Solving the Sokoban Puzzle using an Embodied Artificial Intelligence Mobile Platform

A semester project in the course of
Introduction to Artificial Intelligence

written by

Martin Androvich (Group 21)
marta16@student.sdu.dk

The code for this project is available at
`https://github.com/martinandrovich/aibot`

University of Southern Denmark (SDU)
Technical Faculty (Faculty of Engineering)

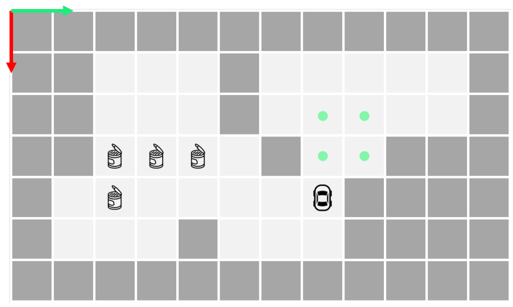
December, 2020

Contents

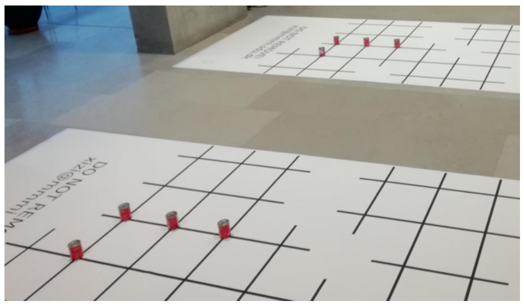
1 Introduction

1.1 Problem definition

Sokoban is a puzzle game in which an actor can push objects (in this case tomato cans) around within a predefined map, as shown in Figure 1a. The objective is to move all objects onto goal positions, where the puzzle is considered solved when each objects is uniquely placed on a goal position.



(a) Virtual Sokoban map with positions of wall, robot, cans and goals (green dots).



(b) Corresponding physical Sokoban map with tomato cans.

Figure 1: Representations of a Sokoban map.

In this project, a LEGO® MINDSTORMS® EV3-based mobile robot [1] is used to navigate a physical Sokoban map with real tomato cans, as shown in Figure 1b, where the traversable paths are represented as a grid with black line with each vacant field being an intersection. The objective is to: (a) solve the Sokoban puzzle by computing a sequence of actions (planned solution), and (b) design and build a mobile robot that can physically execute the solver's planned solution by moving the cans onto the goal positions on the physical map.

1.2 Approach

The overall problem is decomposed into two main components:

- **Designing and programming mobile robot** that can robustly navigate the physical Sokoban map with the ability to push cans; the robot must be interfaced using a sequence of actions.
- **Implementing a Sokoban solver** which can solve an arbitrary map with a maximum size of 50×50 tiles within a reasonable time span and produce a sequence of actions for the robot to execute.

The general approach is to work on the components in a highly decoupled manner and follow a minimalistic approach, such that, for example, the solver can be replaced with another, as long as it uses the same interface to the robot. Furthermore, both components must have evaluable metrics in order to analyze their performance.

2 Robot

The mobile robot is to be designed as an intelligent agent that is embodied into the physical world using behavior based robot control. For the robot to traverse the physical Sokoban map and push the cans to a desired position, three high-level functionalities are desired, namely: (1) an interface, (2) navigation capabilities, and (3) object relocation capabilities.

2.1 Requirements and behaviors

The three high-level functionalities are defined as:

- **Interface**
  The robot must be able to take an input sequence (string) and execute the actions parallel to a Sokoban game (up, down, left, right).
- **Navigation**
  The robot must be able to execute a desired action (e.g. up) by navigating the Sokoban map, i.e. moving between the grid intersections whilst following the lines.
- **Object relocation**
  The robot must be able to move around the cans whilst navigating the Sokoban map, allowing it to center a can on a grid intersection.

Both the navigation and object relocation capabilities are desired to attain a repeatability rate of $\geq 90\,\%$ and an accuracy of $\pm\,5\,\text{cm}$, thus providing a robust execution whilst also maximizing the execution speed.

The actual interface to the robot is considered as an input to the control system, alongside the perceived environment. That is, the robot only engages in behaviors relevant to the current input. The navigation and object relocation capabilities are decoupled using separate, behavior-based modules, each specialized in its functionality:

- **Follow line**
  Drive forward while following a black line.
- **Locate intersection**
  Detect if the robot is at intersection between the black lines.
- **Turn**
  Turn the robot around its center of mass (CoM) for a specified angle.
- **Push can**
  Push a can for one field to the center of an intersection and return robot to preceding intersection.

