Cognitive Approach to Robot Spatial Mapping

A
thesis
submitted in partial fulfilment
of
the requirements of the degree
Bachelor of Science (Honours)
of
Rhodes University

Ghislain Fouodji Tasse

November 9, 2009

Abstract

One of the major issues in Robotics or more precisely in Intelligent Systems in current research and innovation is self awareness. The world of innovation in Robotics has been developing autonomous robots with the ability to reason, learn and accomplish basic tasks. Self mapping is one of these abilities and it is very crucial to any physical agent that claims to be aware of its environment. To validate this claim, many approaches are being used, the most successful of which are cognitive approaches. Part of this document discusses these approaches together with their applications in robot mapping. Further, it proposes a new cognitive approach to solve the mapping problem. Based on the concept of an occupancy grid, the proposed solution appears to be a valid and efficient solution to the problem. In fact, we demonstrate in this report the results achieved by implementing this approach to solve the mapping problem in an indoor environment.

Acknowledgements

I am grateful to the sincere attention and support that my supervisor, Dr. Karen Bradshaw offered for the successful completion of this project. It was a remarkable experience to work with her and to overcome difficulties that were encountered throughout the project. Additionally, this work was greatly supported by the Rhodes University Computer Science Department as a whole. Through its Centre of Excellence, I received financial and technical support from Telkom SA, Business Connexion, Comverse SA, Verso Technologies, Stortech, Tellabs, Amatole, Mars Technologies, Bright Ideas Projects 39 and THRIP. Finally, I acknowledge all my relatives, colleagues and friends that contributed to the overall success of the project.

Contents

List of Figures

List of Algorithms

Chapter 1

Introduction

1.1 Problem Statement

The aim of the project is to design and implement an agent able to learn from its environment, gather new information, and record this in its knowledge base. Through some reasoning and common sense, our robot should at least have the ability to understand its surroundings, and generate a map of the objects and their positions. To achieve this level of cognition, the project goal is to propose and discuss a spatial modelling that can be implemented in a programming language. This goal can be subdivided and categorised as follows:

- Collection of data: the robot should be able to collect data from its sensors.
- Data integrity and interpretation: consistency and validity of the data are encouraged. The robot can then interpret the data to derive information.
- Building the map: a robust map should be implemented by incrementally inserting known information into an initially empty map.
- Usage of the map: the robot should finally be able to use its map to navigate efficiently in the environment.

1.2 Background

As an overview of the research, an explanation of the two main concepts evoked in the project topic is necessary. The first of these is cognitive science, which is the study of the nature of intelligence. A cognitive approach may, therefore, be seen as employing reasoning, self

learning, and intelligence to resolve a particular problem. In artificial intelligence, this approach is concerned with an in-depth study of fundamentals and mechanisms of intelligence formation, how it evolves into knowledge and how it can be implemented in robotics [1].
The second concept is spatial mapping which implies representing the environment at a particular level of abstraction which in our case is a mathematical model [4]. We are talking about mapping the space into logical information or knowledge that can be understood and utilised by an intelligent robot. A robot uses a number of different sensors to ascertain its surroundings. For example, ultrasonic sounds and laser light emitted respectively from ultrasonic and vision sensors are used to determine the distance between the sensors and an object using the time of flight of the reflection. So, range sensors provide spatial information (size, shape and texture information), depending on their accuracy [3]. Hence, the basic requirement for a map modelling is the sensor information collected during robot navigation. Now we should be ready to design a map corresponding to a collection of data. Since we want to use these data efficiently, we first consider the concept of data integrity.
This is of great concern because it is difficult to deduce or conclude that two different observations refer to the same object. In our world of robot mapping, this difficulty is referred to as the correspondence problem [2]. A method for approaching this problem is noticing that the data are collected in a local frame of reference and therefore, must be transformed into the frame of reference used in the map (we call this the global frame of reference). So, it is a matter of answering the question "Where am I?" or "To which frame of reference is this observation meaningful?", for each record of data. This view of the problem is called self localisation. These concepts and issues are continually referred to throughout the project.

1.3 Overview

In this document, we expose the magic behind cognitive mapping approaches in spatial robotics. For this purpose, we dedicate Chapter 2 to relevant attempts and related work done by other researchers to solve the mapping problem introduced in Section 1.1. In Chapter 3, we view the problem from all its different perspectives and as a result, propose our own cognitive mapping approach. We present a conceptual overview of the proposed solution. Subsequently, the approach is implemented and built into a robot. In Chapter 4, we explain certain implementation considerations and details which are necessary for a deeper understanding of the approach. Chapter 5 is dedicated to an evaluation of our implementation. Finally, in Chapter 6, we conclude the presentation of our approach and present further work in the area.

Chapter 2

Related Work

2.1 Introduction

Several issues need to be considered while building a map in an indoor environment. One of these is the concurrent mapping and localisation problem, which is a pivotal problem in mobile robotics. If the position of the robot is known, building a map becomes straight forward as shown by Elfes in [3]. To handle these issues, different algorithms use different approaches which are based on different hypotheses. In this chapter, we describe these differences by presenting various cognitive approaches for robot spatial mapping. To introduce these concepts, the first section of the chapter is dedicated to robot spatial perception. The way the robot perceives its environment determines which mapping approach should be used and how it should be applied. In the second section, we describe some of these mapping approaches. Before attempting to implement these approaches, it is important to explain the Simultaneous Localisation and Mapping problem which is an important factor in mapping algorithms. More details on this problem are given in the third section of this chapter. Then, with the knowledge gathered in the previous sections, we appropriately describe the subsequent mapping algorithms and their characteristics. We also present a sample robot on which these mapping algorithms can be tested and show how their efficiency also depends on the physical capabilities of the robot.

2.2 Robot Spatial Perception

2.2.1 Overview

As artificial sensors and organic sensors are different, the robot's perception and human's perception of the same environment are also different. Regardless of this difference, a robot by definition is expected to interact with the world in the same way that a human does. This is only true if a robot spatial representation carries the same information as a human spatial representation. To satisfy this criterion, two notions of space representation are defined and considered: the notion of a representation for the local space, i.e., the small area of the environment the individual is currently in, versus a global representation in which the individual's total experience of its spatial environment can be represented using a single coordinate system. Related to this contrast, is the contrast of a metric representation, where properties such as distance, size and location are explicitly or implicitly represented, versus a topological representation where relationships such as connectivity between individual elements are represented [10, p.2]. More details on these representations are given in the following subsections.

2.2.2 Metric Representation

A metric map is the capture of the geometric properties of the environment [15, 16]. By geometric properties, we refer to the geometric relations between the objects and a fixed frame of reference defined in the map. This is perhaps the most explicit map in robotics since it explicitly represents the occupancy of space by storing the exact position of objects in a global frame. For example, a tourist's scale of a map is a metric map [2, p.213]. So, a metric map only reproduces the spatial state of an environment, which carries no functional information (see Fig. 2.1).

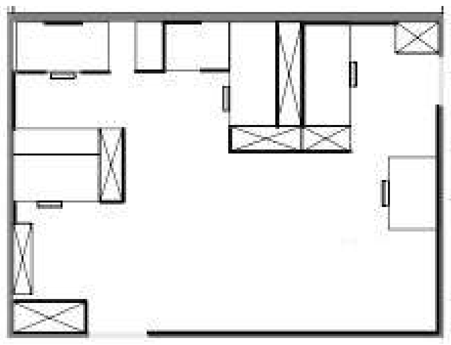


Figure 2.1: An office of the institute and the lines representing it in the local metric map [16]

This type of map is commonly used to represent the environment as a two dimensional space in which it places the objects. With such a map, the information available to a robot is the object's precise coordinates [x,y,theta], where the pair (x,y) gives the object's position and theta its orientation [5, p43]. Hence, the precision of the information given by a metric map depends to a large extent on the quality of sensors. As sensors are often subject to noise, this dependence is a weakness of the metric map. Additionally, this weakness becomes more relevant as the map increases in size.

2.2.3 Topological Representation

A topological map naturally captures qualitative and relational information from the environment. For example, a subway map is a topological map [2, p.213]. It represents the environment as a list of significant places connected via arcs. The latter usually carry information on how a robot can travel from one place to another. Unlike the metric map which is an absolute representation, a topological map represents objects relative to one another. Hence, it can be simulated by a graph in which the places correspond to nodes and arcs correspond to paths which connect two nodes if they are adjacent in the real environment [13, p.597]. Fig. 2.2 shows an example of a topological framework that only considers places and the relations between them.

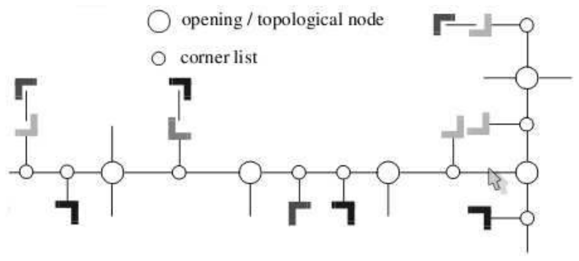


Figure 2.2: The topological map is represented by a graph. It contains nodes connected to each other with a list of corner features lying between them [16].

However, using topological representations, one cannot easily determine a previously visited part of the environment if it is approached from a different side [10, p.1]. Effectively, each place is only identified by the arc that leads to it. So if the arc or path used by the robot is not shown on the map, the place will be marked as unknown.

2.2.4 Hybrid Representation

From the explanation giving above, metric and topological maps are different representations of the same environment. Since each focuses on different aspects of the environment, they can be combined to give a *robust* map which contains both qualitative and quantitative information of that environment [17]. The idea is to enrich the topological map with metric information. This is achieved by representing each node (place) in a topological framework with a local metric map as shown in Fig. 2.3.

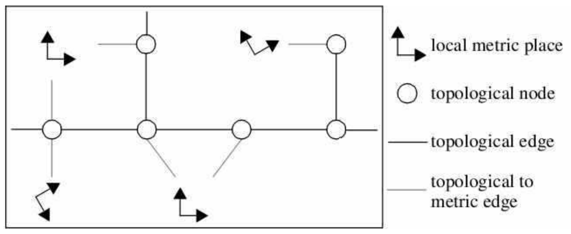


Figure 2.3: The environment is represented by places given by their metric maps and nodes representing topological locations.

So to move from one place to another, a robot moves metrically in that place and then outside the place, it moves topologically till it reaches the goal place where it switches back to the metric map[16]. Since metric maps only represent local spaces, this method contributes to the reduction of the impact of noise on the map. This is the description of a hybrid representation.

2.2.5 Mapping Problems

Although many approaches focus on representing the environment in the best way, there are some additional issues that need to be considered. These issues are difficult to handle since they depend on the nature of an environment which appears to be unknown.

Odometry Errors

Robot sensors are naturally limited in what they can perceive. These limitations are either induced by the sensor itself or by the nature of the environment. The range and resolution of a sensor is subject to physical limitations. For example, the resolution of a camera image is limited. Sensors are also subject to noise, which perturbs sensor measurements in unpredictable ways and hence limits the information that can be extracted. *Noise* is a global term which is used to describe wheel slippage or surface imperfections.
Additionally, odometry errors can be caused by robotic software. Models are just abstractions of the real world [19]. Uncertainty in the map data can be created through algorithmic approximations. Furthermore, robots are real-time systems and can only carry out limited computational operations. Because of this, many popular algorithms are approximate, and achieve timely response by sacrificing accuracy.

Dynamic Environments

Most mapping algorithms assume that the environment is static. This assumption reduces the computational complexity of the algorithm by eliminating some variables that should have been considered. But these algorithms are most likely to fail in the real word which has a continuously changing environment. To overcome this problem, most mapping algorithms consider and handle places with high dynamics as noise [4].
However, the resulting map is still static and the dynamics of the environment is ignored. As a better solution to the problem, Nikos and Costas [12] propose a filtering algorithm: the Temporal Occupancy Grid Algorithm. This algorithm classifies objects into three categories: static objects (e.g. walls, bed), objects with low dynamics (e.g. chairs, doors) and objects with high dynamics (e.g. humans). In so doing, the algorithm can model the dynamics of the environment.

2.3 Simultaneous Localisation and Mapping (SLAM)

When a robot is placed in an unknown environment, it discovers its environment whilst navigating there in. So the map is continuously updated whilst the robot gradually moves through the environment. Fig. 2.4 is a flow chart demonstration of the SLAM process.

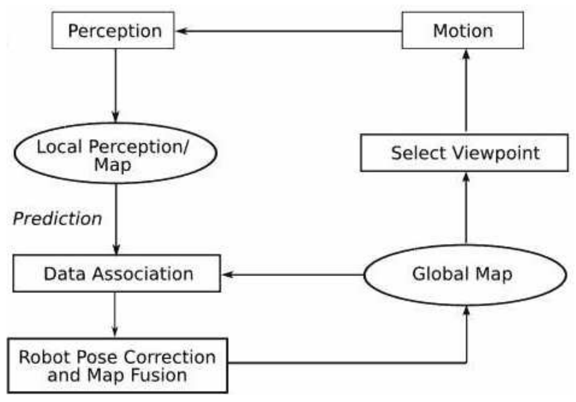


Figure 2.4: Flowchart of the SLAM process [5].

But, SLAM posed a serious difficulty in mobile robots. The SLAM problem asks if it is possible for an autonomous robot to be placed at an unknown location in an unknown environment and for the robot to incrementally build a consistent map of this environment while simultaneously using this map to compute its location [1]. Effectively, it is practically impossible to localise an object without a map or raw knowledge of the environment. It is like looking for a point in a graph without the graph itself. Conversely, it is difficult for a robot to build a map if it has no knowledge of its position in the environment, which is similar to building a graph without an origin.
Therefore, both localisation and mapping should be implemented simultaneously. For mapping, it is sufficient to localise the initial position of the robot which will be the "origin or starting point" of the map. Since we are in an unknown environment, various cognitive mapping approaches define their starting points in different ways. But generally, to solve the SLAM problem, landmark based approaches are used in building the map.

2.4 Cognitive Mapping

2.4.1 Overview

Human and animal brains employ very intelligent methods to build their own knowledge or representations of their environments. Using these methods, they end up with a relatively robust map which can be used confidently during navigation. We define a "robust map" as an "intelligent" map in which the capabilities of reasoning, self awareness and adaptation to dynamic environments have been implemented. A common expression for referring to a "robust map" is: cognitive map. What is of importance here is not the resulting maps but the processes used in their construction. This section focuses on explaining diverse and successful mapping methodologies as proposed by researchers.

2.4.2 Shape based Mapping

Overview

This approach studies the geometrical configuration of the environment to be mapped. We believe that if one can get enough information about the size and relative positions of each object in the environment, one can build a robust map. Accordingly, the shape based approach aims at representing objects in space by defining mathematical relations between them. This results in a graph (the map) with a set of points (objects) and mathematical functions. The pioneer of this approach is a researcher called D. Wolter who was the first to propose a novel geometric model for robot mapping [21]. He affirms that this approach is an improvement to bridge the gap between metric information and topological information. Figure 2.5 illustrates his architecture for the shape based approach.

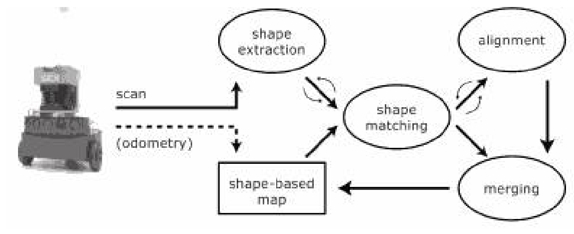


Figure 2.5: Shape based Architecture.

