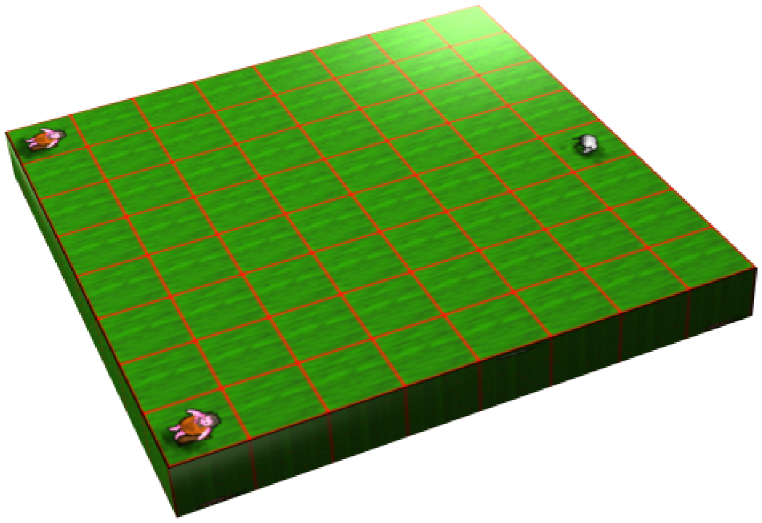
Using Bayesian Networks for Modeling Computer Game Agents



Andreas S. Værge
Henrik Jarlskov

11th June 2003

The Faculty of Engineering and Science

Department of Computer Science

TITLE: Using Bayesian Networks for Modeling Computer Game Agents

PROJECT PERIOD:
DAT6,
4th of February - 11th of June, 2003

PROJECT GROUP:
E1-119

GROUP MEMBERS:
Andreas Schou Værge
Henrik Jarlskov

PROJECT SUPERVISOR:
Uffe Kjærulff

NUMBER OF COPIES: 6

REPORT PAGENUMBERS: 58

APPENDIX PAGENUMBERS: 6

TOTAL PAGENUMBERS: 72

SYNOPSIS:

In this report we are going to analyze how an agent with a Bayesian Network can use it for predicting another agent, in order to enhance its performance.
In order to do this we first implement a simple example game which will be used as a basis for our tests. This includes the implementation of two simple agents, which can be used in conjunction with the more advanced AI.
We are going to model the Bayesian Networks needed for predicting the agents of the game, followed by the training of the Bayesian Networks and tests to measure how well they perform. We found that one of the models had a problem learning the strategy of the agent it was supposed to learn from. Reasons for this were found and discussed.
We are also going to discuss our experiences with Bayesian Networks through this project, and compare it to some of the other technologies which could have been used, as well as come with some propositions for future work.
In the conclusion we found that the approach we have used had some problems. The main problem was the size of the conditional probability tables, which prevented the agent from learning effectively.

Summary

As the game industry leaps across many different technologies in many areas their advancement in the field of artificial intelligence is limited. Agents in games today are mostly still simple, scripted agents, offering un-humanlike behavior. There are many technologies available that could offer benefits beyond humanlike behavior, but the risk of adding the technology is not easily overcome in such a competitive market. One of the technologies that has not been adopted as of yet is Bayesian Networks. This type of technology offers humanlike reasoning, adaptation and much more. Even though the technology offers many advantages it is necessary to weigh the advantages against the disadvantages.

Before any solutions could be found the problem had to be be analyzed. We started by analyzing the problem area, which in this case deals with artificial intelligence, Bayesian Networks and other agent technologies and the related work available. When the problem area had been analyzed we were ready to describe the expected results. In this report Bayesian Networks were used in an example game development process to find both advantages and disadvantages, in order to weigh these against one another. An example game was be developed, offering the basis to test the different aspects of Bayesian Networks against scripted agents. Before agents can be equipped with Bayesian Networks it is important to model the problem first. This process is not straightforward and requires a large analysis of the problem area in order to model the problem correctly. This can be seen in the numerous models we found before finding a model which seemed to fit the problem sufficiently. The first thing we did was to create two models to have as reference. The first model showed how all the information available would effect the decision we tried to predict. The second model went a different way, and tried to model exactly how the decision was made. These two models were made to give us an idea of the two different directions in between which we needed to find the answer. The final model took ideas from both models and used them in order to perform the prediction.

The other part of the process was to implement the game, so that it would be possible to both train the models we ended up using, and later test the effect of using the models. The basic idea of how we created the game, was that it had to as simple as possible, while still having enough headroom to leave sufficiently interesting decisions to predict. The game was made to consist of a number of parts. First the basic game, which took care of asking the agents for their decisions and carrying them out, and the agents. The agents were of three types. The first type was a sheep which was to avoid being caught. The other agents were two types of cavemen. The first caveman was very simple, and had the purpose of chasing the sheep without any thought. Later on another kind of caveman was added. This caveman used the models we had trained in order to predict both the simple caveman and the sheep, and thus enhance his possibilities of capturing the sheep.

When the Bayesian Network has been modeled and the game implemented, the models need to be trained in order to be able to reason in the environment. There are two different ways we could learn the Bayesian Network the strategy of the agent. The first was to use Sequential Learning, which would allow us to update the beliefs of the network after each move, and also allow us to use fading. The second was to use EM Learning, provided by the tool we used to model the networks in, called Hugin. EM Learning allowed us to run thousands of games, saving the results in a file, and then at a later point feed this file to Hugin together with the network, and we would have a trained network. This second method did not support fading, which is the possibility of letting the influence of the past fade. This method enables the model to adapt to new environments faster. We decided to start the training with

EM Learning and then when we needed the model to adapt to the changing environment we changed to sequential learning, thus gaining access to fading. The choice of what method to use for training was not the only choice. In order to have training data you need a training method. We tested different training methods. The first we considered and tested, was to take all the game's possible configurations and in that way be certain that we had full coverage through the training cases. This method was only tested for one reason, namely that we wanted to see what the optimally trained model was capable of. The main model we used was trained in a different manner, namely through running a number of games with random start locations. Both methods had their use. The first method gave us the idea of how well the model was to predict the world, while the other method gave us an idea of how many "real" world cases were needed for the model's predictions to be sufficiently accurate.

After implementing the game and training the Bayesian Network the performance of the implemented agent needed to be tested. In some cases the tests were showing a surprising result, as the network suffered from several setbacks. The cause of these setbacks were ambiguity in the model and an error during training. However, the tests clearly showed an increased performance, as expected. The test of the adaptiveness clearly showed the gain in adding fading to the network.

With the tests finished the advantages and disadvantages of the process could be described. Bayesian Networks offered many intriguing aspects, including adaptation, structural learning and structural control, but weighed against the disadvantages of the time required in prior analysis and the problem of sufficient coverage it was difficult to see a clearcut answer. However, during the process different aspects came up, that offered room for future work. One of the aspects that could add great dimensions to human side in agents was the addition of communication and trust. Bayesian Networks offers an easy way to model these aspects. Another aspect for future work was to combine structural learning with the principles of GoCap, a system for training adaptive models to mimic human behavior. The last aspect treated in future work is the aspect of dealing with Bayesian Networks when realtime games are considered. The requirements in this application area are far more strict, than the area we deal with in this report.

The conclusion to this report was that there were many advantages in using Bayesian Networks compared to many of the technologies in use today, but the need for the advantages has to be weighed against the disadvantages of the difficulty in modeling the problem as opposed to the current method. It takes more time to model something in Bayesian Networks compared to scripting it, but the advantage is that in many ways the player of the game is presented with a far more humanlike opponent. For most games this is not worth the risk of testing a new technology, but for some this would be a way of profiling their game. Our approach of using the Bayesian Network for every part of the agent's decisions appear to be too low-level, and that the technology would serve better at a higher abstraction level. Even though our example game shows many sides to Bayesian Network, it can be difficult to predict the real interest in the technology until it has seen use in a commercial game.

Preface

The results of this report were reached using software made available by Hugin Expert and the technical assistance of Uffe Kjærulff.

The program code produced and the Bayesian Networks we used can be found on the CD-ROM attached to this report.

Andreas S. Værge

Henrik Jarlskov

Contents

1 Introduction **1**
1.1 Introduction 1

2 Problem Analysis **4**
2.1 Analysis of the Problem Domain 4
2.2 Problem Domain 6
2.3 Introduction to Bayesian networks 8
2.4 Agent Technologies 11
2.5 Related Work 14

3 Solving The Problem **16**
3.1 The Game “Caveman’s Struggle of Life” 16
3.2 Step-By-Step Approach 17
3.3 Algorithms 18
3.4 Implementation 19

4 Modeling **23**
4.1 Sheep’s Strategy 23
4.2 Fellow Caveman’s Strategy 28
4.3 The Final Network 30

5 Training And Testing The Network **34**
5.1 Strategies for Training and Testing 34
5.2 Test Results 37

6 Potential for Bayesian Network Agents in Games **45**
6.1 Advantages of Bayesian Networks 45
6.2 Disadvantages of Bayesian Networks 46
6.3 Summarizing Benefits and Costs of using Bayesian Networks 46

7 Future Work **48**
7.1 New Rules 49
7.2 Introducing Communication 49
7.3 Introducing Trust 51
7.4 Other Areas 53

8 Conclusion **56**

A Appendix **59**
A.1 Source code for the sheep 59
A.2 Source code for the DFA-agent (Caveman) 60
A.3 Loading and compiling a domain in the Hugin API 61
A.4 Source code for the BN-agent with a lookahead of two. 62

Chapter 1

Introduction

1.1 Introduction

Each year the computer gaming industry increases its revenue. “With an estimated global value of some $10 to $20 billion, the industry rivals Hollywood in revenues and is now recognized as a propulsive force behind the creation of markets for information and communication technologies.”, as stated by Chido Onumah, University of Western Ontario, April 2002 - *it has become big business*. To please the ever growing consumer market many new algorithmic methods and technologies are integrated into games each year. The advances in graphics, sound and network-playability over the last ten years is enormous. However one area has not seen as much technological advancement yet, the *artificial intelligence* in game agents. Many interesting approaches have been proposed by the academic society, but few of these have seen actual use in the game industry. An example of this is the concept of *adaptation*. It is more fun to play against a computer opponent that does not always use the same strategy, but adapts to the player’s strategy. The advantages of such an agent is expressed by John Manslow in [AIGPW02].

‘.., solutions to problems that are extremely difficult to solve “manually” can be discovered by learning algorithms, often with minimal human supervision... Moreover, in-game learning can be used to adapt to conditions that cannot be anticipated prior to the game’s release, such as the particular styles, tastes, and dispositions of individual players..”

An opponent that cannot be overcome by only one strategy is considered far more fun to play. Even though it is important to have your games provide a long lifespan of fun adaptation has seen very few actual use in the industry. The methods are not often without disadvantages. Deadlines are extremely important in this fast-paced industry. There are two main reasons for not introducing more adaptation into the industry, even though it is craving for new “revolutionary improvements in new areas”, according to John Manslow in [AIGPW02].

