Application of search algorithms in solving a NP-hard problem

Group 3: Mikael Moghadam, Kenni P. Isaksen, Morten S. Laursen

October 24, 2009

1 Introduction

This report documents the development of a system that can solve a sokoban challenge in a real world environment. The report consists of two parts, the first one concerns the construction of a robot able to traverse a sokoban field given the correct movement commands. The second part describes the development of a practical sokoban solver that is able to generate a command file for the robot, enabling it to solve the sokoban challenge.

Contents

2 Architecture

2.1 Division of responsibilities

The architecture of the system is composed of two separate systems, a planner and a physical robot. The planner runs on a PC and is responsible of creating a file with the actual plan, for solving the task at hand. The planner takes a map as input and uses a search-algorithm to find the optimal path to the goal in a search tree. When the optimal path is found a file with the robot-navigation-commands is created.

The robot-navigation-command-file contains a string of commands. These commands are up, down, right, left and turn around, they are represented as 'u', 'd', 'r', 'l' and 'a' in the file.

The file created by the planner is used by the robot to correctly navigate the map as fast as possible (the optimal path is the fastest). The physical sokoban map is made op of fields consisting of black lines, that the robot must be able to follow. The robots task is to move from one field on the map to another as fast as possible. The robot itself does not perform any path-finding, but only navigates the map using the commands from the file created by the planner software.

3 Robot description

The purpose of this prototype is to perform the assignment as described by the planner. In order to accomplish this task a robot has been created. The robot is created using the LEGO NXT platform.

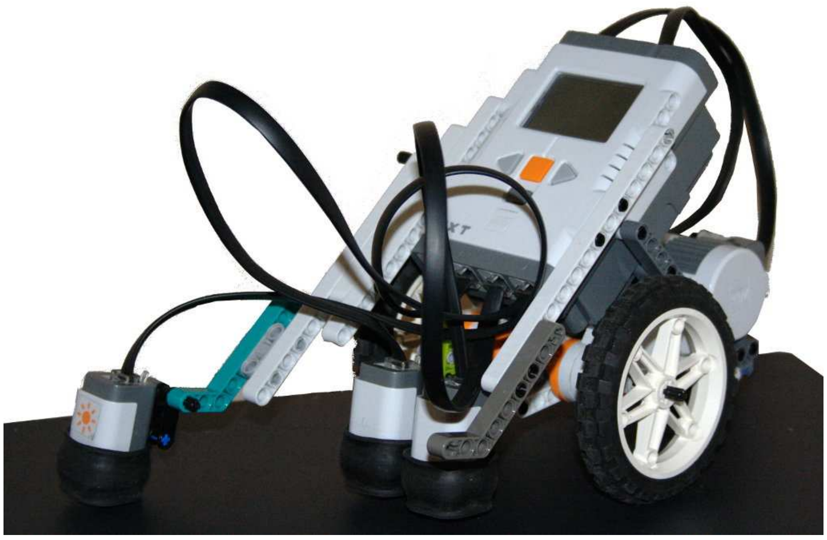

Figure 1: Picture of the robot

3.1 Physical Construction

The robot is constructed using two motors which applies force to each of their front wheels, this allows the robot to steer using skid steering. In front of the wheels two light sensors are placed to detect the line. Using two sensors, in place of just one sensor, on top of the line allows for a wider range of detection scenarios. If only one sensor was used the robot would be forced to follow one of the edges of the line. Having two sensors allows for seeing the line as in the middle (none of the sensors see's the line), slightly to the left (the left sensor see's the line edge), far to the left (the left sensor see's the line), of course the same goes for the other direction. Line edges can be detected because the light sensors measure the amount of reflected light. When the area scanned by the sensor is partially covered by line, the sensors values also only changes partially. Having more than one sensor results in a larger coverage area and a higher detection resolution, which allows for more fine grained control, and recovery of a larger misplacement of the robot.

