AI00 - Sokoban solver
Final report in AI00 2007

Title	AI00 - Sokoban Solver
Written by	Brian Horn, Bjørn Grønbæk & Jon Kjærsgaard
Deliverydate	November 26, 2007
Course	AI00

Contents

Todo list

Chapter 1

Introduction

1.1 The Objective

The objective is to navigate a course for a Sokoban game. A robot will be required to effectuate a solution to a given problem. The solution to the problem will be calculated offline, and the robot must then function as a means to translate the solution into the real world.

A complete model of the course is known in advance, and a plan for a solution is calculated on a computer separate from the robot, and transferred to the robot as a series of commands.

This document describes the implementation of a system to solve this problem.

The real Sokoban course is a grid of black tape on a white background, the points where two tape lines meet, the intersection, corresponds to a field in the model. The model does not contain any data about the distance between points, nor does it contain data on irregularities in the playing field etc. The robot must therefor compensate for these on its own.

1.1.1 The Competition

All groups taking the AI00 course must participate in a competition, where the objective is to solve the real world puzzle in the shortest amount of time.

1.2 This Report

This report consist of two major parts.

Part One: The first part describes the robot used to solve the puzzle. This part is mostly the report that was delivered as a preliminary report, but modified in accordance with received feedback.

Part two: The second part consist mainly of the offline path planning. Also there is a part describing the modifications made to the robot in response to problems revealed by running the actual solution on the competition course, rather than the test course.

All relevant source code is placed in a separate appendix section.

Part I

The Sokoban Robot

Chapter 1

Description of the Robot

1.1 Requirements

From a cursory inspection of the problem it is evident that the following components is needed:

Sensors In order to navigate the course some kind of input from the physical world is required.

Actuators In order for the robot to solve the problem it needs some way to affect the world.

Stable frame In order to use the sensors and actuators in a meaningful manner, knowledge about their position relative to the rest of the robot is needed. Also there must be some kind of guarantee that they will not move significantly from this known position. This means that the frame/chassis must be a stable construct.

A "brain" Some way to evaluate the sensor input, and activate the actuators is needed.

The design must be able to achieve the following three goals:

- Navigating the field.
- Moving a "diamond"
- Placing a "diamond"

It is not necessary to lift a diamond, and it is not legal to turn while moving a diamond. It is however legal to pull the diamond back if it is done in order to place it accurately.

1.2 Implementation choices

The robot is build from LEGO Mindstorm, which means that a lot of factors are predetermined. The actuators will be the LEGO rotational motors. The "brain" will obviously be the LEGO NXT block. As this has three output and four inputs, the number of sensors and actuators is limited. Also physical dimensions of the blocks and weight must be taken into consideration.

1.2.1 Navigation

The playing field is marked in black and white, and it can be assumed that a full model is known to the program that plans the movements. Further we assume that we will not have to deal with unknown obstacles, such as other vehicles. In this case only sensors that detect the black line that is to be followed is really necessary. It was determined that two light sensors placed close to each other, and at a distance from the turning point would be sufficient to detect if the robot follows the line. Additionally the sensors will provide enough information to correct the direction of the robot as needed.

LEGO Mindstorm come with building instructions to a number of designs. Most of these use the same basic chassis. As this chassis is a very stable design, we chose to use this as the basis for the frame.

The design uses three motors as a integral part of the design. Two for driving and turning, and one for other purposes. We only need the two, but have kept the third as it adds stability to the chassis.

Figure 1.1: Chosen chassis

Figure 1.1 is a CAD drawing showing the robot as it is currently implemented. The boom in front pushes the "diamond" between fields. The sensors behind the boom are used for following the black lines, and detecting intersections. The front most sensor is used to detect intersections while pushing "diamonds", in order to ensure that the "diamonds" are placed exactly on the intersection.

The two wheels are used to both drive and steer the robot, with a single Bogey wheel for balancing the tail. Each wheel is driven by separate motors, allowing for a very sharp turning radius.

1.2.2 Sensor Placement

The two front sensors are placed centrally on the robots front end, at a specific distance from the robot's turning axis, as shown in figure 1.2. The two front sensors are placed as close as possible to each other, while still being placed on each side of the black line that the robot follows.

The position of the sensors are important. If they get to far from the robot's turning axis, there is a danger that both sensors will get on the same side of the line before the direction can be corrected. This happens because the turning speed of the sensors, if placed to far from the robot's turning axis, gets to fast for the sensor sampling rate, and thus the robot cannot react in time. This results in the robot straying from the path, which is an unrecoverable error.

If they get too close to the robot's turning axis, the robot might already have turned a substantial number of degrees, before the turn is detected by the sensors. This results in a "zig-zag" movement of the robot, which significantly slow-down as a consequence.

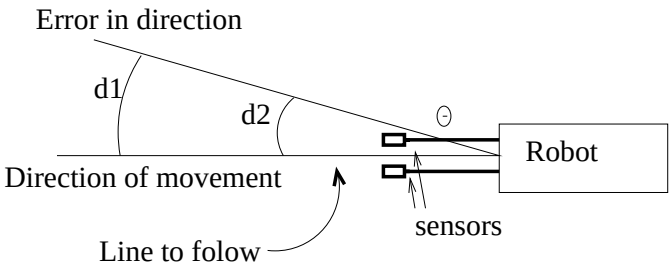

Figure 1.2: Sensors placement

1.3 Modifications

The final implementation of the robot, is the result of a iterative process, in which the robot was subjected to a series of tests, interspersed with redesigns.

The physical design of the robot changed as a result of both physical requirements of the game, as well as modifications to the behaviours. For example it became evident that the initial design had a turn circle that was too wide, and as a result the front end, where the sensors are mounted, was shortened. This gave a much smaller turn circle.

When placing the sensors on the robot it was important to keep in mind, that if the sensors came too close to the axis around which the robot turns, it would no longer be able to drive in a straight line. Therefor it became a matter of iteratively changing the placement of the sensors, in order to maximise the line following ability, while at the same time keeping the turn circle small enough.

Similarly it was detected that the initial design had no way of stopping the robot when the "diamond" was exactly on the intersection. This was solved by adding the front sensor. This sensor is only used when pushing a "diamond".

Chapter 2

Robot Behaviour

The behaviour of the robot can generally be separated into three parts, which combined controls the robot in its entirety. The three behaviours are basically:

- path following
- rotation
- decision making

The behaviours are discussed in detail in the next section.

The behaviours are implemented as a mix of tasks and functions, and complex behaviours are generally made up as a combination of more primitive behaviours, to ease the implementation.

2.1 Behavioural Analysis and Design

To design the robots software routines, an analysis of the needed behaviours were performed.

The primary and very basic behaviour needed, is following the paths/lines on the Sokoban field. This means following a path from one field, to another field.

The obvious behaviours needed are: **Forward**, **Right turn**, **Left turn** and **Reverse**. Further analysis of the robots behaviour and the playing field revealed the need for some additional behaviours: **Forward with diamond** and **Turn 180**. The behaviours are summarised and described in table 2.1.

All behaviours are based on the specific sokoban board used in this project. This means that behaviours are based around the black lines on the board, and most importantly: the intersections between the black lines.

No	name	description
1	Forward	Follow a line until the next intersection is reached.
2	Reverse	Reverse along a line until the next intersection is reached.
3	Turn Left	Rotate left until the left line of the intersection is reached, and then go forward (1).
4	Turn Right	Rotate right until the right line of the intersection is reached, and then go forward (1).
5	Forward with diamond	Like Forward (1), except that the robot must stop when the diamond is on the intersection.
6	Rotate 180	Like performing two Right turns (4) in a row, except that the first turn must not be followed by a Forward

Table 2.1: Behaviours for the Sokoban robot

The robot uses the sensors to detect when the robot is placed exactly on top of an intersection. Only on intersections will new behaviours be performed. If for example the robot is performing the Forward behaviour, it will keep doing that, until it detects an intersection.

All behaviours will automatically take the robot from one intersection and to the next intersection. When the robot is placed on an intersection, and starts the Turn left behaviour, it will rotate 90 degrees left, and the automatically proceed forward to the next intersection.

2.2 Behavioural Implementation

The software for the robot is written in the Not eXactly C (NXC) programming language using the BricxCC IDE. The most important fact to remember when discussing the software design and implementation, is that NXC allows multi-tasking to take place. This means that all the **task** sections of the code, are run in *parallel*.

As stated, the complete behavioural system of the robot, is composed of several sub-systems responsible for a limited functionality. The complete systems consists of several tasks all running simultaneously and continuously, and a number of functions for performing limited functionality specific to a certain situation.

2.2.1 Tasks

The system utilises three task for controlling the robot's motion and current state. Additionally the main task is responsible for the configuration of the various sensors, and is run prior to the three controlling task. The three control tasks are started simultaneously and once started they cannot be interrupted. To allow for a task to be temporally stopped and later restarted, a double while construct, as shown below, is utilised:

```
task SomeTask(){
  while(true){ //run always
    while(somevariable){ //only run when somevariable is true
      // ... some code here
    }
  }
}
```

By setting the inner variable **true** or **false**, the running can be disabled or enabled as needed.

Motion control tasks The two motion control tasks are the most important tasks in the system, and are the basis upon which all other motion is based. Each task is responsible for controlling the speed of one of the robots two motors. As long as the sensor, placed on the same side as the controlled motor, is *observing* a white surface the motor is kept running. If the sensor observes a black surface, indicating the sensor is now over a black line, the controlled motor stops. The basic functionality is illustrated in the following code:

```
task MotionTaskRight(){
  while(true){ //run always
    while(right_motor){ //only run when true
      if(right_sensor > RIGHT_SENSOR_THRESHOLD) OnFwd(RIGHT_MOTOR);
      else Off(RIGHT_MOTOR);
    }
  }
}
```

Due to the placement of the sensors, relative to the turn-point of the robot, this keeps the robot aligned with a sensor on each side of the black line, when moving forwards. When an intersection is reached, both sensors will observe a black surface, and the robot will stop.

No	name	description
1	RunStraight	Move the robot forward.
2	RunRight	Turn 90 degrees right.
3	RunLeft	Turn 90 degrees left.
4	RunRightRight	Rotate right 180 degrees.
5	RunLeftLeft	Rotate left 180 degrees.

Table 2.2: Main behavioural functions for the Sokoban robot

State control task The state control task continuously evaluates the input from the three light sensors on the robot. When the two sensors in front of the wheels both report black, the robot has reached an intersection. When this happens a list of commands is queried for the next command to be performed, e.g. go forward, turn left, etc. The principle is shown in the pseudo-code below:

```
task ControlTask(){
  while(true){ //run always
    if(both sensors show black){
      cmd = getNxtCmd();
      Switch(cmd){
  case FORWARD:
    //some code here
  case LEFT:
    ...
      }
    }
  }
}
```

The state control task first ensures that the motion tasks are disabled (the motors are already stopped, since both sensors are over a black line), so that the motors will not start again, before the robot is ready to perform its next command. The **switch** control structure then evaluates the next command, and calls one or several functions, to get the robot to do the queued command. Finally when the function report it is done, the state control task enables the motion control tasks again.

2.2.2 Functions

Several functions implement specific behaviours needed in specific situation. In general the functions are invoked by the state control task, when the robot is navigating before running forwards again. The main functions are listed in table 2.2.

RunStraight

The **RunStraight** function makes the robot drive forward a specific distance. In contrast to the motion control task, the sensor values are ignored, and the robot drive straight forward (synchronised motors) without regard for the black lines. This is useful for moving the robot away from an intersection, so the sensors get back on the white surface, without triggering the state control task again.

RunLeft and RunRight

The **RunRight** and **RunLeft** functions turn the robot 90 degrees right or left respectively. To turn the robot both motion control tasks are disabled, and the RunStraight function is called to move the robot off the intersection manually. The right or left motor is then activated manually, to turn the robot a fixed number of degrees (about 45 degrees). This is done to ensure that the

sensors are now all away from the black lines. Finally the left or right motor is activated, by enabling one or the other of the motion control task, according to the direction the robot should turn. When the active motion control task senses a black line again, e.g. that it has turned 90 degrees, the other motion control task is activated, and the robot drives forward along the line again.

RunLeftLeft and RunRightRight

The **RunRightRight** and **RunLeftLeft** functions are, as the names apply, continuation of the RunRight and RunLeft functions, just rotating the robot 180 degrees instead. Basically they are identical to the 90 degrees version, except that they repeat the turning-part of the function twice, before driving forward again.

2.3 Sensor Adjustment

The sensors are used in a mode that gives a percentage value. A lower value means that the sensor reads less light, in this case the black line. Likewise a high value means that the sensor reads the white board. Under different lighting conditions the precise threshold value between a black and a white reading differ somewhat. However the behaviours are made sufficiently robust that a exact value are not required. Experiments have shovn that a threshhold of 50% is almost always good enough.

Chapter 3

Performance Test

The following section contains a description of the various test scenarios of the robot. These tests should not be seen as the final evaluation of the robot, but rather as preliminary experiments of the morphology and physical design of the robot. We have performed a series of tests, where the purpose of each test is to reveal potentially weak design decisions, primarily with provide us with enough knowledge to be in a position to correct these potentially bad design choices successfully. Moreover, the tests should bring useful information regarding the correct adjustment of the different parameters; like optimal power values of the motors, sensitivity of the light sensors, etc.