Methodology

In his book [20], Wolter proposes a shape based approach which combines a boundary based approach together with a structure based approach. The structural approach represents shapes as a colour graph representing metric data alongside configuration information [21]. But this approach appears to face difficulties in identifying shapes lacking structural information. This could be due to the error in range sensor data. So he decided to consider a boundary based approach which focuses more on the boundaries of obstacles than their overall structure. Shape is represented as a structure of boundaries in which the boundaries are defined using polygonal lines (polylines). So a polygonal map is a set of polylines (which describe obstacles) and vectors of polylines (which establish the relations between two obstacles). Using this method, let us describe how information (an obstacle' s shape) can be extracted from an unknown environment.
For the purpose of retrieving this information, D. Wolter proposes a simple heuristic: "Traversing the reflection points in a (cyclic) order as measured by the LRF (Large Range Finder), an object transition is said to be present wherever two consecutive points are further apart than a given distance threshold". But this heuristic does not remove noise in the sensor readings. So we introduce a technique called Discrete Curve Evolution (DCE) proposed by Latecki & Lakamper to first make the data more compact without losing valuable shape information and next, to cancel out noise. DCE is a process which proceeds iteratively: Irrelevant vertices get removed until no irrelevant ones remain (see Fig. 2.6). Though the process is context-sensitive, it depends on a vertex v and its two neighbour vertices u and w according to the following formula:

$$\mathrm{K(u, v, w)} = |\mathrm{d(u, v)} + \mathrm{d(v, w)} - \mathrm{d(u, w)}| \qquad [21]$$

If K(u,v,w) is less than the given threshold, then vertex v is removed.

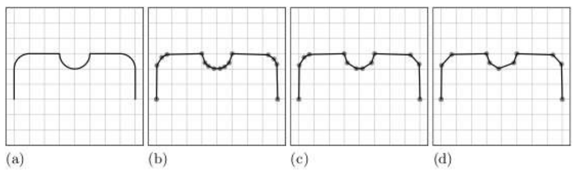


Figure 2.6: Illustration of the DCE technique.

With the computational processing power of actual robots, this approach seems to be very successful.

2.5 Cognitive Robot Mapping

2.5.1 Probabilistic Algorithms

Every algorithm used in robot mapping should be able to handle uncertainty. As discussed in Section 2.5.1, sensor readings are not exact and can badly influence the robot' s perception of the environment. This problem is solved or partially solved at two levels: the physical level, which means improving the sensor accuracy, or the software level, which leads to probabilistic algorithms. Most mapping algorithms use the calculus of probability theory to improve the reliability of their resulting map [19]. Instead of relying on a single "best guess" value from sensor data, the probabilistic algorithm computes a probability distribution over a set of sensor values [14]. So the sensor values used in the algorithm computation are likely to match those in practice. In the algorithm flow chart, this computation is referred to as data integrity or data validation. Before constructing the map (the final output), the algorithm sanitizes the inputs giving by the robot range sensors. As a result, a probabilistic robot can gracefully recover from errors, handle ambiguities, and integrate sensor data in a consistent way.

2.5.2 Grid Algorithms

Overview

Grid algorithms are a very simple but efficient method for spatial mapping. The grid-occupancy representation treats the world as an unstructured array, composed of cells that are independently either occupied or unoccupied [5, p.137]. Depending on the algorithm, the cell can be any polygonal (see Fig. 2.7). This approach eases the construction of maps and spatial reasoning based on those maps. Ultimately, it handles sensor data noise by estimating the size of the cells using probabilistic sensor models [3]. Occupancy grid representation gives the robot sufficient information for its navigation and path planning. A number of robotic tasks can be accomplished through operations performed on such a representation. So the basic idea of an occupancy grid algorithm is to partition the space into cells where each cell is qualified with probabilistic estimates of its state (occupied or empty).

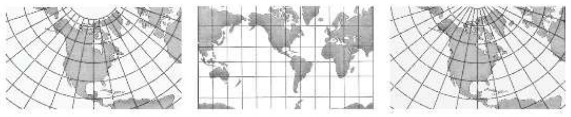


Figure 2.7: Different projections maintain different kinds of spatial knowledge, which leads to different forms of grid cells [7].

Methodology

An occupancy field is a function O(x) where x belongs to a set of continuous spatial coordinates, x=(x1, x2, ..., x3) [3]. Let us define a cell C as C=(i,j) and its occupancy state as occ(i,j) where i,j are the coordinates of the cell in the unstructured array. So, the function O takes x and maps it into the corresponding cell (i,j coordinates). We also define a probability function P which given a cell, returns the probability of its occupancy state P(occ(i,j)). occ(i,j) is a discrete random variable with two states, occupied and empty, denoted by OCC and EMP respectively. Since the cell states are exhaustive and exclusive, P[occ(i,j)=OCC] + P[occ(i,j)=EMP] = 1. A good probability function is given by Bayes' theorem which takes as variables two sensor readings of the same cell.
In summary, grid occupancy mapping involves two steps: the first step is to represent the space as an array of cells; the second step is to determine the occupancy state of each cell. Using a probabilistic approach to estimate the cell's state overcomes the problem of generating maps from noisy and uncertain sensor measurement data. But the weakness of grid algorithms is that they are useful only when robot poses are well known.

2.5.3 Local Space Representation

Overview

The environment in which a robot is currently navigating is referred to as the local environment. In the case of an indoor environment, a room is an example of a local environment. Representing a local environment is the first step in cognitive mapping. The robot should map its current environment as it experiences it. Contrary to grid mapping where the position of the robot is assumed to be known, a local space representation should define its own starting point. Yeap [42] argued that an important basis for computing a cognitive map is the ability to compute and recognise local environments [23]. So a mapping algorithm must first recognise the local environment and define a coordinate system (the starting point) for it. But the difficulty is to determine where one local environment ends and another begins

[23]. Of course, the algorithm should know which space to map before starting to map it. The general problem is to find suitable connections between surfaces to form a boundary surrounding the viewer. Finding the boundaries of a local environment is analogous to finding exits in that environment.

Methodology

Ref. [23] proposes a cognitive approach for identifying exits described as follows: ”Whenever one surface is viewed as occluded by another surface, a gap exists which we label as an occluded edge. The occluded edge and the exit are thus virtual surfaces. An exit is the shortest edge covering the occluded edge.” First, the surfaces in the current view are divided about the occluded edge FG so that F is in group I (FD, DC, BA) and G in group II (GI, IJ) (see Fig. 2.8). Then the exit is identified by taking the occluding vertex F and connecting it to the nearest vertex on a surface in the group opposite to it, i.e. group II. As shown in Fig. 2.8, J is the nearest point to F. Hence, JF is an exit.

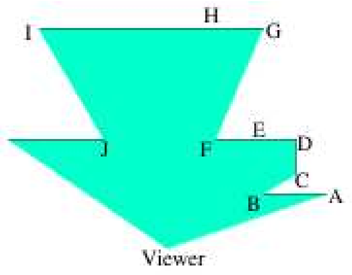


Figure 2.8: A view of the environment (in 2D) [23].

2.5.4 Global Space Representation

Global Space representation is used in the case where we wish to represent not only a room, but the whole house. A global or single coordinate system is defined relative to which the map of each room is computed. But this method increases noise in the inputs. In fact, the robot has to navigate for a longer time and cover a larger space before generating a usable map. If even one data is invalid, the whole map may also be invalid too. This is the reason why instead of using this approach, researchers propose that a global map can be generated by building a network of local environments. In others words, each room is mapped independently and then these maps are combined to give the map of the house. In this case, our local space representation defined above is referred to as an Absolute Space Representation (ASR). ASRs are used to identify and describe local environments which have

been visited by the viewer [24].
Moreover, the different local spaces that have been computed can be connected together in the way they are experienced to form a topological network (see Fig. 2.9). So the robot perception of the environment can be defined in a topological representation, as a collection of local space representations, each with its own coordinate system, and connections between them which will allow the robot to travel from one to the other. Since this method maps the environment as the agent is navigating through it, the connections between ASRs result in how one passes from one ASR to another. In other words, the algorithm should figure out which rooms' exits match, since practically, the exit from one room is the entry to another room. This idea was developed by Yeap and Jefferies [10] and is referred to as a topological network of metric local space representations.

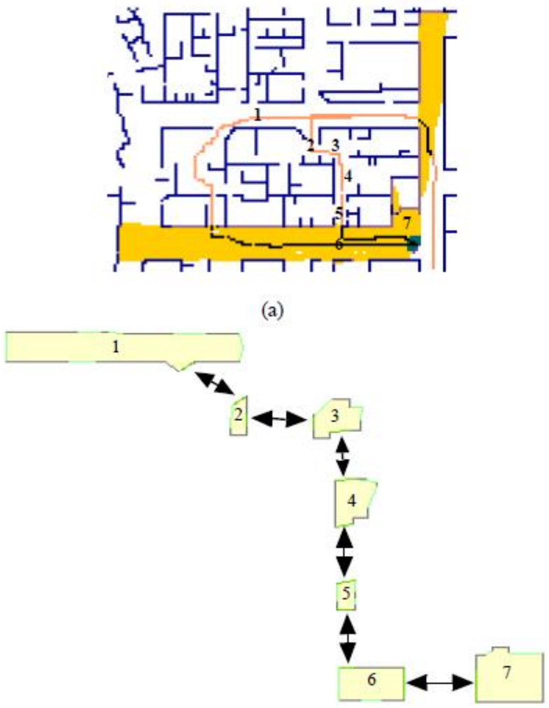


Figure 2.9: The topological network of ASRs computed as a simulated viewer follows the path in (a).

However, this approach is incomplete when it comes to navigating using the computed map. Effectively, connections between the ASRs are defined as the robot leaves one ASR to enter another. So not all the possible connections are defined, but only those experienced by the robot. Kuipers and Byun [9] propose a solution to this using topological matching [10]. They use the information of a current ASR to find all possible connections to it. Then the robot follows each found path to validate the connection and come back to its initial position. In so doing, the network represented in Fig. 2.9 can be extended to the one in Fig. 2.10.

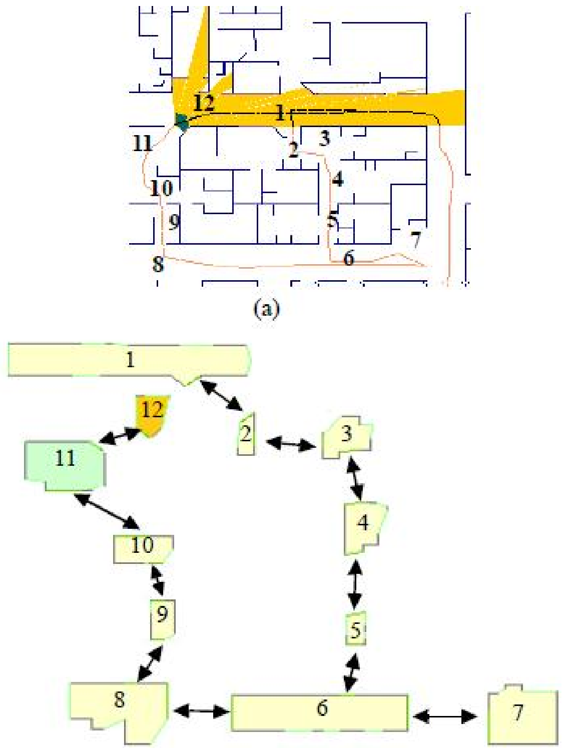


Figure 2.10: An extended topological network of ASRs.

2.6 Path Finding Algorithm

2.6.1 Overview

As an add-on to a cognitive mapping approach, a good path finding methodology can be developed. Path-finding on its own is an area of speculation among robotics researchers, but, with respect to a given mapping methodology, there is a corresponding path finding algorithm that is suitable. Such an algorithm is basically a search algorithm which operates on the constructed map to generate an obstacle-free path from one place to another. It uses the search algorithm to find obstacles in the map so as to avoid them. Path-finding is concerned with issues such as the shortest path, least-cost path, safest path, etc. It is implemented in most game programs by the A* search algorithm [22] is optimal and complete. Depending on the practical requirements of the game, the IDA* (Iterative Deepening A*) which avoids A*'s memory overhead can be used. This is a combination of the A* and iterative deepening depth-first search. But, it is usually more computationally expensive than the A* and its time complexity is proven to be proportional to O(b^{D-H}), where D is the distance to the goal state, b is the average branching factor and H is the effect of the heuristic.

2.6.2 Basic A* search algorithm

In this section, we review the A* search algorithm as presented by Heyes Jones [6]. The algorithm is simply a best-first graph search algorithm which results in a least-cost route from an initial state to a desired state (goal state). A* is generally characterised by its use of several methods which guide the search through a tree: these methods are called heuristics. As the emphasis is on the cost of the path, heuristics are used to generate a solution rapidly and help cut down the complexity of the search problem. There are typically two types of heuristics:

- Distance heuristic (d(x,y)): this is based on a function which determines how far a node (x) is from the goal. Generally, nearest nodes are expanded first.
- Cost heuristic (h(y)): this is based on an estimation of the cost of the "solution" path (the path from the initial node to the goal node).
  The above two heuristics are used to define a condition that restricts the expansion of new nodes during the search. Furthermore, nodes in the graph can be expanded in either a depth-first or breadth-first search manner. In so doing, the search ends up with a "best" path solution.

2.7 Application: Lego Mindstorms NXT

Lego Mindstorms NXT is a mobile robot designed by Mindstorms which has the basic and sufficient properties to perform autonomous spatial mapping. It is equipped with a NXT brick which acts as a CPU in the robot architecture. These are the technical specifications of the NXT brick taken from the Mindstorms website [11]:

- 32-bit ARM7 microcontroller
- 256 Kbytes FLASH, 64 Kbytes RAM
- 8-bit AVR microcontroller
- 4 Kbytes FLASH, 512 Byte RAM

- Bluetooth wireless communication (Bluetooth Class II V2.0 compliant)
- USB full speed port (12 Mbit/s)
- 4 input ports (4 sensors) and 3 output ports (3 motors)
- 100 x 64 pixel LCD graphical display
- 6 AA rechargeable lithium batteries.

Two motors are used to drive the robot's wheels and the third one is an extra motor. Four types of sensors can be connected to the NXT brick: ultrasonic, touch, sound, and light sensors. Our focus here is on the ultrasonic sensor, which is the key sensor used by the Lego NXT for mapping.

Ultrasonic Sensor

The Ultrasonic Sensor helps the robot to judge distances and "see" where objects are. Using the NXT Brick, the Ultrasonic Sensor is able to detect an object and measure its proximity in inches or centimetres.

2.8 Summary

In this chapter, we have provided sufficient evidence to show that if a robot is placed in an unknown environment, it will be able to experience navigation without collisions by learning the spatial properties of its surroundings. Cognitive algorithms provide the robot's with this learning capability to the robot thereby making it truly autonomous. In our discussion, we sequentially improve the robot perception of its environment, a cognitive mapping approach that suits this perception and how the approach can be implemented through an algorithm. These three steps in cognitive mapping are correlated and moving from one step to the other requires an understanding and consideration of several issues. These issues have been presented to a lesser or greater degree and corresponding solutions have been discussed where necessary. In this way, we have covered fully the knowledge needed to implement a mobile robot that can learn its environment and navigate autonomously through it.

