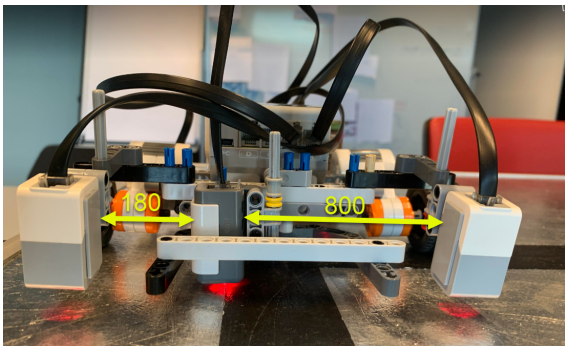


Figure 4: Sensor layout from the front

of the black tape it returns values smaller than 10. In the opposite case when the sensor is on top of the reflective surface it returns values higher than 90. We use proportional control with a gain constant to correct the robot's error which is calculated in the following way:

$$error = \text{sensor reading} - 50$$
$$\text{turn rate} = error * gain$$

3 Week 3 + 4 - Hierarchical and hybrid architectures

During weeks 3 and 4 our goal was to make a robot controller based on a hybrid architecture consisting of two parts. First, the planner solves a given maze on a PC and generates a list of instructions such as go straight, turn 90° etc. Second, the robot controller executes those instructions on the robot. To tackle this challenge we were provided with a text file containing a maze represented by characters, an explanation of what each character means and we were also given a hint during the lecture that we can solve this problem using a search algorithm such as BFS or A*.

3.1 Planner

The planner receives a text file as an input and outputs a list of instructions to another file. At first, the planner parses the text file into appropriate data structures. Afterwards, it either solves the maze and prints statistics about the process in the console or prints that the maze has no solution. An example of the console output can be seen in figure 5. A special feature of our planner is that it can display the animated solution in a console as seen in our repository[1].

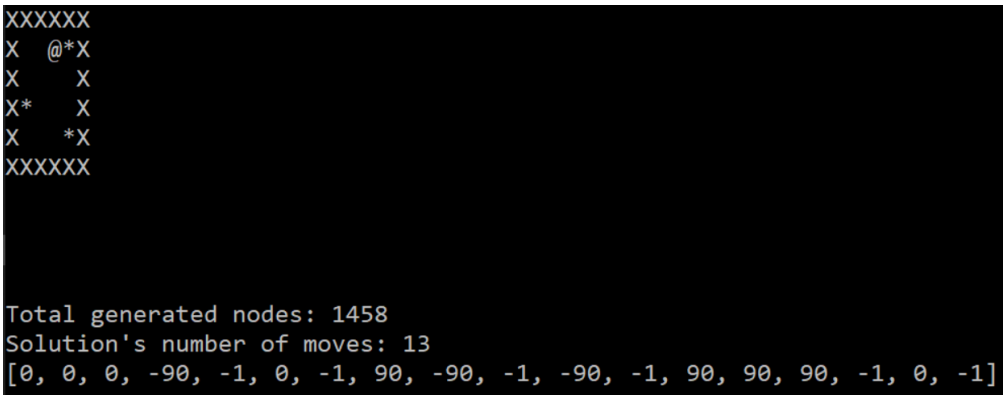


Figure 5: Output of the planner

3.1.1 Parser

We decided to call each character in the input file a tile. We refer to each tile by its ID number which also represents its coordinate in the flattened map. The tiles are numbered from the top left corner starting at number one and the numbers wrap around the end of each line. We demonstrate this on an example in figure 6. The parser creates an instance of a `Map` class that stores the tile IDs of the:

- goals
- walls
- cans
- robot position

[1] Animated maze solution: https://github.itu.dk/miim/AdvancedRobotics

```
XXXXXXXXX
X@      X   <--- Robot (tile ID = 11)
X X X X X
X  $    X   <--- Can on an intersection (tile ID = 31)
X X X X X
X       X
X X X X X
X  .    X   <--- Goal on an intersection (tile ID = 67)
XXXXXXXXX
```

Figure 6: Demonstration of our coordinate system.

