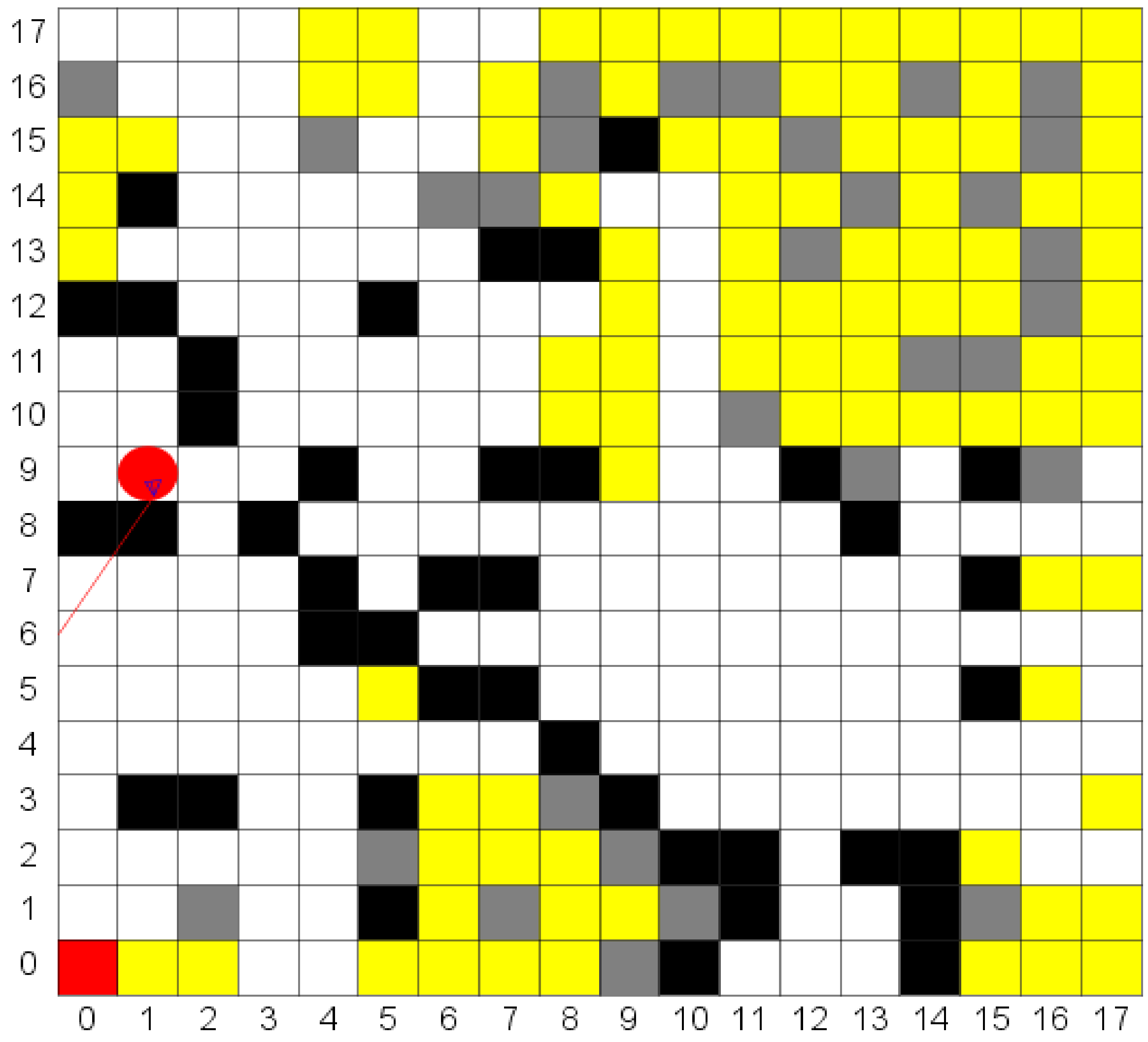
Design and Implementation of a Path Finding Robot Simulator



Roskilde University (RUC)

Department of Communication, Business and Information Technologies (CBIT)

Module 1 Project in Computer Science, Fall 2010

Project group: Truls Mosegaard, Chris Humphrey, Christian Henriksen

Supervisor: Keld Helsgaun (keld@ruc.dk)

Abstract (en)

This project documents the implementation of a path finding robot simulation. The application has been programmed in Java 6, and is not intended to include any user interaction. The application is primarily made up of a Robot, a Map and a Path Finding algorithm.

The application has been programmed with a theoretical base in Path Finding Algorithms as well as Behaviourally Based Robotics and Object Oriented Programming allows it to be transferred to a Lego NXT Brick allowing it to be used on a Lego MINDSTORMS™ robot.

The application has been programmed with a degree of success and in conclusion the A* search algorithm in conjunction with behavioural robotics has made it possible for a simulated robot to successfully explore and determine the fastest route between points in a static but unexplored environment.

Abstrakt (dk)

Dette projekt dokumenterer gennemførelsen af en path finder robot simulation. Ansøgningen er blevet programmeret i Java 6 og er ikke beregnet til at indeholde nogen brugerinteraktion. Ansøgningen består hovedsagelig af en robot, et kort og en path finder algoritme.

Applikationen er blevet programmeret med en teoretisk base i path finder Algoritmer samt Behaviourally Based Robotics og objektorienteret programmering gør det muligt at blive overført til en Lego NXT Brick således at den kan bruges på en Lego MINDSTORMS™ robot.

Applikationen er blevet programmeret med en vis succes og afslutningsvis er A * søgealgoritmen sammenholdt med adfærdsmæssige robotteknologi, har gjort det muligt for en simuleret robot der med succes kunne udforske og finde den hurtigste rute mellem punkter i et statisk men uudforsket miljø.

Foreword

This project investigates aspects of implementing a simulation of a path finding robot. The application has been developed with the intention of being implemented into a Lego Robot in the future and as such part of the program's structure tends to favour coding used for Lego NXT robotics.

The project originated from the idea that it might be possible to track a Lego MINDSTORMS™ robot by use of an over-head infrared sensor. The idea was to use a WiiRemote as produced by Nintendo and mount it on a ceiling with the infrared sensor looking down. The MINDSTORMS™ robot was to be fitted with an infrared transmitter that could be picked up and tracked by the WiiRemote's infrared sensor. By getting the WiiRemote, and the robot to communicate with a computer via Bluetooth, the idea was to implement an application that could pick up the Robot's position via the infrared sensor, and then receive information about the terrain at that point from the robot as it encounters obstacles with either its pressure or ultrasonic sensors, and then map the area as the robot was exploring it.

On the basis of this we proceeded to develop a Java application that maps an area as a simulated robot explores it, and then also determines the routes between points and attempts to guide the robot to goal points via the least taxing routes.

The application has been developed using the object oriented programming approach, so that the code could be inherited or extended by several different robots without having to go through many changes in order for it to be useful for different robots.

Contents

Introduction

This project revolves around implementing a path finding algorithm for use with a robot, as well as making it able to explore areas with a given goal to move towards. The robot will at first have no knowledge of the area it is entering and will gain knowledge of the area it is in by exploring.

The robot is given a goal and will use the A* algorithm as the path finding algorithm to find the shortest path between points. We will not make a real Lego MINDSTORMS™ robot, but a simulation of one that illustrates what we would like to be able to implement into an actual robot. This will consist of a grid that is visually represented as an unexplored area and the robot as well as its goal destination. With this simulation we hope to gain some insight into what it would take to make a path finding robot.

The reason we have made use of a simulation is two-fold. By use of a simulation we avoid technical difficulties that could arise from running the MINDSTORMS™ robot, the WiiRemote and a computer program at the same time. The overhead of solving these technical difficulties could overtake the actual project idea, and thus making use a simulation for pilot research is an effective way of avoiding spending too much time solving technical problems. We experienced this in the start of our project by spending way too many hours trying to solve technical problem such as getting the WiiRemote to communicate with a computer. Secondly by making use of a simulation we can also avoid having to allow for uncertainties as experienced in the real world. Eg. When a Lego MINDSTORMS™ robot moves forward the slightest difference in wheel movement can result in it getting sidetracked. Thus, if one wheel moves ever so slightly faster than the other or for some reason is hindered in moving, the robot will not be where a tracking application would believe it to be. Even within miniscule proportions it will make a robot with predetermined movement move in a wrong direction. Because of this, we have decided to make a simulation of the robot, so we can be sure that the path-finding program works, by eliminating the technical factors introduced when making the actual MINDSTORMS™ robot.

We use Java 6 for programming since it is the language that we feel gives us the best foundation for making the robot simulation because it is the language we as group feel most comfortable with, and for its object oriented qualities. As a project group, Java has been chosen as our programming as the Lego MINDSTORMS™ uses a Java based OS known as LeJOS.

The simulation is designed to be as object oriented as possible, because we want to keep in mind that the path finding program should be admissible to many different kinds of robots and not be specialized into a single type use.

This program should in the end be transferrable to a MINDSTORMS™ robot in theory. To follow up on this, we use some functions found in behavioural robotics as they relate to the MINDSTORMS™ robots that should in theory make a possible transfer of the application to a MINDSTORMS™ robot much easier as well as allowing thorough testing. Much of our application's design was conceived with this in mind.

Problem Analysis/Definition

As mentioned in the foreword the project evolved from an idea to simply track a live action robot using an over-head infrared sensor. The goal for this project as it stands is to develop a simulation of a robot and the overhead tracking, and then implement a data structure that can find routes between points, and guide the robot along these said routes.
The computer science problems our project seeks to address are found in the fields of Path Finding, Robotics, and Object Oriented Programming.

We are interested in Path Finding, specifically the A* path finding algorithm and its general application as a generic search problem solver. We would like to implement an A* that can be used by a MINDSTORMS™ robot. We will do this in conjunction with Behavioural Based Robotics because it is widely used for MINDSTORMS™ and it is an architecture that can be used for most robotic problems. Finally we would like to look at these concepts in terms of their Object Oriented qualities (if any)

In order for this goal to be realised there are requirement specifications we would like our application to realise. In short we can summaries the expectancies we have of the application as follows:

1. To initialise a simulated robot object which the application can track, and eventually guide.

2. To include a representation of the robot’s surrounding including possible obstacles as it may encounter in the real world.

3. To determine routes between points (more specifically a route between the robot and a goal point) and to communication this information to the robot and to guide the robot

4. To display a visual representation of the mapped area, as well as a representation of the robot’s movements on the map.

5. To implement behavioural based architecture into the robot.

Theory

The Application design and implementation is rooted in theory from three areas namely; Object Oriented Programming, Path Finding and Behavioural Robotics. We will also briefly present some theoretical aspects of software and unit testing.