The overall goal of the project is that the robot should be able to play the Sokoban game. However, before considering strategies and algorithms to solve this task, we have taken a bottom up approach; meaning that we have implemented basic motion behaviours like the ability to follow a line and performing turns when necessary. The test base for these experiments is shown in figure 3.1. The physical model of the field for playing Sokoban is a white square, with an area of approximately 1.5 m^2. The valid pushing paths are indicated by black tape, forming a grid-like pattern as in figure 3.1.

Figure 3.2 shows a magnified outline of the grid from 3.1. The black dot represents a can, which is the object the robot must push around the grid path. In the real Sokoban game the objects are diamonds - here cans resemble diamonds.

3.1 Test 1

In this test, the robot tracks a path formed by two squares, where the perimeter of one square touches, without intersecting, the perimeter of the other square thereby forming the number eight (in digital). By navigating this particular pattern, the robot is forced to perform both left- and right turns. The test is performed ten times and with different power values of the motors. The result of the test is shown in table 3.1

Test	Speed	Rounds	Completed	Error %	Average lap time [s]	Remark
1	60	10	10	0	32	None
1	70	10	10	0	29	None
1	80	10	10	0	27.9	None
1	90	10	1.5	15	30.6	Fails in turn

Table 3.1: Result of test 1.

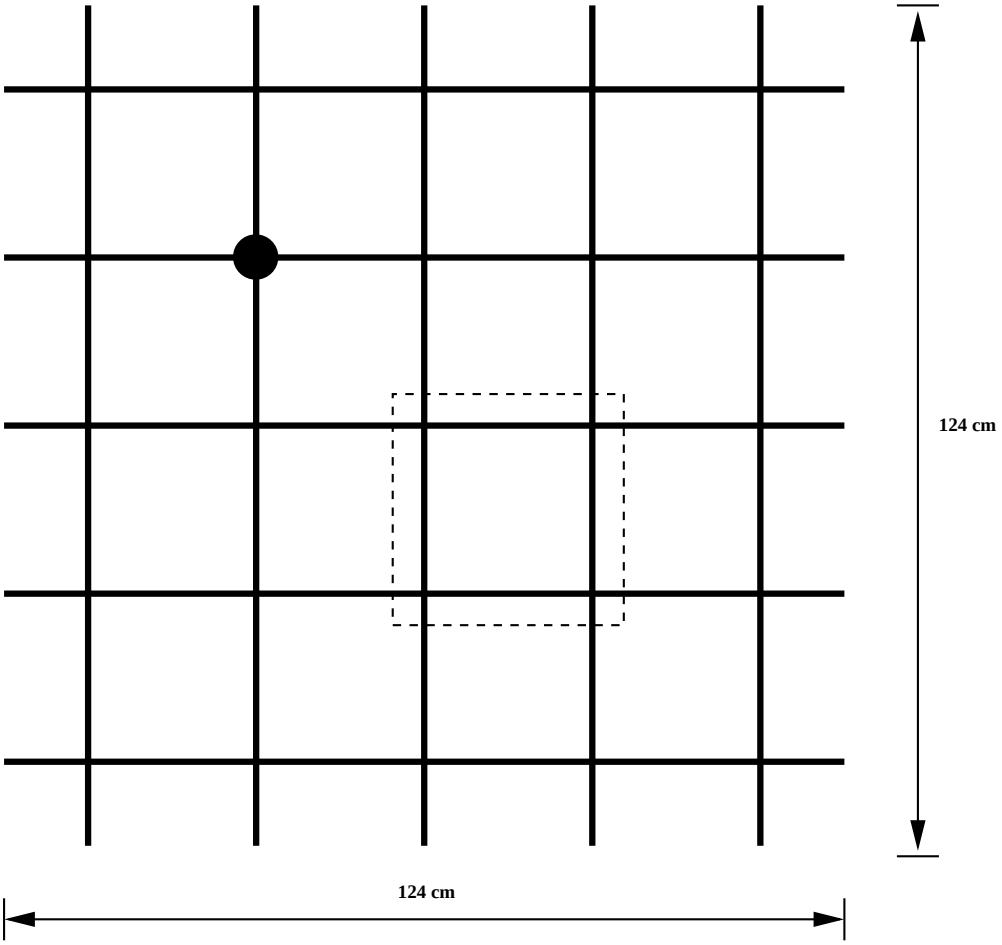

Figure 3.1: The grid layout representing the environment that the robot operates in.

3.2 Test 2

In this test, the robot tracks a path between two points. A strip of black tape connects the points. The distance between the points is approximately 45 cm. The robot starts from one point, with the line properly placed between the two front sensors, thereby facing directly towards the opposite point. When the robot reaches the opposite point it performs a 180-degree turn and continues toward the staring point. This cycle is repeated ten times with different power values of the motors. The result of the test is shown in table 3.2

Test	Speed	Rounds	Completed	Error %	Average lap time [s]	Remark
2	60	10	10	0	14.6	None
2	70	10	5	50	13.4	Fails after turn
2	80	10	1.5	15	16.6	Fails in turn

Table 3.2: Result of test 2.

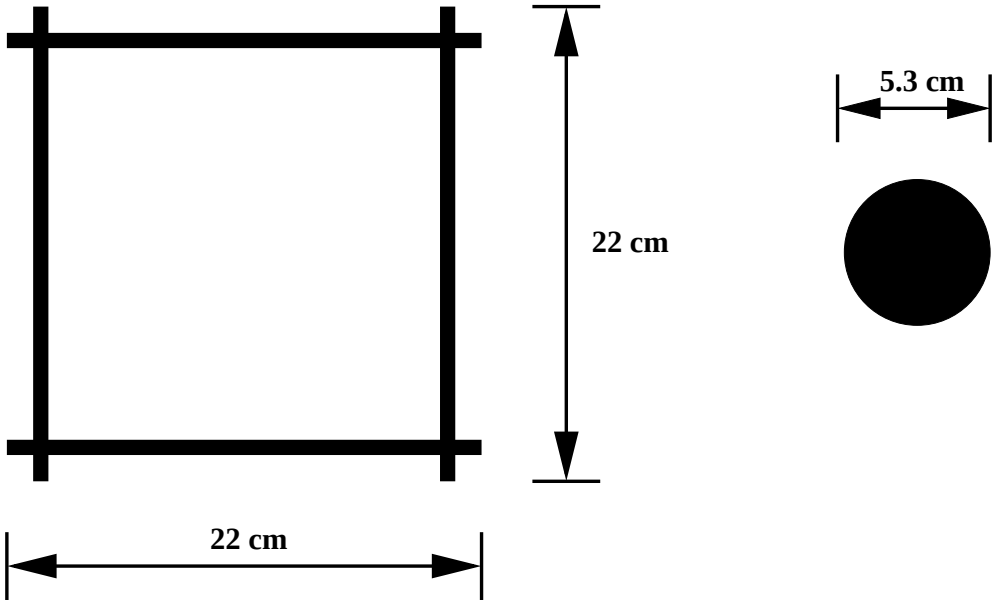

Figure 3.2: Close view of the field in which the robot operates.

3.3 Conclusion

By observing the performed tests, and specifically their point of failure, several additions to, and fine-tuning of, the robot's behaviours were done. The most important is the introduction of variable power setting for the motors, based on the *previous* command. This for example enables the robot to set the Forward speed setting, to a lower value after performing a 180 degrees turn, where it potentially has a problem finding the black line again.

Table 3.3 shows the optimal speed setting derived from the performance tests, for several behaviours. With the adjusted speed settings, the robot is able to perform ten runs in every test, with 100% success rate.

Situation	Speed	Description
Forward	80%	When running directly forward between intersections.
Turn left / right	70&	This is the maximum reliable speed when turning 90 degrees.
Rotate 180 degrees	60%	Maximum reliable speed when turning 180 degrees.
Forward after 180 rotation	60%	This is the maximum speed, where the robot is able to determine a line after 180 rotation, 100% reliable.

Table 3.3: Variable speed settings.

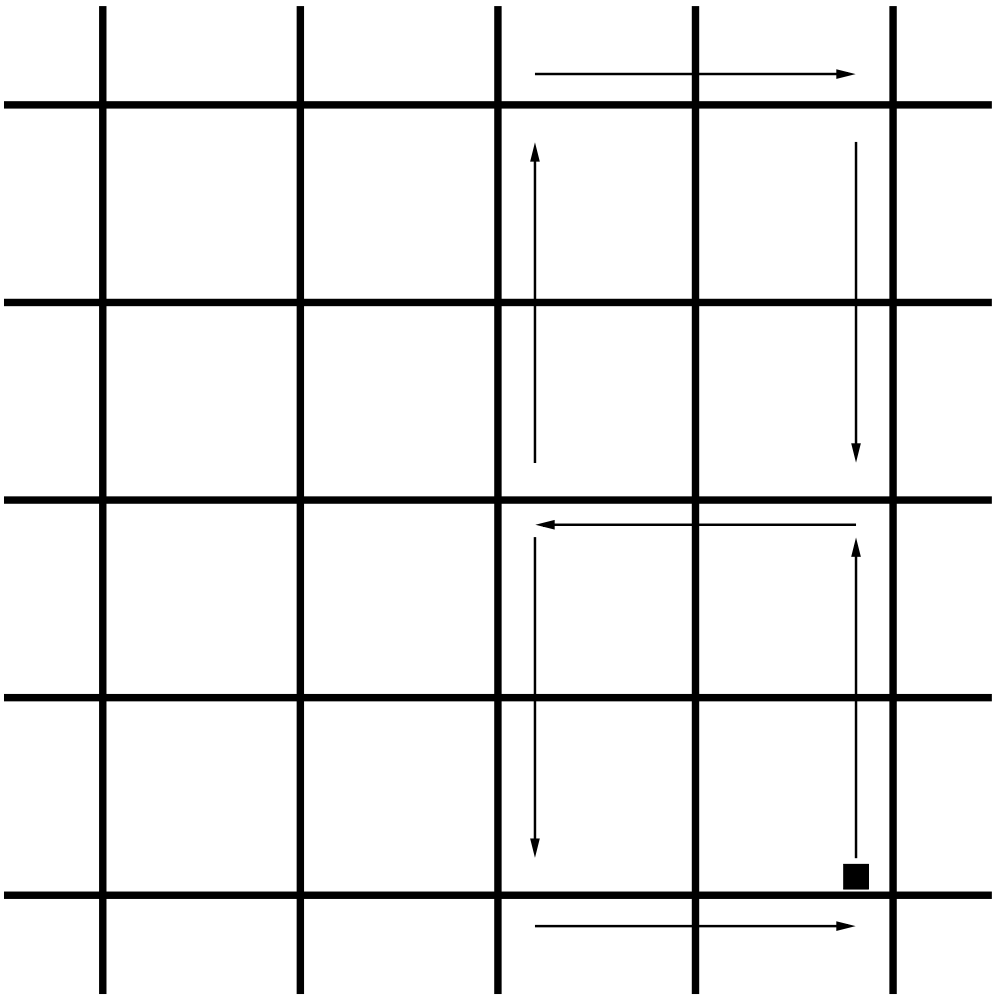

Figure 3.3: Field layout for test 1.

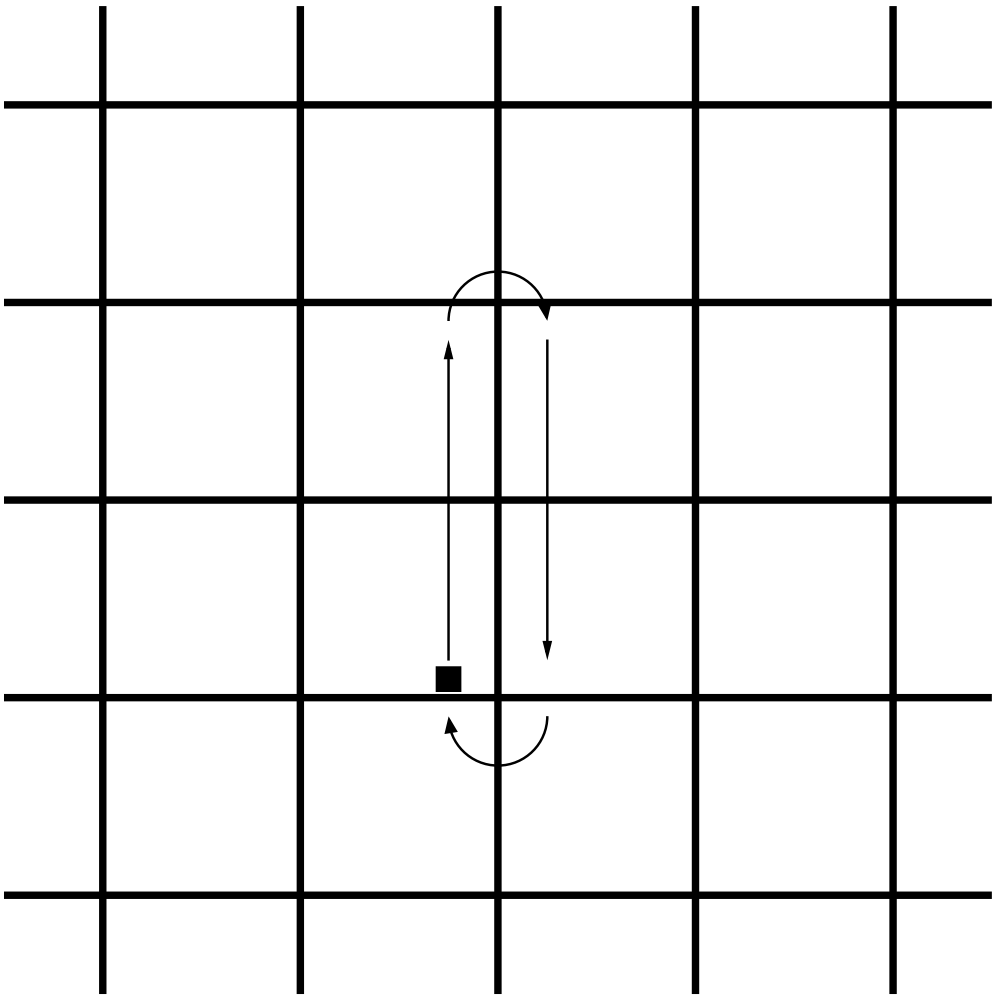

Figure 3.4: Field layout for test 2.

