Controlling a Robot using Association Rules with a Temporal Component

Daniel Strand

23rd June 2005
Master's Thesis in Computing Science, 20 credits
Supervisor at CS-UmU: Thomas Hellström
Examiner: Per Lindström

Umeå University
Department of Computing Science
SE-901 87 UMEÅ
SWEDEN

Abstract

Data mining techniques can be used to find non-obvious local patterns (association rules) in data. If the data is generated by controlling a robot manually and logging the information from the robots sensors and actuators, these rules can be used to create a robotic controller which replicates the robots original behavior, thus reducing or removing the need to manually write controlling software.

This thesis finds that the method of building robotic controllers using association rules works well for creating relatively simple behaviors, and by giving the robots sensors data preprocessing abilities more complex behaviors can be created. Experimental results suggest that different behaviors require different methods of evaluating the usefulness of individual rules. It is found that by applying various filtering criteria, it is possible to remove redundant rules from the rule base which can greatly reduce the number of rules needed to replicate a given behavior.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Robots are used in many fields today. They are used in various industries for dangerous, difficult or repetitive tasks: building cars, harvesting crops, examining pipes, etc.. In medicare they perform remote controlled surgery, and assist disabled patients. They explore dangerous environments, from volcanos and disaster areas to space and other planets. Lately they are increasingly also being used as domestic helpers and even pets.

Designing and building robots is difficult. A robot must be able to handle many different situations depending on its intended use and this places great importance in its controlling software. Several techniques and methods exists for designing controllers and no single method is the best for all types of robots.

Robots having reactive behaviors have been shown to work well in many situations (see section 2.1.2). A large advantage of reactive behaviors vs. other more complex behaviors using planning in various forms, is that reactive behaviors generally need less powerful hardware to run on. Another advantage is that they allow the robot to react faster, which can be very important. A purely reactive robot needs no internal representation of the world, or memory of past events and can therefore be relatively simple in structure and thus cheaper and easier to produce while more complex systems of control are harder to design and more prone to bugs and failure. Because of this it is desirable to keep a robotic controller reactive if possible.

A common problem when designing reactive controllers is the difficulty of creating complex behaviors. A simple behavior is one that depends solely on external stimuli of the robots sensors and might be *run away from light*. This type of behavior requires no planning, memory or abstract representations to function. A complex behavior on the other hand, typically requires the robot to remember things in order to execute an ordered sequence of simple behaviors. An example could be the "cockroach" behavior where a robot exposed to light turns around and runs away until it reaches a wall where it changes to a wall-following behavior until it finally reaches a hole in the wall where it hides. This behavior is composed of several simple behaviors and is difficult to implement in a reactive controller. It is virtually impossible to create even more complex behaviors in a controller without moving away from the reactive paradigm and give the robot non-reactive traits like memories of past events or abstract representations of the world-state.

One way to create reactive behavior controllers that has been looked at is to use *Association Rules* that are extracted from large sets of data using *data mining* techniques [1].

Data mining is a term used to describe several different techniques and algorithms used to find and extract useful information from data sets (see section 2.2). Data mining for *association rules* is typically done to find more or less frequently occurring patterns in large datasets. These patterns are expressed as *if-then* rules with probabilities, of the type $R : A \rightarrow B$ with $(P = p)$ meaning rule R says that if A is true then so is B with a probability of P. This simple structure makes the rules well suited for use in programming. Today association rules are used extensively in the retail industry to find customer patterns based on data from purchases and similar information.

The idea looked at in this thesis is to use association rules as a base for a reactive robotic controller. This has been done previously by Hellström [1] [2], but here more complex behaviors are the goal as well as looking at ways of removing redundant rules from the rule base, and resolving situations where multiple rules can be applied. Also a test is made to try a method designed to minimize the need for customizing of the robot sensors by flattening the unprocessed sensor data.

The basic technique used is to make a training run with the robot controlled either remotely by a human supervisor, or by a custom designed software controller. During this run all data from the robots sensors together with the corresponding actions taken are logged. This data log is then searched for association rules using data mining software. For this thesis the commercially available *Magnum Opus* program has been used. The rules it finds are used to control the robot, hopefully giving it the same behavior as the original controller.

Chapter 2

Background

To better understand the unrelated concepts of reactive robots and data mining, this chapter provides an introduction to the subjects.

2.1 Robotic behaviors

Robotics is the field of science studying robots. Many definitions of *robot* exists, but in general a robot is defined as an *autonomous machine* (a machine that can operate without human supervision). This thesis looks at *intelligent* robots, which is a concept that is even harder to define so no attempt is made.

Making a robot act in a seemingly intelligent way is difficult. Over the years there has been three main schools of thought: The Hierarchical paradigm, the Reactive paradigm and the Hybrid Deliberative/Reactive paradigm [4].

2.1.1 The Hierarchical Paradigm

This is the oldest method of attacking the problem of making intelligent robots. It can be thought of as having three main steps: *sense*, *plan*, and *act* [4]. Each step is discrete and they are performed sequentially in a loop (figure 2.1) In

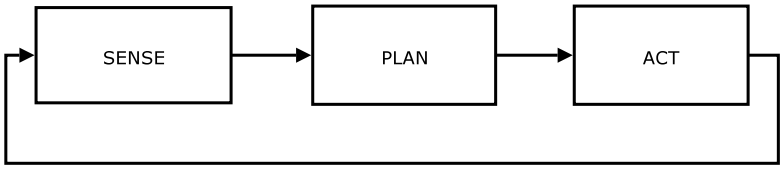

Figure 2.1: The Sense-Plan-Act loop in the Hierarchical paradigm

the *sense* step the robot senses the world through its sensors and uses the information it receives to create som kind of global representation of the world e.g. a map. During the *plan* phase it then analyses the world representation and plans the actions needed to reach its goal. Finally during the *act* phase it performs the action decided upon. This sense-plan-act sequence is then restarted

and the world is sensed again. The three main steps of the cycle should ideally be performed many times every second in order for the robot to be able to react quickly to events affecting it or its environment.

There are several problems with this approach. It is difficult to design a world model that can deal with everything that can affect the robot. In general the model needs to hold many types of information. It probably needs a map of the physical environment which can be both previously known to the robot as well as dynamically generated from its sensors. Also general rules need to be known e.g. "don't drive around between 13.30 and 14.45" as well as more specific rules e.g. "I can re-charge my batteries in blue wall sockets" or "trash should be left in room 5". The robot must deal with the *closed world* problem meaning that it assumes that its internal world model holds every bit of information it needs, which it obviously doesn't. So the designer of the robot controller must try to anticipate every single thing that can happen to the robot and incorporate it into the world model.

All these rules must be taken into account when the robot plans its actions. This often makes it slow moving since the *plan* step takes a long time. The early robots using this technique often moved extremely slow, as evidenced by Shakey (1967), the very first robot to employ it. Shakey could take several minutes up to close to an hour to plan its next action [4]. Even though over the years the increasing power of computers made the hierarchical paradigm more useful, clearly something faster was needed.

2.1.2 The Reactive Paradigm

During the 1980's a new way of looking at robotic behaviors emerged. By looking at animals and insects and how they behaved the realization was made that in many situations they don't consciously plan their actions at all. So maybe in some cases extensive planning like in the Hierarchical paradigm was not the way to go?

The idea is to remove the *plan* phase from the controller, and instead have pre-decided reflex-behaviors that are triggered when certain stimuli are presented. Logically this can be thought of as $If - then$ rules, e.g. "If light hits the robots right-hand sensor, turn left". By combining several of these rules, complex behaviors can emerge. The removal of the *plan* stage (figure 2.2) means the response time for a certain stimuli can be very fast, often as fast as reflexes in animals [4].

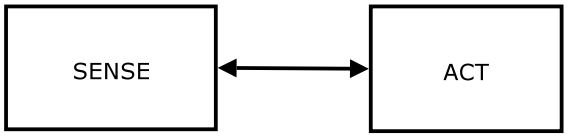

Figure 2.2: The Sense-Act loop in the Reactive paradigm

There is some contention over what exactly constitutes a reactive behavior, here it is intended to mean a behavior that is strictly *stimulus-response* (S-R) i.e. it requires no computations to activate, it is "hardwired" from the sensors to the actuators (actuators are the devices the robot has that interacts with the world e.g. wheels or arms). This hardwiring might be simulated in software but in

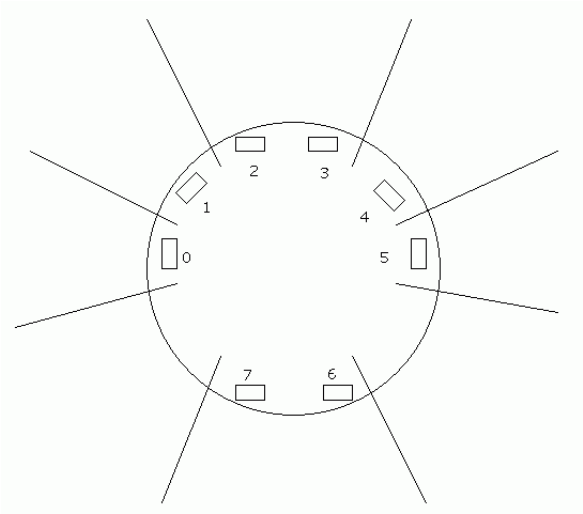

Figure 2.3: The Khepera robot and its sensors viewed from above

principle it could be implemented in hardware. This specific type of behavior is sometimes called a *reflexive* behavior.

Generally reflexive behaviors are divided into several types: *reflexes*, *taxes* and *fixed-action patterns*. Reflexes are behaviors where the response lasts only as long as the stimuli and is proportional to the intensity of the stimuli. Taxes are specific reflexes where the response is to move in a particular direction, e.g. move towards or away from light. Fixed-action patterns are reflexes where a stimuli triggers a sequence of actions that may continue even though the stimuli is removed [4].

A mechanism is needed to deal with the case of conflicting reflexes i.e. if more than one reactive behavior is triggered simultaneously. Many methods exists, including weights, where each rule has a priority/weight and the highest weighted behavior is the "winner", and voting, where all rules vote for an action and the action with the most votes gets carried out or alternatively if the actions can be combined (e.g. if they make the robot turn X degrees) then an average of the triggered behaviors can be carried out.

Fixed-action patterns are hard to create without going away from the strictly reflexive "hardwired" approach. Since the behavior should continue after the original stimuli is gone or change while the same stimuli remains, the robot needs some way of remembering the triggering stimuli or its actions which means it needs a memory or an abstract world-state representation.

2.1.3 The Khepera Robot

The robot used in this study is the Khepera. It is a small cylinder-shaped robot on wheels with 8 light sensors and 8 proximity sensors mounted around it. The sensors have an effective range of approximately 50 mm and both types are mounted in pairs at the same position. The figure 2.3 shows a schematic image of the Khepera viewed from above with the sensors numbered 0 to 7 and the sensors' field of view indicated with "cones".

On advantage of using the Khepera is the availability of the simulator *KIKS*[8] for the Matlab environment. The tests done in this study have been run in this simulator.

2.1.4 Odometry

The Khepera robot has two *proprioceptive* sensors. Proprioceptive sensors have the function of sensing internal movements, or more formally they measure movements relative to an internal frame of reference [4]. A biological example are the nerves and organs that help the human body know where its arms are located even when they can not be seen or touched by other parts of the body.

In the case of the Khepera, the proprioceptive sensors are *shaft encoders* that measure the rotation of the robots two wheels. They function by shining a light at a rotating disc located on the wheel axis. This disc contains small holes and by counting the number of light pulses that shines through the holes in the disc onto a light sensor on the other side, a measurement of the number of rotations performed by the wheels axis is made. The information is then used to calculate how far the individual wheel has traveled, creating an *odometry* sensor.

