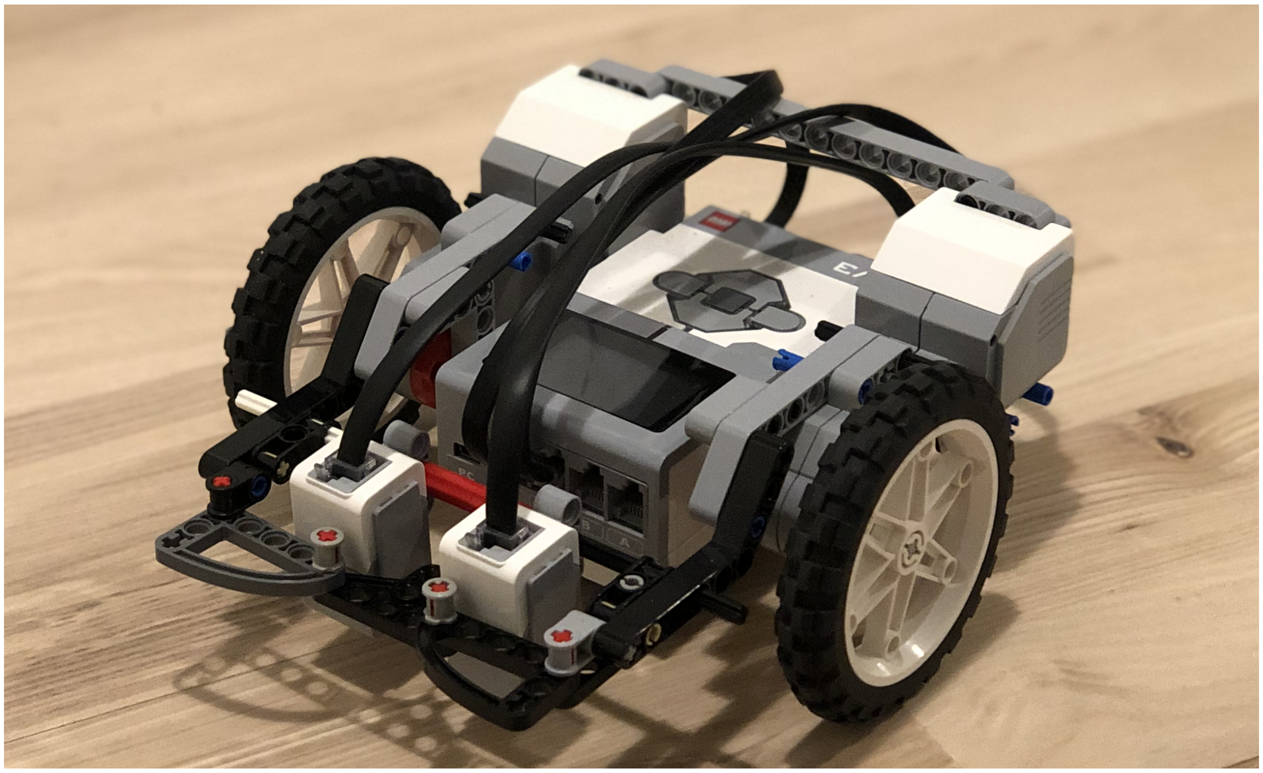
University of Southern Denmark

Faculty of Engineering

# Sokoban x LEGO Mindstorms EV3

## Introduction to Artificial Intelligence



**Sina P. Soltani** siesm17@student.sdu.dk 454200

**Group 38**

**Jens O. H. Iversen** jeive17@student.sdu.dk 455605

11th December 2020

## List of Figures

# Contents

## 1 Introduction

Sokoban stems from the Japanese word for ”warehouseman”. The game from which it is named, models a warehouse worker pushing boxes from some initial positions to their respective intended and allotted positions. The different warehouses or Sokoban maps can be of different complexity, depending on the number of free spaces and boxes, that need to be moved. Additionally, warehouse work is physically demanding and the need for automation is therefore an interesting topic of robotics. For this project a LEGO Mindstorms EV3 and a white poster with black lines (figure 1) representing the Sokoban map are supplied.

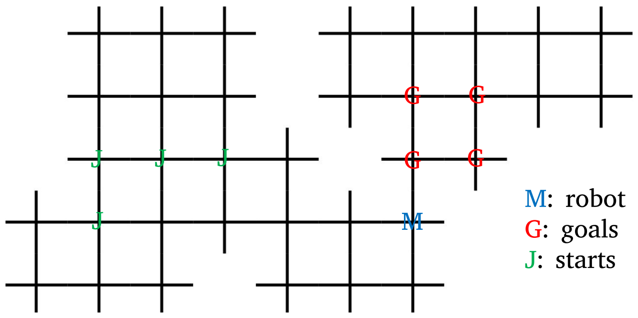


Figure 1: An example of a Sokoban map.

In this paper we ask two questions;

1. How can algorithms and techniques within the field of Artificial Intelligence be employed to solve complex Sokoban Maps?
2. How should the hardware and firmware of a warehouse robot be designed to reliably solve the Sokoban maps as quickly as possible?

We try to answer these question with a report on how simple techniques from embodied artificial intelligence can be utilised to complete a Sokoban map with the use of a LEGO Mindstorms EV3 robot.

### Software and Hardware

The programs used to complete the objective of the project are written in Python and are run on a LEGO Mindstorms EV3.