In front of the two sensors, on the right side of the vehicle, another light sensor is placed. This sensor is placed in front of the vehicle in order to detect crossing lines as early as possible, to allow the vehicle to slow down, as it cannot stop instantly. The placement of the three optical sensors are as illustrated in figure 2

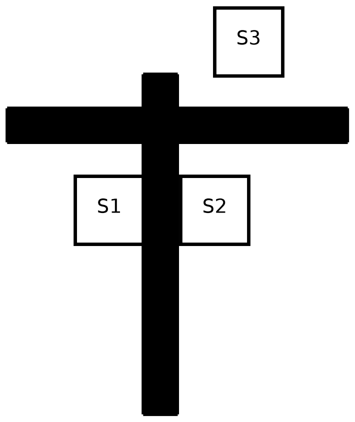

Figure 2: Illustration of light sensor placement, with the sensors mentioned S1-S3

3.2 Robot architecture

The software for the robot is implemented as a statemachine, which interprets the commands given from the planner. A statechart of the statemachine is shown in figure 3 The states is described in the following.

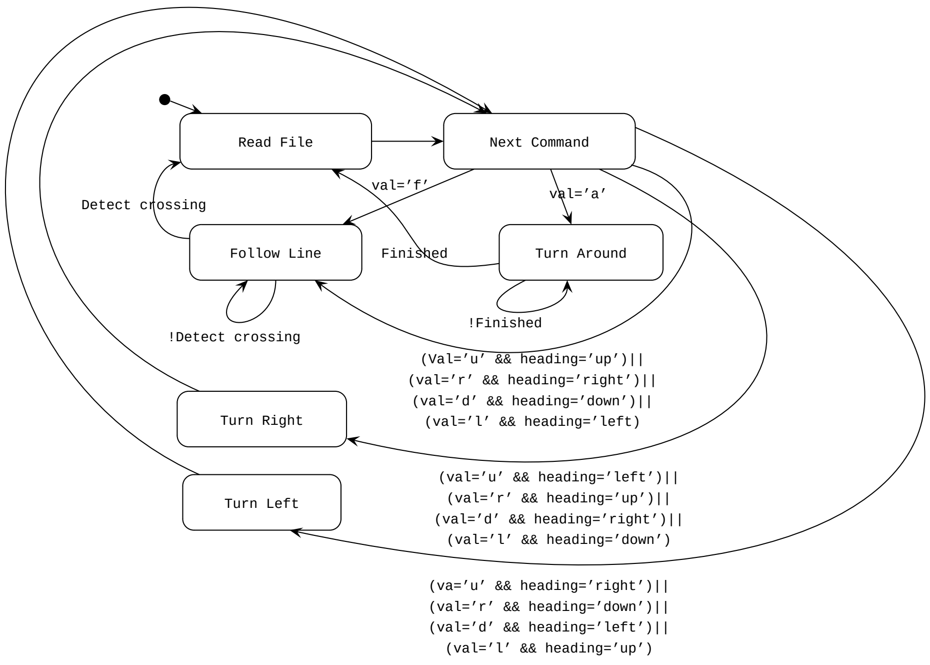
Read File
Next Command
val='f'
val='a'
Detect crossing
Follow Line
Finished
Turn Around
!Finished
!Detect crossing
(Val='u' && heading='up')||
(val='r' && heading='right')||
(val='d' && heading='down')||
(val='l' && heading='left)
Turn Right
(val='u' && heading='left')||
(val='r' && heading='up')||
(val='d' && heading='right')||
(val='l' && heading='down')
Turn Left
(va='u' && heading='right')||
(val='r' && heading='down')||
(val='d' && heading='left')||
(val='l' && heading='up')

Figure 3: Statemachine of the Sokoban robot

3.2.1 Follow line

The line following state has the responsibility of keeping the robot constantly on the line using the sensors to allow for precise navigation.