These sensors can be used to implement a direction sensor. This sensor measures which direction the robot is currently facing by continually measuring the individual distances traveled by the wheels (using the odometry sensors based on the shaft encoders), the difference between them is then used to calculate how the robot has turned.

Since both wheels are mounted on the same axis they are locked in place parallel to each other. If the wheels move at different speeds, they are forced to rotate around a common point of rotation, often called *ICC* (Instantaneous Center of Curvature). If t is the time then the equation used for finding the new heading θ' based on the old heading θ is:

$$\theta' = \omega \delta t + \theta$$

Where ω is the speed of rotation around the ICC and is defined as:

$$\omega = (d_r - d_l)/l;$$

Where d_r and d_l are the distances traveled by the right and left wheel respectively, and l is the distance between the wheels i.e. the length of the wheel axis. Figure 2.4 helps visualize the concepts.

This gives the complete equation for calculating the new heading θ' based on the old heading θ:

$$\theta' = \theta + t(d_r - d_l)/l$$

Which can be used directly by measuring d_r and d_l with the help of the shaft encoders, and knowing that the distance l between the wheels of the Khepera is 49 mm.

Note that in reality there is always a slight slipping and skidding in the wheels' interaction with the ground surface which introduces errors in the calculation. These errors accumulate quickly, making the direction sensor somewhat inaccurate.

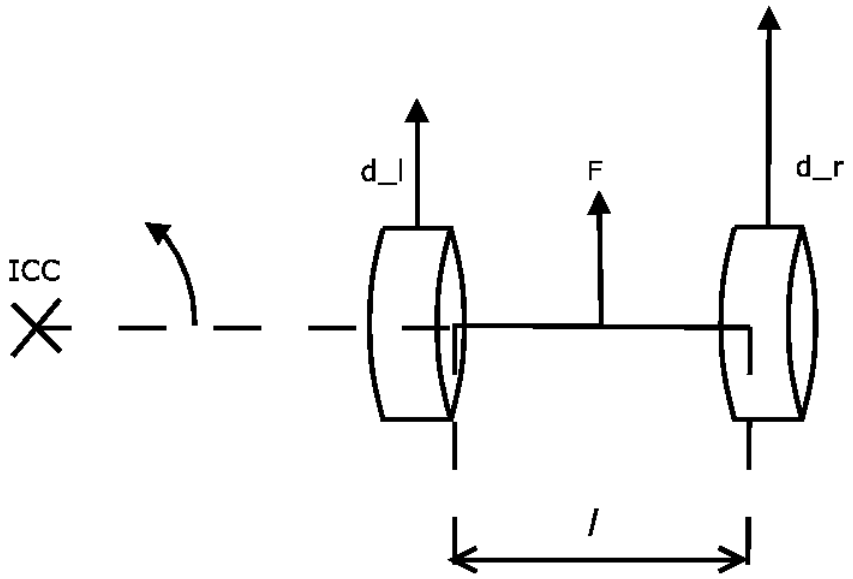

Figure 2.4: Visual display of the variables and concepts used for implementing a direction sensor. *ICC*: Instantaneous Center of Curvature. l: The distance between the two wheels. d_l, d_r: The distance traveled by the left and right wheel respectively. F: The direction the robot is facing.

2.2 In-depth Study: Data mining

Data mining is a field of science that has received much attention during the last decades.

2.2.1 What is data mining?

Initially the term *data mining* was used in a slightly derogatory sense when referring to "nonsense" information found by using statistical methods to find non-causal statistical correlations [6]. For example the finding that the length of womens' skirts where directly related to the global economy, which is/was merely a statistical fluke (pure coincidence if you like) and can not be used to predict anything about the economy [6].

The size of databases has increased in step with the rapid advancements of digital storage technology and data acquisition during the last decades. Many different fields, from everyday businesses to medical research to astronomy, have access to enormous databases filled with information. This has given rise to a desire to extract useful information from these data sets that are way too large for a human to manually search through. The scientific discipline concerned with this task is today called *data mining*. Hand, Mannila and Smyth (2001) define data mining as:

> Data mining is the analysis of (often large) observational data sets to find unsuspected relationships and to summarize the data in novel ways that are both understandable and useful to the data owner [6].

Data mining is related to statistics, and statistical analysis is in fact an important tool in many types of data mining. But there are several differences between the fields. One important difference is often the size of the data set [6]. While classical statistics generally deal with hundreds or possibly thousands of data points (*samples*), data mining often deal with millions of samples. Also an important difference is that data mining often uses "opportunity" samples as opposed to statistics that wants to use "random" samples [6].

In practice this means that a classic statistical analysis carefully decides the questions to ask and then selects the data samples (survey subjects) to achieve a good average representation of the population base. A data mining search on the other hand, would first take the data already available, which would not necessarily be a good random representation of the population, and then analyze the data to see what questions could possibly be answered by it.

2.2.2 Components of data mining

Data mining is not a single technique. It is an umbrella term for many different tools, algorithms and methods to achieve what is sometimes called *Knowledge Discovery in Databases* (KDD). What method to use depends on the type of data available and the type of result wanted. Generally the process of KDD consists of a number of steps:

- Determining what type of result is wanted.
- Determining how the available data is structured. Does it need to be transformed or filtered etc. ?
- Determining what method or algorithm to use.
- Evaluate the result. Is it useful?

2.2.3 Determining the type of result wanted

The knowledge found by a data mining exercise can be structured in many ways. The goal is often to find a *model* of the data. A model is a *global* summary of a data set i.e. something that holds true for the entire set.[6] As opposed to a *pattern* which is something that holds true for only part of the data.

An example of a global model could be the finding that reveal that working peoples income is related to their age by the rule:

$$Y = aX + b$$

Where Y is the income, X is the persons age and a and b are constants found by the data mining algorithm. This particular *model* holds true for all data in the set (probably with a margin of error).

An example of a local pattern, on the other hand, could be the rule that says[6]:

$$if X > x_i \rightarrow prob(Y > y_i) = p$$

Which means in words that if the variable X is larger than a number x_i then the probability for the variable Y to be greater than the number y_i is equal to

p, where x_i, y_i and p are constants found by the data mining process. In other words the rule only applies to the data points that has the variable X larger than a certain value i.e. it only applies to a local part of the data set.

The structure of the result wanted need to be specified before a choice of tools can be made. If the goal is to find a linear formula to describe the likely distance between pine trees of a certain age, other tools need to be used than if the goal is to find rules that explain the likely purchases on a friday afternoon by a 32 year old married male working in real-estate.

2.2.4 Different approaches to the data

Hand, Mannila and Smyth (2001) categorizes data mining into several types of *tasks*[6]:

- Exploratory Data Analysis (EDA)
- Descriptive Modeling
- Predictive Modeling
- Discovering Patterns and Rules
- Retrieval by Content

A brief explanation of each task follows.

Exploratory Data analysis

If the goal of the data analysis is not defined and the data is just being explored in a general way to look for *any* interesting knowledge, then this is the task at hand. The techniques are often visual and interactive [6]. If the data points/samples have low-dimensionality this can be effective but if there are many variables then this becomes difficult and some type of data preprocessing must often be done before work can start.

Descriptive Modeling

The goal here is to describe all of the data or the process generating it. This can include the overall probability distribution (*density estimation*), the partitioning of the dataspace into groups (*cluster analysis*) and models describing the relationships between variables (*dependency modeling*) [6].

Predictive Modeling

In this task the aim is to find a model that will allow the value of one variable to be predicted based on the known value of other variables (*Classification* and *Regression*). The difference between this and Descriptive Modeling is that Prediction looks at one unique variable while in Description no single variable is more important than another [6].

Discovering Patterns and Rules

This is related to Predictive Modeling but here the search is for local patterns, not global models. An example could be finding which grocery products are often purchased together, or detecting fraudulent behavior in bank transactions. This area has been the focus of much research and several algorithms exists for finding *association rules* [6].

Retrieval by content

If the goal is to find patterns in the data that are equal to a pattern already known, this is the task at hand. An example could be finding an image looking like another image, or finding a sentence of text in a huge text database. Finding relevant documents based on keywords appearing in the documents (i.e. a typical Internet web-search) is a common goal [6].

2.2.5 Discovering Patterns and Rules

As has been stated the field of data mining covers many tasks and methods. This study has focused on using Association Rules as a tool for machine learning, therefore this topic will be looked at in more detail.

This area of research has many uses. One example could be to find patterns in the sales data from a supermarket (this is one of the early uses of data mining, called *market basket analysis*). If every item purchased by a customer is recorded and tied to a "basket" i.e. a record is made of which items are sold together, lots of potentially interesting patterns could be found in this data. A store owner would be interested in knowing that if a customer buys bread, then he/she is likely to buy cheese as well with a probability of e.g. 0.8. Or maybe the knowledge that a customer who buys a certain brand of cereal is more likely to buy running shoes than other customers.

2.2.6 Rule structure

A *rule* has a left hand side (LHS) and a right hand side (RHS). Both sides are boolean statements. The meaning of a rule is that if the left hand side is true, then the right hand side will also be true. An extension to this is the *probabilistic rule* which says that if the left side is true, then the right side is true with a certain probability [6].

An example from the realm of basket data could be one where each item available for sale is represented by a binary variable which has the value *true* if the item is in the current basket and *false* if it is not. Thus a product basket from a supermarket with thousands of products would most probably be represented by a high-dimension vector with almost all values set to *false* since most customers would be buying only a small part of the entire set of products available. A customer only buying tomatoes and salad for example, would have a basket with only the variables representing tomatoes and salad set to *true*, while all other (thousands) variables would be false.

A useful pattern could be this:

$$X_{tomatoes} = true \wedge X_{sallad} = true \rightarrow p(X_{bread} = true) = 0.75$$

Which means that a customer buying both tomatoes and salad has a 75% probability to buy bread as well. So the store manager should probably think about placing the shelf containing bread close to the shelf containing salad and tomatoes.

2.2.7 How to find patterns

If patterns can be represented in a way that can be applied to the data, then a trivial way would be to search for every possible pattern in the data. This might work for very small patterns but is likely to be totally unusable. If the purchase basket has 2000 possible items then the number of patterns needed to test for are 2^{2000} which is way to many. Fortunately there exists several better algorithms for this purpose.

2.2.8 The Apriori algorithm

Several algorithms for finding association rules exists. They can be roughly categorized into two categories depending on weather they employ a Breadth-First search (BFS) or a Depth-First search (DFS) when evaluating candidate patterns [3]. Examples of algorithms using DFS are *FP-growth* and *Eclat*. One algorithm using BFS is *Apriori* and all of its derivatives like *AprioriTID* and *DIC*. Apriori is one of the most popular algorithm for finding association rules[3] and will be described briefly here.

The Apriori algorithm was one of the earliest algorithms for finding association rules (*probabilistic rules*). Several improved algorithms exists based on Apriori, one of them is the OPUS algorithm used in the program "Magnum Opus" which is a commercial program for finding association rules, used in this study. The algorithm is particularly well suited for searching sparse data sets which is appealing when dealing with basket analysis[6]. The rules it produces have the following structure:

IF $A = a$ AND $B = b$ THEN $p(C = c) = p_{ab}$

Where $p(C = c)$ is the probability for the variable C to have the value c. The probability p_{ab} is often called *accuracy*, *strength* or *confidence*. The strength of a rule is considered a good indicator of the rules quality which is logical since it tells how often the rule holds true.

