TECHNICAL UNIVERSITY OF CRETE

Heuristic Rule Induction for Decision Making in Deterministic Domains

by

Stavros Korokithakis

Thesis committee:
Michail G. Lagoudakis, Supervisor
Minos Garofalakis
Nikolaos Vlassis (Department of Production Engineering and Management)

A thesis submitted in partial fulfillment
for the Diploma degree
in the
Department of Electronic and Computer Engineering

July 30, 2009

ΠΟΛΥΤΕΧΝΕΙΟ ΚΡΗΤΗΣ

Ευριστική Επαγωγή Κανόνων για Λήψη Αποφάσεων σε Αιτιοκρατικά Πεδία

Σταύρος Κοροκυθάκης

Τμήμα Ηλεκτρονικών Μηχανικών και Μηχανικών Υπολογιστών

Abstract

A large corpus of work in machine learning and decision making focuses on planning and learning in arbitrarily stochastic domains. However, these methods require significant computational resources (large transition models, huge amounts of samples) and the resulting representations can hardly be broken into easily understood parts, even for deterministic or near-deterministic domains. This thesis focuses on a rule induction method for (near-)deterministic domains, so that an unknown world can be described by a set of short rules with well-defined preconditions and effects given a brief interaction with the environment. The extracted rules can then be used by the agent for decision making. We have selected a multiplayer online game based on the SMAUG MUD server as a model of a near-deterministic domain and used our approach to infer rules about the world, generalising from a few examples. The agent starts with zero knowledge about the world and tries to explain it by generating hypotheses, refining them as they are refuted. The end result is a set of a few meaningful rules that accurately describe thez world. A simple planner using these rules was able to perform near optimally in a fight scenario. The proposed method is general and applicable to other domains with a minimal set of changes.

Περίληψη

Ένας μεγάλος όγκος δουλειάς στη μηχανική μάθηση και τη λήψη αποφάσεων επικεντρώνεται στο σχεδιασμό και τη μάθηση σε αυθαίρετα στοχαστικά πεδία. Ωστόσο, αυτές οι μέθοδοι απαιτούν σημαντικούς υπολογιστικούς πόρους (μεγάλα μοντέλα μετάβασης, τεράστιους αριθμούς δειγμάτων) και οι αναπαραστάσεις που προκύπτουν δε μπορούν εύκολα να διασπαστούν σε ευνόητα τμήματα, ακόμα και για αιτιοκρατικά ή σχεδόν αιτιοκρατικά πεδία. Αυτή η διπλωματική εργασία εστιάζει σε μία μέθοδο επαγωγής κανόνων για (σχεδόν) αιτιοκρατικά πεδία, ώστε ένας άγνωστος κόσμος να μπορεί να περιγραφεί από ένα σύνολο μικρών κανόνων με καλά καθορισμένες προϋποθέσεις και επιδράσεις, δεδομένης μιας βραχείας αλληλεπίδρασης με το περιβάλλον. Οι κανόνες που εξάγονται μπορούν να χρησιμοποιηθούν από τον πράκτορα για τη λήψη αποφάσεων. Επιλέξαμε ένα διαδικτυακό παιχνίδι πολλών παικτών (multiplayer online game) βασισμένο στον SMAUG MUD server ως ένα μοντέλο ενός σχεδόν αιτιοκρατικού πεδίου και χρησιμοποιήσαμε την προσέγγισή μας για να εξάγουμε κανόνες για τον κόσμο, γενικεύοντας από μερικά παραδείγματα. Ο πράκτορας ξεκινά με μηδενική γνώση για τον κόσμο και προσπαθεί να τον εξηγήσει παράγοντας υποθέσεις, εκλεπτύζοντας τις όπως διαψεύδονται. Το τελικό αποτέλεσμα είναι ένα σύνολο από λίγους, περιεκτικούς κανόνες οι οποίοι περιγράφουν με ακρίβεια τον κόσμο. Ένας απλός planner που χρησιμοποιεί αυτούς τους κανόνες επιδεικνύει σχεδόν βέλτιστη απόδοση σε ένα σενάριο μάχης. Η προτεινόμενη μέθοδος είναι γενική και εφαρμόζεται σε άλλα πεδία με ελάχιστες αλλαγές.

Acknowledgements

I would like to thank the following people:
My parents and grandparents, for always supporting and helping me.
My supervisor, for his invaluable guidance, help and open mind.
My friends, who always stood by me.

Contents

List of Figures

Chapter 1

Introduction

From the moment it is born, every (sufficiently evolved) organism begins to learn things about the world it lives in. The mechanics of cause and effect soon become apparent to it, and it begins to discover the ways in which it can interact with the world by experimenting.

Intelligent agents that can learn through reinforcement learning have been the focus of research for many years. Reinforcement learning focuses on mapping each state of the world to the best action choice in that state [Kaelbling et al., 1996]. The agent explores the world, observing the actions it took at each state and their immediate effects, and eventually learns the action choices that yield the desired long-term outcome.

This means that, to use reinforcement learning in the real world, the agent would have to observe more or less every state and perform every action available to it in that state, in order to see how they contribute to the desired outcome. With the complexity of the real world, this approach quickly becomes intractable.

Induction learning, on the other hand, aims to train an agent on a few known inputs and outputs, so that it will create rules, which can then be applied to previously unseen inputs to produce meaningful predictions as to the outputs that will be observed [Wexler, 1996].

1.1 Multiplayer Online Games

It quickly became clear to us that, if we wanted our research to be applicable to the real world, we would need a moderately complex model of it. Fortunately, such models have existed for a few decades in the form of multiplayer online games (Multi-User Dungeons,

MUDs). MUDs are text-based (this is an advantage because the world variables are easier to parse, but our research can easily be extended to other models) and they offer complex and detailed worlds in which one can perform almost any task one would in the real world.

The overarching task in these games is to fight enemies and accrue gold, items and experience, which you can use to unlock new abilities. Input in these games takes the form of **commands**, which a player enters to perform actions in the world. Each command effects a change in the state of the world, and this change is reflected in the state variables that the user can observe.

For example, there are, among others, a **fight** variable (boolean) that indicates whether or not the player is currently fighting an enemy, and a **health** variable (integer) that reflects the player's current health. If **health** reaches zero, the player dies.

The player can influence the value of these variables by performing the associated commands, namely **strike** <**opponent**> causes the player to engage in a fight (the **fight** variable becomes **True**) and the enemy to take some amount of damage. Conversely, this command is usually entered by the enemy as well, so a player that is currently in a fight can expect to take an amount of damage, regardless of his actions.

The fighting player's **health** will diminish until it reaches 0, at which point the player will die. Obviously, this scenario is something to be avoided by the player. The player can increase his health by the **heal** command, which is subject to constraints in itself, and so on...

1.2 Rule Inference in the MUD

Our ultimate goal is for an agent to be given a list of desired outcomes and nothing else, and for the agent to be able to divine enough aspects of the world to achieve these outcomes. We must, therefore, find a way for the agent to understand the effect that its actions have on the world and on its current state.