On figure 4, a plot of what the sensors measures when driving on a line is shown. It is clear when looking at the plot that the signals do not contain a lot of higher frequency components and will therefore not be improved much from being filtered, it would however cause less phasemargin because of the filterdelay. These data was measured doing a evaluation of a proportional feedback control system.

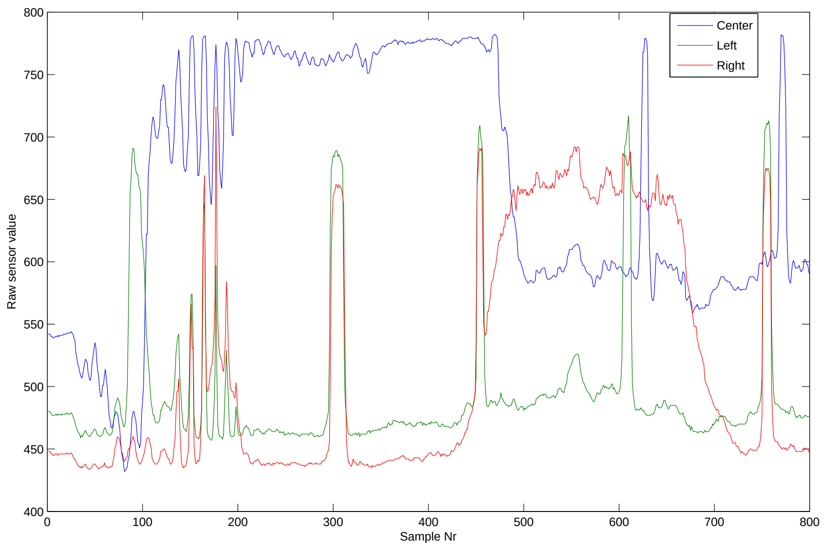

Figure 4: Raw samples of sensor measurements

Statemachine based For making the control loop, two different implementations have been tested. One using a simple state machine based solution, where the state machine tries to assess the position of the line as either in the middle (none of the sensors see's the line), slightly to the left (the left sensor see's the line edge), far to the left (the left sensor see's the line) then from this approximation of the relationship between the line and the robot an action is performed by lowering the speed on the wheel too close to the line according to the approximation.

PD-feedback based Another option which has been tested for the line following, is a PD control loop. The PD control loop has the advantage of potentially providing a much smoother and precise

level of control, contrast the state machine based approach which is more jittery because it does not start correcting before reaching a threshold value.
On figure 4 between sample 100 - 200, the sensed values can be seen during a correction using the PD-regulator. This causes the regulator to correct for the measured error continuously, where the state machine based regulator, only corrects on state changes.
Having implemented both solutions on the vehicle the control loop based solution was chosen as it had clearly the best performance. This was proven by allowing both algorithms to follow a 2m stretch of straight line for 100 runs and comparing the amount of errors of both algorithms.
In this test the state machine based method failed in 13% of the cases, where the PD regulator failed in only 2% of the cases.

3.2.2 Turn left and turn right

The turn states makes the robot turn it's heading to the right or the left, it does this by using the tachometer for feedback. To achieve the correct turning radius a combination of turning one wheel forward while the other is turning backwards is used. This can be seen from the example in listing 1.

Listing 1: TurnLeft Pseudo example

```
reset Tacho
until tachoCount > 1800 do:
    TurnRightMotor Forward at 90% power
    TurnLeftMotor Reverse at 30% power
return
```

3.2.3 Turn around

Turning the robot 180 degrees is more advanced than the simple turn left and right states, because the precision requirements for this turn are higher. It therefore consists of a number of steps which it must pass through:

1. Back off to avoid hitting the jewel when turning
2. Turn 160^o around based on the tachometer
3. Turn until the frontmost sensor see a line
4. Turn the last bit using the tacho

The amount of tachometer ticks needed to perform the remainder of the 180 degree revolution (from step 3 to step 4) is fairly constant, and is therefore implemented as a constant value. This algorithm turns out to work accurately, but was first devised towards the end of the project and therefore comprehensive testing and tuning could not performed.