Another important measure is the *support*. Support is defined as $p(A = a, B = b, C = c)$ (from the example rule) i.e. the fraction of entries in the data set that fulfill the rule. E.g if A = 1 for 50% of the transactions and B = 1 for 50%, and both A and B equals 1 in 45% of the transactions then the support for the rule IF $A = 1$ THEN $B = 1$ would be 45% but the strength of the rule would be the support for the pattern $A = 1 \wedge B = 1$ divided by the support for $A = 1$ alone i.e. $0.45/0.5 = 0.9$

The search for association rules uses a score function to see weather a rule is interesting or not. This score function is a simple binary function. In the basic

search there are two thresholds: p_s for the minimum support required and p_a for minimum strength required. A pattern gets a score of 1 if it satisfies both requirements and 0 otherwise [6].

As has been noted, the actual search space is enormous ($O(p2^{p-1})$ for a basket with p possible items), however this space can be effectively pruned.

An event is called *frequent* if the probability of it occurring is greater than the support threshold p_s and using the simple fact that if either $p(A = 1) < p_s$ or $p(B = 1) < p_s$ then $p(A = 1, B = 1) < p_s$, and useless candidates can be discarded[6]. This means that going from frequent sets of size *k-1* to size *k*, any sets containing subsets that are not frequent can be discarded. Any remaining candidate sets of size k are then examined for actual frequency and the ones failing to reach p_s are discarded. This is repeated for sets of size *k+1* and so on until no more frequent sets can be generated.

This means that in the worst-case where all possible sets are frequent the algorithm takes exponential time, but in practice the data sets analyzed by this and similar algorithms are very sparse, making the pruning process very efficient if the value for p_s is not too low[6].

There exists several techniques to optimize the performance of this algorithm, and a common one is to use sampling instead of doing the actual exact counting of each subset of variables. Since the data sets are often huge, a size on the order of 10^6 entries is not uncommon. A random sampling of a much smaller number of data points often yields good results in regards to subset pattern frequency without reducing accuracy too much. This method has it's drawbacks though e.g. if a pattern is not uniformly distributed or if the number of data points is small.

2.2.9 Rule quality

Several measurements of a rules usefulness exists, they include *support* and *strength/accuracy* (explained above), and also *coverage*, *lift*, and *leverage*. In the following explanations, X and Y are records in a database, each containing one or more terms, and a rule has the general structure of $X \rightarrow Y$:[1]

coverage

Coverage is defined as

$$coverage(X \rightarrow Y) = cover(X)$$

where $cover(X)$ is the fraction of records or number of records containing X.

lift

Lift is defined as

$$lift(X \rightarrow Y) = \frac{support(X \rightarrow Y)}{cover(X) * cover(Y)}$$

which corresponds to the ratio of the number of records containing both X and Y and the number of records expected to contain X and Y if the two where independent. Note that just as *strength*, *lift* gives no information on how often a pattern occurs.

leverage

Another measure of rule quality is *leverage* which is defined as

$$leverage(X \rightarrow Y) = support(X \rightarrow Y) - cover(X) * cover(Y)$$

This value indicates the number times the pattern occurs beyond what would be expected if X and Y where independent.

These qualifiers can be used to determine the suitability for a rule, but it is hard to tell beforehand which of these will be important since different patterns and processes have different structures and therefore en event/pattern deemed important in one data set might not be important in another. Generally substantial human supervision is needed to create and evaluate a set of association rules.

Chapter 3

Experiments

3.1 Using Association Rules in a Robotic Controller

As has been stated, association rules can have many uses in many fields. Often they are used to search and analyze data for non-obvious knowledge. Hellström [1] showed that a set of these rules can also be used to build robotic controllers.

3.1.1 Generating data

The general idea is that a "training run" is made where the robot is either controlled remotely by a human operator or directly by a handwritten control program. During this run, the robot performs the desired behavior. All data generated by the robots sensors, as well as the actions the robot executes, are logged during this training run. This procedure by necessity generates a number of discrete sensordata-and-robotic-action samples, one for each time step. Since a typical robot should react as quickly as possible the time steps probably need to be fairly short, on the order of several steps every second.

This data is then searched for association rules. Interesting rules would be of the type:

IF sensordata $= X_v$ *THEN action* $= A_x$ *with a probability of P*

Where *sensordata* is one or more variables corresponding to one or more sensors on the robot, and X_v is a vector containing the data from the sensor/sensors in question, and A_x is the action taken by the robot when X_v has a certain set of values. The final probability constraint means that the statement holds true for a fraction P of all cases where the rule could be applied (i.e. the accuracy/strength/confidence of the association rule).

A simple actual rule could be:

IF $X_1 = 2$ *THEN p(Y = 18) = 0.83*

Where X_1 could be the data output from a forward looking sensor on the robot

and the value 2 would indicate that an obstacle is close, and p(Y = 18) = 0.83 means that in 83% of the cases where this happens, the robot should execute action number 18 which could be "turn around on the spot".

3.1.2 Usefulness

It is easy to see that a number of rules could be generated like this and the resulting rule-base has the potential to be used for controlling a robot. It should be pointed out that in order for this to be useful in practice, the process would need to be automated to a large degree. A human controller during robot training is useful since it would more or less eliminate the need to write any custom software for the robotic controller. Having to program a customized controller for each robot and each behavior in order to generate the data needed would of course make the technique useless since the desired behavior would have been already created for the training run and a new controller based on association rules would not add anything of value. However it could be possible to write a more general and powerful "training controller" with the ability to create many different behaviors, and then use customized association rules for a slimmer, less general controller with lower hardware requirements with respect to computational power.

3.1.3 Problems

Due to the nature of the algorithms used when searching for association rules, some problems arise when doing the training runs. One specific problem is that important events might be rare. This means that when the robot reacts to something in its environment (or more correctly it reacts to sensordata generated by events in the environment), those reactions are sometimes very brief. For example if the robot is designed to avoid sudden obstacles by turning sharply to the right, and each cycle in its sense-act loop (see section 2.1.2) takes 0.1 seconds, then the obstacle might be out of the robots field of view after 1 second i.e. 10 cycles of turning. If the training run was 100 seconds (1000 cycles) this means that the important event only affects 1% of the the sensor data samples. Since the OPUS algorithm in part decides what patterns are important based on their frequency of occurring, this could be a problem since the frequency would be very small. The patterns can also be filtered by their strength or any other quality measure, but by removing patterns with very low frequency some "noise" is removed from the rules e.g. a pattern that occurs exactly once in the entire training data set is probably not very useful even if it's strength is very high and by setting a limit of minimum support or coverage these types of patterns are removed from the rule base. This means that the user must be careful not to set to high a limit (see section 2.2.8) and it is also important that the training run is designed so that events judged to be important for the robot to react to happen fairly often.

3.1.4 Rule limitations

Association rules are good candidates for running robots with reactive behaviors (se section 2.1.2) since their structure is if the type *IF-THEN* with no deliberating or planning or looking at other data than is included in the rule itself. In

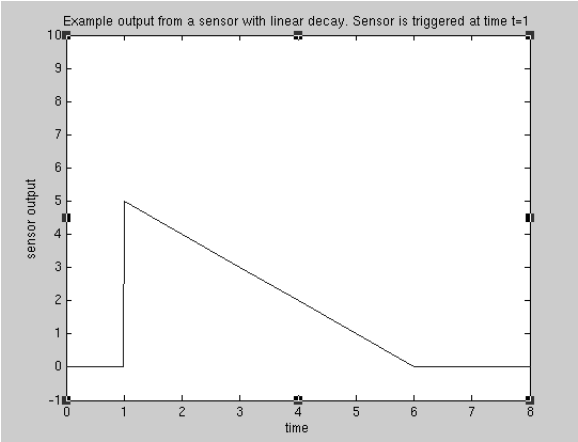

Figure 3.1: Example of linear sensor decay

other words, the left hand side of the rule looks at the values of certain stimuli and if the stimuli is "correct" a corresponding behavior is triggered. Very much like a reactive behavior or reflex.

Rules generated from "plain" sensordata seems to be quite limited in the type of behaviors they could create in that they can only be based on the immediate data given by the sensors i.e. they can only react to the current state of the world and do not "remember" anything from the past. More clearly, the rules can only be of the type:

IF the current *state of the world is* X_v *THEN...*

and not of the type:

IF the previous *state of the world was* X_v *AND the* current *state is* Y_v *THEN...*

It is clear that if rules of this second type could be found, then more complex behaviors could be created, more specifically *chains* of actions taking place after each other depending on *sequences* of events in the world.

Hellström [1] [2] successfully showed that by introducing time-dependent variables into the sensordata more complex behaviors can be created. The idea is to introduce a level of data preprocessing into the sensors themselves. A simple example could be *sensor decay* (see figure 3.1)which means that a sensor that has been triggered by a certain stimuli continues to produce output indicating that the stimuli is present even *after the stimuli is gone* for a period of time.

Other possibilities include, but are certainly not limited to, *habituation* (see figure 3.2) and more advanced processing like *odometry* (see section *3.1.10*). Habituation means a sensor "gets used to" a certain stimuli, resulting in the initial response to the stimuli being high i.e. creating a high value on the sensor output, but gradually the response lowers even though the stimuli is still present.

It is perhaps important to note that this is strictly a step away from the purely reactive paradigm. However the point of the reactive paradigm is not necessarily to forbid the use of other techniques but to strive for simplicity and avoid

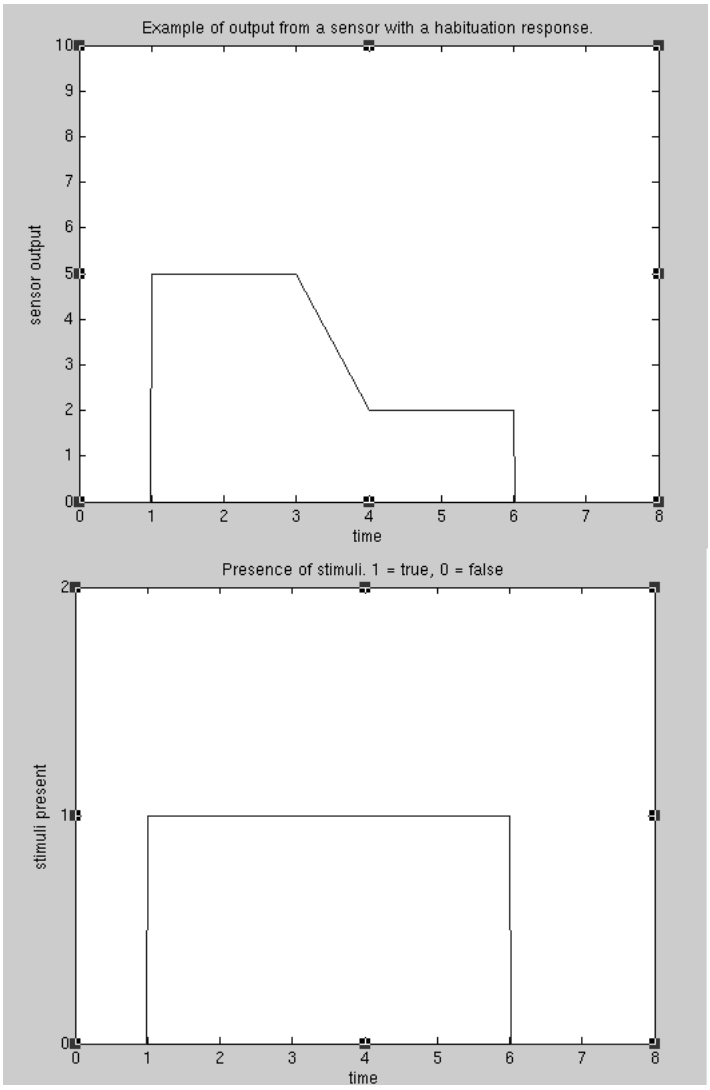

Figure 3.2: Example of sensor habituation

unnecessary processing and "deliberating" while creating behaviors and deciding actions.