‘.. few games have been released to date that have contained any level of adaptation, despite the fact that it is well within the capabilities of current hardware. There are a variety of reasons for this, including: 1, Until recently, the lack of precedent of the successful application of learning in a mainstream top-rated game means that the technology is unproven and hence perceived as being high risk. 2, Learning algorithms are frequently associated with technologies such as neural networks and genetic algorithms, which are difficult to apply in-game due to their relatively low efficiency.”

The industry has chosen to stay with what it knows, *scripting*. It needs to be easy to predict the consequences and it needs to be easy to test and debug.

Among AI Designers[1] in the industry there are several reasons for not using a more advanced type of technology. One of the more important reasons is *speed*. It needs to be fast and efficient, so that the game can have great graphics and sound rendered with acceptable framerates. All these things mentioned here needs to be taken into account, advantages and disadvantages explained, for the industry to consider any other type of artificial intelligence for their agents.

Why then should Bayesian Networks be suited as artificial intelligence technology in game agents? The main reason for choosing to examine this type of network is expressed in [AIGPW02].

‘Game AI agents exist within virtual worlds of our own design. Because of this, we have the power to grant them perfect knowledge of the world if we desire. However, there are many cases in which we would prefer to limit an AI agent’s knowledge of the world and force it to perform humanlike reasoning in order to form a less accurate - but more interesting - assessment of its current situation.”

- Paul Tozour, Ion Storm

Bayesian Networks provides a tool to reason about uncertainty and also allowing the agent to use adaptation, thereby adding another dimension to its use. This gives us both the learning and adaptation John Manslow is requesting and also the strength to reason about a world where we do not have complete knowledge in a humanlike manner, as promoted by Paul Tozour. From what we know now this would sound like Bayesian Networks could prove a good candidate. It may then seem strange that the methods of Bayesian Networks have not been tested in commercial games as of yet. A study of much of the newer literature among people from academia and the game industry also seems to point towards a general interest in the possibilities of Bayesian Networks. Examples of attempts to promote the use of Bayesian Networks can be seen in [AIGPW02] and [AIUGDN99]. However, they do not go into both advantages and disadvantages equally. This is important in order for the industry to even consider using this method. They need to know both the good and the bad, not just an introduction to the concepts and the *possible* advantages of its use.

In this report we will try to find both advantages and disadvantages and measure the impact of using Bayesian Networks in a game development project. In order to do this we will use scientific articles and books, such as [BNDG01] and [ML97], and from AI Designers and other people working in the industry, such as [AIGPW02], [GEMS1] and [GEMS3]. In addition to this we will also try to make an example game where agents, using Bayesian Networks, will act as players. To show both the advantages *and* disadvantages of using Bayesian Networks we will describe the entire process of making the agents, not just the results. We have chosen to utilize the strengths of Bayesian Networks to predict the behavior of an opponent agent and use this knowledge to optimize the strategy of an agent. The focus of the game type will be on *turn-based* board games. Even though we have chosen to focus on using Bayesian Networks for prediction they can also be utilized otherwise in games. An example could be to use a Bayesian Network to find the best strategy to use among a set of choices. This type of utilization can be seen, using the Unreal Tournament™engine, in [RBBN02].

This report will start by analyzing the problem domain, which includes both Bayesian Networks and other types of agent technologies used in the industry. The results expected in this report and prior considerations are discussed throughout the problem domain analysis. Afterwards, an introduction to the game, *Caveman’s Struggle of Life*, and the algorithms used in our implementation will be

[1]We use the term AI Designer for the person who is responsible for designing and implementing the AI.

described. It will be followed by an explanation of the actual implementation. To show the results of the implementation a test is planned and executed. The results are discussed together with those aspects that could lead to future work. In the conclusion the overall results of the report will be presented and summarized.

Chapter 2

Problem Analysis

2.1 Analysis of the Problem Domain

In order to specify the problem we want to solve, it is important that the problem domain is analyzed. There are four main areas that this report will be dealing with, *Computer Games, Artificial Intelligence, Bayesian Networks* and *Alternatives to Bayesian Networks*. In the analysis of the problem domain we will focus on three of these. The fourth, *Bayesian Networks*, will be dealt with in more detail in Section 2.3.

2.1.1 Computer Games

Computer games are intricate work, which requires coordination from many different areas of software development. In a game there needs to be someone taking care of graphics and sound, both coding the software in the game for handling it, but also for making the graphics, the music and the sound. An engine needs to be coded, which coordinates graphics, music, sound and all the rest with the user input. It also needs to handle input/output for accessing the file system and network if required, and it *all* needs to be tested and debugged *thoroughly*. Even before all this is made both a game plot and design needs to be finished. When the game is finally finished, advertisement and production can begin. The coding of the game agents is but a small part of the overall process. Testing is important, especially in console games, where you cannot send out a patch to fix a problem. The release you send out *is* the final product. However making a good, stable product is not enough. The industry is always on the move. There is an everlasting struggle to add new technology that gives your game something that makes it visible in the ever growing selection of games on the market. If you add a new exciting technology that sounds revolutionary your game will receive more attention that it would have otherwise. The game also needs to last as long as possible. An example of a game producer that succeeded in this is Sierra with the game, Half-Life. The game first offered a great gameplay, and became Game of The Year, then stayed on the list of the most selling games for more than 4 years[1], because of the possibility for people who had bought the game to add new stuff to it - changing it completely. An uncountable number of modifications[2] to Half-Life have appeared each year made by enthusiasts, none however as popular as Counter-Strike[3]. Up until now it is estimated that the game Half-Life has sold more than 3 million copies worldwide [VUGF03]. This

[1]http://half-life.sierra.com

[2]A modification is an addon to a game, which changes the plot, graphics, sounds, everything, but still running on the original game's engine

[3]http://www.counter-strike.net

comes to show that a long life span in any game is crucial. One of the things that made Half-Life seem interesting to many players at first was that the artificial intelligence in the game was seen as revolutionary for its time. The AI Designer, Lars Lidén of Valve Software, had simply scripted the entire Squad Tactics book of the American Army into the tactics used by the opposing force [AIGPW02].

2.1.2 Artificial Intelligence

Behind game agents there is always some type of artificial intelligence. As mentioned earlier it is not always the area that receives the most attention in the game creation process, unless the designers makes the agents' role central to the game. The requirements made to the AI Designers are often hard in this business. The game agent needs to be well within strict bounds of *speed, efficiency, predictability, testability, easiness to debug and easiness to implement.* To make it even more difficult the agent is also often added late in the process as it needs the engine's input and output to be able to tested correctly. This common situation is expressed in [AIGPW02].

'Far too often, AI has been a last-minute rush job, implemented in the final two or three months of development by overcaffeinated programmers with dark circles under their eyes and thousands of other high-priority tasks to complete"

- Paul Tozour, Ion Storm

While the AI Designer makes sure to stay within these bounds he must at the same time make the agent seem as intelligent as possible to the player. You also want it to challenge the player, making him feel like he is playing against an intelligent opponent. A player does not want the agent to have to cheat in order to win. An example of cheating is for example if you are playing a war game. The player is attacking the agent all the time at the same place with a small force every once in a while. However he is also secretly building a large invasion force somewhere else, preparing to launch an assault from an entirely different place (for example he could have sent his newly built invasion force secretly onto the other side of the agent's base). If the AI Designer made their agent cheat, they could have chosen to let it be able to see the entire world, even though it should not be able to. The agent would then prepare a defense on the other side of the base, even though it has never before seen an attack from here, and therefore should not expect it, unless it has played against a player before that chose to use the same strategy. The player will see his invasion force meeting an already prepared defense force, and be countered, even though he did everything in total secrecy. This would never have happened against a human player, and therefore it would feel unrealistic to the player. To sum up: *designing a good artificial intelligence for a game agent is not an easy task, and all the restrictions put upon it by the industry is only slowly being removed, as the focus is turning toward the agents in games. For any new technology in artificial intelligence to be accepted into the industry it needs to show how well it performs within these bounds, what it would add and at what price.*

2.1.3 Alternatives

It is however not enough only to consider the advantages and disadvantages of Bayesian Networks. To add a perspective alternatives also need to be taken into account for comparison, both the ones in use today in the industry and also ones that might help with a disadvantage in Bayesian Networks or be a better choice in the given situation. There are many types of methods to use for artificial intelligence in games - a list can be seen in [AIGPW02]. We have chosen to focus on the main types, which are introduced briefly in Section 2.4. However, often you will not see a “clean” use of one type if a more advanced type of artificial intelligence is used. It will often be a hybrid of different methods. An example of this could be a fixed number of strategies, programmed in Fuzzy Logic, but the choice of strategies is made using Genetic Algorithms. Such hybrid considerations will not be taken into account, as they are often used to solve a specific problem.

2.2 Problem Domain

In order to demonstrate the different aspects of developing agents for computer games using Bayesian Networks we have chosen to develop a computer game on which the developed agents can interact. The rules for this game will be described in Section 3.1. As there are many aspects to be demonstrated and tested it is important to use a step-by-step approach in such a way that the changing parameters are controlled and the effect caused by each changed parameter may be observed. This step-by-step approach is described in Section 3.2. When choosing a game as the base of the tests there are a number of considerations to be made.

2.2.1 Considerations about the game

When choosing an example game for testing a new artificial intelligence technology considerations about the game prior to implementing it is crucial. The game must be generic enough for the observations made during testing to be usable in a similar commercial game development process. The overall complexity of the game must be accounted for. How many possible board configurations exists? If the game is expanded with more actors or a larger board how would this affect the agent making decisions or the set of possible board configurations? Such considerations are vital to commercial game development as you would often be adding something later that was not thought of when the first design was made. If the agent cannot handle such an expansion it would be useless and a waste of time. The agent also needs to be trained, so that it will not appear as a complete moron in the first game. For training it is important that the agent is put through as many scenarios as possible to make it as prepared as can be done before shipping it. To assure that it is trained properly a method of sampling is needed. An agent does not only need to be well trained, it should also be able to adapt to new situations. If the player uses a strategy the AI Designers did not think of, it could easily appear stupid if it fell for this strategy again and again. Therefore adaptiveness in the agent is a feature, which is being demanded by the customers. This aspect also needs to be considered before shipping it. How many examples does it need to see in order to change its strategy? If it begins to lose every game, should it let old knowledge “fade” to learn the players new strategy faster? In this section we will go into considerations about: *Game Complexity, Sampling Methods and Adaptiveness.*

- Game Complexity: The problem with modeling games is often the complexity, even with simple games such as the one we have chosen to implement. Using an $8 * 8$ board with 3 agents you get $64 * 63 * 62$ (249984) possible board configurations. This is a simple game, but make it just a bit more complex, like adding one more agent, and the complexity rises to 15.5 million possible board configurations. Imagine the number of possible board configurations of a game like chess, where you have an $8 * 8$ board, only now with 32 agents. Therefore when designing a Bayesian Network to use in these simple games it is vital that you first of all consider *accuracy* - how accurate you want the model to be. If we wanted the model to be able to predict all possible situations we would need to incorporate all 249984 situations somehow. The second problem that needs consideration is *ambiguity*. If you remove some of the accuracy of the model you might introduce ambiguity. This is when two or more different situations on the board will give the same output from the network, because the level of detail it models cannot distinguish them. This could add conflicting evidence into the network, as the agents you are trying to predict will act differently in situations the network perceive as identical.