# 2 Robot

Enabling the robot to navigate and solve Sokoban maps, it will need a physical structure and firmware. The physical structure is needed to be sufficient in pushing the boxes around the map and the firmware is needed to execute the movements of the robot In this section, we begin by explaining the design choices for the necessary hardware. Subsequently, we expand upon the firmware and the implemented methods.

## 2.1 Hardware & Physical structure

To make it possible for the robot to solve a physical Sokoban map, a few things are necessary. The robot needs a set of wheels, color sensors, and an arm to push the boxes around the course. The robot has been equipped with two main wheels that are on either side of the robot. It is also equipped with a rear-wheel that is used for balance. Since the physical Sokoban maps are constructed using lines, the robot has been equipped with two color sensors that is used as a line follower. The two sensors are placed such that the line can be in between both sensors with a small margin. This enables the robot to detect if it is deviating from the course and thereby taking an action to prevent it. An image of the robot with the aforementioned hardware can be seen in figure 2.

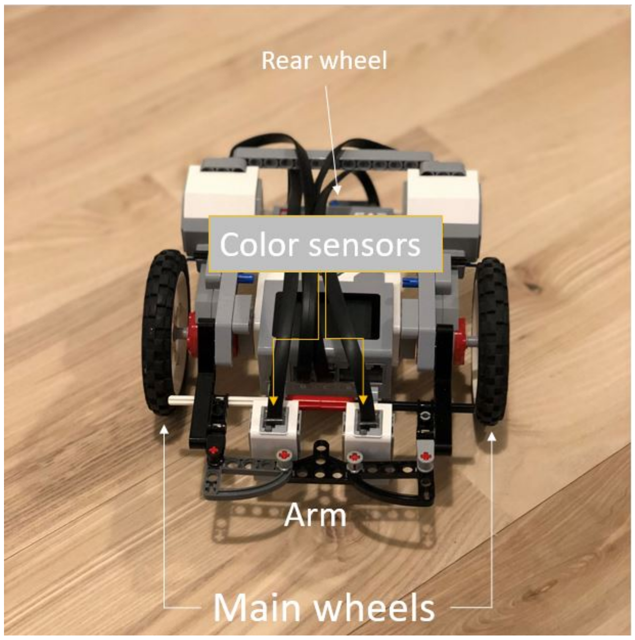


Figure 2: The robot with its hardware marked.

The robot was constructed to have a low center of mass, which resulted in the two main wheels being wider apart. The wheels were picked to have the largest diameter possible with the idea that if the motors could keep the same RPM with a bigger diameter, it would cover more distance. Therefore the wheels with the largest diameter were picked. The diameter is 8 cm. The two main wheels ended up being 15.3 cm apart. The arm was constructed so it is wide enough to catch the boxes, even if they were misplaced. It was designed so the box would always be pushed from the middle of the robot. This would lead to the can being place consistently every time and make it easier to push again.

### 2.2 Firmware

The program running on the EV3 robot, which executes the solution was designed to be as simple as possible. The program consists of a few key features with a couple of upgrades that reduce the time expenditure of the robot as it is completing the course.

The principal components of the program are deciphering the solution, turning and moving forwards and backwards while staying on the path. To increase the robustness of the robot, and make it less vulnerable to external disturbances, the first process is a calibration step. As the robot will always begin positioned on an intersection of two lines, by moving forwards and back it will gather information about the light intensity with its colour sensors. This information can be used to dynamically set a threshold that is used to distinguish between white and black parts on the map.

With the threshold set, the robot starts reading the content of the solution. The solution has its syntax, which can be seen in figure 3. The robot sets its orientation based on the letters, counts the number of equivalent consecutive moves and then moves forward appropriately. The robot uses its color sensors to track its moves, by counting the number of perpendicular lines it passes, whenever both sensors report reading a black spot.

Whenever the current letter is capital and the succeeding letter in the sequence is different from the current one, the robot knows that it is about to finish pushing a can. It, therefore, moves one spot forward to deposit the can and returns.

To remain on the path between two spots on the map, a line follower was implemented. The design of the line follower resembles that of a very simple Breitenberg vehicle - 2b, emulating aggression [1]. When the robot deviates from the line to the right, the left colour sensor will read

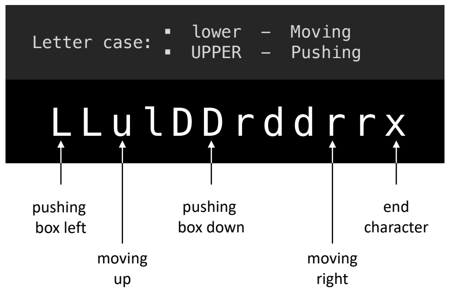


Figure 3: The format of a solution for a Sokoban map.

the black line. This signals the right motor to increase in speed temporarily, which gets the robot back on track. The symmetric example holds as well for when the robot deviates to the left.

These are all the necessary functionalities for the robot to move around, push and deposit cans on the map.

