

Università degli Studi di Napoli Federico II

Facoltà di Ingegneria

Dipartimento di Informatica e Sistemistica



Delft University of Technology

Knowledge Based Systems Department



TESI DI LAUREA

**AGV IN DYNAMIC ENVIRONMENT: A PROPOSAL
FOR AN INTELLIGENT PARKING LOT**

RELATORE

Prof. Mario Vento

CANDIDATO

Luca Porzio

Matr. 45/4033

CORRELATORE

Prof. L.J.M. Rothkrantz

ANNO ACCADEMICO 2002/2003

Dedicated to my Grandfather Gennaro

Index

| | |
|-----------------------------------|-----------|
| <i>Acknowledgements</i> | <i>2</i> |
| Chapter 1: Overview | 4 |
| 1.1 Abstract | 5 |
| 1.2 Introduction | 6 |
| 1.3 Problem settings | 8 |
| 1.4 Lego simulation | 9 |
| 1.5 History of AGV | 11 |
| 1.5.1 Automated Highway System | 12 |
| 1.5.2 FROG | 14 |
| 1.5.3 Researches in progress | 15 |
| Chapter 2: Path Finding | 17 |
| 2.1 Introduction | 18 |
| 2.2 Some definition | 18 |
| 2.3 Best first algorithm | 20 |
| 2.4 Dijkstra's algorithm | 22 |
| 2.5 A* Algorithm | 24 |
| 2.6 Extended Dijkstra's algorithm | 27 |
| 2.7 The optimum | 30 |
| Chapter 3: The Garage | 33 |
| 3.1 Introduction | 34 |
| 3.2 AGV Movement | 34 |
| 3.2.1 Holonomic vehicle | 34 |
| 3.2.2 In practice | 37 |
| 3.3 Requirements | 38 |
| 3.4 Local or global computation | 39 |
| 3.5 The Lego robot | 41 |
| 3.6 What's on RCX's mind | 42 |
| Chapter 4: The Environment | 44 |
| 4.1 AGV positioning system | 45 |
| 4.1.1 Guidelines | 45 |
| 4.1.2 Landmark positioning system | 45 |

| | | |
|---------------------|---|-----------|
| 4.1.3 | Dead Reckoning system | 46 |
| 4.1.4 | Global Positioning System (GPS) | 46 |
| 4.1.5 | Local network positioning system | 47 |
| 4.2 | Car detection sensors | 48 |
| 4.3 | The simulated environment | 50 |
| 4.3.1 | Sensor in simulated environment | 50 |
| 4.4 | Considerations | 51 |
| Chapter 5: | Implementation | 52 |
| 5.1 | Programming environment | 53 |
| 5.2 | Requirements | 54 |
| 5.3 | Visual part | 55 |
| 5.3.1 | Visual component extensibility | 57 |
| 5.4 | The Graph class | 58 |
| 5.5 | Interfacing with the robot | 60 |
| 5.5.1 | Synchronization | 60 |
| 5.6 | Finally | 63 |
| Chapter 6: | Conclusion | 65 |
| 6.1 | Results | 66 |
| 6.2 | Problems and Further improvements | 69 |
| 6.3 | Conclusion | 71 |
| Bibliography | | 72 |
| • | About Lego Mindstorm | 73 |
| • | About path planning algorithms and implementations | 73 |
| • | About AGVs | 75 |
| • | About image manipulation routines and other car sensors | 79 |
| Appendixes | | 80 |
| • | Appendix A: Beyond the Subject | 81 |
| | Brief introduction to Lego | 81 |
| | The brain: RCX Brick | 82 |
| | The body: Sensors and actuators | 82 |
| | The communication: Infrared TX/RX | 83 |
| • | Appendix B: Tutorial | 84 |
| | The Robot | 84 |
| | The Environment | 85 |
| | The Application | 86 |
| | Section 1: Download | 87 |

| | |
|--|-----------|
| Section 2: Sensors Calibration | 88 |
| Section 3: Choose the parameters for the local sensors | 88 |
| Section 4: Path finding simulation | 90 |
| • Appendix C: The Source Code | 92 |

Acknowledgements

Firstly I would like to thank Prof. Leon Rothkrantz; he is the one who accepted me into his team and let me take part to this wonderful experience; his guidance and company has been a great experience both in a professional way and for personal life.

Secondly I would like to remember all the people in the KBS department who were kind, helpful and tolerated my presence for so long. Between all, Priam and Guillaume with whom I used to spend a lot of time.

Then I would like to thank all the people who made possible my journey that is Prof. Niccolò Rinaldi for his help and the staff of the International Exchange Office of both universities who were helpful and kind.

A special mention to my friend Biagio since he introduced me to the possibility of this thesis and also because if I didn't, I will lose one of my best friend.

I would like to thank my love Sabina for being next to me in any situation and supporting me during my studies; also I thank my brother Marco and my friend Giovanni for their moral support during my stay in the Netherlands.

Let me reserve some space for my parents which were the main sponsor of my work and for all the great love they have showed me so far.

Last but not least I would like to thank Prof. Mario Vento for hosting me at my home university and for his guidance over the last phases of my work.

Thank you all.

Chapter 1: Overview

1.1 Abstract

The main purpose of this project is trying to understand where current technology could lead us both in metaphorical and literal way. The subject of this thesis is to build up an **Automatic Guided Vehicle** (AGV from now on) and define its usability in real life environment and, obviously, its limits.

According to this, we chose to implement an AGV which we could simulate real environment with. Great care was paid in choosing the environment for it should be enough simple to be handled by the robot and at the same time enough complex to resemble real life. For our purpose we decided to simulate a dynamic maze, much similar to what a big parking lot is.

Actually the idea is to construct an AGV which is able to pick up drivers who just parked their car and bring them to a near big facility which could be an airport or a station. We will show later in this paper that the idea is not completely new; the Schipol airport near Amsterdam has an AGV called FROG which is able to do what we explained before; however this FROG AGV simply follow pre-established paths therefore it does not take into account any dynamic information and the main consequence of its routing algorithm is that it will blindly go in a congested road contributing itself to the jam in the parking lot.

Our goal in this research is to study and finally make a reliable proposal of an AGV which will move through the parking lot ‘reasonably’, avoiding jam or blocked road and safely transporting people to their goal maybe playing music meanwhile.

1.2 Introduction

The century we just passed few years ago has shown us what kind of wonderful miracle (and abominable things) humanity is capable of. One thing we want to point out now is the astonishing technical progress which characterized last century: we started the century with the light of an electric lamp and we finished it by exploring Mars surface.

Progress helps mankind to overcome many limits imposed by nature.

However sometimes progress is slowed down by many factors independent by the technical capabilities itself. It is for example the case of train guidance: although is technically possible to build automated driven trains, still we have human drivers because it is a *general assumption* that being driven by “Machines” is somewhat unreliable and discomforting. The field we study in this report is related to this problem: Automated Guided Vehicles (or AGV from now on).

As microprocessors and sensors continue to shrink in size and cost, the deployment of vehicle control systems has become technically and economically feasible. Vehicle automation programs around the world have demonstrated remarkable capabilities, such as cars that drive themselves along highways, based on such inputs as video images from on-board cameras and satellite-based positioning data. Despite of all these technical advances, many people still refuse the idea to be led by computers.

Lately we are beginning to see some changes in people attitude, handheld smart route computers, which suggest to the driver the best path, are quite common also some luxury class cars (e.g. BMW 700 series) mount a device called Adaptive Cruise Control which

behaves like a normal cruise control but is also capable of keeping safety distance between the vehicle and the one ahead.

Since we don't want these to be isolated episodes, in this report we propose a new AGV which is a step ahead into the purpose of bringing people near to AGVs. Our proposal is an Intelligent park where people can park their car and an AGV will come, pick them up and bring them to a near facility like an Airport or train station.



Figure: an aerial view of a parking lot in India

We observe that such an agent would have to face many problems: traffic situations and traffic jam avoidance, pedestrians and other cars safety between all; we observe also that these situations have been thoroughly studied in literature and mostly resemble metropolitan environment even if on a smaller scale. With this in mind this report aim at studying current technology and propose a real simulation of such an AGV.

Due to economical and time constraints our simulation has been conducted in a lab where we built a robot which is going to simulate our AGV and an adequate environment.

It is important to underline that we don't want to build the ultimate AGV, we just want to show that building an AGV for real life purposes is a feasible project and show a way to build it, so to finally have people who open their cars, seat and say "bring me to the nearest Italian restaurant".

1.3 Problem settings

The main objective of an AGV capable of moving through real environment condition is finding a path to the goal. We have many examples in literature about path finding algorithm. This report itself is mainly concerned about the path finding algorithm. Indeed, what we can find in literature are many examples about static path finding but the environment in which the AGV will drive through is not static. While our AGV is moving, many roads are being occupied by cars and many others are being left free. Even the robot itself will change the ‘state’ of a road simply passing on it.

For this purpose, the algorithm to be used should satisfy the following criteria:

1. Finding a good way to the goal (optimum would be welcome but not necessary)
2. Being able to handle dynamic changes
3. Real-time algorithm (i.e. low computational complexity)
4. Robustness

These constraints interfere with each other. An optimal solution is difficult to realize in real-time. This means that we must find a trade-off between optimum solution and computational time.

Furthermore, this AGV should be able to handle unexpected real life situation like lack of information (communication losses, broken sensors, etc.) or real time problems (unexpected objects on path, mechanical problems, etc.). A partial solution about some of these problems, together with a possible method to implement the control system for the AGV, will be shown throughout this report.

We must observe that the problem of AGVs is a well-known problem in literature and still mainly unresolved. On the other hand as stated before, the goal of this research is not to build the ‘ultimate’ AGV: our AGV is not going to travel in everyday-life situation but in a controlled environment such as a parking lot could be. This yields to the idea of building an ‘intelligent’ environment’ that could help our AGV in its basic actions. For this reason we can make some assumptions that will simplify our work: we will assume that our AGV will drive through a ‘friendly’ environment, i.e. the environment will be able to sense its own state and translate it into information our AGV (or a server in constant communication with it) will be able to handle and use to compute its path.

According to the previous assumption we must decide what kind of information we need to examine. Due to the variety of pathfinding algorithms available, the information needed could change from algorithm to algorithm. In this paper we will show many candidates for our pathfinding algorithm and by time in time we will analyze the type of information our environment should be able to examine.

1.4 Lego simulation

Engineer has a practical mind. No engineering project could really be considered finished if it is not tested under real world condition. For this purpose during this research a small robot and an environment has been built so to let us test our AGV under real conditions.

The materials used to assemble the robot are the famous Lego bricks, more precisely the 'Robotics Invention Kit' and 'Vision command kit' from Lego were used to build and program the robot.

In the Appendix section of this report, you can find a more detailed description about the components used to build the robot.

The Lego kit shows some interesting features. The controller on board is a Hitachi H8 microcontroller which supports multitask programming, the communication with the



Figure: Vision Command Kit

computer is wireless and uses an Infrared transceiver with a copyrighted protocol for exchanging data; the Controller can receive up to three sensor inputs and can control till three different actuators.

Available sensors shipped together with the Robotics Invention Kit are touch sensors and light sensors; other sensors are available on Lego shop such as Temperature sensors, torque sensor and pressure

sensor; more sophisticated sensors can be built and some third party brands produce, between other, proximity IR sensor. Available actuators include two torque Engines; by shopping you can find lamps and other kind of Actuators.

If we want to find a drawback into this kit it is certainly its limited numbers of sensors input and limited number of actuators that the controller can handle at the same time. On



Figure: Robotics Invention Kit

the other hand the best characteristic of this Kit is its modularity and ease of use: change any part of your robot is as easy as putting Lego pieces together!

1.5 History of AGV

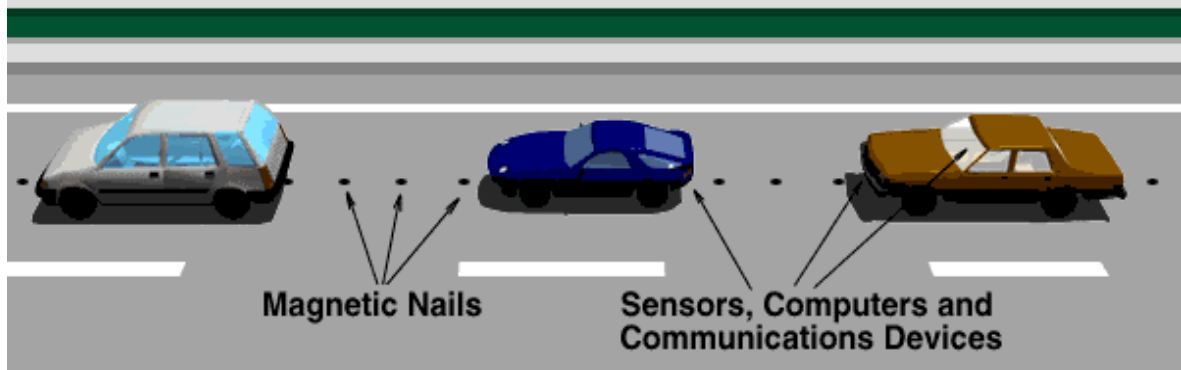
The use of automated transport systems goes back to the 1950s. The very first Automated Guided Vehicle was a converted tractor that followed a guide wire on the ceiling of a factory. By the spreading of this technology, the wire guide went from the ceiling to the floor so that it could easily be integrated in any environment. The technology on which this system was based is that these wires generate an electromagnetic field that can be measured by the vehicles. Magnetic field sensors and a servo control system ensure that the vehicle is capable of following the wire.

The second generation of unmanned vehicles also followed the induction wire along the straight sections, but was able to navigate freely around the bends. This capability meant that a great many difficulties could be overcome. The flexibility of the vehicles and the system increased as a result.

Nowadays many different systems have been developed to let AGVs drive through any environment and some of these systems are analyzed further in this report; let us show now some example and commercial solutions currently available.

1.5.1 Automated Highway System

An Automated Highway System (AHS) or Smart Roads, is an advanced Intelligent transportation system technology designed to provide for driverless cars on specific rights-of-way.



The AHS system places computers in cars. They read a passive roadway, and use radar and inter-car communications to make the cars organize themselves without the intervention of drivers. The principle is based on two structures which need to be changed respect to normal environment: roadway and car.

The roadway must have magnetized stainless-steel spikes driven one meter apart in its center; the car senses the spikes to measure its speed and locate the center of the lane. Further the spikes can have either magnetic north or magnetic south facing up. The roadway thus has small amounts of digital data describing interchanges, recommended speeds, etc.