Part II

The Sokoban Solver

Chapter 1

A* In General

1.1 Pathfinding

This section describes the A* algorithm in general and is therefore not concentrated at pathfinding in Sokoban in particular, but rather on pathfinding in a broader sense. Later the modifications used to adapt A* to solve Sokoban are described, in the design section of this report.

The planning of the path that the robot must follow is calculated offline, meaning that the path is found in advance and not determined dynamically as the robot moves along. The pathfinder will define a path through a virtual world to solve a given set of constraints. Often the constraints is to find the shortest path from the current position of the agent to a specified target position. Pathfinding systems typically use pre-procecessed representations of the virtual world as their search space. The common scenario when pathfinding in computer games, is that the representation of the virtual world is made in form of a map.

1.2 Approaches to Pathfinding

There are many different approaches to pathfinding, but overall pathfinding can be divided into categories; undirected and directed. These approaches are briefly described in the following sections.

1.2.1 Undirected

This approach is analogous to a rat in a maze running around blindly trying to find a way out. The rat spends no time planning a way out and puts all its energy into moving around. Thus the rat might never find a way out and uses most of the time going down dead ends. Thus, a design based completely on this concept would not be useful in creating believable behaviour for an AI agent.

There are two main undirected approaches that improve efficiency. These are Breadth-first search and Depth-first respectively. Breadth-first search treats the virtual world as a large connected graph of nodes. It expands all nodes that are connected to the current node and then in turn expands all the nodes connected to these new nodes. Therefore if there is a path, breadth-first will find it. In addition if there are several paths it will return the shallowest solution first. The depth-first approach is opposite of breadth-first searching in that it looks at all the children of each node before it looks at the rest, thus creating a linear path to the goal. Only when the search hits a dead end does it go back and expand nodes at shallower levels. For problems that have many solutions the depth-first method is usually better as it has a good chance of finding a solution after exploring only a small portion of the search space.

1.2.2 Directed

Directed approaches to pathfinding all have one thing in common, that they do not go blindly through the maze. This means that using a directed strategy ensures a method of assessing the progress from all adjacent nodes before picking one of them. This is referred to as assessing the cost of getting to the adjacent node. Typically the cost in game maps is measured by the distance between the nodes. Most of the algorithms used will find a solution to the problem but not always the most efficient solution - that is the shortest path. The main strategies for directed pathfinding algorithms are:

- **Uniform cost search g(n)** modifies the search to always choose the lowest cost next node. This minimises the cost of the path so far, it is optimale and complete, but can be very inefficient.
- **Heuristic search h(n)** estimates the cost from the next node to the goal. This cuts the search cost considerably but it is neither optimal nor complete.

The two most commonly used algorithms for directed pathfinding in computer games; Dijkstra's algorithm and the A* algorithm use one or more of these strategies. Dijkstra's algorithm uses the uniform cost strategy to find the optimal path while the A* algorithm combines both strategies thereby minimizing the total path cost. Thus A* returns an optimal path and is generally much more efficient than Dijkstra's algorithm making it the backbone behind most pathfinding designs in computer games. Therefore we have chosen A* as the primary tool in the implementation for solving the Sokoban problem.

1.3 A* Pathfinding Algorithm

A* is a directed algorithm, meaning that is does not blindly search for a path - like a rat in a maze. Instead it assesses the best direction to explore, sometimes backtracking to try alternatives. This means that A* will not only find a path between two points, if a path exists, but it will find the shortest path if one exists and do so relatively fast.

To use A* in computer games, the game map has to be pre-processed before the A*-algorithm can work. This involves breaking the map into different points or locations, which are called nodes. These nodes are used to record the progress of the search. In addition of holding the map location each node has three other attributes. These are fitness, goal, and heuristic commonly known as f, g, and h respectively. Different values can be assigned to paths between the nodes. Typically these values would represent the distances between the nodes. The attributes g, h, and f are defined as follows:

- **g** is the cost of getting from the start node to the current node i.e. the sum of all the values in the path between the start and the current node.
- **h** stands for heuristic which is an estimated cost from the current node to the goal node - usually the straight line distance from this node to the goal.
- **f** is the sum of g and h and is the best estimate of the cost of the path going through the current node. In essence the lower value of f the more efficient the path.

The purpose of f, g, and h is to quantify how promising a path is up to the present node. Additionally A* maintains two lists, an Open and a Closed list. The Open list contains all the nodes in the map that have not been fully explored yet, whereas the Closed list consists of all the nodes that have been fully explored. A node is considered fully explored when the algorithm has looked at every node linked to it. Nodes therefore simply mark the state and progress of the search. Pseudocode for the general A* algorithm is given in algorithm 1.

The pseudocode outlined in algorithm 1 is the pathfinding method used in most computer games. Its simply tries to find af path from a given starting point to a specified target. Due to

Algorithm 1: A* pathfinding - normal version

```
1  Pre-conditions:
2  Both Open and Closed lists are empty.
3  Variables B and P are nodes.
4  Variables f, g, and h represents fitness, goal, and heuristic respectively.
5  Let P = starting point
6  Assign f, g, and h values to P.
7  Add P to the Open list. At this point P is the only node in the Open list.
8  while Open list is not empty do
9      Let B = the best node from the Open list (i.e. the node that has the lowest f-value).
10     if B is the goal node then
11         Quit - a path has been found.
12     end
13     else
14         Move the current node to the closed list and consider all of its neighbors.
15         for Each neighbor do
16             if This neighbor is in the closed list and the current g value is lower then
17                 Update the neighbor with the new, lower, g value.
18                 Change the neighbor's parent to the current node.
19             end
20             if This neighbor is in the Open list and the current g value is lower then
21                 Update the neighbor with the new, lower, g value.
22                 Change the neighbor's parent to the current node.
23             end
24             else
25                 Add the neighbor to the open list and set its g value.
26             end
27         end
28     end
29 end
```

the rules of Sokoban the general implementation of A* is not sufficient to solve the pathfinding problem. There are various reasons for this. One of them is is described in the following. The problem of solving the Sokoban puzzle can be broken down in two subproblems. The first subproblem is finding the best path from the current position of the man to a given diamond. The second subproblem is finding the best path that the man, while pushing the diamond, must follow to place the diamond onto a goal area.

At first the two problems seems to be similar, but due to the rules of Sokoban they are not. The difference is that the man, while not pushing a diamond, is allowed to move up, down, left, and right under the assumption that he is not moving through any obstacles by doing so. At the point when the man has reached a diamond, his maneuverability becomes more limited, because the man is only allowed to push the diamond. To overcome these problems we have made different modifications to the general A* algorithm. These modifications are described in section 2.1.

Chapter 2

Design and Implementation Strategies

When deciding on an implementation strategy, several factors in the design of the game was considered. First of all, there is really two elements in the game, that needs to be controlled. First there is the robot, and secondly the diamonds.

The diamonds are of course the whole basis for evaluating the puzzle, since the final goal is to move the diamonds from their starting positions, and to the goal fields. But on the other hand, it is the movements of the robot that is important in this project. Both in terms of that it is the robot we control, and also considering the fact that the robot should move in an optimal way.

After some deliberation and several design and test implementations, a general algorithm for solving the puzzle was agreed upon. The design separates the solving of the puzzle into two main areas:

1. finding the optimal route the diamonds should be moved
2. finding the optimal route the robot must follow, to ensure the first requirement.

The two requirements are co-dependant, since the optimal route for a diamond is of course dependant on where the robot is positioned, and where the robot can move to. And the the robot's route is dependant on the diamonds positions, since this dictates where the robot can move.

2.1 Design: Moving the Diamonds

The general strategy for finding a optimal route for the diamonds involves using a tree data structure for storing different states of the map, including the diamonds and the robot's position. For each node in the tree a complete "situation" is stored, and all possible next states are found. These are stored as children of the current node, and then processed later. Each node in the tree is visited in a search, until a solution is found. In addition to the tree, a list of situations already visited/investigated is kept, so that traversing identical sub-trees is avoided.

A "situation" is the data stored in a node. This includes the positions of all the diamonds, the robot, the cost of the node and the parent of the node. When a node is processed a Sokoban puzzle is populated with the information from the node. What this practically means, is that each node contains a complete Sokoban map with diamonds, goals, walls, the robot etc. This is used when finding new nodes to add as children. Looking at the map for the current node being processed, all possible derivatives for the map is found. In theory this means four new nodes for each diamond, since each diamond can be moved in four directions, but practically there are fewer nodes since some of the diamonds moves will be blocked by walls or other diamonds. Additionally the robot needs to have a clear path to the position behind the diamond, so that

the diamond can be pushed. For each of the new valid moves a new node is created and the diamond is moved to that new position. This means that a parent node has a number of children nodes, and each of these nodes have almost identical maps, except that in each map one of the diamonds are moved to one of its possible new positions, relative to the map in the parent node. Additionally the cost and the position of the robot is also updated to reflect the diamonds new positions.

2.1.1 Sokoban Solver: Main

The strategy for traversing the tree and adding new children is shown as pseudo-code in algorithm 2 Lines one to three is the precondition, and on line four the main construct of the solver is

Algorithm 2: Main section of the Sokoban Solver class

```
1  SET initialnode.map to initialmap
2  SET initialnode.parrent = null
3  ADD initialnode to opennodes
4  while opennodes not empty do
5  |  SET currentnode to first node in opennodes
6  |  REMOVE first node from opennodes
7  |  if currentnode.map is the_solution then
8  |  |  DO return the_solution
9  |  end
10 |  for each diamond in currentnode.map do
11 |  |  SET newValidPositions to CALL findNewValidPositionsForTheDiamond(diamond)
12 |  |  for each newValidPosition in newValidPositions do
13 |  |  |  SET tempmap = currentnode.map
14 |  |  |  CALL moveDiamond(tempmap, newValidPosition)
15 |  |  |  SET tempnode.map = tempmap
16 |  |  |  SET tempnode.parent = currentnode
17 |  |  |  ADD tempnode to opennodes
18 |  |  end
19 |  end
20 end
```

started. This while runs until either a solution is found, which is checked on line seven, or there is no more open nodes. If no solution if found, and there is no more open nodes, the puzzle has no solution that can be found by this algorithm.

Apart from the while loop the solver utilises two extra functions here. On line 14 `moveDiamond()` is used to update a map with the new position of the diamond. On line 11 a call to the `findNewValidPositions()` is important for the solver, since this call is responsible for detecting new positions the diamond can be moved to. This is shown in more detail in subsection 2.1.2

In the pseudo code shown in algorithm 2 some important parts are omitted for increased readability. The two most important parts are:

1. each node has a cost associated, and the open list is sorted accordingly
2. a list of nodes visited is stored in a closed list, and used to eliminate revisits of identical sub trees.

The cost of each node is calculated as with a classic A* algorithm. This means the cost reflects the distance travelled from the starting position, and a heuristic function calculates an additional cost. The nodes are then sorted accordingly to the cost, so that the cheapest node is at the first position in the open list.

The second omission is the closed list. When ever a new node is created it's added to a closed ass well as the open list. Before a node is added to the open list, it is checked if there is an identical node in the closed list. If that is the case, it is already in the open list, and there is no need to add the node again.

2.1.2 Sokoban Solver: Finding New Positions

When finding new valid positions for a diamond, the pseudo code in figure 3 is used. The

Algorithm 3: The findNewValidPositions pseudo code

```
1  for currentpostion.x - 1 to currentposition.x + 1 do
2  |  for currentpostion.y - 1 to currentposition.y + 1 do
3  |  |  if position not equals currentposition AND position not eqauls diagonal move then
4  |  |  |  if position.type equals type.GROUND then
5  |  |  |  |  robotPath = CALL getRobotPath(oppositeposition)
6  |  |  |  |  if robotPath not equals null then
7  |  |  |  |  |  ADD position AND robotPath to newnode
8  |  |  |  |  end
9  |  |  |  end
10 |  |  end
11 |  end
12 end
13 return listOfNewNodes
```

`findValidPosition()` method is called with the position of a diamond as argument. Then all positions neighbouring that position are investigated for validity. The condition on line three eliminates the starting position, which the diamond are moving from, as well as the illegal diagonal positions, which are by default not valid positions in a Sokoban puzzle.