We tried to represent the maze used for the competition with a different number of characters. Initially, a map size of 7x7 was used to represent the maze, where each intersection and each straight line corresponded to a tile. However, we simplified this down to a 4x4 map size by representing only the intersections as tiles as seen in figure 7. As explained in the next section, this improved the search algorithm's performance.

```
XXXXXXXXX                XXXXXX
X@      X                X@  .X
X X X X X                X $  X
X  $    X                X.$  X
X X X X X                X $ .X
X       X                XXXXXX
X X X X X
X  .    X
XXXXXXXXX
```

Figure 7: Map before and after simplification (left and right respectively).

3.1.2 Solver

We decided to use Breadth First Search (BFS) because it is one of the easier search algorithms to implement and it always finds the optimal solution if there is one. Before implementing it, we checked through a calculation that it will not require more memory or computational power to solve a 4x4 or even a 7x7 map than our PC can provide. In the case that the performance would be worse than predicted we were planning to replace BFS with A*. In our implementation of BFS each state is a list with the following format *[parentState, robotTile, can1, can2, can3]*, and holds the following information:

- reference to a parent state (root state has parent None)
- tile ID of the robot's position
- the positions of the cans in the map

This means that each *state* is associated with a *single* map's *tile*, but each *tile* can be associated to *multiple states* that differ in parents, cans positions or both. A parent state is the state in which the search was previously in, before moving into the current's state i.e. the state associated to the tile in which the robot was before moving to the current tile. We took advantage of the size of a reference in Python being only 8 bytes (assuming 64bits build of Python), and decided to pass a reference to the parent list. We explored various data structures for representing a state, but with the goal of minimising the space overhead, we settled on using a 1-dimensional list: this way -knowing that a reference is 8 bytes and an integer is 24 bytes- the total space complexity can be computed as follow:

- O(space) = 8 bytes + 24 bytes + (24 bytes * number of cans)

The algorithm runs until the goal state is found, with the end condition being that each of the cans is on a goal, or all the goals have a can: this would catch both situations where there are more cans than goals, or vice-versa. The generated result is a list containing the sequence of states from the start state to the final state.

3.1.3 Compiler

After the solver finds a sequence of Sokoban moves that solve the maze these moves are compiled into a list of more specific instructions that include the necessary changes in the orientation of the robot. Specifically, the generated list contains the degrees of rotation needed for the robot to turn when reaching an intersection in order to reach the next tile. A variable is used to keep track at all times of the current orientation of the robot, where:

- 0 = upwards
- 90 = right
- 180 = downwards
- 270 = left

By using the list of states to get from start to end (thus, being able to see for each step the current tile and the next tile) in combination with the current orientation of the robot, the list of degrees of correction needed can be easily generated.

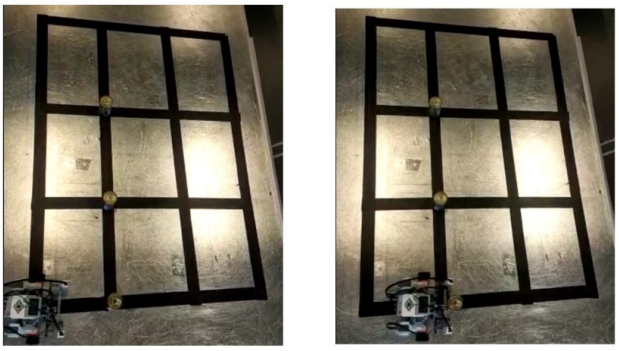
One issue that arose, however, was that the interaction between Sokoban and the in-game items does not translate to the real-world environment. When in Sokoban the character is on a tile next to a diamond, moving on the tile of the diamond causes the item to be pushed onto the next tile. This is shown in 3.1.3.

In our real-life maze, however, this is not the case: the can has to be pushed all the way to the next tile in order to be moved correctly. This is shown in 3.1.3. This means that there is a mismatch between the Sokoban actions and the real world. To fix this problem, we decided to implement a new push action:

- -1 = push

that would trigger a sequence of actions, making the robot push the can one tile ahead (so that the can is in the tile ahead of where the robot is, similarly to in Sokoban), reverse for 5 cm (to avoid hitting the

(a)



(b)

Figure 8: (a) Sobokan example: when the robot (@) is next to a can ($) (left) and moves one tile to the right (right), both elements move simultaneously forward.

Figure 9: (b) Real-world example: when the robot is next to a can and moves one tile to the right, the can remains on the same tile.