Object Oriented Design and Implementation

There are many formal definitions, and sets of requirements for Object Oriented Programming or OOP, and not all literature seems to agree on the foundational elements. Despite this there are certain foundational elements represented in every description of OOP from reputable texts such as Big Java (Horstmann 2008), Applying UML and Patterns (Larman 2005) and Data Structures and Problem Solving (Weiss 2010) that are attributed as core elements of OOP and their consensual description can be given as follows; The foundation of OOP is the use data types that have structures and states that can be manipulated. These data types are known as objects.

The OOP paradigm is one that is defined by its use of these objects, and their interactions, to design and implement computer programs. This means that a program consists of a number of classes to represent the objects and these classes contain methods that allow them to interact with other specified classes. The object-oriented approach encourages the programmer to place data where it is not directly accessible by the rest of the program. The data is instead accessed by calling the methods contained within that object, and these methods also act as the intermediary parties between classes that allow for the retrieving or modifying that data from the classes. The programming construct which combines data with a set of methods for accessing and managing that data is called an object. (Wikipedia, 2010[1]) One of the goals of the OOP paradigm is to support reusable code, rather than requiring continuous reimplementation. OOP languages such as Java have tools that support code reuse. Of these tools our application makes use of *inheritance*, which allows one object's functions to extend to another.

[1] http://en.wikipedia.org/wiki/Object-oriented_programming, last accessed on the 18-12-2010

Path Finding

For our project we use the very common A* search algorithm that is widely used to find the shortest path between two nodes in a graph as long as the following conditions are satisfied:

1. There is a well defined starting point
2. There is a well defined goal point
3. It is possible to calculate a heuristic value

This search method is used for graph traversal were the shortest path between points has to be calculated and can therefore be used for wide variety of problems, for example is it often used in games for moving virtual objects about. It's also possible to solve puzzles and the like with this algorithm. The A* algorithm is derived from Dijsktra's algorithm by inserting a heuristic function but we will get back to that. The main difference between the methods is that Dijkstra goes breadth first, meaning it checks all possibilities, and A* uses best/depth first.

A* implements the best-first method, meaning it goes for the straightest path to the goal node. This is also the algorithms weakness, as it can wind up using a lot of time, searching in dead ends that appear to be the quickest path.

For each step in the algorithm a node is considered, all possible paths from this node are called its successors. For all successors the A* formula is used to determine how well this successor node does for the shortest path to the goal. The formula for the algorithm: $F = G + H$ is basically just counting the cost of the shortest path to the goal node. G is the immediate cost to the next node on the path adding the cumulative cost over the route taken, while H is the heuristic cost from the current point to the goal point. The first step has a cost to reach and if you continue to the next point in that line, the G value will have doubled. This plays a part in a node reaching problem that every A* can be troubled by. Calculating H is a guess, as it tries to measure the shortest path possible, and there may be walls and impassable objects in the way. The way of calculating H is what gives A* the

diversity , as different programs implement it, ranging from going to the right or left first or having an influence in the way the search ends within the goal node.
Finally F then gives us the estimated end cost of the route to the goal node. The scale of G and H has to be the same, as measuring these two numbers as different types would result in wrong paths. After these three values have been estimated, pointers from the successors to the starting node will be set, keeping track of which path actually led to this state of the node. The starting node is also referred to as the successor's parent, and the successors can be referred to as children.

To implement the A* search algorithm into a program where we can successfully integrate it with the functions we need, there won't be any major changes to the common use of the algorithm. To keep track of all nodes that have been visited and generated A* introduces 2 lists. These are called the open list and the closed list and they monitor which nodes are available and which have already been visited (Lester 2005[2]).

Nilsson (Nilsson 1971 p. 55-56) describes the steps in an A*-algorithm as follows (he calls it an ordered-search algorithm).

1. Put the start node s on a list called OPEN and compute F
2. If OPEN is empty, there is no solution.
3. Remove from the OPEN list the node with the smallest F value. And put it on the CLOSED list. We call this node c, for current. (chose nodes with equal f arbitrarily)
4. If c is the goal node, obtain the path by going back through the pointers. If c is not the goal node: continue.
5.
    a. Expand c, meaning find its successors c_i (if there are no successors, go to 2). Compute f for each successor.
    b. If c_i is not on either the OPEN or CLOSED list put it on the OPEN list and make pointer back to c.
    c. If c_i is already on either list and its f value is lower than the one in the list. Associate the node with the new f and make the pointer accordingly to c.
6. Go to (2)

This list can be seen as the flowchart below (Figure 1).

[2] http://www.policyalmanac.org/games/aStarTutorial.htm last accessed on 20-11-2010

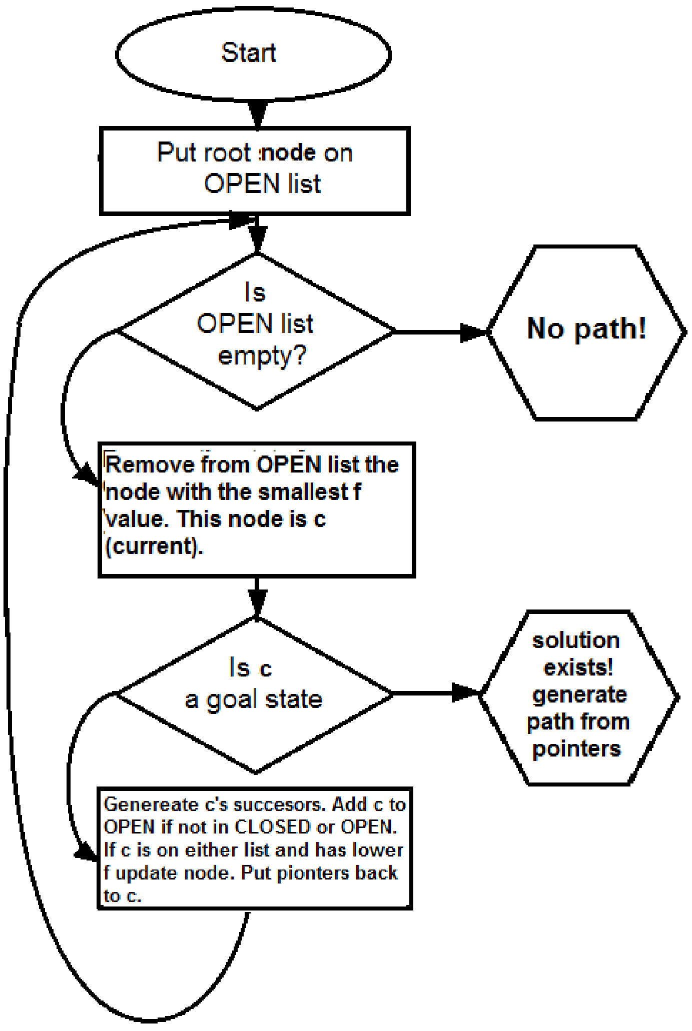

Start
Put root node on OPEN list
Is OPEN list empty?
No path!
Remove from OPEN list the node with the smallest f value. This node is c (current).
Is c a goal state
solution exists! generate path from pointers
Genereate c's succesors. Add c to OPEN if not in CLOSED or OPEN. If c is on either list and has lower f update node. Put pionters back to c.

Figure 1: A flowchart for A*. Rectangles are "do" statements, diamonds are "if" statements, and hexagons are "end" statements.

In our implementation of A* we work with a grid consisting of quadratic cells, each node being one of the cells in the grid. From each cell the path can be lead to any of the up to eight neighbouring cells, though some of the nodes are obstacles and can't be visited. Our robot cannot "cut corners" i.e. it cannot pass diagonally if there is an object in any of the 2 neighbouring cells that the path would guide the robot along. We have set the G cost to get from one cell to another horizontally or vertically to 10 and diagonally to 14. Calculating the heuristics h is done by using crows flight from the current cell to the goal cell multiplied with 10. A example of problem could be the one seen here (Figure 2).

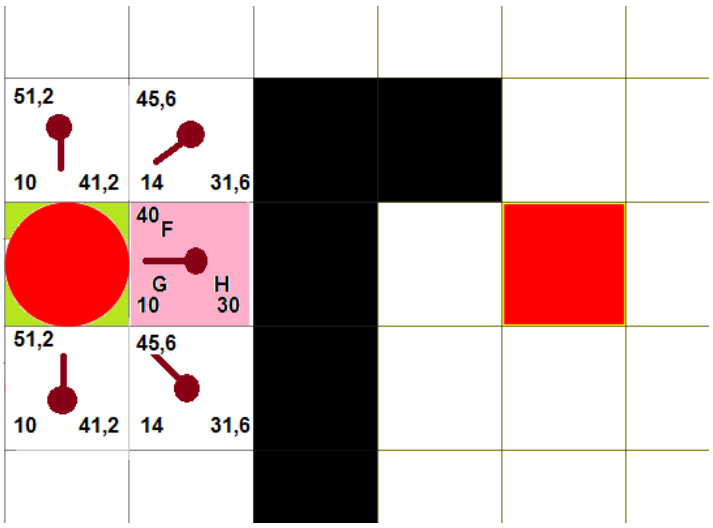


Figure 2: The first step of our A* implementation

The red dot is the starting cell for the calculation, the pink cell is the next step in the algorithm, the G and H values are listed in the left bottom and right bottom corner respectively, the F value in the top left corner. The pointers are represented by a purple circle with a line pointing towards the nodes parent. Green fields are on the closed list, while white cells with pointers are on the open list.

The above figure (Figure 2) shows the A* goes for the best node first. This means picking the one closest to the goal, being the one to the immediate right. It then checks the surroundings and submits the new nodes to the open and closed lists thus realizing that there is wall in the way. Checking the remaining options the next best way is to move up or down around the wall. But moving on from the current node and up or down will not be as cost-efficient as moving from the starting node going diagonal to avoid the wall. But for the current problem as the algorithm always chooses the best first, we will inevitably have to go to the right node first. So if the program isn't capable of knowing that the former route will be more efficient, the path will not be the shortest possible. It is for exactly that reason we need the lists of the available nodes presented in a way that show, that if the next node chosen for the path is already on the open list, then it should go to its parenting node, and move from that. The rest of the calculations can be seen (Figure 3) below.