Chapter 3

Proposed Cognitive Mapping Approach

3.1 Introduction

In this chapter, we propose a solution to the mapping problem described in Chapter 2. The cognitive mapping approach is described in detail and the common mapping issues are discussed extensively. The first section gives a brief definition or presentation of the problem to be solved and some factors that should be taken into account. Subsequent sections are used to show how a robot can gain knowledge of its environment, store the knowledge and finally, use the knowledge. Furthermore, to optimise our knowledge storage strategy, we have decided to use the concept of distributed knowledge which is concisely described in Section 3.4. Each of these components is developed and then combined to build a sophisticated mapping methodology.

3.2 Mapping problem modelling

To demonstrate our cognitive approach, a proper modelling of the problem at hand is necessary. The problem can be specified as follows:

- A robot with four ultrasonic sensors and two motors
- An unknown environment
- An empty map

Our robot is required to navigate through this unknown environment and acquire spatial knowledge that is stored in a cognitive map for future use. For a practical explanation of our approach and without loss of generality, a four-sided room represents the physical design of the environment to be mapped. This is in effect a common design for indoor environments in the real world or a "human's" world. This choice is even more relevant to us as the purpose of our project is to develop an agent dedicated to human personal use (a good example of this is a sweeper robot). Additionally, the word "room" as used in the context of this research, represents a 2D indoor environment with the following characteristics:

- Walls: Though we are dealing with a 2D environment, this word still has the same meaning as that given in most dictionaries. It is always critical for a map builder to identify and remember particular boundaries present in the environment. So in our approach, we identify a wall as a "big" obstacle (obstacle with length greater than a predefined length). With respect to the contextual perception of an indoor environment, the number of walls is logically set to four.

- Exits: These are openings that allow the robot to move in and out of the environment. An exit is identified in a room by detecting openings in the boundaries thereof. A room can have a maximum of four exits; that is, zero or one exit on each side of the room.

- Size: For an objective discussion of the mapping approach, the minimum and maximum size of the environment need to be defined with respect to the robot characteristics. Any environment of size bigger than the defined maximum size will not be identified by the robot as a room, but as a combination of more than one room. And, if the environment size is found to be smaller than the defined minimum size, then the robot will not interpret it as a room either. There is probably no reason for mapping such an environment as it will be too small for any useful navigation.

To clarify the case where the environment is bigger than the maximum size of a single room and the robot has to identify it as a combination of multiple rooms, it is still expected that the robot will be able to create a distinct mapping thereof represented as a single global environment. Thus, we should end−up with a known environment represented by an informative map.

3.3 Designing a knowledge based agent

The aim of this project is to design an agent that will be able to acquire knowledge of its environment cognitively and use it intelligently for navigation. Knowledge is referred to as the interpretation of sensor data (for example, the presence of an obstacle at a certain distance from the robot). So, to achieve our aim, we need both an efficient means for representing this knowledge and a learning mechanism.

3.3.1 Knowledge representation

The occupancy grid

To represent a map, we use the occupancy grid as the means to encode knowledge of the environment derived from robot sensors. This approach is often used in mapping and can provide the robot with efficient and reliable map storage features. The robot virtually superposes a grid on a 2D representation of the room and the state of each cell (occupied, unoccupied or unknown) is determined by the presence of an obstacle in the specific area. Fig. 3.1 shows an illustration of this scenario.
With the above concept in mind, we can model a map using a 2D dynamic grid. At creation, such a grid has an initial size which depends on the previously assumed maximum room size. Then, according to the actual size of the room found as a result of the mapping, the grid is resized. Later in this chapter, we show how this space representation allows us to model the characteristics of a room effectively. The proposed grid structure is illustrated in Fig. 3.2.

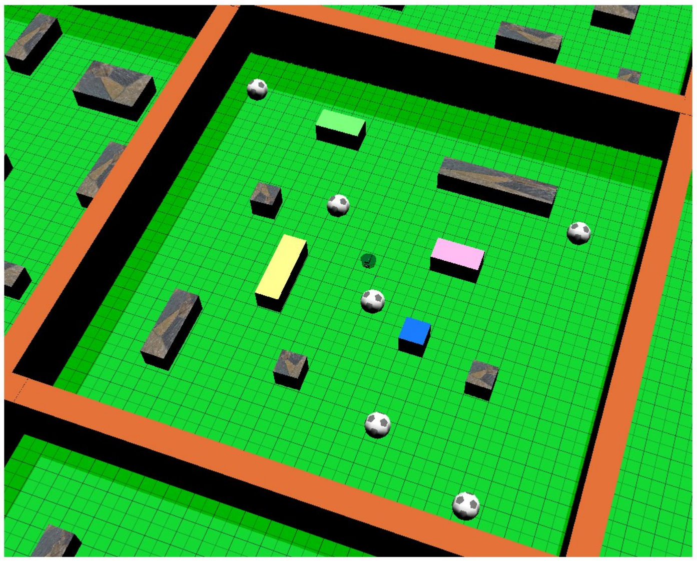


Figure 3.1: Picture of a grid superposed on a sample environment.

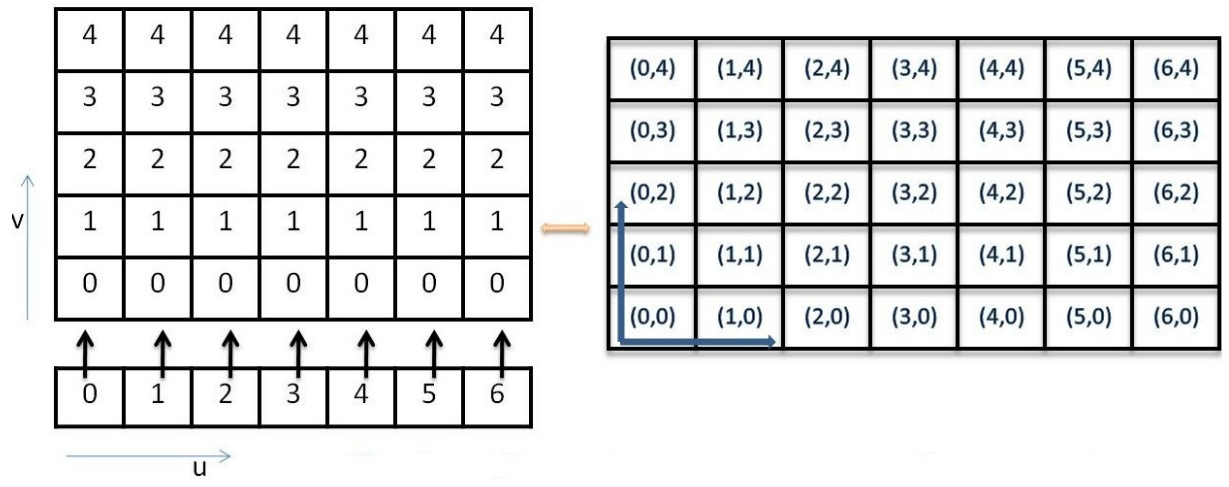


Figure 3.2: A simple grid structure. Each cell is uniquely identified by its 2D coordinates.

Frame of reference

Since our grid is used to model information about a spatial map, it is judicious to define an origin that represents the centre of the map. The latter is arbitrary as the robot has no knowledge of its surroundings. For consistency, we simply choose the cell corresponding to the starting point of the robot in the map as the origin. In doing so, the structure of the grid shown in Fig. 3.2 is altered to produce that shown in Fig. 3.3, using an arbitrarily chosen origin. The pairs of numbers in Fig. 3.3 denote the order of propagation of data in the grid in the horizontal and vertical directions. Cells are simply ordered in the way they are experienced.

However, this knowledge model can be difficult to use for deriving information when the origin is not one of the vertices of the grid. For example, the grid in Fig. 3.3 is much more difficult to interpret than the one in Fig. 3.2. This situation is analogous to that where points in a Cartesian coordinate system are referenced with respect to a given origin (2,2) instead of the usual origin (0,0).

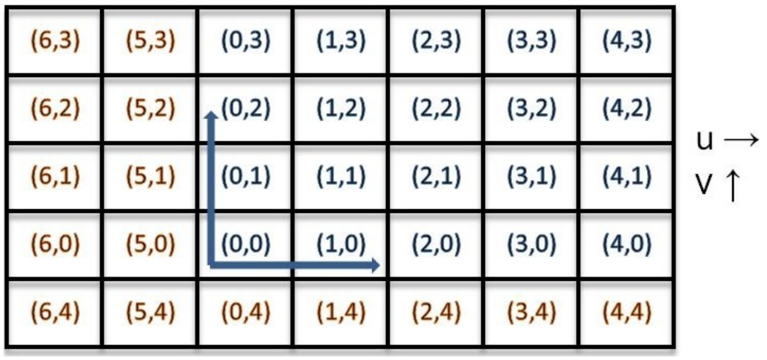
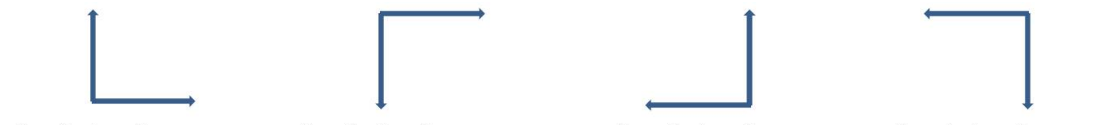


Figure 3.3: Structure of a grid with an arbitrarily chosen origin and coordinate system. Since u and v are positive values, we keep incrementing in the negative quadrant of the coordinate system. $(0,-1) = (0,4)$

Nevertheless, since the problem is mathematical, we can always find a mathematical solution to it. Effectively, the solution is to apply a proper function to rotate and translate the initial coordinate system with arbitrary origin (a,b) (from a local space representation of a room) to a known coordinate system with static origin (0,0) (to form an absolute space representation of a room). With a static origin, the robot can always access a map, that is, its knowledge base in the same way. When the robot has sufficient knowledge of the room, in other words, when it stops mapping, this grid transformation can safely occur. So, one of the benefits of this knowledge representation is that we can always translate the origin of the map, i.e., the structure of a grid as shown in Fig. 3.3 can transformed to the structure of a grid in Fig. 3.2. Subsequently, the grid can have four possible coordinate systems as illustrated in Fig. 3.4.



Case 1: the robot starts by propagating in the positive directions for both the horizontal and vertical axes .

Case 2: the robot starts by propagating in the positive direction for the horizontal axis, but in the negative direction for the vertical axis.

Case 3: the robot starts by propagating in the negative direction for the horizontal axis, but in the positive direction for the vertical axis.

Case 4: the robot starts by propagating in the negative directions for both the horizontal and vertical axes.

Figure 3.4: A presentation of the four possible coordinate systems.

3.3.2 Learning mechanism: Rules based reasoning

A rule based reasoning methodology is used to obtain information of object positions in a room while navigating through it. Initially, we set each cell's occupancy in the grid to "unknown" since the entire environment is unknown. Then, the robot is expected to start at a random position as defined in the subsection above. It has also been shown that the grid obtained is a non uniform grid. Hence the navigation and mapping behaviour of the robot is critically subjected to its referential position in the map. Thus, any rule that changes the state of the robot should be able to take into account each of the coordinate systems represented in Fig. 3.4.
Considering this, when an object is detected at a given position, a rule is applied accordingly to save the information in our knowledge model. This learning mechanism basically relies on two rules which are defined as follows:

- The robot reads as far as possible with its ultrasonic sensor and if an obstacle is detected, the location of the obstacle is saved in the knowledge representation (the occupancy grid). Depending on the chosen constant cell size, the robot can convert a position at relative distance (in cm) to a cell in the grid. Eventually, that cell is marked as occupied in the grid and the grid slots before it marked as unoccupied.
- If no obstacle is found, the robot just marks every grid slot as unoccupied up to where it stops reading. Additionally, the robot has the potential to remember where it stops reading, which may be necessary to avoid redundancy.

3.4 Distributed knowledge: a network of metric maps

Up to this point, we have discussed how it is possible for a robot to learn and build a map of an environment (a room). Now, we need to pursue our approach further and explain how the robot can map , not only a room, but a house (an ordered combination of a number of rooms). To achieve this, we represent the knowledge about a house as a concatenation of individual rooms' knowledge: distributed knowledge. As shown in Fig. 3.5, we can build a house merely by connecting a number of rooms in an appropriate manner. So the aim of this section is to design a model of a house map based on our previously defined model of a room map.
The approach we discuss here is derived from the concept of a topological network where a number of metric maps are connected together to produce a bigger map called a topological map (refer to Section 2.4). Each room in a house is mapped individually and independently,

and once the rooms' exits have been identified, the robot can autonomously leave one room and move to another. As seen in Fig. 3.5, a topological network is a suitable model for a house environment. Nodes (room maps) in such a network are connected by gateways which carry basic knowledge of how a robot can move between the rooms in the house. A gateway relies on the fact that the exit in one room is also the entrance to another. So, to move from one room to another, the robot moves from its current position to an exit and then from the exit (entrance of the other) to the designated position in the other room.
So, we can then build a distributed knowledge representation by combining several independent grids together with gateways.

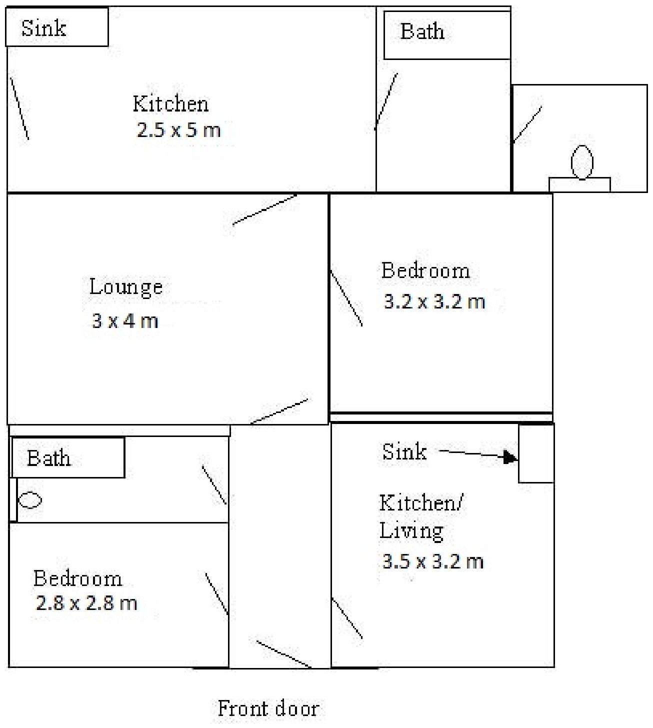


Figure 3.5: Representation of a house.

3.5 Mapping algorithm: Case based reasoning

This is a core part of our proposed approach which describes how the robot navigates through the environment and eventually maps it. The SLAM problem described in Section 2.3. must be considered. The location of the robot in the grid needs to be updated as it navigates and maps the room. To locate the robot in the map, we use a pair (u,v) which gives its position on the grid. As the robot moves around, the values of u and v are updated accordingly;

localisation is thus embedded in the navigation during mapping.

