Bachelor's Final Project – Investigating Artificial Intelligence Applied to Robotics

Thesis · June 2013

1 author:

Aleksandar Mitrevski
Bonn-Rhein-Sieg University of Applied Sciences
2 PUBLICATIONS **1** CITATION

Bachelor's Final Project

Investigating Artificial Intelligence Applied to Robotics

Aleksandar Mitrevski

University for Information Science and Technology "St. Paul the Apostle"

Contents

1 Preface **vi**

1.1 Purpose of This Document vi

1.2 Tools Used vii

1.3 Acknowledgments vii

2 Introduction **1**

2.1 What is Artificial Intelligence? 1

2.2 Applications of Artificial Intelligence 2

2.3 Artificial Intelligence and Robotics 3

2.4 Organization of the Thesis 4

3 Computer Simulations and Visualizations **6**

3.1 Introduction 6

3.2 Finding Shortest Paths With A* 6

3.2.1 What is A*? 7

3.2.2 Related Work 8

3.2.3 Simulation and Results 9

3.2.4 Summary 14

3.3 Recognizing Handwritten Digits With a Neural Network 14

3.3.1 A Very Brief Introduction to Neural Networks 15

3.3.2 Related Work 17

3.3.3 Simulation and Results 18

3.3.4 Summary 23

3.4 Recognizing Characters in an Image 23

3.4.1 A Simple Algorithm for Extracting Characters in an Image 24

3.4.2 Related Work 27
3.4.3 Simulation and Results 28
3.4.4 Summary 30
3.5 Conclusion 30

4 Robotic Simulations **32**
4.1 Introduction 32
4.2 Technology Items and Software Architecture Used in the Project 32
4.2.1 Lego Mindstorms NXT 2.0 33
4.2.2 Additional Technology Items Used 34
4.2.3 Software Architecture Used by the Robot 35
4.3 Related Work 37
4.4 Robotic Simulations 37
4.4.1 Estimating Position Change of the Robot 38
4.4.2 Finding Goal Locations Using A* 39
4.4.3 Robot Localization Using a Particle Filter 41
4.4.4 Solving Simple Algebra Problems by Recognizing and Classifying Characters 48

5 Limitations and Future Work **52**
5.1 Introduction 52
5.2 Limitations 52
5.3 Potential Future Work 53

6 Summary **55**

7 References **56**

8 Index **60**

List of Figures

1 UML diagram of the classes used in the A* simulation 10
2 UML diagram of the classes used for visualizing A* 11
3 Results produced by A* on a 5 x 5 grid 12
4 Results produced by A* on a 10 x 10 grid 13
5 Results produced by A* on a 50 x 50 grid 13
6 Number of milliseconds required by A* as a function of grid size 14
7 UML diagram of the classes used in the neural network simulation 19
8 Training error vs number of iterations for a network with 30 hidden neurons 20
9 Training error vs number of iterations for a network with 50 hidden neurons 21
10 Training error vs number of iterations for a network with 75 hidden neurons 21
11 Training error vs number of iterations for a network with 150 hidden neurons . . . 22
12 Test set classification accuracy for the four networks 22
13 Input image to the character extraction algorithm 24
14 The input image converted to a binary form 25
15 Identification of character rows in the image 25
16 Identification of vertical character boundaries in the image 26
17 Identification of horizontal character boundaries in the image 26
18 Pseudo-code of the algorithm for extracting characters 27
19 UML diagram of the classes used in the program for extracting characters 28
20 Image used for testing the algorithm for extracting characters 29
21 Another image used for testing the algorithm for extracting characters 29
22 Description of the Alpha Rex robot 34
23 The robot including the wooden stick and the phone bag 35
24 Software architecture used in the project 36

25 Software architecture used in the project, including frameworks 36
26 The robot's world 40
27 The environment for testing the particle filter algorithm 43
28 The initial configuration of particles in the world 45
29 Distribution of particles after one iteration 46
30 Particle distribution after two iterations 46
31 Distribution of particles after the third iteration 47
32 Particle distribution after four iterations - the robot succeeded in localizing itself . . 47
33 Plus signs used for training 49
34 Minus signs used for training 50
35 Sample image used for testing the extraction/recognition process 51

1 Preface

1.1 Purpose of This Document

This document is my Bachelor's thesis and represents my attempt to examine a small part of artificial intelligence with simple applications to robotics. The work described here has two goals:

1. Developing intuition for the algorithms described in the thesis.
2. Applying the described algorithms to a robot, examining various issues that are faced by robotic machines.

Following these goals, the thesis has five main parts:

1. Section 2 introduces the reader to artificial intelligence and its applications to various fields.
2. Section 3 presents simulations and visualizations of different artificial intelligence and machine learning algorithms; the simulations developed here are introduction to the part where the algorithms are applied to a robot.
3. Section 4 of the thesis presents applications of some of the algorithms developed in part 2, as well as other algorithms, to a robot.
4. Section 5 discusses various limitations of my approach.
5. Finally, section 6 summarizes the document.

The final product of this thesis is a robot that is able to locate a goal within a known environment, localize itself within that same environment, and recognize numbers and some mathematical signs in an image, thus being able to solve simple mathematical problems. I should note, however, that the ideas presented in this document do not really offer new solutions to open problems in artificial intelligence; instead, the purpose of the work described here is investigation of a field that is relatively new for me.

1.2 Tools Used

The algorithms examined in this thesis were developed on a Windows 8 machine. More precisely:

- The simulations described in section 2 of the thesis were developed using the *Visual Studio 2012* development environment in the *C++* programming language.
- The visualization of the A* algorithm (pronounced A star), described in section 2.1, was made using *OpenGL.*
- The various plots in the document were generated using the *Gnuplot* plotting utility.
- The UML (Unified Modeling Language) diagrams presented were created using *StarUML.*
- An algorithm for extracting digits from an image was developed with the help of the *OpenCV* library.

While working on this project, a *Lego Mindstorms NXT 2.0* robot was used, programmed using the *MindSqualls* open-source .NET library for Mindstorms. The robot was controlled using a Windows Phone application and a Windows Communucation Foundation (WCF) service.

1.3 Acknowledgments

Working on this thesis would not succeed without the help and support of other people. First of all, I would like to thank my supervisor, Dr. Dwight Fultz, for accepting to supervise my work even though his career is not focused on artificial intelligence and robotics. I would also like to thank my university, the University for Information Science and Technology ”St. Paul the Apostle”, for allowing me to take one of their Lego Mindstorms robots while I was working on the thesis; without the robot, I would not have been able to see some algorithms in real action. Finally, I would like to thank the members of the *MakeBot* Robotics Club that were patiently listening while I was explaining some of the algorithms presented in this paper.

Aleksandar Mitrevski

2 Introduction

Artificial intelligence (AI) is a fascinating area which aims at making programs that are able to solve problems and reason like intelligent beings. Robotics, on the other hand, is a field that, using ideas from control theory, kinematics, dynamics, artificial intelligence, and others, is trying to design and develop machines that can automate tasks that are frequently performed by humans. Introducing new ideas, concepts, and applications, AI and robotics are constantly pushing the limits of what current technology can do. Both robotics and AI are very challenging, so one can spend the whole career trying to master just a small part of the story.

Combining different ideas from machine learning, computer vision, and robotics, my project is a modest attempt at examining both artificial intelligence and robotics. However, before going into details about my work, I introduce the term AI formally; after that, I mention applications of AI in different areas, paying considerable attention to the significance of artificial intelligence for robotics.

2.1 What is Artificial Intelligence?

According to Russell and Norvig (2010, p. 1), artificial intelligence is a field that focuses on development and analysis of *agents* that interact with the environment in an autonomous way. This broad definition contains five terms that need to be explained further:

- *Intelligence*: Artificial intelligence tries to develop systems that are as similar to intelligent beings as possible, which generally means able to mimic the way animals or people behave and act in the world. We can think of these systems as either software systems or a combination of software and mechanical parts.

- *Agents*: As Murphy (2000) discusses in length, agents are programs that are able to plan actions, sense the world, and act accordingly.

- *Environment*: The environment is the world in which a system exists and acts.

- *Autonomous*: Just as human beings and animals are able to act autonomously, intelligent agents are expected to exhibit similar behavior; in other words, agents should be able to act in the environment with as little supervision as possible.

- *Development and Analysis*: AI can be thought of as a separate scientific field with its own theory and practice: AI theoreticians are concered with developing models and theories about intelligent agents and practitioners apply the acquired knowledge.

Simulating intelligence is not simple; while talking about the history of artificial intelligence, Russell and Norvig (2010, p. 16-28) mention numerous attempts that have been done for this task, many of which were not that successful. However, the fact that this field still has many open questions is quite motivating; that is one of the main reasons why I started working on this topic for my thesis.

2.2 Applications of Artificial Intelligence

A big interest for artificial intelligence exists because of the vast amount of problems that can be solved with its help. A non-exhaustive list of AI applications is given below:

- *Robotics*: With the help of AI, robots are able to reason about the world and make decisions autonomously. A more detailed coverage of the significance of artificial intelligence for robotics is given in section 2.3.

- *Games*: In games, players expect their enemies and allies to make decisions that seem reasonable for a given situation. A classic example of game AI is *Pacman*, where enemies can purposelessly move around a two-dimensional world, but can also act intelligently when a player is nearby; *The Sims*, on the other hand, is a very nice example of a game where

characters are only partially supervised by the player. A much longer list of applications of artificial intelligence in the gaming industry is given by Millington and Funge (2009).

- *Expert decision systems*: Even though predicting actions and making expert decisions is not an easy task, numerous attempts at developing systems that can reason have been made in practice. Chajewska, Koller, and Ormoneit (2001) develop an approach for learning utility functions of a rational agent just by observing the agent's behavior over time. According to them, their method can be applied to situations where another agent is able to use the learned information for maximizing its own utility, and their experiments actually demonstrate this quite well. Other studies in this category are those by Rossini (2000), who reports a system for real-estate forecasting, and Vallejos de Schatz and Schneider (2011), who talk about decision systems in medicine.

The list given above can be extended with many more applications; moreover, it can be expected that other applications will be invented as computer science advances more. The possibilities of intelligent algorithms are practically limitless and exploring these possibilities is a very challenging, demanding, and enjoyable activity.

2.3 Artificial Intelligence and Robotics

Robots can be seen from two different points of view:

1. As purely mechanical creatures that do some specific task (for example, robotic hands that operate in factories).

2. As machines that are able to act autonomously. Acting autonomously in this context means being able to plan, execute actions, and learn from experience without (or with very little) human supervision.

Although the first point of view is valid and we can simply think of robots as mechanical creatures that just do what they are programmed to do, the second point of view now becomes dominant in the field of robotics because robots are expected to be able to sense changes in the environment and adapt accordingly. As a consequence, artificial intelligence, or, more precisely, its subfields: computer vision, machine learning, planning, and so forth, are very important for robotics.

Current robots are still not able to simulate human intelligence completely; however, advances in robotics are incredibly fast and many robotic systems are quite fascinating in what they do. Klingbeil, Carpenter, Russakovsky, and Ng (2010) explain an autonomous robot that is able to detect and press a button that leads to a desired floor in an unknown elevator; with their system, which is composed of a quite complex combination of computer vision and machine learning algorithms, a robot was able to correctly identify buttons in all of their test trials, failing to press a button in only one of the trials. Another amazing example of what AI can do for robotics is given by Thrun et al. (2006); their paper describes a robot that won a race of vehicles that are able to drive completely autonomously. Furthermore, Anguelov, Koller, Parker, and Thrun (2004) describe a model of a robot that can correctly sense doors in an environment, taking into account both open doors, that are easier to detect, and closed doors, whose detection is more difficult due to possible confusion with walls; Coates and Ng (2010), on the other hand, present a system that allows a robot to correctly identify a given object in a scene that is seen from multiple viewpoints. Numerous other examples can be added to this list; however, the ones presented are good representative of the great possibilities of AI and robotics.

2.4 Organization of the Thesis

Having introduced artificial intelligence and its significance for robotics, the remainder of this document is dedicated to my work in the project. More specifically, part three covers computer simulations and visualizations of algorithms that I was examiming and part four describes the application of some of the algorithms developed in part three to a Lego Mindstorms NXT 2.0

robot. In part three:

- Each of the algorithms is first briefly explained.
- The explanation is followed by a section in which I reference numerous authors that have used the algorithm in some applications.
- My simulations and visualizations of the algorithm are then explained in detail.
- Finally, I conclude each simulation by mentioning various assumptions and limitations of my approach.

In part four:

- I introduce the reader to the hardware components of the Lego Mindstorms NXT 2.0 robot that I was using.
- Numerous interesting applications of Mindstorms are then referenced.
- My attempts for programming the robot are explained.

In part five:

- I discuss various limitations of my work in this project.
- Potential plans for future work are then mentioned.

Finally, part six summarizes the thesis.

3 Computer Simulations and Visualizations

3.1 Introduction

Before applying a program to a more complex system, such as a robot, it is very useful to test the program in a smaller environment and make simple visualizations of it because visualizations can aid understanding and might develop intuition for certain problems; moreover, simulations can discover implementation errors that might have catastrophic consequences in real systems. Because of that, I dedicated the first part of my thesis to developing simulations of the following algorithms:

- The A* (pronounced A star) algorithm that is used for finding a shortest path between two endpoints. The algorithm and my simulation are explained in section 3.2.
- An artificial neural network that is able to learn characters from given data. Section 3.3 is dedicated to the neural network.
- A simple algorithm for extracting objects from an image. This algorithm is covered in section 3.4.

I should note here that I am using the unified modeling language (UML) for showing the relationships between different entities in my programs while explaining the simulations. An interested reader should consult Somerville (2011, pp. 118-142) for more information about UML.

3.2 Finding Shortest Paths With A*

Pathfinding, or searching more generally, is an important problem for AI and robotics because it arises in many situations: as a simple example, a robot that is supposed to turn lights on and off should know the position of a light switch, find its way to the switch, and then accomplish

its primary task. Throughout the years, different algorithms have been developed for solving this problem (see Russell and Norvig (2010, pp. 64-109) for a detailed coverage of the search problem and the algorithms that can solve it). One of them, the A* algorithm, is very popular in the artificial intelligence community, which is the main reason why I developed a simulation for it. In this part, its performance for a fairly simple problem is examined: a maze in a two-dimensional grid.

This section is divided in four subsections: 3.2.1 gives an explanation of the A* algorithm, 3.2.2 discusses related work, 3.2.3 presents results of my simulation and visualization, and 3.2.4 concludes the section.

3.2.1 What is A*?

A* is an algorithm that is used for finding a shortest path between two endpoints. In the context of graph theory, A* can be used for finding paths in directed graphs with non-negative edge weights.

Good introductions to the algorithm are given by Russell and Norvig (2010, pp. 93-99) and Millington and Funge (2009, pp. 215-237); therefore, I will include only some points that will be important when I explain my simulation in section 3.2.3. In the explanation below, the term *node* has the meaning from graph theory.

