Sokoban - Group 18

Jannes Helck
jahel18@student.sdu.dk
University of Southern Denmark
Campusvej 55, 5230 Odense, Denmark

Rasmus Haugaard
rahau15@student.sdu.dk
University of Southern Denmark
Campusvej 55, 5230 Odense, Denmark

Abstract

Sokoban is an old game in which an agent must push diamonds into goal positions in a square-tiled map. The agent may not pull the diamond nor drag the diamond to the side. A physical representation of the game has been made, where lego mindstorm robots move cans, and where the square-tiled map is represented with intersecting black lines on a white floor.

The problem was approached with a combined effort from the symbolic- and behavioural domains of Artificial Intelligence. The physical robot design and the implemented behaviours enabled the robot to drive forward one field in 0.93 s, turn left or right in 0.37 s, drive forward one field with a diamond in 0.98 s and push a diamond to the next section and return in 1.05 s.

Multiple heuristics with different levels of information and time complexities were implemented and tested. The fastest heuristic to find the optimal solution on the 2018 map was the can-closest heuristic in 61.6 s.

By using the unit move costs, the expected solution execution time on the 2018 map was found to be 150 s. The robot won the competition with a time of 142 seconds, 7, 17 and 28 seconds in front of the 2nd, 3rd and 4th place, respectively.

1. Introduction

The task of this project is to implement a solver algorithm for Sokoban puzzles and to build a robot agent that is capable of executing the solutions in a physical version of the problem. The task should be solved with approaches from Artificial Intelligence. A few different domains exist within the field of Artificial Intelligence. The solver and the robot implementation requires two different A.I. domains.

It is sensible to implement the Sokoban solver algorithm as a search in an abstract action space, after the symbolic A.I. paradigm. The solution can be found in advance, so there are no real-time requirements, and all the needed information about the puzzle is known in advance: the rules

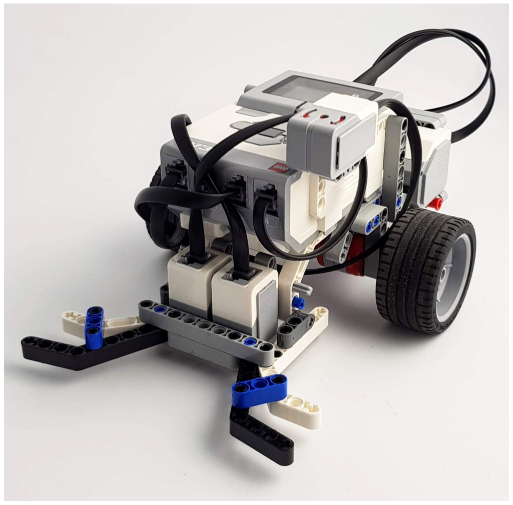


Figure 1: Robot agent - 3D view

of the game are well defined, and no sudden changes to the environment can make it necessary to reevaluate the action sequence.

The robot agent on the other hand needs to perform the sequence of abstract actions while responding to sensor data in real time. It needs to be able to handle a near-infinite combination of sensory input, and an attempt to find the optimal response is futile. Instead, the approach is based on the paradigm of behavioural A.I, where the intelligence is put into behaviours, wiring the sensors and the actuators togehter with simple logic and state, and even the body of the agent.

The approach to the given task is thus a combination of symbolic- and behavioural A.I domains.

2. Robot

The robot agent needs to execute the Sokoban solution found by the solving algorithm and to interact with the physical environment. This section deals with both the hardware design and the software implementation to accomplish this goal. Figure 1 and Figure 2 show the final physical design.

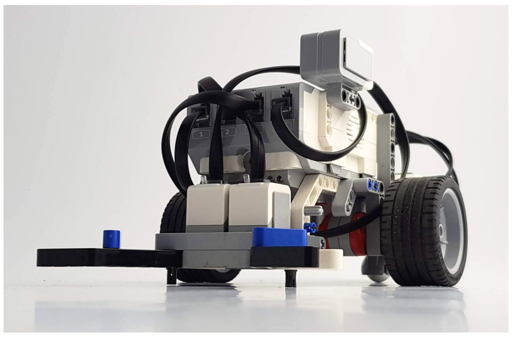


Figure 2: Robot agent - front view

Physical design

Two physical designs were tested. In both, a gimbal ball and two motorised wheels were used to enable in-place rotations. Two colour sensors were placed in front of the wheel axis to regulate the robot while following the line, and a gyroscope was used to indicate how close the robot was to completing a turn. With the first design, the robot rotated slowly and lost traction. In the second design, we attempted to solve this by moving the wheel axis much closer to the centre of gravity. This reduced the angular momentum and put more weight and thus more friction on the wheels, both leading to an increased turn speed. The second and final design can be seen in Figure 1 and Figure 2. Figure 3 shows the bottom view of the robot with the placement of the gimbal, the wheels and the colour sensors.