3.2.4 Read command

This state reads the next character from the text file and transitions to the next state.

3.2.5 Next command

This state interprets the command read, it just directs the call to the correct action state, left, right, turn around etc., based on an internal variable containing the vehicle's heading.

3.3 Conclusion

The completed vehicle was able to perform very well when going straight and when performing turns in most cases, however turning immediately after a 180 degree turnaround proved to be very difficult, even though the final position after the 180 degree turnaround varied very little. If time had permitted it this could be improved a lot by relying more on the optical sensors when performing the turn. It is however noteworthy that the vehicle proved to be the fastest vehicle at placing the first jewel in the competition, and during the competition for ten consecutive trials performed without error until the first turn following a 180 degree turnaround, where it failed consequently.

4 Planner description

4.1 Problem discussion, in the Sokoban domain

Sokoban is a game originating from Japan. The essence of the game is to push jewels around in a room containing obstacles, in a manner that will enable the player to place all the jewels on so called goals in the room. See figure 5 for a screenshot of a sokoban game. Sokoban is an interesting game to study in the domain of Artificial Intelligence, because it presents a so called NP-hard problem, meaning in this case that it is often fairly easy for humans to solve such puzzles in a manner of minutes, but very difficult to develop an efficient algorithm able to do the same. This is due to the fact that NP-hard problems spawn huge search trees and branch factors.

4.1.1 The rules of the game

The rules of the game are simple. The game is best described as consisting of squares, each having an (x, y) coordinate. A game consists of the following squares:

- One 'Man' square, representing the player. The 'Man' square can move around.
- One or more 'Jewel' squares representing objects that have to be pushed onto 'Goal' squares by the player.
- One or more 'Goal' squares. There are always the same amount of 'goal' squares as 'Jewel' squares.
- One or more 'Wall' squares representing impenetrable boundaries which can not be passed by neither 'Man' nor 'Jewel' squares.
- One or more 'Empty' squares representing space where the player/'Man' square is able to move.

As stated previously the purpose of the game is for the player to push 'Jewel' squares onto 'Goal' squares. That the player is only able to push jewels signifies that if a jewel is at position (x,

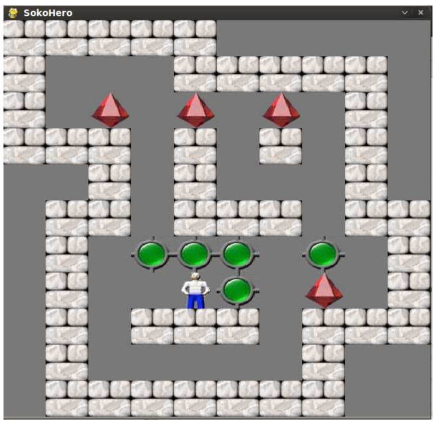

Figure 5: A screenshot of a sokoban game

y), and the player is at position (x - 1, y) then the player will be able to move the jewel to position (x + 1, y) if (x + 1, y) is occupied by an 'Empty' or a 'Goal' square. The same logic applies for other directions.

4.1.2 Challenges faced when solving Sokoban algorithmically

Given the fact that the player can on average move up, down, left and right, we see that the problem presents a so called branching factor of 4. If an algorithm is able to solve the puzzle in 100 moves, the complexity becomes 4^{100} . This enormous complexity, illustrates the need for some sort of heuristics or pruning of the search tree, if the problem is to be solved in an acceptable time frame. Deadlocks are one of the main challenges faced when trying to solve the Sokoban puzzle. A deadlock is best described as a situation where a jewel is pushed into a position that eliminates the possibility of solving the puzzle, for instance pushing a jewel into a corner.