The cars have power steering, and automatic speed controls and these are controlled by the on-board computer. Moreover the cars organize themselves into platoons of eight to twenty-five cars. The platoons drive themselves a meter apart, so that air resistance is

minimized. The distance between platoons is the conventional braking distance. If anything goes wrong, the maximum number of harmed cars should be one platoon.

The feasibility of such a project was demonstrated onto the Interstate 5 automated highway prototype located in San Diego, California.

Although on a larger scale, the AHS is based on the same idea which guides us through this report: the construction of an AGV in a friendly environment. The cars are instructed to follow guidelines dug into the floor and the basic actions the AGV can perform on its own are “keeping the center of the lane” (i.e. follow the guidelines) and “keeping a minimum safety distance between the vehicle ahead and itself”.

When the vehicle reaches the destination set by the driver, it will return the control to the driver.

We point out that no path planning is done (the vehicle keep going forward as long as the destination has been reached) and that the AHS are not suitable for mixed traffic (man driven cars are not allowed into AHS lanes).

Also the AHS suffers, commercially speaking, of the chicken-and-egg problem: no one will buy an AHS for his car if there are no highways supporting them and no highway will build an expensive AHS lane unless there are sufficient number of AHS vehicles supporting it.

Although the technical feasibility has been demonstrated, commercial applications of AHS are slowed down by many factors and as a matter of fact the project is still on its development stage.

With our project we aim at a prototype with path planning capabilities and mixed traffic capability. We will try to find a solution to the problems highlighted by the AHS.

1.5.2 FROG



The Schiphol Airport near Amsterdam presents a nice solution for its long-term clients: the ParkShuttle by FROG company. The ParkShuttle is an AGV which is much similar to what is described into this report; it is a bus which drives through a parking lot where drivers can park their

cars for long term travels. The ParkShuttle has stops where the driver can wait for it to come and then it will accompany them to a bus stop outside the parking where another bus (man driven) will carry them to the airport.

The ParkShuttle uses image sensors to recognize images on the floor by the mean of which it is able to recognize its position in the parking lot; it also uses odometers to compute the position with an increased precision. Although



The ParkShuttle's Route in Schiphol Airport parking lot

the Frog ParkShuttle is substantially much similar to our work, it presents some problems which we are trying to overcome with our proposal.

At first the ParkShuttle has very limited movements; its movement are preestablished, it follows always the same route and no effort is done to avoid obstacles. Secondly Parkshuttle lanes are forbidden to human driven cars yielding to expensive additional lanes for the FROG AGV. However the FROG ParkShuttle is a good attempt and first trial to make everyday AGVs come true and so it should be welcomed. The FROG company has also proposed the European Combined Terminal in Rotterdam that uses 54 automatic stocking cranes and 120 AGVs to automatically transport containers to and from ships which is also a remarkable result but always limited by the problems shown above.

1.5.3 Researches in progress

Other researches are still in progress. Mainly they aim at creating fully autonomous vehicles independent by the environment. We would like to mention the ALVINN project which trains a Neural Network with images coming from a camera and tries to mimic the behavior of a human driver (from large highways to off-road environment). Also another important research in progress is the "System for effective Assessment of the driver state and Vehicle control in Emergency situations" (SAVE) that is a program which aims at substituting the driver only in emergency situation (but no attempt is done for long term driving).

So far no standard has been reached and although the results are remarkable, it is our opinion that autonomous guidance is still far to be reliable and suitable for a large scale use or commercial purposes, therefore we are going to focus our attention on a currently feasible project.

Chapter 2: Path Finding

2.1 Introduction

A large variety of path finding algorithms can be found in literature and most of them have their own advantages and disadvantages.

One of the most important algorithm is the Dijkstra's algorithm. Edsger W. Dijkstra was one of the first to really make his mark in the field of path finding, by formalizing the problem and proposing his initial solution in 1959. This set the premise for a great number of successors, all of which improved the original performance. Some of these variations will be presented in this report. Here we just want to point out that the Dijkstra's algorithm aim at optimize one static parameter, which is distance in our case; no dynamic information is computed since the Dijkstra's algorithm does not care about time.

An improvement of Dijkstra's algorithm known as 'extended Dijkstra's algorithm' is shown later in this section which is able to use dynamic information to calculate an optimum path; there is however a major drawback: we need dynamic information about the maze i.e. for a real time problem we need a way to collect data and translate these into a likely future state of our maze. But we don't want to anticipate how things will going on.

2.2 Some definition

Before we can talk about pathfinding algorithms, we first define some basic terminology about graphs.

A **graph** G is a pair (V, E) , where V is a finite set and E is a binary relation over V . The set V is the set of nodes in the graph, while the set E is the set of edges in the graph. In an undirected graph, the edge set E consists of unordered pairs. In a directed graph, the edge set E consists of ordered pairs. Edges in a graph may be associated with weights, which can represent anything from physical distance between nodes to the carrying capacity of the edges.

A **path** p is a sequence of nodes $\langle v_1, v_2, v_3, v_4 \dots \rangle$, where $(v_i, v_{i+1}) \in E$. If the edges are associated with weights, then the **shortest path** from v_i to v_j is the path where the sum of all the edge weights in the sequence is the lowest of all possible paths from v_i to v_j (There may be multiple shortest paths between two nodes).

We notice here that pathfinding is a combinatorial problem, therefore the time needed to compute a solution increase exponentially according to the cardinality of the V and E sets. To decrease the time complexity of a graph spanning, we could help ourselves by using a function called a “Heuristic function”.

A **Heuristic function** $H(v_1, v_2)$ is a function able to estimate the distance (or any other measurable parameter) between the node v_1 and the node v_2 onto our graph. An **admissible heuristic** is a Heuristic function which never overestimates the distance between v_1 and v_2 where v_1 and v_2 belong to the V set; an **inadmissible heuristic** is a Heuristic function which overestimates the distance between v_1 and v_2 for at least one couple (v_1, v_2) of nodes in the V set.

In the following paragraphs we will introduce some of the most well-known pathfinding algorithm and try to show their main advantages and disadvantages.

2.3 *Best first algorithm*

Let's assume we are in Paris. If we are not too far from the historical centre, we can admire the beauty of the Eiffel tower hanging over all the buildings around. What if we want to reach it? We have two options: we can buy a map and plan a path to the tower or we can simply aim at 'what we see' and try to find a way.

This second method is known in literature as the Best first algorithm.

By formalizing it better, we assume we have a function $H(v_{\text{current}}, v_{\text{goal}})$ which is able to estimate the distance between our current position and the goal: this is the Heuristic.

Table 1: Best first algorithm pseudo code

Best first Algorithm Pseudo code

```

Set a node as goal,  $v_{\text{goal}}$ 
Set a node as start,  $v_{\text{start}}$ 
Set the cost of  $v_{\text{start}}$  equal to  $H(v_{\text{start}}, v_{\text{goal}})$ 
Set the cost of any other node to infinite
Create an Empty list called CLOSE
Set  $v_{\text{start}}$  as  $v_{\text{current}}$ 
While CLOSE doesn't contain the whole graph
{
    If  $v_{\text{current}}$  is the same as  $v_{\text{goal}}$  we have found the solution; break from the while loop

```

```

Generate a list of each  $v_{\text{successor}}$  connected to  $v_{\text{current}}$ 

For each  $v_{\text{successor}}$  of  $v_{\text{current}}$ 
{
    Set the cost of  $v_{\text{successor}}$  to  $H(v_{\text{successor}}, v_{\text{goal}})$ 

    If  $v_{\text{successor}}$  is in the list CLOSE but the node in the list is better, discard this successor
and continue

    Remove occurrences of  $v_{\text{successor}}$  from CLOSE

    Set the parent of  $v_{\text{successor}}$  to  $v_{\text{current}}$ 

    Add  $v_{\text{successor}}$  to the list OPEN
}

Add  $v_{\text{current}}$  to the list CLOSE
}

```

By using the heuristic as shown in the table we can find (If existing) a path to the goal. In our search for the Eiffel tower, the Heuristic covers the role of ‘What we see’ and usually, for pathfinding problem, this Heuristic represents the distance between our current position and the goal.

The path found by using this method is much intuitive (in most cases is what animals and humans do to find their way) and as a matter of fact is one of the fastest and easy algorithm known in literature; it presents however a major drawback: the path found is not guaranteed to be the optimum path, not even we can know whether it is near the optimum or not.

This algorithm is usually pretty fast compared to the other pathfinding algorithms and it is sometimes used as a lower bound term of comparison for time needed to find a path.

2.4 Dijkstra's algorithm

The Dijkstra's algorithm is as simple as powerful.

The **Dijkstra's algorithm** can find the shortest path from a starting node v_{start} to a goal node v_{goal} . In the table below, we can find a description of how this action is performed.

Table 2: Dijkstra's algorithm pseudo code

Dijkstra's Algorithm Pseudo code

```

Set a node as goal,  $v_{goal}$ 
Set a node as start,  $v_{start}$ 
Set the cost of  $v_{start}$  equal to 0
Set the cost of any other node to infinite
Create an empty list called OPEN
Create an Empty list called CLOSE
While the list OPEN is not empty
{
  Get the node off the open list with the lowest cost and call it  $V_{current}$ 
  If  $v_{current}$  is the same as  $v_{goal}$  we have found the solution; break from the while loop
  Generate a list of each  $v_{successor}$  connected to  $v_{current}$ 
  For each  $v_{successor}$  of  $v_{current}$ 
  {
    Set the cost of  $v_{successor}$  to the cost of  $v_{current}$  plus the cost to get from  $v_{current}$  to  $v_{successor}$ 
    If  $v_{successor}$  is in the list OPEN but the node in the list is better, discard this successor and continue
    If  $v_{successor}$  is in the list CLOSE but the node in the list is better, discard this successor and continue
    Remove occurrences of  $v_{successor}$  from OPEN and CLOSE
    Set the parent of  $v_{successor}$  to  $v_{current}$ 
    Add  $v_{successor}$  to the list OPEN
  }
}

```

```
Add  $v_{current}$  to the list CLOSE
```

```
}
```

The Dijkstra's algorithm time complexity is $O(V^2)$ where V is the number of nodes in the graph but it can be reduced to $O((E + V) \log V)$ where E is the number of edges and V is the number of nodes in the graph, if we use a heap to maintain the Open and Closed list instead of simple linked list.

Dijkstra's algorithm is one of the most studied pathfinding algorithm; in the following lines we will briefly show main advantages and disadvantages of using this algorithm.

It can be proven that no other algorithm can find a path better than the path found by the Dijkstra's algorithm therefore we can say that the Dijkstra's algorithm is the starting point for pathfinding.

The major drawback of the Dijkstra's algorithm is that it needs to span most (if not all) the graph before serving us with the solution we are asking for. This last statement implies the utilization of a large quantity of resources at our disposal: the time needed to span most of the graph plus the memory needed to store all the information generated by the algorithm.

Many studies are currently working on improving original Dijkstra's proposal. Mainly their concerns are to improve Dijkstra's performance by trading off resources for the "goodness" of the path or to trade off the demand for one resource with the other: for example,

“Iterative Deepening” algorithms aim at exchange memory resource for time needed to compute the path. Usually the solutions found are proven to be good for particular cases, but still Dijkstra’s algorithm performances on the generic case are quite difficult to reach.

2.5 A* Algorithm

The problem of all Pathfinding algorithms (Dijkstra’s included) is the enormous amount of resources (in memory needed and time complexity) they require. For example in the implementation chosen for this report the Dijkstra’s algorithm tends to expand nearly any node and have a time complexity of $O((E + V) \log V)$ where E is the number of edges in our graph and V the number of nodes.

In some cases, we don’t have enough time available or maybe we just want to keep the memory needed to a minimum level since we are low in resources: this yields to the conclusion that we must find a way to limit the global resource request.

The A* algorithm works much like the Dijkstra’s algorithm only it values the node costs in a different way. Each node’s cost is the sum of the actual cost to that node from the start plus the heuristic estimate of the remaining cost from the node to the goal. In this way, it combines the tracking of previous length from Dijkstra’s algorithm with the heuristic estimate of the remaining path.

The A* algorithm is guaranteed to find the shortest path as long as the heuristic estimate is admissible (we remember that an admissible heuristic is one that never overestimates the

cost to get to the goal). We point out that if the heuristic is inadmissible then the A* algorithm is not guaranteed to find the shortest path, but in most cases it will find a path faster and using less memory. The way normal cost and heuristic cost are mixed together usually is expressed by the formula:

$$\text{Node Cost} = \text{Cost to get there from start node} + \text{weight} * \text{Heuristic Cost}$$

In literature, we have a variety of heuristic functions. The heuristic function must be chosen according to the environment. We notice that a parking lot is very similar to a orthogonal maze: rows and columns are roads and the islands between them is the space for parking car; such an environment requires for best result a heuristic function called Manhattan heuristic.

The Manhattan heuristic formula is:

$$H(x_g, y_g, x_p, y_p) = |x_g - x_p| + |y_g - y_p|$$

Where x_g, y_g are the coordinates of the goal node and x_p, y_p are the coordinates of the point on which we are computing the cost. We remark that this heuristic never overestimates the distance as long as the weight used to compute the node cost is 1.