The goal of A* is to find a path that minimizes the cost function

$$f(n) = g(n) + h(n) \tag{1}$$

where *g(n)* is the cost to reach a node *n* from the starting node and *h(n)* is the estimated cost for reaching the goal node from *n* (*h(n)* is called the *heuristic function*). The algorithm heavily depends on our estimate of the cost for reaching the goal, such that it will work correctly only if the value of the heuristic function is less than or equal to the true cost of the path from node *n* to the goal node; in such cases, the AI literature says that the heuristic is *admissible*. At each iteration, A* processes the node that has the lowest value of *f(n)*, such that it stops when the processed node is

the goal node. The nodes that have already been examined by the algorithm are kept in a list called *Closed* in the AI literature; on the other hand, the nodes that are potential candidates for expansion in the current iteration are stored in a list called *Open*.

When implementing the algorithm, a trade-off has to be made between speed and memory efficiency: a fast implementation requires use of data structures that will increase the memory requirements of a program; conversely, a memory-efficient implementation will be considerably slower. Because the goal of this thesis was not to examine the relationship between speed and memory, I chose an impementation that is fast, but not necessarily memory-efficient.

3.2.2 Related Work

The A* algorithm is a great theoretical achievement, but it has also been practically used in many different contexts since its discovery. Recently, Gürel, Parlaktuna, and Kayir (2009) proposed a system composed of a mobile robot and a master station, such that the station gets the current coordinates of a robot, calculates a shortest path to the robot's destination using A*, and sends the path to the robot, which then follows this path and gets to its goal; even though the system has not been practically implemented at the time of writing their paper, a simulation showing that the idea is feasible has been tried by the authors. Popirlan and Dupac (2009) talk about another system that is based on communication between a path planner and a robot; once again, A* is used for finding an optimal path to the mobile robot's goal. Another interesting study by Masudur Rahman Al-Arif, Iftekharul Ferdous, and Nijami (2012) compares different pathfinding algorithms used in the context of an autonomous robotic vehicle that should rescue people in water; this study shows that A* is a very efficient algorithm, but also points to the difficult problem of choosing a good heuristic. Cui, Wang, and Yang (2012) propose an application of A* to robots used for rescuing people in earthquake accidents; in their paper, they demonstrate that A* is far more efficient than Dijkstra's algorithm (this is another commonly used pathfinding algorithm), which makes it more applicable to realistic situations in which fast and efficient decisions are necessary.

As A* is so often used in practice, there have been many attempts at improving its performance. In their paper, Rios and Chaimowicz (2010) list five different groups of algorithms whose goal is to make A* more appropriate for problems that have memory and speed limitations; in fact, they mention more than 20 different variants of the general A* algorithm that is examined in this thesis.

Finally, it is worth mentioning that, according to Millington and Funge (2009), pathfinding in games is almost always done with A*.

3.2.3 Simulation and Results

My A* simulation is developed in C++ and is visualized with OpenGL. For simplicity, I developed a program that simulates a two-dimensional grid world with obstacles, such that diagonal movement between grid cells is not allowed. The implementation is based on the min-heap data structure that is used for accessing the element with lowest value of the *f(n)* function in constant time, hence speeding up the algorithm significantly (see section 3.2.1 for an explanation of the function *f(n)*). The heuristic function that I used is the Euclidean distance which gives the straight-line distance between two vectors. The general equation for calculating Euclidean distance between two n-dimensional vectors *a* and *b* is given below:

$$distance(a, b) = \sqrt{\sum_{i=1}^{n}(a_i - b_i)^2} \tag{2}$$

A UML diagram of the classes (some of them are C++ structures, to be more precise) that are used in the simulation is given in figure 1:

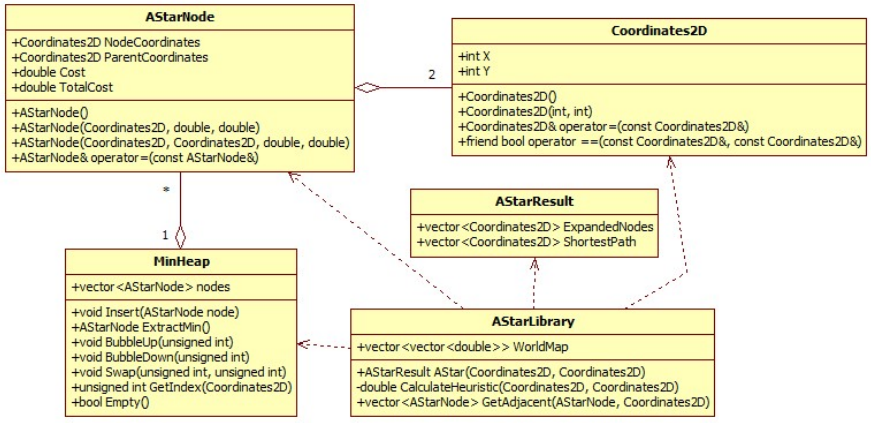

Figure 1: UML diagram of the classes used in the A* simulation

The classes on the diagram are explained below:

- *Coordinates2D* is a class that stores grid coordinates of a field and allows easy retrieval of the expanded nodes and the shortest path.
- The class *AStarNode* stores the coordinates of a node, the value of the functions *g(n)* and *f(n)*, as well as the coordinates of a node that leads to the current node with the lowest cost.
- *MinHeap* represents a heap structure and is used for storing the *Open* list of nodes in my implementation (see section 3.2.1 for the meaning of this list).
- *AStarResult* is a structure storing two lists: a list of coordinates of nodes expanded by the algorithm and a list of nodes that are part of the shortest path.
- Finally, *AStarLibrary* is the class that contains the actual implementation of the A* algorithm.

The visualization of the algorithm is handled by a different set of classes, whose UML diagram is given in figure 2:

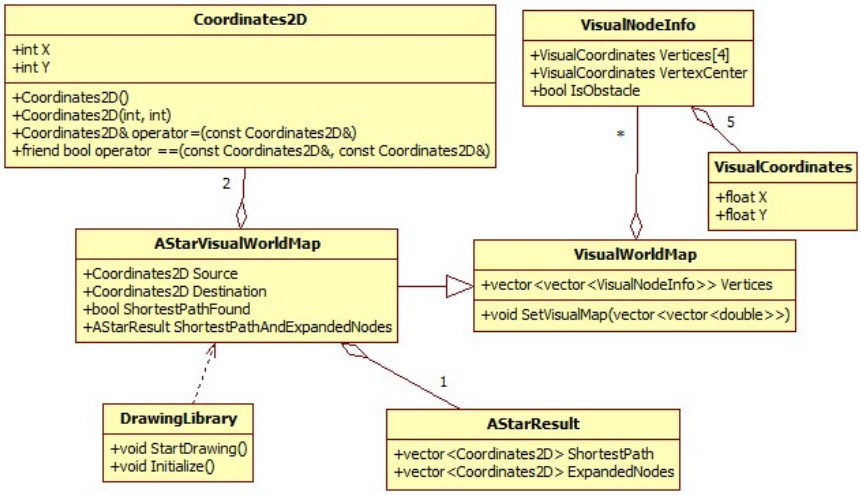

Figure 2: UML diagram of the classes used for visualizing A*

An explanation of the classes is given below:

- *Coordinates2D* and *AStarResult* were explained previously.
- The structure *VisualCoordinates* stores x and y coordinates in a form appropriate for OpenGL.
- *VisualNodeInfo* is a structure containing the coordinates of the four vertices of a grid cell, the coordinates of the center of the cell, as well as a boolean variable indicating whether the grid cell is an obstacle or not.
- *VisualWorldMap* is a class that stores the grid cells.
- *AStarVisualWorldMap* inherits from *VisualWorldMap* and stores the data produced by A*.
- *DrawingLibrary* is the class that takes care of the drawing in the program.

The algorithm was tested under the following scenarios:

1. A very simple 5 x 5 grid with few obstacles (figure 3).

2. A more complex 10 x 10 maze (figure 4).

3. The most complex grid on which the algorithm was tested had size 50 x 50 (figure 5).

In all tests, I assumed that the cost of moving from one cell to another is 1 and the cost of hitting an obstacle is 100. In the figures given below, the white cells represent expanded nodes by A*, the green cells represent obstacles, the black cells represent unexplored nodes, and the red line is the path found; the letter S in the figures denotes the starting position, and the letter G denotes the goal position.

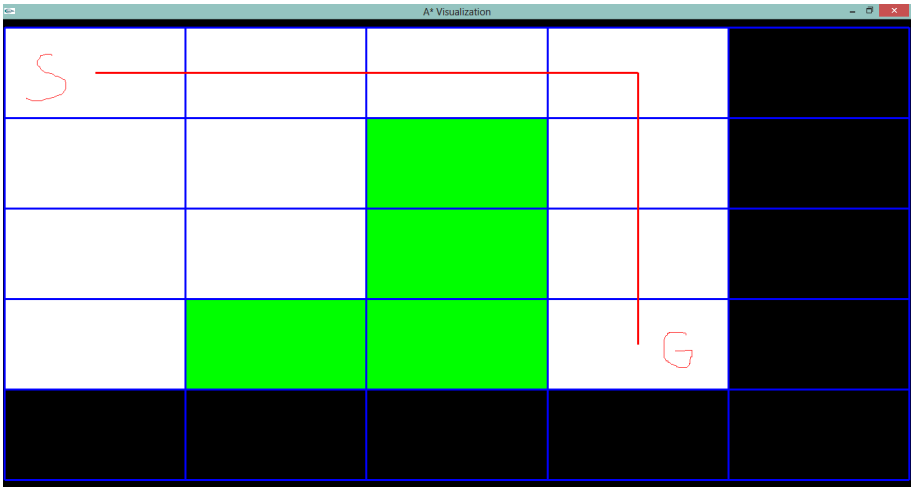

Figure 3: Results produced by A* on a 5 x 5 grid

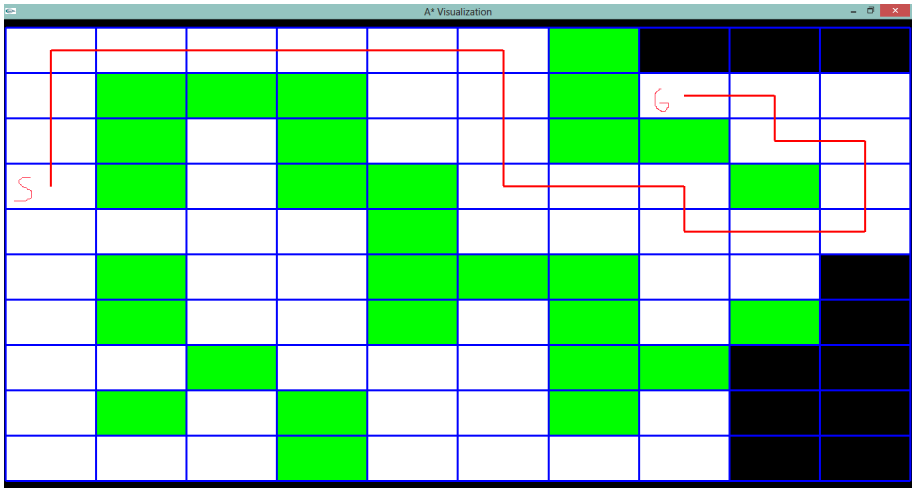

Figure 4: Results produced by A* on a 10 x 10 grid

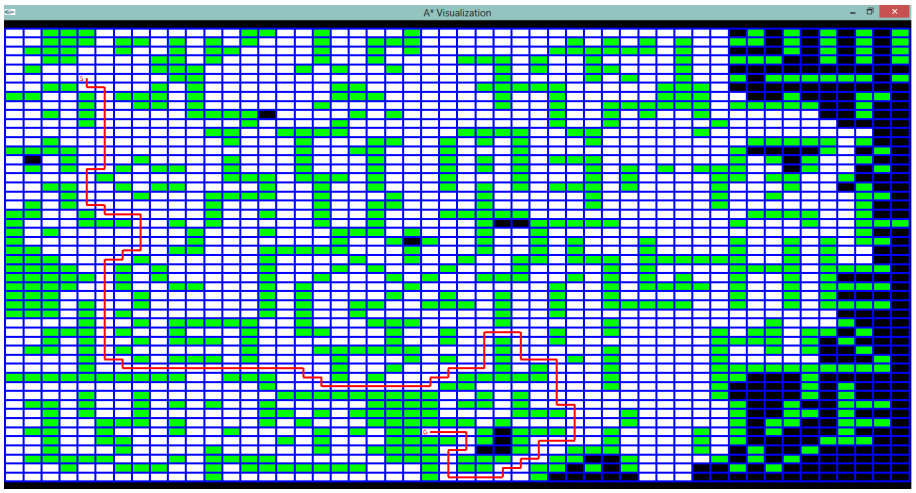

Figure 5: Results produced by A* on a 50 x 50 grid

As can be seen on the figures, the results produced by A* are quite satisfactory because an optimal path was found in all three cases. It is worth mentioning, though, that the time required for finding an optimal path increases considerably with increasing grid size. A plot showing the relation between the grid size and the number of milliseconds that pass before A* ends is given in figure 6:

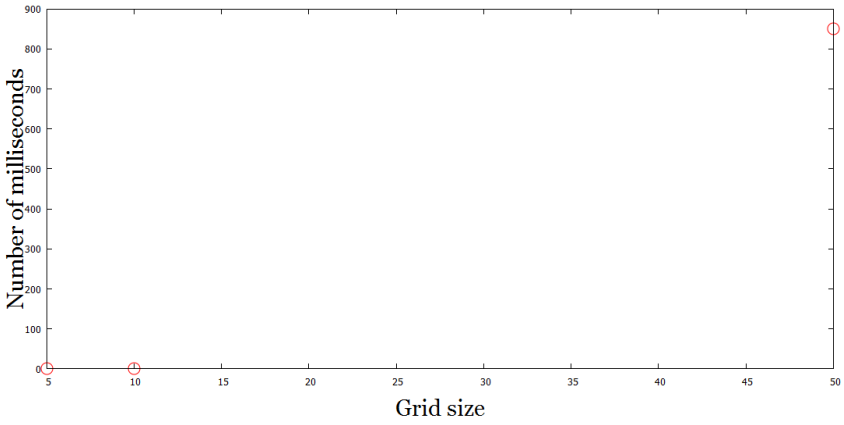

Figure 6: Number of milliseconds required by A* as a function of grid size

3.2.4 Summary

The visualizations presented in section 3.2.3 increased my confidence in the implementation of the A* algorithm because a shortest path was found in all of the test cases; however, it is worth noting that I didn't test A* with different cell transition costs and different heuristic functions. Figures 4 and 5 show that my algorithm visits most of the grid cells before finding an optimal path; for reducing the number of cells visited, the values of the heuristic function and the transition costs should definitely be reconsidered.

Before closing this section, I would like to mention that if the algorithm used in the simulation is directly applied to a robot, the robot would need to have a predefined map of the environment; otherwise, it would not be able to search and find the desired destination. This and other issues that arise in practical applications are discussed in part four of the thesis.

3.3 Recognizing Handwritten Digits With a Neural Network

One of my goals in this thesis was to develop algorithms that allow a robot to recognize numbers and mathematical signs in an image (handwritten or printed) and make simple mathematical calculations. This problem is essentially composed of two related subproblems: detecting characters in

an image and then recognizing the characters. The more difficult problem of detecting objects in an image is addressed in section 3.4, where I present a simple algorithm that can be used for this task. The focus of this section is the problem that comes after the detection step: correct classification of characters.