The idea of introducing "virtual" or "artificial" sensordata can be viewed as putting a limited processing capability into the sensors themselves which is certainly not unknown of in nature, for example the information from the human eye seem to be subject to some fairly advanced image pre-processing before the information is sent to the conscious mind to deal with [9]. The important thing is that the mechanism that decides on actions based on the (possibly pre-processed) sensordata is strictly reactive and makes no deliberation and keep no internal representation of the world.

3.1.5 Using the Rules in a robot

The goal is to both verify Hellströms results [1] [2], and test if more complex behaviors can be created using association rules while still staying with the reactive paradigm. The software used is Magnum Opus which is commercially available. It is configured to order the association rules it finds by their strength, and also to discard a rule that it deems to be "insignificant" which means that the rule does not add enough strength when compared to another more general rule which is a subset of the candidate rule[7].

3.1.6 The road-sign problem

A very simple behavior that will be looked at is a variation of the "road-sign problem" wherein a robot travels though corridors and at intersections it turns in one direction (e.g. right) if it has recently "seen" a light and another direction (left) if it has not. This behavior can be seen as constituting two "sub-behaviors" (*follow left wall*, and *follow right wall*), which one to execute being dependent on the time passed since detecting a light. Care must be taken to make sure both behaviors are executed fairly often during the training run, making each pattern more easily detected by the data mining software.

Hellström successfully created this behavior by adding a new variable to the data indicating a light source was present, affected by perceptual decay meaning the light sensor still indicated light was shining on it even some time after the light was gone [1].

The setup of this experiment is almost identical, with the addition that while Hellström used a binary variable (more like a boolean flag) to indicate light[1], here a "counter" will be used, implementing a form of sensor decay indicating the number of time cycles since a light was last seen. This can be seen as making the process slightly more general since it could trigger different behaviors after different time periods has passed since the light was detected (though this is not done).

The actions available to the Khepera are:

- -3: turn left sharply
- -2: turn left moderately
- -1: turn left slightly

- 0: don't turn (move straight ahead)
- 1: turn right slightly
- 2: turn right moderately
- 3: turn right sharply

The sensors used by the khepera are:

- prox1, prox2: The left and forward-left facing proximity-sensor (IR-sensor)
- prox3: A composite sensor that gives the average output of the two forward facing proximity-sensors (IR-sensors)
- prox4, prox5: The forward-right and right facing proximity-sensor (IR-sensor)
- light: Triggered when the average output from the 8 light sensors goes above a threshold, and is then affected by sensor decay until it fades out.

When triggered, the light sensor is set to the value 50 and counts down by one every loop-cycle, effectively giving the robot a 50-cycle memory (see figure 3.1). The proximity sensors normally give readings between 0 (no object in sight) and 1024 (object is extremely close), and these outputs are discretized into 3 ranges by the following rule:

```
if sensorsvalue <= 100 then
   sensorsvalue = 0;
else if sensorsvalue <= 400 then
   sensorsvalue = 1;
else
   sensorsvalue = 2;
```

Which means a close object will register as a value of 2 on a sensor, an object at medium or far distance will register as a value of 1, and a "clear view" will give the value 0.

The environment for the training run is pictured in figure 3.3. White color represent walls and grey represents clear areas where the robot is free to move. The robot itself is represented by a dark grey circle. The arena is designed to make the training run trigger many "important" events i.e. has many intersections where the robot has to turn based on previously detected light. This is to avoid the problem of important events being infrequent making it more likely that the data mining software will find patterns containing the relevant actions.

The robot is run with a cycle time of 0.1 seconds for 60 seconds resulting in 600 samples. The data logged for each time step is 5 values for the forward and side facing proximity sensors, 1 value for the counter indicating time steps since a light was last seen by the light sensors, and finally 1 value for the action taken this particular time step.

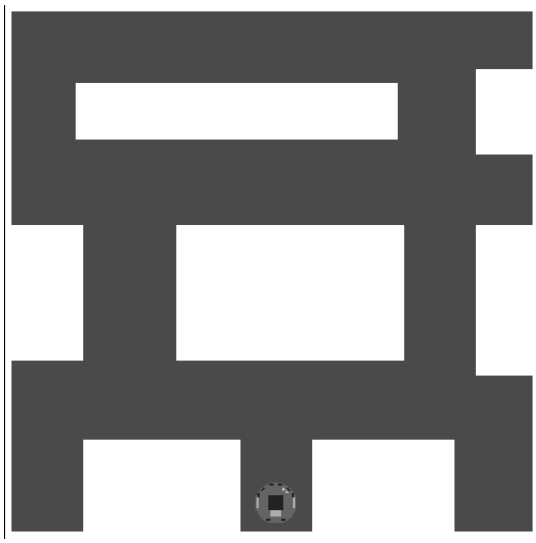

Figure 3.3: The layout of the arena used for the road-sign controller. White color means impassable terrain (walls), and grey represents clear space where the robot can move (roads).

3.1.7 Road-sign performance

It should be clarified that an exact measure of how many correct vs. faulty decisions the rule-based controller produces is not that informative. Considering that the actions taken all indicate how much the robot should turn (or not turn) a decision to "turn left slightly" could be completely acceptable even if the correct action to take is "turn left moderately". As long as the robot turns left/right depending on previously detected light, and avoids wall-collisions, the behavior is essentially correct. Because of this a subjective judgment is made on the quality of the rule-produced behavior.

3.1.8 Strength-based rule ordering

Hellström [1] [2] successfully created a rule based controller using strength-ordering only, and majority voting for multiple rules firing. This method is first tested.

With no limits at all besides the test for rules being "insignificant", Magnum Opus finds 52 patterns in the data from the training run (see table 3.1 for part of the rule base)

It's possible that all rules are not needed for a useful controller, therefore different limits on the minimum strength a rule is allowed to have are tested. If a rule has a lower strength than the limit, it is not used in the rule base. The table 3.2 shows the results of the different controllers. The *Error* measure in the table indicates the fraction of incorrect decisions the controller makes over the entire run. The error rates are not very good indicators of performance so a subjective judgment is listed for each test below:

no.	Rule LHS	Rule RHS	Strength	Support
1	prox3=2	action=-3	1.000	0.010
2	prox2=2 & light<1	action=-1	1.000	0.067
3	prox4=2 & 1<=light<50	action=1	1.000	0.073
4	prox1=0 & prox5=1	action=-2	1.000	0.037
5	prox5=1 & 1<=light<50	action=-2	1.000	0.080
6	prox5=0 & 1<=light<50	action=-2	1.000	0.037
7	prox2=1 & prox3=1	action=2	1.000	0.007
8	prox4=0 & prox5=2 & 1<=light<50	action=0	1.000	0.267
9	prox1=2 & prox2=0 & light<1	action=0	0.983	0.193
10	prox1=1 & light<1	action=2	0.969	0.103

Table 3.1: The first 10 rules (out of 52) of the rule base found with no limitations on the data mining run, ordered by strength

Str. limit	No.	Error
1.0	8	53%
0.95	10	63%
0.9	14	65%
0.8	18	75%
0.7	26	66%
0.6	30	48%
0.5	32	42%
0.4	36	60%
0.3	40	74%
0.2	47	68%
0.1	52	37%
0.0	52	37%

Table 3.2: Error results for a controller using rules from an unfiltered rule base of 52 rules. *Str. limit* indicates minimum allowed strength for a rule to be used in the controller. *No.* indicates how many rules qualify for the strength limit.

- Strength limit 1.0, 0.95 and 0.9: Avoids walls but always turns right at intersections, never reacting to light.
- Strength limit 0.8: Avoids walls but always turns right at intersections, never reacting to light. Sometimes runs in circles in open spaces.
- Strength limit 0.7: Avoids walls. Starts to turn left at intersections but "turns back" before completing the turn. Collides with walls if hit dead-on.
- Strength limit 0.6: Runs in circles in open areas. Starts to turn left at intersections but "turns back" before completing the turn.
- Strength limit 0.5: Avoids walls. Starts to turn left at intersections but "turns back" before completing the turn. Turns right sometimes at intersections.
- Strength limit 0.4: Avoids walls. Starts to turn left at intersections but "turns back" before completing the turn. Reacts to light by moving straight.
- Strength limit 0.3: Turns left at intersections (correct). Has a tendency to collide with walls, especially when exposed to light which makes it move straight ahead.

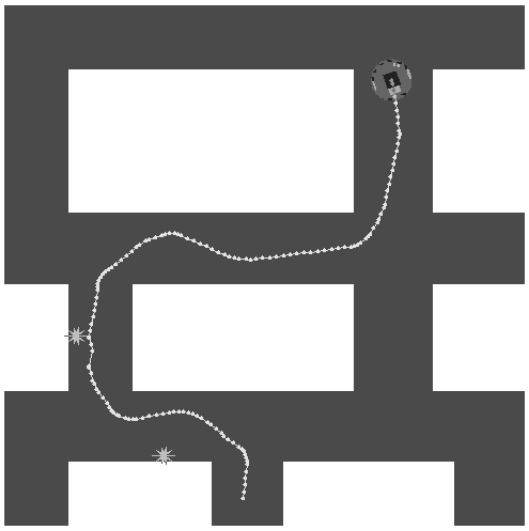

Figure 3.4: The movement path for the robot using the entire unfiltered rule base. Majority voting is used when multiple rules fire.

- Strength limit 0.2: Turns left at intersections (correct). Has a tendency to collide with walls, especially when exposed to light which makes it move straight ahead.
- Strength limit 0.1 and 0.0: Avoids walls. Turns left at intersections (correct), and right if recently exposed to light (correct).

The movement path of the robot in the successful run with a lower strength limit of 0.1 is shown in figure 3.4. The robot is supposed to turn right at intersections if it has recently seen a light, and left otherwise.

The result indicates that in this case rule importance can not be judged by the rules strength. The number of rules (52) needed corresponds fairly well to Hellström [1] who needed 43. Differences in the size of the training data set and different values used for sensor discretization probably accounts for the difference. As has been pointed out it is clear that in this case the exact number of errors is not necessarily a good indicator of performance. Despite a large number of errors (37 %) the controller is able to replicate the target behavior well. It is possible and even likely that some rules are redundant but it seems that in this case *strength* is not a good way of measuring rule quality.

3.1.9 Lift-based rule ordering

It is likely that some of the 52 rules in the rule base are redundant. By trying different ways of filtering the rules during the data mining process a smaller but still useful rule base can hopefully be found.

Systematic testing finds that filtering the rules based on their coverage gives a good result in this case. Setting the minimum limit for *coverage* to 0.3 yields a rule base of 14 rules with varying strength (see table 3.3).

no.	Rule LHS	Rule RHS	Strength	Support
1	dist4=0 & dist5=2	action=0	0.870	0.29
2	dist1=2 & dist5=2	action=0	0.830	0.44
3	dist1=2 & light<1	action=0	0.816	0.28
4	dist5=2	action=0	0.757	0.51
5	dist1=2	action=0	0.736	0.530
6	dist4=0 & 1<=light<50	action=0	0.714	0.267
7	light<1	action=2	0.272	0.133
8	1<=light<50	action=-2	0.229	0.117
9	dist2=0	action=2	0.209	0.107
10	dist2=2	action=-1	0.204	0.067
11	dist4=0	action=-2	0.167	0.107
12	1<=light<50	action=1	0.144	0.073
13	light<1	action=-1	0.136	0.067
14	dist5=2	action=1	0.109	0.073

Table 3.3: The rule base found with a limit of 0.3 for *coverage*, ordered by strength

A problem is that often several rules fire at the same time. In the previous test a majority vote was used to resolve this type of conflict, but other methods are possible and to evaluate this new rule base 8 different mechanisms are tested:

1. *Highest Strength*: The rule with highest strength is selected. If more than one rule has the highest strength, the first one found is chosen.
2. *Majority vote*: Each rule has one vote and the action with the most votes is chosen.
3. *Average action*: Since the actions in these tests are scalar responses that turns the robot in different directions, an average turn is calculated where all rules have the same weight.
4. *Strength weighted majority vote*: Each rule has a number of votes equal to its *strength* and the action with the most votes is executed.
5. *Strength weighted average action*: Each rule has a number of votes equal to its *strength* and since each action is an order to turn in a specific direction, an average action is calculated (same principle as (3)).
6. *Lift weighted majority vote*: Each rule has a number of votes equal to its *lift* and the action with the most votes is executed.
7. *Lift weighted average action*: Each rule has a number of votes equal to its *lift*, and an average action is calculated based on the number of votes each action gets (same principle as (3)).
8. *Highest Lift* The triggered rule with the highest *lift* is executed.