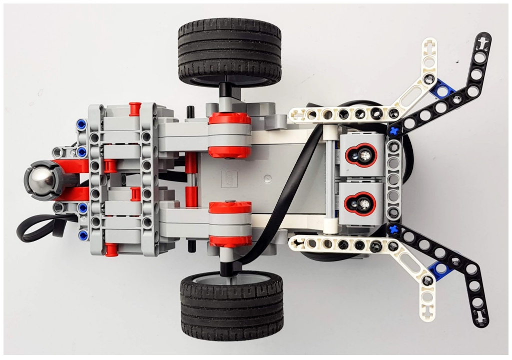


Figure 3: Robot agent - bottom view

Placing the sensors in front of the wheel axis enabled a stable regulation. To keep the effect of external light to a minimum the sensors were placed close to the ground. Spacers enforced a constant distance from the light sensors to the map surface, ensuring a consistent light measurement. The tachometers integrated in the servo motors are used to measure the travelled distance. To simplify the design no work was put into an active grabbing action. Instead, a simple, static v-shaped 'grabber/pusher' was used, which turned out to be sufficient without any sensors, actuators or additional software. This is a good example how the robot follows the paradigm of embodied A.I. [3], with intelligence built directly into the physical structure of the agent.

Behaviour Architecture

All behaviours are implemented as state machines. These behaviours and their relationships are shown in Figure 4. The state machines are executed in a cycle with lowest priority first. An info object is passed to the state machines, which are able to assign information to the object, accessible by higher priority state machines. A behaviour can request to set the wheel speeds by setting two specific values on the info object, and since higher priority behaviours access the info object last, they can overwrite the actuator output of the prior behaviours.

Compared to the subsumption architecture described in [1], our architecture is, due to the usage of the info object, not just able to override the output of lower priority behaviours, but also to access and use all information provided by them. This also allowed us to implement sensor value processing (see Implementation Details) as state machines as well, and to run them in the cycle before the behaviours, in order to provide higher level sensor information to the behaviours.

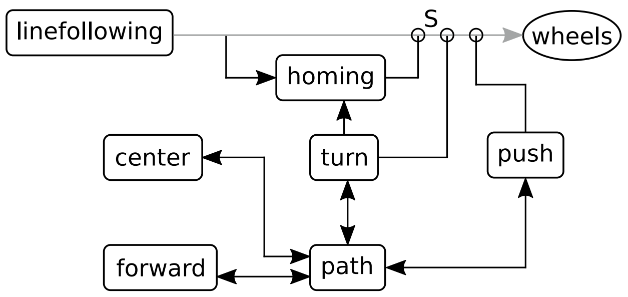


Figure 4: Behaviour architecture. Desired actuator output is set on the info object (grey line). Higher priority behaviours can read and overwrite the info object. Lines between the behaviours indicate a form of communication. The path behaviour starts the forward, centre, turn and push behaviours which again signals the path behaviour when they are done. The turn behaviour is able to reset the homing behaviour.

Implementation details

For a better intuition we used a conversion factor for the number of ticks obtained from the motor tachometers to cm.

For one cm we measured 617 ticks. This allows us e.g to set our threshold values directly in cm or to obtain our result data in cm.

For the angle measurement we used the gyroscope sensor. The built-in gyro integrator was drifting too much to be useful (but only when rotating), so a simple angle velocity integrator was implemented which provided much more consistent angle measurements. The reason for the poor performance of the built-in integrator is unknown.

For line detection we worked with a threshold value of 0.7 of a maximum sensor value, which is determined during a calibration phase in the beginning. A cross section is detected when both sensor values are below this threshold value.

Line following

The robot uses the sensor output of both colour sensors (see Figure 3 to follow a detected line. For instance when the right colour sensor detects a line and the left does not, the right the wheel is slowed down to a value of 0.6 of the base speed (95 % of max motor speed) to move the robot to the right. This works similarly for the left direction. In the case that both sensor values do not detect a line, the line following function preserves the current direction of the robot.

In order to make the line following more stable while crossing perpendicular lines, we implemented a debouncing function (see Figure 5). When the robot hits a perpendicular line it drives strait. After leaving the line the debouncing function makes the robot continue driving strait for a defined debouncing distance. After this distance, if none of the colour sensors is detecting a line, the robot is turned in the direction of the colour sensor that first detected the end of the line.