## 2.3 Results

The final robot has been tested on physical representations of map 1 and map 2 which can be seen in the appendix. Four parameters were tweaked to find the fasted time while still being robust. The first parameter is cruise speed which is how fast the robot drives while moving forward or backward. The second parameter is turn speed which is how fast the robot turns. The third is consecutive moves which means that the robot will not stop in between moves if it has to move forward more than one space. The last parameter is calibration which means if calibration is done before the run. The test for the two maps can be seen in table 1 and 2.

| Cruise speed | Turn speed | Consecutive moves | Calibration | Time [min:sec] |
|---|---|---|---|---|
| 200 | 250 | No | No | 5:25 |
| 200 | **500** | No | No | 5:01 |
| **300** | 500 | No | No | 3:43 |
| 300 | 500 | **Yes** | No | 3:43 |
| **400** | 500 | Yes | **Yes** | 3:04 |
| **450** | 500 | Yes | Yes | N/F |

Table 1: Test on physical resprensentation of map 2

| Cruise speed | Turn speed | Consecutive moves | Calibration | Time [min:sec] |
|---|---|---|---|---|
| 400 | 500 | Yes | Yes | 2:15 |
| **450** | 500 | Yes | Yes | 2:05 |
| 500 | 600 | Yes | Yes | N/F |

Table 2: Test on physical resprensentation of map 1

## 2.4 Evaluation

From the two tables (1 & 2) it can be seen that a cruise speed of 400 and 450 for map 2 and map 1 respectively is the maximum. It can also be seen that a turn speed of 500 is reliable. A higher turn speed has also been tested, but the robot failed each time, therefore was the turn speed not investigated further. It can be seen from table 1 that consecutive moves do not have an impact on the completion time. Calibration has not been tested independently, but it was not implemented for the sake of a faster time, but rather for reliability and robustness.

Map 1 was used in a competition where 21 teams competed. The robot designed in this report came in second place with a time of 2:05 minutes.

## 2.5 Conclusion

It is concluded that a maximum cruise speed of 450 and a maximum turn speed of 500 can complete a physical representation of a Sokoban map. It is also concluded that the robot is well designed and implemented since it came in second place with a time of 2:05 on map 1 with a total of 21 contestant participating.

# 3 Solver

The solver is the brain of the system. It reads and analyses the map and comes up with a solution, upon which the robot body executes a path. A couple of different algorithms have been implemented to attain a solution to the Sokoban map; Breadth-First-Search (BFS), Depth-First-Search (DFS), A* and as a bonus Q-learning has also been implemented. To more easily follow the solution and entertain the miserable life of the programmer, the mechanics of the Sokoban game were implemented and visualised using Pygame. An example of this can be seen in figure 4.

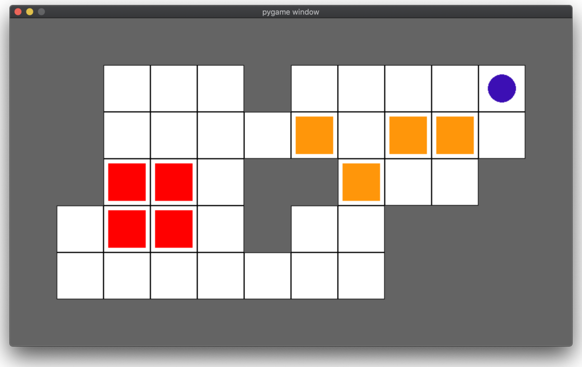


Figure 4: The simulation environment to visualise the solutions for a Sokoban map.

## 3.1 Implementation

The implemented algorithms all follow the same template, as can be seen in figure 5. The crucial difference between the algorithms is their respective data structures. As the BFS is a level order tree traversal, i.e. the children of a node are visited when all siblings of the same node have been visited, the data structure for this method can be represented by a FIFO queue. The DFS on the other hand visits the child of a node before the siblings, and can therefore be represented by a LIFO queue. Finally, the nodes of the A* algorithm are associated with a cost, on which their position in the queue is dependent; therefore it can be represented by a priority queue.

A queue holding the nodes are implemented, and the only discrepancy between the algorithms is the way they add nodes to the queue. BFS adds the nodes by appending the queue, DFS adds nodes by prepending the queue and A* adds nodes by inserting the nodes to the correct position depending on their associated cost.

```
FORWARD_SEARCH
$Q.Insert(x_I)$ and mark $x_I$ as visited
while $Q$ not empty do
    $x \leftarrow Q.GetFirst()$
    if $x \in X_G$
        return SUCCESS
    forall $u \in U(x)$
        $x' \leftarrow f(x, u)$
        if $x'$ not visited
            Mark $x'$ as visited
            $Q.Insert(x')$
        else
            Resolve duplicate $x'$
return FAILURE
```

Figure 5: A general template for the implemented algorithms [2]

## 3.2 State Representation

As most modern computers can visit the entire search tree in practically no time, the focus of this project is to reduce the space complexity of the task. As the entire state of the positions of the boxes and the agent on the map is needed, alternatives to storing the map more efficiently - than a text file of characters - are explored. This section covers the thoughts and considerations that went to developing a compressed state representation.

In its most basic form, the state representation must encode the positions of the agent, boxes and goals. Doing so without thought would require storage of the x and y pairs for each element. For four boxes, this state representation would require 18 integers; two for each box, two for each goal and two for the agent position. However, this representation does not account for the position of the walls, thus a separate environment must be stored to use as a lookup when implementing the game mechanics.

To remove the need for the separate environment, the 2D positions can be indexed, so that all free spaces correspond to singular value. This exchange between 2D and 1D positions is encoded in a hash-table, allowing for constant lookup time between the two formats. This method further simplifies the implementation of the game mechanics, since the checking of a valid position is done by a lookup in the hash-table; if a value is returned it is valid, otherwise it is not.