The test runs show that several methods give fairly good results. Method 6 (Majority vote weighted on *lift*) is judged to give a completely acceptable road-sign following behavior shown in figure 3.5 and 3.6.

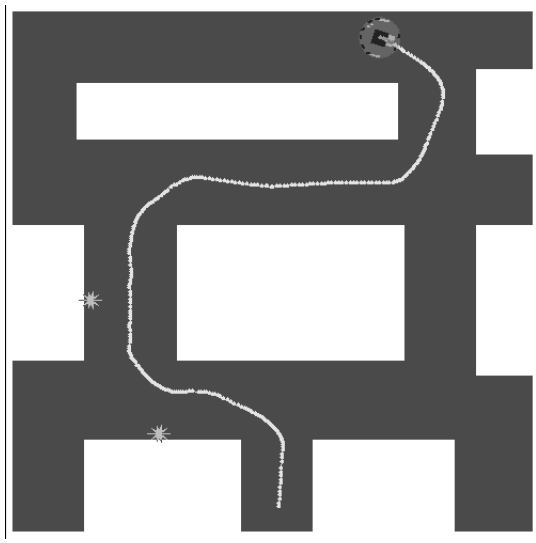

Figure 3.5: Robot performance in a successful test run, using a rule base of 14 rules and resolving multiple rules firing by a majority vote weighted on each rules *lift* (method no. 6). The controller makes a wrong decision 48% of the time.

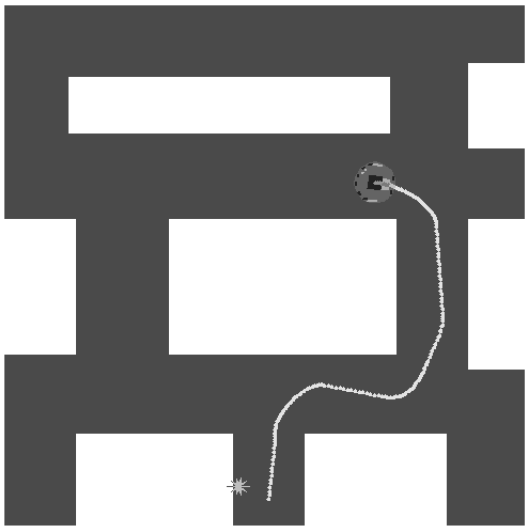

Figure 3.6: Robot performance in a successful test run, using a rule base of 14 rules and resolving multiple rules firing by a majority vote weighted on each rules *lift* (method no. 6). The controller makes a wrong decision 44% of the time.

As before, the robot is supposed to turn right at intersections if it has recently seen a light, and left otherwise. The test runs show that in the case of this particular behavior, the methods calculating an average response (methods 3, 5 and 7) generally produces poor performance. The "winner takes all" methods (methods 1, 2, 4, 6 and 8) typically give better results. Only one run showing the poor performance of the methods using averaged actions is presented (figure 3.7) but the same tendency is present in all three methods.

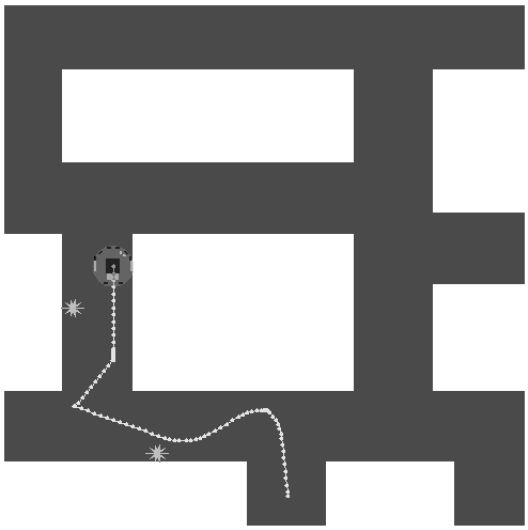

Figure 3.7: Robot performance in an unsuccessful test run using a poor-performing method of resolving multiple rules firing. The method used is no. 3 (average action with no weights) and the controller is using a rule base of 14 rules. The robot tends to turn very slowly and therefore constantly gets stuck against walls. The controller makes the wrong decision 74% of the time.

Interestingly the number of rules needed for a useful rule base is 14 (the entire rule base) which is significantly less than the 52 needed in the previous test. In essence, by applying stricter quality limits to the data mining process, the rule-quality filtering is moved to the data mining, away from the process of selecting rules when building the controller. Keep in mind however that because of the previously mentioned difficulty of judging performance in an exact analytical way it's quite possible that the controller built here has worse performance even though it is subjectively judged to be "good".

Nonetheless it's clear that if a good quality measure can be found and applied to the data mining process, the rule base can be significantly reduced. In this particular case a high *coverage* i.e. a rules LHS appears in the training data with a high frequency, indicates high usefulness for the rule.

Removing rules with low strength

Some rules in the rule base have very low strength which indicates that they are not of very good quality, or at least not very reliable. The next test however, shows that all rules are indeed needed. By specifying a lower limit of 0.12 for strength, the weakest rule is removed from the rulebase (see table 3.3), and the result is shown in figure 3.8 As can be seen, the removal of the low-strength rule makes the robot try to turn in circles when close to a light, which results in it colliding with the wall multiple times.

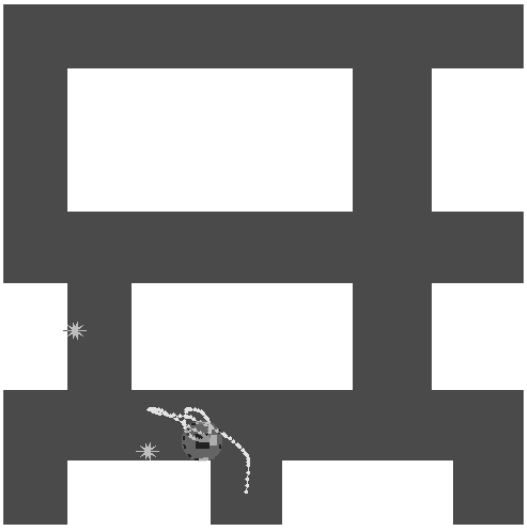

Figure 3.8: Robot performance in an unsuccessful test run, using the 14 rules from the original rule base and removing the weakest rule (strength = 0.109), resulting in 13 rules being used. Multiple rules firing are resolved by a majority vote weighted on each rules *lift* (method no. 6).

There might be other rules that are redundant (for example rule 4 is a more general subset of rule 1 but with a lower strength), but it seems that removing rules based on their low strength is not possible with this rule base.

3.1.10 Cockroach behavior

This is a much more complex behavior than the road-sign turner. A successful cockroach behavior means the robot moves "randomly" until it detects a light source, then it turns around and runs away. It runs until it reaches a wall and then it follows the wall until it finds a hiding place (a "hole"). Inside the hole it turns around and stops, ending its flight facing outwards from the wall. Like the road-sign behavior this behavior can be viewed as consisting of several "sub-behaviors": *avoid walls* (when not running away), *turn around* (when light is detected), *run straight ahead* (moving towards a wall) *follow right wall* and *follow left wall* (when running away), and *hide in hole*. All of these behaviors depend on the time passed since last a light was detected.

The actions available to the robot are the same as for the road-sign controller (section 3.1.6) with the addition of $action = 4$ which means the robot stops in place and is needed to ensure the ability to stop and hide when it finds a hole. This necessity more or less eliminates the possibility of calculating an average action when more than one rule fires. Additionally one new sensor is needed. In order to allow the robot to sense when an object is behind it so it is able to detect when it is in a "safe hole", a new sensor "prox6" is enabled. It outputs the average value given by the Kheperas two backwards-facing proximity (IR) sensors. The value is discretized in the same way as the others, resulting in a final value of 0, 1 or 2. This value is logged together with the data from the other sensors. The light sensor works similarly to the one in the road-sign experiment i.e. it has a linear decay, the only difference being it decays from

100 to 0 over 100 control-loop cycles (the road-sign sensor decayed from 50 to 0 over 50 cycles).

The training run has a cycle time of 0.2 seconds and generates 500 samples in the arena pictured in figure 3.9.

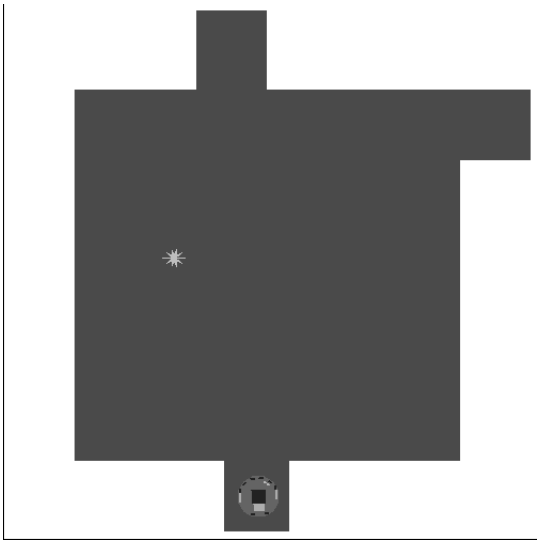

Figure 3.9: The training arena for the "cockroach" behavior. White color indicates walls and grey indicates open areas where the robot may move. The star indicates a light source.

The data mining finds 58 interesting patterns in the data. By setting a minimum lower limit for *strength*, different-sized rule bases are tested. Multiple rules firing are resolved by a majority vote. The results are as shown in table 3.4.

Str. limit	**No.**	**Error**
1.00	9	66%
0.95	14	63%
0.90	19	20%
0.85	22	10%
0.80	26	15%
0.70	32	15%
0.60	39	15%
0.50	44	47%
0.40	44	47%
0.30	49	29%
0.20	54	23%
0.10	58	27%

Table 3.4: Error results and general behavior for a controller using rules from an unfiltered rule base of 58 rules. *Str. limit* indicates minimum allowed strength for a rule to be used in the controller. *No.* indicates how many rules qualify for the strength limit.

As in the road-sign test the error results do not necessarily give a good indication of performance, therefore a subjective judgment is given:

- Strength limit 1.0 and 0.95: Collides with walls. Turns slowly. No reaction to light.
- Strength limit 0.90: Avoids walls. No reaction to light. No hiding.
- Strength limit 0.85 and 0.80: Avoids walls. Runs from light. Hides in hole but sometimes hides even without light.
- Strength limit 0.70 and 0.60: Avoids walls. No reaction to light. Hides in hole sometimes.
- Strength limit 0.50 to 0.10: Collides with walls. No reaction to light. Hides in holes sometimes.

Limiting the used rules to a strength of 0.85 or 0.80 looks promising but the controller is far from perfect.