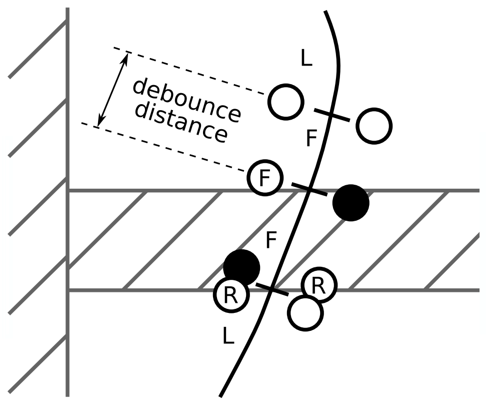


Figure 5: Debouncing functionality for line following. (Meaning of circles: R = rising edge, F = falling edge, black = on line, white = not on line

Forward

The forward function moves the robot forward until the colour sensors detect the next cross section. Figure 7 1. shows the position of the robot after an executed forward operation. To make sure that we do not the detect the cross section of the initial position we implemented a small debounce distance of 10 cm, before the end condition of this function is enabled.

Turn

Figure 6 shows the turning operation of the robot for a 90° turn to the left. For a turning operation the robot has to be centred. In the first phase the robot starts turning with 100 % motor speed (one wheel is driven with -100 % to enable in place rotation). The gyro-sensor is used to measure the angle the robot has rotated. After the angle threshold value of 40° is reached the motors are slowed down to 40 (-40) % motor speed to reduce the angular momentum. The robot stops turning when the colour sensor in turning direction detects a line. Therefore the robot is not completely aligned in 90°, but this turned out not to be a problem. To make the robot rotate around 180° we simply performed two 90° rotations.

Centre

The centre behaviour moves the agent from the end of a forward move, where the sensors detect a cross section, to the start of a turn move, where the wheels are centred on the cross section (see the robot in state 1 and 3 of Figure 7). The centre behaviour monitors the travelled distance since the end of the forward move, and signals when the distance from the sensors to the wheels has been travelled.

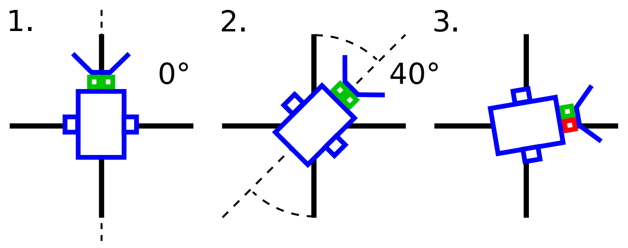


Figure 6: Turn operation.

Push

The push behaviour consists of two stages between three states, which are shown in Figure 7. In the first state after the previous forward move, the line sensors has just detected a cross section. At this point the, can is already placed in the robots grabber. In the first stage, from state 1 to 2, the push behaviour lets the line following behaviour control the wheels while monitoring the travelled distance. When the travelled distance reaches the distance between state 1 and 2, stage two starts. In order to place the robot in the correct position of state 2, we compensated for overshoot (see implementation). In the second stage, from state 2 to 3, the push behaviour takes control of the wheels, and drives straight back, until the travelled distance reaches the distance between state 1 and 3.

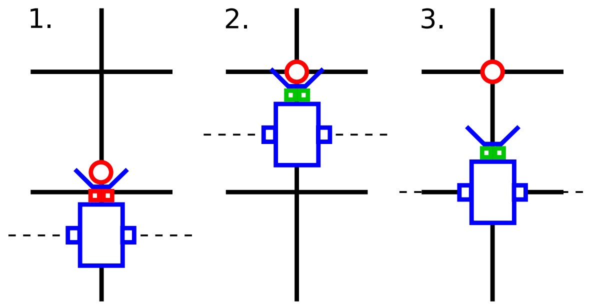


Figure 7: The three states of the push operation.

Homing

The push behaviour relies on the agent being oriented almost straight forward when entering stage 2, or else the agent will be much off centre in stage 3. The robot could be 'straightened' by not turning as hard in the line following, but this would make the robot less able to recover from angular displacements of the robot after turns.

Instead, to have the robot oriented more straight forward, a "homing" behaviour was implemented (see Figure 8, which slowly reduces how hard the line following behaviour is allowed to turn. After every turn, the behaviour starts monitoring the travelled distance. The allowed speed difference on the wheels is then linearly decreased as a function of the travelled distance. With 0 cm travelled, the slow wheel speed is 60 % of the fast wheel speed. From 60 cm travelled and above, the slow wheel speed is increased to 80 % of the the fast wheel speed.

2.0.1 Overshoot compensation

When pushing a can and moving the robot to a centred position on a cross section, the robot moves a certain distance measured by the integrated tachometer in the motors before stopping or reversing the motors. Since the motors are first stopped, when the target is reached, it overshoots. In order to compensate for this, the overshoot was measured for different velocities. The velocity-overshoot relation is approximately linear (see 9). That allowed us to compensate

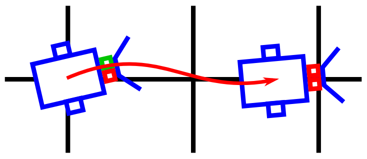


Figure 8: Homing. After a turn, the robot turns quickly to be able to recover from undesirable angular displacement. The angular displacement is then reduced during the following 60 cm by reducing how hard the robot turns during line following.

the overshoot by simply subtracting the predicted overshoot from the desired travelled distance.