A baby learns about the world by trial and error. It learns that touching a hot object will cause a burn, and that eating food will take away its feeling of hunger. In the same way, our agent must learn that **heal**ing will increase its health and **fight**ing will decrease it, but will eventually provide it with gold, if it wins.

Another primary design goal for us was for the agent to be able to generalise the rules it discovers and then refine the circumstances it believes they apply to, so that it goes from the general to the specific, instead of the other way around.

1.3 Thesis outline

Chapter 2 provides the necessary background for the ideas this thesis builds upon. A brief introduction is given to the Markov property, reinforcement learning, and induction learning.

Chapter 3 discusses the motivation behind this work. The advantages of induction learning over more traditional methods are explained, along with a summary of related work in the area. The strengths and weaknesses of existing approaches are discussed, along with usual practices.

Chapter 4 introduces our heuristic rule induction algorithm, a novel approach to inferring rules in noisy deterministic domains. Its efficiency is analysed and its properties are discussed.

Chapter 5 demonstrates the applicability of the algorithm on multiplayer online games and discusses its efficiency.

Finally, Chapter 6 discusses the strengths and weaknesses of the approach, gives a number of guiding directions for future work, and concludes this thesis with a brief summary.

Chapter 2

Background

2.1 Agents and environments

In building artificially intelligent systems, we need to create an analogue, something that will act upon its environment to (usually) effect some desired result. We call this analogue an **agent**.

An agent can be anything that can perceive its environment (or, more often, a useful subset of it) and act to achieve a goal. The environment is what the agent perceives as "the outside world". This can be a room, a computer simulation, or, as in our case, an online game. The environment is usually abstracted down to a sequence of **state variables**, which hold all the information of the environment that is relevant to the agent.

Environments can vary from the deterministic (each action has a specific outcome every time) to the stochastic (each action can have a variety of outcomes). Another factor we must take into account is **noise**, which may range from the non-existent to the debilitating. An appropriate learning algorithm must be chosen according to the type of environment at hand, in order to produce good agent behaviour.

2.2 Dynamic Bayesian networks

Bayesian networks are data structures for representing compactly large joint probability distributions over a number of random variables which are weakly dependent on each other. A Bayesian network is typically a directed acyclic graph in which the nodes represent random variables and the edges represent conditional dependencies [Ghahramani, 1998, Jordan, 1999]. For example, in the network shown in Figure 2.1, the value of node

C depends only on the values of its **parent** nodes, A and B. Thus, C is a **child** node of A and B. The probability of observing a given set of values a, b, c, d, e, f is given by the formula:

$$P(a, b, c, d, e, f) = P(c|a, b)P(d|c, f)P(e|c)P(a)P(b)P(f)$$

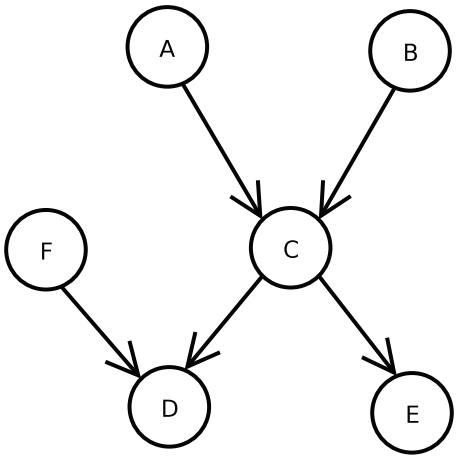

FIGURE 2.1: A simple Bayesian network

A **dynamic** Bayesian network is a network that can be used to build a probabilistic model of sequential processes. The well-known Hidden Markov Model is an example of a dynamic Bayesian network, and can, in fact, be considered as the simplest possible one. Other variants of the HMM (such as the factorial HMM in Figure 2.2) can be considered DBNs.

Consider a sequence of observations. Each "slice" of the sequence contains n random variables, and it can be thought of as an ordinary Bayesian network, which is duplicated for each slice. The sequential dependencies between variables are represented by edges between the duplicated slices.

Therefore, the DBN is defined by two components:

1. A Bayesian network that contains all the random variables and their dependencies within the slice. This Bayesian network will then be duplicated as needed on each slice in order to model the structure of the sequence.
2. A set of edges that represent the dependencies between two slices in the sequence and connect one to the other.

We must clarify here that the nodes and edges in and between each slice are repeated throughout the DBN. We can, therefore, specify the entire DBN by only specifying a starting and ending slice and the edges between them.

Additionally, each node is associated with a conditional probability table (CPT), which gives a probability distribution over the possible values of the variable corresponding to that node given the values of its parents. In unknown domains, the structure and CPTs of Bayesian networks or dynamic Bayesian networks can be learnt from data using a variety of methods.

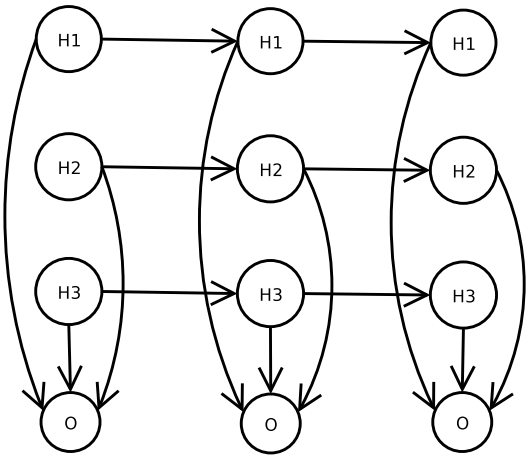

FIGURE 2.2: Three slices of a factorial HMM (a type of DBN)

2.3 Inductive learning

Inductive learning is the process of learning by example, and constructing general rules from a series of observed instances. The agent gathers data from the world and tries to explain the events that take place in it. It can then use the rules it has learnt from the environment to plan its future behaviour.

Consider a function $f(x)$. Consider then a set of training data $\{(x_i, f(x_i)) : i = 1, 2, 3, \ldots, N\}$, and our aim is to determine f by some adaptive algorithm. Inductive learning aims to construct a hypothesis h such that, for all i, $h(x_i) \approx f(x_i)$. If there are two hypotheses h_1 and h_2 with $h_1(x_i) \approx f(x_i)$ and $h_2(x_i) \approx f(x_i)$, then one of them may be chosen by some process called **bias**. The purpose of using a hypothesis in the first place is that the hypothesis may be applied to new inputs $\{x_j\}$ to attempt to discover what $f(x_j)$ would be without actually evaluating $f(x_j)$.

Training data may consist of positive and negative examples. Positive examples are known correct pairs and negative examples are known incorrect pairs. A good hypothesis must explain as many of the positive examples as possible while explaining none of the negative ones. It can then be generalised to extrapolate past the examples already given to new ones. This helps us predict the future and anticipate what will happen if we evaluate each $f(x_j)$.