On average approx. 20% of the rules in the rule base fire every cycle for this particular rule base. Instead of using a simple majority vote other methods are tested to resolve the conflicting actions. Since it was seen in the tests of the "road-sign" behavior that the exact number of errors are not necessarily a good measure of controller quality, the error results for the different methods are not specifically listed. The methods used for the road-sign test are all tested again with this rule-base.

Method no. 6 (majority vote weighted on *lift*) is found to give an acceptable result when used with the rule base limited to rules with a strength of 0.80 or higher. Interestingly the error rate increases from 15% to 24% but the behavior is subjectively judged to be better. The resulting path taken by the robot is shown in figure 3.10

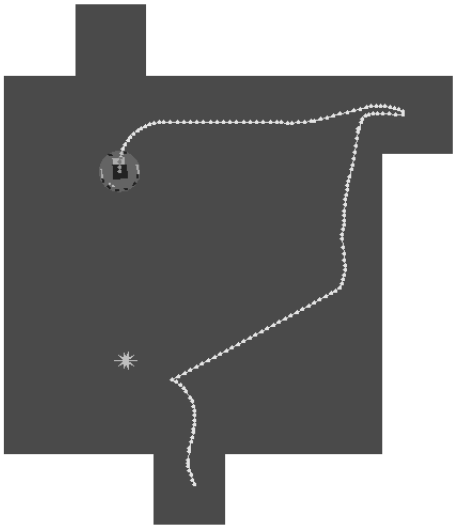

Figure 3.10: When using a lower limit of 0.80 for *strength* and resolving multiple rules firing by a majority vote weighted by *lift*, the robot imitates a cockroach by moving in the pattern indicated by the bright line in the image. The position of the light source is indicated by the star. The robot stays still in the hole for a short time before moving on, as it should.

While not visible in the figure, the behavior is not perfect. The robot "stutters" a bit when turning around in its hiding place and has some small problems with leaving the hole when it is supposed to. It stays for a while before moving out again.

Using the entire rule base of 58 rules and using the method of selecting the rule with highest strength (method no. 1) also reproduces the cockroach behavior fairly well as shown in figure 3.11 With this rule base the error rate is 22% but

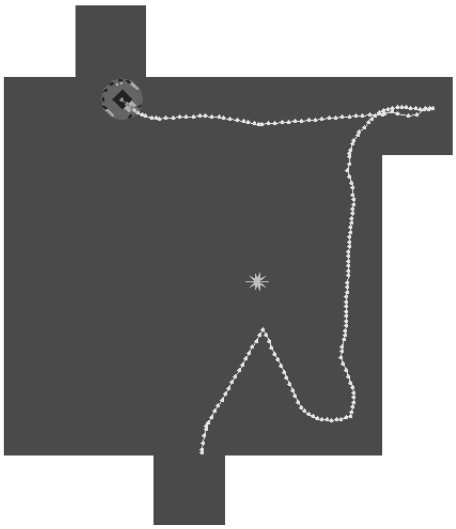

Figure 3.11: When using the entire rule base of 58 rules and resolving multiple rules firing by selecting the rule with the highest strength, the robot imitates a cockroach by moving in the pattern indicated by the bright line in the image. The position of the light source is indicated by the star.

subjectively the behavior is judged to be less well replicated than when using the previous technique. The controller has a stronger tendency to get stuck in its hiding place and even stops sometimes while not in a hiding place.

Filtering rules

The road-sign test generated a much smaller but still usable rule base with a lower limit of 0.3 for lowest allowed *coverage* (see section 3.1.7). This time the behavior is more complex and has more "sub-behaviors" which logically means that each behavior pattern has a lower frequency i.e. coverage and support, so a lower limit is probably needed. Extensive tests of the different limits for *coverage* and *support* fail to produce a useful rule base smaller than the one previously used.

This behavior was successfully created by Hellström [2]. Interesting here is that while Hellstöm slightly changed the sensor-preprocessing on the Khepera to create a *habituation* response from the light sensors to trigger a sequence of two time-dependent behaviors, here the same behavior is created using the exact same sensor pre-processing and logging as in the road-sign problem setup i.e. the light sensor implements a linear decay from 100 to 0 over 100 loop cycles. The controller used for collecting the training data is of course different but this

do not affect the association rules generated since they are based purely on the data and are not affected by the underlying data generating process.

This means the tailoring of hardware and sensors/preprocessing is less needed and instead this customization is moved into the association rule mining process. Instead of for example designing a sensor to give a habituation response tailored for the specific behavior, a more general sensor that simply counts time passed since the last interesting event can be used. This sounds promising but unfortunately the nature of the data mining algorithm (Section 2.2.8) forces the human user to indicate interesting values or value-ranges for each numeric variable. In practice it means that instead of customizing a sensor to give a certain output in two or more intervals of time, the interesting numeric intervals need to be specified by the operator during the data mining run, using knowledge of the problem.

3.1.11 More complex behaviors

As has been shown, time dependent behaviors can be generated using association rules. By introducing fairly advanced sensor preprocessing, potentially complex behaviors could be created. To test this a *proprioceptive* sensor is introduced.

The proprioceptive sensor takes the form of a direction sensor. This is implemented by having the robot sense how fast each of its two wheels are moving, and then continuously calculate the resulting direction it ends up facing (see section 2.1.4 for details). This is fairly complex preprocessing, but it does not require the robot to keep an internal representation of the world.

3.1.12 Moving in a square

This behavior means the robot will move around in a square, turning sharply at each corner. It should be pointed out that this behavior is impossible to reproduce in a purely reactive way with no sensor "intelligence", since the non-pre-processed sensors have no time-awareness making it impossible to have them produce different outputs depending on time or previous actions.

Training run

By implementing a form of sensor memory on the direction sensor the robot is made to turn 90 degrees at regular intervals. The memory is achieved by having the direction sensor output a vector containing the relative turns the robot has performed each second for the last 5 seconds. This can be viewed as having five separate direction sensors connected in series with a fixed latency as shown in figure 3.12. The "first" sensor $turn_1$ takes one sample s_n from the robots direction sensor dir every second (the subscript n indicates which second the sample was taken). By taking $s_n - s_{n-1}$ it then finds how much the robot has turned in the time between the two samples. This result is then output continuously to both the controller and the next sensor $turn_n$ in the series until a new sample is taken and the process repeated. By further introducing a latency in each sensor after the first, the original reading will propagate down the "chain" of sensors, creating a form of sensor history available to the controller.

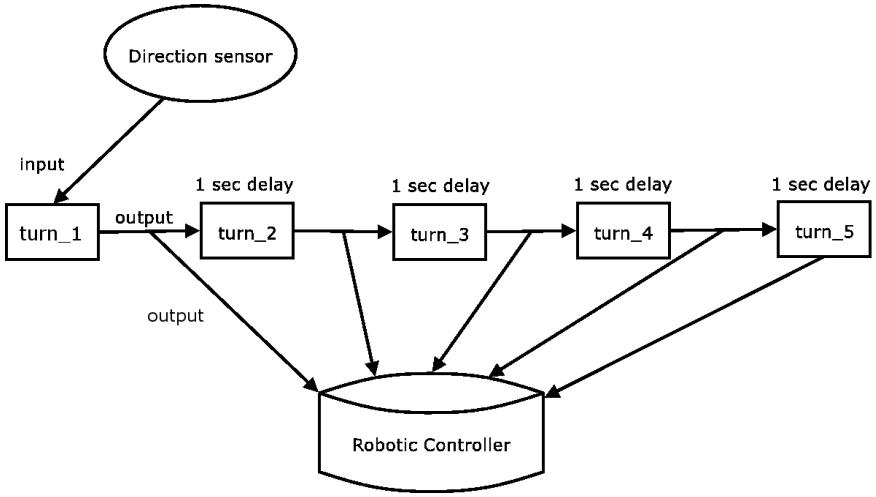

Figure 3.12: Logical layout of the 5 direction sensors. The first sensor detects how much the robot has turned every second and outputs the result to the controller and to the sensor next in line. The remaining sensors reads the output from their predecessors and outputs the value with a 1 second latency. In essence creating a memory of previous sensor readings.

An example result is shown in figure 3.13 where the direction sensors $turn_1$ to $turn_5$ at the end of the run will produce the output 0, -90, 90, 45 and 90 respectively.

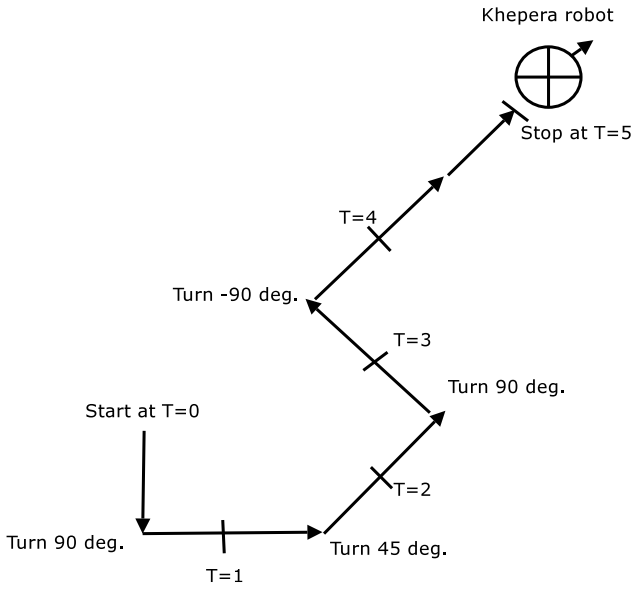

Figure 3.13: Movement pattern for the robot which would generate the direction sensor output 0, -90, 90, 45, 90.

The training run is made in an empty square area for 40 seconds with a cycle time of 0.1 seconds resulting in 400 data samples. This is a fairly low number of samples but the behavior is judged to be simple, even though the preprocessing of sensordata is considerable. The actions allowed for the Khepera are the same as in previous tests:

- -3: turn left sharply
- -2: turn left moderately
- -1: turn left slightly
- 0: don't turn (move straight ahead)
- 1: turn right slightly
- 2: turn right moderately
- 3: turn right sharply

The data logged in this run is the output from the previously described five direction sensors.

The Kheperas' resulting movement path for the training run can be seen in figure 3.14

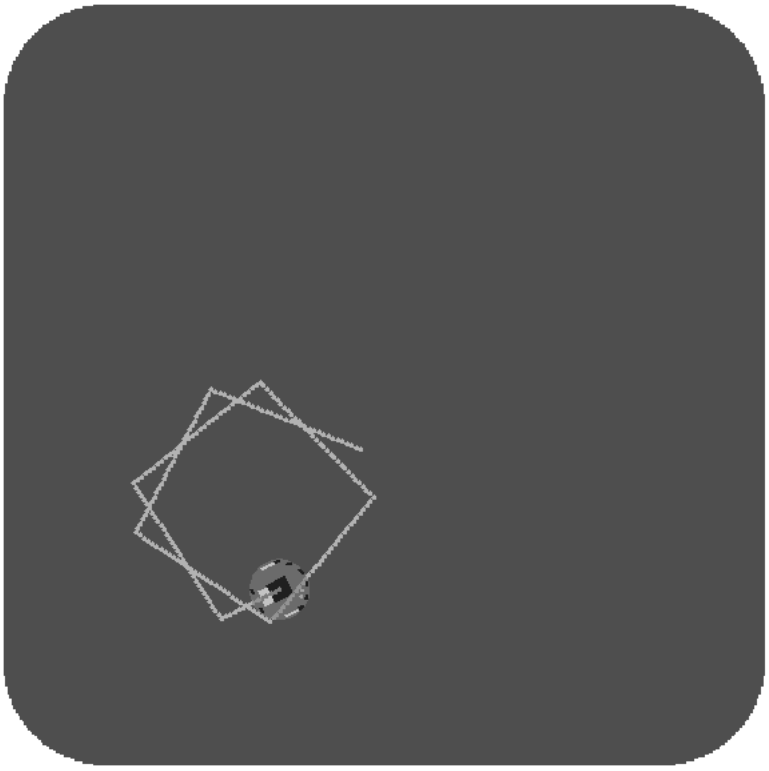

Figure 3.14: Movement path for the training run with "move in square" behavior

It is obvious from figure 3.14 that the path does not perfectly match up to a square. This is because the direction sensor has quite a bit of error margin in it. Due to limitation in the precision of the robots movements and also to the fact that the robot is always slightly slipping and sliding, those errors accumulate quickly.