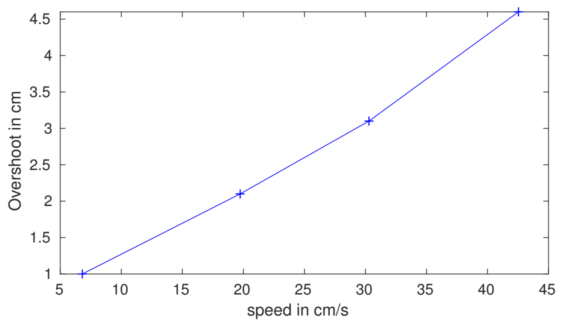


Figure 9: Overshoot as function of speed.

Path

The path behaviour is initialised with a string consisting of four basic operations; forward(f), push(p), left(l) and right(r) and introduces a centre(c) between a forward and a turn step ("fr" $\rightarrow$ "fcr", and "fl" $\rightarrow$ "fcl").

The Path goes sequentially through the letters of the string and signals the appropriate behaviours. When a behaviour is done, it signals the Path behaviour back, which proceeds to the next letter in the string. With no more letters in the string it stops the motors.

Validation

For validating the correct function of the robot we used different test sequences. Simple forward steps, a combination of forward moves and turns, and an integration test that includes forward moves, turns and pushes. After each test sequence, the path behaviour logged the average duration for each move type. We continuously used the test sequences to improve the speed and robustness of the behaviours.

Results

The robot design turned out to be simple yet effective. The usage of a passive grabber/pusher was sufficient. Placing the wheels close to the centre of gravity enabled fast rotations. The decision to place the two colour sensors in front of the wheel axis and using state machines, line following with debouncing and homing, to control an on-off turn direction resulted in an amusingly erratic, yet quite fast and robust line following behaviour. Table 1 shows the measured time "unit cost", the robot agent needs to execute the basic movement functions.

Function	Unit cost
Forward	0.94 sec
Turn	0.37 sec
Forward (with diamond)	0.98 sec
Push	1.05 sec

Table 1: Measured unit costs for the different actions of the robot agent

3. Solver

The solver should be able to produce a solution represented as a string of moves to the robot. Since the solver can be run ahead of the competition, the primary focus should be on assuring, the solution is indeed the solution with the lowest physical execution time.

In the standard Sokoban game, no cost is connected with turning the agent, nor pushing cans. With the physical implementation though, both actions take time. Instead of defining the cost as the move count, the four unit move costs were defined for the agent, namely; forward, turn, forward with diamond and diamond push, each cost representing the time the move takes. The cost of the path from state a to state b is then the lowest sum of unit moves able to take state a to state b.

State representation

The choice of state representation has at least two requirements apart from correctly representing the state: It should be memory efficient, and it should be efficient to expand the state's neighbours in the search tree.

The most obvious state representation might be a matrix, including goals, walls, agent and cans - but it violates both requirements of the state representation. Instead, we make a clear separation between state and map. We define the map to be a matrix containing walls and goals, while the state describes the agent- and can positions in the matrix as well as the agent orientation. If the agent is currently pushing (or 'holding') a can, it has to do a push move before it can make a turn. We therefor also represent whether the agent is currently pushing a can, in the state. The state should also have a pointer to where it came from in the search tree, so the whole solution can be found from the goal node.

A Sokoban map consists of both the actual map and an initial state. Since we also take agent orientation into account, we expand the initial state into four initial states, representing all possible start orientations, each with zero cost.

State Manipulation

We keep the possible moves from the original Sokoban problem. The agent can either go up, right, down or left at any point if there is no wall there, and no can there followed by another can or a wall.

We create a three-dimensional move cache table, of size $h \times w \times 4$, where each entry holds the possible moves from the corresponding agent position and orientation in the map. For each possible move, the rotation cost, the new agent position and orientation is cached. It is also cached where a can in the new position would be pushed, if there is not a wall there.

When the neighbours of a given state node is needed, a simple look-up in the move cache is done. If there is a can in the new position, the can can be pushed if there is not also a can in the position where the can would be pushed. The neighbour state nodes can thus easily be found, and after the heuristic is added to the node, it can be added to the open list.

3.0.1 Heuristic thoughts

The heuristic has three requirements: It must be an underestimate for the search to be A*, so we can rely on the first expanded goal to represent the optimal path. We would like the time complexity of the heuristic evaluation to be low. And we want it to be as informative as possible to discard as large portions of the search tree as possible, exploring fewer state nodes before reaching the goal.