If the Heuristic function returned always zero, then the A* algorithm becomes the Dijkstra's algorithm, from this we understand that the A* algorithm could be seen as a

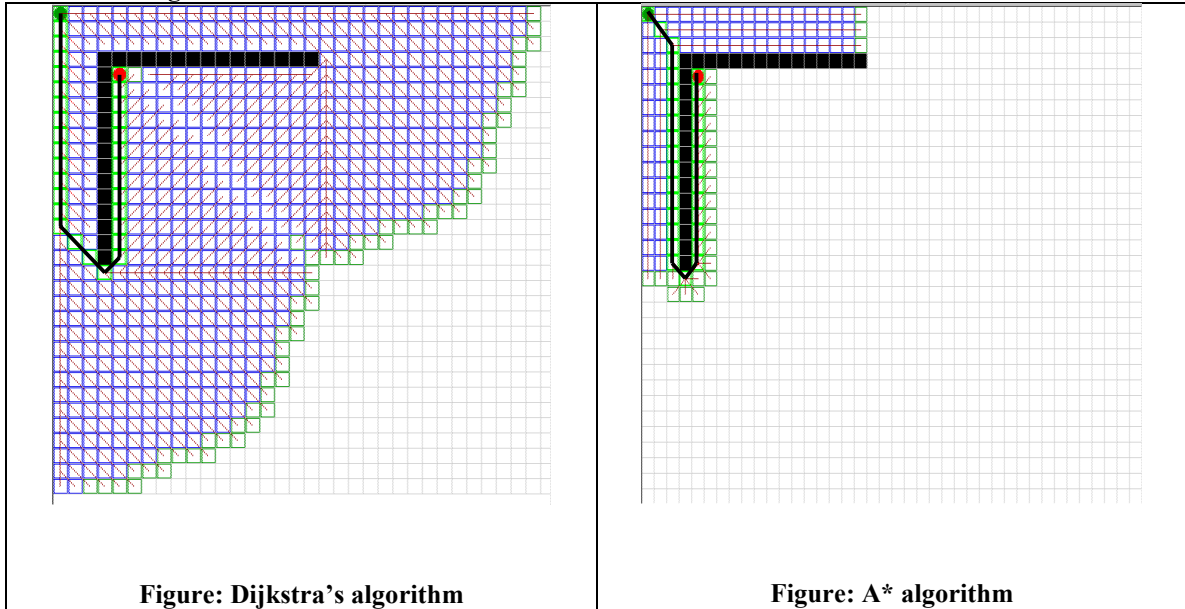
different version of the Dijkstra's algorithm. The Dijkstra's algorithm tends to expand any node which has the same weight: it is a Breadth-First algorithm. The A*, on the contrary, by using the heuristic tends to expand first the most promising nodes according to the Heuristic estimate of what 'promising' is: it is a Depth-First algorithm.

Even if it is not strictly possible to demonstrate which one of the two is better, it is reported by many studies that under the same circumstances (same data structures, same problems, same implementation) the A* outperforms most of the times the Dijkstra's algorithm.

This ability of the A* of focusing onto the goal more than the Dijkstra's algorithm, let us prefer it instead of the simple Dijkstra's algorithm. As a matter of fact, what we need is to minimize the demand for resources and the A* from this point of view is a big win since finding the solution in minor time means also a less demand for resources.

However the similarity with the Dijkstra's algorithm carries also its main drawbacks: high resources demand, high time complexity (even if complexity is less than Dijkstra's algorithm, the A* complexity is however high) and the lack of handling any dynamic changes onto the graph.

Table 3 : Comparative table. The colored squares are all the nodes expanded from each algorithm; in the left image, we see the Dijkstra's algorithm that expands many nodes more than A* algorithm before finding the solution.



2.6 Extended Dijkstra's algorithm

We exposed in previous paragraphs the main problem of using the Dijkstra's algorithm (or A*): the dynamic constraint.

From this point of view an improvement of Original Dijkstra's algorithm could be found in literature; this new algorithm is known as **Extended Dijkstra's Algorithm (EDA)**.

The basic idea of this new algorithm is the answer to the following question: "What does the time affect into the Dijkstra's algorithm?"

Here we define a dynamic graph G_D as a pair (V, E) where V is a finite set representing the nodes in the graph and constant with time. E is a binary relation over V that represents the edges of our graph; $E = E(t)$ is a function of time now.

This modification of graph definition leads to a slightly different algorithm, which we can see in the table.

Table 4: Extended Dijkstra's algorithm pseudo code

Dijkstra Algorithm Pseudo code

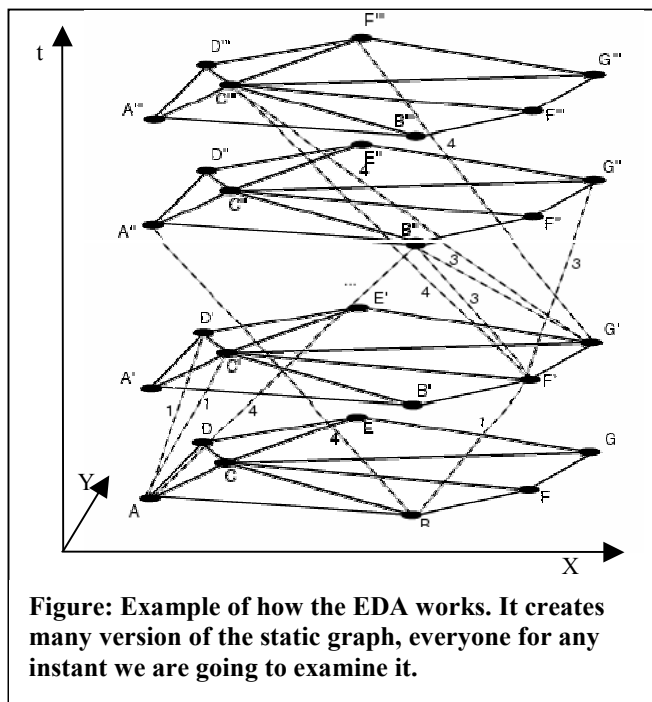
```

Set a node as goal,  $v_{goal}$ 
Set a node as start,  $v_{start}$ 
Set the time we pass by  $v_{start}$  equal to 0
Set the time we pass by any other node to infinite
Create an empty list called OPEN
Create an Empty list called CLOSE
While the list OPEN is not empty
{
  Get the node off the open list with the lowest time and call it  $v_{current}$ 
  If  $v_{current}$  is the same as  $v_{goal}$  we have found the solution; break from the while loop
  Generate a list of each  $v_{successor}$  connected to  $v_{current}$  at the time set by  $v_{current}$ 
  For each  $v_{successor}$  of  $v_{current}$ 
  {
    Set the time we pass by  $v_{successor}$  to the time we passed by  $v_{current}$  plus the time needed to get from  $v_{current}$ 
    to  $v_{successor}$  at the time set by  $v_{current}$ 
    If  $v_{successor}$  is in the list OPEN but the node in the list is better, discard this successor and continue
    If  $v_{successor}$  is in the list CLOSE but the node in the list is better, discard this successor and continue
    Remove occurrences of  $v_{successor}$  from OPEN and CLOSE
    Set the parent of  $v_{successor}$  to  $v_{current}$ 
    Add  $v_{successor}$  to the list OPEN
  }
  Add  $v_{current}$  to the list CLOSE
}

```

The EDA is a Dijkstra's algorithm which handle time changes onto the graph, both morphological changes (unavailable edges or new available edges at particular moments of our graph spanning) or simply weight changes (changes onto the weights of the graph edges), so it inherits all the results and problems of the Dijkstra's algorithm: high time and memory complexities. Beside there is a new problem which concerns this algorithm: the dynamic information. We observe however that dynamic information are usually available, for example in literature we have a good example on how to handle dynamic changes under metropolitan traffic situation to compute the best path and avoid traffic jams.

If we want a more intuitive explanation about what the EDA does, we could imagine this: let's imagine that we have many versions of our graph, each one of them represents the



graph in a particular instant: the dynamic changes onto our graph could be represented by new edges between two nodes which are accessible at given instants i.e. for given graphs. By creating these new graphs we translated a 2D dynamic graph in a 3D static graph where the new dimension is the time.

The ability to handle dynamic changes

let this algorithm be a good algorithm which is feasible under dynamic circumstances, provided that we have the dynamic information we need.

We observe that nothing denies us from combining an Heuristic with the EDA so that we have what we could call an Extended A* algorithm, in this case however we observe that an heuristic estimate of the remaining path is quite difficult and far from being reliable in most cases due to the dynamic changes which could occur to our graph.

2.7 The optimum

The first question we had to face during this research was the choice of the optimal path finding algorithm.

Dijkstra's algorithm is proved to be one of the most efficient algorithm. Its time complexity is $O(V^2)$ in the worst case and few other algorithms can provide this complexity and at the same time grant you the optimum path. Moreover we can find many different versions of this algorithm in literature and all of them add a speed more to the algorithm in various practical cases.

Dijkstra's algorithm is part of those algorithms referred to in literature as 'greedy algorithm'. A greedy algorithm is every algorithm that requires a complete spanning of the structures used to be performed in its best way; a greedy algorithm is a kind of resource-eater. In literature many changes about Dijkstra's algorithm aim at the minimum necessary spanning of the graph. Indeed, what is done is a trade off between the optimal solution and

resource usage: a good path is found (not necessarily the best one) but not every possibility is checked so to save memory and computational time resources.

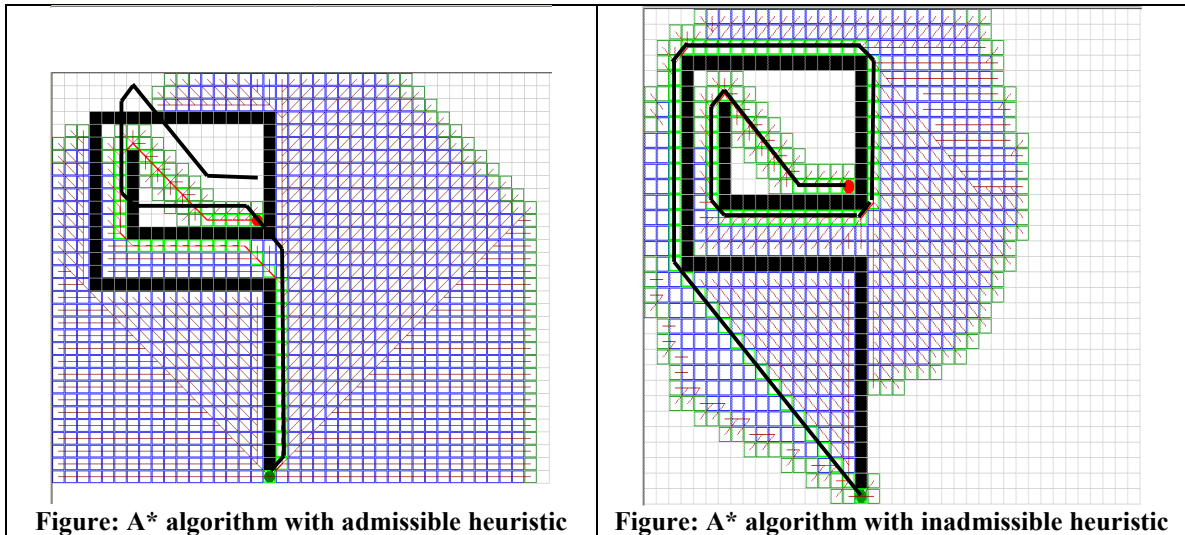
A trial in this way is the above explained A* algorithm that combines the Dijkstra's algorithm with the use of a heuristic function trying to forecast the nodes to expand first. As stated before, the heuristic function must be admissible. The problem is that if the Heuristic is inadmissible the algorithm doesn't guarantee us to find the optimum path (but it finds however another one). It is possible to demonstrate that no other algorithm that uses the same heuristic will expand less nodes than A* and at the same time it will guarantee you the optimum path in any graph.

What happens if we change the weight with which the heuristic function is weighted for computing the node cost? If we put the weight to 0 we return back to the Dijkstra's algorithm for which all the cases has been thoroughly studied in literature. If we set the weight of the Heuristic function to a value more than 1 we have transformed an admissible heuristic, such as the Manhattan heuristic used in our case, in an inadmissible heuristic function. The Manhattan heuristic is a function that is able to tell us the minimum distance to cover from our position to the goal in the case that our orthogonal maze is completely free. According to this, changing the weight used in the node cost formula, means that we are aiming at the goal more than trying to find an optimum path. This property is particularly useful.

During the testing of the pathfinding algorithm, it turned out that choosing a value greater than 1 leads the algorithm to find a good path much faster than normal. The path found was usually not the optimum one but it was found usually faster and expanding fewer nodes.

We tried to use this characteristic to our advantage. In our application we maintain a counter of the numbers of edges that are obstructed and we compute a weight according to this number. The weight range is between 1 and 2 and has proved to give worthwhile results. A typical example of this is shown on the table below. In this table we can see that the algorithm using the inadmissible heuristic focused on finding the goal more than trying to expand nodes and find the optimum path. The result is that it expands much less node than its counterpart and that the path found even if not optimal is not too bad.

Table 5: Comparative table. We can see here the A* algorithm with an inadmissible heuristic found a non-optimal path but expanding less nodes than the A* algorithm with an admissible heuristic saving time and resources.



Chapter 3: The Garage

3.1 Introduction

In this section we would like to show a feasible project for our AGV. We will try to follow as much as we can, the guidelines shown below:

1. **Safety**: the AGV must satisfy safety criteria for the passengers on board, for the other vehicles and eventual obstacles (persons, animals, etc.), for the environment.
2. **Robustness**: the AGV must undergo real circumstances and in many cases it could possibly happen that such goals are unreachable or that we have deadlock situations.
3. **Reliability**: the AGV should not require external assistance unless strictly needed.

Safety is a top priority requirement since safety of people possibly involved into the AGV surrounding is a must both by law and ethic.

3.2 AGV Movement

3.2.1 Holonomic vehicle

Let's suppose we have two points in a space A and B and no obstacles obstruct the way from A to B.

A **Holonomic** vehicle is able to go from A to B irrespective of any internal state; on the other hand a **Non-Holonomic** vehicle is a robot which can't go freely from A to B for some of its internal state.

An example of Holonomic vehicle is a shopping cart: since its wheels are capable of 360° degrees turn, it can go in any direction anytime despite of the current direction of the

wheels. The most common example of Non-Holonomic vehicle is a Car: it can't move on its side (think what you have to do when you ought to park a car).

Beside these two categories we could define a third one which we could name **Nearly-holonomic** vehicle that are those vehicles which belong to the Non-holonomic class but with some very easy fixed passages they can reach any point as if they were holonomic. The tank, for example, belongs to this class: it can't move on its side so it is a Non-Holonomic vehicle but since it has the ability to turn on itself it could turn by 90° degrees move forward and turn again by -90°, as if it has moved on its size.