Rule controlled run

A total of 21 rules are found with no special filtering of the data mining result. All rules are shown in table 3.5. Visual inspection seems to indicate some

No.	LHS	RHS	Strength	Support
1	rot5>=75	action=3	1.000	0.187
2	rot1<75 & rot2<75 & rot3<75 & rot4<75	action=3	1.000	0.210
3	rot3>=75	action=0	1.000	0.187
4	rot4>=75	action=0	1.000	0.187
5	rot2>=75	action=0	1.000	0.203
6	rot1>=75	action=0	1.000	0.213
7	rot5<75	action=0	0.971	0.790
8	rot1<75 & rot2<75 & rot4<75	action=3	0.529	0.210
9	rot1<75 & rot2<75 & rot3<75	action=3	0.529	0.210
10	rot1<75 & rot3<75 & rot4<75	action=3	0.508	0.210
11	rot2<75 & rot3<75 & rot4<75	action=3	0.496	0.210
12	rot1<75 & rot2<75	action=3	0.360	0.210
13	rot1<75 & rot4<75	action=3	0.350	0.210
14	rot1<75 & rot3<75	action=3	0.350	0.210
15	rot2<75 & rot4<75	action=3	0.344	0.210
16	rot2<75 & rot3<75	action=3	0.344	0.210
17	rot3<75 & rot4<75	action=3	0.335	0.210
18	rot1<75	action=3	0.267	0.210
19	rot2<75	action=3	0.264	0.210
20	rot4<75	action=3	0.258	0.210
21	rot3<75	action=3	0.258	0.210

Table 3.5: The entire 21-rule set found by searching the data from the move-in-square training run. The variables *rot1* to *rot5* contain the data from the turn sensors 1-5.

redundancy in the rule base e.g. rule 8 is a more general subset of rule 2. Tests show that only the first 6 rules that have a strength of 1.0 are needed to create a good controller. In case of more than 1 rule firing a majority vote is made to select which action should be performed. The result is excellent and the behavior is perfectly reproduced. Figure 3.15 shows the resulting path for the Khepera using these 6 rules in the controller.

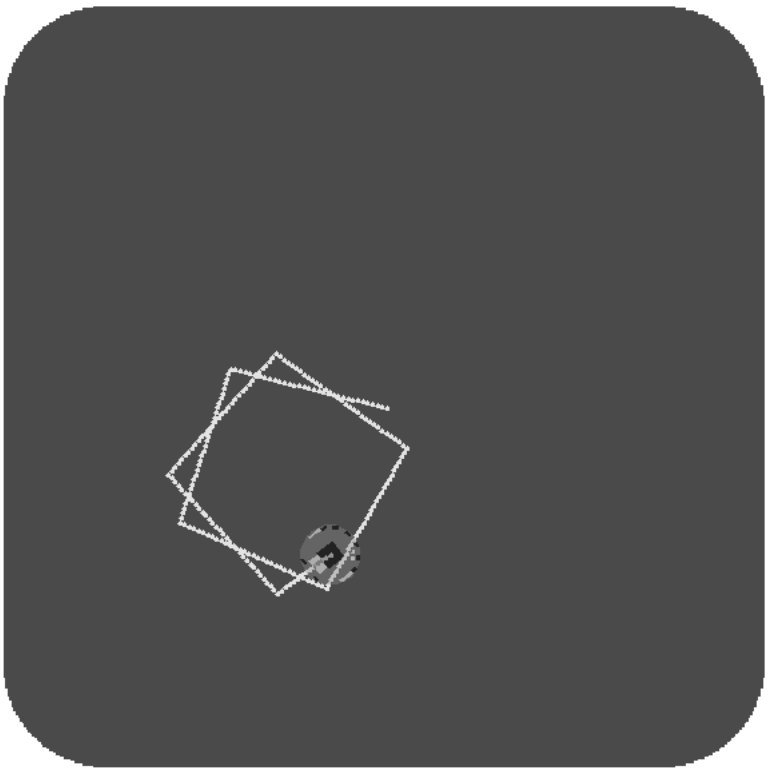

Figure 3.15: Movement path for the Khepera run with the 6 rules having a strength of 1.0 as a controller.

Even though this behavior is impossible to create without preprocessing of sensor data, it is still a fairly simple behavior (as made obvious from the fact that only 6 association rules where needed to reproduce it). The fact that this behavior is not affected by any proximity sensor input i.e. it never reacts to the environment but rather moves around with "eyes closed" only relying on its proprioceptive sensors, makes it less interesting but indicates that the "cascading" output of the five direction sensors is useful.

3.1.13 Slalom

Is it possible to create a really complex behavior using only association rules? A candidate complex behavior is "slalom". Slalom requires the robot to move in a straight line across an obstacle course while moving alternately to the left and right side of obstacles it encounters. Since the Kheperas' sensors have very short range (approx. 5 centimeters) they are not able to detect the next obstacle ("gate") until they are close to it. This combined with the requirement that the robot pass on different sides of gates without losing its orientation makes for a difficult task.

The sub-behaviors making up this task are: *move in the direction originally facing when starting* (to find the gates), *follow left wall* and *follow right wall* (when moving around the gates). These are fewer than the ones making up the cockroach controller but the problem of switching between them is more

complex. The robot needs to be able to break off the *follow wall* behavior when it has moved around a gate at the exact right time or else it will miss the next gate. Also the behavior must switch between left and right wall following, but unlike the cockroach test, no convenient single sensor like the light sensor exists to control this switching.

The training controller successfully creates a slalom behavior using the 5 proximity sensors, a direction sensor (using odometry) and two sensors indicating the time passed since the robot last faced left and right respectively. These two last sensors trigger when the robot faces the appropriate direction and then decays over time (100 cycles in the tests). The sensors also block each others output meaning that only the the sensor with the currently "highest" output is allowed to output it to the controller, the other sensors output being "drowned out" and outputting a 0 to the controller. One last sensor measuring the distance the robot has traveled since last facing either left or right is used to help the robot know when to break of its obstacle-following behavior. Figure 3.16 shows the path the robot follows using the training (non-association rule based) controller in a short test run.

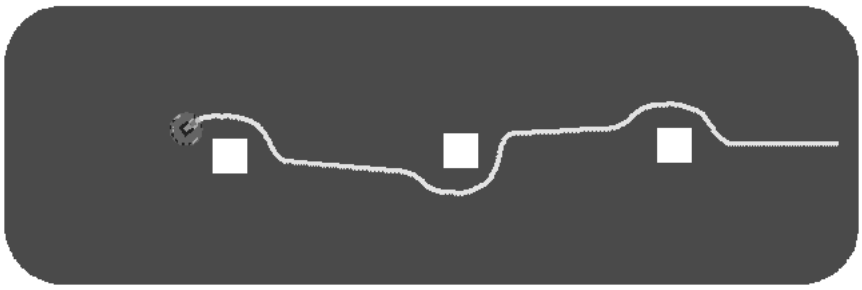

Figure 3.16: The robots movements when using the training controller.

A 1000 seconds training run is made with a cycle time of 0.1 seconds resulting in 10000 data samples for which the data mining produces 340 rules which are used as a rule base for a controller.

Unfortunately no successful rule based controller is found. Many different rule sets are tried in combination with different methods of resolving rule conflicts, but none is deemed acceptable. The "best" controller created makes the wrong decision more than 60% of the time and more importantly behaves erratically. The attempt is a failure.

3.1.14 Creating a general method

Despite the failure of the "slalom" behavior, the previous results with creating behaviors of varying complexity are fairly promising. However they all require extensive customization of the sensors and relatively large amounts of human supervision when it comes to collecting the training data. The tests have shown that when a temporal component exists in the underlying data generation process, a typical association rule mining algorithm can find it. Intuitively this makes sense since the "temporal" variables are just treated as ordinary data by the algorithm.

Now the goal is to create a more general technique for finding this type of rules, while not having to manually design and customize preprocessing sensors specifically tailored for a certain behavior. The idea is to log all data from the sensors available with no preprocessing or advanced capabilities in the sensors, and still be able to find temporal patterns in the data. We are not interested in rewriting the existing data mining algorithms and software, so the only option is to restructure the data set with the goal of introducing temporal patterns.

The straightforward way of "flattening" the data will be used. Flattening means that a depth d corresponding to an amount of time is selected and then the d data samples ("records") are merged to form a new record. Each original record contains one sample of every sensor during one time step in training run. Figure 3.17 illustrates this.

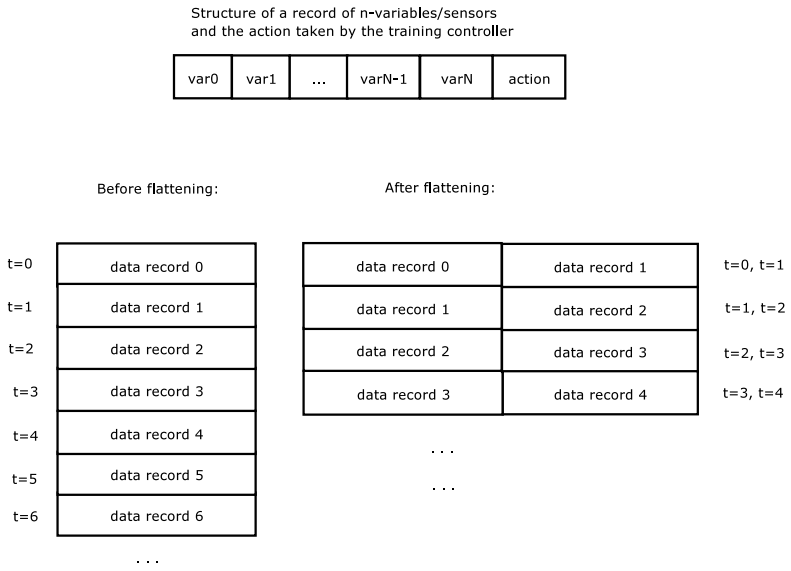

Figure 3.17: Flattening means that several data records are merged into one. The figure shows the result when flattening with a depth d of 2.

This process requires the variables used for identifying the data to be renamed to avoid confusion, but this is easy to automate.

In order for the controller to be able to trigger a behavior that depends on sensor readings n time steps ago, the robot must keep a history of its n latest sensor readings and actions so they can be compared to the rules in the rule base. It can be argued that this is a step away from a purely reactive behavior since it requires the robot to keep a memory of the latest sensor readings and actions. The argument has merit but since the robot does no planning whatsoever, keeps no abstract representation of the world, and does no special processing of its sensordata, it can still be said to be reactive with the special condition that the reactions depend on past stimuli. This means there are no complexities in the controller, hidden or otherwise, and it can be kept simple and fast which is the idea.

3.1.15 Problems

The main problem with this approach is that the dimensionality of the data increases greatly. If 1 record contains samples from 10 sensors, 1 record flattened with a depth of 5 will contain data from 50 sensors and the data mining software will by necessity treat each value as a unique variable. This combined with the problem of setting too high a limit for support mentioned in section 3.1.3 is clearly an obstacle since it ensures that the number of candidate patterns will be huge. This is both a problem performance wise with increasing computation time required to run the data mining algorithms, and also a problem in regards to if the rule base consists of possibly thousands of rules, which ones are interesting and how should they be ranked when many fire simultaneously.