2.4 Model-based learning

There are two mainstream approaches to agent behavioural learning, namely model-free and model-based learning. In model-free learning, the agent uses the data to learn a good decision policy directly, according to the performance criteria given. In model-based learning, on the other hand, the agent uses the data to learn a model of the underlying process and uses this model to compute a decision policy, again according to the performance criteria.

There are advantages and disadvantages to both approaches. Many domains can only be described by huge and complicated models, whereas good behaviour can be achieved by fairly simple policies, therefore model-free learning is more appropriate. In other domains, the underlying model is fairly simple and can be easily learnt from data, and therefore the derived policies will be much more accurate and better. In this case, model-based learning is a better fit.

Chapter 3

Problem statement

3.1 Models and the real world

Usually, our modeled worlds are subsets of the real world, and much simpler. An airplane guidance system typically only needs to concern itself with yaw, pitch, roll, and speed as its inputs, and aileron and throttle adjustments as its outputs. With some quantization, the possible values that these variables can take are reduced to a very manageable set. Typical learning techniques for control work very well in these circumstances. The system initially learns how to fly correctly (not crash) and eventually learns how to handle other circumstances that might arise by coming across them and noting how it did.

What happens, though, if the state space is so large that it is impractical to see even a small subset of the state space? How can a system develop an effective policy if it can't know what might happen in most cases? Exhaustively exploring the state space is infeasible (especially when one has to go through the same state multiple times to discover all the outcomes that might arise from a certain action one takes in that state).

Several approximation methods have been proposed as a way of coping with large state spaces. These methods work well when small changes in the state variables do not bring about large differences in the outcome. To continue the example above, the state space over yaw, pitch, roll and speed is amenable to approximation, since nearby states result in similar outcomes. In contrast, consider the state space of a chess game (all possible board configurations), where nearby states can result in radically different outcomes. In general, domains that include diverse and discrete state variables can hardly be approximated compactly.

Our particular model, the MUD, has about 20 state variables, each one of a different nature and with a different range (some are positional, some are integer ranges, some are boolean), and upwards of 50 actions someone could take, each of which influences the world differently, sometimes with subtle changes that are only observable in the long term and sometimes with immediate effects. Therefore, our case is more similar to the chess example presented above than to the airplane, and thus we cannot rely on function approximation for good results.

Clearly, to exhaustively explore this state space we would need to visit some 10^30 states, a vast number that is not explorable in a lifetime. At first glance, this would appear impossible. It clearly isn't, however, since we know for a fact that human players have no problem functioning in it every day, and even more so in the real world. How do they do it?

3.2 The human approach

As we already know, humans (and, indeed, most living organisms) don't need to explore the entire state space of the world (as this would be nigh impossible to do) in order to theorise about the results of their actions. A human may have never put their hand on a burning stove, but they have a very good idea of what would happen if they did, so they avoid it.

This is because humans are very good at generalising. Generalisation is the ability of inferring a general rule which is true always (or most of the time) from a few experiences. If, for example, something is not poisonous for a few people, it is likely that it is not poisonous for **all** people. Of course, there are modifiers to the result, such as immunity towards a poison, that cause the action not to have the intended outcome. Humans can effectively discover the variables that are vital to the desired outcome and ignore irrelevant ones, greatly simplifying the problem [Salzberg, 1985].

3.3 Problem formalisation

As previously mentioned, the state space S is described by a finite number of state variables $s_1, s_2, \ldots, s_n$. A tuple

$$S(t) = \Big(s_1(t), s_2(t), \ldots, s_n(t)\Big)$$

reflects the state of the world at time t. The agent has a set of actions $\mathcal{A} = \{a_1, a_2, \ldots, a_m\}$ that it can perform. By $a(t)$ we denote the action taken by the agent at time t. We assume that the world has the Markov property, which means that

$$P\Big(S(t+1) \;\Big|\; S(t), a(t), S(t-1), a(t-1), S(t-2), a(t-2), \ \ldots\Big) = P\Big(S(t+1) \;\Big|\; S(t), a(t)\Big)$$

The goal for the agent is to find a policy π for choosing an action $a(t) = \pi\Big(S(t)\Big)$ at each time step t that maximises some long-term reward given the current state $S(t)$. To do this, however, it is important to know how each action impacts the state. In other words, we need a transition model T:

$$S(t+1) = T\Big(S(t), a(t)\Big)$$

Notice that our choice for the transition model is both Markovian (in the state and the action) and deterministic (there is only one outcome for a given state and action pair). Since the transition model is unknown to us, we cannot form any sort of policy. We must, therefore, discover **what** changes each action imparts on the state in each case, and also **when** these occur, i.e. the conditions under which they occur. Our goal is to infer a transition model for the world that takes the form of a collection of rules. The general form of the rules is the following:

> If a particular action a is taken at time t, then a subset of the state variables $\{s_i, s_j, s_k, \ldots\}$ will change by $\{d_i, d_j, d_k, \ldots\}$ at time $t+1$, provided that the values of another subset of state variables $\{s_l, s_m, s_n, \ldots\}$ are $\{v_l, v_m, v_n, \ldots\}$.

Our approach is predicated on the assumption that the world is mostly deterministic (i.e. the same action will not bring radically different results if performed at two different times in the same state), but we must account for noise, as our measurements might not be completely accurate. This assumption is valid in our case, since the MUD is generally deterministic. Another assumption we make is that there are weak dependencies, i.e. an action affects a small number of state variables, and that the preconditions under which this happens involve only a small number of state variables as well.

3.4 Environment noise

There are various problems one might encounter in environments that complicate the process by introducing noise or otherwise change the measured values of the variables. In order to make our approach more general, we desire our transition model to be able

to accommodate small levels of noise. In particular, we assume the following kinds of noise:

Measurement noise: For every state variable change, noise N may have been introduced in the measurement, such that our observation of the state gives $S(t)+N(t)$ instead of the true $S(t)$. This is deleterious to the induction, as the agent will not know if the measured value is what its action effected or if it was merely a side-effect. We do, however, make the assumption that noise levels are low and that the noise is Markovian.

Post-quantized variables: In some cases, the value that the agent observes is not of the same accuracy as in the internal state of the world. For instance, for some variable v, the observed value o might be quantized as follows:

$$o = \left\lfloor \frac{v}{10} \right\rfloor, v \in [0, 100]$$

This can lead to the agent perceiving no change to the state for a long period of time for an action, and then suddenly perceiving a large state change.

Since our problem formulation does not make any effort to account for a stochastic model, any stochastic behaviour that cannot be accounted for in the two previous cases, cannot be accommodated. In such cases, a truly probabilistic model must be used.

3.5 The MUD domain

Our test domain is a subset of the full MUD domain, and it focuses on combat situations. We have selected the following state variables to include all the necessary state information for such uses:

Health is an integer in the range of approximately $[0 - 1000]$. It reflects the player's well-being, and it is generally desired that its value is as high as possible. If a player's **health** falls to zero, the player dies, to his great inconvenience.

Mana is an integer in the range of approximately $[0-700]$, and it reflects the amount of available "magic power" the player has for casting spells. This variable also needs to be kept from reaching zero, as then the player cannot cast any more spells, which are generally useful to the player.