A more advanced Heuristics might be more informative, but might also have higher time complexity, so there is a compromise between the latter two requirements.

To be able to keep the time complexity of the heuristics somewhat low, we completely ignore can-can interactions in our heuristic, which enables us to do a lot of caching.

Move cost cache

Ignoring can-can interactions enables us to pre-compute the lowest cost of moving a can in any position to any other position. We initialise a table of size $h \times w \times 4 \times h \times w \times 4$ with infinity, where each entry represents the cost of moving a can from one position with an agent orientation, to another can position with another agent orientation. If the agent is orientated upwards, the agent is assumed to be below the can, etc.

The table is populated with unit costs. It costs one *forward with diamond* to move a can from (y, x, UP) to (y-1, x, UP). Note that our coordinate system is setup so that the y-axis is oriented downwards, and the x-axis is oriented to the right. The cost of changing the direction will be split into two parts. First part is actually pushing the can to the can position (since the robot is currently 'holding' it), turning and driving once forward. The optimal solution would always have to do this when changing agent direction. The second part is turning, driving forward, turning and driving forward once more. This second part though, as in the case when turning during the move of the diamond immediately above the agent in Figure 10b, might actually be used to push other diamonds, so adding the cost of the second part would violate the underestimate requirement.

With the table populated with unit costs, the shortest path from all to all states is found by the Floyd-Warshall algorithm [2]. Simplified, the Floyd-Warshall algorithm works by updating the cost $a \to b$, if for a pivot state c, $a \to c \to b$ is cheaper. The time complexity is $O(n^3)$, where n is the number of vertices, or states in our case: $O((h \cdot w \cdot 4)^3)$. By utilising, that $a \to c \to b$ cannot be cheaper than $a \to b$, if $a \to c$ or $c \to b$ is infinite, we are able to reduce the execution time significantly: For the 2018 map, the move cost cache build time was reduced from 23.5 seconds to 0.09 seconds.

Since we might not care about the initial and final agent orientation, we expand the matrix with an imaginary fifth orientation which represents the orientation with the lowest cost of the four real orientations.

Following the same procedure as above, the cost of moving the agent from any position and orientation to any other position and orientation can be cached, assuming there are no diamonds in the way.

See *solver/src/h/CostCache.py*. Some details are left out here for brevity.

Heuristics

First we will look at estimating the cost of moving cans to goals, assuming the agent does not have to travel between cans. Then we will look at estimating the cost of travel between cans. Unless stated otherwise, future uses of *n* refer to the number of cans (or goals).

The first two heuristics are quite similar and only differ in whether cans are assigned to goals or the other way around. In the **can-to-closest-goal** heuristic, the lowest can move cost from a can to any goal is chosen and summed for all the cans. If a can cannot be moved to any goal, this heuristic will be infinite, representing a deadlock. In the **closest-can-to-goal** heuristic, the lowest can move cost from any can to a goal is chosen and summed for all the goals. If no can can be moved to a goal, this heuristic will be infinite. We define a combination of the two, the **can-closest** heuristic, to be the maximum value of the two heuristics. All three heuristics have time complexity of $O(n^2)$.

To utilise the fact that there can be only one can on a goal, the **min-matching-cans-to-goals** heuristic populates a square cost matrix of size n with the costs of moving cans to goals and finds the linear sum assignment by the Hungarian method [4]. This is more informed, but also has time complexity $O(n^3)$.

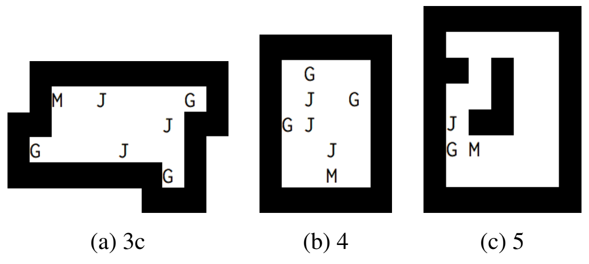
For now, we have only looked at the cost of moving cans, but moving the agent to cans also costs. The agent needs to move to a can before pushing it. If there are empty goals left after having pushed a can to a goal, the agent needs to move from the goal to a new can. To be able to cache the following heuristics independent of the agent, the first agent move from the start position to a can is not considered. In the **goal-to-closest-can** heuristic, the lowest agent move cost from a goal to any can is chosen and assigned to the goal. Since the agent doesn't need to move after the last can push, and since we don't know the order of the can pushes, we sum all but the largest goal cost to avoid overestimating the cost. In the **closest-goal-to-can** heuristic, the lowest agent move cost from any goal to a can is chosen and assigned to a can. Like above, all but the largest can cost is summed. We define a combination of the two, the **agent-closest** heuristic, to be the maximum value of the two heuristics. All three heuristics have time complexity of $O(n^2)$.