Table 6: Holonomic. equations governing the movement of a generic Holonomic vehicle in a 2D space

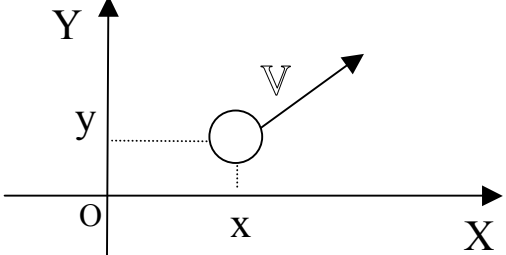
| | |
|---|--|
|  $x_{new} = x_{old} + v_x \Delta t$ $y_{new} = y_{old} + v_y \Delta t$ | <p>(x_{new}, y_{new}) : coordinates of vehicle's new position</p> <p>(x_{old}, y_{old}) : coordinates of vehicle's old position</p> <p>$\mathbb{V} = (v_x, v_y)$: Velocity vector</p> <p>Δt : time</p> |
|---|--|

Table 7: Non-Holonomic. Equations and Movements of a car-like vehicle (two steering wheels and two fixed wheels)

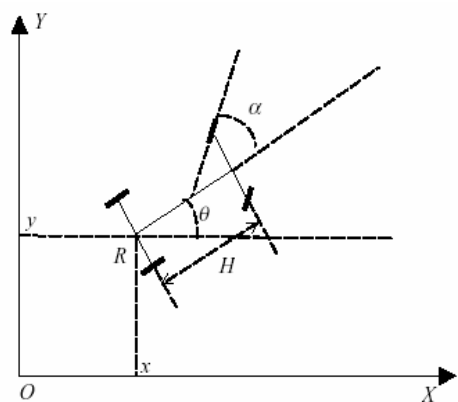
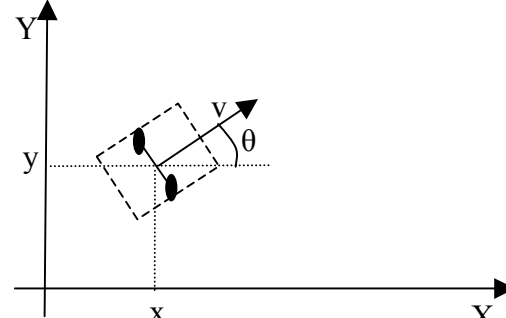
| | |
|--|---|
|  $x_{new} = x_{old} + v\Delta t \cos(\theta_{old}) \cos(\alpha)$ $y_{new} = y_{old} + v\Delta t \sin(\theta_{old}) \cos(\alpha)$ $\theta_{new} = \theta_{old} + v\Delta t \sin(\alpha) / H$ | <p>(x_{new}, y_{new}) : coordinates of vehicle's new position of the middle point of the rear axis</p> <p>(x_{old}, y_{old}) : coordinates of vehicle's old position of the middle point of the rear axis</p> <p>$\theta_{old}, \theta_{new}$: old and new angles respect to the X axis (counter-clockwise)</p> <p>α : steering angle</p> <p>H : distance between axes</p> <p>R : rear axis middle point</p> <p>v : wheel velocity</p> <p>Δt : time</p> |
|--|---|

Table 8: Nearly-Holonomic. Equations and movements of a tank-like vehicle

| | |
|---|--|
|  $x_{new} = x_{old} + v\Delta t \cos \theta_{old}$ $y_{new} = y_{old} + v\Delta t \sin \theta_{old}$ $\theta_{new} = \theta_{old}$ | <p>(x_{new}, y_{new}) : coordinates of vehicle's new position</p> <p>(x_{old}, y_{old}) : coordinates of vehicle's old position</p> <p>v : Velocity</p> <p>$\theta_{old}, \theta_{new}$: old and new angles respect to the X axis (counter-clockwise)</p> <p>Δt : time</p> |
|---|--|

3.2.2 In practice

Computing the movements for Non-Holonomic vehicle is quite difficult. Many studies are currently working on it especially for car-like vehicles; also all the proposals we studied although technically feasible are not 100% reliable and, as we pointed before, reliability is one of the most important requirements of our application: as a matter of fact we must find a way to make it easy.

We observe that in our environment the movements are restricted to the following categories:

- **Turning on crosses** : when our AGV turns on a cross or when it reaches a curve
- **Going forward** : when our AGV is on a generic road and need to go on
- **Stopping by drivers to pick them up** : when our AGV needs to pick up a driver and must come near to him, stop, wait for him to get on and start again
- **Inversion of direction** : occasionally when a road is blocked or jammed, our AGV could require 180° degrees turning

Even if a car-like vehicle is a common choice for these kind of AGVs, we observe that the maneuver required by our AGV would be accomplished also by a holonomic or nearly-holonomic vehicle: in this way we do not need special additional computation power to compute the necessary action a non-holonomic vehicle must perform, so that the movements are simplified to some basic action pretty easy to compute, implement and perform.

According to this, we chose to implement a tank-like vehicle which leads to a much simpler equation for the movements, shown above in the previous paragraph.

3.3 Requirements

Now that we have chosen which kind of movements our AGV is capable of, we have to decide which kind of other requirements our AGV must undergo.

At first, we notice that the AGV is expected to carry some basic sensors on it like distance sensors, communication sensors and global positioning sensors: these sensors are needed for the simple movements of the AGV. These sensors on our AGV are a critical requirement.

Secondly, our AGV must be able to handle all these kinds of information therefore it is expected to carry also a processing unit enough powerful to handle these signals. In theory, this computational power is not a strict requirement. We can think, for example, to implement just a simple radio transmitter that sends all information to a main station and receives raw commands for engines, lights and so on. In a real environment this is not a suitable choice: for security reason the AGV must be able to decide really fast in some critical situation (like for example the sudden presence of persons on the road) and a radio transmitter is not reliable in these situations. Therefore a minimal computation power on our AGV is a critical requirement.

Thirdly, our AGV must carry additional calculation power to handle the path finding algorithm, that, we remember here, it is extremely consuming either for computation complexity or for memory usage. This is not a strict requirement.

In this chapter we will examine in detail how these requirements are met.

3.4 Local or global computation

We examine now the main advantages and disadvantages of letting our robot carry additional calculation power for path finding.

Path finding is a resource-consuming problem. The complexity is, even in better case, more than linear this means that we need much computation power. Moreover, we know that mounting extra equipment on a moving robot will be much more expensive than having a fixed server that communicates the results to the robot: we must decide whether equipping the AGV with a powerful computer is worth.

In case we do, our AGV will be able to calculate its own way always, under normal condition and, more important, also under non- normal condition as for example lack in radio communication with environment sensors.

In this case the signals to exchange with the environment would be the entire state of the parking lot: the on-board computer will have to translate them in knowledge about the state of the parking lot and then compute the best way.

The other option is to install a powerful server that will handle the information coming from the environment, translate them in knowledge about the state of the graph, compute the best path and then send it to the AGV.

At a first look, the on-board choice could look better, but sometimes a centralized knowledge better suits some kind of problems like for example the presence of more than one AGV in the environment and problems like optimizing more than one path. This kind of investigation is left for further improvements.

In this report due to limitation imposed by the Lego hardware at our disposal, we used a centralized approach with a server that translates the knowledge coming from environment sensors into an optimum path.

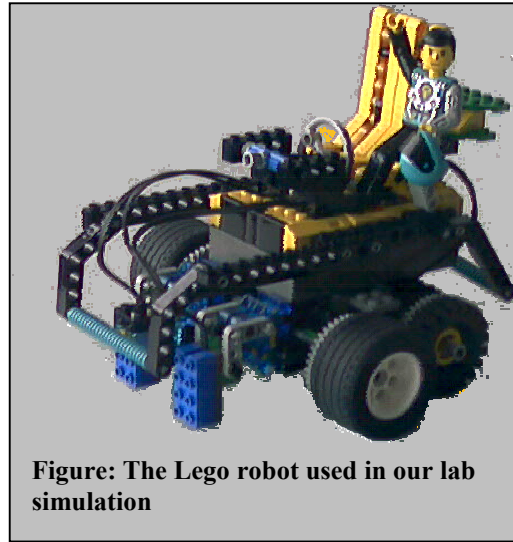


Figure: The Lego robot used in our lab simulation

3.5 The Lego robot

We talked about the necessity of providing our robot with sensors. Since this simulation should resemble as much as possible reality problem, in building our robot we tried to stick as good as we can reality situation.

The minimum requirements for a moving robot are: knowing its position, capability of understanding bumping situation and communication with a server.

Communication with a server could be easily accomplished thanks to the built-in IR transceiver of the RCX.

To make sure the robot get stick to its path, we used two light sensors. One of these sensors is used to know whether the robot is

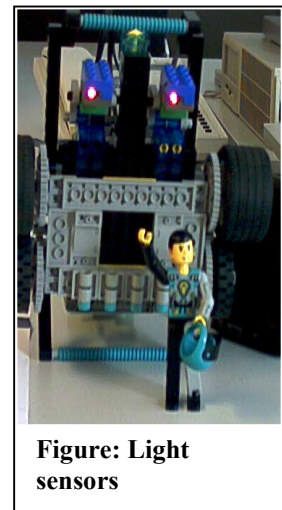


Figure: Light sensors

on a track, the other sensor is used to decide if the robot has arrived on a cross. For better explanation of tracks and crosses, we suggest you go to the environment section. Both sensors were positioned in front of the robot. These sensors are used to simulate the local position knowledge (i.e. how far the robot is from its ‘ideal path’).

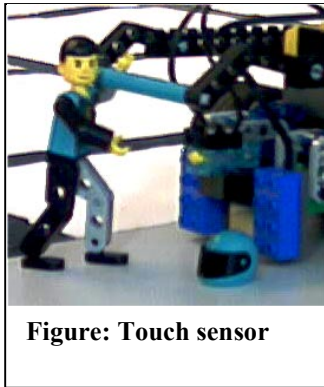


Figure: Touch sensor

To resemble distance sensors we used a touch sensor. This sensor switches on when the robot touches something: a ‘zero distance’ sensor. A real robot should use some more sophisticated distance sensor especially for security reason: we want the robot to react to a crash before it crashes! Actually, this sensor let us have a robot independent from environment

sensors: if the robot bumps into something then that road is obstructed and so we need to find another path.

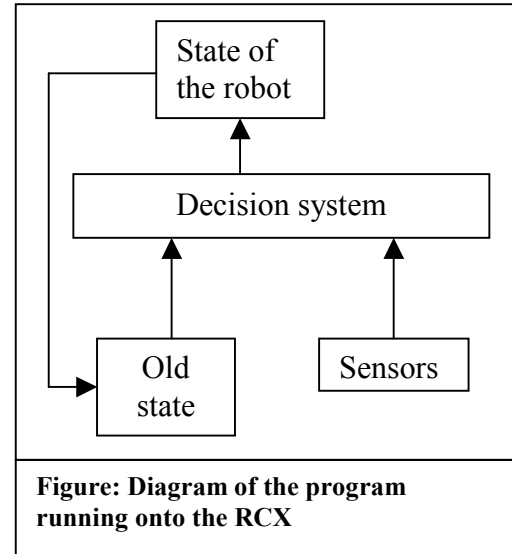
Two motors accomplish the movements; the movement is accomplished in a tank-style way i.e. the turning is accomplished by using different direction or velocities of the two engines.

3.6 *What's on RCX's mind*

As stated before the limited capabilities of the RCX led us towards a centralized approach. During the research, we uploaded onto the RCX only the code needed for navigation and state generation code.

Our objective was that the server would tell the robot only the way for it to follow when arrived at a cross. This objective was accomplished.

The program uploaded onto the RCX is able to handle the information coming from sensors and translate them in knowledge about the current state of the robot; in this way the robot is able to accomplish tasks like following a line drawn on the floor, deciding if it has reached a cross and



turn on a cross according to the server indication. The server role is covered by a computer.

For The RCX program used in this research, a diagram has been provided.

Moreover, the robot is also able to sense an obstacle on its way, through the bumping sensor, and then it asks back to the server for help on what to do.

Chapter 4: The Environment

4.1 AGV positioning system

The environment must be known. We supposed throughout our research that server has to know the environment in which it is moving and its current position.

Despite of what it could look like, this requirement is not difficult to implement. Nowadays we have at our disposal many way by the mean of which an AGV will be able to know its position on a map.

4.1.1 Guidelines

On small areas, especially in factories or automated environments, there is a special positioning system which relies on preestablished paths. This system requires the AGV to carry some (simple) sensors which are used to “know” where the path is. Usually the path is drawn onto the floor as a black line but lately a new technology is spreading which is much reliable and as cheap as the black lines, this technology is the inductive loop: it consists of iron wires buried into the floor which react as inductance to the sensor placed onto the AGV.

This positioning system has many advantages: it is very cheap compared to others, requires very little effort to be implemented and maintained and it is reliable; however it presents a drawback: if an AGV loses its path, then it is lost and must require for external help.

4.1.2 Landmark positioning system

The cheapest positioning system is the landmark system. It consists of special symbols drawn on the ground over the path the robot is going to explore. These symbols could be read by light sensors or other kind of sensors positioned on the bottom of the robot: every time the robot passes on one of these symbols, its position is updated. The accuracy of this system depends on the distance between two symbols. This system is subjected to destruction by environmental factors so it requires time periodic work for maintenance.



Figure: The popular Goose Game is an example of Landmark positioning system: the number of the box in which we are represents our position on the map

4.1.3 Dead Reckoning system

Another possible positioning system is the so-called dead reckoning. The idea is that if you know your starting position, you can keep track of your direction and the distance covered so you know exactly your relative position respect to the starting point. This system suffers of the problem that the error accumulates. Sometimes a hybrid system of Dead Reckoning and Landmark is used, dead reckoning for relative position and landmark to correct the error accumulated.

4.1.4 Global Positioning System (GPS)

The most known way to know an absolute position on a map is the GPS system. GPS has in its best case an accuracy of 22 meters that is not enough for our purposes. We observe that

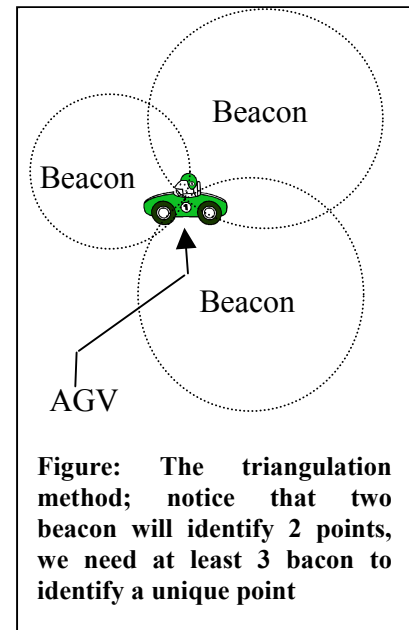
the resolution of GPS can be improved up to few cm, if we have a reference point on ground surface, a special tracking system and the object we are tracking is within 10 km from the reference point. This system is known as Differential GPS (or dGPS) and is used in applications where an exact position is a strict requirement such as airports and large harbors.