Fighting is a Boolean variable that indicates whether the player is currently in combat or not. If the player is in combat, he will usually take (and deal) damage.

Enemy health is an integer variable in the range of [0-11], which indicates the current health of the enemy. This is analogous to our own health, and causes the enemy to die if it reaches zero. One of our goals is to do damage to the opponent, lowering its health.

In addition to the above state variables, we have used one action variable which can take four values, corresponding to the four actions that the agent can perform. These actions are the most relevant to combat (it would not make sense to include trading commands while in a fight), and they have the most diverse results (there are other actions that have the same effects as the ones we have chosen, only varying in degree). The actions we have chosen are:

Pause is a control action that does nothing. We include it so that we can see what happens in the world without our intervention. For example, while in a fight, we can see that we take damage regardless of whether we do anything or not. We can then remove the changes that would happen anyway from our other actions and arrive to a much more accurate conclusion.

Strike opponent is an aggressive action that causes the opponent to take damage. Additionally, if we are not in a fight, the angry opponent starts one, and **fighting** changes to True.

Heal causes our **health** and **mana** to increase by certain amounts. Obviously, if these variables have already reached their maxima, this action has no effect.

Cast spell is another aggressive action that causes the opponent to take damage. This action also causes us to enter a fight if we are not already in one, and it also causes **mana** to diminish by some amount.

3.6 Related work

Inductive logic programming (ILP) [Muggleton, 1992] is a fairly new active research area which combines principles of inductive machine learning with the representation of logic programming, with the goal of automatically learning logic programs from examples. ILP has been successful in generating new scientific knowledge in various domains involving mostly Boolean variables but has not produced significant results in domains containing numerical data.

Salzberg [Salzberg, 1985] developed HANDICAPPER, a system for predicting horse race outcomes. HANDICAPPER uses various heuristic inductive learning methods to explain

and predict races, faring markedly better than human experts and chance. This paper describes various heuristics at a high level, providing inspiration for our work, however it was not clear how to combine them at an algorithmic level and apply them to our domain.

Amir and Chang [Amir and Chang, 2008] developed an exact solution for identifying actions' effects in partially observable STRIPS domains. Their methods apply in other deterministic domains with conditional effects but may be inexact, as they produce false positives, or inefficient, as the resulting model can grow arbitrarily).

Diuk et al. [Diuk et al., 2006] developed an algorithm to improve the speed of hierarchical reinforcement learning while preserving state abstraction and to use hierarchies to augment existing factored exploration algorithms to achieve low complexity for both learning and planning in deterministic domains. This work is geared towards producing good agent behaviour and does not produce knowledge at the level of rules we would like.

Boutilier et al. [Boutilier et al., 1999] offer an extensive survey on decision-theoretic planning, covering several methods and techniques for deriving and learning optimal (or near-optimal) agent behaviours. The methods covered in this survey focus mostly on planning aspects ranging from deterministic to stochastic environments but do not cover the problem of model learning, which is our focus.

Chapter 4

The proposed approach

4.1 Our approaches

Our model, being a sequential process with dependent variables, looks like a good fit for a dynamic Bayesian network. One can include all state and action variables in a DBN, determine their dependencies from the data, and then discover the effects of each action. This is what we initially attempted, but the results, as we will show, were less than satisfactory, since it was very difficult to extract the kind of knowledge we wanted from the resulting network.

To better address our needs, we decided to develop our own heuristic rule induction algorithm. This algorithm was designed to emulate the human way of reasoning in such domains. Both approaches are presented herein, with their advantages and disadvantages (though, naturally, more weight is given to our approach).

4.2 Using dynamic Bayesian networks

Initially, we used a dynamic Bayesian network to discover the causal effect of the actions to the state. Experimentation utilised the open-source Mocapy toolkit [Hamelryck, 2009] for inference and learning in DBNs with Python. This choice was dictated by the fact that it was easy to integrate with the rest of our code. Mocapy provides functionality for learning the conditional probability tables for the nodes of a fixed network structure, but, unfortunately, it does not implement structure learning.

The nodes in each time slice of the dynamic Bayesian network include all the state variables and the single action variable, as described in Section 3.5, at the corresponding time step.

Before learning the CPTs we need to establish a structure of dependencies. In the absence of domain knowledge and the lack of a structure learning process in Mocapy we followed a brute-force approach.

In particular, we set each state variable to be conditionally dependent upon all state variables and the action variable in the previous time step. This is so we can discover the dependent variables after we learn the conditional probability tables of the network. Therefore, the initial structure given to Mocapy for learning the CPTs is as shown in Figure 4.1.

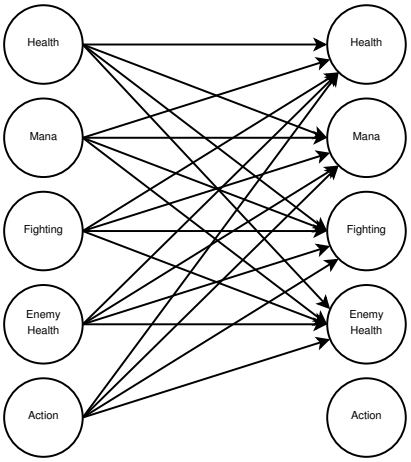

FIGURE 4.1: Our dynamic Bayesian network structure

We have not set the action to depend on any of the variables because there are no conditional probabilities on the action, as the action is chosen by us regardless of the preconditions at this stage (our action choice may, of course, depend on the state, but we do not need to concern ourselves with this at this point).

Our next step is to collect a set of data which will be used by Mocapy to learn the CPTs from the data using the expectation maximisation (EM) algorithm [Nielsen, 2000]. This data was collected by executing a purely random policy. Mocapy returns one CPT for each node in the network, describing the probability distribution over the values of a state variable s_i given the values of the state and action variables in the previous time step:

$$P\Big(s_i(t) \;\Big|\; s_1(t-1), s_2(t-1), s_3(t-1), s_4(t-1), a(t-1)\Big)$$

In order to infer the dependencies between variables and actions, we calculate the marginals M for a CPT, each time summing out the variables we want to ignore. For each calculated marginal we check to see if the variables have any correlation with the action or not. To do this, we compare the marginal of each action value to the corresponding marginal of a "control" action (**pause**) which we know is uncorrelated with

any of the variables. This "control" action also serves as a baseline that shows us what happens when we take no action at a given time slot. For example, when fighting, our health will slowly decrease as the opponent damages us. The control variable allows us to see that the health decrease will happen anyway and factor it out of the results.