2.2 Mechanical design

Based on the desired behaviors for the mobile robot, there are several mechanical design options to consider, herein the wheels, sensors, can pushing mechanism and the general structure (chassis) of the mobile robot; the final design of the robot and with the selected components can be seen in Figure 2. The design philosophy is not to rely on constant offsets, but to instead apply a perception-driven approach, using the environment to navigate the physical Sokoban map.

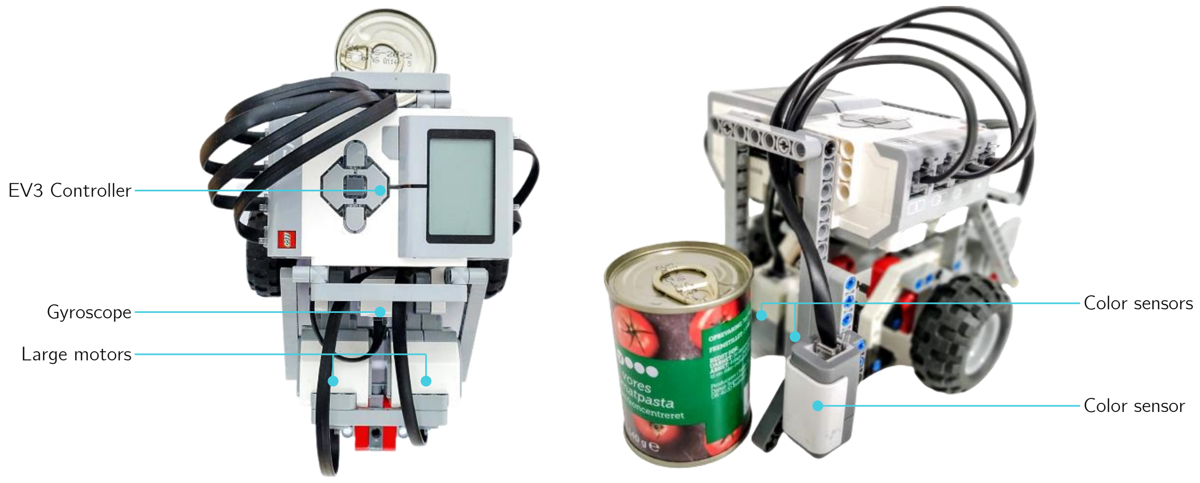


Figure 2: The EV3 mobile robot (aibot) and its components.

2.2.1 Chassis and wheels

The CoM of the robot and the transport mechanism (wheels or tracks) are significant to how well the robot is able to maneuver the physical map and at what speed it is able to do so. For the structure of the robot, the general approach is to fasten components in manner that increases the rigidness whilst preferring a low CoM positioned at the wheels in order to improve traction.

Generally, tracks provide more friction (slip resistance) at the cost of speed [2]. Assuming the wheels can provide enough traction to push the cans, they are preferred due to their capability for increased speed - however, slip may occur, which must somehow be accounted for; either in software or by limiting the maximum speed of the mobile robot.

2.2.2 Sensor selection

One way to facilitate movement between the grid intersections is using a line follower; that is, using the reflection intensity values of the color sensors to infer in which direction the robot should drive in order to stay on a line. Despite one light sensor being enough to follow a line, the general consensus is that an increase in the number of light sensors correlates to an increase in reaction time (with respect to staying on the line) [3, 4] and thus an increase in robust line following. The placement of the sensors is also crucial to the robot's reaction time, since placing them opposite to the driving direction typically introduces oscillatory behavior, as the sensors cannot "lead" the robot in such a configuration.

The intersections can be located using the color sensors; using a threshold to infer that a line is present, i.e., an intersection is detected when both color sensors observe an intensity value below some threshold, since higher reflection intensity values indicate a white or otherwise reflective surface. However, if the robot stops once an intersection is detected, it is necessary to offset the robot in order to align the center of the wheel axis with the center of the intersection, ensuring that the robot revolves around the center of an intersection when turning, easing the process of finding a line after a turn.