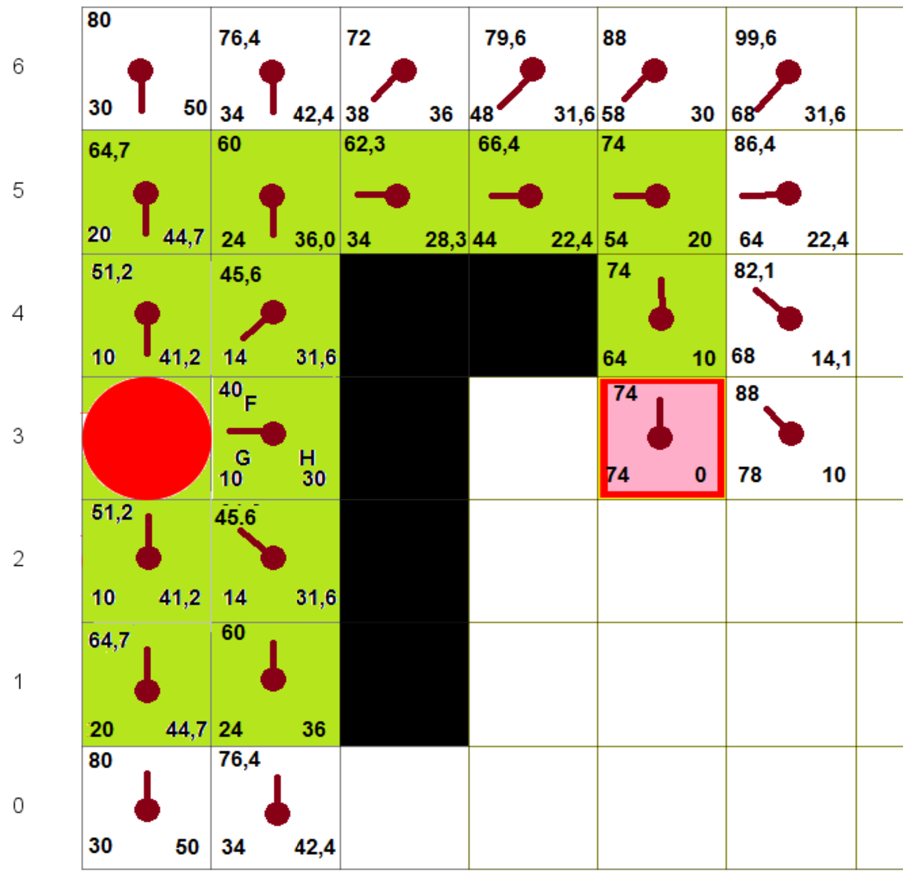


Figure 3: The full path found by our A* algorithm

Behavioural Based Robotics (BBR)

Behavioural based robotics is commonly used architecture for programming software of robots, which is also known as the subsumption architecture. It is a clever way of making the robot react appropriately under different circumstances.

The idea behind BBR is having a directive that tells the robot what to do depending on the situation. Each of these directives is called a primitive behaviour. The common way to describe how these behaviours works is to think of the robot as an insect, for example an ant. The ant will go look for food by following the markings of the other ants. If it has acquired some food it will return home. It will eat if hungry, but run or attack if threatened ignoring anything else.

The different parts of the robot that do the real work are called actuators; it could be the wheels or a robot arm and etc. A part of the robot that detects and “senses” the robots surroundings is called a sensor. A behavioural based-robot will work in the same way as the ant, its behaviours uses the sensory input, to tell it which actuators should do what. A simple behaviour consists of two things.

1. A control component that uses some sensory input to make commands for the actuators.
2. A trigger, a component of the robot that determines if a control component should act.

The control component is usually referred to as an arbiter or arbitrator. The arbitrator has a prioritized list of the behaviours that decides which behaviour should take command over an actuator, should several triggers tell multiple behaviours to take the control. There are different ways of doing the arbitration, but the simplest one and the one we use, is the single arbiter with a single fixed priority list of behaviours.

There are two kinds of behaviours the first being servo behaviours and the second being ballistic behaviours. Servo behaviours are usually behaviours that use some kind of feedback control loop; its trigger is always active and will only be inactive if a higher priority needs to use its actuators. It will react immediately to changes in the sensory input.

Ballistic behaviours, like the name implies, works in a way like a projectile – once triggered it will continue its path until completion. A ballistic behaviour should be used with care, as once its triggered the robot will complete its sequence and not take other behaviours into consideration (Jones 2004 p49-105).

BBR is a modular architecture for robot programming. A BBR system is rather easy to expand with extra behaviours, to an extent. A BBR robot also has the ability to trooper on even if some of its sensors and actuators have stopped working as other behaviours might still work properly. This process is called graceful degeneration.

Software testing

There are many different approaches and ideas to testing a specific piece of software. Most of the methods are made for larger programs on business level. Looking in the scope of our small implementation a lot of the concepts aren’t really feasible for us. We don’t have any testing team and our specifications are our own. Also considering the time span of our work we have been limited to use white box testing, and hereunder unit testing. Throughout our coding work we have both made use of static and dynamic testing, all these concepts will be briefly introduced here under. It is worth mentioning that software testing does not guarantee absence of errors (Lewis et. al. 2008 p 245-247)

White box testing as opposed black box testing is done with insight into the code implementation, data structures etc. Not having any testing team, doing all the testing ourselves and considering the final state of our program, makes white box the only option. During the development of our simulation software we used Static testing. That means inspecting and doing walkthroughs of the code to find faulty or unused code.

During creation of the program we also used dynamic testing i.e. tests where you actually execute programming code. This was mainly done using unit testing. A unit test verifies that individual units (like method calls or objects) work correctly on their own, outside of the main program cases (Horstmann 2008 s.102).

After finishing the program we used the unit testing to check for corner cases. Corner cases are specific situations that might occur when using some boundary parameters, i.e. values close to the boundaries of methods or special case values. We also used our unit tests to examine how well some of our classes scaled for use with bigger problems.

Program description

The program for this project can be broken down into three core aspects: The robot, the map and the classes associated with path finding.

These are the very foundational building blocks of the program as a whole. As a many factors are in play when making an actual MINDSTORMS™ robot we have constructed a simulation to make sure the path-finding works correctly.

The simulation we have created shows to some extend how a robot would behave in a real life situation. The goal of the simulation is to create an unknown territory with unseen obstacles and then have the robot explore it, while being able to determine the most effective route through the environment. The viewable part of the simulation consists roughly of only a map and a visual point symbolizing the robot while the real work is being done behind the scenes by the path finding algorithm.

Our program makes a random map for the robot to explore, so we acquire a diversifiable way to see how the A* algorithm reacts to with the randomly placed obstacles. The map size also differs for each initialization of the program so both short and longer paths can be observed. The visualization of the program is made into a quadratic grid. The positioning of obstacles is kept by their x and y values in relation to the grid and referred in terms of their row and column value. The A* star algorithm has been made with generic elements, making the program available for future uses, without having to be completely changed to fit other robots.

The grid consists of a random amount of cells that each have a state assigned to them. The state keeps track of the cell's current visibility and if the cell is passable or is to be considered an obstacle by the path finder. We have chosen to paint unseen cells yellow, and seen cells white. If there is an obstacle inside a cell that has not yet been seen it will be gray, and then it will be painted black once it has been discovered.

The robot's position is shown visually with a small arrow with a line down the centre, so is it possible to know the direction that the robot is facing and moving in. The cell the robot currently is inside is given a red marker for visibility showing when the robot goes from one cell to the next. A Lego robot has access to an ultrasonic sensor that can pick up information regarding the surroundings of the robot, so in our simulation we have a rotating scanner that checks the unknown area around the robot. With this scanner the path finding becomes much easier, as some obstacles will be found without the robot having to drive into them.

The path finding part of our program is simply given the robot's location as the start location and a goal location from the map (these locations are assigned randomly when the program is started) and then tries to find the shortest path whilst avoiding the obstacles. When the goal has been found, we want the robot to return home, which we have set to be the lower left corner of the grid at row and column coordinates (0, 0).

We will now describe the individual class as well as their individual functions and methods.

Implementation

In order to visually illustrate how our application's classes and their various methods interact we will make use of a Unified Modelling Language class diagram, also known simply as an UML class diagram. In the diagram each class or object is represented as a box and divided into three parts. The upper part holds the name of the class, the middle part contains the attributes of the class and the bottom part gives the methods or operations the class can take or undertake. The relationships are indicated by the use of connecting lines between these class boxes. Dashed lines are used to represent relationships of dependence, while solid lines are used for associations. To further differentiate between relationships these lines are topped with arrows that indicate different levels of dependence and association, as well as the relationship directions of these dependencies and associations. We only make use of four primary relationship types in this model, namely; aggregation, dependency, extension and implementation.

The first relationship, aggregation, is when one class "has" another class, and is also known as the "has-a" relationship(Larman 2005). In the case of our application, our robot class has behaviour objects. Thus the Robot class aggregates these various Behaviour objects such as `BackOffBehaviour` and `DriveForwardBehaviour`. This relations ship is indicated by the use of a solid line with an open diamond shaped arrow head pointing to the class that aggregates the other, so in our example the arrow will be pointing at the Robot class as illustrated in Figure 4.

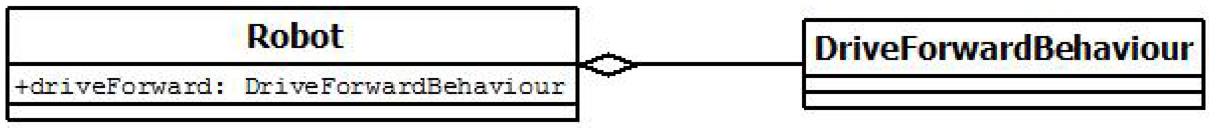


Figure 4: aggregation

The second relationship that we encounter often in our application is dependency. Dependency is a considered a weaker form of relationship than aggregation, which means that one class is dependent on another. This dependency exists if a class is passed as a local or parameter variable to another class or object. This form of relationship is indicated by a dashed line with an open arrow head facing the class that is being depended on (ie. The class being passed as a parameter) and indicates the direction of the relationship, as illustrated in Figure 5 below.

