The University of Southern Denmark

The Faculty of Engineering

Introduction to Artificial Intelligence - T550000101

Final report - Group 9

Figure 1: SDU logo from [1]

Oliver Holck Vea
olvea16@student.sdu.dk

Simon Lyck Bjært Sørensen
simso16@student.sdu.dk

Exercise period: 10-2019 - 18-12-2019

Contents

1 Robot

The robot has been designed around the behaviours necessary to solve a Sokoban map. To be able to do this, the necessary behaviours has to be identified.

1.1 Behavioural design

For the robot to solve a Sokoban map, the required behaviors are:

- Driving along a line forwards and backwards.
- Turning at intersections.
- Pushing cans to a temporary or final destination.

Driving along lines and turning at intersections allows the robot to traverse the entire map according to the movement rules of the Sokoban game. The only other required behaviour necessary is to be able to push cans from one intersection to the next.

1.2 Physical Implications

For the first behaviour, driving along a straight line, the gyroscopic and color sensors are used. The color sensors are used to detect if the robot is off course. One sensor on each side of the line the robot is driving along. This requires that the sensors are placed pointing down on the ground surface, centered with at least a lines width between them. This will make sure that if the robot crosses the line, the sensor corresponding to the direction in which it is going off course will detect a line. The color sensors will also be used to detect when the robot is in an intersection and thereby how far it has come on the map. The gyroscopic sensor is needed so the robot is not dependent on the color sensors while it is on course. With the gyroscopic sensor, the robot can keep the course by maintaining a constant angle instead of just getting information about the course from the color sensor. With continuous information about the course, the robot will be able to drive faster than if it only had information about the course from the color sensors where the robot only will get the information if it crosses a line. In addition to sensors, the robot will have to be equipped with a method of locomotion. Two independent motors equipped with wheels are used for this purpose. The reason for selecting wheels is twofold. Firstly, the radius of the wheels is slightly larger than the tracks meaning that the top speed of the wheels will be a bit higher assuming equal angular velocity. The second reason for choosing wheels is that the turning midpoint is less prone to error as it shouldn't depend on traction like the tracks do.

For the second behaviour, turning, the color and gyroscopic sensors are used. The color sensor is needed to detect when the robot is aligned with the line it has to follow after the turn. The gyroscopic sensor is required to make the turn faster, as the robot can use the angle to turn faster while it is not yet close to the desired angle for the turn. Utilizing the gyroscopic sensor should also result in a more precise alignment.

For the third behaviour, the color sensor, gyroscopic sensor and ultrasonic sensor are used. As the third

behavior just pushes a can, it is quite similar to the first behaviour. There are two major differences between the behaviors. Firstly, the behaviour has to stop before reaching the intersection, as the can is a bit in front of the robot. Secondly, the behavior drives a bit slower to not push the can over on impact. A claw has also been included on the robot for this purpose. The two segments of the claw are angled so they push the can towards the center of the robot.

The behaviours imposes the following requirements on the physical design:

- Two color sensors spaced with at least a lines width.
- A gyroscopic sensor.
- An ultrasonic sensor.
- A method of locomotion, i.e. motors equipped with wheels.
- A claw for maintaining control of a can.

With these physical requirements, the robot was made. It can be seen in figure 2. (color sensors shown in green, gyroscopic sensor in red, ultrasonic sensor in yellow, motors equipped with wheels in blue and claw in purple)

1.3 Programming Implications

One of the fundamental requirements of every behaviour is to be able to detect the lines using the color sensor. For this purpose a software module has been written.

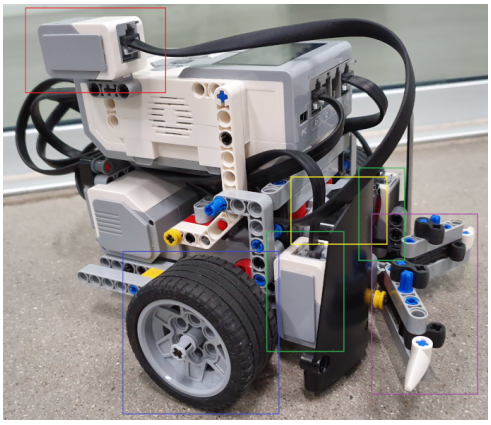


Figure 2: The initial robot design.