- Sampling Methods: It is necessary that the sampling used for training has a sufficiently good coverage over the space of board configurations which can be encountered. One thing which should be considered is that not all configurations may be reachable, in this case it is not necessary to learn what happens in the given situation. There are a number of ways of training the network. The first possibility is to simply loop through all the board configurations, feeding the network with all the situations it might encounter. This method makes certain that all cases have been seen. However it does not show learning from a probable gaming situation, because you would rarely be able to go through all possible situations, both because of complexity, and because training data for all cases may not be available. Just imagine a possible strategy game with a grid size of $800 * 600$, with a maximum of 225 units per player (with a maximum of 16 players), and 17 different unit types for each player, with 10 possible actions for each. This is not an unlikely situation in a strategy game. It is undesirable to go through all these situations with for example a human, who is teaching the agent how to play. Also the method of going through all the possible board configurations seems to be cheating. However if a network could be put through each of the possible board configurations and provided with the "correct" answer to each, you would have a network trained as good as it can be. Therefore it would be a useful method to measure other sampling methods against.

  Another sampling method is to run a number of games with random start positions, assuming that the random function is uniform. This method should yield a good coverage of the different possible board configurations in our game. This method is less cheating and unlike the systematic run-through of possible configurations this can be implemented on all games, where you can find one or more variables to randomize that yields a good coverage of situations. Another reason for this method is that it better shows how the learning would work in a real game, learning how the player acts throughout the game, rather than only looking at specific cases. You would rarely find a player who runs through every possible board configuration in a systematic approach.

- Adaptiveness: The game also needs to allow the models to be tested with respect to the adaptiveness of the agents. This means that changes in the parameters should be possible in some way, for example adding obstacles or changing an agent's strategy. If such things were added to the game it would be possible to test if and how fast a model would be able to adapt its prediction strategy to the new environment. This is an important aspect of Bayesian Networks, as many of the current agent types used do not allow for real adaptation. Bayesian Networks provide this, but we need to test how fast and at what price it does so. If it requires 40.000 games to adapt its strategy it would be completely unusable in most games as most players do not play a game more than 40.000 times anyway. Another consideration in adaptiveness of a model is the use of *fading*. Fading adds to possibility of removing the importance of the past, allowing your agent to adapt to new situations faster, as the present is weighted higher. Therefore considerations about the use of fading, and how much should be used is important. If much fading is used the model would adapt faster to a new strategy presented by the player, but if he suddenly uses a strategy from the past it would appear as a new strategy and should be adapted again if the fading factor is too high. Adaptiveness is important to test and therefore it should be possible in the game chosen.

The problem of analyzing Bayesian Networks with respect to its use in games is not straightforward. Both theoretical and practical advantages and disadvantages needs to be considered thoroughly. This will be the focus of this report, and hopefully lead to the expected results, as described here.

Expected Results - *A functional game with working Bayesian Networks. The game is to be as representative as possible for board games. The game will have shown strengths and weaknesses about using this type of artificial intelligence technology. Throughout the report the problems and solutions found during modeling and implementation is described and compared with alternatives. The final result should describe the strengths, weaknesses and pitfalls in using Bayesian Networks as an alternative to current methods in the development of artificial intelligence agents in the gaming industry.*

As the main focus of this report is Bayesian Networks it is important that the reader knows what it is, and how it works. In the next section a definition of Bayesian Networks is given, followed by a brief introduction to some of the other issues of Bayesian Networks such as learning. Readers that already know of the theory behind Bayesian Networks can skip this part.

2.3 Introduction to Bayesian networks

One way of reasoning under uncertainty is using Bayesian Networks. The basic idea is to predict the outcome of a situation given existing knowledge about the domain. An example of a use for Bayesian Networks is the case where a doctor knows number of symptoms for a patient and needs to find the most probable decease, in order to treat the patient.
The methods used to calculate the outcome can be implemented using a computer, and then used as the basis for an agent in a game. In order to focus on the main aspect - how well does Bayesian Networks conform to the game industry for use as agents, we have chosen to use a tool, which provides a way of modeling and interacting with Bayesian Networks. In this project we are using a tool called Hugin in order to perform the different actions on the network. Hugin consists of two parts: a user interface, which can be used for modeling a Bayesian Network and performing most of the actions needed including compiling and propagating evidence (a screenshot of Hugin can be seen in Figure 2.1).
The second part is the API [HAPI6], which can be used to access a Bayesian Network, both with respect to learning and propagating evidence in the network. For a more complete description of Hugin see *http://www.hugin.com*.

2.3.1 Basics of Bayesian Networks

Bayesian Networks consists of two parts, *Structure* and *Probabilities*. Combining these provides a powerful method for reasoning under uncertainty.

- *Structure:* The first part which constitutes a Bayesian Network is an acyclic directed graph. This graph expresses the dependencies and independencies between the different variables. The reason for expressing the independence is to make it simpler to reason about complex issues, as only variables which are depending on each other are taken into account.

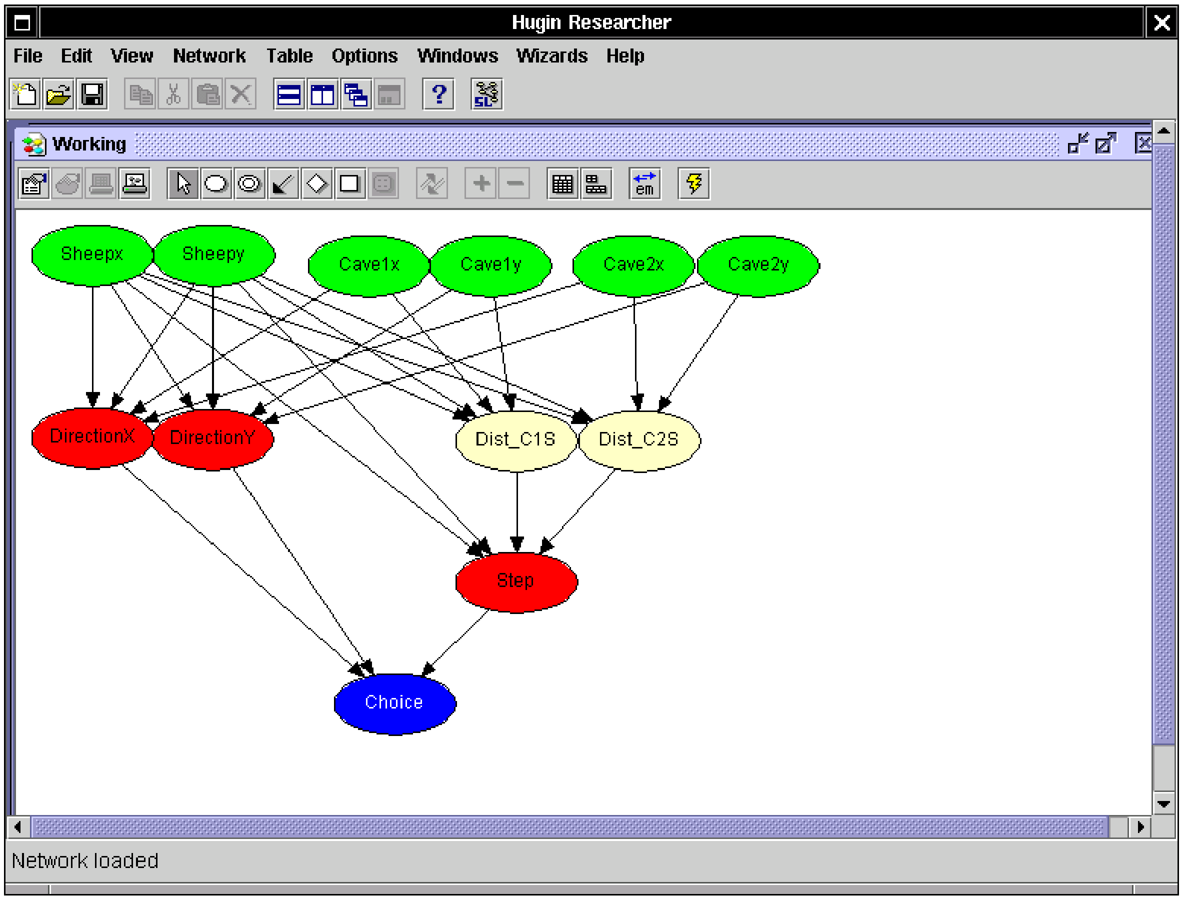


Figure 2.1: The Hugin User Interface

- *Probabilities:* The second part of a Bayesian Network is a collection of probabilities. A table is attached to each variable, such that we have a table expressing the probabilities of the variable in the possible states given the possible configurations of the variable's parents.

Given that the independence properties of the probability distribution $P(U)$ over a set of variables $U = \{X_1, X_2, X_3, ..., X_n\}$ can be represented by an acyclic directed graph, then $P(U)$ can be decomposed as follows:

$$P(U) = P(X_1, X_2, X_3, ..., X_n) = \prod_{i=1}^{n} P(X_i|pa(X_i)) \tag{2.1}$$

That is $P(U)$ can be expressed as the product of the conditional probability distributions of each variable given its parents ($pa(X)$), as seen in [BNDG01].
Further more the following holds for a Bayesian Network:

- A set of variables connected by a set of directed edges such that it constitutes an acyclic directed graph.

- Each variable consists of a finite set of mutually exclusive states.

- Each variable has attached to it a probability table expressing the conditional probability distribution given its parents.

2.3.2 Learning

One of the abilities found in Bayesian Networks is adaptation. This is accomplished by training the network. In this section we are going to explain the basic workings of two of the training methods available for Bayesian Networks. One is *sequential learning* where we are also going to touch upon the subject of *fading*. The second is *EM learning / batch learning*. The reason for focusing on these two is that they are made available from Hugin, and we are going to use both later on.

EM learning

EM learning is useful when a large set of training data is available. It is a method for learning from sets containing both complete and incomplete data. One strength is that when there is complete data available, learning is very fast. However, when data is missing it is necessary to propagate in the network in order to find the most probable value of the missing variable. This can be time consuming if the cases contains a lot of missing information, and thus missing information should be avoided if possible.
The method for training the network can be seen from the following formula:

$$\hat{P}(X|pa(X)) = \frac{n(fa(X) = (x,i))}{n(pa(X) = i)} \tag{2.2}$$

Where, $\hat{P}(X|pa(X))$ is the expected maximized (EM) probability of X given the parent configuration, $n(pa(X))$ is the number of times the given parent configuration of X has occurred in the dataset (and expectations if the dataset was not complete). $n(fa(X))$ is the number of times the whole family configuration for X has occurred.
One issue, which should be noted, is that fading is not possible when only a single dataset is available. However, fading between different datasets is possible by adjusting the experience counts. It is possible to start the training with EM learning and then at a later point switch to sequential learning.
In some cases it can prove useful to let the network have some prior knowledge before training starts. Thus the experience counts can be set to fit the credibility of the prior knowledge. How prior knowledge works can be seen from the following:

$$P(X|pa(X)) = \frac{n(fa(X) = (x,i)) + \alpha_i p_i}{n(pa(X) = i) + \alpha_i} \tag{2.3}$$

Where α_i is the experience count given to the prior knowledge (how credible is the knowledge?), p_i is the probability of the given configuration according to the prior knowledge. From the formula it can be seen that the prior knowledge now also is used for calculating the probabilities. Lowering the experience count, α_i, will let the prior knowledge have less effect on the new probabilities.