Another helpful idea that reduces the state representation is the realisation that the boxes are identical and encoding their respective positions is superfluous. Instead, it is the combination of the spaces they occupy that is relevant, with the order being irrelevant and repetition unallowed, as only one box can occupy a given space at any time. The total number of combinations, c, for the number of boxes, b, on a map with the number of free spaces, s, is calculated as follows:

$$c = \binom{s}{b} = \frac{s!}{b! \cdot (s-b)!}$$

On a map with four boxes and 35 free spaces this would result to:

$$c_{35,4} = \binom{35}{4} = \frac{35!}{4! \cdot (35-4)!} = 52360$$

unique combinations. Here another hash-table is created that maps the sorted positions of the boxes into single values in the range [1, c]. The state of the positions of the boxes is now represented by a single integer. To represent the entire state of the positions of the boxes and the agent, we simply add the state of the agent as the fractional part to the state of the boxes (See figure 6).

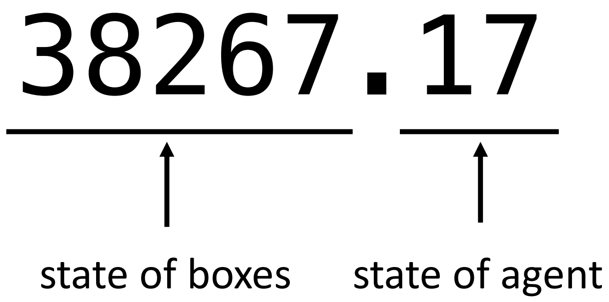


Figure 6: The single float state representation of the Sokoban game.

The entire state is now represented by a single float! If we are only interested in the position of the agent, we use the `modulo` operator, multiply by 100 and convert it to a 2D position. If we are interested only in the positions of the boxes, we use the `floor` operator on the state and convert that to a list of positions. Additionally, since the goal positions can be encoded similarly, testing whether all the boxes are on a goal position, is simply done by comparing their respective states.

## 3.3 Improvements

To avoid cyclic paths and quickly disregard states, where a solution is unattainable, some improvements have been added. This in turn reduces the time expenditure of reaching a solution. These improvements are expanded upon in this section.

### Dead States

The first and most effective improvement is the removal of *dead* states. There is spent a lot of time exploring branches in the search tree where the state has reached a dead-end from which returning is impossible. These dead-end states include boxes trapped in a non-goal corner, boxes beside a wall by which a goal is not located and more. The simplest dead-end states to register is when the boxes are positioned up against a non-goal corner. The method is written that detects the corners and marks these states as dead-ends. Whenever a state like this is reached in the search tree no further nodes are generated from this branch. This heavily reduces the number of nodes generated and equivalently reduces the time it takes to reach a solution.

### Already Visited & To Be Visited

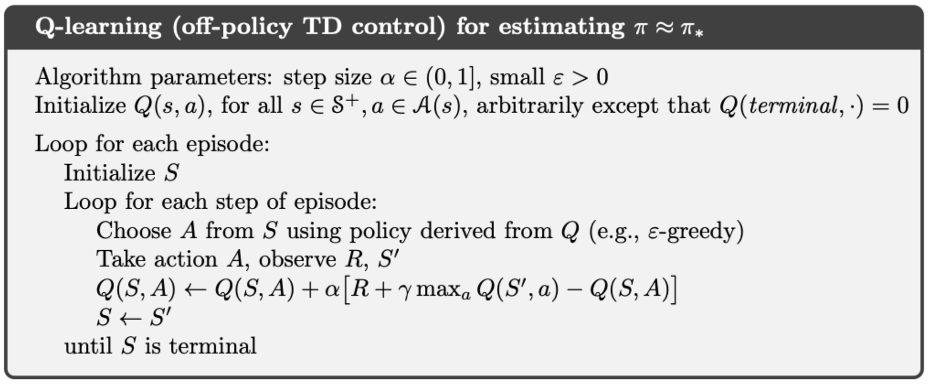
A further improvement can be made by not generating nodes with states that have already been visited. If a state is marked as visited, it signifies that a node closer to the root has already explored this state. As we are looking for the shortest path, there is no reason to visit the exact same state further down the tree. The same principle can be made for states that are already queued to be visited. In a BFS this would occur when a sibling has already found a path to the same state, therefore there is no reason that other nodes on the same level add paths to the same state. This improvement might skew the search tree towards the left - as the nodes are searched from left to right - but this artefact should have no effect on the performance of the algorithm.

### Heuristics - A*

A final improvement that has been implemented concerns the A* algorithm. As the A* algorithm requires a heuristic that underestimates the distance to the target, a cost function is implemented. The cost function adds the distances of the boxes to the goal with the depth of the node in question. This cost function should prioritise states where the boxes are closer to the goals, where the agent has taken the fewest steps.

## 3.4 Bonus: Q-Learning - Sina

As an extra aside, Q-learning was implemented to test whether this approach would be able to solve the Sokoban game. Q-learning works by storing a table of values representing the expected



**Q-learning (off-policy TD control) for estimating $\pi \approx \pi_*$**

```
Algorithm parameters: step size $\alpha \in (0,1]$, small $\varepsilon > 0$
Initialize $Q(s,a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(\mathit{terminal}, \cdot) = 0$

Loop for each episode:
   Initialize $S$
   Loop for each step of episode:
      Choose $A$ from $S$ using policy derived from $Q$ (e.g., $\varepsilon$-greedy)
      Take action $A$, observe $R$, $S'$
      $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma \max_a Q(S',a) - Q(S,A)\big]$
      $S \leftarrow S'$
   until $S$ is terminal
```

Figure 7: The Q-learning algorithm as prescribed by Sutton and Barto [3]

values when taking some action, A, at a given state, S. The algorithm tries to approximates the optimal state-action function, q*, so that the expected total sum of rewards is as great as possible. By continuously updating the values in the Q-table based on the rewards that are returned when taking actions around the map, the algorithm arrives at the optimal policy. The pseudocode for the Q-learning algorithm can be seen in figure 7.