To utilise the fact that there can be only one can on a goal, the **min-matching-agent-to-cans** heuristic populates a square cost matrix of size $n + 1$ with the costs of moving the agent from the goals to the cans, and finds the linear sum assignment by the Hungarian method. The last column and row in the cost matrix is set to zero to take into account, that the agent doesn't need to move after the last can push. This is more informed, but has time complexity $O(n^3)$.

Lastly, to estimate the cost from the agent start position to the first can, we define the **agent-to-any-can** heuristic to be the lowest agent move cost from the start position to any can, since we don't yet know, which can is the first one to move in an optimal solution. This adds only little information, has linear time complexity $O(n)$ and has to be evaluated every time the state changes, not only when the can positions change.

We assemble five combinations of the heuristics:

- **can-closest**



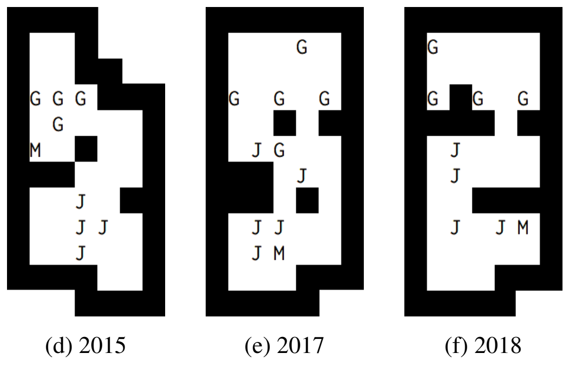


Figure 10: Test(a, b, c)- and competition(d, e, f) maps. M: Agent, G: Goal, J: Can

- **closest**: can-closest + agent-closest
- **min-matching-can-to-goal**
- **min-matching**: min-matching-can-to-goal + min-matching-agent-to-can
- **min-matching+**: min-matching + agent-to-any-goal

All underestimates, but with different levels of information and different time complexities.

Comparing a can to goal Manhattan distance, the can-closest (which is the least informed heuristic above) is much more informed and has the same time complexity. The Manhattan distances are therefor not examined.

Validation

The solver was validated in a few different ways. First, we made a program that enabled us to go through a graphical representation of the solution to make sure, we had something sensible. Then we carefully designed some test maps with known optimal solutions and made sure the solver found those solutions.

Six maps are used in the following sections. Three test maps as well as three competition maps. The maps are shown in Figure 10.

Test map 3c and 4 were carefully designed to enable the min-matching+ heuristic to be optimal at the initial state of the map. In Figure 11, the heuristics are plotted for each step along the solutions for all six maps. The theoretical optimal solution (final cost - current cost) is also plotted for reference. If any of our heuristics would be higher than the theoretical optimal heuristic at any point, our heuristic would not be an underestimate, which helped us find and correct a few bugs in the heuristics. It is clear from the heuristic plots, that the min-matching+ is indeed the most informed heuristic, and it also clearly shows how our heuristics generally perform poorly on test map 5, since even the most informed one, min-matching+ will calculate the distance from the robot to below the can, and then from above the can, pushing the can down. See the discussion of the search trees for test map 4 and 5 with the min-matching+ heuristic in Figure 12.

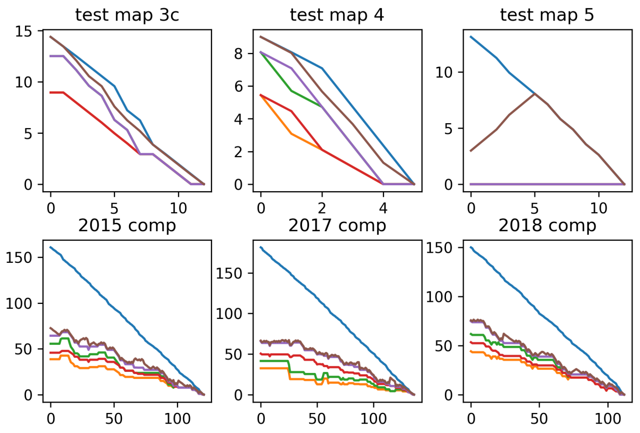


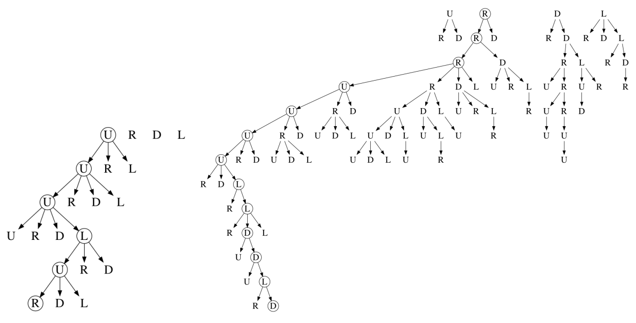
Figure 11: Solution step heuristics. For each of the six maps, six heuristic costs are plotted along the second axis as function of the solution step index along the first axis. The six heuristics are: the theoretical optimal heuristic (Blue), can-closest (Orange), closest (Green), min-matching-can-to-goal (Red), min-matching (Purple), and min-matching+ (Brown).