Sequential learning and Fading

Sequential learning is a method for learning from new training data one case at a time. This is useful when the network needs to gradually adapt to the environment. The main problem with sequential learning is that it becomes time consuming, the more cases the learning is based on.

A feature, which makes sequential learning useful is that it supports *fading* with individual cases. Fading is a method for making the experience of the past have less effect compared to the latest learning cases. This is useful when the environment may change over time, and thus it is useful when adapting to these new conditions. The smart thing with fading is that it does not have to be constant for the whole training period. It is fully possible to learn from the first cases without using fading, and then at a later point add fading and then even later either stop using fading once again or change the fading factor.

2.3.3 Influence Diagrams

Sometimes the possibility of making choices and finding the gain or loss with Bayesian Networks can be required. There is a method for doing this known as *Influence Diagrams*, which will be introduced here.
In order to make Bayesian Networks more usable when it comes to decision making, an expansion has been made, known as *influence diagrams*. The basic idea of influence diagrams is that apart from the normal variables, we also have *decision variables* and *utility functions*. A decision variable represents a choice between the mutually exclusive states. A utility function expresses a gain, positive or negative, from taking the different choices. This enhancement makes it possible to find the expected utilities and select the maximal expected utility among these.

This section provided some brief insight into Bayesian Networks, which will be the basis of this project. However, it is not the only technology on the scene of available agent technologies. In the next section some other technologies are introduced.

2.4 Agent Technologies

When faced with the choice of the technology to use in an agent for a game, a series of considerations must be made. The first is whether or not to make the agent adaptive. Agents with adaptive technologies tend to give a longer lifetime for games in general. When the player cannot predict the agent's strategy, and find a counter-strategy for it, it will also be more intriguing. However, using adaptive agents often comes with a price. If computing power is scarce for the agent the increase in computational complexity can be too expensive. If you need the agent to be completely deterministic in situations important to the plot of the game fully adaptive agents could prove an ill choice. A solution to this could be only to let the agent be adaptive in some cases and then use a different non-adaptive agent in cases where predictability is needed. There are two main advantages of using deterministic agents. The first one is that it is often the most efficient technology, requiring less computation. The second is, as mentioned earlier, you know the exact result of an action. To understand the usefulness of Bayesian Networks in games it is essential that they are compared with the other alternative technologies. It must be noted though, that not all agent technologies are described here, for example we have chosen not to explain Genetic Algorithms. For a list of most of the commonly used agent technologies beyond the following, see [AIGPW02].

2.4.1 Different Agent Technologies

In this section we are going to look at some of the most widely used agent technologies according to [AIGPW02], [GEMS1] & [GEMS3]. As the demands for longer lifetime for the games is increasing so is the use of simple deterministic agents declining. The earlier focus on graphics,

sound and plot instead of the opponents is slowly being replaced with companies that want their games to last longer on the market. The focus has turned towards the Artificial Intelligence. AI Design is now a natural part of game development, as expressed in [AIGPW02].
'It is frankly amazing to this developer how the field of game AI has grown and become an integral part of the game-development cycle over the past few years."

- Steven Woodcock, Wyrd Wyrks

As computer power is available in an ever increasing scale more and more computer power has been given to the agents of the game. Some games, such as Black & White [AIGPW02] using small Neural Networks, actually centers the gameplay around the agent.
The following five agent technologies show the different approaches one might take in the process of selecting a technology to use in a game development project. They each have their advantages and disadvantages, which will be described briefly.

- DFA: Agents constructed using Deterministic Finite Automata (DFA) or scripted agents, as they are also called, are extremely popular. They are by far the most widely used technology when implementing agents. The advantages are what many companies used to look for in an agent technology. They are fast, predictable and use little computing power. However their lifespan is also limited. They do not change their actions, or adapt to changing environments. For more informations on DFA's in games, see [AIGPW02] and [GEMS1].

- Fuzzy Logic: An agent technology used to make a scripted agent more unpredictable, thereby expanding its lifespan. The idea behind fuzzy logic is to use intervals instead of true/false. Then instead of asking "if" you can ask "how much?", and then set some actions when a threshold value is reached. Fuzzy Logic also requires little computing power, however still more than scripted agents. For more information on Fuzzy Logic, see [AIGPW02], [GEN5] and [GEMS1].

- Dempster-Shafer: An agent type that has been proposed as an alternative to using Bayesian Networks in [AIGPW02]. The Dempster-Shafer Theory is based on Belief Theory. It uses *credibility* (How much evidence explicitly supports an assertion?) and *plausibility* (How much evidence fails to discredit an assertion?) to form *belief*, which is a subjective measure between credibility and plausibility. The two types of evidence allows to model the agent in a humanlike manner. The agent can be modeled pessimistic, e.g by using credibility mostly, or optimistic by letting him base decisions on plausibility instead. Evidence from different sources is combined with *Dempster's Rule*, see page 361 in [AIGPW02]. The methods provided by the Dempster-Shafer Theory is used when faced with incomplete and ambiguous information from not too contradictory sources.

- Neural Networks: The network consists of several neuron-like perceptrons placed in layers. It receives some input, multiplies this input by a weight and returns 1 or 0 given that the result is greater or lower than a certain defined threshold. The threshold is often given by a function, like the Sigmoid-function. Learning is done by adjusting the weights. This is done by the use of an error-function denoting how wrong the result was compared to the true result. Neural Networks is one of the first adaptable agent technologies that has reached the game industry. However only a very few games has yet been used with this technology, examples are Lionhead Studios' *Black & White* and Codemaster's *Colin McRae Rally 2* [AIGPW02]. For more information on Neural Networks, see [ML97], [AIGPW02], [GEMS1] and [GEN5].

- Bayesian Networks: A type of agent technology that works on uncertainty and conditional probability as described earlier (See Section 2.3). This agent technology has to our knowledge not yet been used in any commercial game. For more information on Bayesian Networks, see [BNDG01].

2.4.2 Choice of Adaptation

If a developer chooses to make an agent with the possibility of adaptation a series of considerations also needs to be taken into account. The main consideration is to use *Indirect* or *Direct* adaptation. Both have advantages and disadvantages.

Indirect Adaptation: This type extracts statistics from the game world, which is fast as everything is available through the engine. The agent will then choose a strategy according to the statistics among a fixed set of strategies available. This means that the AI Designer needs to design the different strategies it can choose between. The advantages of this approach is that you have easy access to the statistics and it is easy to debug and test. The main disadvantage is that you need to know what strategies it can adapt a priori and incorporate these. This means that the agent cannot discover a new strategy when shipped from factory.

Direct Adaptation: The second type of adaptation applies learning algorithms to the agents behavior itself. The only thing done prior is to inform the network about the aspects to which it should adapt. The main advantage of this approach is that the agent can develop an entirely different behavior, which might lead to a strategy that the AI Designer never though about. However the disadvantages is that this type of adaptation is highly difficult to test and debug, as agent behavior become complex and difficult to anticipate all outcomes. The second problem is how many cases must be seen in order to change behavior. There can be many aspects to test if you want to avoid certain situations that the agent must not choose.

Adaptable technologies are not often seen in commercial games, as described earlier. The games that have tried and succeeded are still few. Some of the technologies described incorporate some level of adaptation, and should offer an easier approach to this aspect. Neural Networks have already seen commercial use, Bayesian Networks also offers adaptation as an integrated part of the technology, as was described earlier. The old school technologies, such as DFA's and Fuzzy Logic does not offer adaptation, as they are monotonic functions that do not change during the course of the game. It should be possible to incorporate some sort of adaptation into these agent technologies, but all options would need to be scripted. This would indicate that Direct Adaptation would be the choice of adaptation for this type of technologies.

2.4.3 Other Considerations

Choosing to use direct or indirect adaptation is not the only choice that needs to be made. Finding a good *performance measure* can be challenging. It is also a good idea to be able to add as much *prior knowledge* as possible before shipping the game, so that the player will not need to play the game several hundred times before actually meeting an intelligent opponent. You might also run into the case that the network is only capable of finding *locally optimal behavior*, which is not a desirable property. As always when using an adaptable agent type you want to try to avoid *overfitting*. Another major problem in games is the many variables that often exist. To model the correlation of the variables without ending up with complexity problems it is important to *minimize dependencies*. Another concern when making a good agent is the problem with choosing *Explore* versus *Exploit*. When should the agent start using its knowledge and when should it search for more? Most of the methods used to allow agents to adapt also require some additional computing power. A consideration as to when this can be done without slowing the game also needs to be dealt with. These issues are discussed in John Manslow's article in [AIGPW02]. For further information and theory on many of these network consideration, see [BNDG01] and [ML97].

Why should an AI Designer choose to use Bayesian Networks instead of one of the other alternative technologies? There are several reasons for this. The choice of using Bayesian Networks instead of Fuzzy Logic or Dempster-Shafer is described in [AIGPW02].
'..the main limitation of both the Dempster-Shafer Theory and fuzzy logic relative to probabilistic reasoning is that both are built on traditional *monotonic logic*, that is, they state propositions that cannot change when new information is added to the system."

- *Paul Tozour, Ion Storm*

However there are also reasons for not choosing to use Bayesian Networks compared to Dempster-Shafer and Fuzzy Logic, namely that of finding the independence among an often large amount of variables or having to estimate conditional probabilities about often abstract concepts without prior data. This concern is expressed in [AIGPW02].
'..proving independence is often impossible, especially in systems like games where a large number of varirables (i.e., units, time, weather, player input, etc.) influence each other in unpredictable ways. Computing reasonable estimates of conditional probabilities is hardly easier: a medical research specialist might be able to access the case histories of thousands of patients.., but game developers rarely have such a massive base of prior data with which to work."

- *Francois Dominic Laramee*

The choice is not always clearcut, and therefore a hybrid solution is used. In the example of Black & White several types of agents where used for different areas [AIGPW02]. It is, as mentioned earlier, however difficult to advice which hybrids to use, as they are often made to solve a specific problem, not as a general approach. The statements made from Francois Dominic Laramee and Paul Tozour makes it clear that there is both upsides and downsides in Bayesian Networks when compared to their alternatives. It can be difficult to assess which side dominates. In the following an example game, *Caveman's Struggle of Life*, will be introduced. The purpose of this example is to show the upsides and downsides of using Bayesian Networks from a practical perspective, offering an important aspect missing in many introductions of the technology.