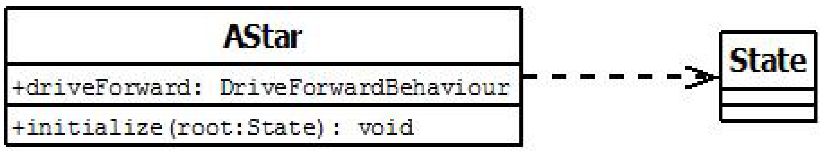

AStar
+driveForward: DriveForwardBehaviour
+initialize(root:State): void
State

Figure 5: dependency

The extension and implementation relationship are highly Object Oriented and indicate that one class uses another class and are also known as the “is-a” relationship. In the case of our application class `RoboState` extends class `State`, and as such `RoboState` “is-a” State. This extension relationship is indicated by a solid line with a closed, empty arrowhead pointing the class another class happens to be on instance of. This means that in our example Where `RoboState` “is-a” State the arrow will point from `RoboState` to State as in Figure 6

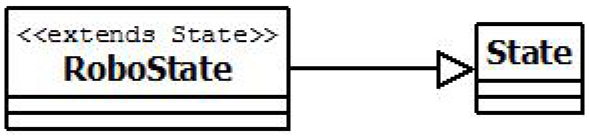


Figure 6: extends

Implementation is when a class implements an interface. This means that the class implementing the interface must implement ALL the methods defined in the interface class. Interfaces are used when classes of different types need to share methods. In our case we have 3 distinct Behaviour types that all need to share the same methods, and thus they implement the `Behavior` Interface. This implementation is denoted by a dashed line with a closed, empty arrow head pointing towards the interface being implemented. In the Figure 7 below we can see the notation for `DriveForwardBehaviour` which implements the `Behavior` interface.

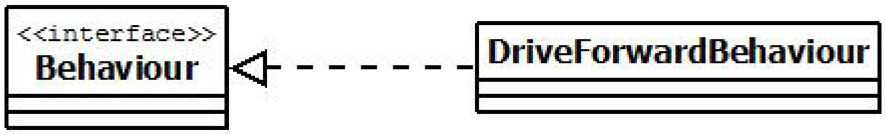


Figure 7: implements

In the case of our application the UML class diagram depicts the relationship between classes, and is not completely layered in order of the conceptual levels on which they are initiated and interact. The diagram starts with the object containing our main method, and then trickles down in order of how classes are initialised, however this is not accurate as some classes are initialised at the same level, but this is not indicated that way due to the multiplicity of relationships. The diagram rather groups the behaviours and associated classes on the left of the diagram, while classes used for mapping and path finding are situated on the right-hand-side of the diagram, please see Figure 8 on the next page. (Larman 2005)

UML class diagram for the Robot Simulator application

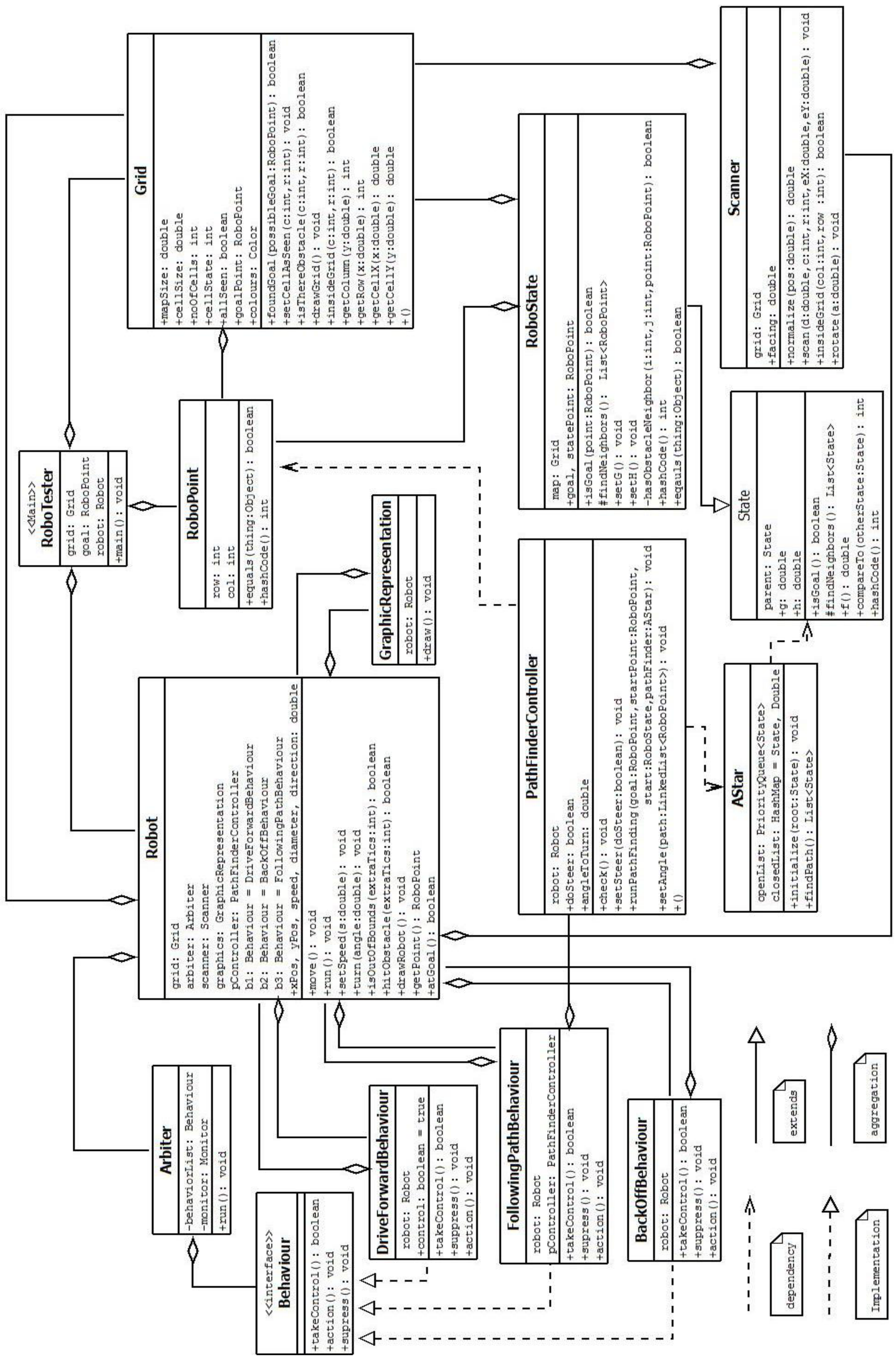

Grid
+mapSize: double
+cellSize: double
+noOfCells: int
+cellState: int
+allSeen: boolean
+goalPoint: RoboPoint
+colours: Color
+foundGoal(possibleGoal:RoboPoint): boolean
+setCellAsSeen(c:int,r:int): void
+isThereObstacle(c:int,r:int): boolean
+drawGrid(): void
+insideGrid(c:int,r:int): boolean
+getColumn(y:double): int
+getRow(x:double): int
+getCellX(x:double): double
+getCellY(y:double): double
+()
Scanner
grid: Grid
+facing: double
+normalize(pos:double): double
+scan(d:double,c:int,r:int,eX:double,eY:double): void
+insideGrid(col:int,row :int): boolean
+rotate(a:double): void
RoboState
map: Grid
+goal, statePoint: RoboPoint
+isGoal(point:RoboPoint): boolean
#findNeighbors(): List<RoboPoint>
+setG(): void
+setH(): void
-hasObstacleNeighbor(i:int,j:int,point:RoboPoint): boolean
+hashCode(): int
+equals(thing:Object): boolean
<<Main>>
RoboTester
grid: Grid
goal: RoboPoint
robot: Robot
+main(): void
RoboPoint
row: int
col: int
+equals(thing:Object): boolean
+hashCode(): int
GraphicRepresentation
robot: Robot
+draw(): void
PathFinderController
robot: Robot
+doSteer: boolean
+angleToTurn: double
+check(): void
+setSteer(doSteer:boolean): void
+runPathFinding(goal:RoboPoint,startPoint:RoboPoint,
start:RoboState,pathFinder:AStar): void
+setAngle(path:LinkedList<RoboPoint>): void
+()
State
parent: State
+g: double
+h: double
+isGoal(): boolean
#findNeighbors(): List<State>
+f(): double
+compareTo(otherState:State): int
+hashCode(): int
AStar
openList: PriorityQueue<State>
closedList: HashMap = State, Double
+initialize(root:State): void
+findPath(): List<State>
Robot
grid: Grid
arbiter: Arbiter
scanner: Scanner
graphics: GraphicRepresentation
pController: PathFinderController
b1: Behaviour = DriveForwardBehaviour
b2: Behaviour = BackOffBehaviour
b3: Behaviour = FollowingPathBehaviour
+xPos, yPos, speed, diameter, direction: double
+move(): void
+run(): void
+setSpeed(s:double): void
+turn(angle:double): void
+isOutOfBounds(extraTics:int): boolean
+hitObstacle(extraTics:int): boolean
+drawRobot(): void
+getPoint(): RoboPoint
+atGoal(): boolean
Arbiter
-behaviorList: Behaviour
-monitor: Monitor
+run(): void
<<interface>>
Behaviour
+takeControl(): boolean
+action(): void
+supress(): void
DriveForwardBehaviour
robot: Robot
+control: boolean = true
+takeControl(): boolean
+suppress(): void
+action(): void
FollowingPathBehaviour
robot: Robot
pController: PathFinderController
+takeControl(): boolean
+supress(): void
+action(): void
BackOffBehaviour
robot: Robot
+takeControl(): boolean
+suppress(): void
+action(): void
extends
aggregation
dependency
Implementation

Figure 8: UML class diagram

The Robot Class

The `Robot` class constitutes one of the core aspects of our application. It defines the robot object in terms of its attributes and methods. The Robot's attributes include its x and y coordinates, the direction it is facing, the speed it travels at, as well as other objects or classes that the robot is reliant on, such as a `Grid` object, the `Scanner` object, or the various behaviours.

The Robot class contains various methods necessary for the elementary functioning of the robot, as well as a method that represents the robot graphically by mean of the `StdDraw` class.