To initialize the sensors, they were set in "Reflected light" mode and calibrated to the white color of the mat. Whenever a line is detected, the sensor readings will decrease. A way of detecting lines could be outlier detection by detecting line readings in an array otherwise populated by white readings. The first method that was attempted is called CUSUM. CUSUM is an algorithm used to detect changes in sequential data [2]. The method worked on data extracted from the robot, but when run on the robot, the computational complexity of the method rendered the it infeasible.

The second method was taking the mean and standard deviation of an array of samples and determining everything below 5 standard deviations below the mean as a line sample. This method still required too much computational time for the program to be able to detect lines while driving at a reasonable speed and was therefore discarded.

The last method just takes the sum of an array of samples. If a new sample is 40% less than the average, it is determined to be a line sample. By messing around with the math, this method exclusively uses sums and multiplications to detect lines. This makes the method fast enough for real-time application

and still keeps the threshold for line detection dynamic so it can handle different lighting conditions.

The function returns if a line is detected on the left or right color sensor and if a horizontal line is detected and if so, if it's the first time that horizontal line is detected.

The first and third behaviour requires the robot to be able to move straight. This is implemented in the robot as a fuzzy controller (see figure 3). The reason this is done is primarily as a tool to combine the distance and target angle while being able to smooth out the velocity curves by tweaking the membership functions and rulebase.

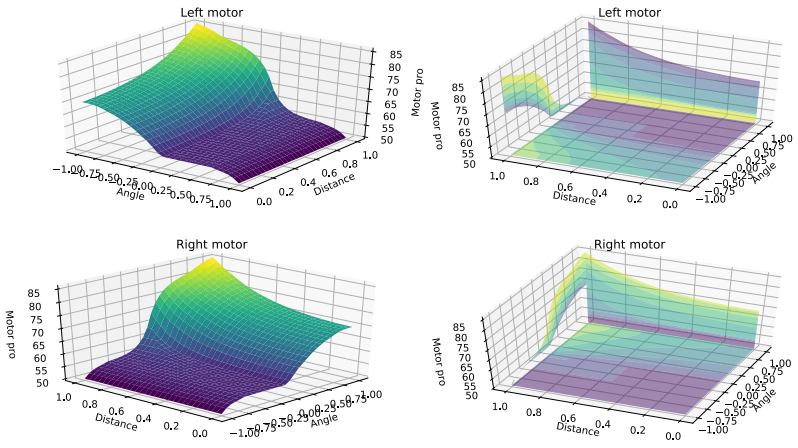


Figure 3: The surface of the fuzzy controller.

The fuzzy controller was programmed in python using the sk-fuzzy module [3]. It was sampled into a lookup array and saved in a file. The robot can use this file to look up PWM values for the motors with a constant time complexity, which means that the robot does not have to calculate defuzzyfication in real-time. The first input to the fuzzy controller is the angle error between the course and the measured angle from the gyroscopic sensor. The course is changed if a line is detected with the line sensor, ensuring that both the gyroscopic sensor and the line sensor are used to set the course of the robot. The second input to the fuzzy controller is the remaining travel distance. This allows the robot to slow down before the target which is especially useful if it is about to collide with a can. This distance is based on either an internal estimate of the remaining distance based on time since the last horizontal line was observed or the ultrasonic sensor if the robot is expected to be facing a can. Before being inputted in the controller, the values are normalized so they could be changed during testing to change the severity of the angle and distance values. Ideally, the fuzzy controller surface would be resampled with the final edge values so the robot wouldn't have to normalize the values as this requires division.