4.1.5 Local network positioning system

Another kind of positioning system could be a radio network by the mean of which our AGV will be able to calculate its position.

The basic idea of these networks is the triangulation model: if we want to determine the position on a 2D map, we have three radio beacon and we know their exact position on the map; by analyzing the phase difference between three beacon signals it is possible to compute the absolute position of an object on the map. The dGPS uses this principle to increase the precision of a simple GPS system.

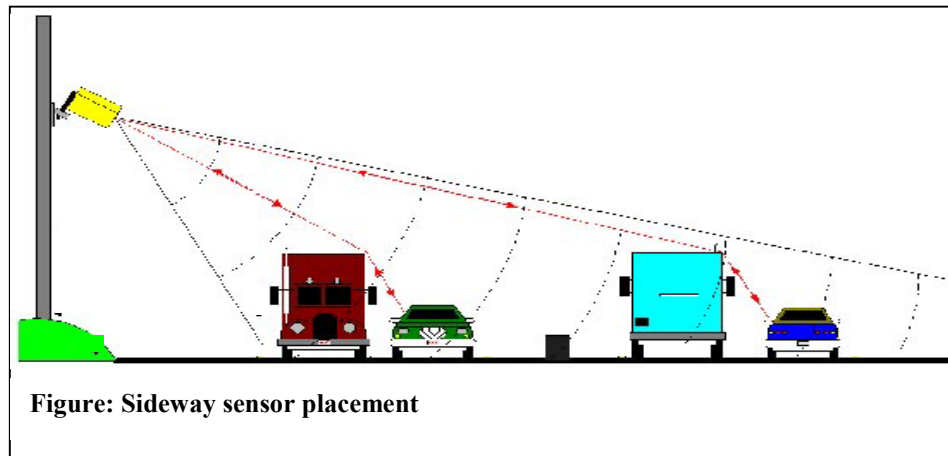
Usually these kinds of networks cover only a small area and need repeater to be installed all over the area on which we need the signal.



These alternatives are the most common used choices for positioning systems in small areas on terrain, the choice between which one is the best, must be done according to economical and environmental parameters; a larger variety can be found in literature and we remand to other sources to those who are interested into this field.

4.2 Car detection sensors

One of the key points of this work is the knowledge about the parking lot. We have a must: we must gather information about the state of a road, this means that we must know if there is a car occupying a particular road or not.



Many different sensors can be found that cover our needs, different sensors have different advantages and disadvantages. A comparative table with the most common sensors and their prices has been provided. Nearly all sensors used for car detection are extremely weather dependant. Under bad weather conditions, only microwave and ultrasound sensors

can provide good results. In the table below we can see also other two parameters that represents the accuracy in computing the presence and the velocity of a moving object passing through the zone scanned by the sensor. The overhead and sideways accuracy refers respectively to a sensor positioned in front of a moving object or on the side of a moving object. All these sensors are used to get feedback about traffic condition. This comparative table has been stilled by the Texas Traffic Institute.

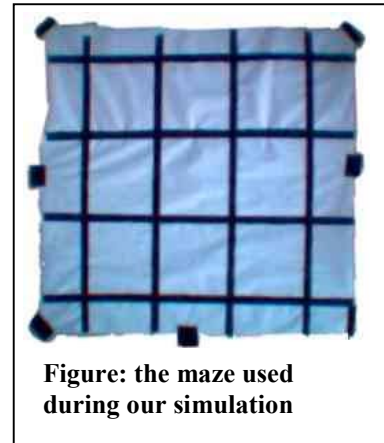
Table 9: Comparative table for different car detecting sensors

| TECHNOLOGY/ PRODUCT | COST/ROAD | OVERHEAD ACCURACY (% of Success) | | SIDEWAY ACCURACY (% of Success) | |
|--------------------------|-----------|--|-------|---------------------------------------|-------|
| | | Count | Speed | Count | Speed |
| Inductive Loops | \$746 | 98 | 96 | N/A | N/A |
| Active Infrared | \$1,293 | 97 | 90 | N/A | N/A |
| Passive Infrared | \$443 | 97 | N/A | 97 | N/A |
| Radar | \$314 | 99 | 98 | 94 | 92 |
| Doppler Microwave | \$659 | 92 | 98 | N/A | N/A |
| Passive Acoustic | \$486 | 90 | 55 | N/A | N/A |
| Pulse Ultrasonic | \$644 | 98 | N/A | 98 | N/A |
| VIDS | \$751 | 95 | 87 | 90 | 82 |

4.3 The simulated environment

For simulation purposes, a guideline positioning system has been used. The robot is able to recognize it arrived on a cross, this ability has been also used to synchronize robot predicted position with its real position.

The simulated environment consists of a tile maze: we have



black rows and columns on a white background. This choice was motivated after the sensors used for the robot, with black lines over a white background the robot was able to follow the line drawn till destination.

As stated before, this environment represents (apart for its dimensions) a parking lot: the white squares between rows and columns represent the space where to park the cars.

4.3.1 Sensor in simulated environment

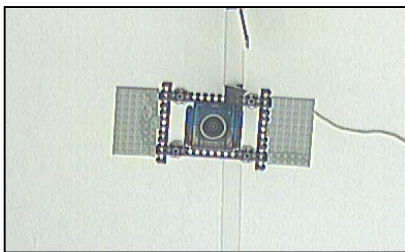


Figure: Lego webcam in 'Vision Command' Kit put on the roof

For the simulated environment, we used a Video Image Detection System (VIDS). The camera was included in the 'Vision Command' kit by Lego. The camera has been placed on the top of the maze. It was connected to the pc via an USB cable.

A program is running on the PC that captures the frames coming from the webcam and processes the image. The image processing capabilities let us know the state of any path of the maze. Obstacles are provided for simulating the presence of objects on the path.

4.4 Considerations

We want to point out in this paragraph what we wanted to simulate.

Considering the hardware at our disposal, we decided to use the Video detection system since, under lab condition, it resulted both cheap and reliable.

Moreover we had to choose what kind of “obstacles” we were to simulate. In a real environment we would have had many cars moving on our map, this kind of environment would have set the premises to gather time dependant information with which improved the performances of our AGV: the problem is that having many agents in a lab environment would have been too much expensive and also too much difficult to handle.

For this reason we decided to take into account only a first approximation of a real parking lot where first approximation states the ability of our detection system to “sense” the state of a road only as “occupied” or “free”.

The approach described until now however presented some problems.

The information about the maze state are limited, this means that every solution we compute is the optimum assuming that the situation will not change in the future. This is because we compute an optimum solution only for that moment. We didn't try to forecast the future state of the labyrinth and so we did not take into account any predicted obstructed position. This problem made us discard any dynamic algorithm such as the EDA since we could not gather any dynamic information and we could not forecast any future state.

Chapter 5: Implementation

5.1 Programming environment

Since this is a first approach to the problem of building an AGV, the main directive during programming was to provide an interface for further improvements and modifications. The best way to follow this directive is to use object-oriented languages that provide programmers with concepts like heritage and polymorphism useful to maintain code.

The language chosen was C++. The C++ has also another important characteristic, which has proved to be worth the choice: velocity. Since our AGV is going to move in real environment with real time problem, the velocity has turned out to be a fundamental requirement for our application.

Between the various C++ compilers the Builder C++ compiler from Borland has been chosen to implement our application. The choice is motivated from two factors: Spirit OCX component and ease of use.

The Spirit OCX is an ActiveX control provided by Lego to let the programmer interface the RCX Brick. An ActiveX control is a software component that integrates into and extends the functionality of any host application that supports it. ActiveX controls implement a particular set of interfaces that allow this integration. To use ActiveX controls, we need a programming environment able to recognize the ActiveX interface and integrate its functionality in user's source code. The C++ Builder provides many wizards that help the programmer in integrating the ActiveX controls in his source code.

Moreover the C++ Builder provides the user with many automatic routines for generating code to handle windows messages and basic actions; since the limited time at our disposal

this has been an important parameter that let us time for concentrating over the visual control and the path finding algorithm. Overall the C++ Builder has shown important characteristics to support programmer's work between them we would like to underline the ease of use, reliability and clean code provided by the compiler and portability (as strict ANSI C++ compiler).

The next paragraphs will describe in detail the techniques used to implement the program interface and the main class called Graph encapsulating the graph storing functions and pathfinding algorithm. We remand to the appendix for source code.

5.2 Requirements

One of the most important requirements in programming is portability and modularity. According to these directives much of the source code is organized in completely autonomous classes. The communication between classes is accomplished by using an external class that has the task of handling the messages coming from the various part of the program and send them to the appropriate receiver; this approach has been chosen for portability reason: building a platform dependent message exchanging system would have let our program depend on specific operating systems. Also wherever it was possible strict ANSI C++ source code has been provided.

The second most important requirement is velocity. The bottle neck of our application is the visual component. During our first period of building the component we were building

a component which used some edge recognition and other image recognition techniques to sense the state of our maze. We chose not to go further into this field when we realized that every image needed many seconds to be analyzed. We decided to simplify as much as we could this component to let us concentrate onto the pathfinding problem and reserve more time on computing the best path, we discuss the details in next paragraph. Another part of our application which required some effort to be implemented was the optimum choice of the structures needed to implement our graph-like which would encapsulate our maze, we dedicated to this subject a paragraph later in this section.

Third important requirement was reliability: a reliable AGV is much important for it must go through real time problems. Considering the enormous variety of problem that could occur to such an AGV, this was maybe the most difficult requirement to satisfy and many times this requirement was in contrast with the previous two. An example of such problems could be lack in communication with our server which will cause our AGV to stand on a cross waiting for data, or a mechanical failure such as a broken engine or fault in batteries which are much more difficult to handle than previous case. We tried to present a solution to these problems later in this section.

5.3 Visual part

The first component built has been the component that is in charge of handling the signals coming from the global environment sensors. In this case this sensor is a webcam

positioned on the top of the maze. The objective of this component is to provide the state of every road present in the maze. Particular attention has been put into the creation of a reliable and fast component.

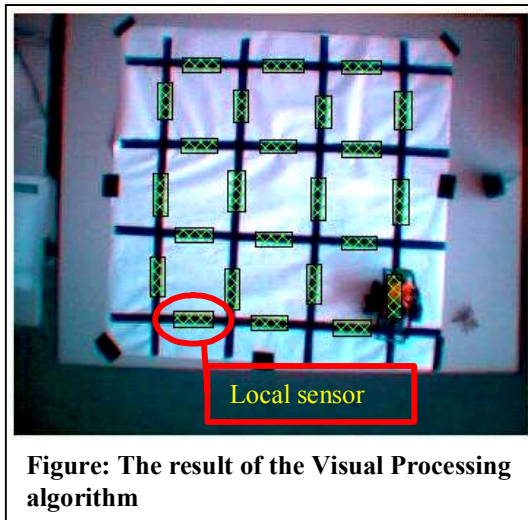


Figure: The result of the Visual Processing algorithm

Using image-processing techniques it took more than 2 seconds to elaborate a simple image (352 x 288 pixels) when no other process was running on our pc. Two seconds in real-time problems makes a lot of difference. A lot of effort has been put in trying to keep the time necessary to elaborate a frame to the minimum possible.

The basic idea is that we don't need the whole image to be processed but only the paths. For this purpose some 'local sensors' has been created. A 'local sensor' is a particular sensitive zone able to recognize whether in that zone there is a path or not.

The position of the path is assumed to be known this means that the user must put the 'local sensor' onto the edge of the maze and then the 'sensor' looks if at that position an edge is effectively existing. In figure, we can see these sensors: the boxes over the edges.

With all this improvements we were able to process a frame coming from the camera in less than 0.20 seconds that is much more acceptable for real-time purposes.

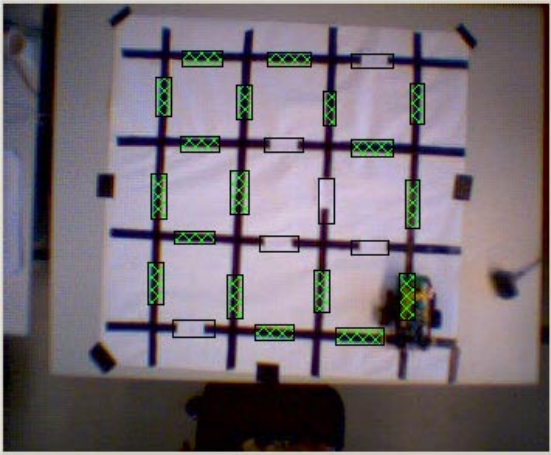
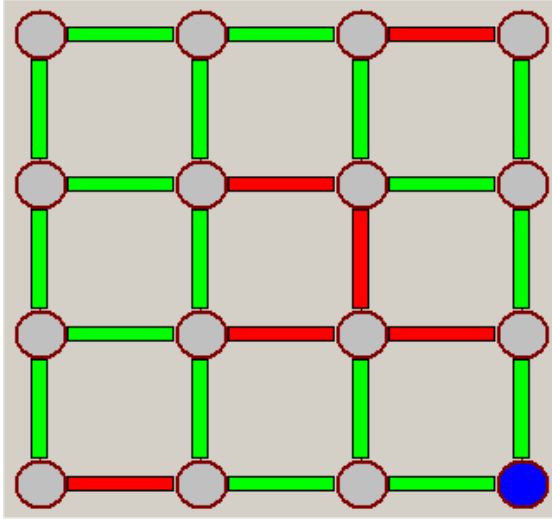
The purpose of keeping the elaboration time to the minimum possible was that in a real parking lot the cameras used could be more than one. If we think for example of a parking

lot with 10 cameras, the time for elaborating the entire parking lot has been improved from 20 seconds to 2 seconds.

The work of the ‘local sensor’ is simple: they compute the black percentage (number of black pixels divided by number of all the pixels) in a rectangular region.

Below we exhibit a demonstration of the capabilities of the local sensor. We can see on the right the image shown to the webcam and on the left the internal reconstruction of the labyrinth.