3.1.16 Filtering and finding interesting rules

The "depth" selected when flattening the data set indicates how "far back" events can happen that influence the rule in question. E.g. a depth of 5 means that a rule can be based on sensor data 5 time steps (loop cycles) ago. Since a robot must be able to react fairly quickly to events, the sensors temporal resolutions need to be high i.e. the loop cycles need to be short. Typically 0.1 second is adequate. A temporal resolution of 0.1 and a flattening depth of 5 means the rules will only be able to take into account data that is 0.5 seconds old which is clearly not very much. So the time depth d needs to be fairly high and/or the temporal resolution low.

Another problem is that if the time between the discrete time steps in the data set is for example 0.1 seconds, it is safe to assume that for the simple behaviors that are the goal in this study, a difference of a few time steps is irrelevant. E.g.: If a rule R1 says:

$$R1 : x_1 = 2 \wedge x_{10} = 3 \rightarrow y = -1$$

and rule R2 says:

$$R2 : x_1 = 2 \wedge x_9 = 3 \rightarrow y = -1$$

where the the variable subscript indicates during which time step the data in the variable was current, the difference is only the 0.1 second between the x_{10} and x_9 terms. These rules should almost certainly trigger the same behavior. As will be seen, due to the very large number of variables in each data record, tens of thousands of association rules are found by Magnum Opus, even for and very simple behavior. The relatively high temporal resolution used (0.2 seconds) suggest that many rules might be very similar with only minor differences in their temporal components.

The proposed technique is to simply ignore the exact temporal component and only look at the *temporal ordering* of the LHS rule components. In other words, the exact point in time a value of a variable should be set is ignored and only the relative ordering (i.e. which appear first, which appear second, etc.) among them is looked at. Example:

If a rule R3 says:

$$R3 : x_1 = 2 \wedge x_7 = 0 \wedge x_{10} = 4 \rightarrow y = -3 \tag{3.1}$$

and the temporal index is removed from it the resulting rule is:

if x is first 2 then 0 then 4 THEN y should be -3

Which means that the two rules R1 and R2 from the first example would both equal:

if x is first 2 and then 3 THEN y should be -1

reducing the number of rules in the rule base.

3.1.17 Road-Sign Revisited with the Generalized Approach

To test the idea, the simple road sign problem is looked at again. The training run generates 1000 samples and the values logged are 5 forward and side-looking proximity sensors, one light sensor, and the action performed by the Khepera. This means 7 values in each record. The depth limit for the flattening of the data is set to 10, which together with a loop time of 0.2 seconds means that each final flattened record used in the data mining contains 70 variables. The "time" covered by one flattened record equals $0.2 * 10 = 2.0$ seconds which is not much but enough to discern time triggered behaviors.

Generating rules

The training run is done with a cycle time of 0.2 seconds and is run for 200 seconds resulting in 1000 samples. The "arena" the Khepera is run in looks similar to the arena used for the hand-customized controller (figure 3.18). Since

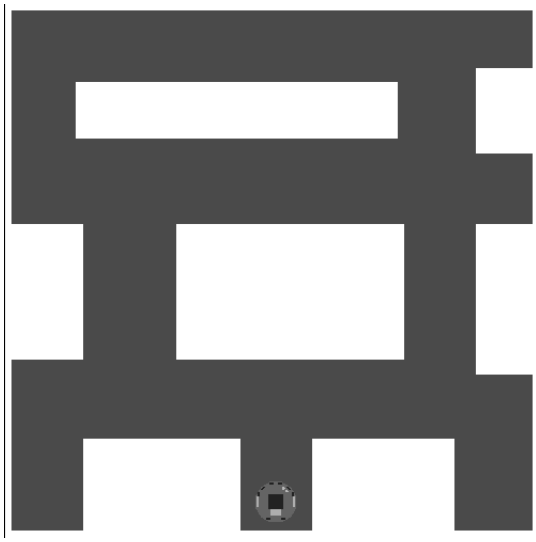

Figure 3.18: The layout of the arena used for the road-sign controller

the dimensionality of the data is very high after the flattening process (70, see above), there is a strong possibility of Magnum Opus finding a huge number of rules. It is not known beforehand what limits can be used to filter the rules in an effective way. The first tests with the road-sign controller showed that filtering the rules by *coverage* reduced the rulebase from 52 to 14 rules, so this is attempted here as well.

Since the original road-sign experiment gave a useful rule base with a limit of 0.3 for coverage, this limit is used in the first data mining run (a fairly stringent limit with the possibility of missing important but less frequent events). With the software set to ignore rules it finds to be "insignificant", a grand total of 9319 rules are produced. This is far too many to be useful so by applying the filtering technique outlined in section 3.1.16 the rule base is reduced to some 143 different rules. This is still quite a large set, especially considering that only 14 rules were needed to reproduce the original behavior when a customized controller and sensors where used.

Unfortunately a manual examination of the rule base reveals that every single rule has a RHS of $action = 0$ which means that all the rules tell the robot to move straight ahead. Clearly these rules are unusable, and the reason is most probably the high limit for coverage selected when starting the data mining process, since the robot spends most of its time moving straight ahead during the training run, only the rules with the "move straight ahead" action on their RHS has a high enough frequency to qualify.

A new data mining attempt is made, this time the limit for minimum coverage is set to 0.1, which is still quite high but hopefully generous enough to let more rules qualify as interesting. The resulting rule set found contains no less than 20678 rules, and the filtering process brings this down to 751. A manual examination of the rule set shows that at least some rules have different values on their RHS, but the "best" (highest *strength*) rule with a different RHS than $action = 0$ has a relatively low strength which results in that around 500 rules with higher strength exists, and they all trigger the action of moving straight ahead. It seems that the limit of 0.1 for support is still to high.

A final data mining run is made with 0.03 as a lower limit for support. 45837 rules are found and the "flattening" process reduces this to 844 rules. Visual inspection of these rules shows some promise. Many different RHS's of varying strengths exists.

Since the rule base is large the method of resolving multiple rules firing is very important. The same methods listed in section 3.1.8 are used and the results are in table 3.6. No method is subjectively judged to give a noticeably better result so in order to give som reference the actual number of erroneous actions taken are listed. The test runs fail to reproduce the original behavior. The first

Method	**Error**
Highest Strength	31%
Majority vote	83%
Average action	44%
Majority vote (Strength weighted)	65%
Average Action (Strength weighted)	44%
Majority vote (Lift weighted)	57%
Average action (Lift weighted)	51%
Highest Lift	71%

Table 3.6: Test run results of the general controller using a filtered rule base of 844 rules to reproduce the road-sign-follower behavior. Error rates for different ways of resolving multiple rules firing simultaneously are listed.

method (highest strength) seem by the numbers to perform better than the other methods, but subjectively no difference is noticed. The robots behavior is

quite unreliable and it has a tendency to bump into walls, move in circles, and turn unpredictably.

Once again it is seen that these numbers are not a very good indicator of performance, even a 40+ % error can give a very good behavior (as seen in the road-sign test) if the errors are of the type "turning a little instead of a lot". Unfortunately this is not the case here but rather the robot performs quite erratically.

It seems the rules produced by the rule-cleaning are to "slack". Removing the time index from the variables and only looking at the ordering of events results in many rules firing (typically about 20% of the rules in the rulebase fires during any given cycle). The filtering method appears to be too coarse and remove to much information to be useful.

Chapter 4

Conclusions

4.1 Results

The idea of using association rules to build robotic controllers seem like a quite powerful method for creating reactive behaviors like the road sign follower or moving in a set pattern like a square. The technique allows operators with no programming skills to create controllers e.g. by simply remote controlling the robot in a training run and then mining the data using commercial software. However the method at the moment probably does not speed up the process of building the controller much since though reducing the need to manually code the controller, the approach creates a new need to manually supervise the mining of data and selection of rules. Further the lack of a standardized method for selecting rules to comprise a rule base is a problem.

By moving some intelligence away from the decision making part of the controller into the sensors clearly more complex behaviors, like a cockroachs flight from light, can be created while still keeping the controller relatively simple. However this advantage is not unique for the association rule approach but can be applied to any technique for creating robotic controllers.

More complex behaviors like slalom seem hard to reproduce using association rules. Such a behavior appears to produce a too large rule base and even though the needed rules seem to be present, it is difficult to find a reliable method of selecting those rules. Hand-selection by a human operator is certainly a possibility but almost guaranteed to be more work than simply manually coding the controller from scratch.

The prospect of decreasing the need for customized sensors and sensor pre-processing by flattening the "regular" data from the unprocessed sensors seems dim. The result from trying to reproduce the simple road-sign follower was completely unsuccessful. Also, if successful, the robots "memory" would only have reached back in time for a very short period (2 seconds in this study), and longer memory exponentially complicates the problem of finding the association rules. The idea of flattening ordinary data is clearly inadequate and creates too high dimensionality in the data resulting in enormous rule sets.

The results found in this study indicate that a rules strength is often *but not always* a good indicator of its quality. For example in the test of the road-

sign problem a rule with a strength as low as 0.109 was needed, while in the move-in-square test only rules with a strength of 1.0 were needed.

The road-sign test also shows it is possible to reduce a working rule base in size significantly by removing redundant rules using a good filtering technique (from 52 down to 14 in the test). Unfortunately no universal filtering technique is found, rather different filters seem to be needed for different data sets.

Using association rules in controllers is a promising technique. At the moment it can reliably create simple reactive behaviors, and with sensor preprocessing more complex behaviors can be created. Really complex behaviors is still difficult to create but if better methods for measuring rule quality and filtering rule bases can be developed the possible applications look promising.

4.2 Further Studies

More research is certainly possible in this area. The problem of filtering out redundant rules from the huge rule sets generated deserves some attention. Using other algorithms than OPUS might also help reduce the number of rules. A reliable way of ranking multiple rules using a combination of strength, support, lift, leverage, etc. would also be very useful.

Another area to explore is the possibility of using *classification rules* i.e. global models, for building the rule bases instead of association rules representing local patterns. Tools for finding global time-dependent models exists, for example the software *timesleuth*[5] could be used.

4.3 Acknowledgments

I would like to thank my supervisor Thomas Hellström, for his support and helpful ideas (and patience =).

Bibliography

[1] T. Hellström *Association Rules for Learning Behavioral Mappings in Robotics*,
UMINF 03.12, ISSN-0348-0542. Dept. of Computing Science, UmeåUniversity, 2003.

[2] T. Hellström *Teaching a Robot to Behave like a Cockroach*,
Dept. of Computing Science, UmeåUniversity, 2004.

[3] J. Hipp, U. Güntzer, G. Nakhaeizadeh *Algorithms for Association Rule Mining - A General Comparison*,
SIGKDD Explorations, ACM SIGKDD, July 2000.

[4] R. Murphy *Introduction to AI Robotics.*
Bradford, MIT Press, 2000, ISBN 0-262-13383-0.

[5] K. Karimi, H. Hamilton Timesleuth: A Tool for Discovering Causal and Temporal Rules,
Proceedings of the 14th IEEE International Conference on Tools with Artificial Intelligence (ICTAI), 2002. University of Regina, Saskatchewan, Canada

[6] D. Hand, H. Mannila, P. Smyth *Principles of Data Mining.*
Bradford, MIT Press, 2001, ISBN 0-262-08290-X.

[7] Rulequest inc. *Magnum Opus Application Help*, Filter mode.

[8] T. Storm *Kiks Is a Khepera Simulator*
`www.tstorm.se/projects/kiks/` (2005-05-09)

[9] F. Tong *Primary visual cortex and visual awareness.*
Nature Reviews, Neuroscience Vol 4, March 2003