The class contains getter and setter methods of varying complexity. In order for the robot to move the next point is calculated by use of trigonometric functions on the angle that the robot is facing. By finding the cosine of the angle that the robot is facing, multiplying it with the robot's speed and then adding it to the x value of its current point we get the x -coordinate of the point the robot will travel to next. Similarly the sine of the angle of the direction that the robot is facing is taken, and once again multiplied by the robot's speed and then added to the current y-coordinate in order to get the y -coordinate for the next point. This is illustrated in Figure 9 below

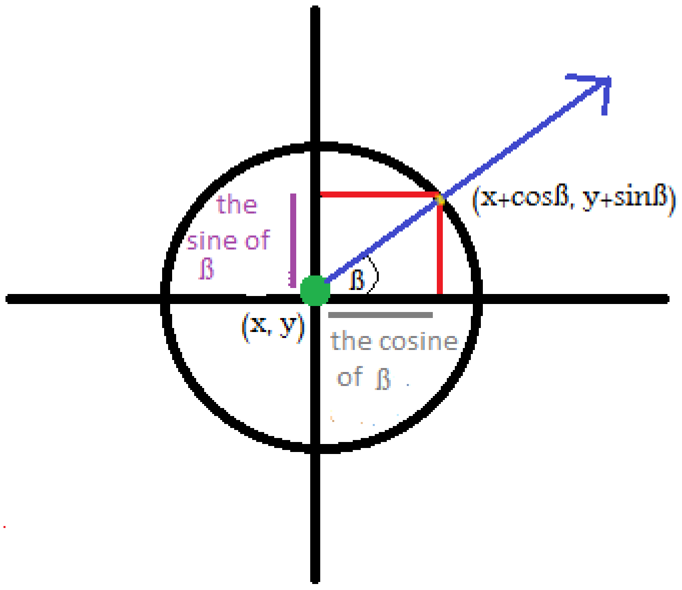


Figure 9: Figure calculating the coordinates of the next move

The robot object is dependent on other objects to the extent of aggregation, and the `run()` method runs the robot by starting the `Arbirter`'s thread and doing he following while the program runs (also see code fragment below):

1. Calls the scan method from class `Scanner` to retrieve information about the area it is exploring in terms of the obstacles that are present as well as the grid cells that have already been explored.
2. The robot is then moved by a call to its `move()` method
3. The turret of the scanner is rotated.
4. The `PathFinderController` then checks to establish if the robot has reached its goal position
5. As long as the robot is within the bounds of the grid, the cells it has explored in this iteration are set as having been 'seen'.
6. The `draw()` method from the `GraphicRepresentation` class is then called, which displays a visual representation of the grid and the states of the cells contained within the grid, as well as the robot's position and direction, the direction of the scanner, and the grid cell within which the robot is currently moving.

```
public void run() {
        arbiter.start();
        pController.runPathFinding();
        while (true) {
            scanner.scan(direction, grid.getRow(xPos),
                         grid.getRow(yPos),
                         scanner.normalize(xPos),
                         scanner.normalize(yPos));
            move();
            scanner.rotate(Math.toRadians(10));
            pController.check();
            if (!isOutOfBounds(0)) {
                grid.setCellAsSeen(grid.getRow(xPos),
                                   grid.getRow(yPos));
            }

            graphics.draw();

            try {
                Thread.sleep(SLEEP);
            }
            catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
```

The `Robot` class also contain an `isOutOfBounds()` methods determines if the robot will be exiting the map area. The method returns true if the robot will be exiting the map area on the NEXT move. This test is done by calculating the x and y coordinates of the next move, and then testing to see if the coordinates fall outside of the map bounds. If the robot's x coordinate is greater than the map's largest value for x, or smaller than the maps smallest value for x (which in our case is 0) then the robot is out of the map bounds and the same goes for its position along the y axis and it is up to its behaviours to steer it back into the map bounds.
If δ represents the maps dimension and r represent the robot we could express it as follows: $r(x, y)$ is within bounds while $0 \leq x \leq \delta$ and $0 \leq y \leq \delta$.

The `Robot` class also contains a method `hitObstacle()` to determine if the robot has collided with an obstacle that has not yet been picked up by the scanner. Once again the methods returns true if the robot will be colliding with the scanner on the NEXT move. This method is highly dependent on the grid class and passes statements that require information about the state of cells contained within the grid class

The `Robot` class also contains a `draw()` method which draws an instance of the robot as a triangle, along with a line which represents the scanner. On top of this a cell-size "tracking-dot" is drawn in red to indicate which grid cell the robot is currently in.

The Behaviours and the Arbiter

Our robot simulation is, as mentioned, supposed to resemble a Lego robot. We are using the behaviour based robotics architecture described in the chapter on BBR at page 13**.** To program Lego robots usually a programming language called leJOS is used (http://lejos.sourceforge.net/). leJOS is an Open Source Java-based operating system for the LEGO MINDSTORMS™. leJOS contains a subsumption package, that has an interface for behaviours and a arbitrator class. We have imported these directly into our program. We have renamed the arbitrator class to arbiter. We have been unable to get the arbiter to work correctly without extending the class as a thread. We are not entirely sure why this problem emerges but we think it has something to do with the way the java OS works on the MINDSTORMS™ NXT brick, compared to a pc.

The `Behavior` Interface consists of 3 methods, `takeControl()`, `action()` and `suppress()`. The take control method is a Boolean that returns true when the behaviour wants to seize control of the robot. The `action()` method details what the robot should do when the behaviour becomes active. The `suppress()` method should stop the current behaviour and could also execute some code; it seems that it doesn't work though. The `suppress()` method is not used in our current program, but could be utilized in multiple ways for working with a real robot when implementing other behaviours.

The `Arbiter` class takes an array of behaviours as an argument. In its main loop it makes sure that at any given point it keeps track of which behaviour that has control by using the behaviours placement in the array as its priority. The `Arbiter` makes sure not to call the action function several times if the same behaviour keeps having control during the loop.

Our behavioural architecture is as shown on Figure 10 below.

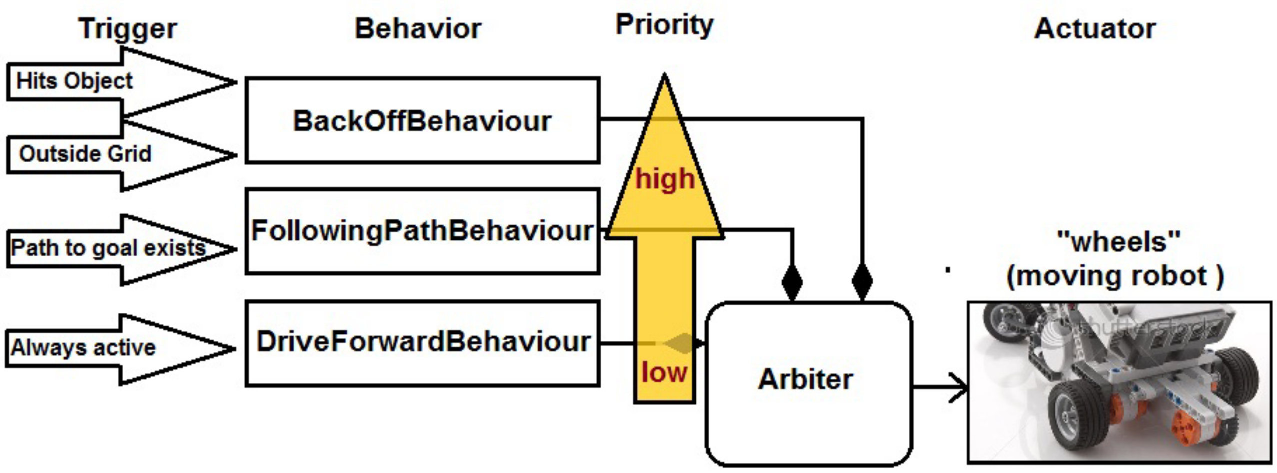


Figure 10: All behaviours control the movement of the robot. The behaviours' priority is represented by their vertical placement and their trigger by the arrows to the left.

The highest priority behaviour is the `BackOffBehavior`. It is a ballistic behaviour that is triggered if the robot moves into an occupied square or out of the map. It sets the robots speed to be negative and then sleeps the arbiter for a couple of tics while the robot backtracks, then turns the robot between 90 and 270 degrees.

The 2nd behaviour in the arbiters' priority is the `FollowPathBehavior`. This will take control if a flag is set by its helping class `PathFinderController` that there is a path to follow (see below). The controller tells the robot which direction it should keep from its current point to stay on the specific path it is following.

If no other behaviours are in command the `driveForward` takes control. It is always active and has no trigger as such. It just sets the robots speed to a positive number.

Affiliated with the BBR part of our program is the `PathFinderController` class. It uses our A* implementation and the seen part of the grid and its goal point to calculate the path for the robot to follow. Ideally the controller should recalculate the path whenever something changes in the seen part of the grid. As it is now the A* will be called for every tick, this causes some redundancies. For big grids this might be a problem (**refer p xx**). The angle to turn is then calculated from the first two `State` objects, from entries in the path array. We calculated the angle using the formula for an angle between 2 vectors: $\cos^{-1}\frac{a \cdot b}{||a||\,||b||}$. Where **a** is the vector between the two points and **b** is the vector from the first point and the point one unit in the x direction (see Figure 11). This actual formula becomes $s\cos^{-1}(\frac{x_2-x_1}{\sqrt{(x_2-x_1)^2+(y_2-y_1)^2}})$, where (x_1, y_1) is the first point and (x_2, y_2) is the 2nd point in the path. As arcos cosine is defined in the domain $[0:\pi]$ we use a variable **s** to set the angle accordingly, whether vector **a** is above or below the horizontal line.

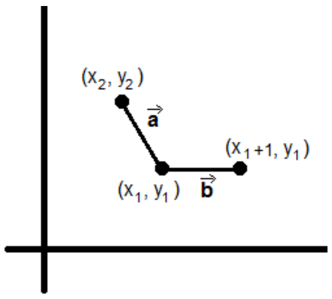


Figure 11: The vector used for calculating the angle to turn to.

If there is no path or the robot already is at the goal point, the controller will set the `doSteer` to false and to stop the `follow path` behaviour.

The grid class

This class represents the map that a Lego robot might stumble upon and be asked to find its way through. Within this class we initialize an area map that has been divided into cells and each cell is initialised as `unseen` and `empty`, even those that contain obstacles. The grid is basically made up of columns and rows based on x and y position.

The grid's primary attributes are `mapSize`, `cellSize`, `noOfCells`, `cellState` and a `RoboPoint`. These relate to respectively; the size of the grid, the size of each cell within the grid, the number of cells in a row or column, whether cell has been seen or not or if it contains obstacles and row and column coordinate.

To make all the cells `unseen` at first, we make a nested for loop, that iterates through each column and row to assign the cells their state.
When the cells are initiated there is a Boolean function that asks if the obstacles should start as `seen` or `unseen`. This was made for testing purposes and is also very useful to see the pathfinder getting the correct path immediately and moving to the goal.