Turning is facilitated using the gyroscope, since it, despite potential drift, provides sufficiently consistent 90° turns within an allowable error margin, especially since the line follower will correct any turning errors, even in more extreme cases (offset slip). The gyroscope can also be used to realign the robot after following a line by measuring the gyro angle before and after a line-following section.

Pushing of tomato cans may be implemented using a line follower that drives for a constant distance, although this is susceptible to slip at higher speeds and is therefore unreliable. Instead, placing a single color sensor at the gripper allows to use the previously mentioned line-following approach, but instead stopping when the gripper color sensor detects a black line.

2.3 Implementation

The robot controller is implemented as the `aibot` Python module using a layer-based architecture, i.e., the desired functionality is implemented in layers that abstract the functionality, as shown in Table 1.

Layer	**File**	**Description**
Application	`app.py`	The main application, invoking any other components.
Navigation	`nav.py`	High level navigation methods and sequence parsing.
Hardware	`hw.py`	Low level hardware control.
	`constants.py`	Global constants for simplified access.

Table 1: The abstraction layers of the `aibot` Python module in hierarchical order.

The EV3 robot is interfaced using the `python-ev3dev2` library [5], allowing to communicate with and control the components of the robot, including the motors and sensors. The motors are interfaced using the `MoveTank` class [6], which is extended in the hardware abstraction layer (`hw.py`) with custom methods to provide the necessary low-level navigational capabilities (line following, turning, intersection detection).

For example, the intersection detection behavior is implemented using the boolean `at_intersection()` method, which extends the `MoveTank` class as shown in Listing 1.

```python
def at_intersection(self, th):
    return ((self.cs_l.reflected_light_intensity <= th) and (self.cs_r.
        reflected_light_intensity <= th))

MoveTank.at_intersection = at_intersection
```

Listing 1: Implementation of the `at_intersection()` method in the hardware abstraction layer (`hw.py`).

The `at_intersection()` method is used to periodically probe whether the robot is at an intersection when it is engaged in line following, providing a stimulus to the intersection detection behavior, which in response increments the number of observed intersections - once a desired number of intersections have been observed, the robot terminates the line following behavior.

The primary navigation methods are described in Table 2. These are implemented in the navigation abstraction layer (`nav.py`) and provide the necessary capabilities to traverse the physical Sokoban map.

Method	**Literals**	**Description**
`forward(n)`	`fn`	Drives `n` number of intersections in the current direction.
`push(n)`	`pn`	Pushes can for `n` number of intersections in the current direction.
`back()`	`b`	Drives in reverse until previous intersection.
`turn(dir)`	`l, r, u`	Turns in a direction specified by `dir`, being either `left` (`l`), `right` (`r`) or `around` (`u`).
`drive(seq)`		Performs the sequence of actions as specified by the `seq` string.

Table 2: The methods of the navigation abstraction layer (`nav.py`) and their corresponding literals.

The sequence of actions is defined by a string of literals (e.g. `f2lp3buf1r`) where each literal (one or two characters) maps to an action, as specified in Table 2 - e.g., `f3` maps to `forward(3)`. The `drive(seq)` methods iterates through the sequence string (`seq`), parses the literals and invokes the corresponding navigation method.

Movement of the robot is controlled by the hardware abstraction layer (`hw.py`). Rotational movement (turning) is implemented using the available `MoveTank.turn_degrees(speed, angle)` method. Translation movement mainly relies on a PID controller [4] which uses the two mounted color sensors the execution of the PID controller is governed by a generic `follow_for` callback method, i.e., the controller continues executing as longs as the callback method return true, allowing to use the PID controller whilst implementing various behaviors.

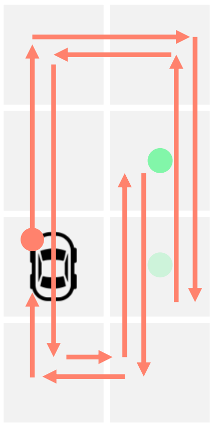
Piecing it all together, the navigation method `forward(n)` uses several methods from the hardware abstraction layer in order to enable forward motion of the mobile robot for `n` number of intersections. The line following behavior is initiated using the PID controlled method `follow_line_dual()` which is given the `follow_until_n_intersections()` method as a callback, providing pseudo-concurrency such that PID controller will try to follow a line until the callback methods returns false - this happens when `n` number of intersections have been observed.