The real challenge of using Q-learning for a task like the Sokoban game is the extreme size of the state space, which quickly explodes because of the curse of dimensionality. To combat this, a very compact state representation, as described in 3.2, is utilised. Using this state representation the total size of the state space becomes:

$$|Q| = \binom{s}{b} \cdot s \cdot |A|$$

Here the entire Q-table is determined by the state of the boxes on the map, the state of the agent on the map and the number of actions that are available to the agent.

To control the learning and greediness of the agent, two hyper-parameters $\alpha$ and $\epsilon$ are set to control the learning rate and exploration chance respectively.

The results of the implemented algorithm show that the agent does in fact learn the optimal policy for completing the game, however it takes many episodes and much longer than the classical algorithms. On the other hand, the Q-learning algorithm arrives at a policy that is able to complete the Sokoban map regardless of the initial position of the boxes or the agent, whereas the search algorithms only solve that specific setup they are running on. Thus even though Q-learning takes much longer to arrive at a solution, it solves a much bigger task - namely every possible starting configuration on the Sokoban map.

## 3.5 Results

The search algorithms BFS, DFS, and A* have been tested on ten different maps. Each map was tested four different times for each algorithm, where it had one, two, three, or four box locations. This was conducted to test the effectiveness of each algorithm performing with a different amount of boxes. Five parameters were noted for each test: number of moves for the solution, number of nodes visited, number of nodes generated that were not visited[1], time for execution, and memory used. The parameters were averaged over all ten maps for each algorithm and can be seen in figure 8, 9, 10 and 11. All the ten maps can be seen in Appendix under the section Maps.

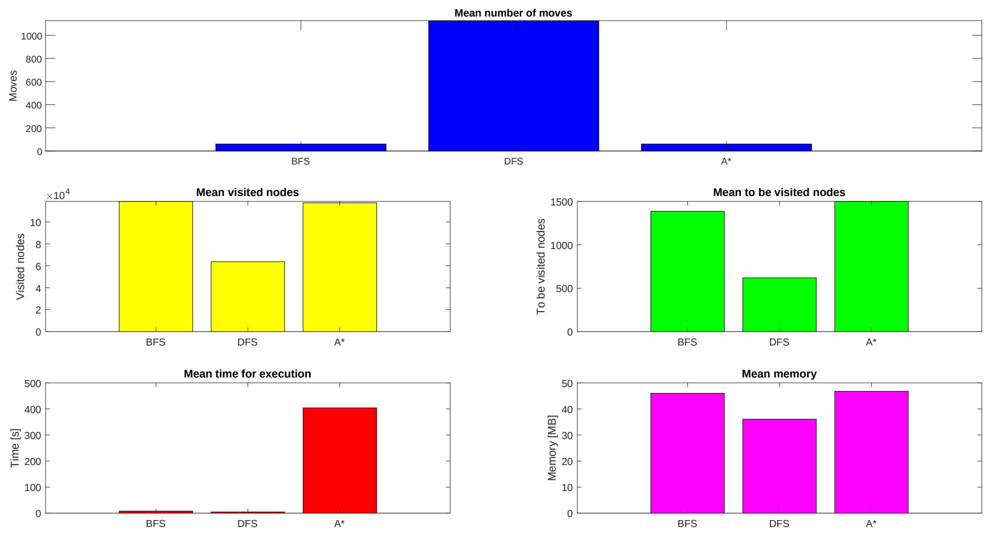


Figure 8: Mean of the five parameters for all ten maps with four jewels on each map.

[1]Also called "to be visited nodes".

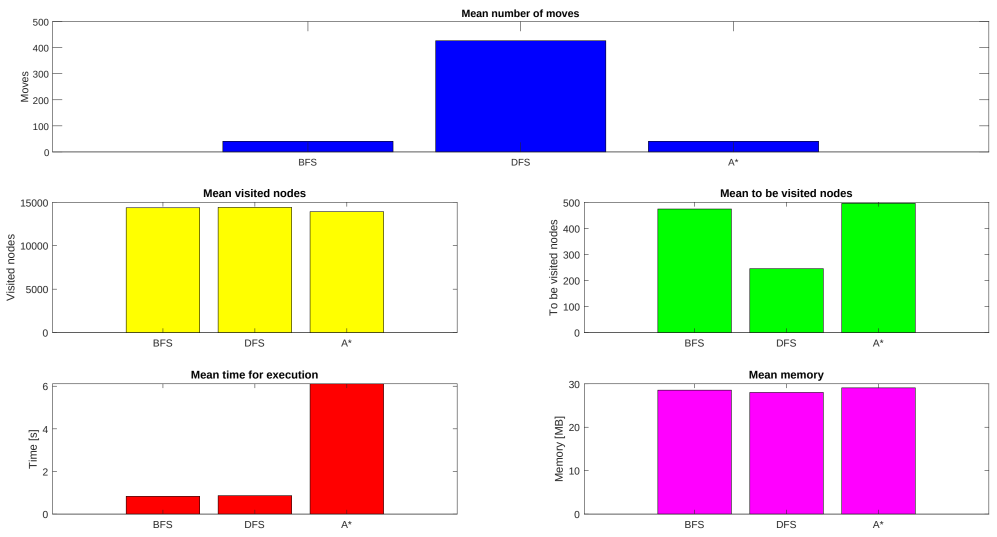


Figure 9: Mean of the five parameters for all ten maps with three jewels on each map.