The terrain of the position is then evaluate on line 4. The terrain must be valid for a diamond, which means not a wall and not another diamond, or just basically of type ground. The robot and the goals are all seen as type ground, since the diamond can indeed move to a field where one of those two are placed. The next check involves the robots position and its path. On line five it is checked if there is a path from the robot's current position, and to the position where it must go to push the diamond. It is important to recognise that it's not the path from the robots start position and to the diamond, or to the target field, but instead to the field that makes it possible to push the diamond. If the path is `null` the robot cannot move to the required "pushing position", and this of course invalidates the move of the diamond to the investigated position. This is checked on line six. If the robot's move is valid, the position is reported valid to the calling function *and* the path of the robots is also returned.

The reason for the path of the robot to be returned is that the path should be stored in the new node created for this update of the tree. Later, when a solution is found, it is possible to traverse up the tree, child to parent, and extract the path the robot has driven. This path is the exact path the physical robot must be instructed to take, to solve the complete puzzle from start to end.

2.1.3 Sokoban Solver: The Closed List

In a standard A* implementation the closed list is used to ensure that the path finding algorithm does not visit the same fields over and over again. That specific situation is not comparable to the Sokoban solver, which does not enforce a demand that a specific position can only evaluated once. Instead the Sokoban solver enforces that identical situations, where the exact position of the diamonds and the robot, is only investigated for possible derivative situations once.

Whenever a node is investigated for possible sub-nodes / children, all possible valid positions for the diamonds in the map are found. This can be used as an identifier for this particular situation. Say, if the robot after several moves, has completely switched the positions of two of the diamonds, and is still only capable of pushing the diamonds to the same positions as in the start situation. Then, the start and end situations are identical, and there is no reason to investigate the end situation for further derivatives. Instead, the path finder should return one situation up the tree, to the current situations parent node, and investigate that node for additional derivative situations.

2.1.4 Sokoban Solver: Cost Functions

The cost functions are used when calculating which cost a certain situation should have, and there by direct the search algorithm to hopefully take an appropriate route down the tree.

Two costs are used in our A* implementation. First the general cost of moving a diamond from field to field. This cost is always the SokobanSolver class, since moving a diamond from one field to another, always amount to the same work. There is only one type of terrain, if the invalid fields like diamonds and walls are disregarded.

The heuristic cost function in the solver is used to ensure that the diamonds in general move towards the goals. In this implementation this amounts to a function calculating the distance from each diamond and to the closest goal for that diamond. This ensures that the diamonds in general are moved towards the fields, and not away. This heuristic is enough to solve the Sokoban puzzle if only considering the diamonds.

An additional point of interest in Sokoban, and in this problem in particular, is the movement of the robot. To increase the effectiveness of the robot, an additional heuristic cost is added to a node, which calculates the distance between the robot and the nearest diamond. This is used to make the robot "prefer" pushing one diamond as long as possible, rather than changing back and forth between the diamonds that brings the whole puzzle closer to the solution. If this heuristic is not used, the robot will always push the diamond that is nearest to the final solution, possibly making robot move a diamond one field, then go to another diamond and move that diamond one field, and finally back to the first diamond. The most optimal is of course to move the first diamond two pushes, and *then* move to second diamond.

Chapter 3

Implementation of Sokoban Solver

The main part of the solver implementation is located in the SokobanSolver class, with some utility classes providing additional functionality. One exception is the SokobanMapReader class which both provides functions for reading a puzzle map from a file, and keeping it updated, but all contains the critical code for finding paths for the robot.

3.1 The SokobanSolver class

The design of the Sokoban solver is discussed in section 2, and gives a general overview of the functionality of the solver. In the next sections only technically and fuctionally important sections of the implementation are described. The complete code for the SokobanSolver class are found in the appendix.

3.1.1 The Open List

The A* algorithm used when solving the puzzle, dictates the use of a list for storing all the position, or in our case: maps, that needs to be investigated. The list should be sorted by cost, so that the cheapest position or node is at the first position.

In this implementation, where the open list contains nodes in our tree, it's the total cost of that node, that dictates its position in the list. The cost of a node is calculate with the cost functions discussed in subsection 2.1.4. Each node in the tree is an object of the type Node, and the class Node implements a compare method (implements the comparable interface). The open list is implemented as a PriorityQueue, which is a build-in Java queue, with automatic sorting. This ensures that the cheapest node is always at the head of the queue.

3.1.2 The Closed List

In addition to the open list, which is part of the A* design, our implementation utilises a closed list, which is described in section 2.1.3.

The closed list is implemented as a double hash map, where the other hash map contains the diamonds positions, while the inner contains the valid positions for the diamonds at that configuration.

To ensure correct hashing and recognition of situations where diamonds have exchanged place, all positions are placed in a sorted list. This list makes sure that the positions it contain are sorted in a specific way, so that if two diamonds have exchange positions, this is correctly perceived as an identical map, as when the diamonds were at their original positions.

When a children is added to a node exactly one diamond has moved. This new list of positions is added to the outer hash map, if it is not already added. For the new map in the child node, all valid positions for the diamonds are then found, and added to a sorted list, in the same way as with the diamonds.

Now it is checked to see if this sorted list of valid positions, are already held in the inner hash map. If that is the case, a exactly similar situation has already been found by an earlier search, and there is no reason to create new children in the tree for these positions. The path finder can close this sub-tree, and go back to the parent node, and try another child. If the list is not found in the inner hash map, it is added and new sub nodes are created for each of the valid positions.

3.2 The SokobanMapReader class

The SokobanMapReader class implements a parser using a buffered reader and the Scanner class to read and parse a Sokoban map in the format given in this course. It outputs a SokobanMapReader object which the SokobanSolver class can use for solving the puzzle.

In addition it contains the robot path finding implementation, used when querying the robot if there is a path from the current position and to a given position.

3.3 The SokobanSortedList class

This class is important for the functionality of the closed list implementation in the SokobanSolver class. It is an extension to the normal ArrayList class, overriding the standard `add()` method with a customised version. In the SokobanSortedList class the add method both adds the argument given to the list, but it also proceeds to sort that list, thus ensuring a specific order of its elements. In particular that the "smallest" positions are found first in the list, with increasing positions following.

Chapter 4

Robot Modifications

4.1 Testing on The Final Course

For the initial construction and testing of the robot, a generic test course was used. In order to adapt the robot for the course on which the competition was held, a number of tests was performed. The final course, which was used for the competition, consited of two thin melamine wood plates. The lines that made up the course was made of the same type tape as on the test course.

The aim was to test the same type of movements that were tested in chapter 3 on page 12. Since the competition course was more complex, than the test course, some of the tests differed somewhat, but the goal was the same. As described in 4.2 page 27 some modifications were made to the robot. After these were made, the robot performed the same as on the test course, with regard to accuracy and stability.

4.1.1 Observed Problems Prior to Modifications

As the final course consisted of two separate plates there was a intersection between the plates, and this caused several problems as it was not completely level. When crossing the intersection, the tin can (representing the diamond from the game) would often get caught in the tape edge at the intersection, which made the can fall over.

The tape marking the course would also rise up in a bump causing light to be reflected in a manner sufficiently different from the average condition, that it would cause wrong sensor readings. In several cases the robot would suddenly leave the course, for no apparent reason.

4.1.2 Method of Problem Solving

It was often not possible to determine why an error happened, as it was often difficult to recreate the events that led to the error. A number of tests with minor modifications to the robot were therefor necessary, amounting to a process of trial and error.

The object was to make the robot behaviours work at least as well on the competition course as on the test course. Also, as the robot had to participate in a timed competition, it was important that the robot was optimised to run at the highest possible speed under the given conditions.

4.2 Structural Modifications to the Robot.

4.2.1 Stabilising the Rig

In early tests the robot would "bob" the front end up and down when stopping after driving fast forward. One solution was to make the robot drive slower, but that would lessen the

chances of winning the competition. A better solution was to mechanically stop the bobbing from happening, which achieved by placing a number of support points immediately in front of the sensors. A positive side effect of this was that tin can stopped hanging in the tape at the intersection of the two plates.

4.2.2 Enclosing the Sensors

Initial tests were made in a room with relatively dark lighting conditions. When testing under other lighting conditions, it was made clear that some calibration of the sensor thresholds were necessary. It was not possible to find thresholds that was valid under all lighting conditions. Rather than make adaptive sensor adjustment a choice was made to control the conditions under which the sensors operated.

This was achieved by enclosing the sensors in a shroud, that blocks exterior light sources on three sides. On the fourth side an additional light source (LED bicycle front lamp) was placed. This gave stable light conditions, thus alleviating the need to change sensor settings. Also, this removed the problem of reflections from the tape at the intersection.

4.3 Modifications to Movement patterns

Modifications were required in order to make some of the movements, that were possible on the test course, possible on the competition course. On the test course the lines were whole, that is there were no gaps. On the competition course the lines were broken to simulate a wall in the sokoban game. Unfortunately the lines were in many cases so short that the sensors on the robot "missed" them when making a 180 degrees turn. This was resolved in two parts. First some of the shortest lines were made longer on the course. Second the behaviour of the 180 degree turn was modified. The modifications were made not to the method used for making the turn, but rather to the constants used in the methods. This was very much a case of trial and error, before the optimal values were found.

The other speed settings, forward, reverse and turn, were also optimised by trial and error. The aim was to get the robot to move as fast as possible and still run the course correctly.

4.4 Test of The Pathfinder Solution

The solution returned by the path finder is not the same format as the instructions the robot needs in order to move correctly. Additional instructions are required in order to place the cans etc. The solution from the path finder, was converted to movement instructions via a purpose written java program that pads the solution with the required extra instructions. This was then inserted manually into the NXC code, before compilation.

The calculated solution was tested simply by letting the robot run the course 6 - 7 times, and there were no errors. Simultaneously the robot was stress tested, by harassing the sensor conditions, e.g. by flashing lights or shaking the course. This revealed several of the errors that were addressed in the previous section. After the modifications were made, an additional 3 flawless passes were made, despite continued harassment of the sensor conditions.

Chapter 5

Improvements of the Pathfinder

5.1 Path-finding Improvements

This section discusses possible improvements to the path finding implementation used to solve the Sokoban puzzle. We have not implemented any of these strategies, but if the implementation should be improved or extended further some of the following considerations might be worth implementing.

5.2 Review Of Existing Sokoban implementations

The most notably project about Sokoban and general path finding algorithms that we have been able to find, is the Rolling Stone program and the accompanying papers describing the evolution of the program. The project started as an extension of a Ph.D. project in path finding and motion planning in computer games, and later turned into research project running over a period of 3 years. The authors of Rolling Stone describes progress of the program as: "The development effort equates to a full-time Ph.D. student, a part-time professor, one summer student, and valuable feedback from many people."

In the following subsections a number of strategies used in the Rolling Stone program are described. In the heading of each subsection the revision number of the program and the number of Sokanban problems the revision was able to solve is given. The goal of the Rolling Stone program was to solve as many problems as possible in a test suite of 90 Sokoban puzzles.

5.2.1 Minimum Matching Lower Bound (R0, 0 solved)

A* with a simple lower bound has no hope of finding a solution to any of the problems in the test suite. An obvious lower bound is the distance of each diamond to its closest goal, a Manhattan distance for Sokoban. However, the gap between the lower bound value and the actual solution length for any non-trivial Sokoban problem so large that the number of A* iterations, and thus their respective tree sizes, make solving these problems effectively impossible. By adding a lower bound to their implementation they were still not able to solve any of problems in the test suite.

To obtain a better admissible estimate for the distance of a diamond to a goal, a minimum-cost algorithm is used. The matching assigns each diamond to a goal and returns the total (minimum) distance of all diamonds to their goals. The minimum cost argumentation algorithm is $O(N^3)$, where N is the number of diamonds. During the search the lower bound only needs to be updated, which requires finding negative-cost cycles, and is therefore less expensive to compute. With the minimum matching lower bound strategy the program was still not able to solve any of the maps in the test suite.

5.2.2 Transposition Table (R1, 5 solved)

Even though the search spaces in Sokoban are generally graphs, most search algorithms treat them as trees. If a state can have several predecessors, this can lead to duplicate work. the search could revisit nodes and even entire sub-trees several times. These "transpositions" or cycles are detected using a transposition table in which useful information about previously visited nodes is stored. Before expanding a node, the transposition table is consulted, and if valid information is found, it is used to potentially curtail the search. Adding transposition tables allowed their program to solve 5 problems in the test suite.

5.2.3 Move Ordering (R2, 4 solved)

Instead of visiting successors of a position in an arbitrary order, one can try to look at "good" successors first. Move (or successor) ordering is not used in the best-first searches; the algorithm itself provides for a global ordering of the alternatives. In depth-first and breadth-first searches, move ordering can lead to efficiency gains because goals are found earlier (left in the tree) rather than later (right in the tree). For A*, ordering moves at interior nodes makes no difference to the search, except for the final iteration. Since the final iteration is aborted once a solution is found, finding a solution early in this iteration can significantly improve the performance. After adding move ordering to their program, they were only able to solve 4 of the test problems. According to their documentation, they categorise this as bad luck and explain that move ordering shows up as a valuable contribution after other features are added to the program.