Table 10: Image acquired from camera compared with internal representation of the state of the labyrinth.

| | |
|--|---|
|  |  |
| Figure: Image acquired from the camera | Figure: Internal representation of the labyrinth |

5.3.1 Visual component extensibility

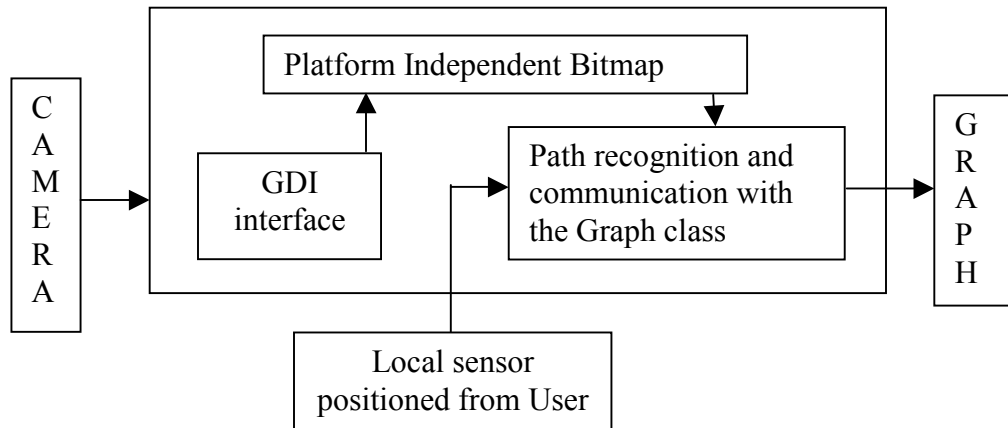


Figure: Visual component

The visual component makes extensive use of Graphic Driver Interface (or GDI) provided by the Windows operating system. This functions are required if you want to build a graphic device, such as images, interface with data acquiring scanners or cameras under windows operating system. For this purpose we split into two parts this component: the first part that acquires the frames form the camera and translates them into platform independent bitmaps, a second part that is the implementation of the local sensors'. The first part is heavily dependent on GDI and so not easily portable, the other is all included in a function named EdgeRecognition of ANSI C++ and for this very portable.

5.4 The Graph class

We gathered all the structures and functions needed to maintain a graph and to solve the path finding problem in a single ANSI C++ class. This class is called 'Graph'.

An intuitive interface is provided by the Graph class to take actions on the graph. The structure used to maintain the nodes of our graph is a Heap. This structure is needed to reduce the time complexity of our graph spanning functions and pathfinding routines from $O(V^2)$ to $O((E + V) \log V)$.

We observe that in literature many studies aim at finding best Data structures to be used in Dijkstra's like algorithms. Many studies have pointed out that the best time complexities are achieved by the mean of a data structure called Fibonacci Heap. This assumption is generated from the observation that a Fibonacci Heap has the time complexities shown in the table. By this assumption we could have reduced the time complexity of the Dijkstra's algorithm from $O(V^2)$ to $O(E + V \log V)$ using a Fibonacci heap.

Table 11: Comparative time complexity table between a simple Binary Heap and a Fibonacci Heap

| Operations | Binary Heap | Fibonacci Heap |
|---|----------------|-------------------|
| Insertion | $O(\log N)$ | $O(1)$ |
| Extract Min element | $O(\log N)$ | $O(\log N)$ |
| Decrease Key (change and repositioning of a node) | $O(\log N)$ | $O(1)$ |

We observe however that what we are looking for is not merely an asymptotic performance but a good performance for our algorithm; from this point of view we must notice that Fibonacci heaps, although they show great asymptotic performances, suffer of slow implementations, therefore many studies have highlighted the question that Fibonacci heaps

are proven to subclass normal binary heap only when the number of elements held into the data structure is more than 10,000 nodes. Let us observe that 10,000 nodes leads to a squared parking lot with approximately 100 nodes per side, 19,800 roads available for parking and, assuming at least 10 car parking spaces per road, at least 198,000 available parking spaces: Can we imagine what kind of facility would require such ‘hugeness’?

According to the previous observation we decided to use a simple binary heap as the main data structure since it shows the best trade off between real and asymptotic performance.

5.5 Interfacing with the robot

The interface with the robot is provided by a class called LegoForm. This class is the only class able to communicate with the robot, since it uses the Spirit ActiveX component.

The robot itself is not able to send messages to the Spirit control but the Spirit can poll the value of any variable of the RCX brick. A variable on the RCX has been especially reserved for communication purposes. This variable is set to a particular value when the RCX is ready to receive the next communication. The only messages that the robot exchanges with the program are just the direction the robot has to take to the next cross.

5.5.1 Synchronization

Since the positioning system used to know the position of the robot is a guidelines positioning system, we need some synchronization between the robot and the application:

anytime the robot is on a cross, it must communicate to the application its position so it can be updated.

Synchronization between the robot and the application is done using a simple message exchange. When the robot arrives at a cross he sets the communication variable on so to let the computer know that it is waiting for data to come. At this point, the LegoForm asks to the graph class (using an intermediate class called LabyrinthForm) the next direction. This system has proved to be very reliable but has a drawback: when there is a lack in communication the robot stands on a cross waiting for data. This problem shows us the main drawback of our original choice of having a centralized approach.

These messages are translated in a more understandable way by the main application and are shown on video during the running of a test. Here we show a table with all the messages exchanged between the robot and the application during a normal test.

Table 12: typical messages exchanged between the robot and the main application during a test

Robot: Engine on, ready to start

Navigator: Easy right!

Robot: I'm turning right now

Robot: I'm following the line

Robot: I'm on a cross, What should I do?

Navigator: Easy right!

Robot: I'm turning right now

Robot: I'm following the line

Robot: Shit! I bumped into something! What are you looking at?

Navigator: Sorry. I will try to find another way

Navigator: Get back!

Robot: I'm going back

Robot: I'm following the line

Robot: I'm on a cross, What should I do?

Navigator: Easy right!

Robot: I'm turning right now

Robot: I'm following the line

Robot: I'm on a cross, What should I do?

Navigator: Speed up!

Robot: I keep on going

Robot: I'm following the line

Robot: I'm on a cross, What should I do?

Navigator: You're done

Robot: That's the goal!

5.6 Finally

The rest of the application is just a collection of functions and routines needed to handle Window's messages and provide the user with a user-friendly interface.

We notice that apart from window's messages all the internal messages are handled by the class `LegoForm`: it is the intermediate class that handles all the messages and forwards them to the right receiver.

Let us have a last observation: for the graphical user interface and lego robot interface we have used Windows operating system routines that means that for these two aspects of our application we have not satisfied the portability criterion, however since Lego Mindstorm Kit does not provide the end-user with a multi platform interface our choice was restricted to this sole option. In figure we can see the main form of our application.

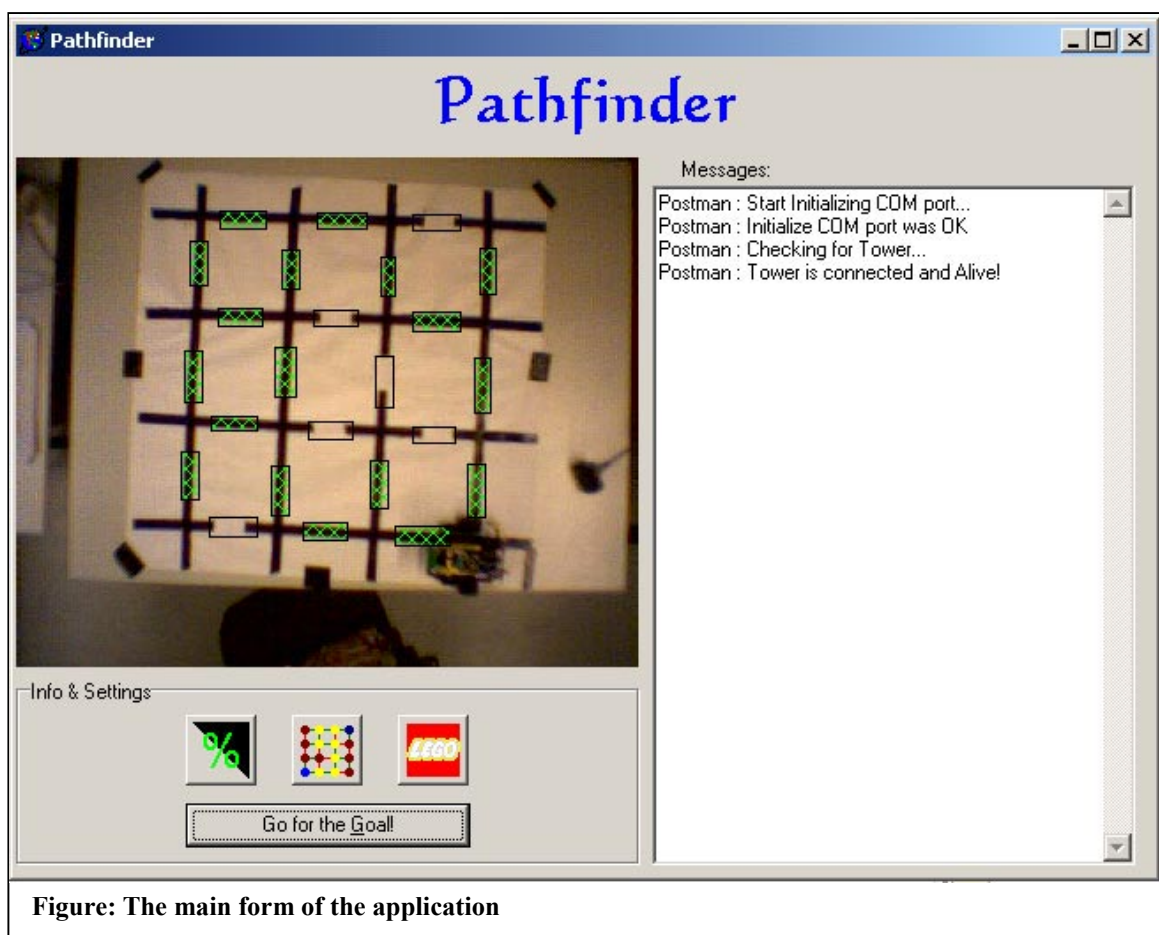


Figure: The main form of the application

Chapter 6: Conclusion

6.1 Results

At first we discussed about the structure of such an AGV; we showed the various possibility that current technology can afford, we showed the main difference between having a holonomic, non-holonomic or nearly holonomic vehicle and we opted for a nearly-holonomic vehicle as the best choice: as a matter of fact an automated driver of a car-like vehicle although technically feasible is still not 100% reliable mostly due to the huge complexity of the related equations especially in non-normal condition (think for example all the maneuver you must perform with a car when you have to ‘simply’ turn it by 180° degrees).

These easier equations let our robot perform more movements since they were more easy to compute. Our robot is now able to go forward by any distance (limited only by the IR transceiver needed to communicate with the computer) and turning on itself by any angle. These ability led our robot to perform all the actions we asked it to perform i.e. it is now able to go from a cross to another, turning right or left on a cross, going back on a cross and turning back if a road is obstructed: not bad for a handful of Lego bricks!

We discussed the requirements of our robot and we stated the importance of having a safe robot according to the environment in which it should drive. After the observation that most of the dangerous situations of a parking lot would occur when there are obstacles on the road (i.e. people who could cross the road, car accidents, ‘unusual’ drivers who drive in a possibly dangerous manner, etc.), we decided to mount a bumping sensor on the front side of the robot. By mounting this sensor we obtained the double advantage of improving the

safety of our AGV and at the same time letting it be more independent by the environment sensors: as a matter of fact even if the environment sensors do not work, our robot was able on its own to understand when a road was blocked and communicate the state of the road to the computer so that it could compute another way: this led to a more reliable system. We induced this behavior in our lab by blackening the camera, in this way the pathfinding algorithm has no other information than the ones provided by the robot itself; during this experiment the robot had some trouble in finding its way to the goal (it had to test all possible paths it wanted to take) but finally it got to the goal without any previous information about the state of the road.

Another interesting part of our work was the environment sensors. We chose to implement the sensors responsible for the computation of the state of the various road of our maze as much simple as we could and, as simplest, we opted for a color checking and a XOR combination with the background. By choosing a simpler way to implement the local sensors responsible for the roads of our maze we managed to improve the global performance of the application leaving the main time of the execution for the communication/synchronization with the robot and the pathfinding problem. We point out that this way of proceeding is not much different from what a commercial VIDS system does: the XOR combination between the current image and a previous background image is *de facto* a standard procedure in the field of image recognition against a background. During our experimentation the time needed to implement an image was a small percentage

of the total application time under many hardware configurations; in the table below we can find the average execution time of the image routines of our application.

Table 13: Comparison of the time needed to analyze one frame by the image routines of the application

| CPUs tested | Time needed (millisec) |
|--------------------------|-----------------------------------|
| AMD K6-III 450Mhz | .45 |
| Intel Pentium III 800Mhz | .22 |
| AMD Athlon XP 2200+ | .081 |

The last but not least part of the application was the pathfinding algorithm. The study for a good choice led towards a good algorithm according to the dynamic constraints we talked about in the related section.

The algorithm we used, as described into the related section, is the A* algorithm. This algorithm is fast and it can be proven that it computes the best path according to the information it has been given. As we described before in this report, the information are static, that means that the solution computed is the best available path, assuming that the state of the maze will not change in the future: the motivation for such a choice is due mainly to the unavailability of dynamic information i.e. we did not try to forecast any future state and so we do not have any information that a dynamic algorithm (such as the EDA) could find useful (we remember here that if we lack in dynamic information, the EDA is equal to the Dijkstra's algorithm).

This choice however led to a faster computation of the path since no additional information had to be considered during the computation of the path; we used this characteristic to our advantage: the path computation time was very limited (far less than the time needed by the robot for going from a node to another), in this way we could compute a new path any time the robot has engaged a road (i.e. when it does not require any communication unless extraordinary situations) without modifying the global application behavior and synchronization problems with the internal state of the robot.

The extreme velocity of the path finding routines were also used to compute a new path anytime the state of the maze changes and so interferes with the old computed path. In this way our AGV is able to react to external stimuli and choose a different path anytime the old path has been obstructed by an external cause.

6.2 Problems and Further improvements

This project has been organized as a first approach to the problem of having an AGV driving in a controlled environment and according to this statement there is plenty of space for improvements.

We will talk now of some problem presented by our application. We want to analyze the pathfinding algorithm chosen for our application. Since the path computation is triggered by any notable change into the state of the maze, the problem arises when we have a stable

state oscillation of our maze i.e. when the state of our maze oscillates between two states which are notable so that this oscillation triggers the computation of an alternative path.

When we have the above situation there is the possibility that the AGV will never reach its goal (notice that the goal is reachable in any moment but the AGV has not enough time available to reach the goal before the state change occurs); in this case we will have the AGV hang around in the maze without any precise path or better trying to follow two paths at the same time without really following any.