4.1.3 Algorithmic approach

The general approach taken to solve the sokoban problem, is creating a search tree where every node in the tree is a state of the game. The goal is then to traverse the tree to find the state where all of the coordinates of the jewels match the coordinates of all the goals.
The general algorithmic search can described as:

Listing 2: Deadlock detection pseudo code

```
if not problem solved:
  get next state in search tree
    if state not goal state:
      for each jewel in state do:
        for every direction in which jewel can move one square do:
          if resulting state is valid and not duplicate of a previous state:
            if robot can get to the position to push the jewel in the desired direction:
              create new state
              move jewel to new position
              move robot to previous jewel position
              calculate state score
              add state to search tree
```

As can be seen from the pseudo code, the problem solving can logically be divided into two separate subproblems, the first one being the movement of the jewels, henceforth known as the jewel problem, and the second being the path finding from the robot to the jewel that has to be moved, henceforth known as the path finding problem. The search approach best suited for the solving of these two distinct areas might not be the same, and therefore each subproblem is considered separately, and the solution implemented will treat the subproblems as two seperate search trees. It is important to note that the optimal solutions is wanted in terms of minimum number of robot/man movements. This is as mentioned due to the fact that the competition is a race, and the assumption therefore is that the less squares that the robot has to traverse, the less time it will take to solve the puzzle.

4.2 Discussion of search algorithms

The following algorithms have been considered to solve the jewel problem

As mentioned the two things that can potentially hope to minimize the search time needed to solve a sokoban problem, namely a 'good' heuristic search and search tree pruning. The search algorithm, and therefore by extension also the heuristic function, is concerned with the 'get next state in search tree' part of the algorithm in listing 2, and pruning is concerned with the 'is state valid' part.

A*

This was the first algorithm considered using for the problem, because it is both optimal, complete and fast if a good heuristic function is used. It has however proven fairly difficult to find a 'good' admissible heuristic for the jewel problem. One of the only possibilities that have come to mind is the sum of each jewels manhatten distance to it's closest goal. This heuristic would be admissible, since the closest distance from a jewel to a goal would naturally be the minimum amount of moves needed to move the jewel in order to solve the problem. It is our assessment that this heuristic unfortunately can not be characterized as a very good one, since the sum of smallest distances between jewels and goals, is by no means an estimate of the number of moves left to solve the problem, due to the intrinsic complexity of most sokoban problems (jewels getting in the way of other jewels, jewels being blocked by walls etc.). In other words, a good A* heuristic should guide the search down the branch of the search tree most likely to contain an acceptable solution, thereby minimizing the nodes/states that would have to be traversed in order to get to a solution. The considered heuristic would not necessarily be able to do so. The cost that would be used in this implementation, would be the total number of moves the robot/man would have to take to move the jewel to the desired position plus the robot moves taken to get from the initial state (root state) to the current state.

Greedy search

By disregarding the Heuristic function from the A* search, and instead using the cost of robot movements as the heuristic, we would in effect get a greedy search. We would use a so called closed list, to avoid loops, thereby attaining completeness. The downside of this algorithm however is that it is not optimal, which is a requirement if we are to get a solution with the minimum required number of robot movements.

Breadth first search

The breadth first search algorithm is inherently slow for complex problems, but is however optimal in so called unit-step cases (the cost from one state to the next is always the same), which speaks greatly in favour of this algorithm in this case. However for this to work it is important to remember that the unit-step requirements necessitates an implementation where each robot movement spawns a new state, as opposed to just spawning a new state for each jewel movement, so that the depth of the solution in the tree will correspond with the number of movements performed by the robot. The use of this algorithm therefore requires that the two subproblems be solved as one. This will, among other things, increase the amount of memory needed by the algorithm.