At a point during the development of the robots code the gyroscopic sensor stopped working. Whenever a sample was requested, the operating system of the robot would crash. This, along with realising there is no need for the ultrasonic sensor, results in the robot design seen in figure 4

This had a couple of implications for the implementation of the behaviours:

As the line detection does not use the gyroscopic sensor, it was not subject to change because of the change in sensors.

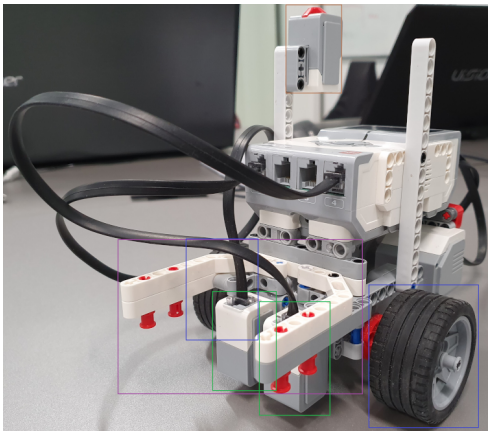


Figure 4: The final robot design. (color sensors shown in green, motors equipped with wheels in blue, claw in purple and touch sensor in brown)

The first behaviour used the gyro sensor as a main part of its calculation. The implementation was therefore changed to only use the color sensor in the form of the result from the line detection. The forward line following was done by changing the speed of the motors to 190% on the motor on the side going away from the line and reducing the speed of the motor on the other side 90%. While reversing the change is reduced to 130%, while the reduction is kept the same. This ensures that the robot will correct itself to drive along the line without the use of the gyroscopic sensor. But due to the non-continuous nature of the line detection, it results in a more jittery movement, and if the robot crosses the line the robot can not recover.

The second behaviour was implemented as a state machine with the following states in chronological order. The video at [4] shows the states with the robot. The specific part of the video there shows the stats are "Turn detailed".

1. pre_turn("Align Wheels" in the video): In this state, the wheels are aligned with the intersection, negating the offset from the space between the wheels and the color sensors. This is done be driving the robot forward in 400 ms. This is done so the robot will turn around the center of the intersection.
2. start_turn("Both Sensors Free" in the video): In this state the robot turns with a lower speed than driving forward so it is more controlled. This state continues until the sensor to the side the robot is turning to, is free and the other is on the line.
3. mid_turn("One Sensor on new line" in the video): This state turns the robot with the same speed. This state continues until the sensor to the side the robot is turning to is on the next line. This means the sensor will have detected the target line.
4. end_turn("One Sensor on new line" in the video): This state still turns the robot with the same speed. This state continues until the sensor to the side the robot is turning to is free again and thereby on the other side of the target line.
5. post_turn ("Post Align" in the video): In this state the robot will turn for 150 ms to correct the robot to be aligned with the target line. After this, the robot has turned either right or left by 90 degrees.

The timings for the offsets are found via. trial and error. The state machine is used to turn right and left. To be able to perform a U-turn, the robot repeats the states starting at start_turn after it is done with the first end_turn state.

The third behaviour was not implemented due to the robots speed causing was reduction the robot will no longer push the cans over on impact.

To choose between behaviours, an action selector was made. It uses a plan in the form of a list of moves needed to complete the task. The format of the commands are [Move, Move argument] where the allowed moves are: Forward (F), Backwards (B), Left (L), Right (R) and U-Turn (T). The F and B moves have arguments which specify the number of interceptions the robot should pass before stopping. The robot will not register horizontal lines before 0.5 s after the movement has started. This is done to make sure the line the move is started on doesn't count as a passed intersection.

1.4 Evaluation

To test the performance of the robot *nigh* test was done.