These circumstances were not studied and so the AGV still presents this problem; we observe however that this problem in a real parking lot is unlikely to happen: it is extremely unlikely that the state of a parking lot could oscillate indefinitely between two states without reaching an equilibrium (car drivers do not simply hang around into the parking lot, they have precise goal and strategies to achieve them).

The following step into this field should be the study of a way to eliminate this problem. According to these criteria, the EDA algorithm has been proposed as a next step which will mostly eliminate this problem and propose better paths. However the EDA algorithm and any other solution which would be adapted to the problem would require a time dependant environment sensor system (it must recognize oscillating situation or possibly forecast them), therefore this improvement would require first the study of a time dependant sensor system which will be able to gather time dependant information and transform them into the information needed by the algorithm.

6.3 Conclusion

The way to build an automatic AGV is still far. The problem of driving in a hostile environment, with no information about it, is still mainly unresolved. On the other hand it's our opinion that the time has come for building an AGV able to move in a suitable environment. The result of this first approach to the field is quite satisfactory. The AGV turned out to be much reliable in our lab simulation providing us with a handful of hopes for a successful commercial implementation.

We can imagine that in a few years when we will park our car, a robot will come and offer us a lift to our destination. The question is: will we accept it?

Bibliography

- **About Lego Mindstorm**

Lego Mindstorms (2000) - Original mindostorm webpage - <http://mindstorms.lego.com>

MIT - Creator of the RCX Intelligent brick - <http://fredm.www.media.mit.edu/people/fredm/mindstorms/index.html>

B. Erwin - *Creative Projects with LEGO® Mindstorms™* - April 2001.

J. Knudsen - *The Unofficial Guide to LEGO MINDSTORMS Robots* - October 1999.

D. Baum - *Dave Baum's Definitive Guide to LEGO MINDSTORMS* - November 1999.

About RCX Internals - <http://graphics.stanford.edu/~kekoa/rcx/>

Lego Mindostorm Internals - <http://www.crynwr.com/lego-robotics/>

- **About path planning algorithms and implementations**

Dijkstra E (1959) - *A Note on Two Problems in Connection with Graphs* - Numerische Mathematik 1, pp. 269-271.

G. Eggenkamp (2001) - *Dynamic multi modal route planning: an artificial intelligence approach* - Delft University of Technology.

ftp://ftp.kbs.twi.tudelft.nl/pub/docs/MSc/all/Eggenkamp_Gerritjan/thesis.pdf

J. H. Kingston (1990) - *Algorithms and data structures: design, correctness, analysis* - Addison-Wesley, University of Sydney, Australia.

L. Nada - *An optimal pathfinder for vehicles in real-world digital terrain maps* - <http://www.student.nada.kth.se/~f93-maj/pathfinder/contents.html>

M.T. Saborido (1992) - *An introduction to expert system development*. In *Application of Artificial Intelligence in process control* - L. Boullart, A. Krijgsman en R.A. Vingerhoeds, Pergamon Press, Oxford, UK.

J.W.C. van Lint - *Robust and adaptive travel time prediction with neural networks*. In *Proceedings of TRAIL 6th annual congress, December 12th 2000*. The Hague/Scheveningen, The Netherlands.

R. Kroon (2001) - *Dynamic vehicle routing using ant based control* - Delft University of Technology. <http://www.kbs.twi.tudelft.nl/Publications/MSc/2002-Kroon-MSc.html>

Implementation Notes on pathfinding algorithms -

<http://theory.stanford.edu/~amitp/GameProgramming/ImplementationNotes.html>

B. Cherkassky, A. Goldberg, T. Radzik - *Shortest Paths algorithms: theory and experimental results* - 1993

Fredman, M. and Tarjan, R. - *Fibonacci Heaps and Their Uses in Improved Network Optimization Algorithms* - Journal of the Association for Computing Machinery, vol. 34, no. 3, July 1987

S. Russel and P. Norvig - *Artificial Intelligence a modern approach* - Prentice Hall Series

M. Ginsberg, M. L. Ginsberg - *Essentials of Artificial Intelligence* - April 1993

Cormen, T., C. Leiserson, and R. Rivest - *Introduction to Algorithms* - Cambridge, MA: MIT Press, 1990.

L. Killough - *Priority Queues* - <http://www.leekillough.com/heap/>

• **About AGVs**

FROG (2000) - <http://www.frog.nl>

Ioannou, P.A. (1997) - *Automated Highway Systems* - Plenum Press, New York.

SAVE project (2000) - <http://www.iao.fhg.de/Projects/SAVE/save/saveinit.htm>

TRAIL (1999) *Automation of car driving: exploring societal impacts and conditions* - Heijden, van der, R.E.C.M., Wiethoff, M. editors, December 1999, TRAIL studies in transportation science, S99/4, Delft University Press, The Netherlands.

A. Bicchi, L. Pallotino - *Optimal planning for coordinated vehicles with bounded curvature* - Technical Report.

PARKIR (2000), *report on parking possibilities near trainstations*, OVR.

Franklin, S. and Graesser, A. (1996) - *Is it an agent, or just a program?: a taxonomy for autonomous agents* - In Proceedings of the Third International Workshop on Agent Theories, Architectures, and Languages, Intelligent Agents III, Springer-Verlag New York, pages 21-35. Also <http://www.msci.memphis.edu/~franklin/AgentProg.html>

Ehlert, P.A.M. (2000) - *Intelligent driving agents: the agent approach to tactical driving in autonomous vehicles and traffic simulation* - Delft University of Technology.
ftp://ftp.kbs.twi.tudelft.nl/pub/docs/MSc/all/Ehlert_Patrick/thesis.pdf

Ehlert, P.A.M. (1999) - *The use of artificial intelligence in autonomous mobile robots* - Research report, Delft University of Technology.
http://elektron.its.tudelft.nl/~s218303/docs/OTreport_ps.zip

Hoedemaeker, M. (1999) - *Driving with intelligent vehicles: driving behaviour with adaptive cruise control and the acceptance by individual drivers* - Trail thesis series T99/6, November 1999, Delft University Press, The Netherlands.

Minderhoud, M.M. (1999) - *Supported driving: impacts on motorway traffic flow* - Trail thesis series T99/4, TRAIL Research school, Delft University Press, The Netherlands.

C. Wang, C. Thorpe, and S. Thrun - *Online Simultaneous Localization and Mapping with Detection and Tracking of Moving Objects: Theory and Results from a Ground Vehicle in Crowded Urban Areas* - IEEE International Conference on Robotics and Automation, May, 2003.

Smiley, A. and Brookhuis, K.A. (1987) *Alcohol, drugs and traffic safety*. In *Road users and traffic safety*, Rothengatter, J.A. and De Bruin, R.A. editors, Assen: Van Gorcum, pages 83-105.

Pomerleau, D. (1993) - *Neural network perception for mobile robot guidance* - Kluwer Academic Publishers, Boston.

G. Capuano, S. Dilillo - *Reproduction of the Trout* - In Proceedings of reproduction theories, January 1993 - Mercalli Liceum press, pages 34-52, See Indice Analitico

C. Thorpe, R. Aufrere, J.D. Carlson, D. Duggins, T.W. Fong, J. Gowdy, J. Kozar, R. MacLachlan, C. McCabe, C. Mertz, A. Suppe, C. Wang, and T. Yata - *Safe Robot Driving* - Proceedings of the International Conference on Machine Automation (ICMA 2002), September, 2002.

C. Thorpe, D. Duggins, J. Gowdy, R. MacLachlan, C. Mertz, M. Siegel, A. Suppe, C. Wang, and T. Yata - *Driving in Traffic: Short-Range Sensing for Urban Collision Avoidance* - Proceedings of SPIE: Unmanned Ground Vehicle Technology IV, Vol. 4715, April, 2002.

Robocup (2000) <http://www.robocup.org>

- ***About image manipulation routines and other car sensors***

Microsoft Software Developer Network - <http://www.msdn.microsoft.com>

S. McNeil, D. Duggins, C. Mertz, A. Suppe, and C. Thorpe - *A Performance Specification for Transit Bus Side Collision Warning System* - ITS2002, proceedings of 9th World Congress on Intelligent Transport Systems, October, 2002.

R. Aufrere, C. Mertz, and C. Thorpe - *Multiple Sensor Fusion for Detecting Location of Curbs, Walls, and Barriers* - Proceedings of the IEEE Intelligent Vehicles Symposium (IV2003), June, 2003.

C. Wang, C. Thorpe, and A. Suppe - *Ladar-Based Detection and Tracking of Moving Objects from a Ground Vehicle at High Speeds* - IEEE Intelligent Vehicles Symposium (IV2003), June, 2003.

Owlnet research - *Laplacian Edge Detection* - <http://www.owlnet.rice.edu/~elec539/Projects97/morphjrks/laplacian.html>

Appendixes

• *Appendix A: Beyond the Subject*

Brief introduction to Lego

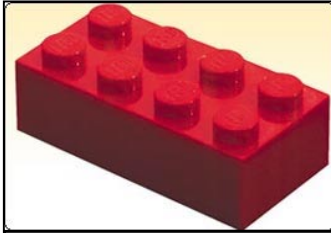


Figure: Lego brick

Lego bricks are one of the most famous toys all over the world.

Mainly they are composed by a technology called 'Stud-and-Tube' that let all the bricks stick together and build any kind of construction.

The idea comes from Mr. Ole Kirk Christiansen and his son Godtfred Kirk that in 1932 invented the first Lego bricks and founded a carpentry business in the village of Billund. From there on till nowadays Lego bricks spread all over the world so that now they are considered toy of the century. The name Lego comes from the Danish words "LEg GOdt" meaning "play well". A funny thing is that in Latin Lego means "I study" or "I put together".

Lego Company is now a great company whose main concern are toys. Everyday they produce new toys which only purpose is to let children (and not only) develop their skills. The success of Lego Company is due also to the continuous researches in child satisfaction and technical collaboration with many universities.

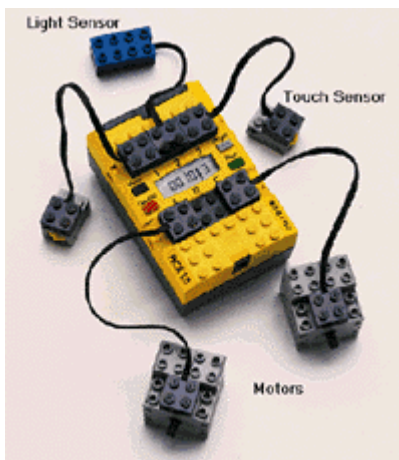
The brain: RCX Brick



The RCX is a LEGO microcomputer used to create robot. It can control up to 3 engines and can receive the input from 3 different sensors. It can be programmed so to react to external stimuli. The firmware, uploaded onto, presents some nice features like subroutine

handling and multitask programming. It comes shipped with visual software and an ActiveX component called Spirit that takes care of interfacing the programmer with the RCX.

The body: Sensors and actuators



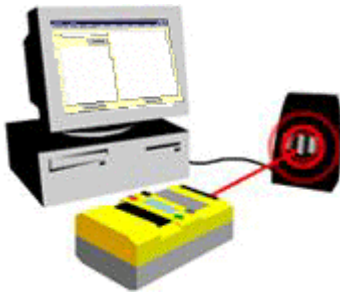
For assembling robots we can choose between a variety of sensors to use. Two kinds of sensors has been used for our purposes:

- ***Light sensors:** these sensors are able to sense the color (in grayscale) of the object they are aiming to. Two sensors of this kind were used in the project.*

- ***Touch sensors:** these sensors are normal switch used as a bump sensor.*

For the movement of the robot two motor bricks have been used. These motors need a 9 Volts power supply provided by the RCX and cover the role of actuators in the robot built using the Mindstorm kit. It is provided with gears so it can change velocity (from 0 to 7) and switch direction.

The communication: Infrared TX/RX



The RCX communicates with the PC via an Infrared (IR) Transmitter. This transmitter is attached to the serial port of the computer.

An interface between IR sensor and programming language is provided by Lego by the mean of an ActiveX control that is called Spirit. The Spirit ActiveX control is able to compile code which will be uploaded onto the RCX for a completely autonomous execution or send direct commands to it through the IR sensor.

• *Appendix B: Tutorial*

In this section we provided a brief tutorial to build and run the experiment.

The Robot

What you need is:

- ❑ *An RCX brick*
- ❑ *2 light sensor*
- ❑ *1 touch sensor*
- ❑ *2 engines*
- ❑ *all the bricks necessary to connect these pieces together*

First, You must position the light sensors in front of the RCX. They must aim to the ground. The Light sensor on the left must be connected to the input number 1 and the Light sensor on the right must be connected to the input number 3.

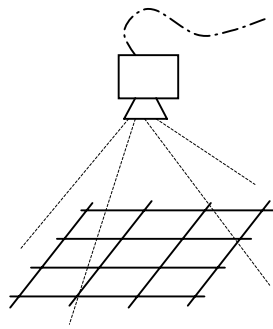
Then mount a bump sensor so that it switches on when the robot bump into something on its front. You can find some example onto the Constructopedia by Lego. Be careful: the sensor must be switched on (i.e. pushed) when the robot bump into something on its front.

No other configurations of the sensors are supported.

Finally you must mount the two engines so that one controls the wheels on the right of the RCX and the other controls the wheels on its right. The engine that controls the right wheels must be connected to the input C, the other to the input B.

The Environment

To build the environment you need a really black tape (in the lab has been used the black DUCK tape and it resulted to be really good) and a huge white sheet (as much "whiter" as you can!).

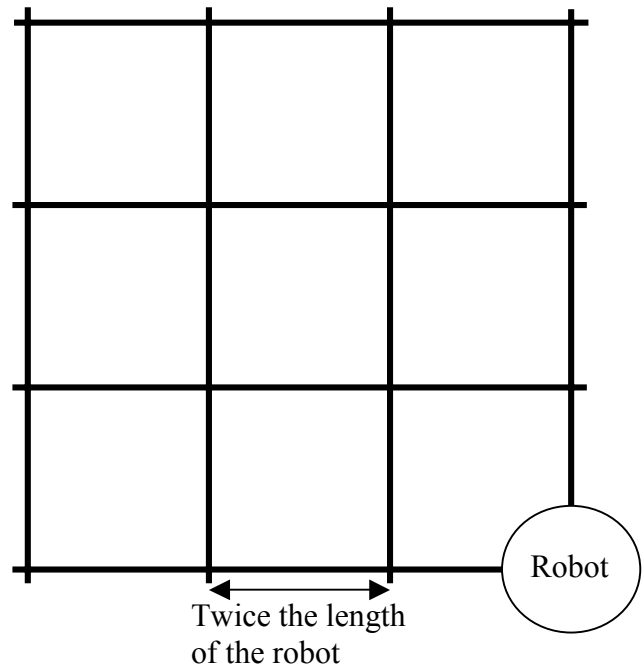


Cut the tape so that its width is larger twice a

light sensor (if you have the DUCK tape just cut it in the middle and it will be really good). Put the cut tape onto the sheet of paper as

shown in figure. Pay attention to the fact that the distance between a node and another must be at least twice the maximum length of the robot or more.