3.5.1 Start-up heuristic

At time t=0, the robot uses a heuristic to determine the initial path it should take. Implicitly, this heuristic also determines which of the four coordinate systems defined in the previous section the robot is going to use. The heuristic function can be a random function that choose one direction arbitrarily or it can be a more clever function which chooses the starting path intelligently. To build such a clever heuristic function, we use the following rule. If one side of the grid is larger (in size) than the other three sides, as initially seen by the robot, then that side should be mapped first. The rationale behind this rule is that, the bigger the area, the greater is the information the robot can obtain from it.

3.5.2 Shortest path algorithm

To increase time and space (redundancy avoidance) efficiency, we have adopted a shortest path algorithm for the robot's navigation during its mapping cycle. The best case scenario is when it always moves following a diagonal path rather than adopting a horizontal or vertical path as illustrated in Fig. 3.6. However, it should be noted that situations can arise where the robot cannot use the diagonal path and is forced to use either a horizontal or vertical path. Fig. 3.7 is an example of such a situation.

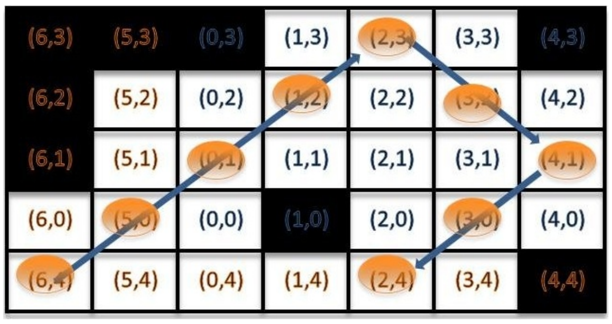


Figure 3.6: Best case scenario when the shortest path algorithm is fully and strictly applied. In this particular case, the robot moves along three paths: forward-right diagonal, backward-right diagonal, backward-left diagonal.

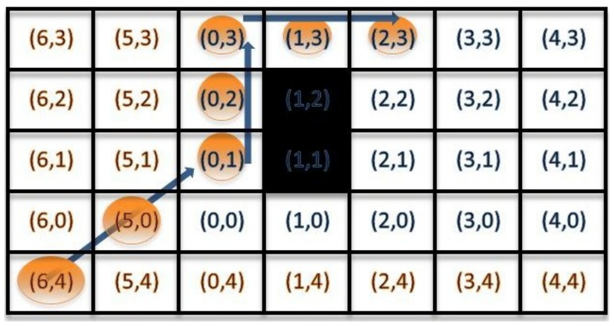


Figure 3.7: This is a specific case which depicts the robot's behaviour when reaching an obstacle.

Using this mapping model in our mapping algorithm, the robot is ideally subjected to navigate and learn the environment using the four directions defined below. While propagating in one of these directions, the robot combines two linear propagations together with their constraints. If one of these linear propagations fails for a particular case, then the diagonal propagation will also fail. Consequent measures are taken to handle such cases as previously shown in Fig. 3.7.
Additionally, if the robot tries in vain to follow one of the linear propagations, then the function responsible for the corresponding diagonal propagation will exit with a return value that describes the nature of the failure. The robot can then deduce the best instruction to execute next. This situation will generally occur when the robot is facing a wall, since it cannot move through a wall. Consequently, a rule for wall detection is used in each propagation to identify such situations. This rule is described in Section 3.5.4.
Below, we explain the four types of diagonal propagations used in our approach.

Forward-right propagation

While propagating in this direction, the robot combines both the forward and right propagations. The corresponding algorithm is given below:

1. Repeatedly move in the forward-right direction.
2. If an obstacle is encountered at the front, deviate by moving strictly in the right direction. Once the obstacle has been avoided (the front cell is unoccupied), return to step 1.
3. If an obstacle is encountered on the right, deviate by moving strictly in the forward direction. Once the obstacle has been avoided (the right cell is unoccupied), return to step 1.

4. In the two preceding steps, if the robot cannot go back to step 1 (the obstacle is a wall), then the forward-right propagation fails. If the failure occurs in step 2, the algorithm exits with a return value that indicates that the forward-right prorogation detected a wall in the front. This return value causes the backward-right propagation to be executed next. Similarly, if the failure occurs in step 3, the algorithm exits with a return value that indicates that a wall was detected on the right. In this case, the next propagation is the forward-left propagation.

5. If obstacles are encountered both to the front and the right of the robot (a corner of the room), the propagation terminates successfully. The robot moves back to either its vertical or horizontal initial position and continues mapping using an opposite propagation: the backward-left propagation in this case.

Forward-left propagation

Using this algorithm, the robot combines both the forward and left propagations by following a single direction. The corresponding algorithm is given below:

1. Repeatedly move in the forward-left direction.

2. If an obstacle is encountered at the front, deviate by moving strictly in the left direction. Once the obstacle has been avoided (the front cell is unoccupied), return to step 1.

3. If an obstacle is encountered on the left, deviate by moving strictly in the forward direction. Once the obstacle has been avoided (the left cell is unoccupied), return to step 1.

4. In the two preceding steps, if the robot cannot go back to step 1 (the obstacle is a wall), then the forward-left propagation fails. If the failure occurs in step 2, the algorithm exits with a return value that indicates that the forward-left propagation detected a wall in the front. This return value directs the choice of the next propagation to be executed to the backward-right propagation. Analogously, if the failure occurs in step 3, the algorithm exits with a return value that indicates that a wall was detected on the left. Consequently, the next propagation that the robot will undergo is the forward-right propagation.

5. If obstacles are encountered both to the front and the left of the robot (a corner of the room), the propagation terminates successfully. The robot moves back to either its

vertical or horizontal position and continues mapping using an opposite propagation: the backward-left propagation in this case.

Backward-right propagation

While propagating in this direction, the robot combines both the backward and right propagations by following a single path. The corresponding algorithm is shown below:

1. Repeatedly move in the backward-right direction.
2. If an obstacle is encountered at the front, deviate by moving strictly in the right direction. Once the obstacle has been passed (the current back cell is unoccupied), return to step 1.
3. If an obstacle is encountered on the right, deviate by moving strictly in the backward direction. Once the obstacle has been passed (the right cell is unoccupied), return to step 1.
4. In the two preceding steps, if the robot cannot go back to step 1 (the obstacle is a wall), then the backward-right propagation fails. If the failure occurs in step 2, then the algorithm exits with a return value that indicates that the backward-right propagation detected a wall at the back. The robot will use this value to determine the next propagation: the forward-right propagation. Analogously, if the failure occurs in step 3, the algorithm exits with a return value that indicates that a wall was detected on the right. In this case, the next propagation that the robot will undergo is the backward-left propagation.
5. If obstacles are encountered both to the back and the right of the robot (a corner of the room), the propagation terminates successfully. The robot moves back to either its vertical or horizontal position and continues mapping using an opposite propagation: the forward-left propagation in this case.

Backward-left propagation

While propagating in this direction, the robot combines both the backward and left propagations by following a single path. The corresponding algorithm is shown below:

1. Repeatedly move in the backward-left direction.

2. If an obstacle is encountered at the back, deviate by moving strictly in the left direction. Once the obstacle has been passed (the current back cell is unoccupied), return to step 1.

3. If an obstacle is encountered on the left, deviate by moving strictly in the backward direction. Once the obstacle has been avoided (the left cell is unoccupied), return to step 1.

4. In the two preceding steps, if the robot cannot go back to step 1 (the obstacle is a wall), the backward-left propagation fails. If the failure occurs in step 2, then the algorithm exits with a return value that indicates that the backward-left prorogation detected a wall in the front. Based on this value, the robot will start a forward-left propagation. Analogously, if the failure occurs in step 3, the algorithm exits with a return value that indicates that a wall was detected on the left. In this case, the next propagation to be executed is the backward-right propagation.

5. If obstacles are encountered both to the rear and the left of the robot (the bottom-left corner of the room), the propagation terminates successfully. The robot moves back to either the origin of the map (0,0) and continues mapping using an opposite propagation: the forward-right propagation in this case.

It should be noted that, the scenario constructed by the four propagations presented above is mutually recursive. Thus, we need to form a rule (or rules) that will force the robot to stop mapping and hence stop populating the grid.

3.5.3 Stop heuristic

The mapping algorithm is designed in such a way that usage of the same diagonal path more than once by the robot implies a large degree of redundancy. This issue is used as the basis for creating the stop heuristic which commands the robot to "stop after" mapping if the next step involves using an already traversed path. The purpose of such a heuristic in our mapping methodology is to avoid redundancy as much as possible in the map. Using a path once only will certainly contribute to this reduction. At this point in the algorithm, a scan is done on the grid to detect the percentage of unknown cells. If big areas of unknown cells are found, the robot will move to them and map each in a separate process. Using this heuristic, there may still be some unknown areas in the map which are not large enough to be detected by the mapping algorithm. The states of the corresponding unknown grids are updated when the robot traverses them during its navigation cycle.

3.5.4 Wall detection

To increase the time efficiency of our mapping algorithm, it is necessary to construct a set of rules that will enable the robot to detect walls or at least ”big” obstacles. The robot interprets a wall as an obstacle and similar to any other obstacle, the robot will try to avoid it. This is purely a waste of time since a wall is an obstacle that cannot be avoided.
Walls are the biggest obstacles in a room. So, the process of wall detection starts by defining a minimum wall size (the biggest obstacle size). While mapping, if the robot encounters an obstacle of size greater than this minimum wall size, it concludes to having hit a wall.
Additionally, walls can be detected using the edges of a room. If the robot hits a ”big” obstacle, and identifies an edge ahead of it, it deduces that it has hit a wall.

3.5.5 Grid interpretation

At the point in our mapping algorithm where the robot has completed the mapping and has a rough map, some additional information is needed to produce a complete map of a room. That is, we need to identify the following characteristics of a room:

- Walls: these are located in the map by studying the grid and looking for a number of continuous unknown cells. The robot cannot read beyond a wall, hence a specific arrangement of unknown cells will denote the presence of a wall. This rule is more effective than that used in wall detection (see Section 3.5.4). So the room dimensions determined by this rule are usually preferred if they are greater than those found during wall detection.

- Room size: Once the walls have been identified, the room size is easily deduced.

- Exits: These are the inverse of walls. The latter are represented by occupied cells, whereas exits are unoccupied cells. So, once the actual room size is known, the detection of unoccupied cells on the boundaries will be a significant sign of the presence of an exit.

Once the map has been interpreted, it can satisfy the definition of a room given above. This results in a metric map as defined in Section 2.2.2.

3.6 Inference mechanism

3.6.1 Overview

So far, we have demonstrated how to navigate through a room and efficiently build a map thereof. The next step in our approach is to show how a robot can use this map to execute basic tasks. We now describe a robust mechanism to infer knowledge from a map. This knowledge is an obstacle-free path from the current position to a desired position; which is the main purpose of constructing a map. With respect to our model of a map, this can be achieved using a search algorithm, that is able to search through the map and derive a path from an initial state to a goal state which is collision free. A path is a set of positions (or grid cells) that the robot traverses in order to reach the desired position in the map (grid). First, it should be noted that an occupancy grid can be interpreted as a graph in which each cell is a node in the graph. Each cell is connected to its immediate surrounding cells. With a given initial state (the root of the graph), we can build a directed graph comprising other relevant nodes in the grid. Then, the challenge in our inference mechanism is to find the best path to the goal state.

3.6.2 Search Algorithm: A* variant

Since the route from one point to another in a map is a directed route, we can safely assume no backtracking in our search algorithm. According to the structure of our grid, a convenient search algorithm has the characteristics defined as follows:

- Shortest path: With respect to the initial position of the robot, we can easily ascertain the shortest path to the desired position by looking at the constructed grid. In a mapping environment such as the one we have, the shortest path between two states is the path closest to a straight line.
- Cost of the path: This is defined as the number of nodes (cells) that need to be traversed before the robot achieves its goal.
- Branch factor: In this context, it is defined as the degrees of freedom of the robot; i.e., the number of legal moves that the robot can make to move from one cell to the next in the tree. By a ”legal move”, we mean a move to a node that has not been visited before and that is unoccupied.

Considering all of the above criteria, we have implemented our own variant of the A* search algorithm with the following characteristics.

First heuristic

This heuristic is based on the shortest path criterion. The search algorithm uses it at each node to generate and traverse child nodes in a specific order; from the closest node to the farthest node.
Let us assume that the current position of the robot is (u,v) and its goal position is (a,b) (See Fig. 3.8). The shortest path criterion requires that the set of five nodes {(u+1, v+1),(u,v+1), (u+1,v), (u-1,v+1), (u+1,v-1)} (shown in yellow in the figure) should be traversed first and in the given order. Subsequently, the remaining nodes are traversed in this order: {(u-1,v), (u,v-1), (u-1,v-1)}. If we look at the figure closely, we observe that any goal node whose coordinates satisfy the inequalities (a>=u+1) and (b>=v+1) will generate the same behaviour from this heuristic. Such goal nodes are grouped and we globally refer to them as Group 3 (illustrated in Fig. 3.8). In so doing, we can create eight groups of possible goal nodes where each group requires a specific behaviour from the heuristic. These groups are constructed with respect to the relative distance between the goal state and the current state. The heuristic rules are described below:

1. Group 1. If (a>=u+1) and (b>=v+1), expand the child nodes in the following order: {(u+1, v+1),(u,v+1), (u+1,v), (u-1,v+1), (u+1,v-1)}, {(u-1,v), (u,v-1), (u-1,v-1)}.
2. Group 2. If (a>=u+1) and (b>=v-1), expand the child nodes in the following order: {(u+1, v+1),(u,v+1), (u+1,v), (u-1,v+1), (u+1,v-1)}, {(u-1,v), (u,v-1), (u-1,v-1)}.
3. Group 3. If (a>=u+1) and (b==v), expand the child nodes in the following order: {(u+1, v+1),(u,v+1), (u+1,v), (u-1,v+1), (u+1,v-1)}, {(u-1,v), (u,v-1), (u-1,v-1)}.
4. Group 4. If (a==u) and (b>=v+1), expand the child nodes in the following order: {(u+1, v+1),(u,v+1), (u+1,v), (u-1,v+1), (u+1,v-1)}, {(u-1,v), (u,v-1), (u-1,v-1)}.
5. Group 5. If (a<=u-1) and (b>=v+1), expand the child nodes in the following order: {(u+1, v+1),(u,v+1), (u+1,v), (u-1,v+1), (u+1,v-1)}, {(u-1,v), (u,v-1), (u-1,v-1)}.
6. Group 6. If (a<=u-1) and (b==v), expand the child nodes in the following order: {(u+1, v+1),(u,v+1), (u+1,v), (u-1,v+1), (u+1,v-1)}, {(u-1,v), (u,v-1), (u-1,v-1)}.
7. Group 7. If (a<=u-1) and (b<=v-1), expand the child nodes in the following order: {(u+1, v+1),(u,v+1), (u+1,v), (u-1,v+1), (u+1,v-1)}, {(u-1,v), (u,v-1), (u-1,v-1)}.
8. Group 8. If (a==u) and (b<=v-1), expand the child nodes in the following order: {(u+1, v+1),(u,v+1), (u+1,v), (u-1,v+1), (u+1,v-1)}, {(u-1,v), (u,v-1), (u-1,v-1)}.