To test how well the robot can recover from being off course on a line, the robot was put in (40°, 35°, 30°, 25°, 20°, 15°, 10° and 5°) angles off the line 10 times each. Then the robot had to correct its course using the line following behaviour. From 5° to 30° the robot corrected its curse 10 out of 10 times. Doing 35° it could correct 9 out of 10 times while at 40° it could correct 2 out of 10 times. (The test setup can be seen in [4] "Angle Test (35 degrees)")

To test the robots ability to move forward, backward, left, right and do a U-turn separately, a test where the robot did the movement 50 times in a row was attempted. It was given at most 5 trials to see how many sequential moves it could do. The robot could drive forward, turn left and make u-turns 50 times in succession in the first attempt. It took 2 attempts for turning right and the robot was only able to do at most 4 backward moves in succession with a mean of 2.6 moves for the 5 trials. (The test setup is shown in [4] "Forward", "Reversing", "Right Turn", "Left Turn" and "U-Turn")

To test if the robot can handle a longer sequence of a combinations of different moves, a test where the robot had to drive in eights was conducted. Here, the robot could do 10 eights in sequence where the robot had to do 1 U-turn, 4 left and 4 right turns and 9 forward movements resulting in 180 sequential moves. Driving backwards was not included in the test due to the poor performance of the isolated tests. (The test setup is show in [4] "Driving in Eights test (Lighting Condition 1)")

The previous tests was performed under the same lighting condition (indoor lighting). To test whether the robot could perform as well in daylight, the robot was set to drive in eights again during the day. (The test setup is show in [4] "Driving in Eights test (Lighting Condition 2)")

The last test performed was to make the robot drive from one end of the competition map to the other. This was done to test a different sequence of moves and a sequence that might be more like what the robot would preform while solving the map. The robot was able to drive the full path from one end of the map to the other. (The test setup is show in [4] "End to End test")

1.5 Conclusions

The angle test shows that the robot can recover well as long as it is not off by more than 35°. At angles above 35°, it will most likely not recover. The test where the moves were tested separately shows that the robot has a stable performance for driving forward, turning left, right and making u-turns independently. The robot does not have good performance for driving backward. The two drive in eights shows that the robot also has good performance when doing a combination of the different moves and can do it in different lighting conditions as there was no noticeable change in how well the robot performed after the change in lighting. The end to end test shows that the robot is able to perform a sequence similar to what it would encounter while solving a Sokoban map.

The code necessary to push cans was not implemented. This was due to time limitations, mainly due to having to reimplement the behaviours after the gyroscopic sensor broke. Therefore there are not included any run on the competition map with cans. The robot was not include in the competition.

2 Solver

To solve the competition sokoban map, a sokoban solver was developed. This solver had a couple of main goals in mind.

- The solver has to find a solution approximating the optional solution based on real world time.
- The solver should be able to solve the map within a reasonable timeframe.
- The solver should be able to solve the map withing reasonable memory limitations.

These criteria are quite loose and will be specified in the evaluation chapter in an effort to measure the performance based on these criteria. The goals are used to guide the design of the algorithm.

2.1 Design

One of the most important parts of the solver is defining the states the solver is discriminating between. The design of the state is subject to different requirements. Firstly it is essential to know the position of the dynamic parts of the map: the robot and the cans. These are represented as index values with the structure elements *playerPos* and *boxPositions* in the code as seen in figure 5.

For the solver to be able to estimate real world execution time, each behaviour on the robot has been timed. Each step in any solution is multiplied by the duration that step should take based on those measurements. The values for forward movement, 90-degree turn and 180-degree turn is 1.31 s, 1.1 s and 1.84 s respectively. Using this time as the criteria of selection between states in the solver, it can approximate temporal optimality, depending on how good of an approximate the estimated solution time is. The estimate for time spent is stored in the states as *currentExpense*, as seen in figure 5.

To be able to estimate the time used it is necessary to know when the robot has to execute the different behaviours, i.e. when the robot has to turn, move forward, etc. This means that an important piece of information to contain in the states as well is the robot direction. This will allow the algorithm to

discriminate between movements based on the amount of turns that has to be done, even though the total amount of steps, and therefore forward movements, are the same. The parameter containing robot direction is called *playerDir* and can be seen in figure 5.