2.5 Related Work

Unfortunately there is not much related work to be found in the industry. In the book, AI Game Programming Wisdom ([AIGPW02]), the theory is explained and the possible advantages are explained. The same can be seen in Scott M. Thomson article "AI Uncertainty" on GameDev.net [AIUGDN99]. They state that it can be difficult to ascertain the dependencies, as they are often found in large numbers in games. This is not enough to make an assessment to use it in a company, that tries to stay in the race of game making. They need to see it work, and see which consequences it has on the rest of the game. Consequences like "does it take much more time to

develop?" or "Is it easier to maintain or debug?" in comparison to how much it adds to the game - the difference it makes compared to their current technology of preference.
However, work has already been done in a project at Aalborg University to show how well it works in the commercial success-game, Unreal Tournament^{TM}. In this project an influence diagram is used to control the strategies chosen by an agent in the game, and tests are run to show the effect. The interesting part of this project is that the effect of adaptation can be monitored. To see the results of this project, see [RBBN02].
AI Designers are constantly looking for new technologies to incorporate, and their focus over the last couple of years has been mainly on Neural Networks. It can appear somewhat strange to see the great interest in Neural Networks, as both [AIGPW02] and [GEMS1] are clear examples of. This technology does not appear to offer what the AI Designers require in a new technology. It is difficult to model exactly, predict effects (especially longterm) and debug. Still this method has seen commercial use in highly rewarded games, as mentioned earlier. Therefore it may seem strange that there is not more to be found concerning Bayesian Networks from the industry.
In the academic society however Bayesian Networks is a well-accepted agent technology, and also seen as having many uses in areas of Artificial Intelligence, where uncertainty is involved, including games. In [UIAI13] some of the research done in this field can be seen.
It is important to notice the differences between the primary concerns of the game industry and the academic society. Whereas the academic society is focused on discovering new knowledge, no matter how practical it may prove, the game industry is instead focused on its practical use, not what new knowledge they may ascertain.
We hope that this report will add something to this scarce area from both from the point of view of the academic society *and* the game industry.

2.5.1 Summary of Analysis

Now many of the aspects that will be the focus of this report have been introduced. An overview of the overall subject and the purpose should now be clear to the reader. The analysis of the problem of introducing new technology into the game development industry, the different alternatives to Bayesian Networks and related work done in this area should offer a basis for aspects discussed in this project. With this background established the example game, that will be the basis of the practical test of Bayesian Network, will be introduced and analyzed.

Chapter 3

Solving The Problem

In order to show the strengths and weaknesses of Bayesian Networks we need to solve a problem using the technology. We have chosen to invent a little game called *Caveman's Struggle of Life*. It is essentially two cavemen trying to capture a sheep, that is faster than they are. If they do not capture it within a certain amount of time (turns) they will starve to death. They will almost never capture the sheep unless they predict where it is going to flee to, or try to lure it into a corner. The rules of the game is explained in the following section. Many aspects can be examined using such an example game, and it is necessary to decide what aspects we want to explore. However, it is also vital that the change in parameters is kept under measure. To ensure this we have described a step-by-step approach, that explains the steps to be taken and the aspect that will then be examined. After this we will describe the algorithms we have chosen, that fulfill the requirements we have set. The cavemen is made using a simple, approach, and the sheep a slightly smarter method.

3.1 The Game "Caveman's Struggle of Life"

The rules for the game can be seen in the list below.

1. The Game is on a $N \times N$ board
2. There are two types of players on the board
3. At most one player must be on each tile
4. There are two cavemen, which is the first type of player
5. There is one sheep, which is the second type of player
6. When a caveman is placed on a tile next to a tile with a sheep on the sheep is captured by the caveman.
7. When the sheep is captured the cavemen have won the game and the game ends.[1]
8. The sheep can move at most two tiles at the time, but only diagonally, both steps in the same direction.

[1]In implementation we choose to let the sheep win if it can escape capture for 200 turns. - The cavemen will starve to death.

9. The cavemen can move one tile at the time, but only vertical and horizontal.
10. The sheep cannot be captured diagonally by a caveman.
11. A player can choose not to move.
12. All players make their move at the same time.

In our specific case we have decided to keep the board size at 8×8 tiles, since we have found that this is a very appropriate size, allowing the game to run for a reasonable time, while still having a reasonable size for the purpose of modelling.

3.2 Step-By-Step Approach

In order to train the Bayesian Networks it is essential that some randomness is used in the game, in order to let the network play different situations. In this game we have chosen to let the three actors, the sheep and the two cavemen each be placed on a random location in the beginning of each game. This will cause some short games, as a caveman could start right beside a sheep. A thing worth considering is if this particular randomness chosen will allow for enough different situations. All possible situations should be possible to reach using random start-locations, as there is no history, no memory of past positions. This means that every situation is independent of its past.
The following is the different steps that will be used in order to demonstrate and test certain aspects. For each new step the change in parameter from the last step is explained.

- The first step
  - A sheep that works with a simple deterministic algorithm, that tries to evade the cavemen and avoid getting trapped in a corner.
  - Two cavemen that works with a simple deterministic algorithm, that always tries to follow and capture the sheep
  - The board is an 8×8 board with no obstacles.
  - All positions of other players are available to all others.
- The second step
  - One of the cavemen uses a Bayesian Network that tries to predict the sheep's next move and thereby learn the sheep's strategy.
- The third step
  - The caveman with the Bayesian Network also tries to predict the other caveman's next move, and thereby learn his strategy.
- The fourth step
  - The caveman with the Bayesian Network will be adaptable and try to reevaluate its estimated values to changing environments (e.g. a change in the sheep's strategy).
- The fifth step

  - The second caveman gets his Bayesian Network.
  - They will try to learn the sheep's movement and their partner's movement.

- The sixth step

  - Communication is introduced into the networks. One caveman can give information/evidence such as coordinates he wishes his partner to move to, etc.

These steps are subgoals of the overall goal.

3.3 Algorithms

There are two main algorithms. The primitive caveman, that does not use a Bayesian Network, but just follow a simple deterministic algorithm. The second is the sheep that stays on the grass eating, unless someone enters its "radius", which is the area in which it feels threatened. If it feels threatened it will try to evade the caveman and corners until finding a place where it may continue to eat in peace. To make the game more challenging and realistic we chose to let the sheep escape if it can survive for 200 turns. If this amount of turns is reached in a game the two caveman will have starved to death. Another reason for doing this is that to begin with everything in the game is completely deterministic. The sheep will do the same thing in the same situation always, and the same for the simple caveman algorithm, described in the next section. This means that if they reach a board configuration within a single game which they have already been in, the game will loop forever.

Caveman's Simple Algorithm

The method the simple caveman uses in order to catch the sheep is the one shown in Figure 3.1.

```
For each possible move
  Calculate the distance to sheep
Select shortest distance to sheep after move, and execute.
```

Figure 3.1: Algorithm for the caveman.

This is a very simple algorithm which has some problems. It does not take into consideration that it may be at better idea to actually utilize the other caveman in order to corner the sheep nor the fact that the sheep can choose to jump over the caveman. It simply assumes that the sheep is stupid, and will not move as well. However this algorithm will suffice for the purpose of demonstration.

Sheep's Algorithm

The sheep's strategy is slightly more complicated. The reason for this is mostly that it has to take more things into consideration in order to avoid being caught. It has to take into account both cavemen as well as not getting caught in a corner, which would leave it defense less. The algorithm can be seen in Figure 3.2.

```
IF there is no cavemen within radius (2 moves of a caveman)
       do not move
       ELSE for each possible move
        calculate the sum of: distance to nearest corner,
        distance to caveman 1 and distance to caveman 2,
          for each possible move.
            Choose the move with the highest sum.
```

Figure 3.2: Algorithm for the sheep.

In the algorithm the variable radius is used for the sheep. It is as described the radius that holds those tiles on which a caveman needs one more move to capture the sheep. These tiles are shown below.

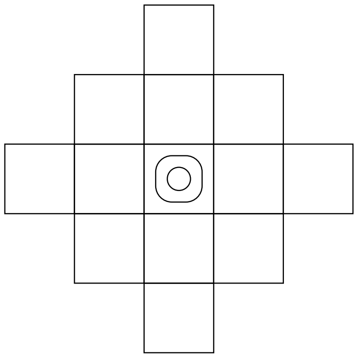


Figure 3.3: The tiles, that are considered within radius of the sheep, that can be seen in the center. It is designed so that outside this radius the caveman needs at least two turns to capture the sheep.

This method gives the sheep the possibility of avoiding capture, while still keeping the method fairly simple. If it should be necessary to change how the sheep reacts, there are a number of methods. The first one would be to change the radius within which the sheep senses cavemen. The other way is to change how the different distances is weighted. For example it could be interesting to change the weighting of either corners or the cavemen.
Now the rules, the steps and the algorithms are described everything is ready for the implementation.

3.4 Implementation

In this section we are going to explain how main components in the game is implemented. The first part we are going to explain is the implementation of is the game itself, while the other part is used for explaining the implementation of the agent using a Bayesian Network.

3.4.1 Implementation of the game

The game has been made to be as simple as possible, since it was not a main part of the objective of this report. The main idea is that we have a list of cavemen, each with an ID, and a list of sheep who also have an ID. These are used to perform the placement of the players on the board, and once this is done each of the players are asked to perform a choice as to where he wants to go. Once all players have given their answer the players are moved to their target, and board configuration is tested for whether it is a winning configuration or not. If the game is not won the players are once again asked to perform an action, and so on. If the game has not been won in 200 turns the games is ended, since once the game has been running for this long a time it is most likely that the game has entered a loop in the game configurations. The game was programmed in C++, for a number of reasons. First of all it is a language we know very well, and Hugin has an API which lets us interface to it through C++. Apart from the basic game we also made a user interface which could be used to show how the game was running. This is useful both when debugging the game, since it is easier to see how the game progresses, and when demonstrating how the game works. It was implemented using SDL[2]. A screenshot of the user interface can be seen in Figure 3.4.

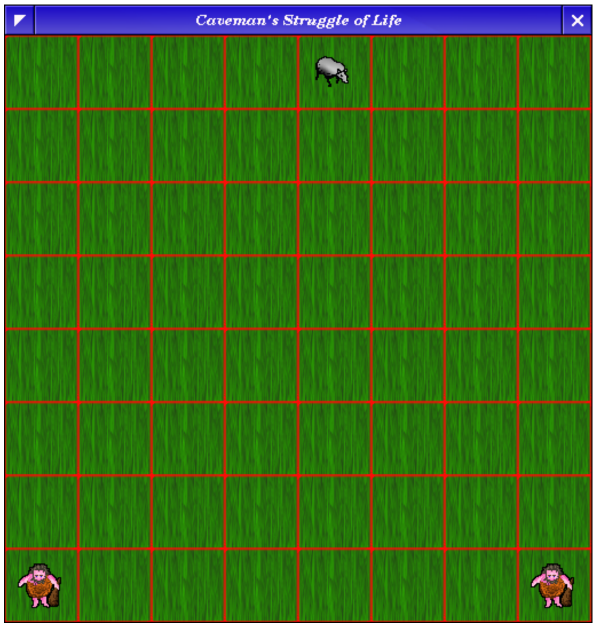


Figure 3.4: The “Caveman’s Struggle of Life” user interface

The cavemen are made as a base class (Caveman) which does not perform any actions, but is only used as an interface for creating cavemen. The main caveman classes are:

- Simple (the non-adaptive agent), which uses the strategy simply moving to the tile which has the shortest distance to the sheep. This is done by evaluating which moves are possible, and then calculating the distance to the sheep for each of these moves, finally selecting the one with the shortest distance. The source for this caveman can be seen in Appendix A.2. When referencing the use of this agent later on we will call it “DFA-agent”.
- Learner (the agent using a Bayesian Network), which uses the strategy explained in Section 3.4.3. This agent will be referenced as “BN-agent”.