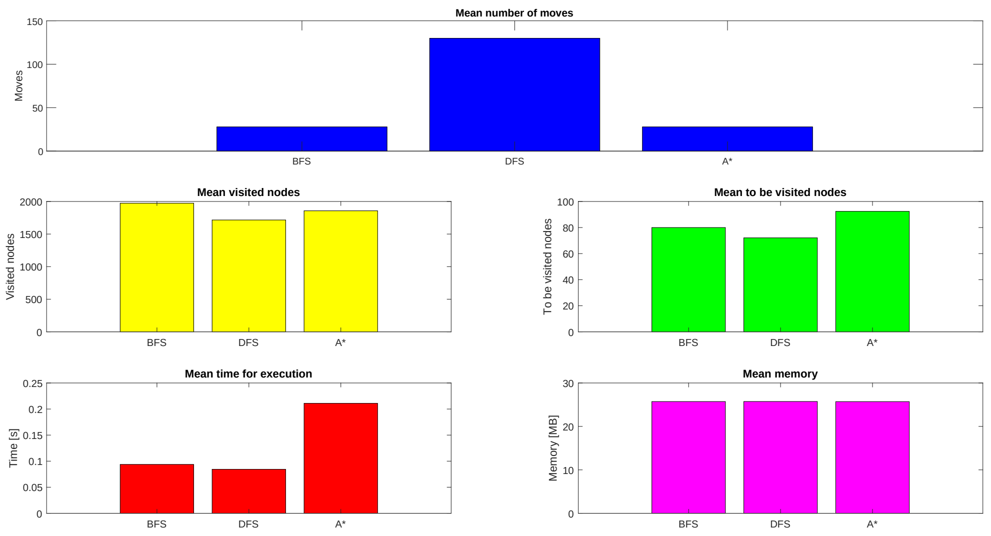


Figure 10: Mean of the five parameters for all ten maps with two jewels on each map.

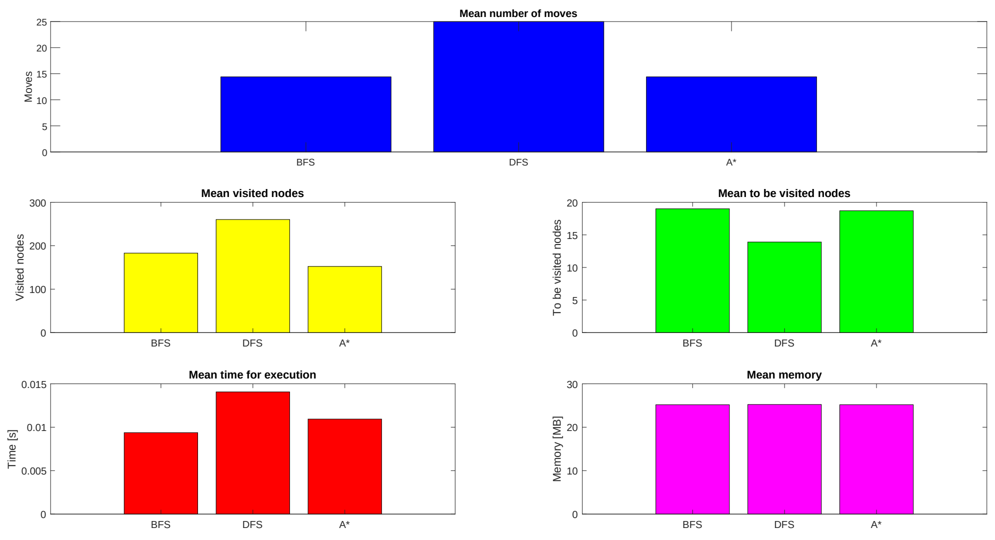


Figure 11: Mean of the five parameters for all ten maps with one jewels on each map.

## 3.6 Evaluation

From all four figures (8, 9, 10 & 11) it can be seen that the average length of a solution for DFS is a lot longer than that for BFS or A*. BFS and A* has the exact same length solution for each map[2]. This means that A* finds the shortest solution each time, but it also has an increased execution time compared to BFS. Therefore is the A* algorithm not superior to BFS which it is intended to be. It can also be seen that with more boxes on the map DFS generates fewer nodes than the other algorithms and its execution time is less. This is not useful at all since its solution is significantly higher. A* visit on average fewer nodes than BFS independent of the number of boxes. This should mean that A* is faster than BFS in theory, but since A* has to sort its queue for each child generated, this could result in a slower execution time. The memory uses for all algorithms is not vastly different for each test and is fairly low.

## 3.7 Conclusion

It can be concluded that the BFS search algorithm is the fastest algorithm to find the shortest solution. It can also be concluded that memory usage of all three search algorithms is not a problem

[2]This can not be seen in the figure, but was obvious from the data collected.

at all and it seems like it could handle even larger maps. Since all three algorithms use a small amount of memory it can be concluded that the state representation is well chosen. It can also be concluded that the complexity of four boxes placed on a map makes the A* algorithms extremely unstable.