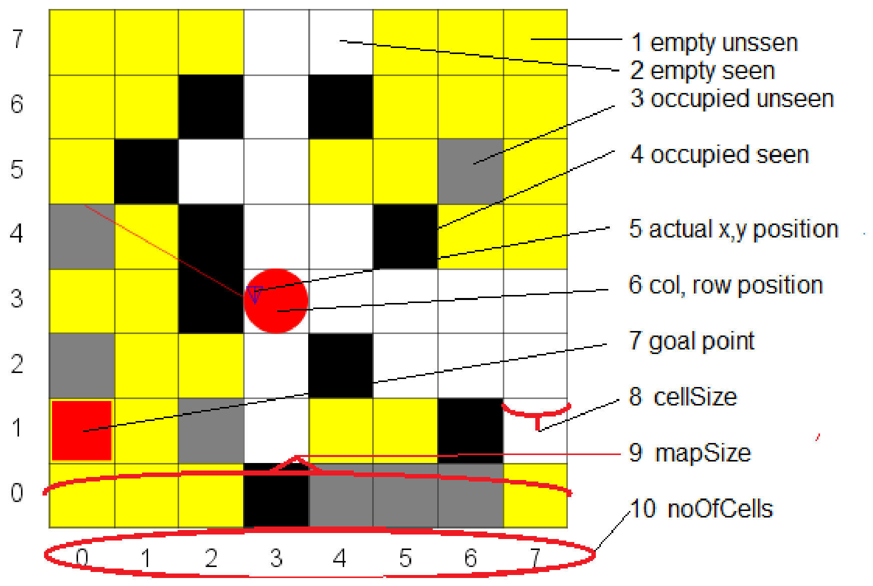


Figure 12: an explanation of how the variables are used in the Grid class

The `grid` class contains a lot of getter methods which return variables for the other classes in the application to use, such as the number of cells, their location and dimensions.
Methods can also be called to the `grid` class to check whether there is an obstacle inside a cell or not or if a cell has been encountered and needs to be stated as seen - this returns the `cellState`.

The next method `insideGrid()` takes it to a little bigger scale, as it checks whether the column and row given is within the grid or not. This method is used extensively throughout the program, to make sure nothing goes out of bounds. By using a simple Boolean we make sure that a call to a point outside the given grid, won't result in an "arrayOutOfBounds" error.

The last thing we do in the `grid` class is drawing the map. This method is called from our `GraphicsRepresentation` class and the drawing function is called in our borrowed class `StdDraw`. The map gets drawn each time the run method in the `robot` class is called which gives us a lot of drawings of the map. We draw our randomized map by using a nested for loop that goes though each column and row just like how we initialized our cell's states earlier, giving each cell the appropriate colour according to their given `state` before we move on to make the lines that separate the cells and writing out the individual column and row numbers.

The Scanner Class

The `scanner` class is supposed to represent an ultrasonic sensor on a Lego robot. The ultrasonic sensor uses ultrasound to detect objects in front of it and returns the distance to the closest object. By knowing the direction of the scanner it is then possible to identify objects at a distance. In our simulation the sensor is represented by our `scanner` class. We represent the scan by a following a line in the given direction and marking every cell passed by the line as `seen` until an occupied field is hit.

We encounter a problem, when we want to check which cells the `scanner` line goes through. Going through the entire grid would be to tedious and time consuming, so we had to find another way to get the output regarding which cells have been seen

The algorithm works as follows: We scale and translate the above problem to a problem just concerning one cell. Keeping track of which column and row this cell represents, we find the exit point of the line out of the cell in the given direction. This exit point is then used as the start point in the next cell as we call the method recursively.

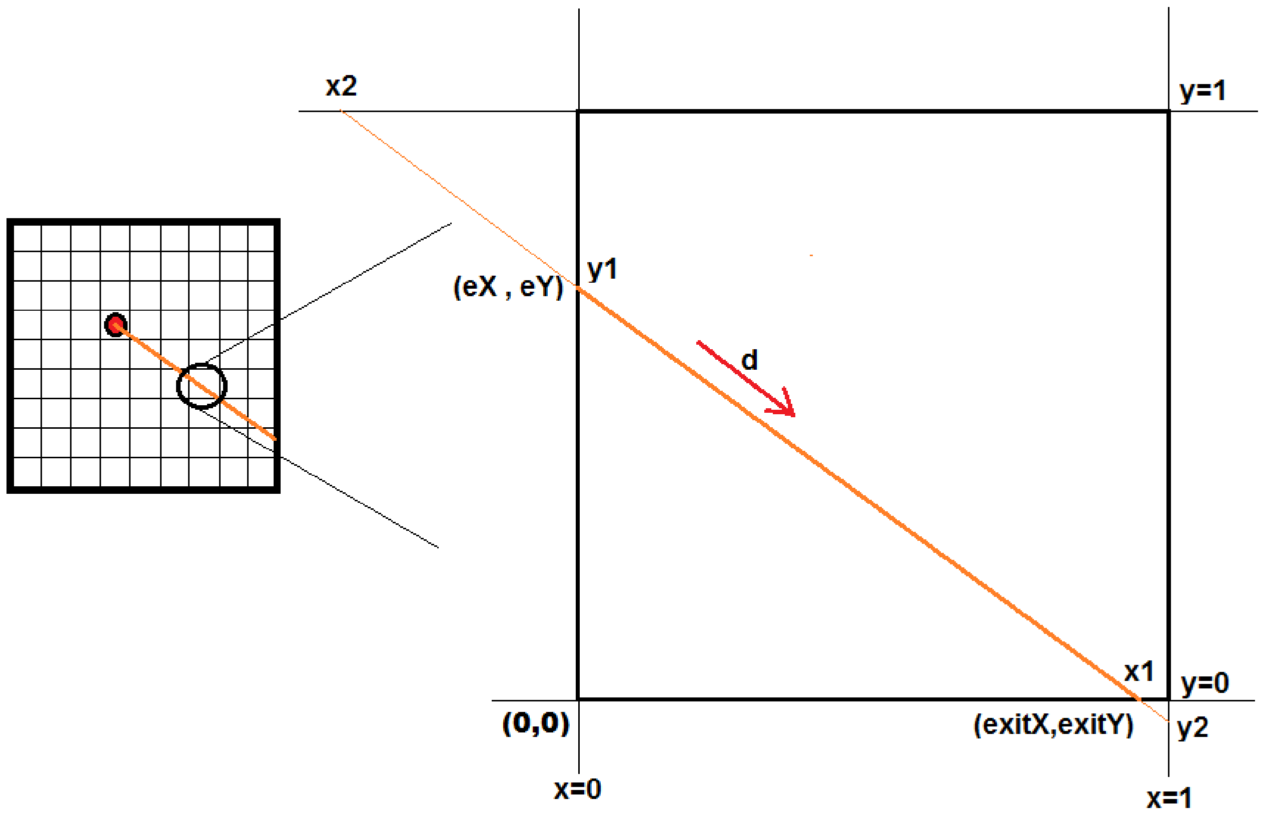


Figure 13: The scanning line gets translated to a 1x1 located at origin in the first quadrant

A `scanner` is constructed with a facing which represent the scanners direction of scanning (measured in radians) compared to the direction of the robot. A facing of 0 means the `scanner` scans the same direction as the robots front is facing.

The methods in the class are `normalize()`, `scan()`, and `rotate()`.

`Normalize` transforms a x or y coordinate point from the actual map to a value in the interval $[0,1]$. Using the formula $\frac{p - d*c}{d}$ where p is the coordinate value, d is the dimension of a cell and c is the actual cell of the point. This will be done for the first point as the scanner only works inside a 1x1 cell.

The `rotate()` method just changes the facing of the scanner.

The main body of the code is the `scan` method. Most of the code body consists of `if` statements that do the same thing (this could of course be improved but works for now, and this class wouldn't be used on a real robot). A `scan()` is called with a start point, a direction and a column and row. The scan method has the following steps (refer to **figure 13**).

1. Marks the cell corresponding to current column and row in the grid as seen. The method will return here if it's outside the grid or if an obstacle is reached.
2. The equation of the "scanning line" $y = ax + b$ gets calculated from the startpoint ((eX,eY) in **figure**) the direction and the intersection between the scanning line and the lines $y = 0$ and $x = 0$ respectively.
3. The intersections between the scanning line and the lines that bounds the cell are then calculated. These points are, if we refer to the code, (x1 , 0), (x2 , 1), (0 , y1) and (1 , y2).
4. The actual point to be used as exit point will then be found. The point has to be inside the grid and be in the direction of the scan. This is done by means of an if statement, as seen here for (x1 , 0) eg. If
   $(x2 \ >= \ 0 \ \&\& \ x2 \ <= \ 1 \ \&\& \ Math.sin(d \ + \ facing) \ >= \ 0)$
5. This point is then used to call `scan` again, calculating a new row and column. For the example in **figure xx** that would be, $d = d, eX = x1 \, , eY = 1, row = row - 1, col = col.$

The AStar class

As explained in the theory chapter page 9, the A* algorithm calculates our path by using depth/best first search. We have made the class generic. Together with the right `State` implementation it should be able to solve any graph problem.

We store our data using two different data structures in this class, firstly; a priority queue and secondly a hash map. The priority queue contains the `openList` and the hash map then contains our `closedList` and another keeps track of which states should be put in the `openList`.

Our implementation of the `AStar` follows the flowchart closely (Figure 1) seen on page xx. The first method offers the parent root to our priority queue giving us a starting point. The parent root is merely the first node in our path and giving it the root is necessary because

the first node should not have a parent.
From here we just get the next from the priority queue and if there is none we return `null`. In the case of goal reached we make a new linked list and give it the path from the current `state` and then go through each `states` parent and add it to the linked list `fullPath`.

The last thing the `findPath()` method does is find the neighbour nodes around the `current` received from the priority queue and make them a `child state` to the `current point` and offers the `child states` to the priority queue. This is all done in a while loop, that keeps running until we run out of `states` from the priority queue or the goal `state` has been encountered.

The two data structures used are quite helpful for keeping the time complexity of the `findPath()` method down. The priority queue is in fact a sorted list, thereby makes sure that adding `states` has a time complexity of $O(log(n))$, n being the size of the queue, withdrawing from it is done in constant time. The hash map, with a well implemented `hashCode()` and `equals()` methods, makes us able to store State objects so we can check in constant time if a `State`'s point already has been visited.

To properly understand how the `AStar` class works, we have to take a close look to the correlation with two other classes that makes the foundation for our `AStar` the `State` and `RoboState` class.

The State class

The idea behind the `State` class was to make an abstract class that can be used for the `AStar` class that is independent of the concrete problem at hand. The parameters for this class are h, g and a `State` object called `parent`. These are all needed for an A*-algorithm to work. The only non abstract methods in `State` are

- `f()`, the way to get the f value for the A* heuristic function.
- `compareTo()`, an overridden function for comparing `State` classes. Where f values will be compared to determine if one his greater than another. This will be used for the sorting in a priority queue.

The class also has three abstract methods that are needed for A* and thereby `AStar` to work, but are problem dependant.

- `isGoal()`, decides if this `state` is a at a goal.
- `findNeighbors()`, the expanding part of the A* algorithm. Essential for A* but problem dependant.
- `hashCode()`; the `hashCode` for a `State` object needs to be rewritten as we use a hashMap and we need to keep track if nodes has been seen before, and therefore the same nodes needs to map to the same values.