The PID controller proportionally regulates the speed of the left and right wheels based on how much of a line is perceived by each sensor with respect to the reflectively levels. For example, if the right sensor observes a lower reflectively values (more black), the robot is veering off to the left and the left wheel is commanded a higher speed to correct the divergence from the line.

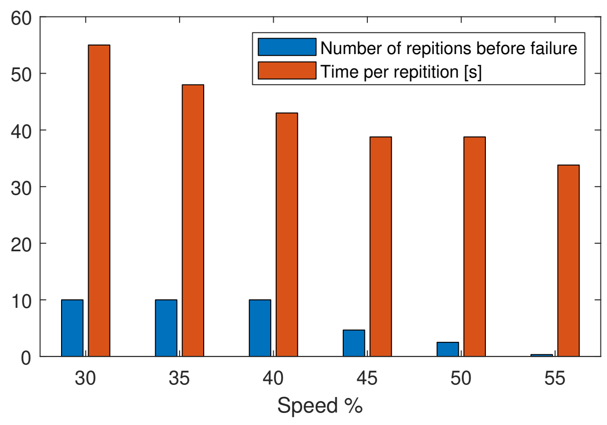
The speeds at which the robots operates are defined in `constants.py` with individual speed defined for different actions (forward, fast forward, push, turn etc.). All speeds are proportionally defined with respect to the forward speed; for example, the fast forward speed is defined an 20 % increase in the regular forward speed.

2.4 Evaluation

In order to evaluate the overall repeatability of the robot's navigation capabilities, a holistic experiment of driving forward, backward, turning and pushing a can was conducted, thus invoking all navigation methods, as demonstrated in this video [7]. In the experiment, the mobile robot is set to drive in a predefined path, pushing a can back and forth (one repetition), as shown in Figure 3a. The experiment is conducted for three sets of 10 repetitions for each speed percentage in the range of speeds of 30 % to 55 % with an increment of 5 % in speed between experiments. The experiment investigates the average number of repetitions before failure and the average time per repetition for a given speed - the results are shown in Figure 3b.



(a) The defined path of the can push experiment for a single repetition.



(b) Data from the can push experiment, showing the number of repetitions before failure and time per repetition.

Figure 3

The average number of repetitions before failure starts at 10 repetitions for a speed of 30 % and declines to 9.667 repetitions at 35 % speed, 9.333 repetitions at 40 % speed, 4.667 repetitions at 45 % and 2.5 repetitions at 50 % speed. The average time per repetitions improves as the speed increases, but at the cost of repeatability. A speed of 40 % allows to stay above the desired 90 % repeatability. The accuracy of can relocation has not been directly measured, but from observations of the experiments, it is seemingly adequate.

During the experiments, the decline of repeatability was seemingly caused by: (a) backing after a can push at higher speeds, or (b) that an intersection was not detected when driving at higher speeds. The speeds could be fine tuned to be lower at these critical operations and higher for when simply moving between intersections. However, a more optimal solution would be to: (a) implement the robot controller using micropython or C programming language, enabling a higher update rate in the control loop, and (b) implement a more fluent controller; that is, the transitions between, for example, driving forward and pushing could be implemented as a fluid motion, instead for first driving forward, stopping, and needing to accelerate again to push the can. The same goes for turning, where fluent corner cutting (turning around a wheel instead of the CoM) would not require the robot to completely halt before turning and thus save time.

3 Solver

The Sokoban puzzle solver program is implemented using the C++ programming language. To provide a solution it is necessary to define and implement: (a) the state representation for the system, and (b) an algorithm that determines a sequence of states leading to a solved puzzle. Thus, the solver program must output a string that encodes a sequence of actions necessary to solve the Sokoban puzzle; this sequence must be able to be parsed by the mobile robot, allowing it to realize the actions in the physical world.

3.1 Design

The solver program is designed after the object-oriented programming (OOP) design paradigm, i.e., the components of the Sokoban puzzle and solver are treated as objects. The structure of the solver is shown in Figure 4.