Machine learning gives many algorithms that can be used for different classification tasks: *Naïve Bayes*, *K Nearest Neighbors (KNN)*, *Linear* and *Logistic Regression*, *Neural Networks*, *Support Vector Machines (SVMs)*, and many others. My decision in this thesis was to apply just one of them to the problem of classifying handwritten characters: the neural network algorithm.

The organization of this section is as follows: 3.3.1 gives a brief overview of neural networks, 3.3.2 refers to research in the area, and 3.3.3 discusses my simulation of a neural network. Finally, 3.3.4 concludes the section.

3.3.1 A Very Brief Introduction to Neural Networks

A neural network is a machine learning technique that can be classified in the group known as supervised learning techniques (see for example Han, Kamber, and Pei (2012, p. 330) for an intuitive explanation of the term supervised learning). Even though this algorithm is inspired by the way our brains work, the ideas behind it are purely mathematical.

The short explanation given below is based on Mitchell (1997, pp. 81-124), so I advise the reader to consult this reference for more details about the algorithm.

In the most general case, a neural network is a fully connected directed graph which consists of: an input layer of neurons, zero or more hidden layers, and an output layer. This graph can be considered directed because information flows from the input layer to the output layer, such that the input is a feature vector representing a particular training pattern and the output is either a number or a vector representing the belief that the input pattern belongs to a certain class. Because of the direction of information flow, these types of networks are often called *feed-forward networks*.

The simplest type of a neural network is called *perceptron*, consisting of an input and an output layer, but no hidden layers; this type of a neural network is suitable for problems in which the classes can be separated by a linear decision boundary. The problem with linear decision boundaries is that realistic problems are rarely linearly separable; in fact, many real-world examples are characterized by ambiguities, so it is quite difficult to draw a separation line that will clearly separate things into different classes.

Neural networks that have hidden layers solve the problem of linear separability, such that they are able to approximate even non-linear decision boundaries; these networks are often called *multilayer perceptrons*. A neural network approximates decision boundaries by learning the *weights* of the connections between the neurons in the network.

One of the most frequently used algorithms for the purpose of learning the weights is known as the *backpropagation* algorithm and is the one that I used in my simulation explained in section 3.3.3. Backpropagation is a function minimization algorithm; the function that it minimizes is the *classification error* on the training set, given by equation (3):

$$Error = \frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{k}(Expected_k - Obtained_k)^2 \tag{3}$$

where *n* is the number of patterns in the training set, *k* is the number of output neurons, $Expected_k$ is the expected output of the k_{th} neuron, and $Obtained_k$ is the obtained output of the k_{th} neuron. When a network introduces classification errors, backpropagation transfers this information from the output layer to the input layer; passing information back means that the algorithm is modifying the connection weights. The algorithm can stop in two cases: when the classification error falls below some threshold level or when the number of iterations exceeds some maximum allowed number, but the desired threshold level was not reached.

When passing information from the input layer to the output layer, a neural network uses what is called an *activation function*, whose purpose is to give the output value of neurons in the

network. Activation functions used in multilayer perceptrons are nonlinear because they make the network capable of representing nonlinear decision boundaries; moreover, these functions scale the output within some specified range, hence eliminating overflow problems with variables in programming languages.

In practical applications of a multilayer perceptron and the backpropagation algorithm, many choices have to be made: the number of input neurons that will be used, the number of hidden layers and hidden neurons, the activation function, the threshold level that we want to achieve with the classification error, the *learning rate* parameter that is used by the backpropagation algorithm, and the number of iterations that we want to allow in the training (if we allow more iterations that necessary, the network can *overfit* the training data, which means that its classification results on the training set will be almost perfect, but it will perform poorly on real data). My simulation, explained in section 3.3.3, was made with these choices in mind.

3.3.2 Related Work

Neural networks have been applied to the problem of character recognition by many research groups. Le Cun et al. (1990) describe an approach for recognizing handwritten characters using a multilayer perceptron having four hidden layers, producing training set error of only about 3%, which is quite impressive. Garris, Wilkinson, and Wilson (1991) present a slightly different method for recognizing handwritten digits, such that they are producing 32-dimensional input vectors for a neural network trained with the backpropagation algorithm, as opposed to the 256-dimensional vectors used by Le Cun et al. Their results are also satisfactory, as they report test set accuracy of about 92%. Knerr, Personnaz, and Dreyfus (1992) also discuss a neural network and its application to the problem of classifying handwritten characters. Being an attempt at improving neural networks classification, their approach does not use the backpropagation algorithm for training; however, the experimental results are comparable to those produced by a conventional multilayer perceptron.

It is worth mentioning that character recognition is not the only application of neural networks, such that several other problems have been approached with this machine learning algorithm. A study examining the potential of neural networks for distinguishing corn crops and weeds is performed by Yang, Prasher, Landry, Ramaswamy, and Ditommaso (2000); the authors compare classification results obtained by varying the number of neurons in the hidden layer and by changing the form of the output layer, reporting 80-100% classification accuracy for crops and 60-80% accuracy for weeds. Furthermore, in three different literature review studies, Kolanoski (1995) reports use of neural networks in particle physics, Shi and He (2010) discuss how neural networks can be used in medical image recognition for problems like edge detection and enhancing, correctly classifying cancer cells, and so forth, and Patnaik, Anagnostou, Mishra, Christodoulou, and Lyke (2004) review their use in the design of wireless antennas and antenna arrays. Finally, Adebiyi, Ayo, Adebiyi, and Otokiti (2012) analyze an approach for predicting price trends on stock markets using a three-layered perceptron trained with backpropagation; the results obtained demonstrate that their network is able to obtain a good prediction of the real trends on the market.

3.3.3 Simulation and Results

Just as the A* simulation that was described in section 3.2.3, I developed my neural network program in C++. The neural network has three layers: an input layer, one hidden layer, and an output layer. The activation function that I used is the *sigmoid function* (equation (4)), which produces outputs in the range [0,1] and, due to its continuity and differentiability, has a well-defined derivative (equation (5)):

$$f(x) = \frac{1}{1 + e^{-x}} \tag{4}$$

$$\frac{df}{dx} = \frac{e^{-x}}{(1 + e^{-x})^2} = \frac{1}{1 + e^{-x}} - \frac{1}{(1 + e^{-x})^2} = f(x)(1 - f(x)) \tag{5}$$

A UML diagram of the classes used in the program is given in figure 7:

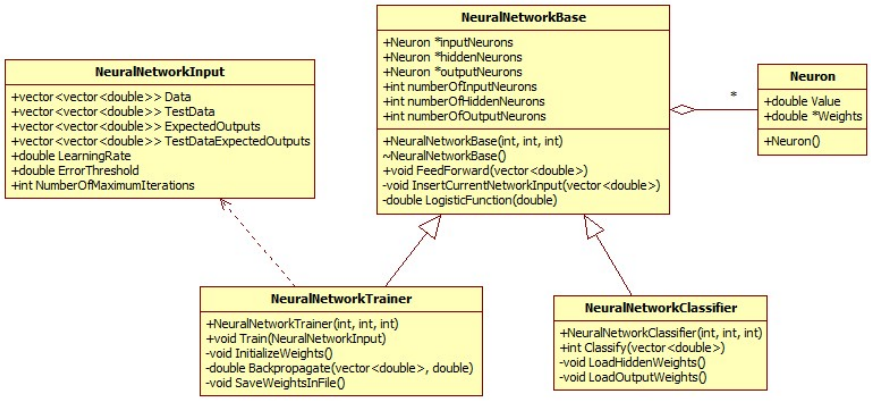

Figure 7: UML diagram of the classes used in the neural network simulation

The reader can see the following classes on the diagram:

- *Neuron* is a class representing one neuron in the network, storing a field for the neuron's output value and an array of weights that connect the neuron to the next layer.
- The class *NeuralNetworkBase* stores the architecture of the neural network.
- *NeuralNetworkTrainer* is a class that inherits from *NeuralNetworkBase* and is in charge of training the network.
- *NeuralNetworkClassifier* also inherits from *NeuralNetworkBase* and performs classification once the network is trained.
- Finally, *NeuralNetworkInput* is a structure that stores data for training and testing the network; more precisely, it stores the training and test patterns with their expected outputs, the learning rate for the backpropagation algorithm, the minimum error that we want to achieve during the training, and the maximum number of iterations of backpropagation.

For testing the network, I used the Semeion handwritten digit dataset that is included in UCI's (University of California, Irvine) machine learning repository. The Semeion dataset consists of 1593 digits given as 256-dimensional binary feature vectors; I used 1195 of the vectors as training set for the network and 398 vectors (which is roughly 25% of the dataset) as test set. The patterns that are part of the test set are obtained using the pseudorandom number generator provided by the C++ standard library; hence, the test set is different at each run of the program.

During my simulation, I used a learning rate equal to 0.1 and an error threshold of 0.05. Figures 8, 9, 10, and 11 show plots of the error as a function of the number of iterations of back-propagation for networks with 30, 50, 75, and 150 hidden neurons respectively.

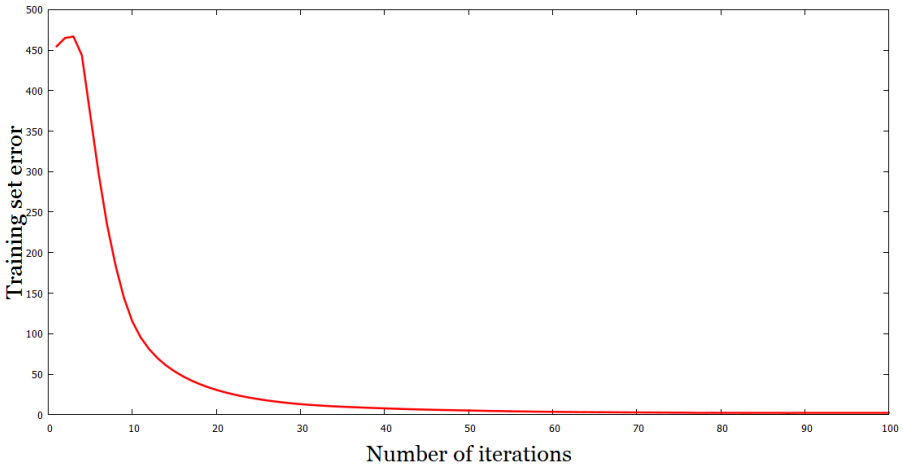

Figure 8: Training error vs number of iterations for a network with 30 hidden neurons

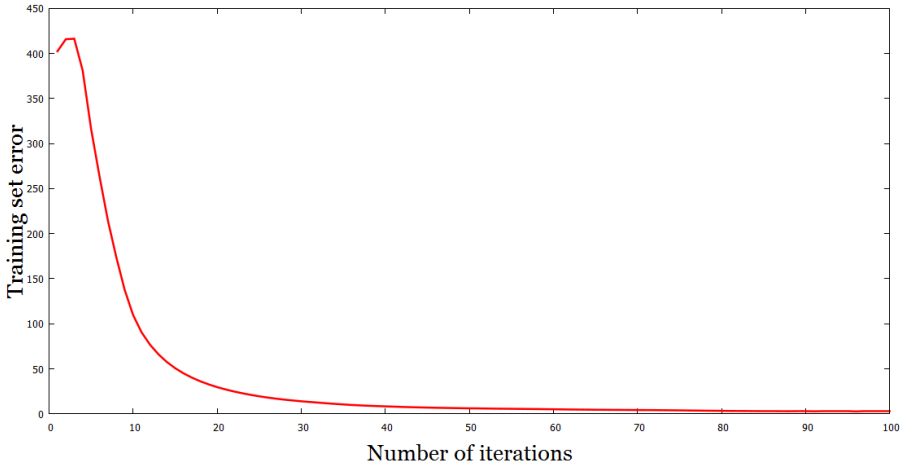

Figure 9: Training error vs number of iterations for a network with 50 hidden neurons

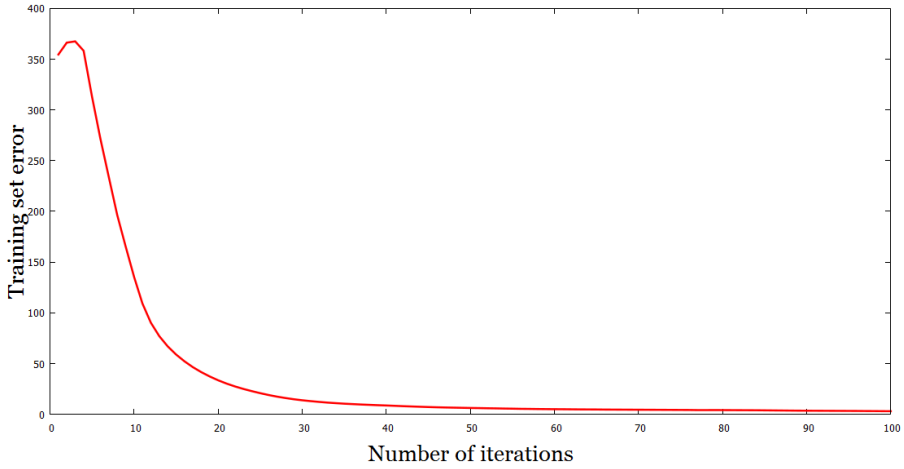

Figure 10: Training error vs number of iterations for a network with 75 hidden neurons

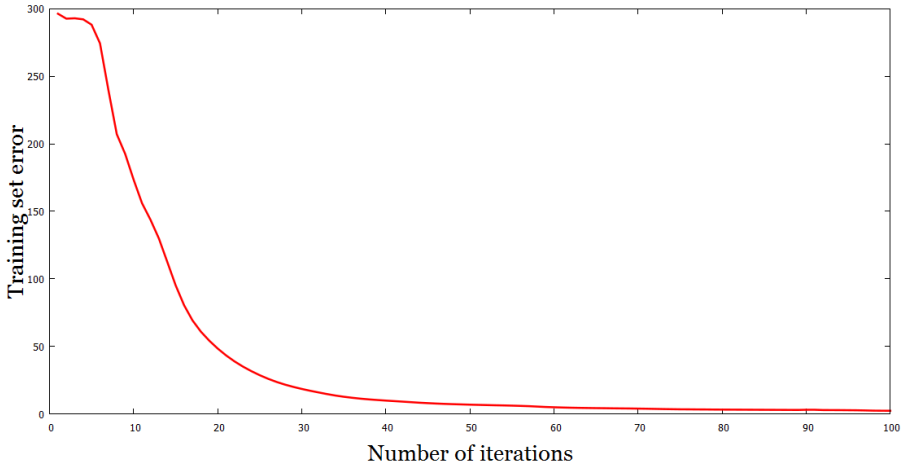

Figure 11: Training error vs number of iterations for a network with 150 hidden neurons

The plots show that after about 50 iterations of the backropagation algorithm, the training error is almost equal to zero; hence, further training might only cause overfitting.