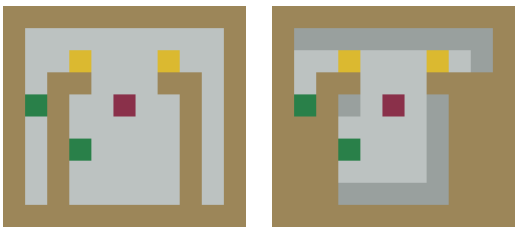
Positions in the solver are all 1-dimensional. This reduces the amount of numbers needed for every position by half. One of the problems with this approach is that it makes positions harder to understand because the map width is needed to calculate the x- and y-coordinate (x = i % width, y = i / width).

The states can be hashed into strings with the function *getHash*. This returns the robot position, direction and all the box positions, all separated by commas. (e.g. "17,-1,18,27,29") This means that the states can be remade from the hash values, but they will be missing the heuristic information. The function footprint can be seen in figure 5.

```
struct State {
    pos_t playerPos = 0; // Player position
    dir_t playerDir = 1; // Player direction
    std::vector<pos_t> boxPositions = {}; // The position of boxes.

    heu_t currentExpense = 0, futureExpense = 0; // Estimate of current and sfuture expense.

    const heu_t getHeuristic() const; // Returns currentExpense + futureExpense.

    const hash_t getHash() const; // Returns a hashed version of the state. Note this hash lacks heuristic information.
    static State fromHash(hash_t val); // Makes a state from a hashed shate. Note that the state will lack heuristic information.

    static bool comparison(const State& a, const State& b); // Comparison between states.

    const bool operator==(const State& s) const; // Comparison between states.
    const bool containsBox(pos_t position) const; // Query if a box is at a given position.

    const State move(pos_t direction) const; // Returns the state if the current state was to move in the direction.
};
```

Figure 5: State definition in C++.

To reduce the complexity of the map, two pre-processing steps are done. Firstly, 'tunnels' (series of connected 1-tile wide free spaces) are made into walls if they fulfill two criteria: they don't contain a target for a can and they are not connected to wider area in both ends. The second preproccesing step is marking free tiles that would get cans stuck next to walls without being able to reach a target. These are marked with the enum value *noBox* in the map and are not considered as potential box locations. These changes reduce the tree depth and branching factor respectively. The pre-processing can be seen in figure 6.



(a) Test map before pre-processing.

(b) Test map after pre-processing.

Figure 6: Comparison between a test map before (a) and after (b) pre-processing. Dark gray tiles are *noBox* tiles.

Another important piece of the algorithm is which 'moves' are allowed - i.e. how to generate children

from a parent state. The valid choices in the solver are the cardinal directions. This is because since a 180-degree turn is timed, there is no reason for the robot to be able to turn twice in a row; moves will always either be just going forward, or a combination of a turn and then a forward movement. By imposing this restriction on the generation of states, the computational time and memory use of the algorithm is reduced, because there is more difference between states. This means that less states will have to be evaluated.

To improve the computational time, the algorithm is multi-threaded. The threads gets the state with the lowest expense (estimated time used) from the open set and generates the possible children, depending on which moves are allowed in the given state. The open set is implemented as a priority queue sorted after lowest expense. When one thread is done, the rest are allowed to finish calculating the current state and the completed state with the lowest expense is selected. From this state, the line of parent states is followed back to the initial state, back-tracing the solution. This back-tracing is done in the closed set, as it contains the smallest expense and parent state of each visited state.

2.2 Evaluation

To evaluate the solver, the criteria from the beginning of the chapter are revisited.