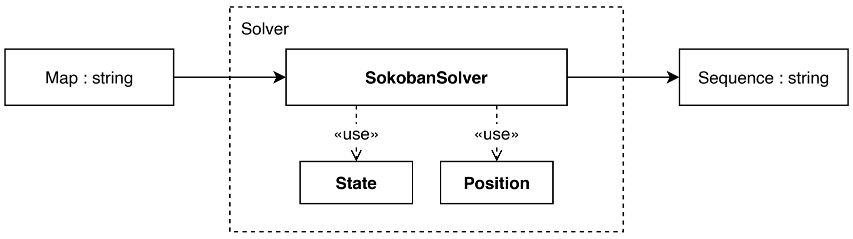


Figure 4: Block diagram overview of the Sokoban solver program.

The `SokobanSolver` class takes in a Map as a string and outputs a sequence as a string, which is to be parsed by the mobile robot. The solver class relies on the `State` class for representing the state and the `Position` class for representing a position on the Sokoban map.

3.1.1 State definition

In the Sokoban game, a state can be uniquely represented by the positions of the cans and the position of the player for a given map - the origin of a map is defined at its top-left corner with the x-axis being positive downwards and the y-axis being positive rightwards, as shown in Figure 1a. The state information is encapsulated by the `State` class as shown by the UML diagram [8] in Figure 5.

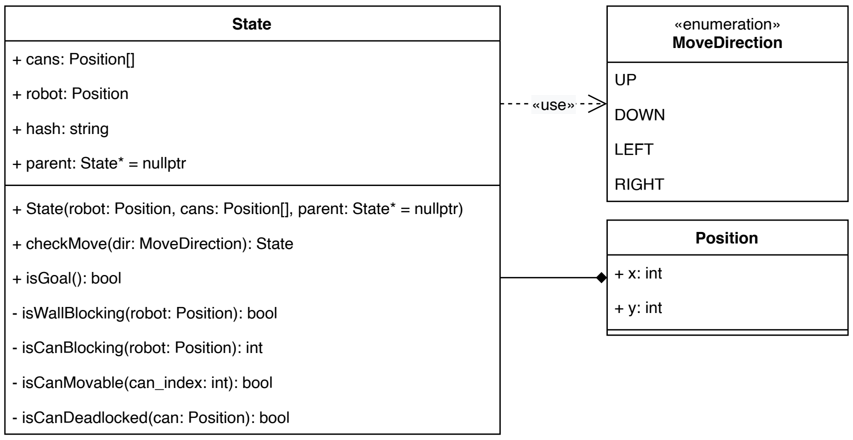

State
+ cans: Position[]
+ robot: Position
+ hash: string
+ parent: State* = nullptr
+ State(robot: Position, cans: Position[], parent: State* = nullptr)
+ checkMove(dir: MoveDirection): State
+ isGoal(): bool
- isWallBlocking(robot: Position): bool
- isCanBlocking(robot: Position): int
- isCanMovable(can_index: int): bool
- isCanDeadlocked(can: Position): bool
«use»
«enumeration»
MoveDirection
UP
DOWN
LEFT
RIGHT
Position
+ x: int
+ y: int

Figure 5: UML class diagram for the `State` class.

The `State` class contains additional information to aid the solver program, including: (a) several static member variables (initialized externally by the `SokobanSolver`), herein the map, a clean version of the map (only walls), the goal positions and any illegal can positions, (b) a string for the hash of the given state, and (c) a pointer to the parent state, allowing the `SokobanSolver` to construct a linked list of states. The `State` class also leverages several methods which enable the `SokobanSolver` to infer information about the current state and future states, such as whether a can is movable in a given direction - these methods are described in Table 3.

Method	**Description**
`isGoal()`	Returns whether all cans are uniquely placed onto a goal.
`isWallBlocking(robot)`	Returns whether a wall is blocking for the desired robot position.
`isCanMovable(can)`	Returns whether a specific can is movable with respect to robots the current position.
`isCanDeadlocked(can_pos)`	Returns whether a specified can position is deadlocked.
`checkMove(dir)`	Returns a new `State` object if the robot can move in the desired direction (up, down, left or right).

Table 3: Methods of the `State` class.

3.1.2 Algorithm

Given an initial state, all possible states can be represented using a tree data structure. Assuming a reasonable map size and enough computing power (primarily memory), the solution is deterministically determinable using breadth-first search (BFS) algorithm [9], which ensures a solutions with fewest number of moves; however, this does not necessarily correspond to the most optimal solution with respect to the robot's physical movement. The `SokobanSolver` class dynamically constructs a tree of possible states and uses the BFS algorithm to exhaust the possible states until a solution is found, as shown in Figure 6.