Since there is only one kind of sheep there is only one sheep class and no interface. However this should be simple to change should there be a need for this in the future. Apart from the algorithm which we explained in Section 3.3, we have also included the source for the sheep in Appendix A.1.

[2]http://www.libsdl.org

3.4.2 Training the Bayesian Network

Once the game itself was implemented the next objective was to find a way of training the network. This is done by capturing the agents' behavior from one turn to the next throughout many thousands of turns. When the learning is finished it should be tested by using the trained network to predict the behavior of the agent. For training we have decided to use EM learning, since this is appropriate given that we a have easy access to large amounts of training data. The learning is done by making a data set of the samples we have gained, and then afterwards running training with the dataset. The training was done through the Hugin user interface, where there is the possibility of loading a training set and running EM learning on the model using it.
One thing which should be noted was that when we started updating the learning (when we tested adaptability) we found that we had to start with a model with the experience count set to one for all configurations before running the first set of EM learning cases, thus adding prior knowledge. This is to ensure the original EM learning would allow for the difference in training cases from the original set to the new set. The problem is that if the dataset does not contain a certain configuration X, the probability of this configuration will become zero, $P(X) = 0$. This is a problem if a new training case contains the configuration X, since now the case has probability 0 and is impossible according to the network.

3.4.3 Implementation of the AI

For coupling the Bayesian Network together with the caveman agents in the code a set of input and output was required. The input for the Bayesian Network is given by the agent code, and the Bayesian Network returns the new probabilities. These are used to make an output choice. The agent gets this information by analyzing the game board. It will use the following Input/Output:

- Input : Location of Sheep, Location of Fellow Caveman, Own Location.
- Output : Move to choose (North, South, East, West, Stay)

In the case that more than one of the agents tries to move to the same tile the first agent in the list would be allowed to make the move. The list is the two cavemen, first and second, and then the sheep.

Lookahead

If the cavemen are to utilize their Bayesian Networks, that now is capable of predicting the movement of the other actors in the game, a lookahead is needed. The idea behind the lookahead is that you use the prediction of the other actors for each possible move you may choose. If you find a winning situation within one of these possible moves you choose it, or else you try again and make prediction of your possible moves after your first series of possible moves. This can be done as many times as required, but it quickly becomes a complex task with many calculations. An example of this could be a lookahead of 3, when you have, like the cavemen, 5 possible moves in each turn. This gives you $5^3 = 125$ predictions, and for each you need to make a propagation in your Bayesian Network. The problem quickly becomes more complex. If we take a lookahead of 4, you suddenly get $5^4 = 625$. Assuming you have a large network it may take a long time for each propagation. There is of course also the other computations to take into considerations. This means that a lookahead of 2 is most likely the best suited for our game. However a test of the cost versus the effect of changing the lookahead (e.g. 1,2,3..) will be described later. Another problem with lookahead is that the error-percent is crucial. If the error-percent is 20%, which means that you predict 80% correct, in a lookahead of 3 you can have the chance of correct prediction reduced to $0.8^3 = 0.5$ on the last move. There are some methods that can be applied to reduce the complexity in calculating lookahead. Many of the methods commonly used to add ply (steps ahead) in board games can be used. A well known method, like *Alpha-Beta Pruning*, as presented by Newell, Simon and Shaw in 1958 can be used to avoid searching down branches that have already shown to be worthless, like one where the sheep gets out of reach again. Alpha-Beta Pruning could easily be combined with *iterated deepening*, where the results from the last ply is used when computing the next. It is important, when using such methods that considerations about *horizon* is taken into account. The horizon effect is that a situation which looks bad seen with the current lookahead, might prove good if only the lookahead went a bit further. This problem is always present when using lookahead, that stops before the entire game-tree is examined. For an introduction to Alpha-Beta Pruning used in Betamax and Negamax algorithm and a discussion of possible optimizations, see Jan Svarovsky's article in [GEMS1].

Optimization

Apart from this the model we used contained a problem. The junction tree was very large. This made propagation very slow, and thus both learning and propagation took a long time. Thus we found that it was useful to enable Hugin's junction tree optimization. The main reason not to use this when not needed is that the optimization takes more time than the standard compilation. However in our case we only needed to compile the network once, and thus it was not a problem. This optimization resulted in a much smaller junction tree, namely going from a junction tree size of 16 million to 5 million. This made propagation a faster action to perform, but not more than a propagation still was a rather time consuming action. Another optimization which came to our attention is Hugin's option of compressing the tables. This option gives a speedup in the cases where a network has a lot of zeroes in the tables. This is the case in our network. However, this was discovered very late in the project and was not tested.

3.4.4 Summary of Implementation

There are always many considerations during the implementation. After carefully designing rules, steps and algorithms to use, there is always a lot of other small issues that appear, when doing actual implementation. As our focus have been on the Bayesian Network we have chosen to discuss those issues that concerned Bayesian Networks. As always there were other problems during implementation, but such problems are not the focus of this report. Considerations such as Lookahead and Optimization are valuable, not only to our implementation, but also to readers considering implementation of a Bayesian Network. However implementing the methods to use Bayesian Networks in code is not the only problem - the perhaps largest one is to model the problem using Bayesian Network methods. This aspect is discussed in the following chapter.

Chapter 4

Modeling

One of the vital parts of using a Bayesian Network is modeling. There are many critical considerations to be done, when selecting a model. Considerations concerning *size, bias, complexity, ambiguity, correctness and conditional independence* are important to deal with. As we develop the two main models, the model for predicting the sheep and the model for predicting the fellow caveman, we will describe the advantages and disadvantages of the choice we make. These considerations should be dealt with, if the modeling part leads to such a situation. Another problem is often that the strengths, and most importantly, the weaknesses of a model is first seen when the model is used. As this chapter will show, modeling is not always a straightforward job, but might require a considerable amount of time and effort.

4.1 Sheep's Strategy

Several models were considered for the model of the sheep in the caveman's Bayesian Network. There are two aspects which have to be dealt with when first choosing a model, *Size versus Bias*, size being the the size of the CPT[1] and bias being the amount of bias in the model.

- Naïve Sheep Model: In this model, which can be seen in Figure 4.1, we have decided to model how a caveman should interpret the sheeps actions, given the state of the board. This model allows for the caveman to learn the strategy of any sheep which uses its own placement together with the placement of its opponents. The model says that the sheep has to make a decision from a list of 9 possible moves, based on the exact placement of itself as well as the cavemen. The problem with this approach is that this leads to a very large CPT ($9 * 8^6 = 2.3$ million) which makes this model useless.

- Biased Sheep Model: This model, shown in Figure 4.2, takes a different approach, instead of being completely unbiased, it takes in as much bias as possible, thus modelling how the deterministic sheep makes its decisions. This approach has two major problems. First of all this leaves us with a model which may not work if the algorithm for the sheep changes, secondly this model still has rather large tables. The model uses the current placement of the sheep and cavemen, in order to calculate the distance to each caveman and the corner, for each possible move. After this has been done the different distances are collected to a weight for each possible move. Then the distances to both caveman from the sheeps position is calculated

[1]Conditional Probability Table

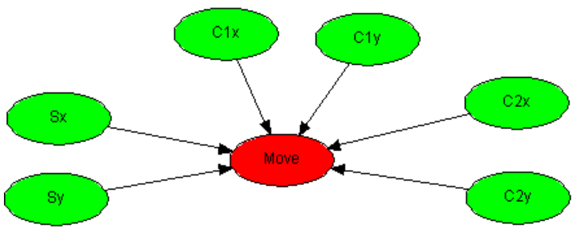


Figure 4.1: The simple model of a sheep. It has a single prediction node, which is affected by the sheep's position and the positions of both cavemen. The green variables are observable. The red variable needs to be trained.

and if both are greater than two the *Choice* node chooses to "Stay", else the action with the greatest weight is chosen.

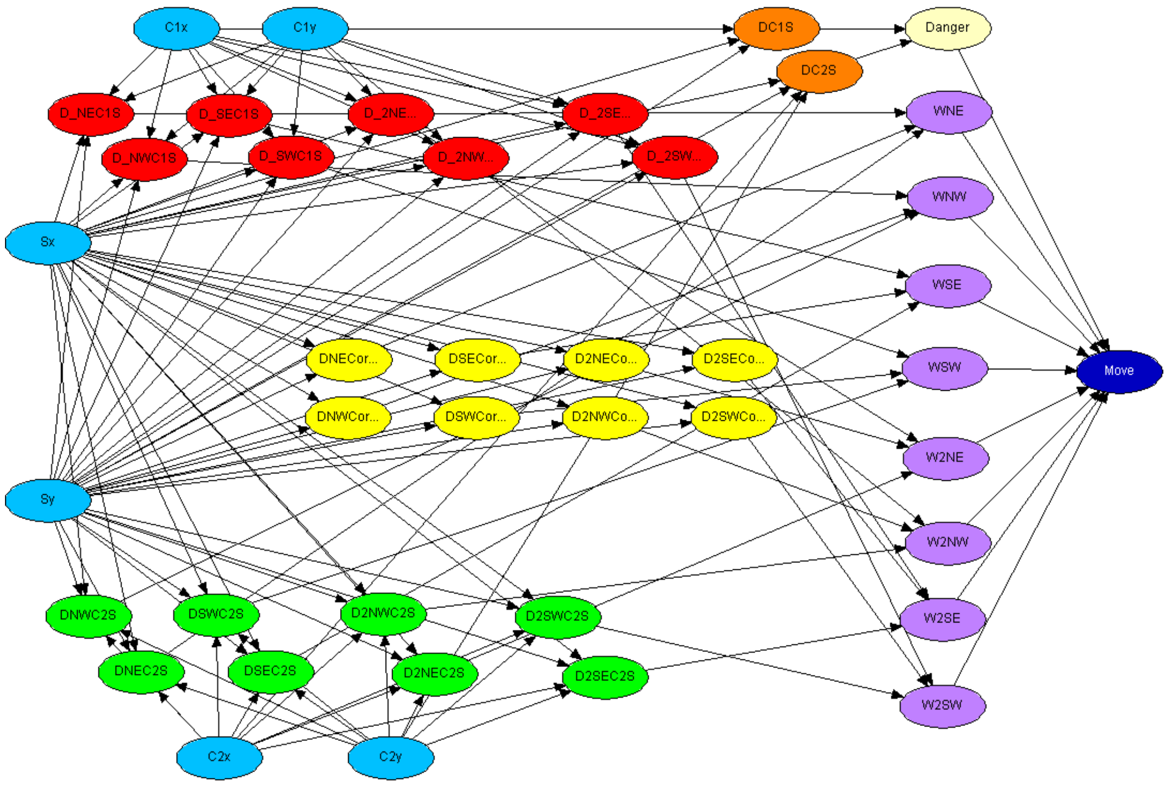


Figure 4.2: Model of sheep's actual behavior. This is a model of how the sheep actually processes its own location combined with the location of the cavemen. This is done calculating distances after each possible move and summing up before deciding. The light blue variables are observable. The green, red, yellow and orange variables are scripted. The beige, dark blue and purple are variables that need to be trained.