Implementation details

The solver is written in Python3.

A min heap is chosen for the open list, which enables $O(logn)$ insertion and extraction time complexity.

Our heuristic is not consistent: The total cost for nodes in an optimal path is not generally increasing: A state might expand into another state along the solution path for which the heuristic is less informative as for the previous state. To be sure the first time we reach the goal is through an optimal path, we then create a closed set map, with states as keys and path costs as values. We can then skip adding a nodes neighbours to the open list, if and only if the value for the node state key in the closed set map is lower than or equal to the node path cost. The closed set dictionary has time complexity $O(1)$ for both look-up and insertion.



(a) test map 4 (b) test map 5

Figure 12: Search trees with the min-matching+ heuristic on two test maps. An edge represents a move. The node letters represent the orientation of the agent, Up, Right, Down or Left, after the previous move. The direction of the move thus equals the orientation of the agent in the following node. (a) For test map 4, the heuristic enables the algorithm to only expand the steps in the solution. (b) For test map 5, none of the described heuristics takes into account, that the agent needs to come from above to push the can to the goal, so the algorithm needs to explore many nodes, not in the solution path.

Space complexity

There are two great contributors to memory usage. The closed set as well as the state nodes that are part of a path in the open list and therefor kept in memory. For the current implementation, each node has space complexity $O(n)$, storing all the diamond positions. Each key-value pair in the closed set map is $O(1)$, storing only a hash and a path cost.

The importance of state node memory vs closed set memory efficiency seems to be different from map to map, since the ratio between referenced nodes and entries in the closed set varies quite a lot. The garbage collector can free memory from nodes that are not referenced anymore, since they are not in any path to the nodes in the open list. For the 2017 map, 3.6 million nodes are expanded, and at the time, where the solution is found, only 1.0 million nodes are referenced and kept in memory. For the 2015 map, 322 thousand nodes are expanded, and only 10 thousand nodes are referenced, when the solution is found.

The referenced nodes could be represented more efficiently. After a node has been extracted from the open list and expanded, it is only used to represent the move and point to the previous node in the path. This means, after expansion, it would be enough to store the direction of the move as well as a pointer to the previous node, with each node then having a constant space complexity, $O(1)$.

The greatest space complexity improvement though, would probably be changing the node-expander to expand diamond moves instead of agent moves. Each expansion could then do an agent bfs to all possible diamond unit moves, calculate the heuristic for each diamond move and only add those diamond moves to the open list. This would reduce the space complexity of both the closed set and the referenced state nodes. Memory usage for the six described maps will be shown in results.

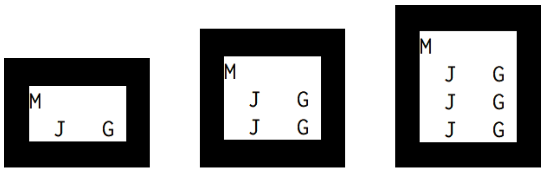


Figure 13: The first three time complexity maps. A new row with a can and a goal is added for each iteration.

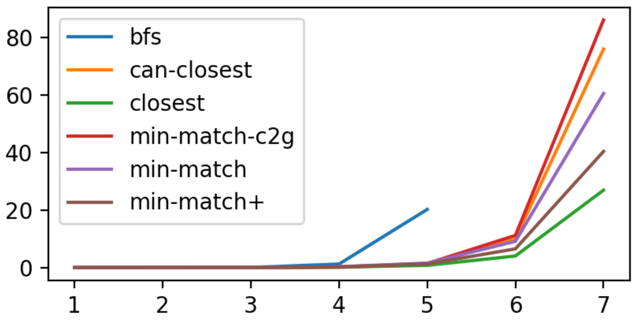


Figure 14: Solve time in seconds along the second axis as function of number of cans along the first axis. Each heuristic was allowed a 150 second run time for each map. None of the heuristics were able to solve the map with eight cans in that time.

Time complexity

A simple procedurally generated map was used to create maps with different amounts of cans. In Figure 13, the first three maps are displayed. The solve times are plotted in Figure 14. Breadth first search (bfs) is added for comparison. While the *closest* heuristic is the fastest for all the procedural maps and solves the map with seven cans in about 30 seconds, it is not able to solve the eighth map in 150 seconds. Note that the solve time for the maps are not just dependent on the amount of cans but also for example the width of the map, which changes the search tree size. With three empty spaces instead of just one between the can and the goal, none of the heuristics solved the map with 6 cans in 150 seconds.