This method did yield some results, but they were very state-dependent and vague. For example, if we marginalise out all other variables from the conditional probability table for the variable **health**, keeping as parents only **health**, **action** and **fighting** in the previous time step, we can see that the CPTs are different. This means that **health** is dependent on all of itself, **action** and **fighting** from the previous time step. In the examples below, we present the CPTs for health for specific values of **fighting** (True and False), and **action** ("pause" and "heal"):

$\mathbf{CPT(health_t|fighting_{t-1} = False, health_{t-1}, action = pause)} =$

$$\begin{bmatrix} 10 & 10 & 10 & 10 & 10 & 10 & 10 & 10 & 10 & 10 \\ 10 & 10 & 10 & 10 & 10 & 10 & 10 & 10 & 10 & 10 \\ 10 & 10 & 10 & 10 & 10 & 10 & 10 & 10 & 10 & 10 \\ 10 & 10 & 10 & 10 & 10 & 10 & 10 & 10 & 10 & 10 \\ 10 & 10 & 10 & 10 & 10 & 10 & 10 & 10 & 10 & 10 \\ 10 & 10 & 10 & 10 & 10 & 10 & 10 & 10 & 10 & 10 \\ 10 & 10 & 10 & 10 & 10 & 10 & 10 & 10 & 10 & 10 \\ 10 & 10 & 10 & 10 & 10 & 10 & 10 & 10 & 10 & 10 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 100 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 100 \end{bmatrix}$$

$\mathbf{CPT(health_t|fighting_{t-1} = True, health_{t-1}, action = pause)} =$

$$\begin{bmatrix} 10 & 10 & 10 & 10 & 10 & 10 & 10 & 10 & 10 & 10 \\ 10 & 10 & 10 & 10 & 10 & 10 & 10 & 10 & 10 & 10 \\ 10 & 10 & 10 & 10 & 10 & 10 & 10 & 10 & 10 & 10 \\ 0 & 0 & 0 & 100 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 100 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 6 & 94 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 7 & 93 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 5 & 95 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 13 & 87 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 21 & 79 \end{bmatrix}$$

$$\mathbf{CPT}(\mathbf{health_t}|\mathbf{fighting_{t-1}} = \mathbf{False}, \mathbf{health_{t-1}}, \mathbf{action} = \mathbf{heal}) =$$

$$\begin{bmatrix}
10 & 10 & 10 & 10 & 10 & 10 & 10 & 10 & 10 & 10 \\
10 & 10 & 10 & 10 & 10 & 10 & 10 & 10 & 10 & 10 \\
10 & 10 & 10 & 10 & 10 & 10 & 10 & 10 & 10 & 10 \\
10 & 10 & 10 & 10 & 10 & 10 & 10 & 10 & 10 & 10 \\
10 & 10 & 10 & 10 & 10 & 10 & 10 & 10 & 10 & 10 \\
10 & 10 & 10 & 10 & 10 & 10 & 10 & 10 & 10 & 10 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 100 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 100 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 100 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 100
\end{bmatrix}$$

$$\mathbf{CPT}(\mathbf{health_t}|\mathbf{fighting_{t-1}} = \mathbf{True}, \mathbf{health_{t-1}}, \mathbf{action} = \mathbf{heal}) =$$

$$\begin{bmatrix}
10 & 10 & 10 & 10 & 10 & 10 & 10 & 10 & 10 & 10 \\
10 & 10 & 10 & 10 & 10 & 10 & 10 & 10 & 10 & 10 \\
10 & 10 & 10 & 10 & 10 & 10 & 10 & 10 & 10 & 10 \\
0 & 0 & 0 & 0 & 0 & 100 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 100 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 17 & 83 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 11 & 89 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 30 & 70 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 12 & 88 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 23 & 77
\end{bmatrix}$$

For example, in the case where fighting is True and action is "pause", we can see that there is a much higher probability to end up in a lower health state than if we are not fighting. We can, therefore, infer that "pausing" when fighting has a deleterious effect to our health. We cannot ascertain, however, precisely **what** this effect is. On the other hand, we see that "healing" has a beneficial effect to our health regardless of whether we are fighting or not. Again, though, we cannot know in detail what this effect is, and one must try all the combinations of each variable in order to derive the dependencies between variables, which is why we decided that this approach was not suitable for us, and was geared more towards stochastic domains.

4.3 Using a heuristic inference algorithm

4.3.1 Rule induction

After the Bayesian network, we used another approach, namely implementing our own heuristic inference algorithm. This algorithm models more closely the way humans reason about the world by constructing rules that try to explain the data.

The first step we perform is to construct rules for each action in the world. We structure our algorithm in this way, because actions are the most essential method of interaction with the world. We want to infer the changes that an action brings about to the world, so it makes sense to start by grouping all the results by actions. Since there may be several outcomes for an action, we allow for multiple disjoint pairs of preconditions and effects for each action. Initially, the outcomes for each action are empty and they are updated as the data is processed.

The update step of the algorithm takes as input a state pair and an action. A state pair consists of a state, $S(t)$, and the one immediately preceding it, $S(t-1)$. The action $a(t-1)$ is the action that the agent took between those two states (and the action that ostensibly caused the state changes). The algorithm then notes the differences between the two states in a vector δ and checks if this particular outcome has been seen before. If it has been seen before, trust in it is reinforced. If it has not, it is added to the set of outcomes for the current action. The state variables are then added to the preconditions of each outcome, as positive examples of the current outcome (if this result happened when the state was $S(t)$) and negative examples of all the others (if this result **did not** happen when the state was $S(t)$).

As we mentioned above, the deltas contain the sets of outcomes. These differences can be of various **types** such as integer, Boolean, or unordered set. The type of the variable governs the relationships between variables of the same type. For example, for an integer variable the delta will be the actual numerical difference after subtraction. For an unordered set, on the other hand, the delta will be a 2-tuple of the previous and current state variable.

The algorithm shown in 4.2 creates the deltas for each action and a measure of confidence for each rule. This, however, creates duplicate rules most of the time, since it is unusual for the world to be completely deterministic and have each action only produce one outcome in any state.

```
update_action_rules (S(t−1), a(t−1), s(t))

    //  S(t−1) : The previous state
    //  a(t−1) : The action performed in the previous state
    //  S(t)   : The current state

    δ(t) ← S(t) − S(t−1)
    if (δ(t) ∈ δ_{a(t−1)})              // If the changes in the state are in the deltas set,
        reinforce(a(t−1), δ_t)          // reinforce the already existing delta.
    else
        δ_{a(t−1)} ← δ_{a(t−1)} ∪ δ(t)  // Otherwise, add it to the deltas set.
    for each δ ∈ δ_{a(t−1)}
        if δ = δ(t)
            pos_precon(a(t−1), δ, S(t−1)) // Positive precondition.
        else
            neg_precon(a(t−1), δ, S(t−1)) // Negative precondition.
```

FIGURE 4.2: The rule creation algorithm.

4.3.2 Outcome preconditions

As we mentioned before, actions may sometimes produce duplicate outcomes. To counter this, we take an extra **merge** step. This step checks the preconditions to see if there are any outcomes in this rule that share them and merges them. Checking is done by looking at the prior and counterprior counts of the variables (positive and negative preconditions). For example, if an action $a(t-1)$ led to a certain outcome $\delta(t)$ while a state variable $s_i(t-1)$ had the value v and never when it had any other value, then we can infer that, for the action $a(t-1)$ to produce the outcome $\delta(t)$, the variable s_i must have the value v.