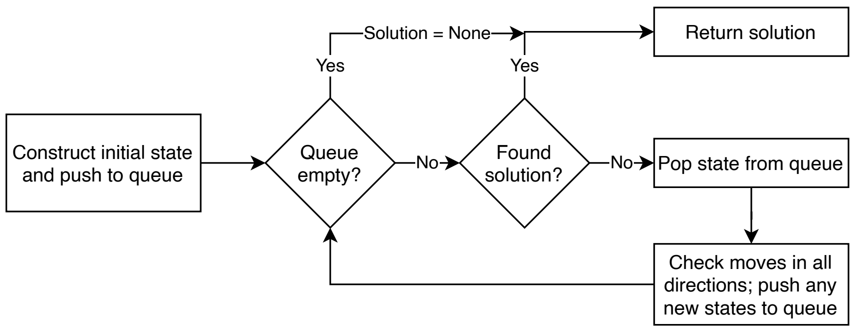

Solution = None
Yes
Yes
Return solution
Construct initial state and push to queue
Queue empty?
No
Found solution?
No
Pop state from queue
Check moves in all directions; push any new states to queue

Figure 6: Flow diagram for the algorithm of the Sokoban solver program.

The solver first parses the input map, extracting the positions of the player, cans, walls and goals; these are used to construct the initial state, which is pushed onto a queue of states. For each state in the queue, the `checkMove(dir)` method is used to check whether a move is possible in the desired direction; the method returns a new state (with the current as its parent), which is then pushed onto the queue.

The `checkMove(dir)` method of a state is what allows to determine whether the robot can move in a desired position and how this affects the state - the flow of the method is shown in Figure 7. However, it is inefficient to check all possible states, since many of them might either be duplicates or deadlocked.

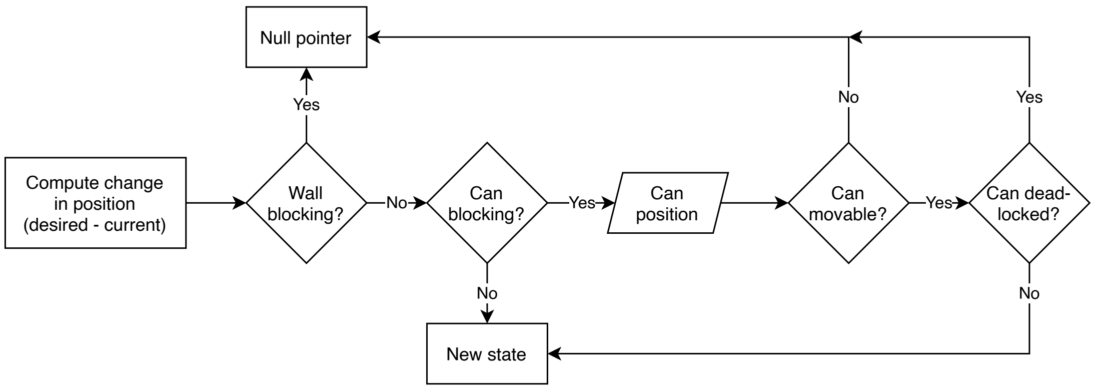


Figure 7: Flow diagram for the `checkMove(dir)` method of the `State` class.

To avoid checking duplicate states, a unique hash can be computed for a given state as the concatenation of the robot position and position of all the cans. This hash is then used in a lookup table (LUT) for visited states; if the `checkMove(dir)` method returns a State whose hash is already present in the LUT, the state is not pushed onto the queue of states.

A deadlocked state is a state from which the puzzle can no longer be solved; for example, if a can is pushed up against a wall in a way that it can no longer be moved free of the wall and thus never able to reach the goal position. Eliminating these reduces the total number of states to examine, which in turn speeds up the solver.

3.1.3 Sequence encoding

Once a solution has been found, a sequence of actions can be readily computed by iterating in reverse from the final state, since each state has a pointer to its parent. Actions where a can is moved are made uppercase; the can movement is inferred by comparing the list of cans between the parent and child states. An example sequence string in the player domain is shown in Figure 8a.

`llUrUUddL`

(a) Player domain.