## References

[1] *AI Lecture: Braitenberg Vehicles.* URL: `http://users.sussex.ac.uk/~christ/crs/kr-ist/lecx1a.html` (visited on 12/17/2020).

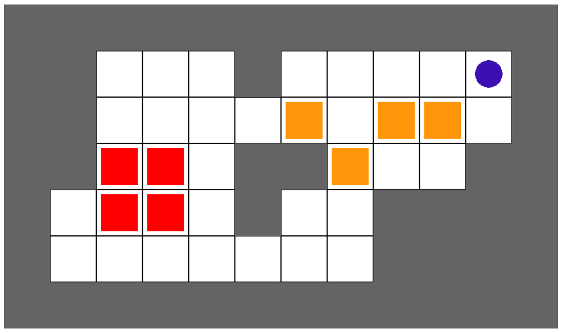
[2] Steven M. LaValle. *PLANNING ALGORITHMS.* Cambridge University Press. URL: `http://lavalle.pl/planning/`.

[3] Richard S. Sutton and Andrew G. Barto. *Reinforcement learning: an introduction.* Second edition. Adaptive computation and machine learning series. Cambridge, Massachusetts: The MIT Press, 2018. 526 pp. ISBN: 978-0-262-03924-6.

# Appendix

## Maps

Red squares are the goal position for the jewels. Yellow squares are the jewels. Blue circle is the player/agent.


(a) Map 1

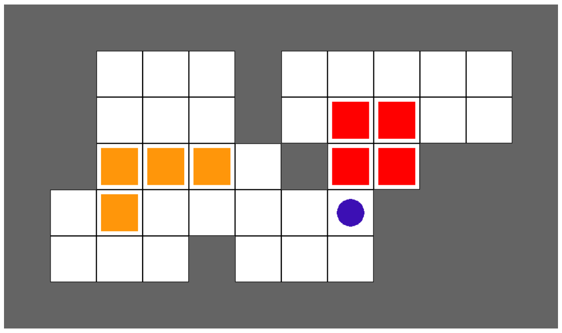
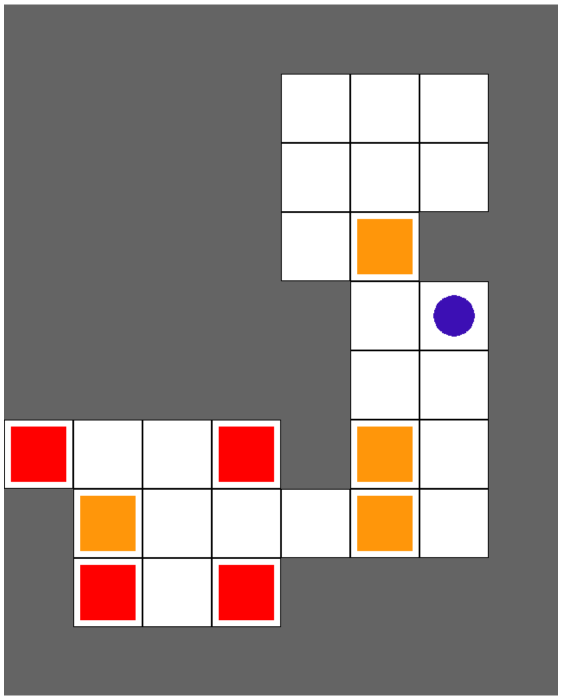

(b) Map 2

Figure 12: Map 1 and 2


(a) Map 3

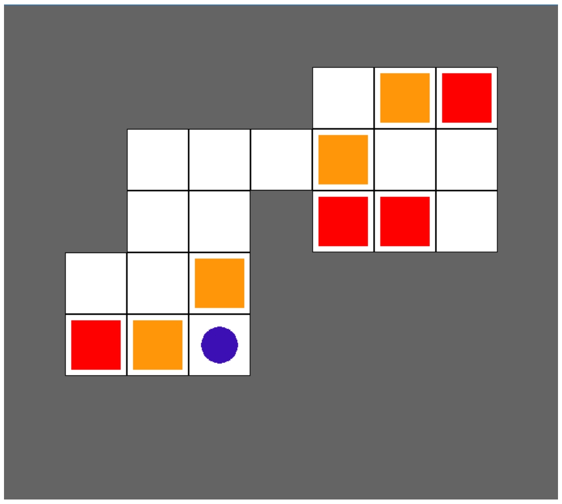
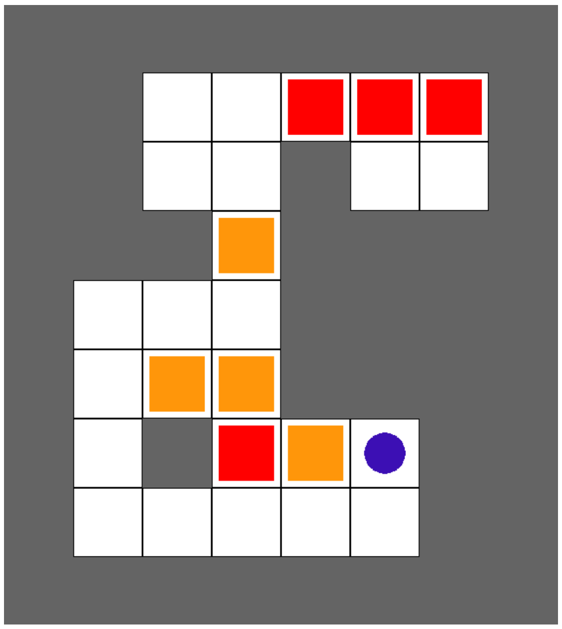