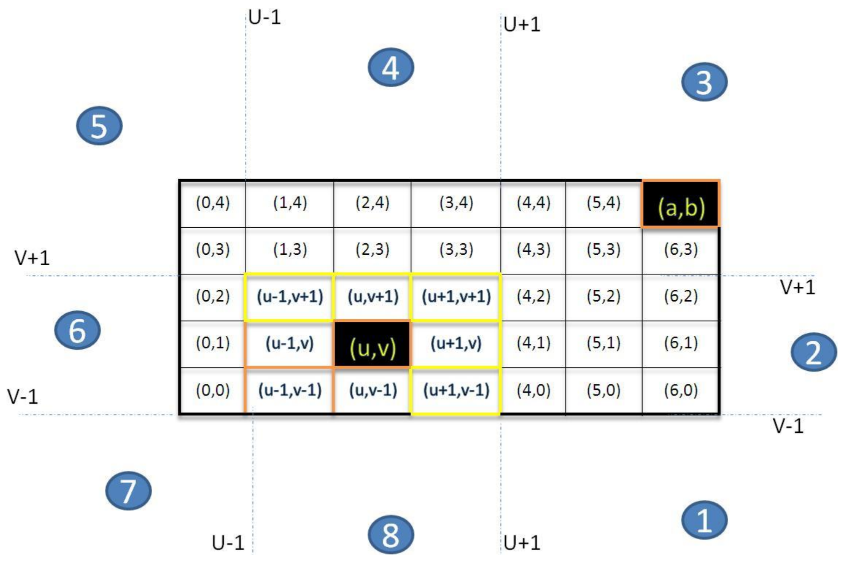


Figure 3.8: A path finding scenario.

Second heuristic

Again, let us assume that the current position of the robot is (u,v) and its goal position is (a,b) as shown in the figure above. With respect to the cost of path criterion, the length of a successful path should not be greater than the value (a-u)+(b-v). In fact, this is the cost of the best path if a diagonal path is not considered to be valid. We define a second heuristic using this criterion to approximate the cost of a successful candidate path. That is, the heuristic determines a limit to the depth of the search tree.
Additionally, the heuristic can be tolerant. During the search, if the depth limit is reached, but the current node is close to the goal, the search will give this path a chance to reach the goal state by extending the depth limit. We include this tolerance to reduce the time complexity of the path finding algorithm. That is, the search can tolerate the cost of three additional cells as opposed to the cost of expanding new paths.

Iterative deepening

Up to this point, the search algorithm has a limited depth and is not complete. Yet, a path finding should always find a solution path if there is one. So the depth limit described above is continuously increased for each iteration of the search, up to the stage where either a solution is found or there are no more legal moves.

Limited length search

With the aim of finding a path to a goal position faster, we limit the branch factor for each node expansion. As briefly described in Section 3.6.2, the set of five nodes shown in yellow in Fig. 3.8 are likely to reach the goal node faster than the remaining set. So, we first perform a search using the set of five nodes at each level, by limiting the branch factor to five (See Fig. 3.9(a)). If no success is achieved, the set of three nodes is used at each level and the branch factor is set to three (see Fig. 3.9(b)).

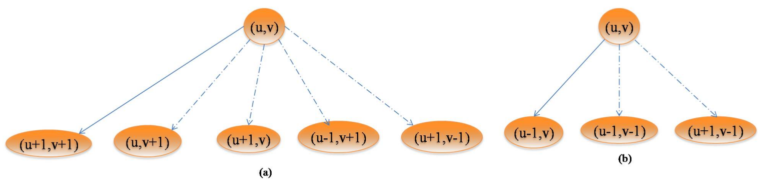


Figure 3.9: A limited search scenario at node (u,v).

3.7 Summary

As observed, we have created a good mapping methodology, analogous to the creation of a knowledge base. We ensure that the robot has a convenient means for storing a map by introducing and developing the concept of grid occupancy and distributed knowledge. Next, the core problem of environment learning is examined, and a learning methodology called the shortest path algorithm developed. Thus, the approach aims at implementing cognitive capabilities in the robot for an optimized self mapping. In the same perspective, we have also described a clever mechanism for inferring useful information from the constructed map. In the next chapter, we present a practical implementation of this approach together with related issues.

Chapter 4

Cognitive mapping and navigation: the implementation

4.1 Introduction

This chapter is devoted to the implementation of the approach described in the previous chapter. Using the C\C++ languages, we have developed a program that complies completely and strictly with the mapping approach presented in the previous chapter. To send basic commands to the robot, we have borrowed a class that achieves this and have implemented our own class which containing all the subroutines necessary for implementing the grid, and our learning and inference mechanisms. These subroutines which underpin the basis of our implementation and consequent development are clearly described in the next sections.

4.2 Implementation setup

The robot required to be equipped with this intelligence is the Mindstorms Lego NXT. It has basic abilities that will allow us to implement the different components of our approach. Some limitations that could be faced however are the processing power and memory space of the NXT brick. To counter these, the program is run on a personal computer and the required commands or data are communicated back and forth between the Lego NXT and computer via a bluetooth connection. This duplex communication once established, is used to obtain data from the ultrasonic sensors, and send commands to the NXT motors to drive its motion. This functionality is embedded in the "borrowed" Visual C++ class introduced in the previous section, and hence has already been implemented and is ready for use.
Furthermore, certain variables and constants need to be initiated as enforced by our model.

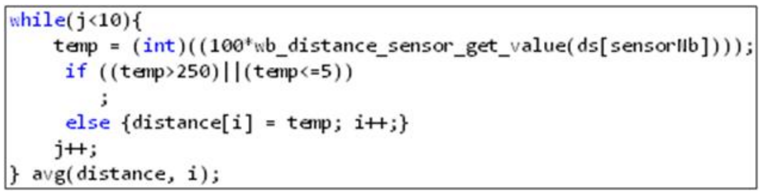
For example, to implement our solution to the SLAM problem, we use two variables (u and v) which depict the horizontal and vertical coordinates, respectively, in the grid and help the robot to localize itself in the maze. These values are updated according to the movement of the robot. Eventually, the initial position of the robot is set to the pair (u=0,v=0) which is the origin of the grid. Also, to be consistent, we define constant speeds (linear and orbital speeds) that the robot motors are constrained to use during the mapping. Additionally, a cell can only be in one of two known states (occupied and unoccupied); but to convey our model with an unknown environment, we have obviously added a third state ("unknown") to which every cell is initialised at the start of the mapping.

4.3 Error detection and correction

When implementing a mapping algorithm., odometry errors have to be detected and corrected. These errors are classified into two types, and the methods used to overcome them are described below.

4.3.1 Sensor errors

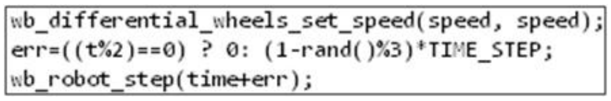
Ultrasonic sensor errors increase with distance. Such a sensor has a range within which the data error is negligible. After some experimentation, we defined a distance corresponding to an acceptable range of error and used this as the maximum value for any sensor reading. Any value above it is discarded. Similarly, sensors tend to behave inaccurately with small distances, and so any value below a defined minimum value is also discarded. These discarded values are likely to be wrong. Furthermore, sensor values are aggregated to obtain a precise or trusted value for use by the mapping algorithm. The algorithm is shown below:



```
while(j<10){
    temp = (int)((100*wb_distance_sensor_get_value(ds[sensorNb])));
     if ((temp>250)||(temp<=5))
        ;
     else {distance[i] = temp; i++;}
    j++;
} avg(distance, i);
```

4.3.2 Motor errors

As the robot moves, motor errors accumulate and localization becomes an exponential problem. At a certain point in the motion, this accumulated error becomes less tolerable and eventually causes the algorithm to fail. To maintain an acceptable error range, such that motion errors can be safely ignored, the robot speed has been altered appropriately. After carrying out a few experiments, we obtained a bias for the robot speed as shown below.



```
wb_differential_wheels_set_speed(speed, speed);
err=((t%2)==0) ? 0: (1-rand()%3)*TIME_STEP;
wb_robot_step(time+err);
```

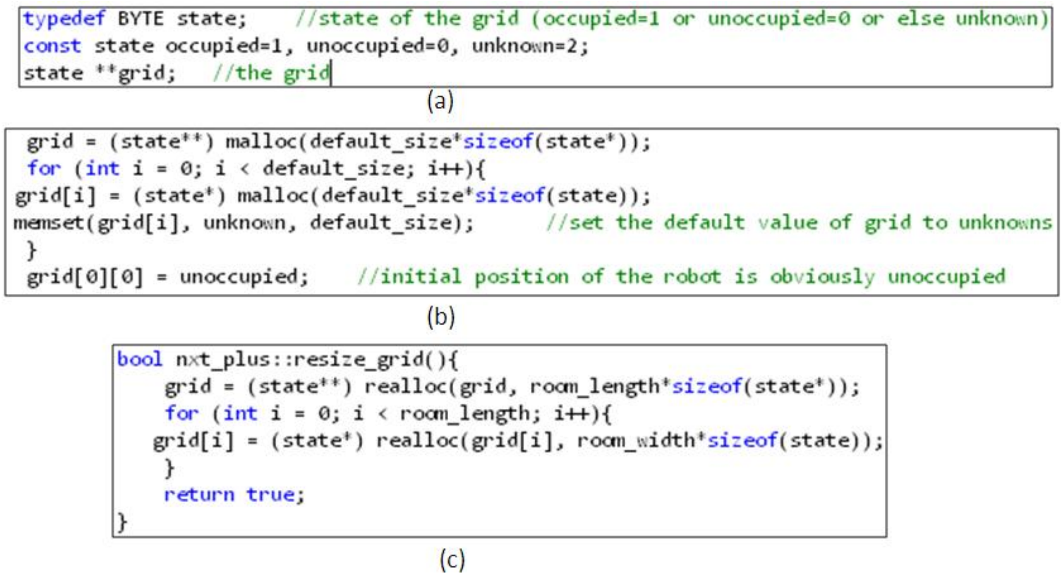
4.4 The occupancy grid: proposed data structure

From the grid representation presented in Section 3.3.1, it is obvious that a suitable data structure to store the robot knowledge is a 2D dynamic array. The properties of a grid can be implemented in such a data structure as described below.

- Grid definition: The grid is implemented using 2D pointers which point to a memory location that holds a value equivalent to the state of a particular cell (see Fig. 4.1(a)).

- Grid initialisation: First, the cell size is chosen such that the four sensors of the robot and the robot itself can fit into a single cell. Once we have defined the cell size (typically 20 centimetres for the Lego NXT), other variables are defined accordingly.
  e.g. u = 2 units which implies u = 2*cell_size cm. So, an increase in the interval [0−30]cm in reality is represented by an increase of one unit in the array.
  Since the grid has to be initialised to its maximum size, the array is initially allocated an amount of space equivalent to this size and the values of all its items set to zero (see Fig. 4.1(b)).

- Grid resizing: This is the most important property of the grid. The robot cognitive knowledge should not use more memory space than necessary. Memory space is critical in a robot agent and this issue should be seriously considered. Fortunately, our data structure definition allows us to solve this problem by resizing the array of space initially allocated for the map. Once the robot is done mapping (a boolean variable, *done* is set to true), and the room size found is used to resize the array appropriately (see Fig. 4.1(c)).



```
typedef BYTE state;    //state of the grid (occupied=1 or unoccupied=0 or else unknown)
const state occupied=1, unoccupied=0, unknown=2;
state **grid;   //the grid
```

(a)

```
 grid = (state**) malloc(default_size*sizeof(state*));
 for (int i = 0; i < default_size; i++){
grid[i] = (state*) malloc(default_size*sizeof(state));
memset(grid[i], unknown, default_size);      //set the default value of grid to unknowns
 }
 grid[0][0] = unoccupied;    //initial position of the robot is obviously unoccupied
```

(b)

```
bool nxt_plus::resize_grid(){
    grid = (state**) realloc(grid, room_length*sizeof(state*));
    for (int i = 0; i < room_length; i++){
   grid[i] = (state*) realloc(grid[i], room_width*sizeof(state));
    }
    return true;
}
```

(c)

Figure 4.1: (a) Grid definition, (b) Grid initialisation, (c) Grid resizing.

This data structure offers us both the flexibility and consistency required for a good occupancy grid.

4.5 Building a network of maps

4.5.1 Adding a new node to the network

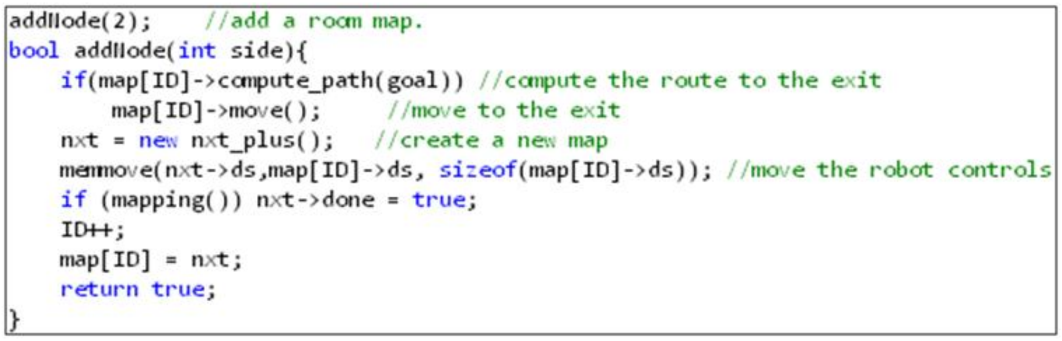
To add the network capability to our implementation, such that the robot is able to map several different rooms and interconnect them, two key functions are needed in our code. One is used to add a node (a room map) to the network (of maps) while the other is used for creating gateways that enable traffic between these nodes. First, we represent the network (topological map) by an array which holds the address (through pointers) of each map. The process of adding a new node or a map address into this array as shown in Fig. 4.2 is explained as follows:

1. Firstly, the initial map is saved into the array.

2. Secondly, the robot has to leave the current map through an exit.

3. Thirdly, the agent creates a new map and the mapping process is restarted.

4. Finally, when it is done mapping the new room, the map address (grid address) is also saved in the array.

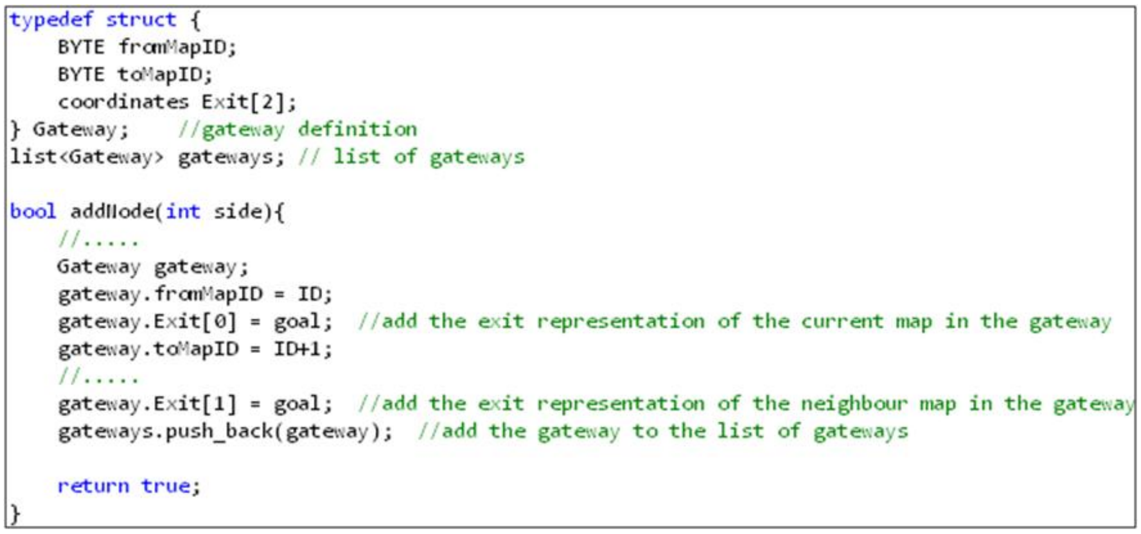


```
addNode(2);    //add a room map.
bool addNode(int side){
    if(map[ID]->compute_path(goal)) //compute the route to the exit
        map[ID]->move();      //move to the exit
    nxt = new nxt_plus();   //create a new map
    memmove(nxt->ds,map[ID]->ds, sizeof(map[ID]->ds)); //move the robot controls
    if (mapping()) nxt->done = true;
    ID++;
    map[ID] = nxt;
    return true;
}
```

Figure 4.2: Network implementation.