Conversely, if an action $a(t-1)$ led to a certain outcome $\delta(t)$ while a state variable $s_i(t-1)$ did **not** have the value v, and did not lead to this outcome when s_i had any other value, we can infer that, for the action $a(t-1)$ to produce the outcome $\delta(t)$, the variable s_i must **not** have the value v. We call these a priori necessities **preconditions**, and the inference engine works to infer them at any point they might be needed. State variables that do not change between two time steps are discarded.

4.3.3 Merging

After we have produced the preconditions for each outcome, we want to **merge** them, because usually most outcomes will belong to the same case. This is done to cope with noise and impart an ability to our algorithm to handle near-deterministic models. For example, an action might sometimes cause a variable to increase by 10 and sometimes

by 5. In this case, the two rules should be merged either to a weighted average or to a local probability distribution (either is possible with our algorithm).

This **merging** is done by sorting the rules by their preconditions and iterating through them, looking for "near" duplicates. It stands to reason that, if a result δ_i of an action a_i has priors p_i and another result δ_j of the same action has the same priors, then the two results must be different instances of the same mechanism (or at least be results of different but indistinguishable mechanisms, in which case we cannot tell the difference).

Merging the rules has the effect of "filtering" them to increase their relevance. It removes duplicate rules and replaces them with one rule that contains their weighted average (if the variable is ordered) or a local probability distribution (if the variable is unordered) for each rule.

If, for example, the set of outcomes to be merged, δ, contains outcomes that have a different number of variable deltas (perhaps one outcome, r_1, contains deltas for variables s_1, s_2 and s_3 and another one, r_2 contains deltas for the variables s_2 and s_3). Now, also suppose that s_3 is an unordered set variable and the others are integer variables. We can replace r_1 and r_2 with another outcome, r_3, that contains the numerical averages of s_2:

$$r_3(s_2) = \frac{r_1(s_2) + r_2(s_2)}{2}$$

but contains a probability distribution for s_1 and s_3. That is, if r_1 had been observed N times and r_2 M times, we could say that:

$$r_3(s_1) = \begin{cases} r_1(s_1) & \text{with probability} \frac{N}{N+M} \\ \text{Null} & \text{with probability} \frac{M}{N+M} \end{cases}$$

and:

$$r_3(s_3) = \begin{cases} r_1(s_3) & \text{with probability} \frac{N}{N+M} \\ r_2(s_3) & \text{with probability} \frac{M}{N+M} \end{cases}$$

We can, of course, store the probability distribution for any type of variable instead of taking the average, since this would retain useful information. We could then use this information at a later time to calculate the averages.

The merging step for the preconditions has a complexity of $O(n * m)$ where n is the number of state variables (not states), and m is the number of **interesting values** for each variable. **Interesting values** are the values we would like to differentiate between in each state variable. For example, an integer variable might have interesting values of 0, $(0, max)$ and max, because we might decide that intermediate values do not hold any special significance, while the values on the edges of this variable's range do.

4.3.4 Data collection

The data needed for rule induction can be collected by acting randomly in the world, but in many domains random actions may not cover the state space sufficiently, therefore it would be better to use directed exploration. Directed exploration the ability to choose which areas of the state space one would like to explore at any given moment.

In directed exploration, the planner would use some metric (such as direness of the situation or opportunity) to decide whether to experiment at a particular action. For example, if the planner decided that it could experiment at the given time, it could check to see whether some action had not been taken at the current state before (or not been taken adequately), and decide to perform that action in order to gather more data.

Directed exploration is an immediate result of the online property of our algorithm, which makes the results available to the planner as the states are encountered, and thus aids it in making its decisions with up-to-date data.

As should be apparent, this method of exploration speeds up the data gathering process significantly and can help us reach conclusions much more quickly and efficiently.

4.4 Using the inferred rules to act

After our rules have been created, we can use other techniques to decide our actions. One way to do this would be deciding which state we would like to be in in the long term, observing our current state and then using rules as transformations on the current state to reach our target.

A planner could observe the current state, decide which state it would (ideally) like to arrive in from the way the game has evolved so far, and construct a "chain" of actions that would lead it through states to the final one. This process is known as **planning** and the "chain" is called a **plan**.

Since the rules are not known beforehand, the planner can take liberties with the generation of the plan. It can, for example, specify initially that it wants a variable v to always be maximised, and the specific action that causes this maximisation will change depending on the rules that are generated during play.

Initially, since the player will know nothing about the actions, it can fall back on a purely random player, which will, nevertheless, aid in exploration. When the rules start being generated, the planner can choose whether or not it wants to explore, and run a random action or a planned one respectively.

Chapter 5

Experimental results

5.1 Test methodology

The following results were obtained by first running a random player (an agent with no set policy which performs actions at random) and let it play for 800 actions. We then generated the rules using our algorithm off-line and added a simple planner for making decisions afterwards.

In the second experiment, we used an online player. Initially the player knows nothing about the world, but is imbued with planning directives, and the player must extract information about the world on which actions it can perform to honour those directives.

5.2 Derived rules

Some of the rules derived in this brief session are shown below. For comparison, the actual rules are included inside the listings in natural language.

The "pause" action, shown in listing 1, is not an actual action but merely a way for the agent to effect the passing of time. This action is our "control", showing us what happens if we do nothing. When fighting, the player takes some damage, while nothing happens when not fighting.

We can see that in this case the preconditions are wrong, namely that this action does not require **mana** to have any specific value. The erroneous preconditions have caused the merging step to err, creating two rules instead of one. Due to the fact that the time spent in a fight is much more than that spent outside it, our agent has not seen enough data to decide that "pause" does nothing when not in a fight.

Listing 1 The created rules for the "pause" action.

```
Actual:
This action causes the player to take damage,
averaging ca. 40 points per action, when fighting.

Detected:
With confidence 18, this action causes:
    hp to decrease by 48.6111111111.
  mana MUST NOT be 1.
With confidence 14, this action causes:
    hp to decrease by 35.
  mana MUST NOT be 2.
```

Listing 2 The created rules for the "cast spell" action.

```
Actual:
This action causes mana to decrease by 8 and
the opponent to take damage.

Detected:
With confidence 142, this action causes:
    mana to decrease by 8.0,
    hp to decrease by 64.5774647887,
    mob_health to decrease by 1.0.
With confidence 131, this action causes:
    mana to decrease by 7.72519083969,
    mob_health to decrease by 1.0.
  hp MUST NOT be 2.
With confidence 26, this action causes:
    mana to decrease by 8.0,
    hp to decrease by 140.0,
    mob_health to decrease by 1.0.
  hp MUST NOT be 1.
```

The "cast spell" action, shown in listing 2, is a way for the player to cast an offensive spell on the target, draining its **health**. We can see that the effects are once more detected correctly, but the priors are misdetected. This is the result of seeing too few states, but with directed exploration our algorithm can infer rules about the world in far fewer steps.