(b) Map 4

Figure 13: Map 3 and 4


(a) Map 5

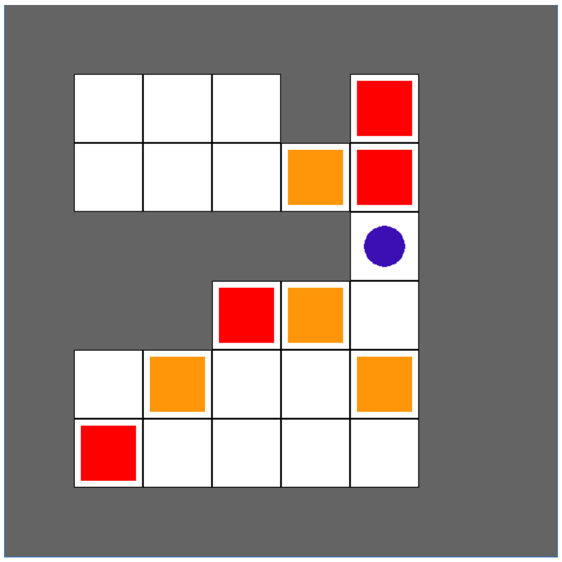
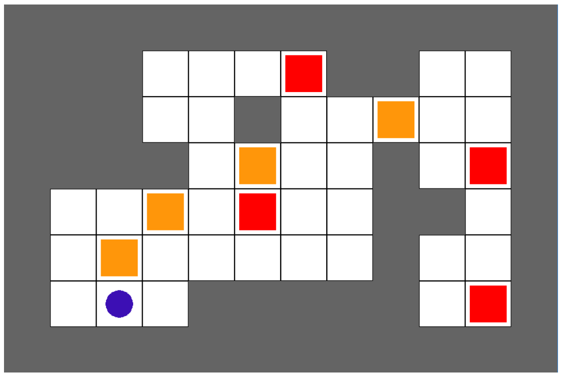

(b) Map 6

Figure 14: Map 5 and 6


(a) Map 7

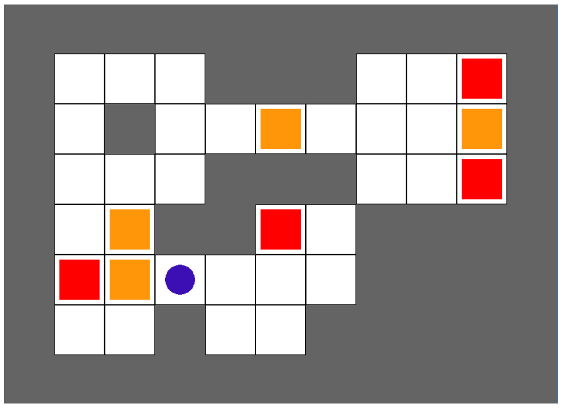
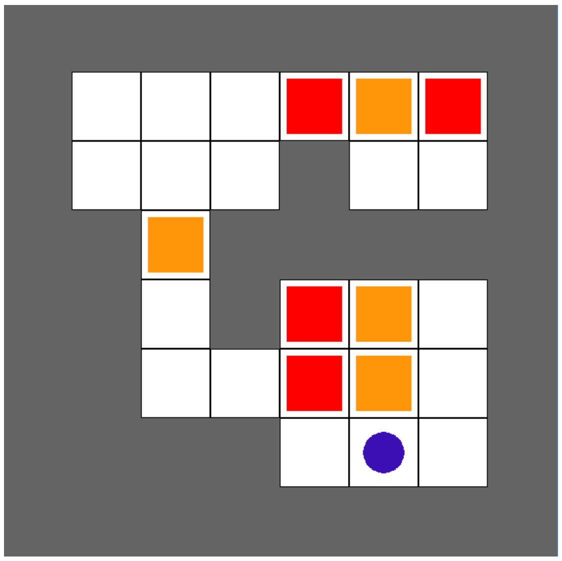

(b) Map 8

Figure 15: Map 7 and 8


(a) Map 9

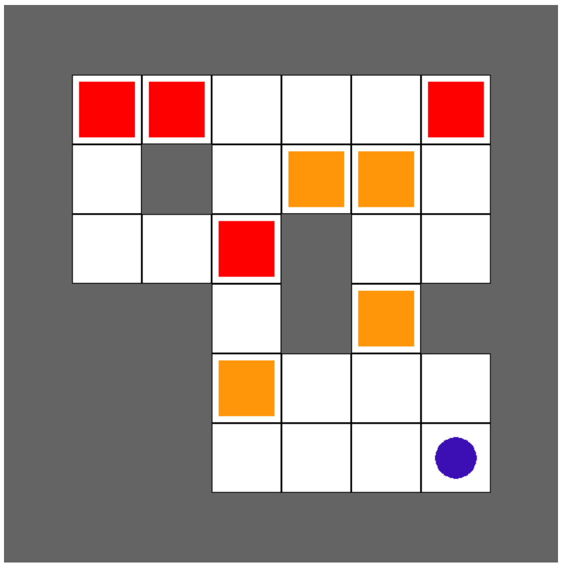

(b) Map 10

Figure 16: Map 9 and 10

## Github

`https://github.com/SinaPourSoltani/EV3_Sokoban.git`