For all four architectures of the neural network, I determined the classification accuracy on the test sets; a plot showing the accuracy as a function of the number of neurons in the hidden layer is given in figure 12:

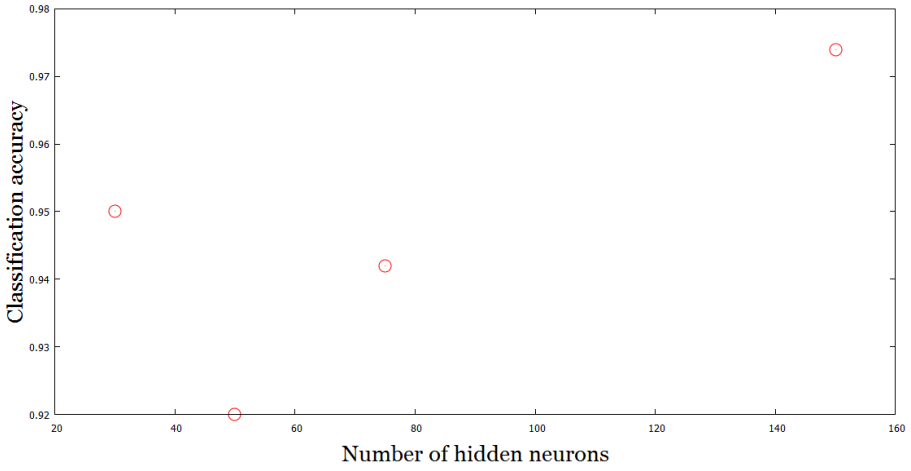

Figure 12: Test set classification accuracy for the four networks

3.3.4 Summary

While working on the simulation of the neural network, I had a chance to learn quite a lot about this impressive machine learning technique in general and about the backpropagation algorithm in particular. Moreover, the results shown in the previous section demonstrated quite promising results, such that the classification accuracy exceeds 90% in all four cases. One thing to note, though, is that I did not experiment with different values for the learning rate or the minimum error threshold in the algorithm.

3.4 Recognizing Characters in an Image

Object recognition is a skill that is ubiquitous to human beings, such that our actions in the real world are based on our skill for identifying objects and taking appropriate actions with them. Having the role of a camera, our eyes are able to scan the environment and pass the scanned images to the brain, which then processes them and makes conclusions about the scanned scene. As this skill is well-trained in humans, we hardly even notice that a very complex process is happening whenever we look at something.

When we think of robots, however, we have to take into account the fact that machines do not have eyes; instead, they are using digital cameras for creating images of the surroundings. Unfortunately, the result produced by cameras is a meaningless array of pixel intensity values; as a result, we need to find a way of analyzing the pixels if we want to extract some meaning from those images.

My goal in this thesis was not to make an algorithm for recognizing all kinds of objects; instead, my goal is narrowed down to identifying characters. Therefore, this section is dedicated to a simple algorithm that can be used for extracting these objects from an image.

The section is divided into four subsections: 3.4.1 introduces a simple approach for extracting characters, 3.4.2 discusses related work, and 3.4.3 presents results of my simulation; finally

3.4.4 summarizes the section.

3.4.1 A Simple Algorithm for Extracting Characters in an Image

Optical character recognition (often abbreviated OCR) is a difficult problem in general because we might have inverted and slanted characters in real-world situations; moreover, our viewpoint might affect the way we perceive characters.

My algorithm is based on an assumption that the image taken is either an image of printed characters or an image of a blackboard; this assumption is reasonable if we treat a robot as a student, for example. The credit for the general approach in my algorithm goes to Jackel et al. (1995). As an illustration of my algorithm, I will use the image shown in figure 13:

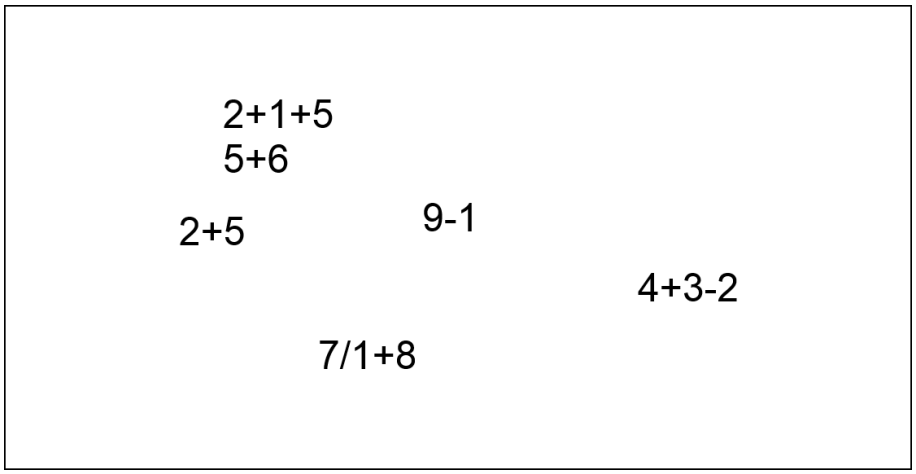

Figure 13: Input image to the character extraction algorithm

The input to my character extraction algorithm is a grayscale image. The first step of the algorithm is converting the image to a binary form using an intensity threshold; the conversion gives white pixels in the positions where the characters are located and black pixels for the background (see figure 14):

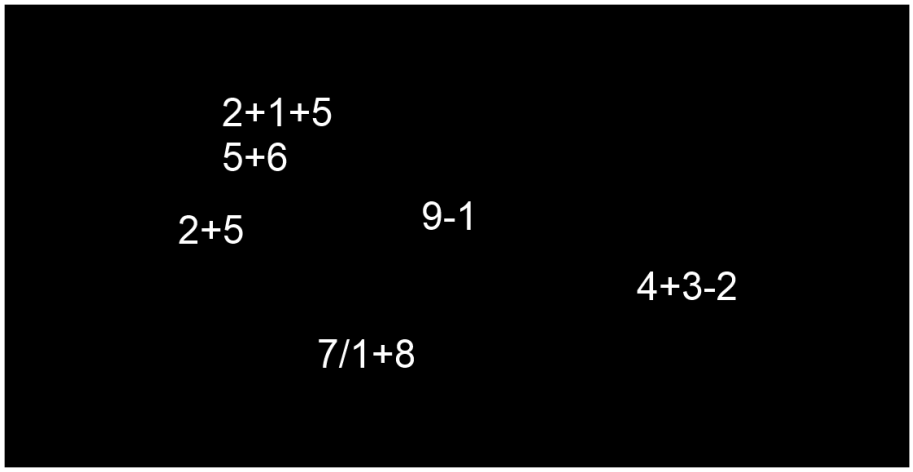

Figure 14: The input image converted to a binary form

Once the image is converted, the algorithm identifies horizontal boundaries in the image; this step is based on an assumption that at least some indentation exists between rows of characters. For finding the rows of characters, the whole image is analyzed (see figure 15):

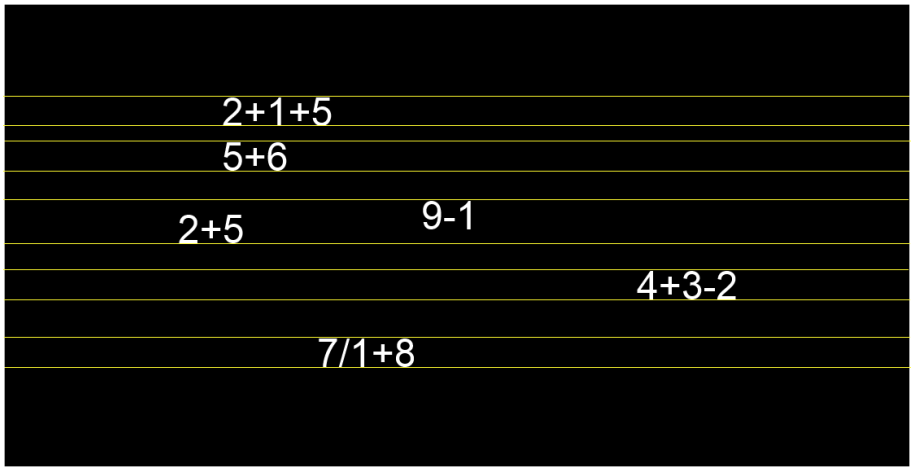

Figure 15: Identification of character rows in the image

After the rows of characters are identified, vertical boundaries between characters are located. In this step, only the regions within the previously identified rows are analyzed, as shown in figure 16:

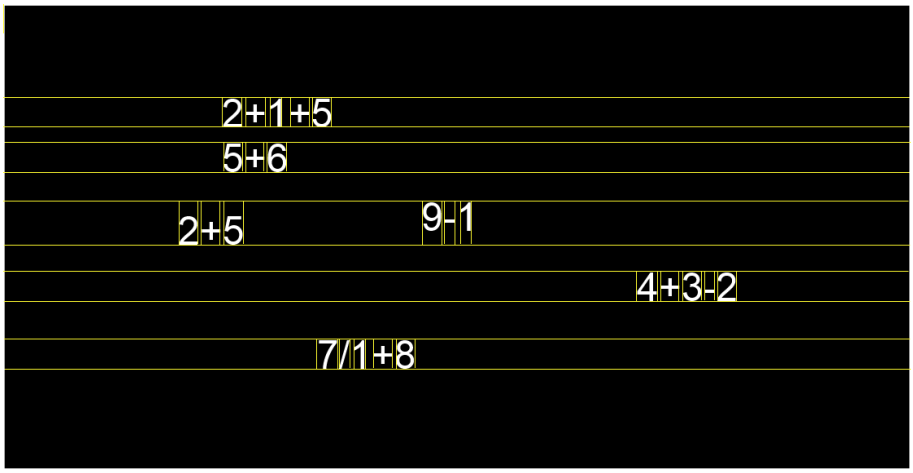

Figure 16: Identification of vertical character boundaries in the image

In the next step, the algorithm finds the horizontal boundaries of the characters in the image (figure 17):

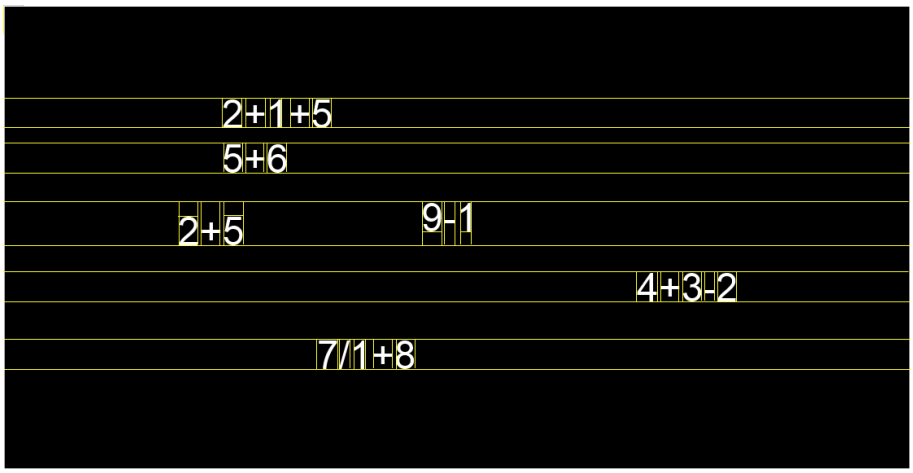

Figure 17: Identification of horizontal character boundaries in the image

Finally, the characters are extracted from the image one by one; several small images are produced in this step.

A pseudo-code of the algorithm is given in figure 18:

```
List<Characters> ExtractCharacters(image)
begin
    binaryImage <- ConvertImageToBinary(image)
    List<Rows> bounds <- RowBounds(binaryImage)
    List<Vertical> vertBounds <- VertBounds(binaryImage, bounds)
    List<Horizontal> horizBounds <- HorizBounds(binaryImage, bounds, vertBounds)
    List<Characters> characters <- Characters(binaryImage, vertBounds, horizBounds)
    return characters
end
```

Figure 18: Pseudo-code of the algorithm for extracting characters

As one might notice, this algorithm would not succeed in identifying numbers that are written connected, a situation that might arise in many handwriting recognition tasks; hence, the algorithm's domain is limited to images of characters that can be separated by vertical line boundaries.

Results of using the algorithm on real images are given in section 3.4.3.

3.4.2 Related Work

Research endeavours in the field of optical character recognition show that OCR is quite difficult to perform correctly in many cases; however, it can be applied in different real-life situations. The paper by Jackel et al. (1995) is a very nice example of a system for recognizing handwritten ZIP codes, which is a combination of character identification algorithm and a neural network for recognizing the characters; their system was practically used not only for recognizing ZIP codes, but for automatic reading of other documents as well. Yamaguchi, Nakano, Maruyama, Miyao, and Hananoi (2003) apply character identification and recognition to the problem of recognizing phone numbers; in their approach, they take into account the orientation of the numbers, and report extraction and recognition rates of 98.8% and 99.2% respectively. In another study, Suri, Walia, and Verma (2010) describe how license plate numbers can be extracted once the edges in an image

are correctly identified.

3.4.3 Simulation and Results

For testing the feasibility of the algorithm described in section 3.4.1, I developed a program in C++ using the OpenCV library. In the program, I read the input image in grayscale format, converting it to binary form using an intensity threshold value of 128 and then inverting the colors for achieving the effect of having white intensity for the pixels that have characters and black intensity for the background pixels. After this transformation, the code given in figure 18 is almost completely followed: first, the row boundaries are identified and returned, and then the vertical and horizontal object boundaries are found. Finally, the characters are extracted from the original image.

A UML diagram of the classes and structures in the program is given in figure 19:

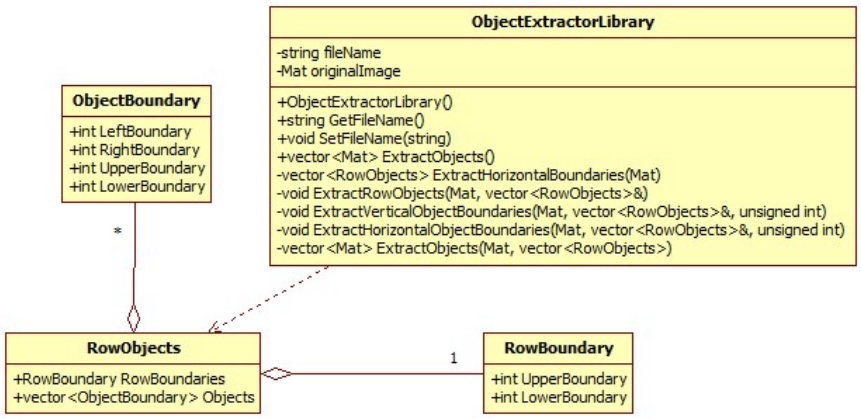

Figure 19: UML diagram of the classes used in the program for extracting characters

The classes and structures in this figure are explained below:

- *RowBoundary* is a structure storing the indices of the upper and lower boundaries of a character row in the image.
- The structure *ObjectBoundary* stores indices of the left, right, upper, and lower boundaries of characters in the image.

- *RowObjects* stores the boundaries of a row and a list of boundaries of objects identified within the boundaries of the row.

- *ObjectExtractorLibrary* is a class that contains the main logic of the algorithm, extracting the characters in an image.