The "heal" action, shown in listing 3, increases the **health** and **mana** of the player. **Health** is increased by 200 and **mana** by 60, but we see that the detected **health** increase is 116, on average. This is, as we said before, because the enemy deals damage which decreases our **health** in every turn.

The "strike" action, shown in listing 4, deals damage to the opponent and starts a fight

Listing 3 The created rules for the "heal" action.

```
Actual:
This action causes health to increase by 200 and
mana to increase by 60.

Detected:
With confidence 618, this action causes:
    mana to increase by 58.0,
    hp to increase by 116.966019417.
  mob_health MUST NOT be 0.
With confidence 158, this action causes:
    mana to increase by 16.0 (capped),
    hp to increase by 116.53164557.
  hp MUST NOT be 2, and mob_health MUST NOT be 0.
```

if there isn't one. We can see that this is correctly detected and the change of the state in **fighting** is correctly interpreted. The decrease in **health** is, again, because we get dealt damage by the opponent when in a fight.

5.3 Planning

To obtain initial test results, we used a rudimentary planner that acts in very basic ways to ensure its survival, yet makes full use of the rules we have created. We have programmed the planner to do three things necessary to our agent's survival and success:

- If our health is lower than the amount we can heal from the maximal health, perform a healing action. For example, if the healing action would restore 100 health and our health is 99 points below the maximum, do not heal. If we are 100 or more points below the maximum, then heal.

  This is done purely to avoid wasting healing power, which might not be abundant in a more complex model, should we choose to use one.

- If the previous action need not be done, then check if we are currently fighting. If not, perform an actions that will lead us to a fighting state.

  This is done because the agent gains experience from fighting, so it needs to do as much of it as it can.

- If we are in a fight and an action causes harm to an opponent, perform it. This is done so we defeat our opponent, obviously.

Listing 4 The created rules for the "strike" action.

```
Actual:
This action causes "fighting" to change to 1 if 0,
mob health gets reset to 10, and damage is dealt
to the opponent (mob). The opponent dies when
mob_health is 0, so fighting turns to 0.

Detected:
With confidence 86, this action causes:
    mana to increase by 1.0,
    hp to decrease by 47.9534883721,
    mob_health to decrease by 0.823529411765.
  mana MUST NOT be 1, and mob_health MUST NOT be 0.
With confidence 79, this action causes:
    hp to decrease by 46.25,
    mob_health to decrease by 1.0.
  mana MUST NOT be 2,
  and hp MUST NOT be 2,
  and mob_health MUST NOT be 0.
With confidence 42, this action causes:
    hp to decrease by 34.3333333333,
    mob_health to decrease by 1.0.
  mana MUST NOT be 1,
  and hp MUST NOT be 1,
  and mob_health MUST NOT be 0.
With confidence 37, this action causes:
    mob_health to decrease by 1.
  mana MUST NOT be 1,
  and hp MUST NOT be 2,
  and mob_health MUST NOT be 0.
With confidence 13, this action causes:
    hp to decrease by 35,
    mob_health to increase by 10,
    fighting to change to 1.
  mana MUST NOT be 1,
  and hp MUST NOT be 1,
  and mob_health MUST NOT be 1.
```

After the random player has run and gathered the data, we can now use the above generated rules to play. Initially, the current state is inspected and the actions with preconditions that do not match ours are culled. Afterwards, the planner will decide which action to use according to its directives.

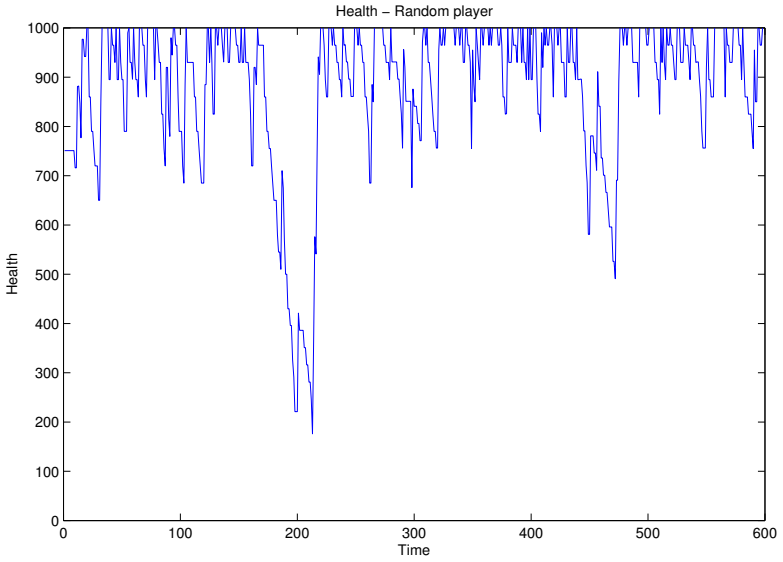

FIGURE 5.1: The health fluctuations of a random player

The graph in figure 5.1 shows the health fluctuations of a random player while it operates in the world. As we see from the graph, the random player's health fluctuates wildly, as we would expect. We, in fact, hard-coded a directive into the player which did not allow it to drop its health below 200 (because it would die and the experiment would end), so the drops in health in the graph could very well have resulted in the player's death, had it not been for our intervention.

In figure 5.2, on the other hand, we can see that the health of the player fluctuates minimally, between about 880 and 1000. This is due to the fact that our algorithm calculated that the healing action will increase the player's health by 120, so it performed this action as soon as it was possible to gain the full amount of health (to avoid wasting time and healing energy). The decreases in health below 880 are because the player can, sometimes, be dealt large amounts of damage that will get its health to a low level before it can heal again.

This player was trained using the results of a random player for about 1800 samples. Our algorithm was run on these results and produced rules, which were kept fixed during the offline player's play.

In figure 5.3, the player starts out with no knowledge (essentially a random player). It then performs actions until it learns enough to establish a few rules. We can see that