The first criteria requires the algorithm to find solutions that approximate real world optimality. This is done by using temporal measurements of the robots behaviour to estimate the total time used for each state. The estimate was tested by making the robot drive in the shape of an 8 and timing each maneuver both with a smartphone stopwatch and internally in the robot. This was repeated 10 times and the time for each was noted. The theoretical time was calculated by performing each behavior in the maneuver by the time each behavior was measured to take. When the theoretical time is compared to the average stopwatch measurement, there is an error of 8.4 s. When compared to the internal measurement, however, the error decreases to 0.033 s. This could be explained by the internal measurement not measuring real time properly, as the theoretical time is based on internal time measurements as well. This test demonstrates that the theoretical time estimates of the plan comes quite close to the actual plan time when using the internal time from the robot. The results of the test is summarized in table 1.

Table 1: Timing test results and comparison to theoretical time.

	Time (stopwatch) [s]	**Time (robot) [s]**
Average	30,841	22,463
Error (ideal)	8,411	0,033

The algorithms performance on the other criteria were evaluated together. The second criteria requires the algorithm to converge on a solution within a 'reasonable' time frame. This does not mean that computational time is the most important factor, just that the algorithm has to finish solving the competition map before the competition. The third criteria requires the algorithm to stay within a 'reasonable' memory limit. This is coded directly into the solver as the maximum amount of states allowed in the closed set is defined in the code. If the closed set reaches the limit, the solver stops looking for a solution. This maximum amount of states can be set depending on time and memory constraints. In the test it has been set to a

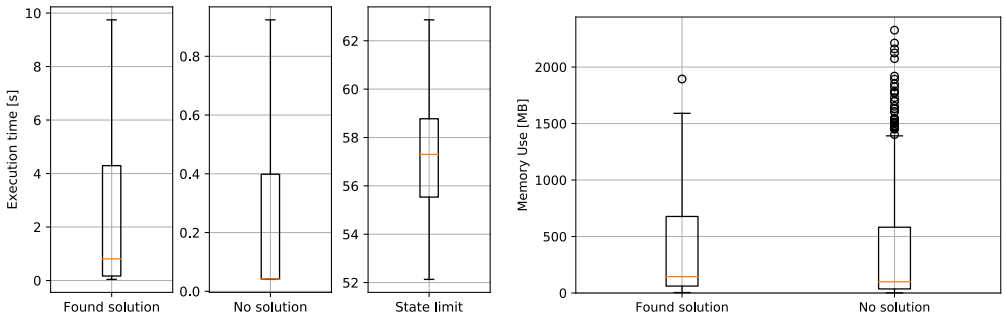
maximum of 10.000.000 states, corresponding to a memory use of around 2.5GB ($\mu = 2.47, \sigma = 0.11$).

To test the memory use, speed and general applicability, a map generator was made to generate random maps for the algorithm to solve. The map generator is programmed in python and is called *map_ gen.py*. The generator is made with the algorithm described in [5]. All combinations of map sizes and can amounts from the lists [[6,9], [6,12], [6,15], [9,9], [9,12]] and [3, 4, 5] respectively was found. For each of these, 10 different maps were made and for each of these maps, 10 variations of player, can and target positions were made. This results in 1.500 ($3 \cdot 5 \cdot 10 \cdot 10$) different maps being generated. The algorithm is applied to each map and the solution size, computational time, memory use, closed set size, open set size and map name is noted in a log file. The breakdown of solution outcomes can be seen in table 2 while boxplots of the time and memory use of maps where the solver either found a solution or found none can be seen in figure 7.

Out of the 1.500 runs, 7 runs where the algorithm found a valid solution succumbed to a bug in the reconstruction of the path. This bug is a result of a states parent being the child of itself, eternally looping between the two states. A way to fix this problem has been found but not implemented before executing the tests so the 7 results have been removed from the data set. 142 runs ran into the hard coded limit of the closed set size. This could be for a couple of reasons. Firstly, a likely reason for the large amount of errors is that the map is unsolvable but that the cans can still be moved around. This would result in every single state being checked before the algorithm would conclude that no solution could be found. Another explanation could be that the map has so many potential states that a solution cannot be found before the limit is reached.