Uniform cost search

A variation on the breadth first search and greedy search exists, where the next node visited is the one with the least total cost from the root node (the same cost as used in the A* algorithm), as opposed to the greedy search where the next state visited is the one with the least cost from the current node, which was the cause of the non optimality of the greedy search. This is therefore both a complete and optimal algorithm in terms of minimum robot movements, when using the robot movements as cost. Furthermore it does not require, as the breadth first search, that the sokoban problem not be solved as two subproblems.

Search tree pruning

Since a good heuristic function has not been found for the search algorithm, the time complexity of the solver will presumably be high, unless the search tree can be pruned significantly. Fortunately the sokoban problem is one where a large amount of pruning is possible in terms of deadlock detection.

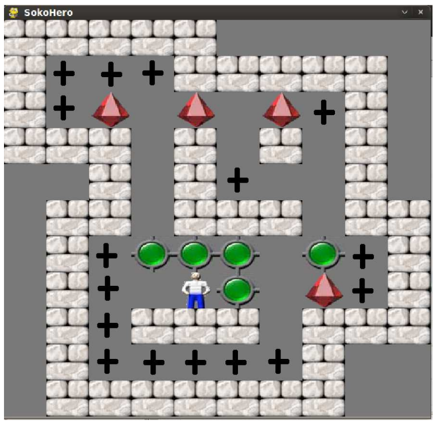

Figure 6: A screenshot of a sokoban game with deadlock zones marked with '+'

Figure 6 shows all the zones where the placement of a jewel would result in an unsolvable problem, meaning that the state and all states spawned from that one are useless, and would only potentially waste the time of the solver if it would have to visit them.

The following deadlock detection algorithm is devised, where return true symbolises that the state has a deadlock:

Listing 3: Deadlock detection pseudo code

```
For each jewel do:
  if jewel in corner:
    return true
  if jewel against vertical wall:
    for each square above jewel:
      if square is goal:
        return false
      if square on the obstructed side of jewel is empty or goal or jewel:
        return false
    for each square below jewel:
      if square is goal:
        return false
      if square on the obstructed side of jewel is empty or goal or jewel:
        return false
  if jewel against horizontal wall:
    for each square to the right of jewel:
      if square is goal:
        return false
      if square on the obstructed side of jewel is empty or goal or jewel:
        return false
    for each square to the left of jewel:
      if square is goal:
        return false
      if square on the obstructed side of jewel is empty or goal or jewel:
        return false
  return true
```

This deadlock detection algorithm could be improved further by checking for deadlocks due to one jewel obstructing another in a irreversible way. Due to the deadline of the project however, it has not been possible to include this feature.

The following algorithms have been considered to solve the path finding problem

A*

This algorithm is very often used as path finding algorithms in games, because good heuristic functions are relatively easy to find when confronted with such problems. Often the manhatten distance to the goal is sufficient enough to give very fast searches compared to depth- or breadth first. Therefore we do not consider any other searches for the path finding problem.

4.3 Choosing the search algorithm

Naturally the A* algorithm will be used to solve the path finding problem. The choice of search algorithm for the jewel problem would have fallen on the Uniform Cost Search algorithm, because it gives an optimal solution in terms of robot movements which is of most importance, since the time to solve the problem is irrelevant because the specific problem as mentioned will be given a week in advance. We will however, out of pure academical interest, implement all of the considered algorithms for the jewel problem, in order to compare their performances.

4.4 Implementation

4.4.1 Choice of programming language

The first language considered for implementing the solver, was Python. Python however turned out to be very slow for this type of problem, in particular when using a debugger. We therefore turned to java, because java still provides a significant programming abstraction compared to C or C++, allowing us to focus more on the algorithms than on memory allocation and deallocation. Furthermore java has a standard library that is more mature and easier to use than the standard template library of C++.