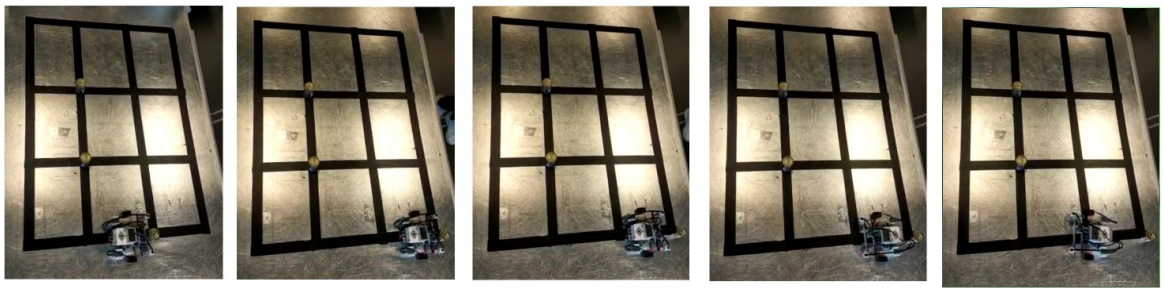


Figure 10: Sequence of moves to push a can forward and return back

can), perform ad 180° turn, and go back to the previous tile (i.e. previous intersection). This behavior is shown in the pictures in 10:

Checking whether a push action is needed is easily done by checking if a change in cans positions is detected when going from one state to the next. Another alternative that was discussed involved the use of a bumper detector sensor: the advantage of this approach is less complicated to implement within the robot controller; the disadvantages are an additional hardware component to test and calibrate, as well as introducing uncertainty on whether the can will hit the bumper detector sensor or not. Eventually, our team agreed that the trade-off of our chosen approach was worth, as it completely removes the uncertainty of not detecting the can.

The final list of corrections generated is the following:

- 0, 0, 0, -90, -1, 0, -1, -90, -90, -1, 90, -1, -90, 90, 90, -1, 0, -1

which means that at the first, second and third intersection (items at index 0, 1 and 2 in the list) the robot will keep driving straight; then it will turn -90° (to the left) at the next intersection, and perform a push operation at the next intersection; and so forth.

3.2 Robot Controller

Finally, the behavior of the robot was implemented so that the default action is to follow the line as explained in 2.4 until an intersection is detected using the lateral sensors. When this happens, the next action is fetched from the previously generated list of instructions.

3.3 Testing

Several experimental tests were run to assess various features and capabilities of the robot. The correct functioning of the robot holds under the following assumptions:

- well-lit room (natural sunlight or artificial lights)
- metal (or material with equivalent reflection properties) platform
- black, non-reflective tape for line detection
- smooth platform surface
- 140g, 6cm height per object (e.g. can) to be moved
- fully charged batteries
- can is placed in the middle of the intersection with a maximum error margin of +- 2.5cm

The success rate of the robot is 9/10 runs complete with all 3 cans in the correct positions, while 1/10 run only completes 1 can. This faulty trial was due to the robot not detecting an intersection after performing a 180-degree turn, and thus being unable to complete the run. The average speed of the 9 runs was 1 minute and 48 seconds. The complete results are in 2

Number of cans	Time	Cans
Run 1	1:20.13	3/3
Run 2	1:21.12	3/3
Run 3	1:23.38	3/3
Run 4	1:22.04	3/3
Run 5	1:22.22	3/3
Run 6	1:21.12	3/3
Run 7	1:23.16	3/3
Run 8	N.A.	1/3
Run 9	1:21.14	3/3
Run 10	1:22.34	3/3

Table 2: Tests with 1 can

Further tests were run with varying weights of cans: 2 cans (total of 680g) stacked both vertically and horizontally; 3 cans (total of 1020g) stacked both vertically and horizontally; and 4 cans stacked horizontally. The results are shown in Table 3

Although we acknowledge that performing and recording the results of more tests would have provided a stronger statement, we can nonetheless conclude from these results that the robot operates correctly until one of two conditions are broken:

Number of cans	Run 1	Run 2	Run 3
2 stacked vertically	1:23.64	1:23.92	1:23.29
2 stacked horizontally	1:23.12	1:24.31	1:23.75
3 stacked vertically	N.A.	N.A.	N.A.
3 stacked horizontally	1:25.21	1:24.89	1:23.12
4 stacked horizontally	N.A.	1:27.89	N.A.

Table 3: Tests with varying number of cans

- height is less than 3 vertically stacked cans (18 cm)
- weight is less than 4 cans (560 g)