5.2.4 Deadlock Table (R3, 5 solved)

In Sokoban it is possible to bring the puzzle in a deadlock state - that is a stat in which the puzzle becomes unsolvable. For instance pushing a diamond into a corner field that is not a goal area, makes every consecutive move irrelevant, because it is impossible for the man to bring the diamond back into the game without pulling it, which is an illegal operation in Sokoban. The implementation of Rolling Stone uses so called deadlock tables, where an off-line search is used to enumerate all possible diamond/wall placements in a $4x5$ region to determine if a deadlock is present. These results are stored in deadlock tables. During the A* search, the table is queried to see if the current move leads to a local deadlock.

In the A* search, before making a move, the program queries the deadlock table to see if the move would result in a known deadlock. If so, the move is not considered further. According to the designers of Rolling Stone, the branching factor is reduced by 20% by using deadlock tables. With deadlock tables the program where able to solve 5 of the test problems.

5.2.5 Tunnel Macros (R4, 6 solved)

The search algorithms discussed so far treat all moves equally. After making a move, all legal moves are considered as successors. These algorithms are therefore treating all moves as if they were unrelated. The method of macro moves is an attempt to group related atomic actions into higher level composite actions: macros.

A tunnel is the part of a maze where the manoeuvrability of the man is restricted to a width of one. Since there can be at most one diamond in a tunnel without creating an immediate deadlock, the remaining tunnel moves can be completed without loss of generality of optimality. If a tunnel is composed of articulation squares, the tunnel is called a one-way tunnel. Whenever the move generator creates a move into a one-way tunnel, the move is substituted with the macro pushing the diamond all the way through the tunnel. This eliminates all the inter-leavings with other legal moves.

Tunnel macros result in one additional problem being solved, bringing the count at a total of 6 solved problems from the test suite.

5.2.6 Goal Macros (R5, 17 solved)

Many of the Sokoban problems have all the goal squares grouped together in rooms. These goal areas are usually accessible through only a few square entrances. One can decompose the problem of solving a maze into:

- how to get each diamond to one of the entrances, and
- how to pack/arrange the diamonds into the goal areas.

Often these sub-goals can be solved independently, thus reducing the search space. This is achieved by defining a goal area, marking its entrances, and precomputing the order in which goal squares are filled without introducing deadlock in the goal area. During the search, if a move is generated that pushes a diamond onto the entrance square of a goal area, that move is replaced with a goal macro that generates a sequence of moves to push the diamond directly to an appropriate goal square. By introducing goal macros the program was able to solve 17 problems.

5.2.7 Goal Cuts (R6, 24 solved

The goal-macro heuristic eliminates all alternatives moves from consideration when a goal macro is present. The reason for this is that if it is possible to push diamond to its final destination, it will not affect other moves and they can be ignored. The same reasoning can be applied to the previous move: the move that pushed the diamond to the square from which it will be "macro"-pushed to the goal square. Goal cuts do exactly that recursively further up the tree: if a diamond is pushed to a goal with a goal macro at the end without interleaving other diamond pushes, all alternatives to pushing that diamond are deleted from the move list. With goal cuts they were able to solve 24 problems from the test suite.

5.2.8 Pattern Search (R7, 48 solved)

Pattern searches find patterns of diamonds that prove that the lower bound is in error. The errors could be small, improving the lower bound by as little as 2, or as much as ∞ in the case of a deadlock. All discovered patterns are saved and used throughout the search. If a pattern matches a subset of diamonds in a position, then the penalty associated with that pattern is added to the lower bound estimate for the position. In effect, the program learns lower bound penalty patterns and uses them to dynamically improve the lower bound function.

Sokoban pattern search two different mazes are used: the original maze, the data structure used by the A* search, and the test maze which will be used for the pattern searches. A pattern search iterates on the number of diamonds in the test maze. By definition, a deadlock is a configuration of diamonds such that not all of the diamonds can reach a goal. If a move $A - B$ is made, it might introduce a deadlock. If this deadlock was not present before the move, then the moved diamond, now on square B, must be part of that pattern. This is the initial diamond included into the test maze for the pattern search. A special version of A* tailored to be efficient at pattern searching, is called to solve the test maze. It either returns in failure (no solution, hence deadlock), or it finds a solution. In the latter case, the number of pushes in the solution may disagree with that determined by the minimum matching lower bound introduced in revision 1. If so the lower bound function is in error and can be improved.

By introducing pattern search into Rolling Stone, the designers were able to solve 48 of the 90 problems in the test suite. Pattern search was the strategy that gave most increase in the number of Sokoban puzzles the program was able to solve.

5.2.9 Relevance Cut (R8, 50 solved

Analysis of the trees built by an A* search quickly reveals that the search algorithm considers move sequences that no human would ever consider. Even completely unrelated moves are

tested in every legal combination - all in an effort to prove that there is no solution for the current threshold. How can a program mimic an "understanding" of relevance? The designers of Rolling Stone suggest that a reasonable approximation of relevance is influence. If two moves do not influence each other, then it is unlikely that they are relevant to each other. If a program had a good "sense" of influence, it could assume that in a given position all previous moves belong to a (unknown) plan of which a continuation can only be a move that is relevant - in the approximation, is influencing whatever was played previously. Relevance cuts eliminate moves from the search that appear to be irrelevant to the preceding sequence of moves. With relevance cuts implemented, Rolling Stone was able to solve 50 problems.

5.2.10 Overestimation (R9, 54 solved

To ensure optimality of solutions produced by A*-based algorithms, the heuristic has to be admissible. This limits the choice of knowledge that can be used. Even if some knowledge correlates well with the distance to the goal, but there is a chance that it overestimates, it cannot be used because the solution optimality would not be guaranteed. This shows that optimality has it price. Instead of fitting the heuristic distance to a solution h as closely as possible to the actual distance $h*$, we are restricted to creating a lower bound. The error of such a lower-bound function is often larger than a function that is allowed to occasionally overestimate. The larger the error of the lower-bound function, the less efficient the search. With overestimation they were able to solve 54 of the test problems.

5.2.11 Rapid Random Restart (R10, 57 solved

In the implementation of Rolling Stone a strategy called rapid random restart (RRR) is used. RRR assumes that by varying parameters to the solution algorithm (here search), it is possible to reduce the solution time dramatically. Therefore, instead of using all the available time with one parameter setting, RRR repeatedly aborts the search after a given effort limit and restarts it with different (random) parameters.

In Rolling Stone, RRR is used to interrupt an iteration and restart it with a different move ordering scheme. With RRR 57 of the 90 problems could be solved.

Part III

Appendixes

Appendix A

Code

A.1 NXC Code

```
#include "NXCDefs.h"

#define POWER          70
#define REVERSE        40
#define POWERTURN      60
#define POWERROTATE    60


#define TURN_PCT       20
#define LIGHT_THRSHOLD  45
#define TURN_ROTATION   40

#define MOTOR_RIGHT     OUT_A
#define MOTOR_LEFT      OUT_B
#define MOTOR_BOTH      OUT_AB

// Definitions of the different motions of the robot
#define CASE_S          0
#define CASE_L          1
#define CASE_R          2
#define CASE_C          3
#define CASE_B          4
#define CASE_TR         5
#define CASE_TL         6
#define CASE_STOP       -1

mutex right;
mutex left;

int LIGHT_LEFT = 0;
int LIGHT_RIGHT = 0;
int LIGHT_FRONT = 0;

int left_run = 1;
int right_run = 1;
int left_run_back = 0;
int right_run_back = 0;
int can_run = 0;