`lf2rp1brf1lp2buf2rp1b`

(b) Robot domain.

Figure 8: Sequence of actions encoded in strings using literals in the different domains.

For the robot to execute a sequence of actions they must be converted to the robot domain. The player domain commands absolute actions whereas the robot domain actions are commanded relative to the robots current orientation and must be compatible with the navigation methods of the robot controller; it is assumed that the robot in its initial state is always facing north. An example sequence string in the robot domain (corresponding to that of the player domain) is shown in Figure 8b.

3.2 Evaluation

The solver is evaluated by attempting to solve various Sokoban maps from an online test suite [10]. However, these levels must be converted from the canonical map format to the format used in this project. For each map, the time taken to find a solution is logged; if the solver fails to find a solution before memory runs out, the map is logged with a solution time of -1, i.e. unsolved. A total of 57 maps were evaluated with the number of cans varying from 2 to 12 cans - the results are shown in Figure 9.

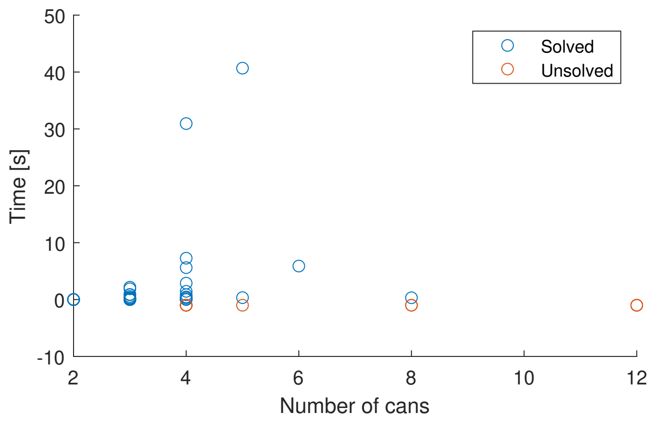


Figure 9: Scatter plot of the Sokoban solver test suite evaluation.

The solver is theoretically "complete", i.e., if a solution exists, it will be found, but the solver is limited by the available memory, since the spatial requirements are directly proportional to map size and number of cans per map. As previously mentioned, due to the nature of the BFS, the solver will always find the optimal solution with respect to the number of moves, but this is not a heuristic that translates well into the physical robot domain.

From the experiment it is evident that the number of cans is not the only factor when it comes to the time taken to solve a map. The map size also affects the complexity of the solver; therefore, a more in depth experiment is needed to determine how the temporal complexity depends on both the number of cans and map size.

References

[1] The LEGO Group. *LEGO® MINDSTORMS® Education EV3 Core Set.* URL: https://education.lego.com/en-us/products/lego-mindstorms-education-ev3-core-set/5003400.

[2] Hornback, Paul. “The Wheel Versus Track Dilemma”. In: 1998. URL: https://fas.org/man/dod-101/sys/land/docs/2wheels98.pdf.

[3] Chowdhury, Nakib Hayat, Khushi, Deloara, and Rashid, Md. Mamunur. “Algorithm for Line Follower Robots to Follow Critical Paths with Minimum Number of Sensors”. In: *International Journal of Computer (IJC)* (2017), pp. 13–22. URL: https://ijcjournal.org/index.php/InternationalJournalOfComputer/article/view/819.

[4] Jim Sluka. *A PID Controller For Lego Mindstorms Robots.* URL: http://www.inpharmix.com/jps/PID_Controller_For_Lego_Mindstorms_Robots.html.

[5] Ralph Hempel et al. *Python language bindings for ev3dev.* URL: https://github.com/ev3dev/ev3dev-lang-python.

[6] Ralph Hempel et al. *MoveTank class of the ev3dev2 Python library.* URL: https://ev3dev-lang.readthedocs.io/projects/python-ev3dev/en/stable/motors.html#move-tank.

[7] Martin Androvich. *YouTube video: Evaluation of EV3 mobile robot for Sokoban puzzle.* URL: https://youtu.be/IFERzhfyKqQ.

[8] Object Management Group. *The Unified Modeling Language.* URL: https://www.uml-diagrams.org/.

[9] *Breadth-first search.* URL: https://en.wikipedia.org/wiki/Breadth-first_search.

[10] *Sokoban levels.* URL: http://sokobano.de/en/levels.php.