- Less Biased Model: This model as shown in Figure 4.3 takes a look at the locations of the three players. It assumes independence among X and Y. It finds the correlation between the coordinates of the players and the possible directions of the sheep. As the rules define the directions in which the sheep can move this is not bias. To find out if the sheep moves zero (stay), one or two steps in the direction found from the vectors, the model uses the distances to the cavemen and the nearest corner. This is not correct according to the Biased Sheep Model, described earlier, however it should be able to find the radius from which the sheep choses to react. The model overall assumes, that given we can assume independence among X and Y directions, the sheep makes it choice based on a direction chosen from the locations of the two cavemen, and the step according to the distances to the three threats of the two cavemen and the nearest corner. The CPT for this model is much smaller than for the naïve model, and the model is not as biased as the other model. However this model proved not to be sufficient, since it could not predict the cases very well.

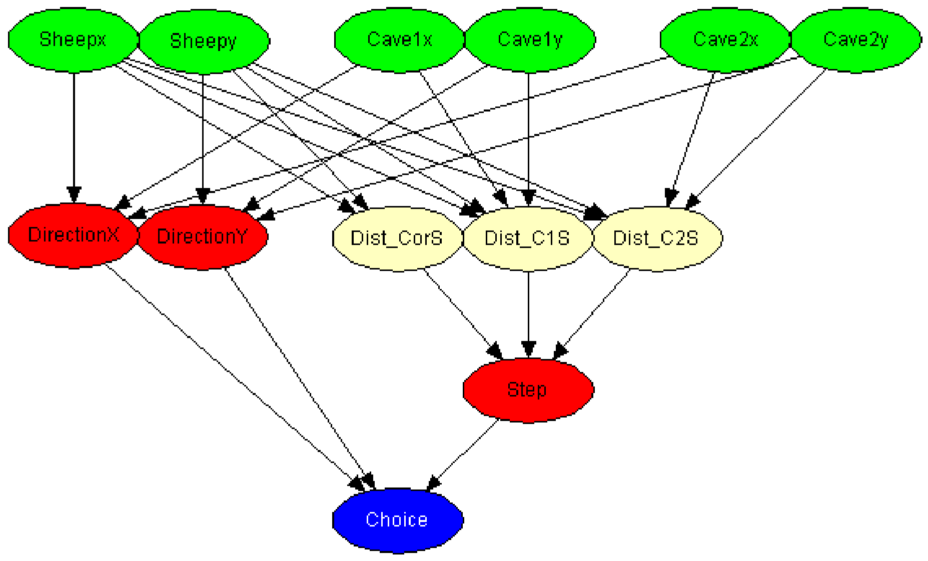


Figure 4.3: This is a model which incorporates little bias, assuming that it is possible to split the decision between X and Y directions as well as how far the sheep should go. How far the sheep moves depends on the distance to the two cavemen and to the nearest corner. The green variables are observable. The beige and blue variables are scripted. The red variables are to be trained.

- External Function: The next attempt was done with the purpose of having as little bias as possible in the model and being able to adapt to a larger board size. The assumptions are:
  - The sheep can move in its 8 possible moves, or stay.
  - Stays if nothing gives any gain
  - It wants to stay as far away from the cavemen as possible (distance) in order to avoid being captured.
  - The sheep does not want to lose possible move options, when selecting a place to move to (avoiding corners and edges)

The CPT of the model, which can be seen in Figure 4.4, in the node *Move* became $3^8 * 9$ (59049). This was acceptable, but for this model to work a function needed to be constructed. This function would model the function for finding an optimal "trap" for the sheep, concerning

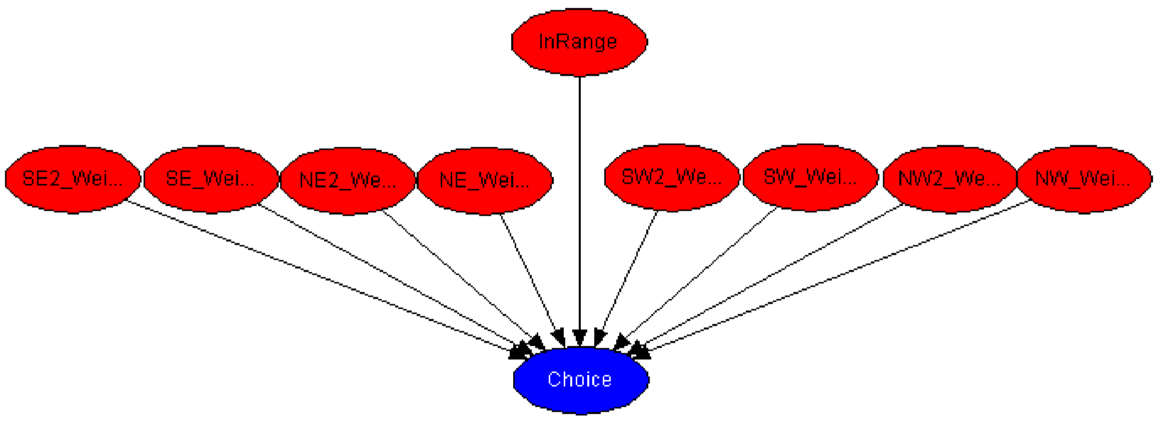


Figure 4.4: Model using external evaluation. The red variables are made observable by an external function in the code, that derives them. The blue variable is to be trained, which should be easy as it would only need to choose the largest.

corners and walls. We assumed that it would be something like a paraboloid, See Figure 4.5, as the paraboloid shows both corners and edges. This function would be made in the coding of the game and feed the Bayesian Network with weights within a given set of intervals, that measured the “trap-value” of the possible moves. This however, proved to be to difficult, as the way the sheep is coded is very difficult to reproduce precisely mathematically. If this function did not work correctly for every situation, it would feed the Bayesian Network with ambiguous “scores” for the sheep’s position. If the function were to be made correct it would mean that everything from the sheep’s algorithm was moved into another function that would just output a score to the Bayesian Network which would only have to find largest score. This would not be much of a breakthrough for proving the strength of Bayesian Networks, especially as the model would actually be as biased as possible, just outside the network.

- Final Model: This model, seen in Figure 4.6, is based on the "Less Biased Model", however the distance to nearest corner is no longer used. Instead the sheeps position is connected to step. Also the sheep’s x-coordinate affects y direction and vice-versa. The reason for this is that the original model lacked support for the edges of the map. This made the model insufficient. The changes take care of both the corners as well as the the edges of the map, thus minimizing the problem of predicting the sheep’s actions when near to the corners.

4.1.1 Problems and Solutions

The first model we tried to learn proved to have too many ambiguous situations. Some things could not be expressed through the model, such as the correlation between the placement of the nearest corner and the choice of direction to go to, or the correlation for making the step-size and the distances. The main problem with the model became apparent through training, as it became obvious that it could not make reasonable predictions, and that the main reason was probably caused by too strong assumptions such as that the exact location of the sheep was not needed in predicting how far the sheep would move.

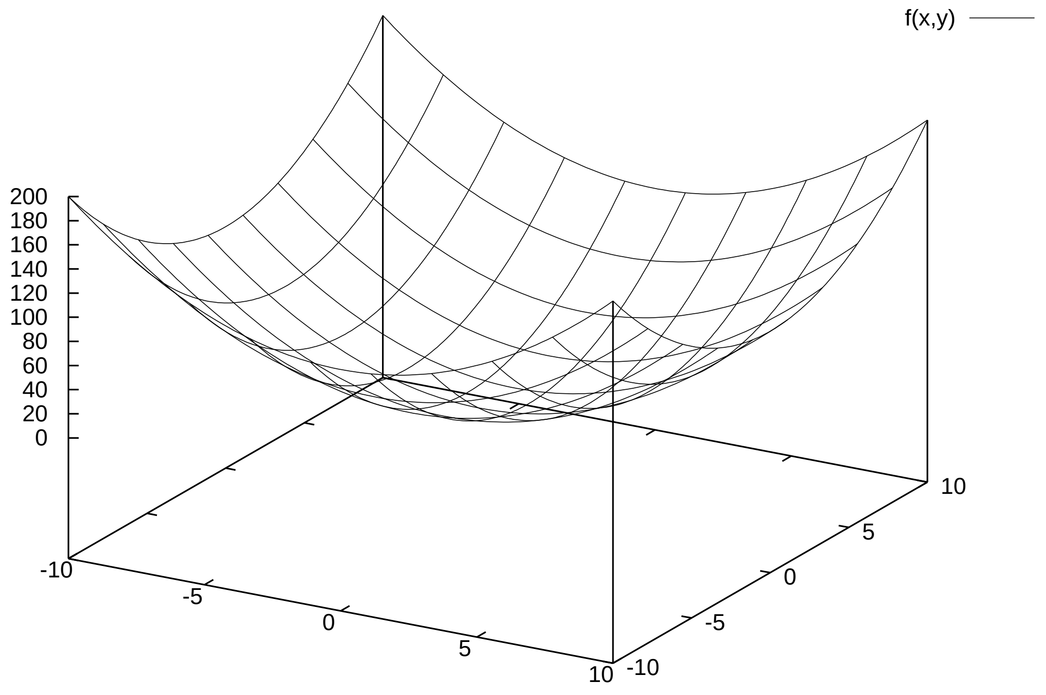


Figure 4.5: The graph we first thought was enough to describe the problem. It is a function of the trap-values for the sheep, which it will try to avoid as much as possible. The larger the value the higher the danger.

4.1.2 Discussion of the Final Version

The model we ended up using is the one shown in Figure 4.6. This model makes the assumption that we can split the decision into three parts which can then be joined together using a deterministic function. The basic idea is the X and Y direction the sheep moves in are independent. The fact that the sheep has decided to go north has no bearing for whether it decides to go east as well. This may prove to be a strong assumption. We also assume that the directions are independent of how long the sheep decides to go.

In the model we have used three nodes containing scripted variables. There are the two distance measures *Dist_C1S* (distance between sheep and the first caveman) and *Dist_C2S* (distance between sheep and the other caveman). These are scripted using the a Hugin expression, such that we will not have to insert evidence on them when making a prediction. The expression for *Dist_C1S* is as seen in Figure 4.7, and calculates the New Yorker distance. The expression for *Dist_C2S* is similar.

```
abs (Sheepx - Cave1x) + abs (Sheepy - Cave1y)
```

Figure 4.7: Calculating the New Yorker distance in Hugin Expressions.