```

```
int run_speed=60;

int disp_cmd;


/* ↩
    ********************************************************************************* ↩

 * Manually defined command string for testing
 ********************************************************************************** ↩
    */
//int cmds[] = {CASE_S,CASE_B,CASE_B};

int cmds[] = ↩
    {1,2,0,2,3,2,3,5,2,0,3,2,1,0,1,0,0,1,0,0,0,3,1,3,5,2,0,0,0,2,0,0,2,0,2,1,0,3,2,1,0,2,2,0

//int cmds[] ={CASE_L, CASE_R, CASE_S, CASE_R, CASE_C, CASE_R, CASE_C, ↩
    CASE_TL, CASE_R, CASE_S, CASE_C, CASE_R, CASE_L, CASE_S, CASE_L, ↩
    CASE_S, CASE_S, CASE_L, CASE_S, CASE_S, CASE_S, CASE_C, CASE_L, ↩
    CASE_C, CASE_TL, CASE_R, CASE_S, CASE_S, CASE_S, CASE_R, CASE_S, ↩
    CASE_S, CASE_R, CASE_S, CASE_R, CASE_L, CASE_S, CASE_C, CASE_R, ↩
    CASE_L, CASE_S, CASE_R, CASE_R, CASE_S, CASE_C, CASE_R, CASE_S, ↩
    CASE_S, CASE_R, CASE_R, CASE_C, CASE_R, CASE_S, CASE_S, CASE_R, ↩
    CASE_L, CASE_S, CASE_L, CASE_L, CASE_S, CASE_S, CASE_S, CASE_C, ↩
    CASE_R, CASE_S, CASE_R, CASE_S, CASE_S, CASE_C, CASE_L, CASE_R, ↩
    CASE_R, CASE_S, CASE_C, CASE_TL, CASE_L, CASE_S, CASE_S, CASE_S, ↩
    CASE_L, CASE_S, CASE_L, CASE_S, CASE_S, CASE_S, CASE_C, CASE_TL, ↩
    CASE_S, CASE_S, CASE_S, CASE_R, CASE_S, CASE_L, CASE_S, CASE_L, ↩
    CASE_L, CASE_C, CASE_L, CASE_R, CASE_S, CASE_S, CASE_S, CASE_S, ↩
    CASE_R, CASE_S, CASE_R, CASE_S, CASE_S, CASE_S, CASE_L, CASE_R, ↩
    CASE_R, CASE_C, CASE_TR, CASE_S, CASE_S, CASE_L, CASE_L, CASE_S, ↩
    CASE_C, CASE_L, CASE_R, CASE_R, CASE_S, CASE_S, CASE_C, CASE_R, ↩
    CASE_L, CASE_S, CASE_L, CASE_S, CASE_S, CASE_L, CASE_S, CASE_S, ↩
    CASE_S, CASE_S, CASE_L, CASE_S, CASE_C, CASE_TL, CASE_S, CASE_R, ↩
    CASE_R, CASE_C, CASE_R, CASE_L, CASE_L, CASE_C, CASE_R, CASE_S, ↩
    CASE_R, CASE_R, CASE_C, CASE_STOP};
//run oneeighty
//int cmds[] = {CASE_S,CASE_S,CASE_TR};

//run in eights
//int cmds[] = {CASE_S,CASE_L,CASE_S,CASE_R,CASE_S,CASE_R,CASE_S,CASE_R, ↩
    CASE_S,CASE_R,CASE_S,CASE_L,CASE_S,CASE_L,CASE_S,CASE_L};

/* End of commans sting definitions */

int cmd_counter = -1;

/* ↩
    ********************************************************************************* ↩

 * In the following we define functions that functions that enables or ↩
    disables
 * behaviours. Note
 ********************************************************************************** ↩
    */

/*****************************************************************
 *
 * Routine     : disableRun
```

```
 * Parameters : None
 * Return     : nothing
 * Purpose    : Disables the autonomous forward movement routines.
 *              And stops the motors
 *
 ****************************************************************/
void disableRun() {
    Acquire(left);
    left_run = 0;
    Release(left);

    Acquire(right);
    //PlayTone(440,1000);
    right_run = 0;
    Release(right);

    Off(MOTOR_RIGHT);
    Off(MOTOR_LEFT);
}

/****************************************************************
 *
 * Routine    : enableRun
 * Parameters : int
 * Return     : nothing
 * Purpose    : enables the autonomous forward movement routines.
 *              The motors are not started here, this only enables
 *              the control task
 *
 ****************************************************************/
void enableRun(int power) {
    run_speed = power;
    left_run = 1;
    right_run = 1;
}

/****************************************************************
 *
 * Routine    : disableRunBack
 * Parameters : None
 * Return     : nothing
 * Purpose    : Disables the runback mode where we drive back one
 *              field after placing the can.
 *              The motors are stoped.
 *
 ****************************************************************/
void disableRunBack() {
    Acquire(left);
    left_run_back = 0;
    Release(left);

    Acquire(right);
    right_run_back = 0;
    Release(right);

    Off(MOTOR_RIGHT);
    Off(MOTOR_LEFT);
}
```

```
129 /*****************************************************************
 *
131  * Routine     : enableRunBack
 *  Parameters  : None
133  *  Return      : nothing
 *  Purpose     : Enables the behaviour where we move a field back
135  *                after having placed a can.
 *
137  *****************************************************************/
void enableRunBack() {
139      left_run_back = 1;
      right_run_back = 1;
141 }


143
/*****************************************************************
145  *
 *  Routine     : disableFrontSensor
147  *  Parameters  : None
 *  Return      : nothing
149  *  Purpose     : Stops the behaviout used while moving a can.
 *                While in this behaviour we use a aditional sensor
151  *                inorder to stop precisely when the can is on the
 *                cross of two intersecting lines.
153  *
 *****************************************************************/
155 void disableFrontSensor(){
      can_run = 0;
157 }

159 /*****************************************************************
 *
161  * Routine     : enableFrontSensor
 *  Parameters  : None
163  *  Return      : nothing
 *  Purpose     : Starts the behaviour that will ensure that the
165  *                robot places the can object accurately.
 *                While this behaviour is in effect we use an
167  *                aditional sensor inorder to stop precisely when
 *                the can is on the cross of two intersecting lines.
169  *
 *****************************************************************/
171 void enableFrontSensor(){
      can_run = 1;
173 }


175
/*****************************************************************
177  *
 *  Routine     : runStraight
179  *  Parameters  : int
 *  Return      : nothing
181  *  Purpose     : Drives forward a fixed amount, using the given
 *                power setting.
183  *
 *****************************************************************/
185 void runStraight(int power){
      //RotateMotor(OUT_AB,POWER,TURN_ROTATION);
187      RotateMotorEx(MOTOR_BOTH,power,TURN_ROTATION,0,true,false);
```

```
}

/*****************************************************************
 *
 * Routine     : runRight
 * Parameters  : int
 * Return      : nothing
 * Purpose     : Turns the robot to the right and drives up to
 *               the next junction
 *
 *****************************************************************/
void runRight(int power){
     runStraight(power);
     if(LIGHT_LEFT < LIGHT_THRSHOLD){
          //PlayTone(220,1000);
     }
     RotateMotor(MOTOR_LEFT,power,180);
     while(LIGHT_RIGHT > LIGHT_THRSHOLD){
          //OnFwd(MOTOR_LEFT,power);
          OnFwdSync(OUT_AB,power,100);
     }
}

/*****************************************************************
 *
 * Routine     : runRightRight
 * Parameters  : int
 * Return      : nothing
 * Purpose     : Turns the robot 180 deg. turning to the right and
 *               drives up to the next junction using the given
 *               power.
 *
 *****************************************************************/
void runRightRight(int power){
     runStraight(power);
     //if(LIGHT_LEFT < LIGHT_THRSHOLD){
          //PlayTone(220,1000);
     //}
     RotateMotor(MOTOR_LEFT,power,180);
     OnFwdSync(OUT_AB,power,100);
     while(LIGHT_RIGHT > LIGHT_THRSHOLD){
          //PlayTone(220,1000);
     }
     Off(OUT_AB);

     RotateMotor(MOTOR_LEFT,power,180);
     OnFwdSync(OUT_AB,power,100);
     while(LIGHT_RIGHT > LIGHT_THRSHOLD){
          //
     }
     Off(OUT_AB);
}

/*****************************************************************
 *
 * Routine     : runLeft
 * Parameters  : int
 * Return      : nothing
 * Purpose     : Turns the robot to the left and drives up to
```

```
247  *                the next junction
     *
249  ****************************************************************/
     void runLeft(int power){
251       runStraight(power);
          //if (LIGHT_LEFT < LIGHT_THRSHOLD){
253            //PlayTone(440,1000);
          //}
255       RotateMotor(MOTOR_RIGHT,power,180);
          while(LIGHT_LEFT > LIGHT_THRSHOLD){
257            //OnFwd(MOTOR_RIGHT,power);
               OnFwdSync(OUT_AB,power,-100);
259       }

261  }

263  /****************************************************************
     *
265  * Routine     : runLeftLeft
     * Parameters  : int
267  * Return      : nothing
     * Purpose     : Turns the robot 180 deg. turning to the left and
269  *               drives up to the next junction using the given
     *               power.
271  *
     ****************************************************************/
273  void runLeftLeft(int power){
          runStraight(power);
275       //if (LIGHT_RIGHT < LIGHT_THRSHOLD){
               //PlayTone(440,1000);
277       //}
          RotateMotor(MOTOR_RIGHT,power,180);
279       OnFwdSync(OUT_AB,power,-100);
          while(LIGHT_LEFT > LIGHT_THRSHOLD){
281            //OnFwd(MOTOR_RIGHT,power);
          }
283       Off(OUT_AB);

285       RotateMotor(MOTOR_RIGHT,power,180);
          OnFwdSync(OUT_AB,power,-100);
287       while(LIGHT_LEFT > LIGHT_THRSHOLD){
               //OnFwd(MOTOR_RIGHT,power);
289       }
          Off(OUT_AB);
291  }
     /*
293  void runLeftLeft(int power){
          runStraight(power);
295       if (LIGHT_LEFT < LIGHT_THRSHOLD){
               //PlayTone(440,1000);
297       }
          RotateMotor(MOTOR_RIGHT,power,180);
299       while(LIGHT_LEFT > LIGHT_THRSHOLD){
               //OnFwd(MOTOR_RIGHT,power);
301            OnFwdSync(OUT_AB,power,-100);
          }
303       RotateMotor(MOTOR_RIGHT,power,180);
          while(LIGHT_LEFT > LIGHT_THRSHOLD){
305            //OnFwd(MOTOR_RIGHT,power);
```

```
            OnFwdSync(OUT_AB, power, -100);
307     }
}
309 */

311 /*****************************************************************
 *
313  * Routine     : genNxtCmd
 * Parameters  : null
315  * Return      : int
 * Purpose     : Returns the next command to be executed
317  *               Stops after executing the whole list in cmds[]
 *
319  *****************************************************************/
int genNxtCmd(){
321     //cmd_counter = (cmd_counter + 1) % ArrayLen(cmds);

323     cmd_counter++;

325     if (cmd_counter > (ArrayLen(cmds)-1)) {
            cmd_counter--;
327         return CASE_STOP;
        }
329     else
            return cmds[cmd_counter];
331
        //return cmds[cmd_counter];
333 }

335 /*****************************************************************
 *
337  * Routine     : runInEights
 * Parameters  : int
339  * Return      : int
 * Purpose     : Test routine used while driving the robot in
341  *               figure eights n times
 *
343  *****************************************************************/
int runInEights(int n){
345      cmd_counter++;
         for(int i = 0; i < n; i++){
347             return cmds[cmd_counter];
         }
349 }

351 /*****************************************************************
 *
353  * Routine     : readSensors
 * Parameters  : null
355  * Return      : void
 * Purpose     : Coutinously poll the sensors and store their
357  *               values in global variables
 *
359  *****************************************************************/
task readSensors(){
361     while(true){
            LIGHT_RIGHT = Sensor(S1);
363         LIGHT_LEFT = Sensor(S2);
            LIGHT_FRONT = Sensor(S3);
```

```
365       }
}
367
/*****************************************************************
 *
 * Routine     : runningWithCan
 * Parameters  : null
 * Return      : int
 * Purpose     : Special control for moving with a can/jewel,
 *               enables the robot to acurately place a jewel/can
 *               on the intersection of two lines, by using the
 *               extra front sensor.
 *
 *****************************************************************/
task runningWithCan(){
      while(true){
            if(can_run){
                  if (LIGHT_FRONT < LIGHT_THRSHOLD){
                        //PlayTone(440,1000);
                        disableRun();
                        disableFrontSensor();

                        enableRunBack();
                  }
            }
      }
}


task motorRight(){
      while(true) {
            while(left_run){
                  Acquire(left);
                  //PlayTone(220,10);
                  if(LIGHT_RIGHT > LIGHT_THRSHOLD)
                        OnFwd(MOTOR_RIGHT,run_speed);
                  else {
                        Off(MOTOR_RIGHT);
                  }
                  Release(left);
            }
            // Off(MOTOR_RIGHT);
      }
}

task motorLeft(){
      while(true) {
            while(right_run){
                  Acquire(right);
                  //PlayTone(220,10);
                  if(LIGHT_LEFT > LIGHT_THRSHOLD)
                        OnFwd(MOTOR_LEFT,run_speed);
                  else {
                        Off(MOTOR_LEFT);
                  }
                  Release(right);
            }
            // Off(MOTOR_LEFT);
      }
```

```
}

task motorRightBack(){
    while(true) {
        while(left_run_back){
            Acquire(left);
            if(LIGHT_RIGHT > LIGHT_THRSHOLD)
                OnFwd(MOTOR_LEFT,-REVERSE);
            else {
                Off(MOTOR_LEFT);
            }
            Release(left);
        }
        // Off(MOTOR_RIGHT);
    }
}

task motorLeftBack(){
    while(true) {
        while(right_run_back){
            Acquire(right);
            if(LIGHT_LEFT > LIGHT_THRSHOLD)
                OnFwd(MOTOR_RIGHT,-REVERSE);
            else {
                Off(MOTOR_RIGHT);
            }
            Release(right);
        }
        // Off(MOTOR_LEFT);
    }

}

task controlDirection(){
    while(true){
        if((LIGHT_LEFT < LIGHT_THRSHOLD && LIGHT_RIGHT <  ↩
            LIGHT_THRSHOLD)){
            disableRun();
            disableRunBack();

            /* krims-krans der undersøger den ønskede retning */
            int cmd = genNxtCmd();
            //int cmd = runInEights(5);
            disp_cmd = cmd;
            //PlayTone(110,1000);

            switch(cmd){
                case CASE_S:
                    runStraight(POWER);
                    //PlayTone(440,1000);
                    enableRun(POWER);
                    break;
                case CASE_L:
                    runLeft(POWERTURN);
                    //PlayTone(440,1000);
                    enableRun(POWERTURN);
                    break;
                case CASE_R:
                    runRight(POWERTURN);
```

```
                        //PlayTone(110,1000);
483                     enableRun(POWERTURN);
                        break;
485                 case CASE_TR:
                        runRightRight(POWERROTATE);
487                     //PlayTone(110,1000);
                        enableRun(POWERROTATE);
489                     break;
                    case CASE_TL:
491                     runLeftLeft(POWERROTATE);
                        //PlayTone(110,1000);
493                     enableRun(POWERROTATE);
                        break;
495                 case CASE_C:
                        enableFrontSensor();
497                     runStraight(POWER);
                        enableRun(POWER);
499                     break;
                    case CASE_B:
501                     runStraight(-REVERSE);
                        enableRunBack();
503                     break;
                    case CASE_STOP:
505                     break;
                    default:
507                     break;
                }
509         }
    }
511 }

513 task displaySensors(){
    while(TRUE){
515         ClearScreen();
            TextOut(0, LCD_LINE1, "L:");
517         NumOut(15, LCD_LINE1, LIGHT_LEFT);
            TextOut(30, LCD_LINE1, "R:");
519         NumOut(45, LCD_LINE1, LIGHT_RIGHT);

521         TextOut(0, LCD_LINE2, "cmd counter: ");
            NumOut(70, LCD_LINE2, cmd_counter);
523
            TextOut(0, LCD_LINE3, "can_run?   : ");
525         NumOut(70, LCD_LINE3, can_run);

527         TextOut(60, LCD_LINE1, "F:");
            NumOut(75, LCD_LINE1, LIGHT_FRONT);
529
            TextOut(0, LCD_LINE5, "Left-run   : ");
531         NumOut(68, LCD_LINE5, left_run);

533         TextOut(0, LCD_LINE6, "Rigtht-run: ");
            NumOut(68, LCD_LINE6, right_run);
535
            TextOut(0, LCD_LINE7, "Case is now : ");
537         NumOut(78, LCD_LINE7, disp_cmd);

539         Wait(500);
    }
```

```
541 }

543

    task main(){
545     SetSensorLight(S1);
        SetSensorLight(S2);
547     SetSensorLight(S3);
        //SetSensorTouch(S4);
549
        //SetCustomSensorPercentFullScale(S1,50);
551
        Precedes(readSensors,motorRight,motorLeft,controlDirection, ↩
            runningWithCan,motorRightBack,motorLeftBack); // displaySensors
553 }
```