The images on which I tested the program are shown in figures 20 and 21:

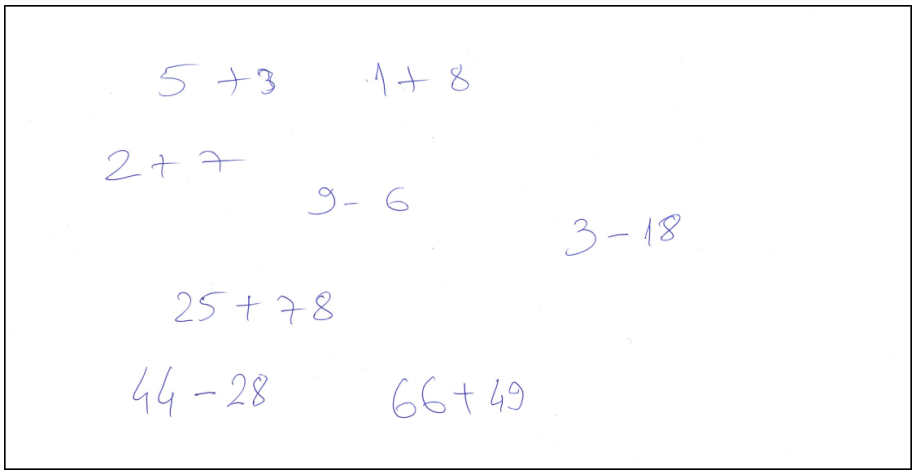

Figure 20: Image used for testing the algorithm for extracting characters

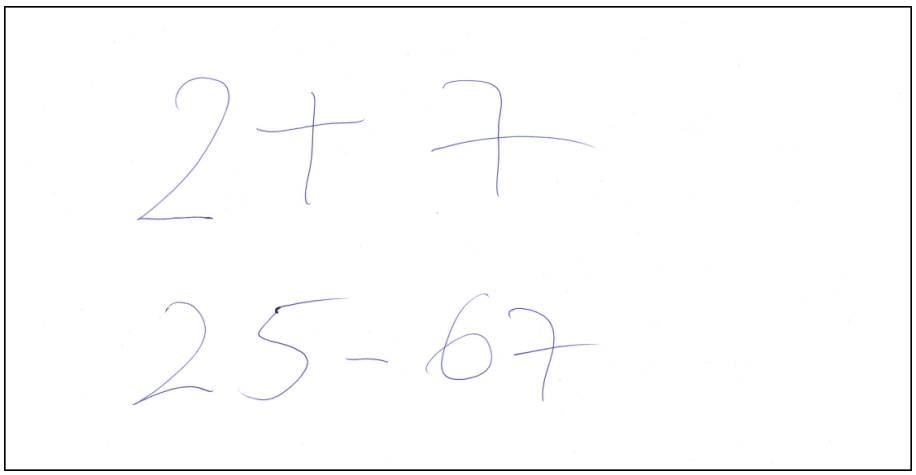

Figure 21: Another image used for testing the algorithm for extracting characters

On both images, all characters were extracted correctly. One thing should be noted, though: few scanning artifacts (vertical lines on the edges) were extracted along with the characters on the second image. However, a classifier trained to recognize both correct and incorrect examples of characters would be able to deduce that these images are not characters, so that is not a huge problem.

3.4.4 Summary

Extracting characters from an image is a difficult task and my algorithm clearly demonstrates that. In any case, I should say that this simulation was quite useful because it helped me identify many potential problems in character recognition and object recognition in general.

The two most important factors that can reduce the performance of my algorithm are image noise and connections between characters; moreover, the algorithm is based on an assumption that there is an indentation between character rows and characters would not be extracted if that is not the case. The main reason why this algorithm might be useful in practice is that printed characters satisfy the main assumptions: indentation between lines exists and characters are not connected. Furthermore, as mentioned in section 3.4.1, I assume that the algorithm will be used on a robot that is treated as a student, so the images that should be recognized can be treated as images of a blackboard, where characters often satisfy the main assumptions.

3.5 Conclusion

In this part, I presented computer simulations and visualizations of few algorithms that are useful for various problems in artificial intelligence and robotics. More precisely, I discussed a simple simulation of the A* algorithm in section 3.2, section 3.3 was dedicated to a neural network for recognizing handwritten digits, and I explained an algorithm for extracting characters from an image in section 3.4. Even though these simulations were just toy examples based on many assumptions, they helped me develop good intuition about the algorithms, pointing me to several aspects that

need to be taken into consideration in practical applications. In the next part of the thesis, I apply some of the algorithms to a Lego Mindstorms NXT 2.0 robot.

4 Robotic Simulations

4.1 Introduction

Computer simulations are useful tool for better understanding of the simulated algorithms, but they become relevant only when applied to real systems. In my project, the real system was a robot built with the Lego Mindstorms NXT 2.0 kit. Using this robotic kit, I made a robot that can find goal positions in a discrete world and localize itself within that same world.

Working with a robot, even a relatively simple one as the NXT, presents many challenges that are not encountered when working with computer simulations; moreover, many design decisions and assumptions have to be made in order to produce desirable results with the algorithms used. Therefore, this entire part of the paper discusses those design decisions and the limitations that they pose on the robot.

This section is organized as follows: section 4.2 discusses the technology items used in the project, giving a brief description of the NXT 2.0 kit and the software architecture on which the robot is based; section 4.3 discusses numerous applications of Mindstorms; finally, section 4.4 is where my work, design decisions, and assumptions are explained.

4.2 Technology Items and Software Architecture Used in the Project

This section describes the technology items used in my project and the software architecture on which the robot is based, such that section 4.2.1 gives some technical details regarding Lego Mindstorms, section 4.2.2 discusses the additional items that I was using in order to go beyond simple programs for the NXT, and section 4.2.3 explains the software and communication architecture on which the robotic simulations are based.

4.2.1 Lego Mindstorms NXT 2.0

The robot used in this project is built with the Lego Mindstorms NXT 2.0 kit; therefore, this section introduces the robot and some of the technical details important for the later discussions. The reader is advised to consult Hansen (2009) for a more detailed discussion about the kit.

Lego Mindstorms NXT 2.0 is a simple robotics kit that includes an NXT brick, sensors, motors, and Lego Technic elements. The NXT is a small computer brick that can process programs, communicate with sensors, motors, or a computer. The brick includes four sensor ports, three motor ports, and one USB port, also supporting bluetooth communication.

The default kit comes with three servo motors and four sensors: an ultrasonic sensor, a color sensor, and two touch sensors; these are briefly explained below:

- Each *motor* includes an encoder that allows measuring the motor's rotations; these encoders can measure 360 ticks per one revolution of a motor.
- The *ultrasonic sensor* can be used for measuring distances to obstacles. The sensor works by sending sound waves that will reflect from obstacles, such that distances are calculated by measuring the time it takes for a sound wave to come back to the sensor.
- The *color sensor* can recognize colors and measure light intensity in the environment.
- The *touch sensor* can be thought of as a button that works like a boolean switch: it gives a result *true* when the sensor is pressed and *false* otherwise.

The Lego Technic elements that come with the Mindstorms kit allow building different shapes of robots with different sensor configurations. In this project, I was working with a model of a humanoid robot, popularly called Alpha Rex (figure 22):

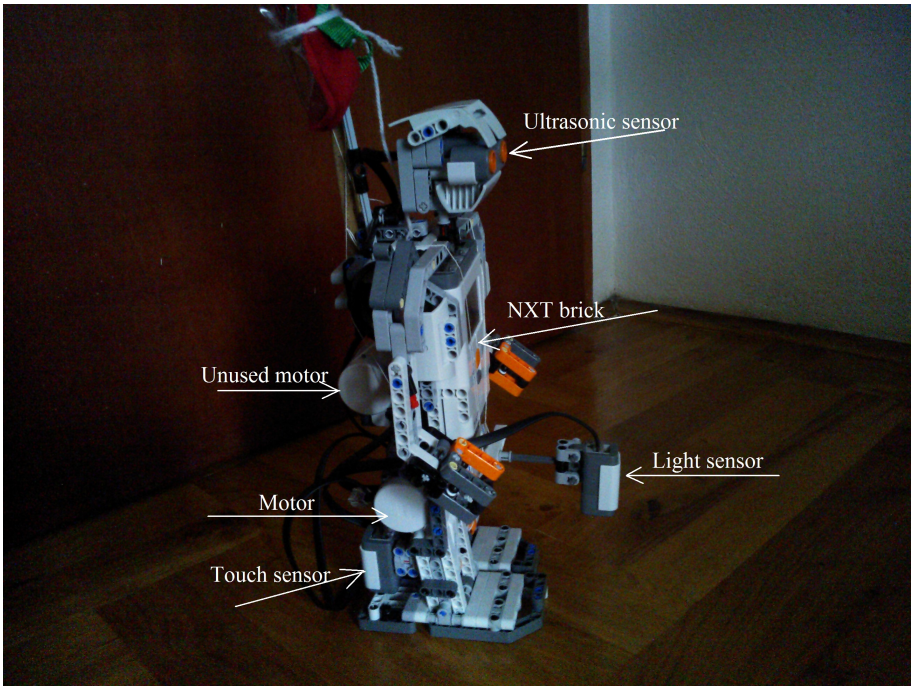

Figure 22: Description of the Alpha Rex robot

As can be seen on figure 22, the Alpha Rex model uses the brick as a torso, has one motor and one touch sensor per leg, and uses the ultrasonic sensor as a head. The color sensor was initially placed on one of the hands; however, I changed its position so that I can use it for scanning the color in front of the robot. Notice that one of the motors that comes with the kit is only mounted on this model, but is not practically used.

4.2.2 Additional Technology Items Used

The sensors that come with the default NXT kit are good enough for experimenting with simple programs; however, these sensors alone do not allow making more serious applications. For example, a robot that navigates the environment or wants to localize itself should at least have a compass that will be used for measuring the heading direction, a robot that wants to scan the environment might need to use a camera, and so forth. In order to give all these capabilities to my robot, I was using an LG E900 Optimus 7 device with Windows Phone 7.1 operating system, which has built-in

compass, accelerometer, GPS receiver, and a camera.

To make use of the phone, I added a wooden stick at the back of the robot and hanged a small bag on it; the bag is used for storing the phone while the robot is moving. The stick is relatively tall in order to keep the phone at a certain distance from the magnetic fields produced by the NXT brick and the robot's motors. The robot including the stick and the bag is shown in figure 23:

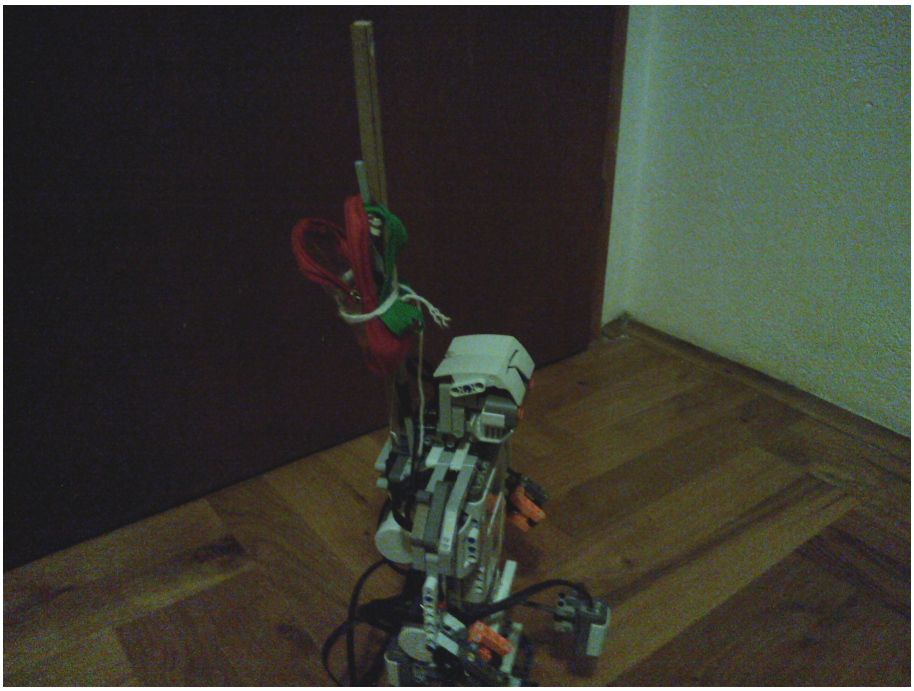

Figure 23: The robot including the wooden stick and the phone bag

4.2.3 Software Architecture Used by the Robot

The software architecture used in this part of the project makes a PC act like a master that receives data from sensor readings and sends appropriate commands to the robot. This architecture is visualized in figure 24:

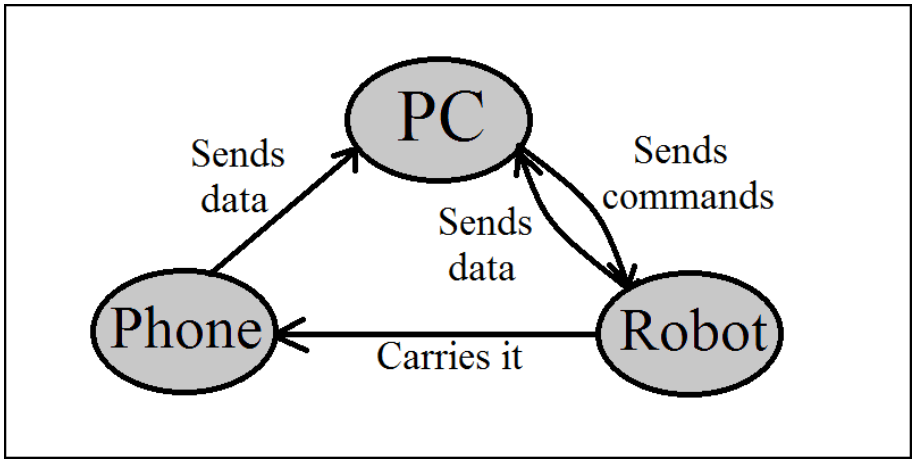

Figure 24: Software architecture used in the project

To be more precise, the PC hosts a *Windows Communication Foundation (WCF)* service that is invoked by a *Windows Phone (WP)* application; this service uses the *MindSqualls* library to communicate with the robot and either sends appropriate movement commands to the robot or polls the robot's sensors, depending on the request by the phone application. After receiving commands from the service, the robot executes them and the operation continues as a loop. This more precise version of the architecture is given in figure 25:

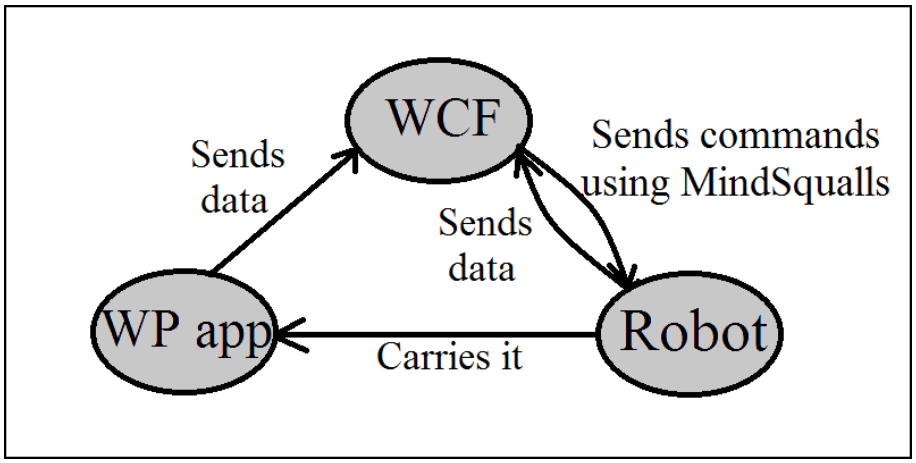

Figure 25: Software architecture used in the project, including frameworks

4.3 Related Work

Due to the relatively low cost (about $300), the Lego Mindstorms NXT kit is used for educational purposes quite often. McNally, Klassner, and Continanza (2007) explain how they use Lego Mindstorms in their robotics and artificial intelligence courses, allowing their students to try out different concepts that can be encountered in robotics; in this paper, they focus on the use of Lego Mindstorms for exploring localization and mapping. Akin, Meriçli, Meriçli, and Doğrultan (2009) also discuss an undergraduate course in which they used Lego Mindstorms for introducing their students to the principles and challenges of mobile robotics, reporting student satisfaction and good results from the course. Leitão, Gonçalves, and Barbosa (2005) mention yet another robotics course that made use of Lego Mindstorms and allowed students to learn by practically applying concepts to a robot.