4.5.2 Gateway implementation

As defined in the model, a gateway merely contains information that tells the robot how to navigate from one map to a neighbouring map. It should also be noted that rooms are constructed such that the exit of one is the entrance to an adjacent room. So our gateway contains the identification of both the exit and the two rooms that are connected by it. However, the same exit is represented differently in each map. So, the gateway has been designed to hold two exit values. These gateways are created as new nodes are generated and they are added and saved in a list of gateways. Fig. 4.3 illustrates this implementation.



```
typedef struct {
    BYTE fromMapID;
    BYTE toMapID;
    coordinates Exit[2];
} Gateway;    //gateway definition
list<Gateway> gateways; // list of gateways

bool addNode(int side){
    //.....
    Gateway gateway;
    gateway.fromMapID = ID;
    gateway.Exit[0] = goal;  //add the exit representation of the current map in the gateway
    gateway.toMapID = ID+1;
    //.....
    gateway.Exit[1] = goal;  //add the exit representation of the neighbour map in the gateway
    gateways.push_back(gateway);  //add the gateway to the list of gateways

    return true;
}
```

Figure 4.3: Gateway implementation.

4.6 Robot Navigation implementation

4.6.1 Overview

In this section, we present the implementation of rules that are used by the mapping program to communicate with the robot sensors and motors. They are contained in a middleware class that hides both the complexity of the grid implementation and the robot interaction, by providing the actual mapping function with a container of simple functions. With respect to the physical capabilities of the robot, we created eight subroutines which are defined below.

1. **bool go_forward(int sofar, int speed):** this function is basically used by the robot to move in a forward direction. It takes two parameters (sofar and speed) which determine respectively the distance the robot should travel and the speed at which it should do so.

2. **int read_forward(int endhere):** This function reads the value returned by the front sensor up to the defined limit "endhere". This variable specifies the maximum distance that the function can return. It also updates the grid data structure accordingly and returns an integer value which is the distance that was read.

3. **bool go_backward(int sofar, int speed):** This function sends an instruction to the robot to move backward by a defined distance "sofar" and at a defined speed "speed".

4. **int read_backward(int endhere):** This function controls the back sensor of the robot and it works in the same way as the read_forward() function.

5. **bool go_right(int sofar, int speed, int angle):** This function commands the robot to turn to its right by a specified angle "angle", then move at the specified speed "speed" up to the specified distance "sofar". To reduce wheel slippage, the angle of rotation can only be 90 or 45 degrees.

6. **int read_right(int endhere):** The mapping program gets data from the right sensor through this function.

7. **bool go_left(int sofar, int speed, int angle):** This function commands the robot to turn to its left and operates in the same way as function 5.

8. **int read_left(int endhere):** The program controls the left sensor of the robot with this function.

These functions are similar to one another and only differ with respect to the direction in which they operate. For this reason, we only describe in detail the function prototypes corresponding to the first two functions. The other functions are implemented in the same way.

4.6.2 Navigation rules implementation

First, the mechanism for updating the grid while mapping depends on the initial structure of the grid which is determined by the initial coordinate system chosen by the robot. Since the robot uses the grid to navigate in the environment, the implementation of our navigation rules should consider the structure of the grid. So, we define two boolean variables *changbehavU and changbehavV*, to keep track of the grid structure. These two variables are updated according to the robot position, and are used by the navigation subroutines to interpret the grid correctly. The basic algorithms for the two navigation rules are given below.

Algorithm 1: Read forward

```
Input: distance
int i= 0, $k = cell\_size$;
while input(frontSensor)>$k$ do
    if changebehavV then
        i–;
    end
    else
        i++;
    end
    if cell[$u$][$v + i$] ==unknown then
        cell[$u$][$v + i$] ==unoccupied;
    end
    if i==distance then
        return $v + i$;
    end
    k+ = $cell\_size$;

end
cell[$u$][$v + i$] ==occupied;
return $v + i$;
```

Algorithm 2: Move forward

```
Input: distance, speed
int i= 0;
while $i<=distance$ do
    if $changebehavV$ then
        i–;
    end
    else
        i++;
    end
    if $cell[u][v+i]$ ==$occupied$ then
        if $changebehavV$ then
            v= $v+i-1$;
        end
        else
            v= $v+i+1$;
        end
        //update the position of the robot.
        return false;
    end
    wb_differential_wheels_set_speed($speed, speed$);
    wb_robot_step($time$); //move to the cell infront
end
if $changebehavV$ then
    v= $v+i-1$;
end
else
    v= $v+i+1$;
end
//update the position of the robot.
return true;
```

4.7 Mapping implementation

4.7.1 Traversal algorithm

At this stage, we implement our shortest path algorithm on the four possible propagation paths that the robot can traverse. These are: forward-left propagation, forward-right propagation, backward-right propagation, backward-left propagation. We implement four subroutines, each of which simulates one of the propagation types as described in Section 3.5.2. We also see that the consequence of each propagation rule determines the next propagation

rule. To emulate this criterion, each of the subroutines concerned has three distinct possible return values. The mapping function then uses one of these values to jump to the location of the code that should be executed next.
Also, at mapping time, the robot has four degrees of freedom, due to the limitations of a robot with four sensors. The robot relies on the sensors to move without hitting an obstacle. So each propagation route in the shortest path algorithm which relies on the robot following a diagonal path is implemented differently. The implementation of the forward-right propagation is examined. The other three propagation algorithms are implemented similarly.

Illustration

To move from position (a,b) to (a+1,b+1), the robot must move as follows:
-Firstly, from cell (a,b) to cell (a,b+1) if the latter is unoccupied. Otherwise, if it is occupied, the robot will try to move to cell (a+1,b) instead. If this latter cell is also occupied, the function exits with a satisfied return value, *done*.
-Secondly, from grid (a,b+1) or (a+1,b) to grid (a+1,b+1) if the latter is also unoccupied. Otherwise, if it is occupied, the robot will proceed using a linear propagation until an empty cell is found on a diagonal path, which will allow it to use a diagonal propagation. If it is not able to find such a cell after a linear propagation over a number of cells, it is deduced that a wall have been hit. So if the linear propagation was in the horizontal direction, the appropriate return value is *hitleft*, or *hitup* for the vertical direction. In doing this, the rule is implemented such that the robot always tries to use the shortest path for its navigation during its mapping cycle.
Additionally, two arrays are used by each function to keep track of the last read position so that it can resume reading sensor data from that point.

4.7.2 Restructuring the grid

To implement a function to transform our grid (displace its centre of origin), we have to take into account the two coordinates x and y. The transformation of the grid also depends on the initial coordinates system of the grid. Since there are four possible coordinate systems, the implementation also considers each of these cases. But the implementation of this overall function still depends on the horizontal and vertical coordinates, so, we have implemented two sub-functions, one to translate around the u-coordinate of the origin and the other to translate around its v-coordinate. This translation is basically a swapping of values in the grid in a specific manner. This example illustrates how the translation in the horizontal direction can be done.

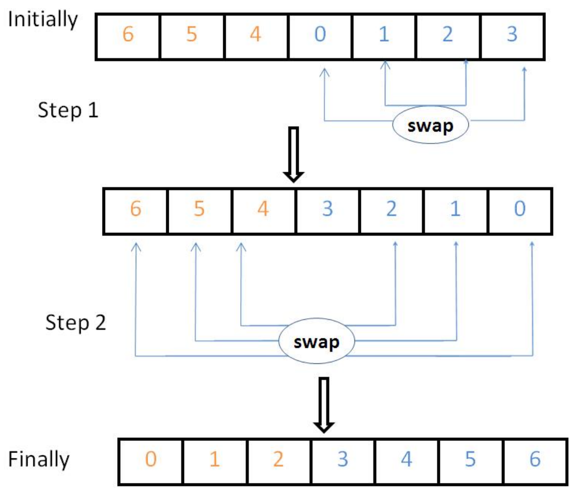
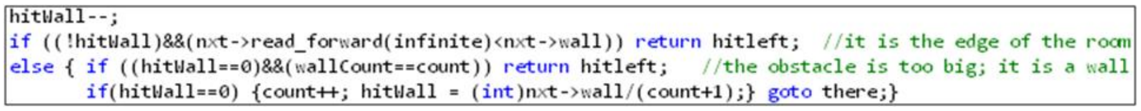


Figure 4.4: An example of translation of the u-coordinates of a grid.

4.7.3 Wall detection implementation

With respect to our definition of a room, the robot can encounter a maximum of four walls. Each of these walls are uniquely identified using the constant values: *hitright*(wall to the right of the robot), *hitleft*, *hitdown*, *hitup*. To detect big obstacles, the robot keeps a count of consecutive occupied cells. When it changes from a diagonal propagation to a linear propagation due to the presence of an obstacle, the count starts; the change may be due to the presence of a wall. When this count becomes greater than a predefined wall size or the robot is linearly moving toward a corner of the room, then the corresponding propagation fails with one of the return values we have just specified. A sample algorithm is shown below:

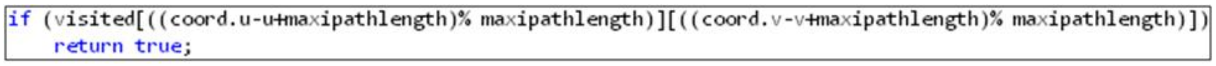


```
hitWall--;
if ((!hitWall)&&(nxt->read_forward(infinite)<nxt->wall)) return hitleft;  //it is the edge of the room
else { if ((hitWall==0)&&(wallCount==count)) return hitleft;   //the obstacle is too big; it is a wall
       if(hitWall==0) {count++; hitWall = (int)nxt->wall/(count+1);} goto there;}
```

4.8 Search algorithm implementation

As in any search tree algorithm, our implementation first needs to define the following parameters:

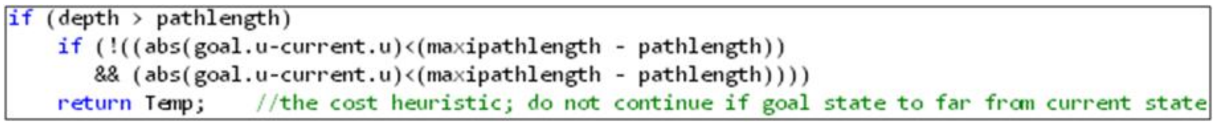
- visited nodes: A temporal image of the grid is created (a 2D dynamically allocated array) whereby each value in the array is set as either "true" (visited) or "false" (not visited). Each index value in the visited array is uniquely matched to a cell (u,v) in the grid array. In this way, we use the visited array to keep track of nodes that have already been examined.



```
if (visited[((coord.u-u+maxipathlength)% maxipathlength)][((coord.v-v+maxipathlength)% maxipathlength)])
    return true;
```

- depth: We define a variable depth to keep track of the depth of the tree. This is necessary, since our algorithm undergoes iterative deepening. The maximum value of the depth of the tree is determined by the ”cost” heuristic which gives a measure of the cost of a possible successful path.

-cost heuristic: This heuristic is implemented by computing the maximum length of a path to the goal without ”strictly” backtracking. But in our search algorithm, we assume the possibility of a small amount of backtracking. If the search feels that it is close to the goal, our search algorithm will allow a small possibility of backtracking. These two factors are bundled in the following heuristic:



```
if (depth > pathlength)
    if (!((abs(goal.u-current.u)<(maxipathlength - pathlength))
       && (abs(goal.u-current.u)<(maxipathlength - pathlength))))
    return Temp;    //the cost heuristic; do not continue if goal state to far from current state
```

-distance heuristic: As describe previously, this is used by the function to determine the order in which the nodes are examined. So we use a set of ”if” statements to return an array of ordered grids to be visited depending on the relative distance of the goal.

-The AstarSearch() function recursively calls itself until the goal is found, hence implementing the depth-first search in a nifty way. Using this method, a simple STL list implementation is suitable to save the nodes that are found to be valid through the search.

4.9 Summary

In this chapter, we have described implementation issues and how they have been solved. A working solution to the proposed mapping methodology has been produced. Our implementation produces two classes and a main program file which when combined and compiled, produces an executable file which is the output of our work. The following chapter is a critic of this work, investigating from all angles the properties of our mapping approach.

Chapter 5

Evaluation

5.1 Introduction

In this chapter, we describe an experiment with a mobile robot doing cognitive mapping. But it should be noted that in reality, we ran several similar experiments but with different parameters and the corresponding results are also presented here. Our mapping program is executed repeatedly with the robot starting its navigation from different position in different ”room” environments. In this way, we prove the performance of the mapping program by aggregating and interpreting obtained results. In the following sections, we present respectively our experimental set-up, a sample experiment, overall results, and discussions focusing on the obtained results.

5.2 Experimental Setup

To test the algorithms used in our program, we found it convenient to use robot simulation software. The simulator provides enormous flexibility in the experimental parameters. Different structures of rooms or even a house are simulated easily and the learning process can also be speeded-up. Practical issues such as robot running out of batteries are not of concern in this case. The simulator we used is the Cyberbotics robot simulator (Webots) and the simulated robot is a modification of the Khepera robot. This is interchangeable with the Lego NXT.
Our experiments were run in eighteen different phases, with each phase corresponding to a different problem description. That is, we created eight room environment simulations, with varying congestion levels, and robot starting positions. The simulated robot is then allowed to explore the environment until it is satisfied and decides to stop. It should be noted that

the objects in the environments are static and the robot is the only mobile objects there in. For a house environment, the robot normally requires human instruction to map a new room in a house. The mapping program is said to be successful in such an environment if the robot can move from one room to another and still localise itself.
As part of our experimental testing, we include a "display()" method is our program to display in the console window the current state of our grid and the current position of the robot in the grid. This is used specifically to ensure that the program is not failing with respect to the SLAM problem; i.e., the robot is at the correct position in the grid. In this display, we use the characters " ", "~~", "X", to represent respectively unoccupied, unknown and occupied cells. We also declare certain benchmarking variables in our program to provide the following data:

- Final number of unknowns in the grids.
- Final number of free space in the room (given in grid units).
- The room size (room length and room width).
- Sensors count: this is the number of times the program requested data from the sensors.
- Time count: this is a record in grid units of the time taken by the robot for every linear displacement and rotational displacement.