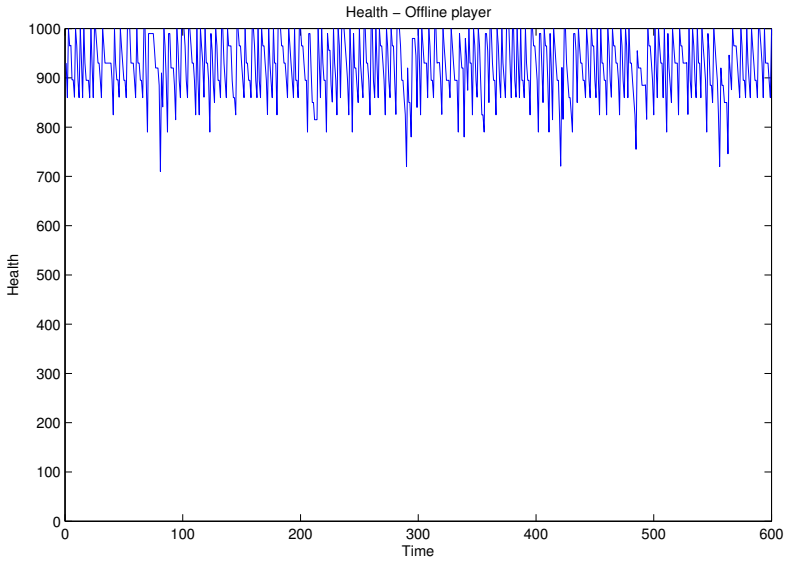

FIGURE 5.2: The health fluctuations of a player with a policy

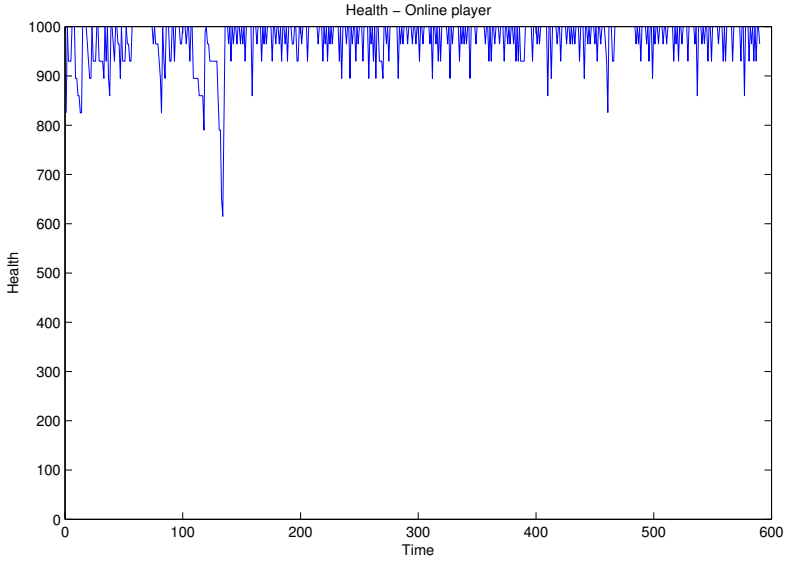

FIGURE 5.3: The health fluctuations of an online player with a policy

by **150 samples** it has learnt rules good enough to play optimally, and its health never drops below 900 since (except for cases of large damage, where the play is still optimal).

Chapter 6

Discussion and conclusion

6.1 Strengths

We believe that our algorithm is an improvement over previous ones for numerous reasons. It is also clear that it has some disadvantages, but we believe that these are significantly outweighed by its advantages. Some of its advantages are:

- It operates at a high level, which means that the rules it generates are very compact. It not only infers which variables changed, but also **how** they changed, usually with specific values. The more deterministic a domain, the better the values our algorithm provides. This offers an obvious advantage to planning, because the planner can predict with greater accuracy how an action is going to alter the current state.
- It needs very few data to start creating rules. Even from the first input datum, it infers useful rules that fit the data as best it can. These rules may later be revised, superseded, or reinforced, but there is virtually no bootstrapping period as we can begin to perform actions that are more relevant to the environment almost immediately.
- It is an online algorithm, which means that there is no lengthy calculation step after a sequence of events. It can refine the rules as they come in, and then pass them off to the planner so the quality of the decisions keeps improving.
- Since it is an online algorithm, it runs in $O(n)$ time and $O(1)$ space with n being the number of observed states. It uses constant space because it does not store all the state data, so the space is not dependent on the number of observations but

on the number of state variables. The $O(n)$ time complexity is because each state only gets processed once.

- It provides the planner with confidence measurements on each rule. This means that the planner can **actively** decide if it wants to experiment with another area of the search space. For example, if the inference has not seen enough cases of a certain state variable having a certain value, the planner can decide to visit those areas of the search space in order to obtain more diverse data. Other algorithms do not provide such an option.

6.2 Weaknesses

As usual, there are always tradeoffs, and our case is no different. Some of the disadvantages of our algorithm are:

- It is not well suited to purely stochastic environments. If the world is purely stochastic, the advantage of precise reporting our algorithm offers will be lost. A probability distribution can still be of use, but our technique might not be as efficient in these cases.
- It cannot handle multiple variable dependencies. If a result is dependent on the **combined** value of two or more variables, our algorithm will not detect this correctly. This does not mean that our algorithm cannot detect dependencies of more than one variable, but rather it cannot detect when a precondition requires two variables to have specific value combinations. For example, if variable A is required to be True when variable B is False, but it is required to be False when B is True, our approach will not be able to reason about the combined state of the two variables. Support for this can be added with a few simple modifications, but our specific problem does not have a demand for this and thus it is left as future work.

6.3 Future work

Our work is by no means complete. Areas on which our approach might be improved include:

- Including support for multiple variable dependencies. If the problem demands it, our algorithm can be modified to support dependencies on combinations of multiple variables by combining the values of the variables when looking for preconditions.

- Using the algorithm with a planner that can use the state information our algorithm provides to arrive to a plan more quickly (directed exploration). This would involve experimenting with unseen states to improve the quality of the rules in a shorter time span.

6.4 Conclusion

Inspired by the way humans approach problems, we have presented a method to infer rules about the world and generalise examples of data so that we can apply the actions to unknown circumstances.

We have demonstrated the advantages of our algorithm, namely the fact that it is online, it is amenable to directed learning, and it is fast with few memory requirements.

The results have been demonstrated, and they show a marked increase of the ability of the player to function in the game, as compared to the random player.

Bibliography

Leslie Pack Kaelbling, Michael L. Littman, and Andrew W. Moore. Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, 4:237–285, 1996.

Mark Wexler. Embodied induction: Learning external representations. In *AAAI Fall Symposium*, pages 134–138. AAAI Press, 1996.

Zoubin Ghahramani. Learning dynamic Bayesian networks. In C.L. Giles and M. Gori, editors, *Adaptive Processing of Sequences and Data Structures*, pages 168–197. Springer-Verlag, 1998.

Michael I. Jordan. *Learning in Graphical Models*. MIT Press, 1999.

Steven Salzberg. Heuristics for inductive learning. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 603–609, 1985.

Stephen Muggleton, editor. *Inductive Logic Programming*. Academic Press, 1992.

Eyal Amir and Allen Chang. Learning partially observable deterministic action models. *Journal of Artificial Intelligence Research*, 33:349–402, 2008.

Carlos Diuk, Alexander L. Strehl, and Michael L. Littman. A hierarchical approach to efficient reinforcement learning in deterministic domains. In *Proceedings of the Fifth International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 313–319, 2006.

Craig Boutilier, Thomas Dean, and Steve Hanks. Decision theoretic planning: Structural assumptions and computational leverage. *Journal of Artificial Intelligence Research*, 11:1–94, 1999.

Thomas Hamelryck. Mocapy, a Dynamic Bayesian Network toolkit. http://sourceforge.net/projects/mocapy/, 2009.

Søren Feodor Nielsen. The stochastic EM algorithm: Estimation and asymptotic results. *Bernoulli*, 6(3):457–489, October 07 2000.