Put the camera on the top of the labyrinth so that it can look at the whole labyrinth. In our lab, we have put the camera on the roof.



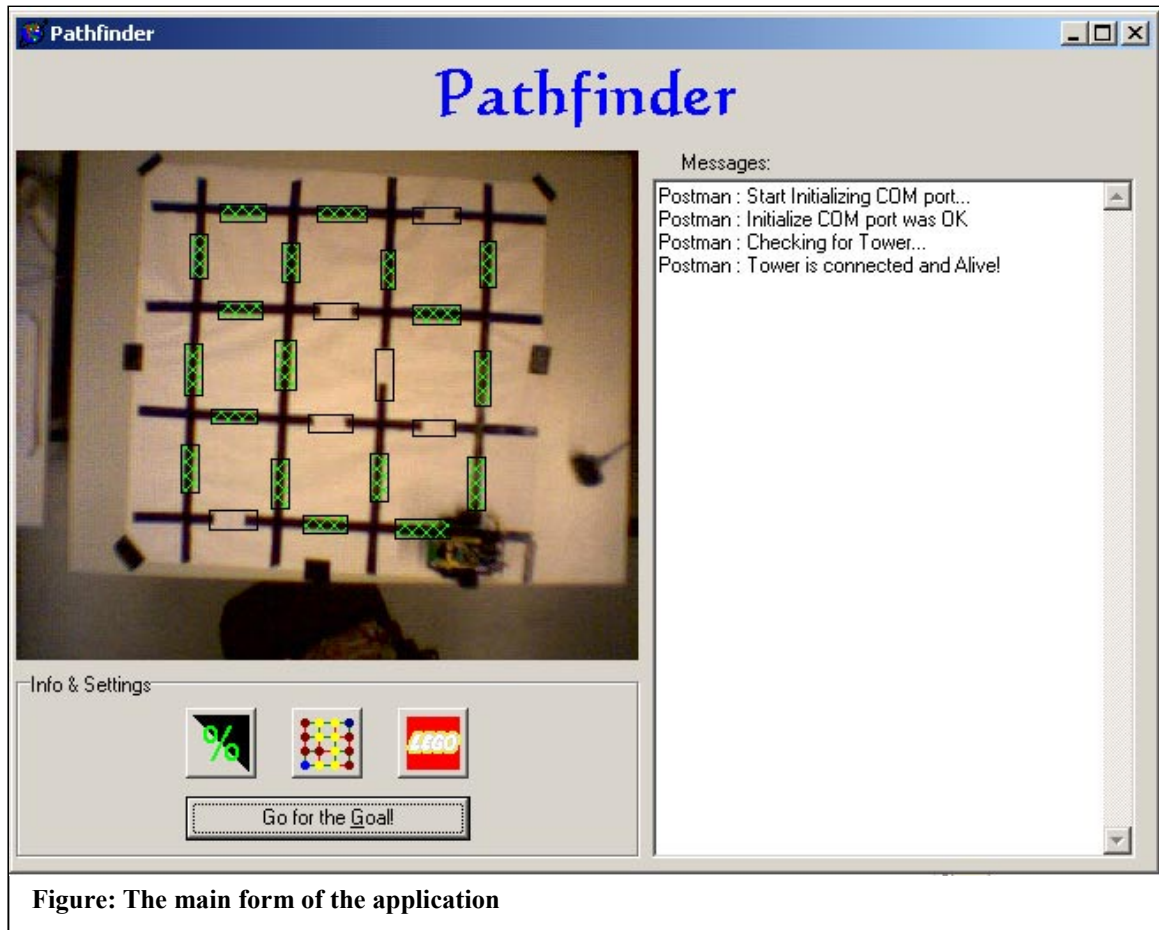


Figure: The main form of the application

The Application

If you think that you're ready with the robot and the environment then it's time to test it. Before starting remember to check the batteries of the Tower and the RCX. Then make sure that the tower is connected to the PC. If it is the first time that you run the application you should follow these paragraphs. If you already downloaded the program onto the RCX and already calibrated the sensors you can skip these two sections and go directly to the simulation part.

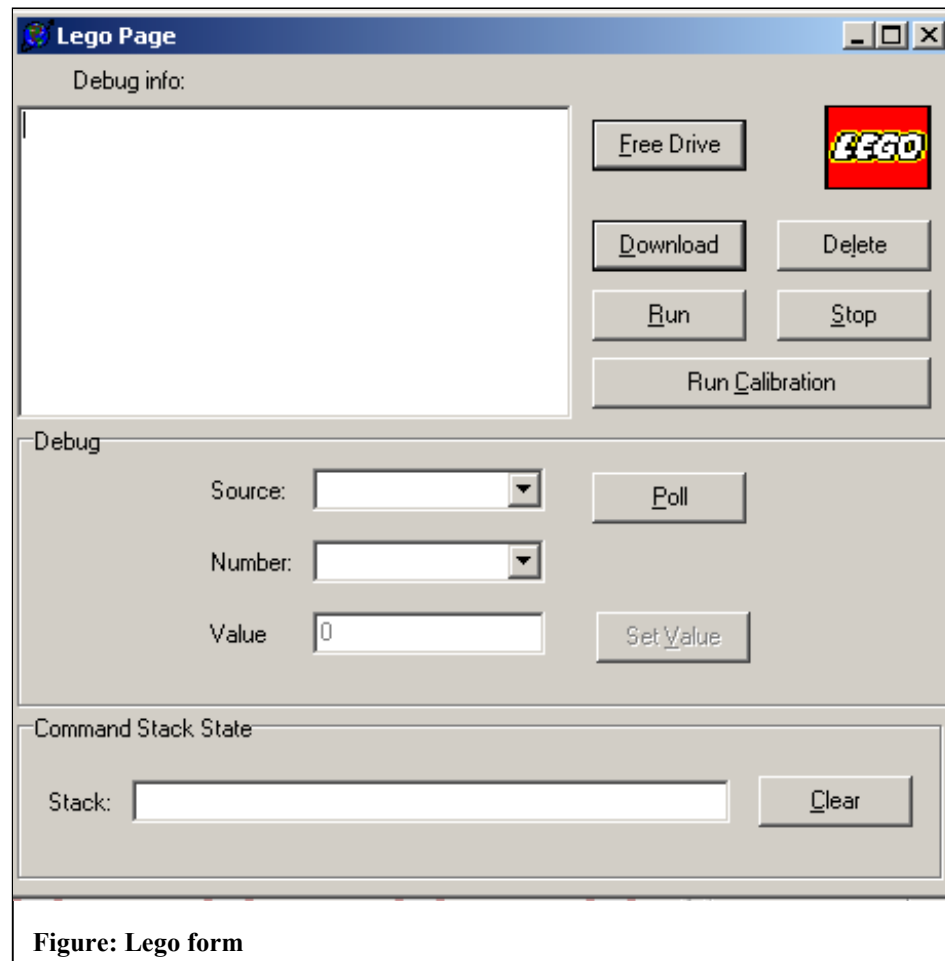


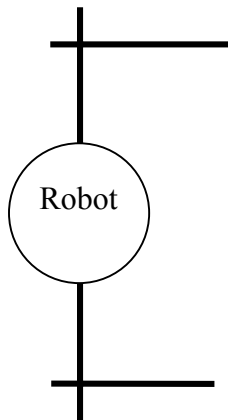
Figure: Lego form

Section 1: Download

Take the following steps to download the programs onto the RCX:

- 1 put the RCX in front of the IR Tower and turn it on
- 2 run the application
- 3 Push onto the LEGO button and the Lego form will appear
- 4 Push the button download and wait for all the download to finish

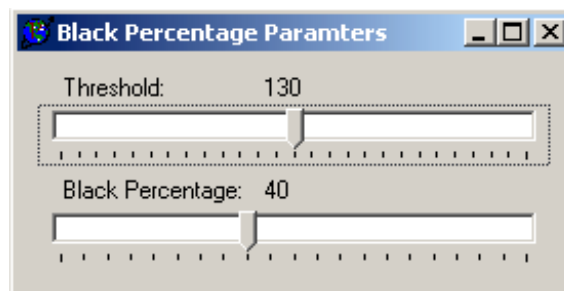
Section 2: Sensors Calibration



To calibrate the sensors:

- 1 Put robot on the middle of an edge like shown in figure and turn it on
- 2 Run the application
- 3 Onto the main form push the Lego button and the Lego form will appear
- 4 Push the button calibrate and wait for the robot to finish the calibration

Section 3: Choose the parameters for the local sensors



Usually the default parameters are ok for the most of the environments. However the Threshold parameter sets which is the edge for a pixel to be recognized as black or as white

and the Black percentage sets how much a sensor must be filled of black to be set on. The last parameter should be changed to find an optimum parameter for your environment. The steps to follow are these ones:

- 1 Connect the webcam to the pc*
- 2 Run the application*
- 3 Run the LucaWebCamControl application*
- 4 Free the whole maze*
- 5 Choose the Black Percentage value that lets all the local sensors to be green*
- 6 Now put white obstacles onto the whole maze*
- 7 Choose the Black percentage value that lets all the local sensors to be transparent*
- 8 Repeat from step 4 till a good value of Black percentage is found*

Section 4: Path finding simulation

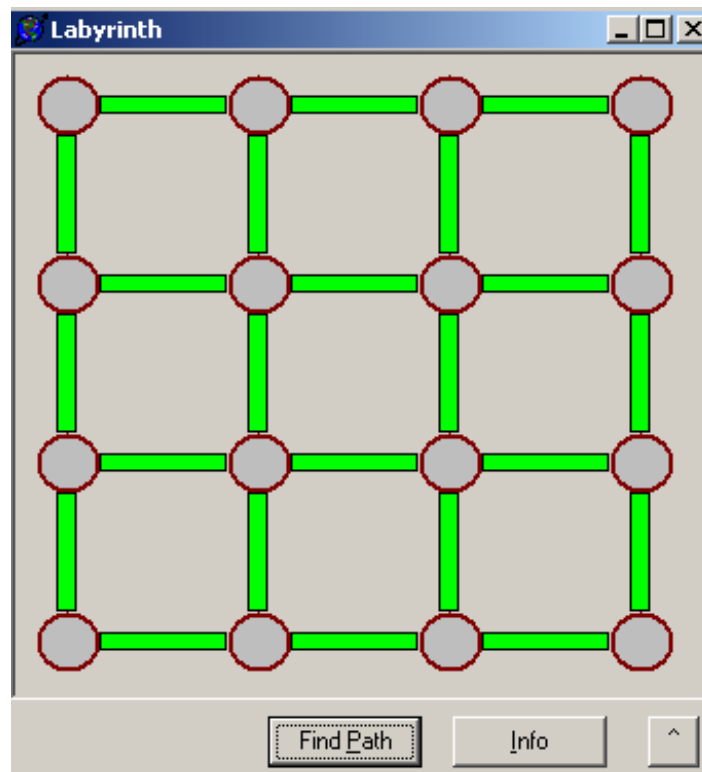


Figure: The Labyrinth Form

Follow the following step:

- 1 Put the robot onto the maze on a cross with both of the sensors over a line as shown in figure and turn it on*
- 2 Connect the camera*
- 3 run the application*
- 4 run the LucaWebCamControl application*
- 5 Push the Labyrinth button on the main form and the Labyrinth form will appear*

- 6 *With the Left mouse button choose the starting point on our maze (i.e. where the robot has been positioned)*
- 7 *With the right mouse button choose the goal point*
- 8 *Push the Arrow button (the button on the right of the info button) till it reaches the direction towards which the robot is aiming*
- 9 *In the main form push the button 'Go for the Goal' and watch the robot move (don't forget to turn on your speakers!)*
- 10 *While the robot is moving you can put some white obstacles on the track so that the robot must change its way*

• *Appendix C: The Source Code*

In these appendix we provide the source code built for our purposes. To compile and run this source code you need:

- ❑ *Borland C++ Builder 5.01 or superior compiler supporting the VCL classes*
- ❑ *Spirit ActiveX control by Lego installed and registered on your computer*
- ❑ *The robot*
- ❑ *The webcam contained in the vision kit by Lego*

Since this application makes extensive use of image processing functions, a good computer will improve the velocity of the application but it is not essential. We notice that we need an external application that loads the windows' clipboard with the frames coming from the Lego Webcam. Such an application can be found in the software provided together with this documentation and it is called LucaWebCamControl.

Classes provided:

TFormPathfinder: *main form. Provide a user-friendly interface with the application. Handle the image coming from the clipboard and the positioning of local sensor. It provides also the local sensors run and black recognizing. The class is able to store in a file the configuration of the local sensors: anytime the application is started, the configuration is loaded and anytime the application is closed, the configuration is saved. The configuration file is called: pathfinder.dat*

TFormLego: interface with the robot. This class uses the Spirit ActiveX control. This class handle the communication from and to the robot and provides synchronization to the TFormLabyrinth Class for the position of the robot. This class also provide some debug, calibration and download functions.

TFormLabyrinth: GUI interface to show the internal representation of the labyrinth. This class is used to set the starting and goal positions and eventually to 'artificially' obstruct some edges of the maze. The Info button calls the TFormHelpLabyrinth that explains how to use this window.

TFormHelpLabyrinth: Form used to help the user understanding the usage of the labyrinth window.

TDialogFreeDrive: This form when called by the Lego Form let the user have a direct control over the robot by using the numeric pad.

TFormParameter: This form sets the sensibility of the local sensors.

TFormCredit: generic credit form

Graph: this class encapsulates the graph storing interface and the pathfinding functions.

IDNode: this class generates a unique ID number for each node.

CostStruct: class used to compare the cost of a node.

Node: this class contains all node info. It contains also a list of linked nodes

NodeList: class implementing the Node Heap.