4.4.2 Map representation

The particular Sokoban puzzle that must be solved by the robot, is handed out beforehand in a textual format. An example of this textual format is:

```
XXXXX
X......XXXXX
X..J..J..J...X
XXX..X..X...X
  X..X.....X
 XX..XXX..XX
 X..GGG..G..X
 X....MG..J..X
 X..XXX..XXX
 X............X
 XXXXXXX
```

where 'X' represents 'Wall' squares, '.' represent 'Emtpy' squares, 'J' represents 'Jewel' squares, 'G' represents 'goal' squares and finally 'M' represents the 'Man' square.

This map representation will be converted directly into a data structure containing x y coordinates (lists), in order to use the map in graph traversal by the chosen search algorithm. Each state will contain one list representing the coordinates of the empty squares, one list containing the coordinates of the jewel, one list with the goals and finally the coordinate of the man. In this way a move can quickly be investigated to determine its legality, and two states can be compared for equality. An alternative representation could be a multidimensional array, directly translating the map into an array of characters. This would however most likely require more programmatical effort in regards of state comparison and move legality checking.

4.4.3 Robot movements in real life

When the lego robot moves to a jewel (A can of tomato pure) in real life, the square movements do not match the actual map representation. This is mostly due to the fact that the can of pure does not fill an entire square in real life, and neither does the robot. The practical consequences of this is:

- When the man/robot gets to a jewel in the map, it has in reality to move an extra square to get to the can of pure.

- When the robot is moving a can, and has to change to another can immediately, it has to back up one square first.
- When the robot has finished moving a can, and has to perform its next move without the can, it has to move back one square first.

These 'extra movements' that have to be taken depending on whether or not previous and next moves of the robot involves a jewel, will be taken into account when implementing the planner (specifically where the number of movements are used to calculate the score of the state), so that it will be possible to get a solution that contains the minimum number of moves that the robot has to perform in real life, as opposed to the minimum number of steps that have to be performed when adhering solely to the rules of sokoban and the textual representation of the map.

4.5 Performance evaluation

For each search algorithm, three consecutive runs have been performed with each search algorithm, and the data from the best run chosen.

Search algorithm	States traversed	Memory used [MB]	Robot moves	Time taken [s]
Uniform Cost	18968	39	142	202
Breadth first (Non unit-step)	19761	40	162	218
A*	18788	39	142	189
Greedy	18215	48	212	179

Table 1: Results from the different search algorithms.

Another search has been performed with the tree pruning algorithm disabled (No removal of deadlock states), to see how big an effect the pruning has on the performance of the solver. The search algorithm used is A*, because it outperformed the other search algorithms.

Search algorithm	States traversed	Memory used [MB]	Robot moves	Time taken [s]
A*	148097	318	142	12223

Table 2: Results from the different search algorithms, when search tree pruning is disabled.

4.6 Planner conclusion

As can bee seen from the Uniform Cost search in table 1, the optimal solution contains 142 robot movements (102 if the solver did not take into account the number of moves that the robot must perform in reality). As predicted the breadth first and greedy searches do not find the optimal solution, and would therefore not be applicable to use for this particular project, even though greedy search outperforms the other searches in terms of speed. As it turns out the A* search outperforms the Uniform Cost Search in number of states traversed and time taken, which was not all that unexpected. It can therefore be concluded that the sum of the manhatten distances between the each jewel and its closest goal, is a heuristic that will yield an optimized search. It can furthermore be seen from table 2, as expected that the effects of pruning far outweigh the gains that one particular search algorithm provides over another in terms of time taken to find solution, the states traversed and the memory used.

5 Conclusion and discussion

The developed system is able to correctly calculate the optimal solution for solving the sokoban challenge, furthermore the robot is able to navigate the board in most aspects, however there is still some challenges remaining with a subset of the directional turning. In section 3.3 a solution is proposed for solving this problem.

The system was able to outperform the competitors during the competition in placing the first jewel, and up until the first time it had to perform, one the actions from the problematic subset of turning. This was due to a design decision of the robot, where the team aimed for achieving the highest possible speed, at the cost of an increased risk of not completing the competition.