Lego Mindstorms are not used only in education, but for other research as well. Oliveira, Silva, Lira, and Reis (2009) discuss the use of this robotics kit for building environmental maps, comparing two different approaches for achieving this purpose: one using a compass sensor and one without it; their quite promising results demonstrate that the NXT kit can be used for solving complex problems in robotics.

4.4 Robotic Simulations

This section is dedicated to the programs that I was writing for the robot, giving a detailed explanation of the models used, the assumptions on which they are based, as well as the results obtained. More precisely, section 4.4.1 talks about the odometry model that I used for the robot, section 4.4.2 describes the implementation of the A* algorithm for the robot, allowing it to find goal positions in a predefined world, section 4.4.3 discusses my particle filter implementation for localizing the robot, and section 4.4.4 mentions the approach for extracting and recognizing characters.

Note that the programs explained in this part were written using the architecture described

in section 4.2.3.

4.4.1 Estimating Position Change of the Robot

Before making the robot move in the environment, I had to choose a model for estimating the change in position and orientation of the robot; the problem of estimating position and orientation change is known as estimating *odometry*. The model of the robot shown in figure 22 is a humanoid, so its accurate description is that of a biped (see Siegwart, Nourbakhsh, and Scaramuzza (2011) for a more detailed discussion of bipedal robots); however, I decided to use a simpler model of a robot: that of a *differential drive robot*.

A differential drive robot is modeled as one that has two wheels, such that the speed at which they are rotating determines the direction in which the robot will be moving. Differential drive robots are not only very simple to analyze, but the odometry equations are also well-known in the robotics community; that is one of the main reasons why I decided to model the robot as a differential drive, even thought this model is not entirely accurate.

The odometry equations for a differential drive robot are given below and can be found in Borenstein, Everett, and Feng (1996, p. 20); they are included here for easier referencing later, as they are the basis of the odometry estimation model for the robot used in this project. Note that I changed some of the notation for convenience.

We can assume that the distances traveled by the left and right wheel of the differential drive are given by d_L and d_R respectively; moreover, we can denote the change in the number of ticks measured by the wheel encoders on the left and right wheel by *left_ticks*$_i$ and *right_ticks*$_i$ respectively. We will now have:

$$d_L = 2\pi R \frac{left\_ticks_i}{N} \tag{6}$$

$$d_R = 2\pi R \frac{right\_ticks_i}{N} \tag{7}$$

where *R* is the radius of the wheels (assuming that both wheels have the same radius) and *N* is the number of ticks per one revolution of a wheel. The change in orientation of the robot will be given by equation (8):

$$\theta_i = \theta_{i-1} + \frac{d_R - d_L}{l} \tag{8}$$

Using equations (6) and (7), we can estimate the distance traveled by the center of the robot as

$$d_C = \frac{d_R + d_L}{2} \tag{9}$$

where *l* is the distance between the left and the right wheel. Finally, using equation (9), we can find the change in position (assuming a two-dimensional plane) using the equations below:

$$x_i = x_{i-1} + d_C cos(\theta_i) \tag{10}$$

$$y_i = y_{i-1} + d_C sin(\theta_i) \tag{11}$$

In the case of the robot shown in figure 22, I measured the radius of a wheel to be 1.1cm and the distance between the two motors about 6.2cm.

4.4.2 Finding Goal Locations Using A*

Having the odometry model explained in the previous section, I added the first ability to my robot: that of finding desired goal locations using the A* algorithm. A simulation of A* was already explained in section 3.2; the code used for the robot is just a translated version of the code mentioned in that section to the C# language. The principle in which the robot is using A* for finding goals is explained below.

The world in which the robot is moving is a 5 by 5 two-dimensional grid of size 1m by 1m,

with each grid cell being a square of 20cm by 20cm (figure 26):

Figure 26: The robot's world

Because the robot starts at an initially known location in the world, the Windows Phone application is able to find a path to a desired goal location before the robot starts moving. After the A* planner finds a path to the goal location, the phone application starts making decisions about the direction in which the robot should move, taking the next grid cell in the path to the goal as current desired cell. As long as the robot's heading does not coincide with the direction of the desired cell in a global coordinate frame, a *PID (Proportional-Integral-Derivative)* regulator makes the robot turn; in order to determine these desired directions, I defined conventions as follows:

- Moving north corresponds to moving up in the grid; moving south corresponds to moving down (both of these correspond to changing a row index in the application).
- Moving east corresponds to moving right in the grid; moving west corresponds to moving left (both of these correspond to changing a column index in the application).

Once the robot has the desired direction, it starts moving forward and the application starts making odometry measurements; once these measurements show that the robot has moved more than 20cm in the x direction, a row or column index in the application is updated, depending on the direction in which the robot was moving. I am using only measurements in the x direction because I am practically rotating the robot's local coordinate system whenever I am changing direction with respect to the global coordinate frame. This means that I am not modifying the heading in the local coordinate frame of the robot when the robot is rotating, which is the same as saying that its local heading remains 0 radians, such that its initial local heading is 0 radians by convention; hence, equation (10) will produce a position update, but equation (11) will not.

Using this approach, I have two main sources of error:

1. The phone's compass is affected by external magnetic fields, especially by the robot's motors and brick; because of these undesirable effects, the robot might start moving in an incorrect direction.

2. While the robot is rotating, I am not calculating odometry measurements; however, the robot might deviate from the position that it had before it started turning, so I might make incorrect grid transitions. In other words, this means that the real position in the world and the logical position in the application might not be the same.

In any case, testing the robot's ability to find desired goals shows satisfying results in the 1m by 1m world and the error accumulation does not affect the results.

4.4.3 Robot Localization Using a Particle Filter

Robot localization is the problem of determining a robot's position within a known world given that the initial position is unknown. In other words, when faced with a localization problem, a robot does not know where it is in the world; therefore, it has to find a way to determine its

location. A good introduction to the robot localization problem is given by Siegwart, Nourbakhsh, and Scaramuzza (2011, pp. 265-266).

The localization problem can be solved using different approaches. Some of them, like the Kalman filter, approximate the results using linear Gaussian functions (Negenborn(2003)), but others, like the extended Kalman filter or particle filters, don't make such approximations: Georgiev and Allen (2004) experiment with an extended Kalman filter combined with a camera estimation model, allowing their robot to localize successfully outdoors; on the other hand, Dellaert, Fox, Burgard, and Thrun (1999), discuss a different probabilistic approach for localizing robots, known as Monte Carlo localization, that is based on taking samples from a desired probability distribution and using those samples for estimating the actual position of a robot. The latter approach is the basis of the *particle filter* that I used for localizing the NXT robot.

Particle filters determine the position of a robot using a set of particles, where the particles represent hypotheses about the robot's position. The basic idea behind the particle filter algorithm is the following (Thrun (2002)):

1. A robot does not know its initial position, so the particles are distributed uniformly throughout the world when the algorithms starts.

2. The algorithm proceeds by:

    - iteratively moving the robot and the particles,
    - making sensor measurements and calculating the probability of a measurement given some particle (called the particle weight), and
    - resampling from the distribution of particles such that the particles with higher weights have a higher chance to be resampled.

As the algorithm progresses, the robot's belief about the current location increases; eventually, the belief reaches a point where the robot is very certain about its position in the world.

The particle filter algorithm has two main requirements for being correct:

1. Either a known map of the environment or a known location of some reference points that can be sensed by a robot.

2. No symmetries in the environment as the belief of the robot will be concentrated on multiple locations at the same time.

I was testing the algorithm using the software architecture mentioned in section 4.2.3 and the same 1m by 1m environment that I used for testing the A* algorithm (section 4.4.2); this time, however, I modified the environment by adding some walls and obstacles, as shown in figure 27:

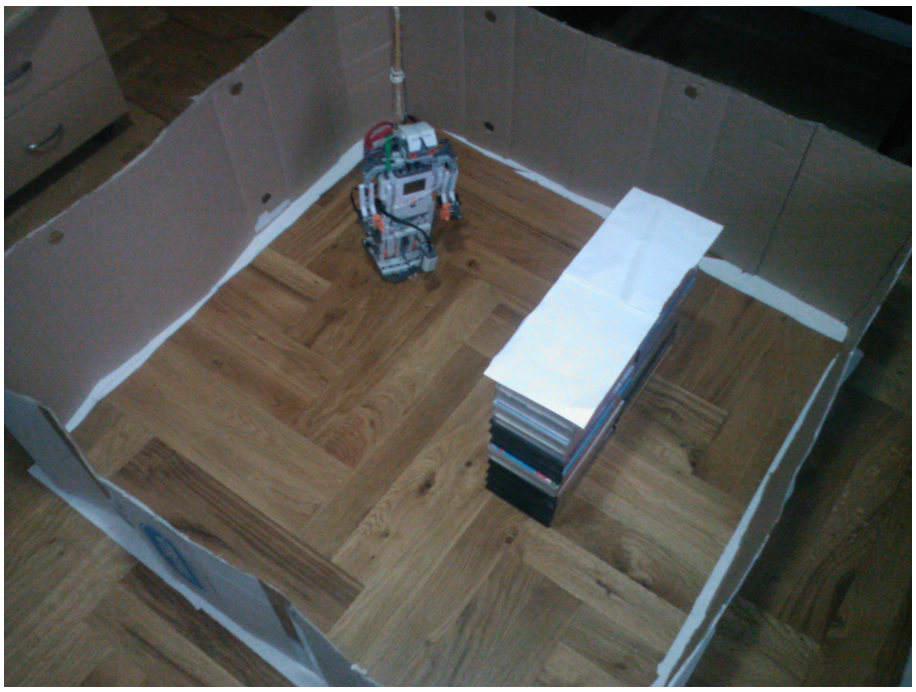

Figure 27: The environment for testing the particle filter algorithm

My implementation of the algorithm starts by randomly placing particles around the world. Once the initial configuration of particles is constructed, the robot starts moving in one of four directions: north, east, south, or west; the direction of movement is determined using the compass sensor that is integrated in the Windows Phone device that I used for testing. After reaching the

desired direction, the robot is measuring the distance to its nearest obstacle using the ultrasonic sensor; if the distance to the obstacle is less than a specified threshold, the robot starts turning into another direction and continues this turning-measuring distance loop as long as there is an obstacle in front of it.

The movement of the robot starts when the robot's direction does not have an obstacle whose distance is less than the threshold. After performing the movement, the positions of all particles are also updated using the following convention:

- The x position of the particles is increasing if the robot was moving east and it is decreasing if the robot was moving west.
- The y position of the particles is increasing if the robot was moving north and it is decreasing if the robot was moving south.

After updating the positions of the particles, the robot is measuring the distance to the nearest obstacle; the weights of the particles are then updated by measuring the distance to their nearest obstacle and calculating the probability that the two measurements are the same; in this case, I am using a Gaussian (normal) distribution centered at the actual measurement performed by the robot and variance whose values were varied during the testing process.

Once the particle weights are updated, I am resampling the particles with replacement. The resampling process starts by creating a distribution of particles; in this distribution, each particle is represented proportionally to its weight: the particles that have higher weights are represented more times and the particles that have lower weights are represented less (potentially zero) times. In order to avoid selection bias when resampling particles and to allow the robot to recover from potentially incorrect measurements, I am still randomly placing a small number of particles around the world during the resampling step.

In my simple world, the particle filter algorithm is able to correctly localize the robot after at most four iterations. Sample data obtained during each iteration of the algorithm and showing

the x and y positions of the particles are shown in figures 28-32:

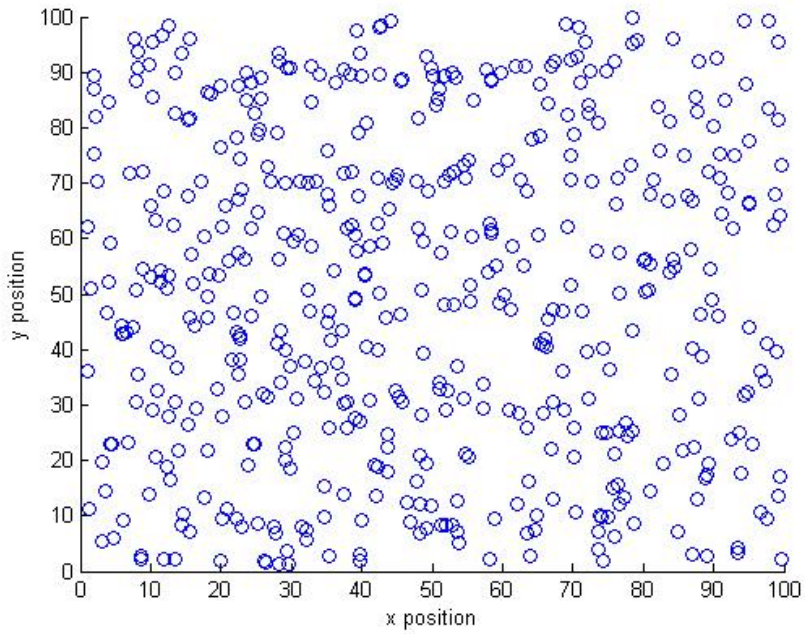

Figure 28: The initial configuration of particles in the world

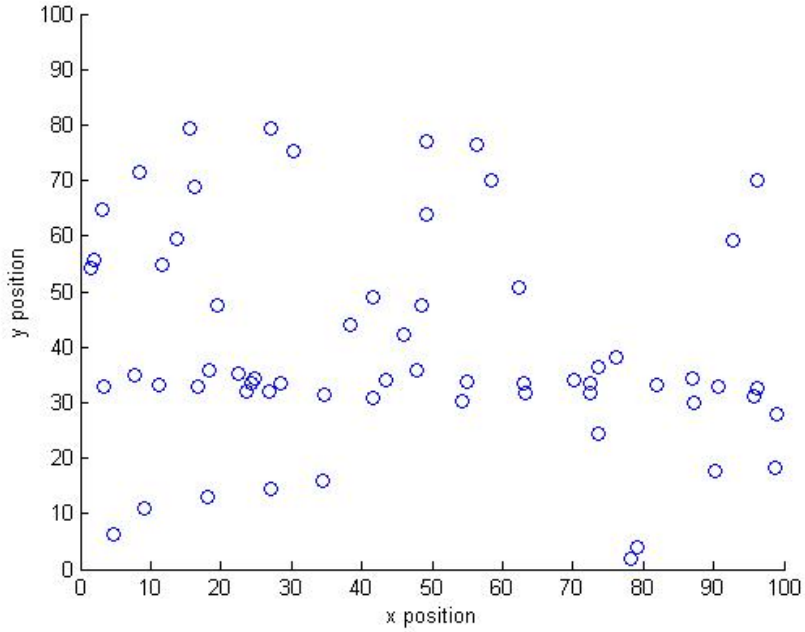

Figure 29: Distribution of particles after one iteration

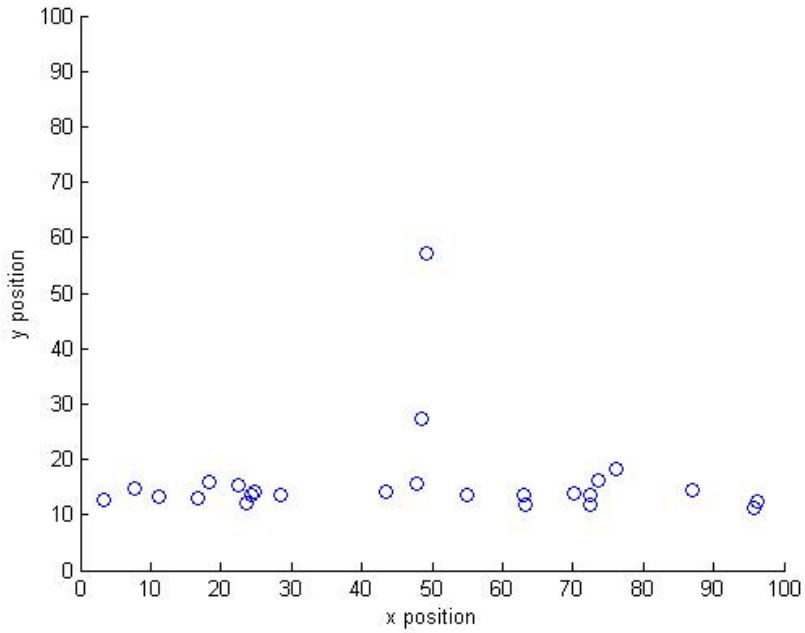

Figure 30: Particle distribution after two iterations

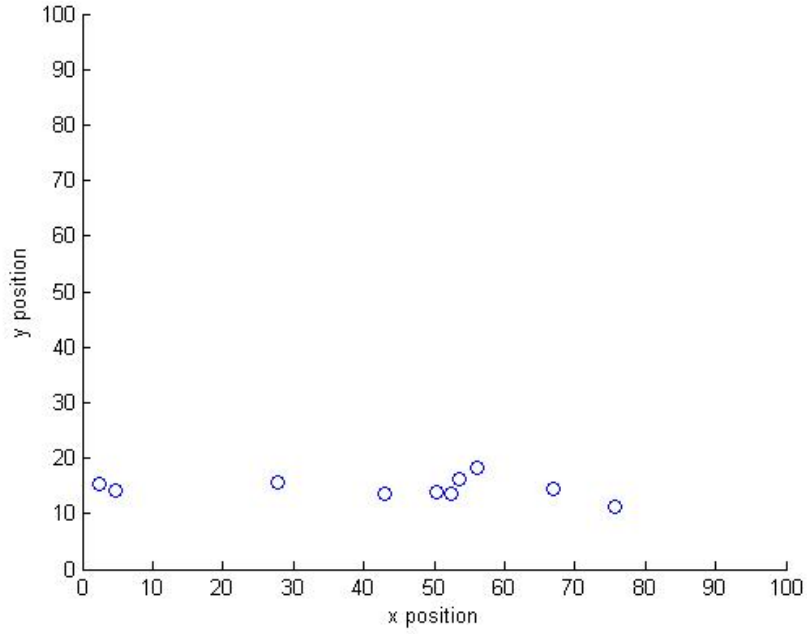

Figure 31: Distribution of particles after the third iteration

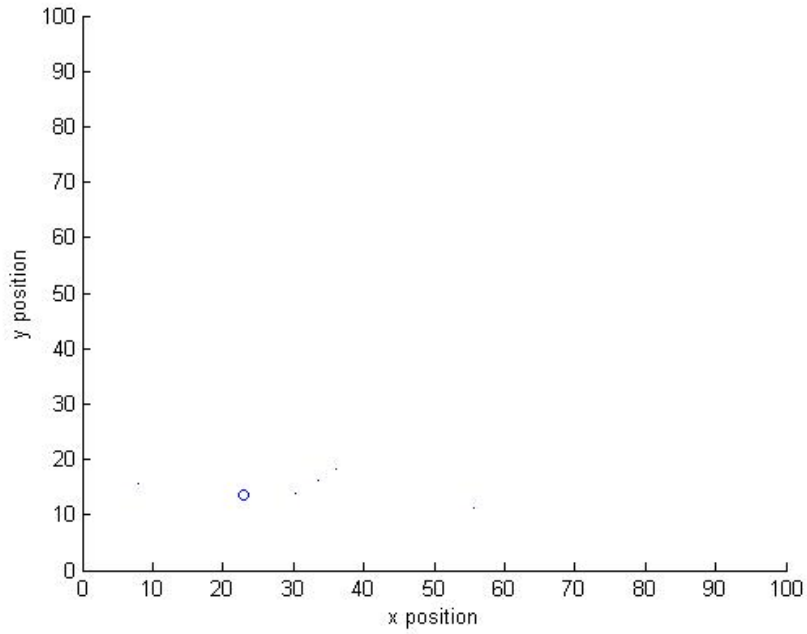

Figure 32: Particle distribution after four iterations - the robot succeeded in localizing itself

The figures above were generated after placing the robot in the middle of the environment facing south. As figure 28 shows, the robot starts being totally uncertain about its location in the world; however, its belief changes as the algorithm progresses and it finally reaches a point where the robot is completely certain about its position in the world; this situation can be seen in figure 32.

4.4.4 Solving Simple Algebra Problems by Recognizing and Classifying Characters

As I already mentioned in the Preface to this document, the final capability that I wanted to add to my robot is that of extracting and recognizing numbers and some mathematical signs in an image. A simple process for extracting characters was explained in section 3.4 and a process for recognizing the extracted characters was covered in section 3.3, so this final capability of the robot required merging these two separate modules into one program.

As mentioned in section 3.3, my neural network used in the simulation was only able to recognize numbers, but not mathematical signs. In order to increase the network's capability, I re-trained the network with the Semeion dataset supplemented with more than 100 samples of hand-written plus and minus signs. The sample signs included in the dataset are given in the following two figures:

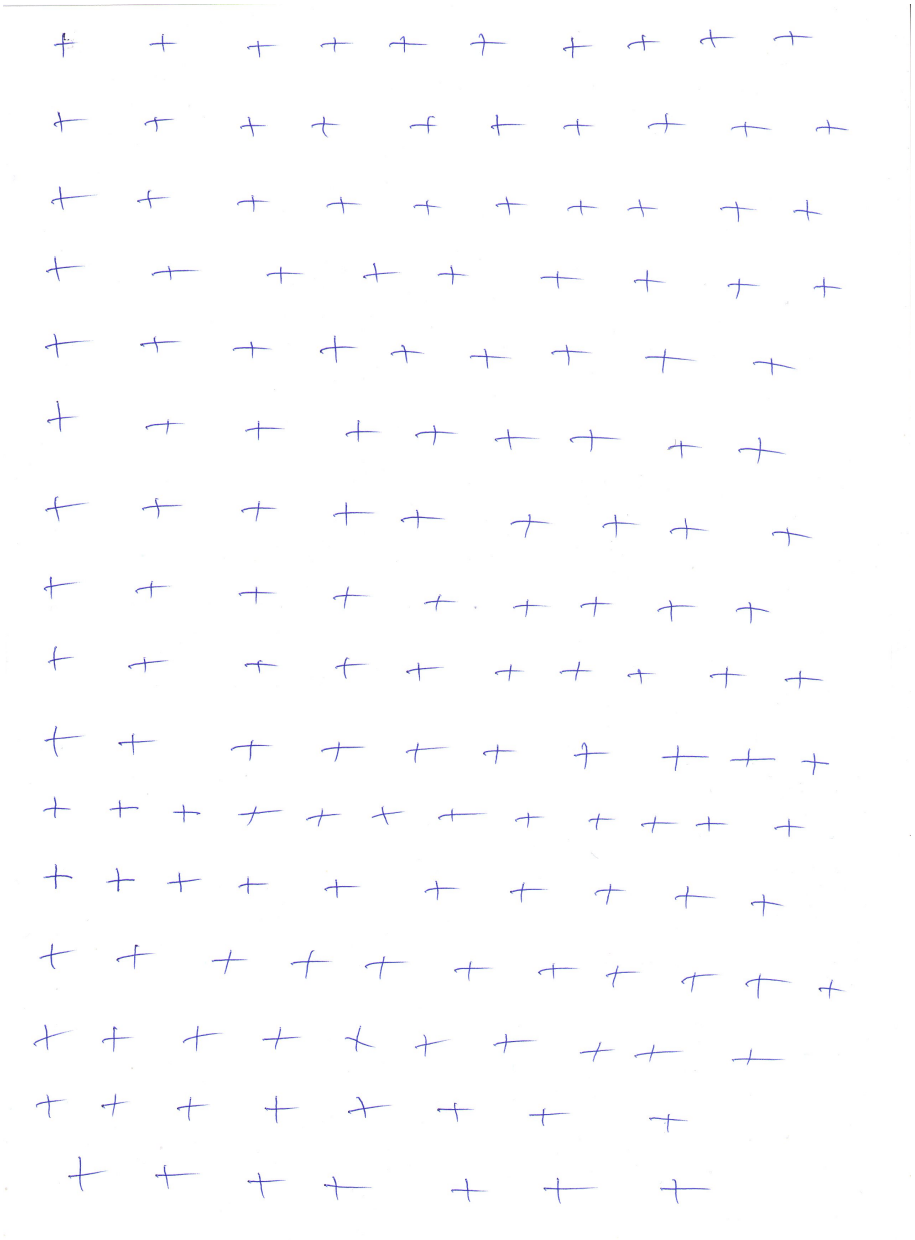

Figure 33: Plus signs used for training

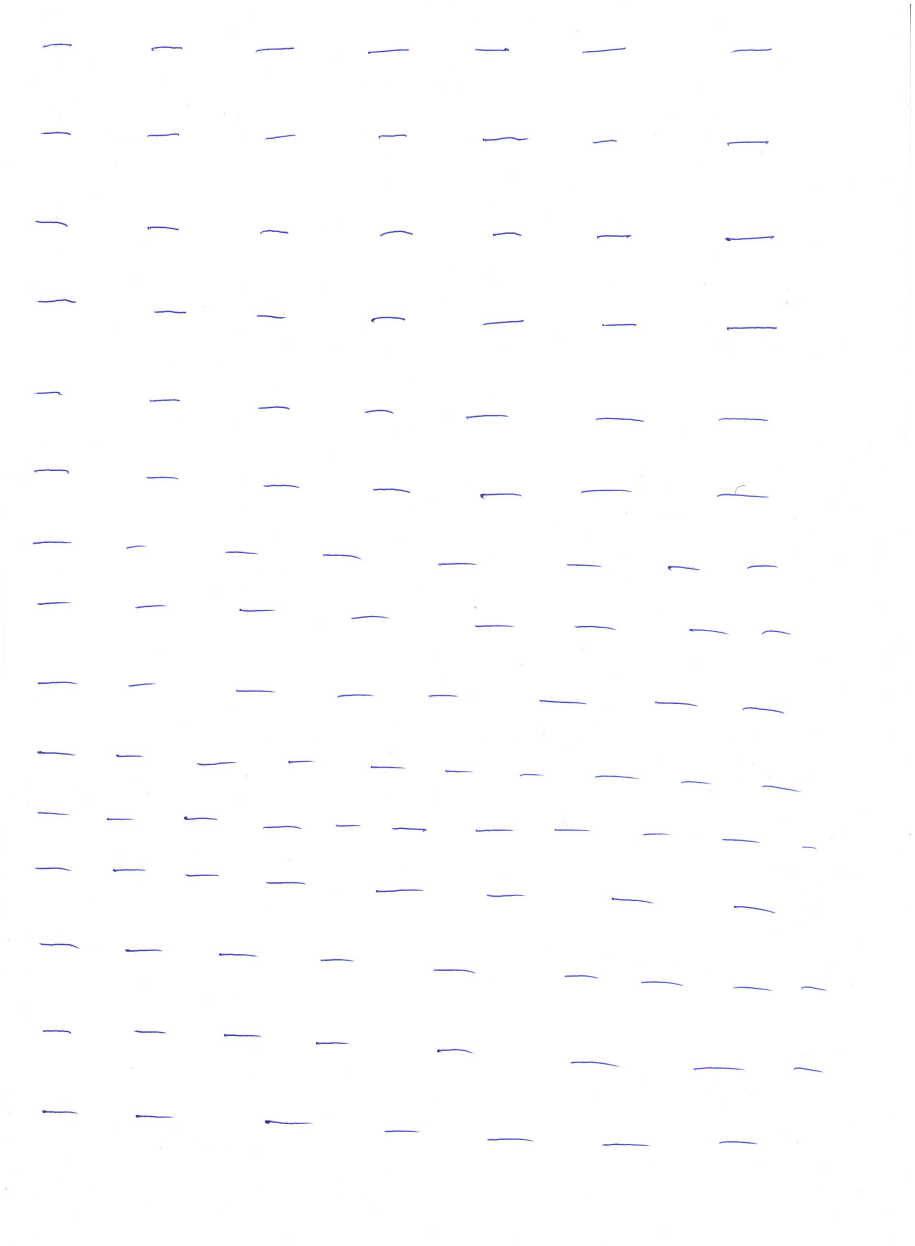

Figure 34: Minus signs used for training

After retraining the network, I created wrapper classes in managed C++ in order to make the previously written code usable for a Windows Phone application. These wrapper classes create dynamic libraries of both the character extraction and the character recognition module; the libraries are then referenced in the phone application.

The process used by this combined module is the following: the phone used for testing takes pictures at a predefined interval; when a picture is taken, it is saved on the PC and a function for extracting characters from the picture is called. Once the characters are extracted from the image, a function for recognizing them is invoked. The recognized characters are then returned and a result of the simple mathematical operation contained in the image is displayed on the phone's screen.

A sample image used for testing the process is given in figure 35:

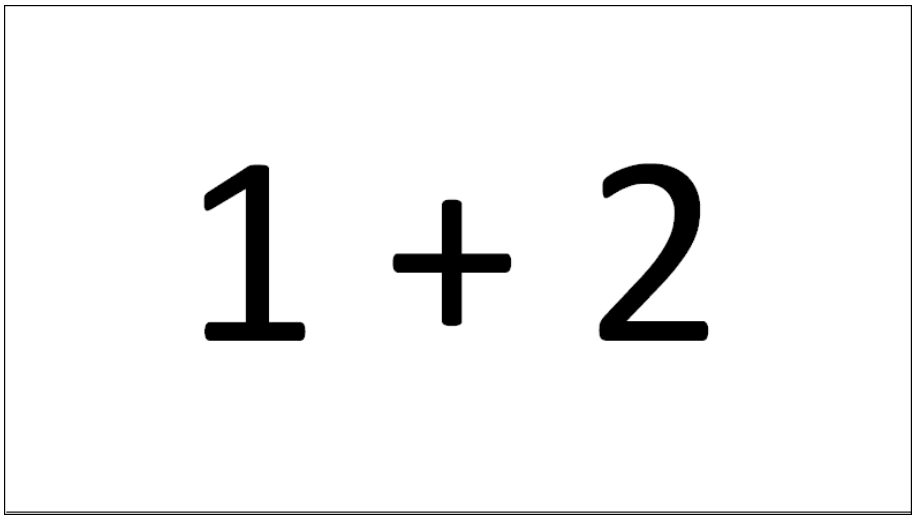

Figure 35: Sample image used for testing the extraction/recognition process

As the discussion above implies, this approach does not use the Lego robot for any of the operations. My initial plan was to play sound files of the recognized numbers after the extraction/recognition process directly from the NXT; however, I was not able to save more than five of these files to the NXT brick because of its memory limitations, so I abandoned this idea completely.