A.2 Java code

A.2.1 SokobanSolver class

```
1  package ai00.sokoban;
2
3  import java.util.ArrayList;
4  import java.util.Iterator;
5  import java.util.HashMap;
6  import java.util.PriorityQueue;
7
8  import ai00.sokoban.Node;
9  import ai00.sokoban.Position;
10 import ai00.sokoban.parser.SokobanParser;
11
12 /**
13  * $LastChangedRevision: 96 $
14  * $LastChangedDate: 2007-10-26 10:50:20 +0200 (fre, 26 okt 2007) $
15  * $LastChangedBy: gronbaek $
16  *
17  * SokobanSolver3 is the primary class in the Sokoban Solver program.
18  * It uses a SokbanMapReader map as basis, and then solves the sokoban puzzle
19  * by utilising a tree structure and an A* (A star) algorithm.
20  *
21  * The requirements for the map is specified in the SokobanMapReader class.
22  *
23  * Each node in the possible solution is processed in three steps.
24  * First step: a while loop runs through each node in the open list. A map is populated
25  * using the information from the node, and it's checked if the current node is
26  * the solution. If not, then check for vaild positions that the diamonds can be
27  * pushed from.
28  *
29  * Second step: Check for valid positions
30  *
31  * @author Bjorn Gronbaek
32  * @author Brian Horn
33  * @author Jon Kjaersgaard
34  *
35  * @version 3.0
36  */
37 public class SokobanSolver {
38   SokobanMapReader map;
39   boolean debug = false;
40   boolean showstate = true;
41
42   /** The set of nodes that have been searched through */
43   private HashMap<Integer, Object> closed = new HashMap<Integer, Object>();
44   private HashMap<Integer, HashMap<Integer, Object>> outerClosed = new HashMap<Integer, HashMap<Integer, Object>>();
45   private PriorityQueue<Node> open = new PriorityQueue<Node>();
46
47   /** The max search depth */
```

```
  int maxSearchDistance = 150;
49 int maxDepth = 0;

51 /**
   * Create a new SokobanSolver object for solving. The map must uphold ↩
     the specifications in the
53 * SokobanMapReader class.
   * If the debug parameter is set true lots of output will be printed ↩
     to system.out. This might take
55 * very long time.
   * If the showstates parameter is set true a small map with the ↩
     position of the diamonds is printed
57 * for each new node processed.
   *
59 * @param mapfile      the map to be solved.
   * @param debug      show debug information.
61 * @param showstates  show map for each node in the tree.
   */
63 public SokobanSolver(String mapfile,boolean debug, boolean showstates) ↩
     {
    map = new SokobanMapReader(mapfile);
65  map.createMap();
    this.debug = debug;
67  this.showstate = showstates;
  }
69

  /**
71 * The main method used when solving a map.
   *
73 * @return  an arraylist with positions the robot should go through.
   */
75 public ArrayList<ArrayList<Position>> solveMap(){
    System.out.println("Starting path solving");
77

    /* clear the open and closed list */
79  closed.clear();
    open.clear();
81

    /* This is our initial node. It has no parent, and is added to the ↩
       open list */
83  Node node = new Node(map.diamonds,map.man);
    node.parent = null;
85  node.depth = 0;
    node.cost = 0;
87  node.heuristic = 0;
    open.add(node);
89

    int numberOfNodes = 0;
91  /* While there is open nodes, continue search */
    while ((maxDepth < maxSearchDistance) && (open.size() != 0)) {
93    numberOfNodes++;

95    /* Get the next node in the open list, and remove it from the list ↩
         */
      Node currentNode = open.poll();
97

      /* Create a new map, with the state information from the node */
99    map.insertPositions(currentNode.diamonds,currentNode.man);
```

```
101         /* Debug information being printed below here */
            if(showstate){
103           System.out.println(map.man);
              System.out.println(map);
105         }

107         if(debug) System.out.print("Open nodes: "+open.size()+"\t Closed ↩
                nodes: "+closed.size()+"\t");
            if(debug) System.out.println("Depth: "+currentNode.depth);
109         if(debug) System.out.println(map.goals+" "+currentNode.diamonds);

111         if(!debug){
              if(numberOfNodes % 1000 == 0) System.out.println("Open nodes: "+ ↩
                  open.size()+"\t Closed nodes: "+closed.size()+"\t depth: "+ ↩
                  maxDepth);
113         }
            /* End of debug */
115
            /* check if we have found the solution */
117         if(map.goals.toString().equals(currentNode.diamonds.toString())) {
              System.out.println("Found a solution!!!");
119           System.out.println("Depth: "+currentNode.depth);
              System.out.print("Open nodes: "+open.size()+"\t Closed nodes: "+ ↩
                  closed.size()+"\t");
121
              return processSolution(currentNode);
123         }

125         /* If we haven't found the solution, proceed to check the new ↩
                valid positions for the new node */
            checkValidPositions(currentNode);
127
            /* Clear the map after processing a node, and start again, with a ↩
                new node */
129         map.removePositions();
          }
131
          /* If we get to here, something is wrong */
133       System.out.println("Done... if we haven't found a path, there's no ↩
              solution!");
          return null;
135     }

137     private ArrayList<ArrayList<Position>> processSolution(Node ↩
            currentNode) {
          System.out.println("SOLUTION HERE:");
139
          ArrayList<SokobanSortedList> diamondList = new ArrayList< ↩
              SokobanSortedList>();
141       ArrayList<ArrayList<Position>> pathList = new ArrayList<ArrayList< ↩
              Position>>();


143


145       diamondList.add(currentNode.diamonds);
          pathList.add(currentNode.path);
147       while(currentNode.parent != null){
            currentNode = currentNode.parent;
149         diamondList.add(currentNode.diamonds);
```

```java
            if(currentNode.path != null){
                pathList.add(currentNode.path);
            }
            //System.out.println(currentNode.path);
        }

        /*
        for (SokobanSortedList list : diamondList) {
            map.insertPositions(list, new Position(0,0));
            System.out.println(map);
            System.out.println("————————————————————————————————————————————————")
                ;
            map.removePositions();
        }
        */

        return pathList;
    }

    private void checkValidPositions(Node currentNode){

        /* For each diamond in the map, check for new valid positions */
        SokobanSortedList allValidPositions = new SokobanSortedList();
        PriorityQueue<Node> allNewNodes = new PriorityQueue<Node>();

        for(Position diamond: currentNode.diamonds){
            if(debug) System.out.println("Looking at diamond "+diamond.x+","+diamond.y);

            /* Get valid positions for the diamond */
            SokobanSortedList validPositions = getValidPositionsForDiamond(diamond);

            /* Create open nodes for the valid positions, for this diamond */
            PriorityQueue<Node> openNodes = createOpenNodes(validPositions, diamond, currentNode);

            allValidPositions.addAll(validPositions);
            allNewNodes.addAll(openNodes);
        }

        if(debug){
            System.out.println("All diamonds treated: "+allValidPositions.size()+" valid positions: "+allValidPositions);
            System.out.println("All diamonds treated: "+allNewNodes.size()+" new nodes.");
        }

        checkForClosedNodes(currentNode,allValidPositions,allNewNodes);
    }

    private void checkForClosedNodes(Node currentNode, SokobanSortedList allValidPositions, PriorityQueue<Node> allNewNodes){
        HashMap<Integer, Object> innerClosed;
        if(outerClosed.containsKey(new Integer(currentNode.diamonds.hashCode()))){
            if(debug){
```

```
        System.out.println("Diamonds ARE in closed list");
201     }
        innerClosed = outerClosed.get(new Integer(currentNode.diamonds. ↩
            hashCode()));
203     if(innerClosed.containsKey(new Integer(allValidPositions.hashCode ↩
            ())){
          if(debug){
205         System.out.println("New positions ARE in closed list");
          }
207     }
        else{
209       innerClosed.put(new Integer(allValidPositions.hashCode()),null);
          open.addAll(allNewNodes);
211       if(debug){
            System.out.println("New positions ARE NOT in closed list");
213         System.out.println("New size of open list are: "+open.size());
          }
215     }
      }
217   else{
        outerClosed.put((new Integer(currentNode.diamonds.hashCode())), ↩
            new HashMap<Integer, Object>());
219     open.addAll(allNewNodes);
        if(debug){
221       System.out.println("Diamonds ARE NOT in closed list");
          System.out.println("New size of open list are: "+open.size());
223     }
      }
225 }

227 private PriorityQueue<Node> createOpenNodes(SokobanSortedList ↩
        validPositions, Position diamond, Node oldNode) {
    PriorityQueue<Node> newNodes = new PriorityQueue<Node>();
229
    for(Position position: validPositions){
231   /* The new position of the man... the old position of the diamond ↩
          */
      Position newman = new Position(diamond.x,diamond.y);
233
      /* Movement of the diamond */
235   int deltaX = diamond.x - position.x;
      int deltaY = diamond.y - position.y;
237
      /* New list of diamods, created from the old list */
239   SokobanSortedList newdiamonds = new SokobanSortedList();

241   for(Position oldDiamond: oldNode.diamonds){
        /*
243     if(oldDiamond.x != newman.x && oldDiamond.y != newman.y){
          newdiamonds.add(oldDiamond);
245     }
        */
247     if(!oldDiamond.equals(newman)){
          newdiamonds.add(oldDiamond);
249     }
      }
251
      /* Remove the diamond at the position of the man */
253   //newdiamonds.remove(newman);
```

```
255       /* And add the moved diamond's new position */
          Position diamondPos = new Position(diamond.x + deltaX, diamond.y + ↩
              deltaY);
257
          newdiamonds.add(diamondPos);
259
          Node newnode = new Node(newdiamonds,newman);
261       newnode.setParent(oldNode);
          newnode.cost = calculateCost()+oldNode.cost;
263       newnode.heuristic = calculateHeuristic(diamondPos);
          newnode.path = map.findPath(position);
265       newnode.path.add(0,newman);

267       if(debug) System.out.println("Adding new open node: ("+diamondPos. ↩
              x+","+diamondPos.y+") cost: "+newnode.cost+" heuristic: "+ ↩
              newnode.heuristic);


269
          if(newnode.depth > maxDepth) maxDepth = newnode.depth;
271       newNodes.add(newnode);
        }
273     return newNodes;
      }
275
      private SokobanSortedList getValidPositionsForDiamond(Position pos) {
277     SokobanSortedList validPositions = new SokobanSortedList();

279     for (int x=-1;x<2;x++) {
          for (int y=-1;y<2;y++) {
281
            // check if tile is the same as current tile
283         if ((x == 0) && (y == 0)) {
              continue; //jump to next for
285         }

287         // check if tile is diagonal placed
            if ((x != 0) && (y != 0)) {
289           continue; //jump to next for
            }
291
            /* Check if the position is not a wall, if the opposite position ↩
                 is not a wall and finally
293          * if the man can reach the position
             */
295         //System.out.println((pos.x+x)+","+(pos.y+y)+": "+map.terrain[ ↩
                pos.x+x][pos.y+y]);
            if(map.terrain[pos.x+x][pos.y+y] == SokobanMapStatics.GROUND && ↩
                //the target position
297             map.terrain[pos.x-x][pos.y-y] == SokobanMapStatics.GROUND){ ↩
                      //the position the man must reach
              if(debug) System.out.println("Position: "+(pos.x+x)+","+(pos.y ↩
                  +y)+" is not a wall");
299           ArrayList<Position> path = map.findPath(new Position(pos.x+x, ↩
                  pos.y+y));
              if(path != null){        //is there a path for the man
301             if(debug) System.out.println("Position: "+(pos.x+x)+","+(pos ↩
                    .y+y)+" is reachablel");
```

```
            validPositions.add(pushDiamond(pos, new Position(pos.x+x,pos.y+y)));          //the new valid position
303         //System.out.println(path);
            if(debug) System.out.println("Adding position: "+(pos.x+x)+","+(pos.y+y));
305       }
        }
307   }
    }
309
    return validPositions;
311 }

313 Position pushDiamond(Position diamond, Position pushFrom){
    return new Position(diamond.x-(diamond.x-pushFrom.x),diamond.y-(diamond.y-pushFrom.y));
315 }

317 private float calculateCost() {
    /* Since the fields are always identical, just return the same value always */
319   return 10;
  }
321
  private float calculateHeuristic(Position diamondPos) {
323   int closestrange = Integer.MAX_VALUE;

325   Position current;
    Iterator<Position> it = map.goals.iterator();
327   while(it.hasNext()){
      current = it.next();
329     int distance = getDistance(diamondPos,current);
      if(distance < closestrange){
331     closestrange = distance;
      }
333   }

335   int manrange = getDistance(diamondPos, map.man) ;

337   return (closestrange + manrange) * 10;
  }
339



341
  private int getDistance(Position diamondPos, Position current) {
343   int deltaX = diamondPos.x - current.x;
    int deltaY = diamondPos.y - current.y;
345   return (int) Math.sqrt(Math.pow(deltaX,2)+Math.pow(deltaY,2));
  }
347
  /**
349  * @param args
   */
351 public static void main(String[] args) {
    boolean debug = Boolean.valueOf(args[1]);
353   boolean showstate = Boolean.valueOf(args[2]);
    SokobanSolver solver = new SokobanSolver(args[0],debug,showstate);
355   ArrayList<ArrayList<Position>> solution = solver.solveMap();
```

```
357     if(solution != null){
          for (ArrayList<Position> list : solution) {
359         System.out.println(list);
          }
361       System.out.println(new SokobanParser(solution).parse2simlator());
          String robotResult = new SokobanParser(solution).parse2robot();
363       System.out.println(robotResult);
          System.out.println(SokobanParser.cleanCanRuns(robotResult));
365     }
    }
367 }
```