5.3 Simulation

This section briefly describes how we developed a simulated model of an environment and how we incorporated our mapping program to control the robot in such an environment. In the Webots simulation model, the user program is called the "controller" and the simulated environment (a room) is called the "world"[18]. These are run as two separate processes and instructions from the controller or data from the environment (the robot) are passed to each other as messages.

5.3.1 World descriptor file

We build an environment by describing it and its constituent objects in a file. This description is done using the VRML (Virtual Reality and Modelling Language) and is interpreted at runtime by the toolkit to create the world. Our simulated room is a 3D graphical representation which contains walls and obstacles. Since the robot is also an object in the environment, we were able to construct a robot, with four ultrasonic sensors and two motors. The sensor's range and noise were also specified here. Additionally, this file contains a pointer to its controller execution file. The source file for a sample world is available for download at `http://www.cs.ru.ac.za/research/g09f5474/downloads/`.

5.3.2 Controller program

The controller is basically our mapping program which interacts with the robot using some libraries available in Webots. Using these libraries, the program obtains a pointer to the robot's sensors and a pointer to the robot's wheels. In this case, our program that was previously built on top of a bluetooth connection with the NXT robot is slightly modified and is now built on top of the Webots robot libraries. The source code of this controller is available for download at `http://www.cs.ru.ac.za/research/g09f5474/downloads/`.

5.4 An experiment

In this section, we give a description of the experimental process, through which we were able to gather quantitative information about our mapping algorithm. The world file is modified for each experiments, to accommodate either the new starting position of the robot or the new structure of the room. The obstacles used in the scope of these experiments are simple blocks which are defined in VRML as bounding objects. These blocks are easy to create, and represent real objects like chairs or tables that are encountered in real indoor environments. We also used the graphical editor ("scene editor" as shown in the toolkit interface) for rapid editing of an environment. An example of an experimental environment, drawn with predefined size and a predefined number of obstacles, is shown Figs. 5.1, and 5.2). The two figures show respectively, the state of the knowledge (grid) the robot has of its environment at the beginning and at the end of the mapping. As can be seen by comparing the resulting cognitive map (in the console window) and the real world, the obstacles' positions, and the room size are correctly represented in the knowledge base. It also illustrates the areas that were not mapped by the robot ("˜˜").
By varying some properties of the mapping problem such as the initial position of the robot in the unknown environment, we produce a number of various experiments. Subsequent results are then used to evaluate and discuss our mapping approach.

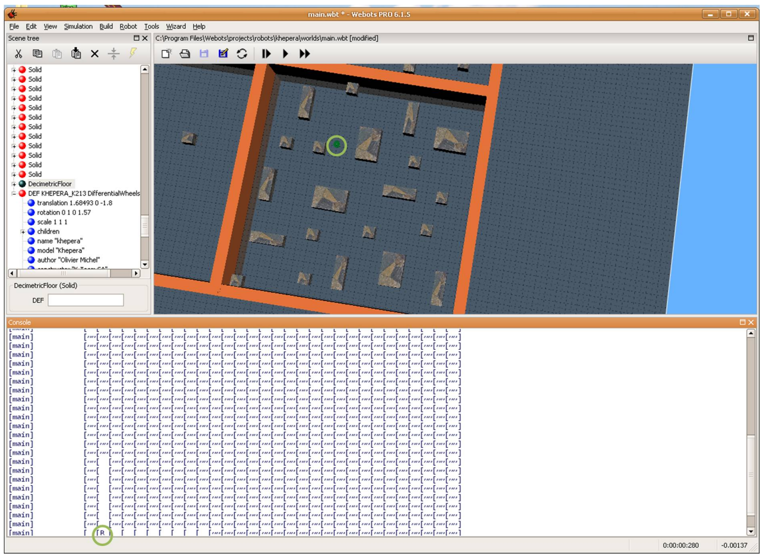


Figure 5.1: A view of the grid at the beginning of the experiment.

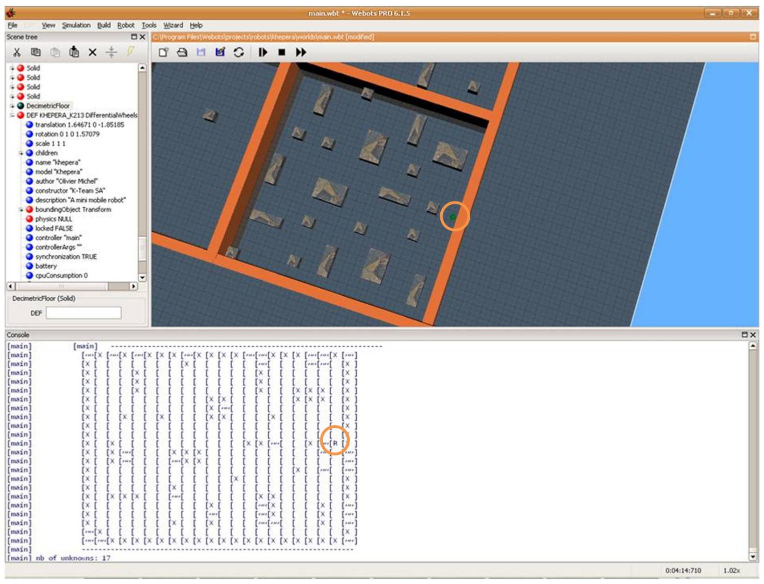


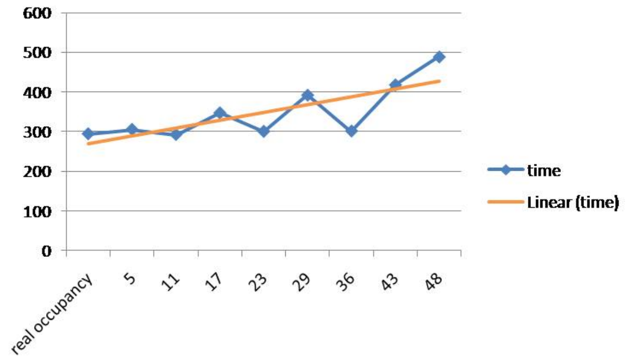
Figure 5.2: A view of the grid at the termination of the experiment.

5.5 Mapping results

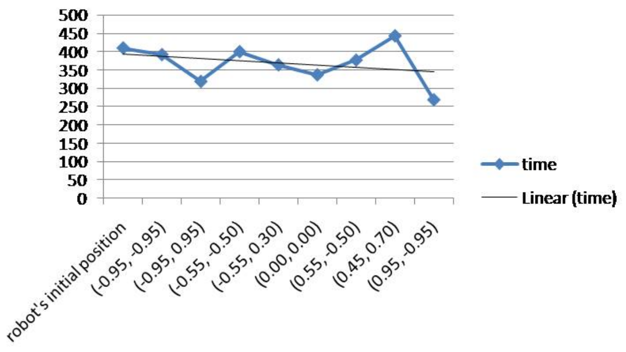
This section presents and discusses the results obtained in the different experiments we carried. With a total of eighteen experiments, we were able to verify the relevant properties of our mapping algorithm. Half of these were done by varying the number of obstacles in the room, while the others were done by varying the starting position of the robot in the room. The full set of data results obtained during these experiments is shown in Appendix A.

5.5.1 Time efficiency of the mapping algorithm

Time efficiency was an important factor considered while developing our mapping approach. So, during the experiments, we recorded the various times that the robot took to learn a room. These times are given in cell units. A cell unit in this case, is the time taken by the robot to move from one cell to the next (0.5 s in our experiment). Figs. 5.3(a) and 5.3(b) show respectively, the records obtained for the two different experiments carried out.



(a) A plot of the learning time versus the occupancy of a room.



(b) A plot of the learning time versus the starting position of the robot.

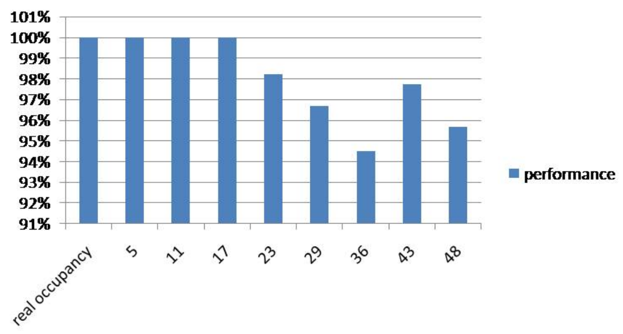
Figure 5.3: An evaluation of the learning time.

Discussion

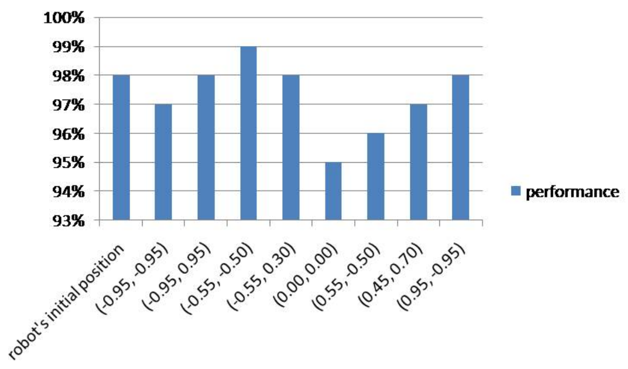
The time taken by the robot for each mapping seems to increase at a slow rate and linearly. As the number of obstacles doubles, the time increases by a factor of 1.5 at most. This is quite impressive as we normally expect the robot to spend more time to map a relatively congested room. This proves that, within a wide range, the mapping algorithm can maintain its time efficiency, irrespective of the number of objects present in the environment. Effectively, our mapping approach is a cognition based approach, which is specifically designed to minimise time consumption during its mapping cycle. By following a diagonal path and using heuristics such as wall detection, the time cost of the algorithm is dramatically reduced.

5.5.2 Map completeness

A map is usable and reliable if it shows all the obstacles in their correct positions. We define a complete map to be one that the robot can use to practice collision-free navigation. In order to prove that our mapping approach aims at producing such maps, we verify the position and the number of obstacles detected by the robot in the environment. In doing this for a number of experiments, we obtained the plots shown Fig. 5.5.2.



(a) A plot of the grid occupancy versus the real world occupancy.



(b) A plot of the grid occupancy versus the starting position of the robot.

Figure 5.4: An evaluation of the map completeness.

Discussion

The first remark one can make about the results shown above is that all maps obtained for different experiments are at least 94% complete. Though, the positions of found objects were correct, the robot did not detect all of them. This remark is particularly relevant because the aim of any mapping process is to learn accurately the position of the obstacles in an environment. This is the only way to ensure that the robot can undergo a collision-free navigation in a previously unknown environment. Even though the map produced by our mapping algorithm is not complete in terms of number of objects, it suffices to prove that it is reliable and therefore complete for a safe navigation.

Proof. First, all the cells in the grid are set to *unknown* before the mapping starts. So, if some objects are not detected after the mapping, their corresponding cells will still be in an *unknown* state. Thus, the robot will not use these cells during its navigation; i.e., although obstacles are not explicitly detected, they are still avoided during navigation. In this way, the navigation remains safe. However, a disadvantage may appear when these *unknown* cells are in reality unoccupied. This causes the robot navigation space to be smaller than it is in reality. But, as long as our mapping algorithm maintains a very small number of unknown areas in the map, the map is perfectly usable. The results of our experiments show that unknown areas are at most 6% (100%-94%) of a map. Adding the fact that the robot has the ability to find the occupancy of these areas during its navigation, we consider the obtained maps usable and the mapping algorithm satisfactory.

Additionally, we observed in the experiments that most *unknown* cells were actually part of previously found (See Fig. 5.5). This situation occurs particularly in rooms with larger obstacles. Failure to find the occupancy of such cells does not affect in anyway the navigation of the robot. So, if we consider cases like this, we can probably increase the completeness of the mapping algorithm to 97%, for congested environments. The latter are more difficult to learn during navigation than less congested environments. So achieving such performance after the mapping cycle is quite desirable.

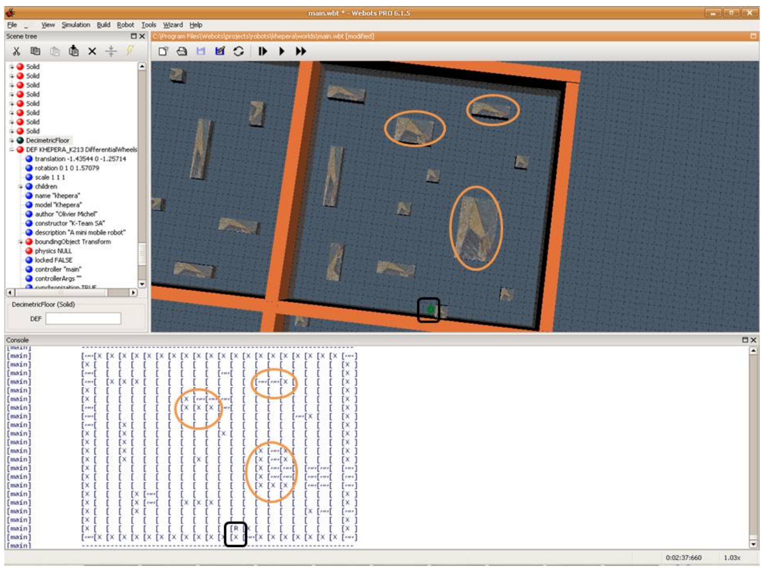
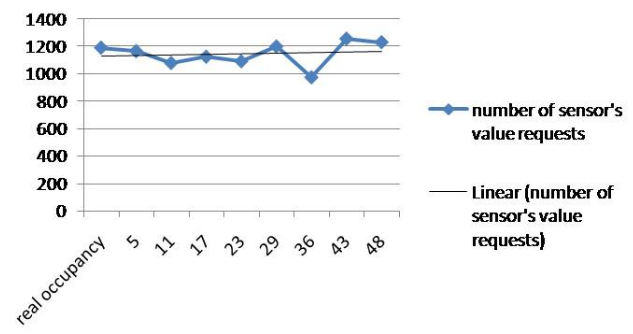

main.wbt * - Webots PRO 6.1.5
File Edit View Simulation Build Robot Tools Wizard Help
Scene tree
Console
DecimetricFloor (Solid)
DEF

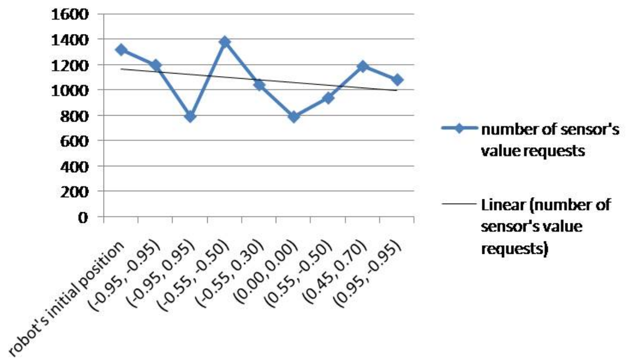
Figure 5.5: A case of unknown cells representing known objects.

5.5.3 Cognitive property of our mapping approach

This section is dedicated to the evaluation of the cognitive property of our mapping algorithm. We show how the robot relies on its cognitive capabilities rather than its physical capabilities (sensors and motors). The graphs in Figs. 5.6(a) and 5.6(b) show the usage of robot sensors versus the room occupancy or starting position of the robot, respectively.



(a) A plot of sensor usage versus the occupancy of a room.