Results

Heuristic\Map	3c	4	5	2015	2017	2018
bfs	30	29	29	561	8,000+	948
can-closest	30	29	30	116	1,218	398
closest	32	30	31	123	1,079	417
min-match-c2g	31	29	30	94	1,114	247
min-match	31	30	31	102	1,084	323
min-match+	31	30	31	114	1,125	363

Table 2: Memory usage in MB for all heuristic / map combinations. Bfs / 2017 did not finish.

Heuristic\Map	3c	4	5	2015	2017	2018
bfs	836	254	109	3.6M	45M+	8.8M
can-closest	71	63	63	359k	6.1M	1.5M
closest	19	11	63	365k	5.6M	1.5M
min-match-c2g	71	43	63	325k	4.7M	1.4M
min-match	19	11	63	326k	4.0M	1.3M
min-match+	13	6	45	322k	3.6M	1.2M

Table 3: Expanded node counts for all heuristic / map combinations. Bfs / 2017 did not finish.

Heuristic\Map	2015	2017	2018
bfs	186	–	517
can-closest	12.4 ± 0.9	317 ± 4.6	61.6 ± 0.7
closest	13.7 ± 1.3	299 ± 4.5	62.4 ± 0.7
min-match-c2g	16.8 ± 1.2	329 ± 4.4	65 ± 0.9
min-match	24.7 ± 0.4	384 ± 3.5	86.2 ± 0.8
min-match+	26.9 ± 0.2	388 ± 3.3	93.7 ± 0.9

Table 4: Solver times in seconds for the competition maps. Mean and standard deviation for 8 solves. Bfs was only done once for each map and didn't finish the 2017 map.

In Table 2, the memory usage for each heuristic / map combination is shown. The memory usage changes a lot between maps, but not significantly between the implemented heuristics (excluding bfs).

In Table 3, the expanded node count for each heuristic / map combination is shown. As expected, the number of expanded nodes are lower for more informed heuristics. The difference though, between the most and least informed heuristics, min-matching+ and can-closest (excluding bfs), is not even a factor of two in the competition maps.

In Table 4, the times for each heuristic / competition map is shown. Interestingly, the fastest and least informed heuristics (can-closest and closest) are performing the best. This shows the need for experimentally choosing the compromise between the information and the time complexity for a Heuristic.

We chose to calculate the node costs based on the robot unit move costs instead of move count. In order to validate this choice, solutions were found simulating the move count cost by assigning 1 to forward moves and 0 for the rest. In Table 5, these results are compared to our cost representation. With our unit cost representation, we estimate to be able to execute the physical solution between 2.6 and 5.5 s faster, than if we used the simpler move count cost representation.

Cost\Map	2015	2017	2018
move count	165.3	183.7	152.7
unit move costs	160.8	181.3	150.1

Table 5: The expected physical execution time in seconds based on the robot unit move costs, calculated for the solutions found using two different costs: move count cost, and unit move costs.

4. Integration test and competition results

Until now, the robot and the solver has been tested separately. A final robustness test was done with 10 attempts on the 2018 map. The robot succeeded in 5 of the 10 attempts with execution times between 144-146 seconds, around 5 seconds faster than the expected execution time of 150 seconds. All teams had three attempts during the competition. Assuming a success ratio of 50%, the chance of completing at least once was $1 - 0.5^3 = 87.5\%$.

On the competition day, with a battery charged over night, our robot executed the solution of the 2018 map in 141 seconds before the competition in the first try, and during the competition the robot took 1st place in 142 seconds, also in the first try. 2nd, 3rd and 4th place were 7, 17 and 28 seconds behind, respectively.

5. Conclusion

The problem was approached with a combined effort from the symbolic- and behavioural domains of Artificial Intelligence. The physical robot design and the implemented behaviours enabled the robot to drive forward one field in 0.93 s, turn left or right in 0.37 s, drive forward one field with a diamond in 0.98 s and push a diamond to the next section and return in 1.05 s. Multiple heuristics with different levels of information and time complexities were implemented and tested. The fastest heuristic to find the optimal solution on the 2018 map was the *can-closest* heuristic in 61.6 s on average. By using the unit move costs, the expected solution execution time on the 2018 map was found to be 150 s. The robot won the competition with a time of 142 seconds, 7, 17 and 28 seconds in front of the 2nd, 3rd and 4th place, respectively.

References

[1] R. A. Brooks. A robust layered control system for a mobile robot. *IEEE J. Robotics and Automation*, pages 14–23, 1986.

[2] R. W. Floyd. Algorithm 97: Shortest path. *Commun. ACM*, 5(6):345–, June 1962.

[3] J. Hallam. Ai1ah architectures: Module notes. 2005.

[4] H. W. Kuhn and B. Yaw. The hungarian method for the assignment problem. *Naval Res. Logist. Quart*, pages 83–97, 1955.