A.2.2 SokobanMapReader class

```
1  package ai00.sokoban;

3  import java.io.BufferedReader;
   import java.io.FileNotFoundException;
5  import java.io.FileReader;
   import java.io.IOException;
7  import java.util.ArrayList;
   import java.util.List;
9  import java.util.PriorityQueue;
   import java.util.Scanner;
11
   /**
13  * $LastChangedRevision: 96 $
    * $LastChangedDate: 2007-10-26 10:50:20 +0200 (fre, 26 okt 2007) $
15  * $LastChangedBy: gronbaek $
    *
17  * @author Bjorn Gronbaek
    * @author Brian Horn
19  * @author Jon Kjaersgaard
    *
21  */
   public class SokobanMapReader {
23   private BufferedReader inputStream;
     private String filename;
25   private Scanner configScanner = null;

27   public int[][][] terrain;
     int width;
29   int height;
     public SokobanSortedList diamonds = new SokobanSortedList();
31   public SokobanSortedList goals = new SokobanSortedList();
     public Position man;
33
     public SokobanMapReader(String filename){
35     this.filename = filename;
     }
37
     private void readMap(String filename){
39     try {
         inputStream = new BufferedReader(new FileReader(filename));
41     } catch (FileNotFoundException e) {
         // TODO Auto-generated catch block
43       e.printStackTrace();
```

```
    }
  }

  public void insertPositions(SokobanSortedList diamonds, Position man){
    for (Position pos : diamonds) {
      terrain[pos.x][pos.y] = SokobanMapStatics.DIAMOND;
    }
    this.man = man;
  }

  public void removePositions(){
    for(int y =0; y<terrain[0].length; y++){
      for(int x=0;x<terrain.length; x++){
        if(terrain[x][y]==SokobanMapStatics.DIAMOND){
          terrain[x][y] = SokobanMapStatics.GROUND;
        }
      }
    }
    this.man = null;
  }

  public void createMap(){
    readMap(filename);
    System.out.println("Creating MAP");
    try {
      configScanner = new Scanner(inputStream.readLine());
      width = configScanner.nextInt();
      height = configScanner.nextInt();

      terrain = new int[width][height];

      System.out.println("New map is: "+width+"x"+height);

      StringBuffer sb;
      for(int y=0;y<height;y++){
        sb = new StringBuffer(inputStream.readLine());
        char tmp;
        for(int x=0; x<width;x++){
          if(x < sb.length()) tmp = sb.charAt(x);
          else tmp = 'E';
          switch (tmp) {
          case 'X':
            terrain[x][y] = SokobanMapStatics.WALL;
            break;
          case 'J':
            diamonds.add(new Position(x,y));
            break;
          case 'G':
            goals.add(new Position(x,y));
            break;
          case 'M':
            man = new Position(x,y);
            break;
          default:
            //map.setTerrain(j, i, SokobanMap.GROUND);
            break;
          }
        }
      }
```

```
103
        System.out.println("x - width = "+terrain.length);
105     System.out.println("y - height = "+terrain[0].length);

107   } catch (IOException e) {
        System.out.println("File Problem!!!!");
109     e.printStackTrace();
      }
111 }

113 public String toString(){
      String temp = "";
115   for (int y = 0; y < terrain[0].length; y++) {
        for (int x = 0; x < terrain.length; x++) {
117       if(terrain[x][y]==SokobanMapStatics.GROUND){
            temp+=". ";
119       }
          if(terrain[x][y]==SokobanMapStatics.DIAMOND){
121         temp+="D";
          }
123       if(terrain[x][y]==SokobanMapStatics.GOAL){
            temp+="G";
125       }
          if(terrain[x][y]==SokobanMapStatics.WALL){
127         temp+="W";
          }
129       if(terrain[x][y]==SokobanMapStatics.MAN){
            temp+="M";
131       }
        }
133     temp+="\n";
      }
135   return temp;
    }
137
    public void printFile(){
139   readMap(filename);
      String line;
141   try {
        line = inputStream.readLine();
143     while(line != null){
          System.out.println(line);
145       line = inputStream.readLine();
        }
147   } catch (IOException e) {
        System.out.println("Read error on file");
149     e.printStackTrace();
      }
151 }

153 /*
    public Set<PathPosition> findPath(Position position) {
155   Set<PathPosition> result = new TreeSet<PathPosition>();
      PathPosition orgPos = new PathPosition(man.x, man.y);
157   orgPos.setOriginPosition(orgPos);
      result.add(orgPos);
159   int pathLength = 100;
      terrain[man.x][man.y] = 100;
161   pathLength++;
```

```
    Iterator<PathPosition> it = result.iterator();
163 while(it.hasNext()) {
      PathPosition pos = it.next();
165   pathLength = terrain[pos.x][pos.y];
      if(pos.x == position.x && pos.y == position.y) {
167     for (int i = 0; i < terrain.length; i++) {
          for (int j = 0; j < terrain[0].length; j++) {
169         if(terrain[i][j] > 99){
              terrain[i][j]= SokobanMap2.GROUND;
171         }
          }
173     }
      }
175   if(terrain[pos.x-1][pos.y] == SokobanMap2.GROUND){
        PathPosition newPo = new PathPosition(pos.x-1,pos.y);
177     newPo.setOriginPosition(orgPos);
        terrain[pos.x-1][pos.y]=pathLength+1;
179     result.add(newPo);
      }
181   if(terrain[pos.x+1][pos.y] == SokobanMap2.GROUND){
        PathPosition newPo = new PathPosition(pos.x+1,pos.y);
183     newPo.setOriginPosition(orgPos);
        terrain[pos.x+1][pos.y]=pathLength+1;
185     result.add(newPo);
      }
187   if(terrain[pos.x][pos.y-1] == SokobanMap2.GROUND){
        PathPosition newPo = new PathPosition(pos.x,pos.y-1);
189     newPo.setOriginPosition(orgPos);
        terrain[pos.x][pos.y-1]=pathLength+1;
191     result.add(newPo);
      }
193   if(terrain[pos.x][pos.y+1] == SokobanMap2.GROUND){
        PathPosition newPo = new PathPosition(pos.x,pos.y+1);
195     newPo.setOriginPosition(orgPos);
        terrain[pos.x][pos.y+1]= pathLength+1;
197     result.add(newPo);
      }
199 }

201 for (int i = 0; i < terrain.length; i++) {
      for (int j = 0; j < terrain[0].length; j++) {
203     System.out.println(terrain[i][j]);
        if(terrain[i][j] > 99){
205       terrain[i][j] = SokobanMap2.GROUND;
          result.add(new PathPosition(i,j));
207     }
      }
209 }
    return result;
211 }
  */
213

215 public ArrayList<Position> findPath(Position targetPosition) {

217   //System.out.println("Finding path to: " + targetPosition+" from man ↩
        : "+man);
    PathPosition startPosition = new PathPosition(man.x,man.y);
```

```
219     startPosition.setOriginPosition(null);  //this is the start, there ↩
            is no parent
        int pathLength = 100;
221     terrain[man.x][man.y]= 100;
        pathLength++;
223     PriorityQueue<PathPosition> openPositions = new PriorityQueue< ↩
            PathPosition>();
        ArrayList<Position> path = new ArrayList<Position>();
225     openPositions.add(startPosition);

227     while(openPositions.size()> 0){
          PathPosition currentPosition = openPositions.poll();
229       pathLength = terrain[currentPosition.x][currentPosition.y];

231       if(currentPosition.x==targetPosition.x && currentPosition.y== ↩
              targetPosition.y){
            for (int i = 0; i < terrain.length; i++) {
233           for (int j = 0; j < terrain[0].length; j++) {
                if(terrain[i][j] > 99){
235               terrain[i][j]= SokobanMapStatics.GROUND;
                }
237           }
            }
239
            path.add((Position) currentPosition);
241         while(currentPosition.orgPosition != null){
              currentPosition = currentPosition.orgPosition;
243           path.add((Position) currentPosition);
            }
245
            //System.out.println(path);
247         return path;
          }
249


251       if(terrain[currentPosition.x-1][currentPosition.y] ==  ↩
              SokobanMapStatics.GROUND){
            PathPosition newPo = new PathPosition(currentPosition.x-1, ↩
                currentPosition.y);
253         newPo.setOriginPosition(currentPosition);
            terrain[currentPosition.x-1][currentPosition.y]=pathLength+1;
255         openPositions.add(newPo);
          }
257       if(terrain[currentPosition.x+1][currentPosition.y] ==  ↩
              SokobanMapStatics.GROUND){
            PathPosition newPo = new PathPosition(currentPosition.x+1, ↩
                currentPosition.y);
259         newPo.setOriginPosition(currentPosition);
            terrain[currentPosition.x+1][currentPosition.y]=pathLength+1;
261         openPositions.add(newPo);
          }
263       if(terrain[currentPosition.x][currentPosition.y-1] ==  ↩
              SokobanMapStatics.GROUND){
            PathPosition newPo = new PathPosition(currentPosition.x, ↩
                currentPosition.y-1);
265         newPo.setOriginPosition(currentPosition);
            terrain[currentPosition.x][currentPosition.y-1]=pathLength+1;
267         openPositions.add(newPo);
          }
```

```
269       if(terrain[currentPosition.x][currentPosition.y+1] ==  ↩
              SokobanMapStatics.GROUND){
            PathPosition newPo = new PathPosition(currentPosition.x, ↩
                currentPosition.y+1);
271         newPo.setOriginPosition(currentPosition);
            terrain[currentPosition.x][currentPosition.y+1]= pathLength+1;
273         openPositions.add(newPo);
          }
275     }

277     for (int i = 0; i < terrain.length; i++) {
          for (int j = 0; j < terrain[0].length; j++) {
279         if(terrain[i][j] > 99){
              terrain[i][j] = SokobanMapStatics.GROUND;
281         }
          }
283     }
        return null;
285   }

287


289
      public boolean isReachable(Position position) {
291     PathPosition orgPos = new PathPosition(man.x,man.y);
        orgPos.setOriginPosition(orgPos);
293     int pathLength = 100;
        terrain[man.x][man.y]= 100;
295     pathLength++;
        PriorityQueue<PathPosition> openPositions = new PriorityQueue< ↩
            PathPosition>();
297     openPositions.add(orgPos);

299     while(openPositions.size()>0){
          PathPosition pos = openPositions.poll();
301       pathLength = terrain[pos.x][pos.y];
          if(pos.x==position.x && pos.y==position.y){
303         for (int i = 0; i < terrain.length; i++) {
              for (int j = 0; j < terrain[0].length; j++) {
305             if(terrain[i][j] > 99){
                  terrain[i][j]= SokobanMapStatics.GROUND;
307             }
              }
309         }
            return true;
311       }
          if(terrain[pos.x-1][pos.y] == SokobanMapStatics.GROUND){
313         PathPosition newPo = new PathPosition(pos.x-1,pos.y);
            newPo.setOriginPosition(orgPos);
315         terrain[pos.x-1][pos.y]=pathLength+1;
            openPositions.add(newPo);
317       }
          if(terrain[pos.x+1][pos.y] == SokobanMapStatics.GROUND){
319         PathPosition newPo = new PathPosition(pos.x+1,pos.y);
            newPo.setOriginPosition(orgPos);
321         terrain[pos.x+1][pos.y]=pathLength+1;
            openPositions.add(newPo);
323       }
          if(terrain[pos.x][pos.y-1] == SokobanMapStatics.GROUND){
```

```
325         PathPosition newPo = new PathPosition(pos.x,pos.y-1);
            newPo.setOriginPosition(orgPos);
327         terrain[pos.x][pos.y-1]=pathLength+1;
            openPositions.add(newPo);
329       }
          if(terrain[pos.x][pos.y+1] == SokobanMapStatics.GROUND){
331         PathPosition newPo = new PathPosition(pos.x,pos.y+1);
            newPo.setOriginPosition(orgPos);
333         terrain[pos.x][pos.y+1]= pathLength+1;
            openPositions.add(newPo);
335       }
        }
337
        for (int i = 0; i < terrain.length; i++) {
339       for (int j = 0; j < terrain[0].length; j++) {
            if(terrain[i][j] > 99){
341           terrain[i][j] = SokobanMapStatics.GROUND;
            }
343       }
        }
345     return false;
      }
347

349   public void showPath(List<Position> positions) {
        for (Position position : positions) {
351       System.out.println(position);
        }
353   }

355   public void testIsReachable(Position p) {
        System.out.println("*******************************");
357     System.out.println("Initial position of robot is : " + new Position( ↩
            man.x, man.y));
        System.out.println("The robot tries to move to position :" + p);
359     System.out.println("Is this possible? " + this.isReachable(p));
      }
361
      public void testFindPath(Position p) {
363     System.out.println("*******************************");
        System.out.println("Initial position of robot is : " + new Position( ↩
            man.x, man.y));
365     System.out.println("The robot tries to move to position :" + p);
        System.out.println("The path for this is : ");
367     System.out.println(this.findPath(p));
        System.out.println("*******************************");
369   }

371   public void testPriorityQueue() {
        PriorityQueue<Position> t1 = new PriorityQueue<Position>();
373     PriorityQueue<Position> t2 = new PriorityQueue<Position>();
        PriorityQueue<Position> t3 = new PriorityQueue<Position>();
375     PriorityQueue<Position> pqueue = new PriorityQueue<Position>();
        t1.add(new Position(1,1));
377     t1.add(new Position(2,2));
        t1.add(new Position(3,3));
379     t2.add(new Position(4,4));
        t2.add(new Position(5,5));
381     t2.add(new Position(6,6));
```

```
    pqueue.addAll(t1);
    pqueue.addAll(t2);
    pqueue.addAll(t3);
    System.out.println(pqueue);
  }



  public static void main(String[] args) {
    SokobanMapReader mr = new SokobanMapReader("maps/testmap1.txt");
    mr.createMap();
    mr.printFile();
    //boolean dotest = false;
    boolean dotest = true;
    if(dotest) {
//      mr.testIsReachable(new Position(8,1));
      mr.testFindPath(new Position(1,1));
//      mr.testPriorityQueue();
    }
  }
}
```