For our implementation we have created a `RoboState` class to extend our state class and we will described under the next heading.

The RoboState Class

The `RoboState` class inherits from the `State` class described above. It is within this class that we define our methods from the `State` class as well as the `RoboPoint` class, by overriding them. The calculation of the h and g values for the A* algorithm is also made within this class, along with this the class contains an `equals()` method to compare if two states have the same `RoboPoint` and a `hashCode()` method that provide each points in the class a hash value.

The `isGoal()` method enables us to check whether a point is the goal point or not.

The main method of this class is the `findNeighbors()` that returns a protected list of `states`. The method goes through finds all legal `states` succeeding the `current state`. Figure 14 shows how the loop goes through the `current state's` surroundings. The new `states` won't be in the same point as `the current` or inside an obstacle point. While checking the neighbour points we have an `if` statement that makes sure we don't move diagonally where it would seem to be an unnatural path. We use our `hasObstacleNeighbor()` method to help prevent the robot form doing this. If it's moving horizontal or vertical then it won't be a problem.

-1,1	0,1	1,1
-1,0	0,0	1,0
-1,-1	0,-1	1,-1

Figure 14: the cells that the nested loop in find neighbours goes through.

The next two methods are vital for calculating the f value for our path finding algorithm, the `setG()` and `setH()` values. Calculating g is a very simple use of `if` statements. If we are at the starting point of the grid, then g equals zero. Other states g value is the g value of it's parent adding 10 when we move vertical or horizontal and adding 14 when a diagonal move is made.

The value of the heuristic h is calculated by using the Pythagoras theorem $a^2 + b^2 = c^2$ to determine the hypotenuse of the right-angle formed by use of the Manhattan heuristic method. The hypotenuse value is multiplied by 10, and this returns the value of h. By calculating our heuristic function this way, we achieve a path finding program that guesses the shortest path by following the crow's flight directly to the goal point. As we can see in Figure 15

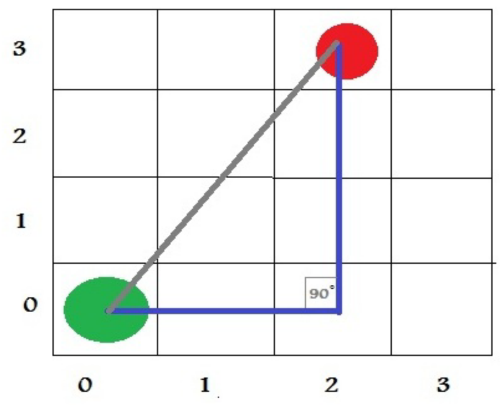


Figure 15: In this figure the blue lines represent the Manhattan Heuristic which will give the cell (0, 0) an H value of 60, while the gray line represents the Crows flight Heuristic which will give the cell (0, 0) an H value of 52 if we round to the nearest int.

The 2nd last method in the RoboState class is the hashCode() method that uses the RoboStates statePoint to calculate a hash value and returns it for the hash map. This hash value will be unique for each point in a grid up to a size of 10000x10000 points.

The equals() method we use for comparing objects, is a Boolean method for checking objects. We use it to see if two points are equal, meaning they are in the same spot on the grid by comparing their column and row placements.

The GraphicRepresentation Class

The `GraphicRepresentation` class simply instantiates the visual representation of the simulation by initialising a `StdDraw` window, along with containing one method `draw()` that calls all available `draw` methods from other classes and instantiates them on the same canvas.

The RobotTester class

The main class of the program is the `RoboTester` class. The idea is to test the robot simulation in a randomly generated map with random start point and a random goal for the robot to drive towards.

Our graphical representation uses the actual size of the map to represent the size of a pixel. This means that for large map the graphics representation will be rather large. At the moment the graphics representation looks best for map sizes between 200 and 500. While the actual size of a cell in the grid depends on the ratio between the size of the map and the number of cells. This means that if the `noOfCells()` gets to large the cells actual size becomes critically small, where the robot can pass through a cell within one tic. Because of this the number of cells in each row and column is actually not that random (limited between 6 and 20). A number of obstacles, approximately a forth of the number of cells are initiated. To create a starting point and a goal for the robot, we make an algorithm that randomly finds these and makes sure neither is inside an obstacle. The algorithm works as follows.

```
while( !has_found_goal_and_start) {

  -  randomly creates new start and goal;
     if (start & goal !inside_occupied_square)
             has_found_goal_and_start = true;

        }
```

The main method then, sets the new goal in the map and creates a new robot with a speed of a $^{1}/_{100}\,th$ of the size of the map, a random direction and start values found.

User Guide

Running the program

The `main()` method for running the simulation and seeing it visually is located in the RoboTester class. Running the RoboTester class will initialize the simulation and show a different map every time the program is run. If you would like to see a specific kind of map, please read the "maintain the code" section for information on hard coding the different values the map consists of. A executable jar file and the source code can be found at the following location: https://akira.ruc.dk/~truls/RoboSim/

The zip file at this location contains an IntelliJ project. It should be possible to compile and run in any IDE or on any OS through the command line or any of its counterparts for macOS or linux.

How to run the robot simulation with a command prompt.

According to the location of the program on your disk the following commands should be used for the windows command prompt (cmd):

```
C:\"where_ever_you_put_the_folder"\RoboSim191210\out\production\RoboSim> java RoboTester
```

Tester main methods

The code we have made contains multiple main methods that can be used for testing the different classes. If the code needs to be changed due to wanted improvement, there is an amount hard coding that must be taken into consideration, to test different variables. After any changes has been made, the program needs to be recompiled. The classes where tester methods are written for are:

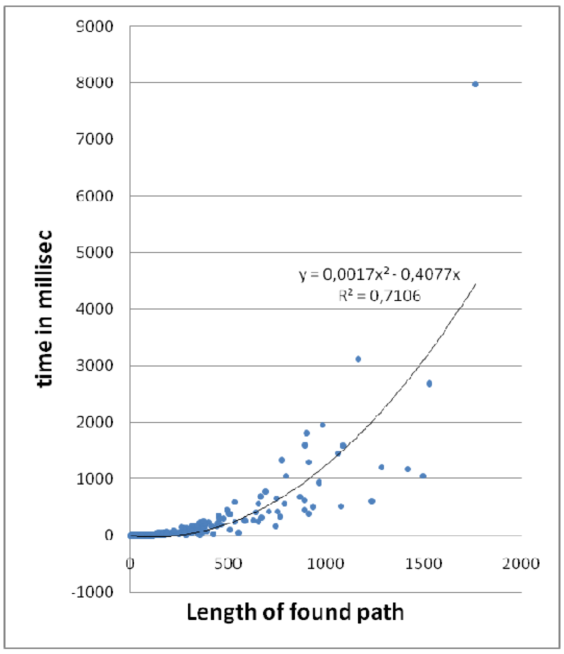
- The `Robotester` has an out commented main method that can be used for testing or robot program with chosen parameters.
- `AStar` has 2 main methods, the first testing the `AStar` for a single predetermined run, the 2nd tests several runs for time consumption.
- The `scanner` tester test the `scanner` class

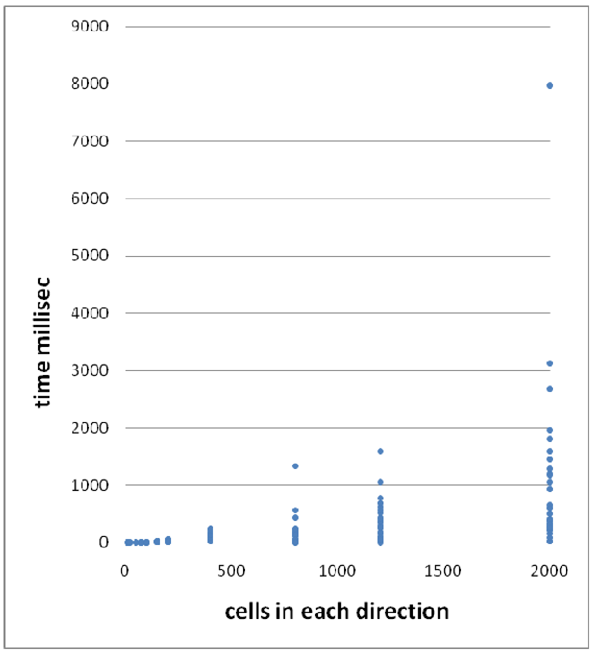
These testers classes might not be up to date so use with care.

Testing and evaluation of Application

Testing AStar for increase of the size of the grid

We have run a series of test on our A* algorithm using increasing sizes of the grid. We made 30 test each of grids with the sizes of 10x10, 20x20, 50x50, 75x75, 100x100, 200x200, 400x400, 800x800, 1200x1200,2000x2000. We made sure that the goal and start were at least 3/4th of the width of the grid apart. We recorded the number cells for a side together with the actual length of the path found and the time for the execution of the calculation. The result can be seen in the two graphs (Figure 16 and Figure 17Figure 16) below.





ding
found.

plotted

We can see that for problems of the size up to 400x400 the execution time stays below 1 seconds, a lot less for 200 cells and below (Figure 16). For bigger problems the times gets out of hand pretty quick, excel suggests a 2nd degree polynomial trend line to be fitted to the graphs, which suggest we have a complexity for problem. . A quick look trough of the code would suggest that we should have a where n is the number of iterations needed (somewhere in the vicinity of the length of the path) and m is the size of the priority queue.

For problems bigger than this, 3000x3000 cell problem, we get an `OutOfMemoryError` exception where the hash map keeping track of the `openList` gets overfilled. The maximum heap size of the machine we were running the `AStar` on, was approximately 260 Mb, so if java's virtual machine gets allocated some more memory we would be able to run larger scale problems.

Unit testing for corner cases

Throughout the programming and implementation phase we have tested our variables through unit testing, actually coming out in some of the corners of the architecture underway. As it is now, there is a lot of parameter values that could make the application behave badly or fail, especially in conjunction with each other, if chosen wrongly. These including but not limited to: `Robot speed`, initial x and y position, `Grid mapsSize` and `NoOfCells` ratio, the obstacles array. We know about these problems and in our main method for the robot tester, this have been taken into consideration, with what might appear as magic numbers.

We have seen that the graphical interface especially fares quite badly in larger values of the number of cells or actual map size, or runs out of memory.

Discussion

After working on this project we have experienced that there are various ways to go about making a path finding application that could be implemented into a Lego MINDSTORMS™ robot. In this section we will be discussing the decisions we have made with regards to our program and how these compare to other possible options. This includes such factors as the Path Finding Algorithm, generic programming, a live robot compared to a virtual robot, the robot behaviours, and to which extent our program is Object Oriented, and where is falls short of the requirement of OOP.