5 Limitations and Future Work

5.1 Introduction

The work discussed in sections 3 and 4 was my attempt to examine small parts of artificial intelligence and machine learning and their application to robotic systems. As such, this attempt was both successful and rewarding, because I was able to develop intuitive understanding of the algorithms that I developed and think about the advantages and limitations of these algorithms, but was also able to reason about their potential applications and improvements.

In this section, I will mention the most important limitations of my aproach and the technology items that I was using in the project in subsection 5.2 and will discuss some ideas for potential future work in subsection 5.3.

5.2 Limitations

The work discussed in this document has several limitations that have to be acknowledged:

1. The Lego Mindstorms NXT 2.0 that was used is a relatively low-cost robot which makes it suitable for investigating robotics; however, this robot has very few sensors, which makes it inappropriate for complex and realistic applications. Moreover, the humanoid model that I was working with makes use of only two motors, so it is not able to perform complex movements and tasks.

   One of the reasons why I used a Windows Phone device that has additional sensors was to make the robot capable of determining its heading direction using the phone's compass and taking pictures of the world with the phone's camera; these sensors, however, are not enough for complex tasks that autonomous systems have to demonstrate in the real world. A more realistic autonomous robot would at least need to have additional and more precise distance

sensors, better cameras for scanning the world, and additional motors for more realistic and accurate movements.

2. The software architecture described in section 4.2.3 depends on a present WiFi connection between the Windows Phone device and the PC; if there is no such connection, the phone application is not able to make requests to the WCF service and is therefore not able to control the robot.

3. The character extraction algorithm explained in section 3.4 is only able to extract characters that have clear vertical and horizontal boundaries between them; this, however, is not true for most handwritten text and tilted characters. The proposed algorithm therefore has limited applicability; the severity of this limitation can be ignored though, as the sole purpose of this approach was to demonstrate a concept.

4. The neural network used for recognizing characters is not able to distinguish characters from other objects in an image; hence, other objects will force the network to make an incorrect classification, such that everything will be classified as either a number, a plus, or a minus. This limitation, however, can be neglected just as the previous one, because the purpose of the network was to demonstrate another concept, not to solve a complete problem.

5. The A* algorithm, covered in section 4.4.2, and the particle filter, explained in section 4.4.3, were tested in a small 1m by 1m grid; I did not have an opportunity to test them in a bigger environment because my working ”lab” was my 3m by 3m room. As a result, I am not able to say whether my implementations generalize to bigger environments or not.

5.3 Potential Future Work

My work in the field of artificial intelligence, machine learning, and robotics, will not stop with this undergraduate project. In particular, I hope that I will be able to continue my studies on a graduate

level, where I will have an opportunity to work on more complex and realistic research projects. Moreover, I will certainly continue examining some of the algorithms explained in this document, such that I will try the particle filter and A* algorithms in a bigger environment when I have an opportunity for that and will try to compare their performance with that of other algorithms for localization and pathfinding. The algorithm for extracting characters can also be enhanced with existing edge detection and feature extraction techniques, and I might try to do that in future projects. The neural network that I currently have is simple and its performance can be improved by adding more hidden layers and more output nodes for recognizing objects other than numbers and mathematical signs. Finally, the character extraction and recognition process might possibly be improved by considering more complex neural network architectures that are able to recognize characters without extracting them previously.

6 Summary

This document describes my work in my final undergraduate project and represents a modest attempt at examining the applications of machine learning and artificial intelligence to robotics. In other words, simulations of a pathfinding algorithm, a character extraction algorithm, and a classification algorithm were presented in part three of the document, and part four was concerned with presenting applications to a Lego Mindstorms NXT 2.0 robot. The final outcome of the project is a robot that is able to localize itself, find a goal location within a known environment, and extract and classify numbers and some mathematical signs that are contained in an image.

Working on this project was both challenge and pleasure. First of all, this was the first time I had an opporunity to work with a robotic system and face many problems that autonomous machines have to overcome: the necessity for a good robot design and precise motion, the lack of sensors and actuators, the uncertainty present in the world where the robot exists and operates, and so forth. Second of all, I was able to experience the excitement present when a robot that I programmed myself was able to operate autonomously and perform simple tasks in the environment, such as finding a goal position and localization. Finally, this project is a source of many new ideas and potential applications that I might work on in future.

7 References

Adebiyi A. A., Ayo C. K., Adebiyi M. O., & Otokiti S. O. (2012). Stock Price Prediction using Neural Network with Hybridized Market Indicators. *Journal of Emerging Trends in Computing and Information Sciences 3*(1), 1-9.

Akin H. L., Meriçli Ç., Meriçli T., & Doğrultan E. (2009). Introduction to Autonomous Robotics with Lego Mindstorms. Proceedings of the *Twelfth International Conference on Climbing and Walking Robots and the Support Technologies for Mobile Machines (CLAWAR 2009)*, 733-740.

Anguelov D., Koller D., Parker E., & Thrun S. (2004). Detecting and modeling doors with mobile robots. Proceedings of *IEEE International Conference on Robotics and Automation*, 3777-3784.

Borenstein J., Everett H. R., & Feng L. (1996). *Sensors and Methods for Mobile Robot Positioning*. The University of Michigan.

Chajewska U., Koller D., & Ormoneit D. (2001). Learning an Agent's Utility Function by Observing Behavior. Proceedings of the *Eighteenth International Conference on Machine Learning*, 35-42.

Coates A., & Ng A. Y. (2010). Multi-Camera Object Detection for Robotics. Proceedings of *International Conference on Robotics and Automation*.

Cui S., Wang H., & Yang L. (2012). A Simulation Study of A-star Algorithm for Robot Path Planning. *16th International Conference on Mechatronics Technology*.

Dellaert F., Fox D., Burgard W., & Thrun S. (1999). Monte Carlo Localization for Mobile Robots. *IEEE International Conference on Robotics and Automation (ICRA99)*.

Garris M. D., Wilkinson R. A., & Wilson C. L. (1991). Methods for Enhancing Neural Network Handwritten Character Recognition. *International Joint Conference on Neural Networks, Volume I*, 695-700.

Georgiev A., & Allen P. K. (2004). Localization Methods for a Mobile Robot in Urban Environments. *IEEE Transactions on Robotics 20*(5), 851 - 864.

Gürel U., Parlaktuna O., & Kayir H. E. (2009). Agent-based Route Planning for a Mobile Robot. *5_{th} International Advanced Technologies Symposium (IATS'09)*.

Han J., Kamber M., & Pei J. (2012). *Data Mining Concepts and Techniques* (3rd edition). Waltham, Massachusetts: Elsevier.

Hansen J. C. (2009). *Lego Mindstorms NXT Power Programming: Robotics in C* (2nd edition). Winnipeg, Manitoba: Variant Press.

Jackel L. D., Battista M. Y., Ben J., Bromley J., Burges C. J. C., Baird H. S., ... Zuraw C. R. (1995). Neural-Net Applications in Character Recognition and Document Analysis, *Neural-Net Applications in Telecommunications*. Kluwer Academic Publishers.

Klingbeil E., Carpenter B., Russakovsky O., & Ng A. Y. (2010). Autonomous Operation of Novel Elevators for Robot Navigation. Proceedings of the *IEEE International Conference on Robotics and Automation*, 751-758.

Knerr S., Personnaz L., & Dreyfus G. (1992). Handwritten Digit Recognition by Neural Networks with Single-Layer Training. *IEEE Transactions on Neural Networks 3*(6), 962-968.

Kolanoski H. (1995). Application of artificial neural networks in particle physics. *Nuclear Instruments and Methods in Physics Research A 367*(1-3), 14-20.

Le Cun Y., Boser B., Denker J. S., Henderson D., Howard R. E., Hubbard W., & Jackel L. D. (1990). Handwritten digit recognition with a back-propagation network. *Advances in neural information processing 2* (396-404). San Francisco, CA: Morgan Kaufmann Publishers Inc.

Leitão P., Gonçalves J., & Barbosa J. (2005). Learning Mobile Robotics Using Lego Mindstorms. *9th Spanish Portuguese Congress on Electrical Engineering*.

Masudur Rahman Al-Arif S.M., Iftekharul Ferdous A. H. M., & Nijami S. H. (2012). Comparative Study of Different Path Planning Algorithms: A Water based Rescue

System. *International Journal of Computer Applications 39*(5), 25-29.

McNally M., Klassner F., & Continanza C. (2007). Exploiting MindStorms NXT: Mapping and Localization for the AI Course. *FLAIRS Conference 2007*, 315-320.

Millington I., & Funge J. (2009). *Artificial Intelligence for Games*. (2nd edition). Burlington, MA: Morgan Kaufmann Publishers.

Mitchell T. M. (1997). *Machine Learning*. McGraw Hill Science/Engineering/Math.

Murphy R. R. (2000). *Introduction to AI Robotics*. Cambridge, MA: The MIT Press.

Negenborn R. (2003). *Robot Localization and Kalman Filters: On finding your position in a noisy world* (Master's thesis).

Oliveira G., Silva R., Lira T., & Reis L. P. (2009). Environment Mapping using the Lego Mindstorms NXT and leJOS NXJ. *Fourteenth Portuguese Conference on Artificial Intelligence (EPIA)*.

Patnaik A., Anagnostou D. E., Mishra R. K., Christodoulou C. G., & Lyke J. C. (2004). Applications of Neural Networks in Wireless Communications. *IEEE Antennas and Propagation Magazine 46*(3), 130-137.

Popirlan C., & Dupac M. (2009). An Optimal Path Algorithm for Autonomous Searching Robots. *Annals of University of Craiova, Math. Comp. Sci. Ser. 36*(1), 37-48.

Rios L. H. O., & Chaimowicz L. (2010). A Survey and Classification of A* based Best-First Heuristic Search Algorithms. Proceedings of the *20th Brazilian conference on Advances in artificial intelligence*, 253-262.

Rossini P. (2000). Using Expert Systems and Artificial Intelligence For Real Estate Forecasting. *Sixth Annual Pacific-Rim Real Estate Society Conference*.

Russell S., & Norvig P. (2010). *Artificial Intelligence A Modern Approach*. Upper Saddle River, New Jersey: Pearson.

Shi Z., & He L. (2010). Application of Neural Networks in Medical Image Processing. Proceedings of the *Second International Symposium on Networking and Network*

Security, 23-26.

Siegwart R., Nourbakhsh I. R., & Scaramuzza D. (2011). *Introduction to Autonomous Mobile Robots* (2nd edition). Cambridge, MA: The MIT Press.

Somerville I. (2011). *Software Engineering* (9th ed). Boston, Massachusetts: Pearson.

Suri P. K., Walia E., & Verma A. (2010). Vehicle Number Plate Detection using Sobel Edge Detection Technique. *International Journal of Computer Science and Technology 1*(2), 179-182.

Thrun S. (2000). Probabilistic Algorithms in Robotics. *AI Magazine 21*(4), 93-109.

Thrun S. (2002). Particle Filters in Robotics. Proceedings of the *17th Annual Conference on Uncertainty in AI (UAI)*.

Thrun S., Montemerlo M., Dahlkamp H., Stavens D., Aron A., Diebel J., ... Mahoney P. (2006). Stanley: The Robot that Won the DARPA Grand Challenge. *Journal of Field Robotics 23*(9), 661-692.

Vallejos de Schatz C., & Schneider F. K. (2011). Intelligent and Expert Systems in Medicine - A Review. *XVIII Congreso Argentino de Bioingeniería SABI 2011 - VII Jornadas de Ingeniería Clínica*.

Yamaguchi T., Nakano Y., Maruyama M., Miyao H., & Hananoi T. (2003). Digit Classification on Signboards for Telephone Number Recognition. Proceedings of the *Seventh International Conference on Document Analysis and Recognition*, 359-363.

Yang C. -C., Prasher S. O., Landry J. -A., Ramaswamy H. S., & Ditommaso A. (2000). Application of artificial neural networks in image recognition and classification of crops and weeds. *Canadian Agricultural Engineering 42*(3), 147-152.

8 Index

A

*A**, 6-14, 18, 30, 37, 39, 40, 43, 53, 54

Activation function, 16-18

Admissible, 7

Agent, 1-3

Artificial intelligence, 1-4, 7, 30, 37, 52, 53, 55

B

Backpropagation, 16-20, 23

Biped, 38

C

C++, 9, 18, 20, 28, 50

Class, 9-11, 15, 16, 19, 28, 29, 50

Classification error, 16, 17

Computer vision, 1, 4

Connection weights, 16

Control theory, 1

Cost function, 7

D

Decision system, 3

Decision boundary, 16

Development, 1, 2

Differential drive, 38

Directed graph, 7, 15

Distribution, 42, 44, 46, 47

E

Edge weight, 7,

Euclidean distance, 9

Environment, 1, 2, 4, 6, 14, 23, 33, 34, 37, 38, 43, 48, 53-55

F

Feed-forward network, 15

G

Gaussian, 42, 44

Graph, 7, 15

H

Heap, 9, 10

Heuristic function, 7, 9, 14

Humanoid, 33, 38, 52

L

Lego Mindstorms NXT, 4, 5, 31-33, 37, 52, 55

Localization, 37, 41, 42, 54, 55

M

Machine learning, 1, 4, 15, 18, 20, 23, 52, 53, 55

Measurement, 41, 42, 44

MindSqualls, 36

N

Neural network, 6, 14-23, 27, 30, 48, 53, 54

Node, 7, 8, 10-12, 54

Normal distribution, 44

O

Odometry, 37-39, 41,

OpenGL, 9, 11

Optical character recognition, 24, 27

Overfitting, 22

P

Pathfinding, 6, 8, 9, 54, 55

Pattern, 15, 16, 19, 20

Particle filter, 37, 41-44, 53, 54

Perceptron, 16-18

PID (Proportional-Integral-Derivative), 40

Probability, 42, 44

S

Shortest path, 6-14

Sigmoid function, 18

Simulation, 4-10, 14-20, 23, 28, 30, 32, 37-51, 55

Supervised learning, 15

U

UCI machine learning repository, 20

UML (Unified modeling language), 6, 9-11, 19, 28

Utility, 3

V

Variable overflow, 17

Vector, 9, 15, 17, 20

Visualization, 4-7, 10, 14, 30

W

Windows Communication Foundation (WCF), 36, 53

Windows Phone (WP), 34, 36, 40, 43, 50, 52, 53