Table 2: Overview of Sokoban solver test results.

	Count	Count [%]
Loop bug	7	0,47%
Closed set limit	142	9.47%
No Solution	1237	82.47%
Solved	114	7.60%
Total	1500	100.00%



(a) Timing results of solver test.

(b) Storage results of solver test.

Figure 7: Timing (a) and storage (b) results of the test of the sokoban solver.

The time and memory use can be broken down by map area size, which results in the data seen in figure 8. This shows that both memory use and computational time increases as a function of map area. Curiously, the 6x15 maps are faster to compute and uses less memory than both the 6x12 and 9x9 maps. This is probably due to the 10 maps generated being comparatively simple due to random noise.

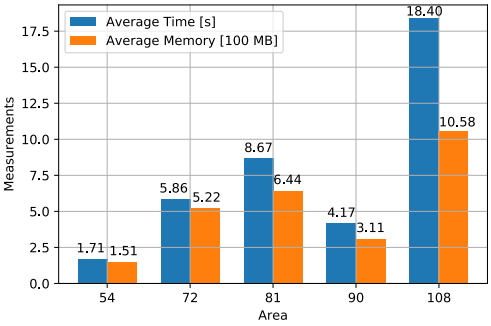


Figure 8: Average time and memory used to find a solution depending on map area.

As seen in the results, the algorithm is relatively fast at finding the majority of both solutions and maps with no solution (usually within 10 and 1 second respectively). When the algorithm is allowed to run to the limit, it uses a lot more time than what would be expected from a solvable map of the size seen in figure 8, which supports the hypothesis that the maps are not solvable, which causes the algorithm to be forced to check all possible states.

2.3 Conclusions

The algorithm seems to fulfill the three stated criteria of optimality, computational time and memory use. When testing the estimated time, the result is within 0.2% of the estimation (see table 1). When tested on maps generated at random, the program fails to comply to the memory limit in 10% of cases (see table 2). This is speculated to be cases where the algorithm is forced to look through every possible state without being able to find a solution. When the program is able to find solutions, it usually does so within 10 seconds of runtime and 2GB of memory (see figure 7).

When the algorithm is run on the competition map, a solution with 110 steps is found in around 8 seconds. This suggests that the map is in the easier end of the map complexity spectrum. The solution can be seen in the animation 9.

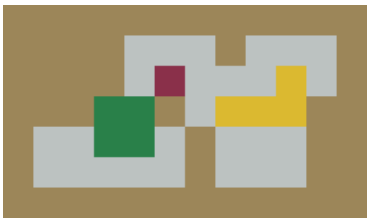


Figure 9: A GIF with the solution to the 2019 competition map. The animation has been successfully tested in Adobe Acrobat Reader DC (version 2019.008.20071) [6]

3 Bibliography

[1] *Sdu logo*. [Online]. Available: https://www.sdu.dk/da/nyheder/presserummet/logo_og_designguide (visited on 11/23/2019).

[2] F. F. Gan, "Cumulative Sum (CUSUM) Chart", *Wiley StatsRef: Statistics Reference Online*, pp. 1–7, 2017. DOI: 10.1002/9781118445112.stat04042.pub2.

[3] *Scikit-fuzzy*. [Online]. Available: https://pythonhosted.org/scikit-fuzzy/overview.html (visited on 12/17/2019).

[4] *Ai2 robot motion test*. [Online]. Available: https://www.youtube.com/watch?v=rN-1DH84HPg (visited on 12/17/2019).

[5] J. Taylor and I. Parberry, "Procedural generation of sokoban levels", *6th International North-American Conference on Intelligent Games and Simulation 2011, Game-On 'NA 2009, 3rd International North American Simulation Technology Conference, NASTEC 2011*, pp. 5–12, 2011.

[6] *Adobe acrobat reader dc*. [Online]. Available: https://get.adobe.com/dk/reader/ (visited on 12/17/2019).