A*

In terms of realising our goal of making a well functioning generic path finding class we feel we have achieved a satisfactory result.

The generic version of our AStar implementation should work for any kind of graph traversal problem where the 3 conditions for A* (mentioned in the Path Finding chapter page 9) are satisfied. As long as a proper State extension is made. Our AStar class makes use of data structures (hash maps and priority queues) that works really well with the problem at hand, making our algorithm fast even when handling larger problems.

Our RoboState class that extends the State class makes the AStar work for our specific problem. We have tested the class and read through the code and it appears to be implemented correctly. Despite this the A* is not completely flawless as we have experienced glitches in rare occasions.

The RoboState is a project-specific extension of the generic State class and is where we define the methods for our specific robot. Results from unit tests show that the RoboState works satisfactorily and that it work quite fast for problems with less than 1000 cells. During the project we have used both the Manhattan heuristic method and the Crow's Flight heuristic method to calculate the h value. It became apparent that the Crow's Flight heuristic method, having the most optimist guess of the h value, was the best to administer for obtaining the optimal path.

Behaviours

We feel that the behavioural architecture works fine for our problem. The different behaviours work fine in conjunction with each other, and without each other, which is the point. We have implemented both kind of behaviours, servo and ballistic so we know that works. We also have each behaviour trying to control the movement of the robot and see that the arbitration works satisfactory. Knowing this we know that implementing more behaviours for our robot wouldn't cause to many problems.

We have some problems with getting the imported subsumpsion package to work properly on the virtual machine. We do know that it works on the actual NXT brick so we have not done anything to further investigate this after we got it running.

OOP design and implementation

In our program we have classes that apply OOP principles as well as classes that, sadly, do not. The classes that comply with the OOP paradigm are; the AStar-, the Behaviour- and the scanner class.

As mentioned the AStar in conjunction with the State class makes use of a generic architecture thereby making it compliant to OOP paradigm, as it can be reused for any problem that can be solved by the A* algorithm. The whole BBR architecture has the same qualities.

We have throughout our coding process tested individual classes and methods through unit testing when it seemed fitting. We also use our IDE's built-in debugger to find and solve bugs as we went along. This goes somewhat hand in hand with OOP's idea, but we do feel we could have been more design heavy before implementing our different classes.

The scanner class is modular and several scanners could be initiated on a robot which could scan in different directions. This can be seen in comparison to the robot class which leads us to realise that we have fallen short on the requirements of OOP on some fronts:

Other areas where our program fails to comply to OOP principles are the following: the grid, the draw function inside the robot class. Also most of our variables are public, that should optimally have been obtained using getter methods, to avoid the possibility of altering the variable values in the other classes. We feel that as long as we are the only ones working on the code this is a minor issue, but for more comprehensive programs with several programming groups this should be a must.

Conclusion

On the basis of the preceding discussion we can conclude we have made a working simulation of our problem at hand. We do not feel that all aspects of our initial goal have been satisfied. We have implemented a satisfactory A* algorithm in our robot simulation. We have with some difficulties got the LeJOS subsumtion architecture to work on the simulation. Our OOP part of the project trails a bit, and the finished product is not at all as polished as we would have hoped.

In terms of our specifications requirements we have realized all our five criteria for success:

- A simulated robot object with an application that can track and guide the robot object.
- A representation of the robot's surrounding including obstacles it may encounter in the real world.

- A class to determine routes between points given and to communication this information to the robot.
- A visual representation of the mapped area, as well as a representation of the robot’s movements on the map.
- We have implemented behavioural based architecture into the robot.

Future Perspectives

With our project there is room for much improvement some we would have like to implement ourselves but have been unable to due to several factors. When making a simulation, it becomes apparent that there are a lot of problems that we can't take into account when creating the simulation that will have to wait until a field test of a real robot is available. This section is dedicated to explaining some future prospects that might be useful for continuing the work on this application and in the end implement it into a robot.

Lego MINDSTORMS™ Robot compared to our virtual robot

As we have mentioned there are various factors that differ from a Lego MINDSTORMS™ robot to our simulated robot. Size and movement are some of the most apparent differences. A Lego MINDSTORMS™ robot's turning function will differ from the way our simulated robot turns as the MINDSTORMS™ robot turns by letting one of the wheel rotate forward, while the other wheel is suppressed or rotated in reverse. As such the method for turning the robot will need to be recalculated to allow for this variance in placement before being implemented into a MINDSTORMS™ robot.

The size of the robot could also prove to be problematic when it comes to a MINDSTORMS™ robot. As it stands the simulated robot is not much more than a point, and can as such run along the edge of the grid, or right up alongside obstacles. This might not seem problematic at first, but with a MINDSTORMS™ robot movement this close to edges could lead to parts of the robot being out of bounds, even though the point being tracked may be inside the grid. The same goes for the obstacle. If a MINDSTORMS™ robot with a pressure sensor hits an object even with the furthest corner of the sensor it might think that there is an obstacle directly in front of it, when in fact it might only have grazed the edge. This will result in the robot backing off and turning between 90 and 270 degrees, when it might in fact just need to turn a fraction in order to avoid the obstacle. When taking into account the tracking of a MINDSTORMS™ robot we may find that a MINDSTORMS™ robot is bigger than a grid cell. This is because the tracking takes place in a grid based on the infrared cast of the WiiRemote's sensor. The grid size is dependent on the height of the WiiRemote from the ground. This should be taken into consideration when implementing the tracking and path finding into a MINDSTORMS™ robot, and the cell sizes may have to be scaled accordingly.

When it comes to implementing the application into a MINDSTORMS™ robot is should be noted that uncertainties would be encountered in the real world that our simulation does not take into account. Variables such as the robot's position can be determined quite accurately in our simulation, but this might not be the case in the real world. As well as uncertainties regarding position the direction a MINDSTORMS™ robot is facing might also be difficult to determine with accuracy, whereas in our simulation this is always known. A way around this problem could be by mounting 3 diodes in a non-equilateral triangular pattern, thereby giving the program a way to track the robot by saying the front diode is the facing by comparing to the two opposite.

The scanner we have implemented for our simulation will be obsolete when transferring our code to the MINDSTORMS™robot as it is just a simulation to represent the functioning of an ultrasonic sensor. The code for an ultrasonic sensor will have to be made from scratch. We have not made any considerations towards making an ultrasonic code for implementation as we prioritized our time on the path finder class.

In the real world situation with a MINDSTORMS™ robot there might be a need for constants or "magic numbers" as Object Orientated variables might not comply with real world physical limitations.

A* and our project

To be able to make a path finding program for a robot so that it can move with ease inside a dynamic environment, we will have to not only be able to calculate the route for the robot but we also have to be able to adapt to newly found obstacles. The difficulty the robot faces by being unable to move set distances in a neat grid and instead having to be told each amount of time to let the motors run, only adds to the hardships a robot must endure. To ensure that our A* code does work, by making a simulation virtually, that removes the hardware and technical difficulties of the MINDSTORMS™robot, that we face trying to implement the program with the live robot.

With the simulation, we are able to move the robot inside a grid to follow its movement and be able to follow it very precisely. The simulation also shows us how we handle newly found obstacles.

The A* algorithm then tells the robot to go in a straight line to get to the goal, the fastest route possible. When the robot starts encountering our randomly placed obstacles, the appointed route will have to be revised. If the robot at any one given point has its chosen path blocked, it revises the path, a new path is given, as the A* algorithm is checking the path for each move the robot makes, ensuring the path is always the shortest. This gives us a somewhat dynamic interaction with the environment, as the robot adapts to its surroundings as they appear.

New questions are quickly raised as the implementation of the live robot, holds some apparent problems. If the environment supposedly is dynamic, then all obstacles will not be impassable. They might be movable or small enough that the robot will be able to pass right over it or maybe if there is some passable terrain, it might be slowed down by said terrain. This gives us some new calculation as to how the robot has to react and how the A* algorithm should take these problems into account.

If we were to implement passable terrain to the A* algorithm, the common solution is to give this terrain a higher cost to cross as the robot will explore other options before passing over it. Having a dynamic environment, the robot might spend a long time searching for another way around the passable terrain, as it has to explore around the goal to figure out if the passable terrain might surround the goal position. In other terms the time spent looking for a new route might be too consuming compared to just passing through the blocking terrain.

Having movable terrain makes great demands on the robot, as it has to be able to properly control the way of pushing the obstacle out of the way. That means calculating the exact place of the object and the robot itself to ensure a proper direction for the push. Every time the robot encounters an object in its path, it should speed up a little to see if the object is indeed capable of being pushed, this of course should only be used on objects in the immediate path laid down by the A* algorithm, as it would be much too time consuming driving the robot around to check every single obstacle of their attributes. Objects not in the way of the calculated path, is just ignored and will not be taking into consideration.

Tracking the robot

With a live robot we are unable to track it as well as the simulation is able to and this is without taking unknown obstacles in as a factor that might deter the robot from its clear-cut path. When we first talked about the details of this project, we wanted to use a Wii-Mote and specifically its infrared sensor to track the robot, by putting infrared emitting diodes on the top if the robot and hanging the WiiRemote from the ceiling. This would seem an ideal method of tracking a miniature scale of the robot for live testing purposes.

Moving the robot out into the real world and administering it to more useful purposes would give us a problem with the tracking since it is not possible to hang a WiiRemote up outside, or in an area you would like the robot to explore for you. In this scenario it possible to think of the robot receiving input from a satellite when we think of the project in bigger terms of usability.

References

Horstman Cay Big Java - 3rd Ed. - USA: Wiley, 2007. - ISBN: 978-0-470-10554-2.

Jones Joseph L Robot Programming: *A Practical Guide to Behaviour-Based Robotics.* - USA: McGraw-Hill, Inc., 2004. - ISBN: 0-07-142778-3

Larman Craig Applying UML and Patterns: *An Introduction to Object-Oriented Analysis and Design and Iterative Development*. - USA: Pearson Education, Inc., 2005. - ISBN: 0-13-148906-2.

Nilsson Nils J Problem Solving Methods in Artificial Intelligence. - USA: McGraw-Hill, Inc., 1971. - ISBN: 07-046573-8.

Weiss Mark Allen Data Structures and Problem Solving using Java 4th Ed. - USA: Pearson Education, Inc., 2010. - ISBN: 9780321456227.

Links

Lester Patrick A* Pathfinding for Beginners, 20. 11 2010. - 19. 12 2010. - http://www.policyalmanac.org/games/aStarTutorial.htm.

Wikipedia Object Oriented Programming, 18. 11 2010 – http://en.wikipedia.org/wiki/Object-oriented_programming.

Appendix