(b) A plot of sensor usage versus the starting position of the robot.

Figure 5.6: An evaluation of the cognitive property of the mapping approach.

Discussion

Despite the various states of the environment shown in the graphs above, the demand for sensor data does not vary much. Irrespective of the initial conditions of the mapping problem, the robot solves it using almost the same amount of sensor data. We even observe in Fig. 5.6(b) that, as the starting position of the robot moves towards the middle of the room, the demand for sensor data decreases. Effectively, in this position, the robot is more exposed to the objects in the room than in any other position in the room. We can place the robot anywhere in a room, and can confidently expect it to map that room. That is, the user (a human) does not have to modify the room to accommodate the behaviour of the robot. On the contrary, the latter accommodates itself to the environment.

This demonstrates that there is a cognitive process that enables the robot to use its sensors more efficiently, by using them less frequently. That is, the robot does not move randomly in the environment, and does not determine the occupancy of each cell by always using the sensors. In fact, there is an intelligent structural learning methodology that the robot applies. This is why, we based our mapping approach on a cognitive approach to enable the robot to "humanly learn". It does an incremental mapping with the current state of its knowledge, and only requests sensor values when necessary.

As a matter of fact, the robot is expected to map an unknown environment and a good mapping approach is obviously one that is not greatly affected by the nature of the environment.

5.6 Path finding evaluation

The path finding algorithm used in our approach is based on an IDA* search, which is an iterative search that ensures that all the cells have been visited before admitting that it has failed to find a solution path. Moreover, as enforced by the heuristics, it is an optimal algorithm with respect to cost of solution. These are the minimum requirements for a successful path finding approach. In this section, we, therefore, evaluate the space and time complexity of the algorithm, to prove its efficiency.

5.6.1 Theoretical evaluation

- Space complexity: An IDA* search is an optimal search with respect to space complexity. This latter is of order O(n) where n is the depth of the tree.

- Time complexity: In [8], the time complexity of an IDA* search for one iteration is proven to be O(b$^{d_i-k}$) where k is the effect of the heuristics on the depth and d_i is the depth of that iteration. If we add to this formula the fact that our algorithm also undergoes a limited length search, it becomes O((b-l)$^{d_i-k}$) where l is the effect on the branch factor. Recall from Section 3.6, that l can only take the values 3 or 5 in the search. So we define the time complexity to be O((b-3)$^{d_i-k}$) or O((b-3)$^{d_i-k}$) + O((b-5)$^{d_i-k}$) for one iteration. But for each new iteration, we increase the depth limit by a constant value c: $d_i = d_0 + i * c$, where d_0 is the depth of the first iteration and i the number of the iterations. Putting all this together, the time complexity of our search algorithm is given by: $\sum_i O((b-3)^{d_0+i*c-k}) + O((b-5)^{d_0+i*c-k})$.

5.6.2 Experimental evaluation

In this section, we observe the practical behaviour of our path finding algorithm through various experimental results. For several experiments, we take a record of the number of nodes visited during a search and plot it against the depth of the solution path. With this methodology, the plot in Fig. 5.7 is obtained.

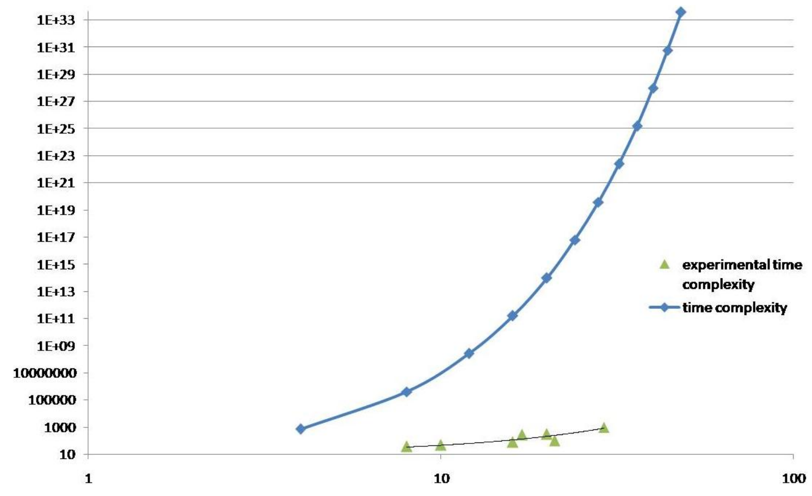


Figure 5.7: An evaluation of the path finding algorithm.

Discussion

As shown in the figure above, the search manages to achieve a shorter time complexity and does not fall into the scary exponential behaviour shown in the second curve (in blue). The use of heuristics enables the algorithm to reach a solution faster if there is one. Slow behaviour of the search typically arises when there is no solution path. Nevertheless, the path finding is based on a known map. So, the robot only requests a path to a position if it is unoccupied; i.e., for every path finding made by the robot, there will always be a solution to the desired position. This is why, it seems in from our experiments that the search scarcely goes into a second iteration. So, practically, the path finding algorithm has time complexity: $O((b-3)^{d_0-k}) + O((b-5)^{d_0-k})$.

5.7 Summary

Through a practical assessment of our approach, we observe that the main concerns in the mapping problem have been resolved. Nineteen different simulations where executed and the robot always succeeded in mapping the majority of the space in the room and in localising itself. We demonstrate that the robot emulates a cognitive behaviour as expected by our approach. Placed in an unknown environment, the robot was always able to detect a significant number of the objects in the room and successfully identify the room size. Furthermore, this was done without loss of efficiency.

Our program has also been deployed in the real world with the Lego NXT. Though odometry errors are difficult to remove, the behaviour of the robot corresponds closely with that predicted by the simulations.

Chapter 6

Conclusion

6.1 Summary

The aim of this project was to propose a working solution to the mapping problem. Mapping an unknown environment is a ubiquitous problem, which involves other issues in Spatial Robotics like localisation and safe navigation. To handle such problems, the robot should possess adaptive and self learning capabilities. As a result, there is need for a methodology, that the robot can use to map environments, irrespective of their physical structure. Therefore, our proposed solution uses cognitive science. In doing so, we developed a robust knowledge base system which enables the robot to build, store, and use a map of a room. The system uses appropriate learning and inference mechanisms which provide the robot with the cognitive capabilities required for a successful mapping. We also proved that this system is capable of solving the mapping problem.

6.2 Conclusions

We can confidently say that the cognitive mapping approach developed in this project is a solution to the mapping problem. As presented in Section 1.1, the principal goals of the mapping approach were successfully achieved. Both correspondence and SLAM were solved. Using the map, the robot can always localise itself or any objects, irrespective of its position in the room. It is possible for a robot to learn the positions of objects (e.g. a chair) in its surroundings, and can use the knowledge to execute human instructions ("go near the chair"). We have proved that, by using its cognitive capabilities, the robot can accurately and efficiently map different types of environments, without any configuration from its user. Additionally, the mapping model is scalable and allows a single robot to possess knowledge about more than one environment. Although we used a fixed number of distance sensors in the design (four in this case), the mapping model can still be implemented with a greater number of sensors.

6.3 Future Work

A practical achievement of our research is the creation of an autonomous sweeper robot. We have built a cognitive mapping approach that enables a robot to acquire spatial information about its surroundings. Additionally, our approach can serve as a basis for the development of more sophisticated robotic agents. This can be achieved by extending our work as described below.

- The mapping approach can be improved to represent the real structure of the environment. For example, by changing the structure of the cells from a square to a polygon (more than four edges), the robot will have a better appreciation of the environment. Moreover, such structure will increase the degrees of freedom of the robot and will enable it to move more humanly.
- For each cell in the grid, it is possible to attach any relevant qualitative information. For example, a picture can be linked to a cell and be used at a later stage to visually recognize part of an obstacle. With the evolution of visual recognition technologies, the robot can distinctly identify objects (for example a bottle of beer), and can subsequently execute specific tasks like ”remove the cake from the microwave”.
- Our mapping approach was designed for an indoor environment. But equipped with a GPS, it is possible to design a robot able to navigate both in an indoor and outdoor environment. In an indoor environment, the robot will recall its map of the house. In an outdoor environment, the robot will use the GPS device to safely navigate to its destination.

6.4 Contributions

Robotic approaches are limited when it comes to dealing with intelligence formation and its evolution. With the growth in AI and Robotics recently, the need for self learning and reasoning has become even more imperative. Also, the robot as a physical agent, interacts with its environment and cannot be efficient if it does not have a proper understanding of that environment. We addressed this issue by providing our own methodology for robot spatial mapping. In fact, this work focused on improving autonomous navigation, self localisation, and spatial learning in robotics.

Bibliography

[1] Dissanayake, M. W. M. G., Newman, P., Clark, S., Durrant-whyte, H. F., and Csorba, M. A solution to the simultaneous localization and map building (slam) problem. *IEEE Transactions on Robotics and Automation 17* (2001), 229–241.

[2] Dudek, G., and Jenkin, M. *Computational principles of mobile robotics.* Cambridge University Press, New York, USA, 2000.

[3] Elfes, A. Using occupancy grids for mobile robot perception and navigation. *Computer 22*, 6 (1989), 46–57.

[4] Fox, D., Burgard, W., Thrun, S., and Cremers, A. B. Position estimation for mobile robots in dynamic environments. In *AAAI '98/IAAI '98: Proceedings of the fifteenth national/tenth conference on Artificial intelligence/Innovative applications of artificial intelligence* (Menlo Park, CA, USA, 1998), American Association for Artificial Intelligence, pp. 983–988.

[5] Jefferies, M. E., and Yeap, W.-K. *Robotics and Cognitive Approaches to Spatial Mapping.* Springer, 2008.

[6] Jones, H. A* algorithm tutorial, 2009.

[7] Klippel, A., and Kulik, L. Using grids in maps. In *Theory and Application of Diagrams. First International Conference* (2000).

[8] Korf, R. E., Reid, M., and Edelkamp, S. Time complexity of iterative-deepening-a*. *Artificial Intelligence 129*, 1-2 (2001), 199 – 218.

[9] Kuipers, B., and tai Byun, Y. A robot exploration and mapping strategy based on a semantic hierarchy of spatial representations. *Journal of Robotics and Autonomous Systems 8* (1991), 47–63.

[10] Margaret E. Jefferies, W. K. Y. *The Utility of Global Representations in a Cognitive Map*. Springer, London, UK, 2001.

[11] Mindstorms, L. http://mindstorms.lego.com/overview/the_nxt.aspx.

[12] Mitsou, N. C., and Tzafestas, C. S. Temporal occupancy grid for mobile robot dynamic environment mapping. In *Control & Automation, 2007. MED '07. Mediterranean Conference* (2007), pp. 1–8.

[13] Pagello, E., Groen, F., Arai, T., Dillmann, R., and Stentz, A. *Intelligent autonomous systems 6*. IOS Press, 2000.

[14] Thrun, S. Probabilistic algorithms in robotics. *AI Magazine 21* (2000), 93–109.

[15] Thrun, S. Robotic mapping: A survey. In *Exploring Artificial Intelligence in the New Millenium* (2002), Morgan Kaufmann Publishers Inc., pp. 1–35.

[16] Tomatis, N., Nourbakhsh, I., and Siegwart, R. Simultaneous localization and map building: A global topological model with local metric maps. In *Intelligent Robots and Systems, IEEE/RSJ International Conference* (Maui, HI, USA, 2001), vol. 1, pp. 421–426.

[17] Verri, A., and Uras, C. Metric-topological approach to shape representation and recognition. *Image and vision computing 14* (1996), 189–207.

[18] Webots. http://www.cyberbotics.com. Commercial Mobile Robot Simulation Software.

[19] Wiskott, L., and Sejnowski, T. Constrained optimization for neural map formation: a unifying framework for weight growth and normalization, 1998.

[20] Wolter, D. *Spatial Representation and Reasoning for Robot Mapping: A Shape-Based Approach*. Springer, 2008.

[21] Wolter, D., Latecki, L. J., Lakamper, R., and Sun, X. Shape-based robot mapping. In *Advances in Artificial Intelligence* (2004), vol. 3238, Springer, pp. 439–452.

[22] Yap, P. Grid-based path-finding. 44–55.

[23] Yeap, W. K., and Jefferies, M. E. Computing a representation of the local environment. *Artificial Intelligence 107*, 2 (1999), 265–301.

[24] Yeap, W. K., Jefferies, M. E., and Naylor, P. An mfis for computing a raw cognitive map. In *IJCAI* (1991), pp. 373–380.

Appendix A

Experimental results

Time complexity of the path finding algorithm

depth	**constant c**	**heuristic effect**	**branch factor**	**number of nodes**
21	4	23	5	95
10	4	10	5	45
8	4	36	5	35
16	4	28	5	75
17	4	27	5	260
29	4	15	5	950
20	4	24	5	290

Varying the number of obstacles in the room

ac free space	**rb free space**	**ac occupied**	**rb occupied**	**Nb of unknows**	**mapping time**	**grid size**	**sensor count**	**sensor range**	**rb room size**	**ac room size**	**initial position**			**performance**
395	381	89	89	14	294	10x10 cm	1189	5cm to 1m	484	484	-0.95	0	0.95	100%
389	374	95	95	15	305	10x10 cm	1164	5cm to 1m	484	484	-0.95	0	0.95	100%
383	372	101	101	11	292	10x10 cm	1078	5cm to 1m	484	484	-0.95	0	0.95	100%
377	362	107	107	15	347	10x10 cm	1123	5cm to 1m	484	484	-0.95	0	0.95	100%
371	359	113	111	14	300	10x10 cm	1090	5cm to 1m	484	484	-0.95	0	0.95	98%
364	351	120	116	17	392	10x10 cm	1198	5cm to 1m	484	484	-0.95	0	0.95	97%
357	344	127	120	20	301	10x10 cm	974	5cm to 1m	484	484	-0.95	0	0.95	94%
352	340	132	129	15	418	10x10 cm	1254	5cm to 1m	484	484	-0.95	0	0.95	98%
345	334	139	133	17	489	10x10 cm	1229	5cm to 1m	484	484	-0.95	0	0.95	96%

Varying the starting position of the robot in the room

ac free space	**rb free space**	**ac occupied**	**rb occupied**	**Nb of unknows**	**mapping time**	**grid size**	**sensor count**	**sensor range**	**rb room size**	**ac room size**	**initial position**			**performance**
364	351	120	116	17	432	10x10 cm	1198	5cm to 1m	484	484	-0.95	0	0.95	97%
364	344	120	117	23	290	10x10 cm	1083	5cm to 1m	484	484	0.95	0	0.95	98%
364	361	120	116	7	398	10x10 cm	1188	5cm to 1m	484	484	0.95	0	-0.95	97%
364	352	120	118	14	400	10x10 cm	1319	5cm to 1m	484	484	-0.95	0	-0.95	98%
364	352	120	117	15	376	10x10 cm	1043	5cm to 1m	484	484	0.00	0	0.00	98%
364	356	120	119	9	388	10x10 cm	1380	5cm to 1m	484	484	-0.55	0	0.30	99%
364	349	120	117	18	314	10x10 cm	795	5cm to 1m	484	484	-0.55	0	-0.50	98%
364	357	120	114	13	314	10x10 cm	792	5cm to 1m	484	484	0.55	0	-0.50	95%
364	358	120	115	11	354	10x10 cm	940	5cm to 1m	484	484	0.45	0	0.70	96%