The last node which uses an expression is the *Choice* node. The script is used for collecting the

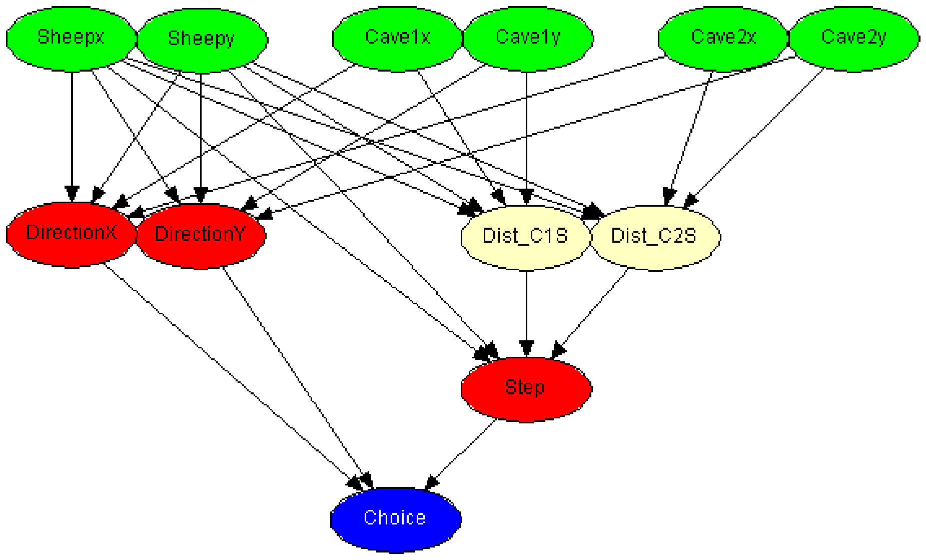


Figure 4.6: This is the final model, which is a modified version of Figure 4.3. The main modication lies in the fact that the distance to nearest corner is no longer used, and instead we take the location of the sheep thus gaining knowledge of distance to edges as well. The other difference is that the direction nodes now have the position of the sheep added. Recall the green variables are observable, beige and blue variables are scripted through expressions and red variables are to be trained.

predictions from the two *Direction* variables and the *Step* variable, thus making the final prediction. The expression can be seen in Figure 4.8.

As mentioned we made a number of assumptions when creating this model. Among others, the assumption that it is possible to split the movement into subcomponents. This may result in the model not being able to fully fit the action hypothesis space. This means that the model may not be able to learn and predict all cases correctly. The problem becomes apparent in the *Step* variable which only takes the distance to the cavemen into account, and not their actual locations. Since it is possible for a single sheep location to have more than one set of caveman locations having the same distances. Thus it may be possible that a single configuration of *Step*'s parents have several possible steps.

The model also has a problem when it comes to adaptation. The *Step* table is very large (43k), thus it will take a lot of training cases in order to adapt to a new sheep strategy. The Direction tables are not quite as large (8k). However they are still too large to be practical. Another problem is they do not get evidence when the sheep stay still, and thus they do not learning from all cases.

4.2 Fellow Caveman's Strategy

For the caveman's model of the other caveman, two models were constructed. One for predicting the caveman using the simple algorithm and one for when he will be using a Bayesian Network himself.

```
if (Step == "Stay", "Stay",
  if (Step == "1",
    if (DirectionX == "East",
      if (DirectionY == "North",
        "NE",
        "SE"),
      if (DirectionY == "North",
        "NW",
        "SW")),
    if (DirectionX == "East",
      if (DirectionY == "North",
         "2NE",
         "2SE"),
      if (DirectionY == "North",
         "2NW",
         "2SW")
    )
  )
)
```

Figure 4.8: The Hugin Expression for *Choice*, which combines *DirectionX*, *DirectionY* and *Step* into the final choice of the sheep.

4.2.1 Problems and Solutions

The model for the simple caveman was easier to construct, as it was made as a complete na ïve model, which can be seen on Figure 4.9. It merely found the caveman's strategy from the sheep's position and the caveman's position. This has a great deal of bias as well, since it assumes that the caveman is stupid and does not consider the other caveman's position only his own and the sheep's position. This model would not be enough when the second caveman starts using a Bayesian Network as well. Then both cavemen's positions needs to be taken into account. Then the same problem occurs. You need a model that uses all three actor's positions. However, as we already have a model that predicts the sheep's movement given all three actor's positions, this can be utilized. If it is assumed that the second caveman uses the same strategy of predicting the sheep as the first caveman (which is a fair assumption as the networks will be a copy of each other), it is possible to take advantage of this. Then you only need to see the correlation between your prediction of the sheep's movement and the second caveman's position.

4.2.2 Two networks cooperating

Once the adaptive agent has been completed it would be interesting to try and let both cavemen be adaptive. This requires that the model used for predicting the other caveman is changed to one, which properly models the caveman.
Once this has been changed, it is necessary to give the caveman time to learn the new strategy, since it probably will find a different strategy than the simple caveman uses.

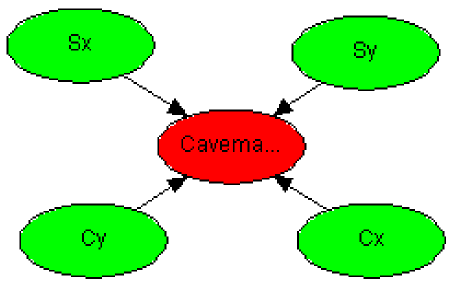


Figure 4.9: The model for the second caveman. A model which is in fact the naïve model. The green variables are observable and the red is the variable that should be learned. *Sx* and *Sy* are the two components of the sheep's position while *Cx* and *Cy* are the two components of the the caveman's position.

A problem arises when the prediction of the caveman may change how the caveman moves, and thus the model which the other caveman holds is no longer valid. Thus the strategy may continuously change, without finding a final equilibrium.

4.3 The Final Network

In this section we are going to describe how the prediction of the model for the caveman can be modified so that it is should be possible to learn the strategy of the caveman which uses a Bayesian Network.

4.3.1 Discussion of the Final Network

One model we have considered as a possibility when it comes to predicting the learning caveman, is the one shown in Figure 4.10. The idea is to use the knowledge of where we predict the sheep to go as input for predicting the other caveman's move. The reason is that the other caveman also uses, where the sheep is predicted to go, in his decision of movement. It should however be noted that the actual model used should not be one large model, but rather two models, where the model for the caveman uses the output of the sheep model as input.

The main problem is when the sheep model makes an incorrect prediction, and thus in theory give a wrong prediction of the caveman. However, if we assume that the prediction of the sheep made by both cavemen is similar, this should not be a problem, as the caveman's actual move will be made based on the wrong prediction.

The other problem with this approach is that the CPT for the caveman model is large (36k), and thus a lot of training cases are needed before the model can predict the caveman.

Apart from these new problems this model also has the same problems as the final sheep model, thus in total this approach has more problems than the sheep model alone.

One problem that the model seem to contain some ambiguity. This is clearly seen in *Step*, where the new yorker distances offers the possibility of ambiguous situations. An example of this can be seen on Figure 4.11, where the possible positions of a caveman with a new yorker distance of 2 is shown.

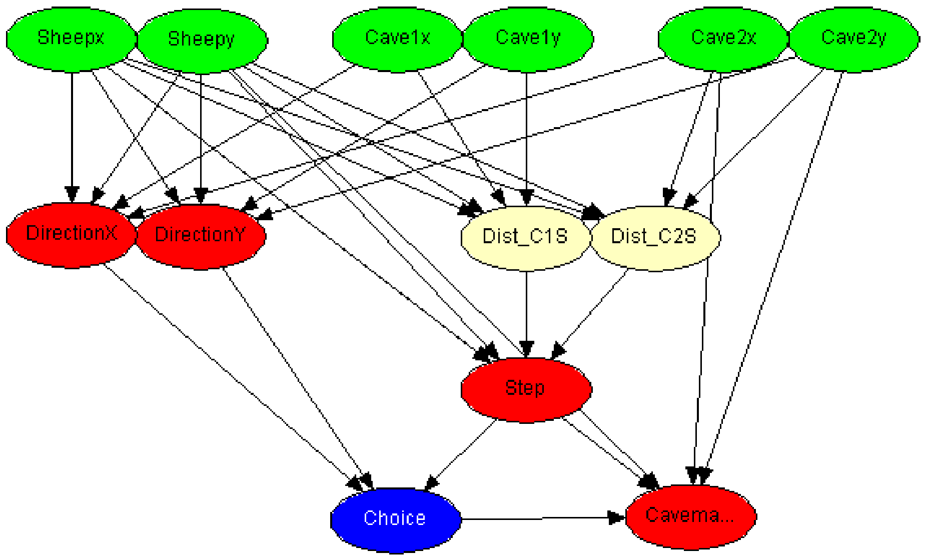


Figure 4.10: Combined Caveman and Sheep model. We assume that our fellow caveman uses the same method to predict the sheep. This means that if we know his location, the sheep's location and our guess of what choice the sheep makes we should be able to learn his strategy from this. The green variables are observable, beige and blue variables are scripted and red variables are to be learned.

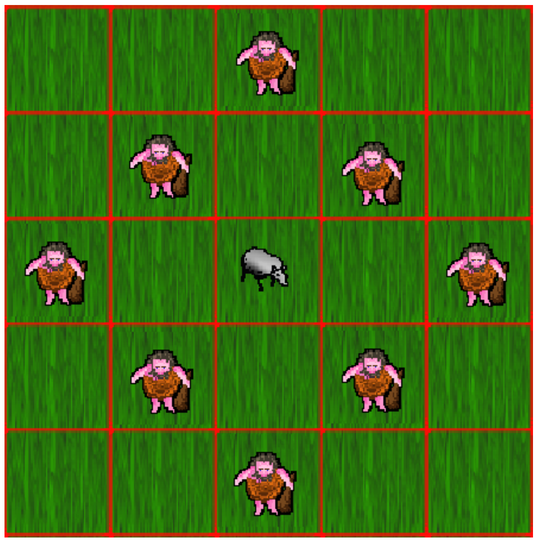


Figure 4.11: Ambiguous positions for a caveman with new yorker distance of 2 to the sheep.

4.3.2 Changing Environments

One aspect which should be taken into consideration when creating a model for an AI, is the ability to adapt to a new environment. Perhaps the strategy of the player modeled changes, and in that case it may prove necessary to learn the new strategy before the model once again will be sufficient. In order to make this adaptation feasible it should not be necessary with too many new training cases in order to learn the new parts of the game. This implies that the tables which have to be learned, should not be to large, since the larger the table the more training cases are needed.
In our model we have three tables that must be trained. The first two, *DirectionX* and *DirectionY*, have the smallest tables (8k), however these tables are still very large. They also have the problem, that not every case in the game ends in these two actually being trained. Whenever the sheep decides to stay, there is no training for these two. The table *Step* has a different problem. This table is trained in every case of the game. However, the table is extremely large (43k), and thus requires a lot of training cases. This could imply that the model has a problem when it comes to adapting to the environment.
There may be situations in which slightly fewer cases are needed. For example if the new strategy simply is to let the sheep have a different alertness radius. Then only the cases where the new radius is used needs to be learned.

4.3.3 Possible Optimizations

Even though the network is now ready to face the tests, it is important to keep possible optimizations in mind. For our purpose the game does not require a decision to made within a time limit. In most games there will be some sort of limit on the amount of time the decision function can use. As we have already discussed, the sizes of the tables are rather large considering the simple game we are using. This prevents quick adaptation. There is a simple way to reduce the table size, but with a loss of precision. The possible values of the input nodes (the positions of the sheep and the two cavemen) is currently complete. These values could be gathered in larger intervals, for example $1-3$, $4-6$ and $7-8$. The cost of this is loss of precision and often additional ambiguity. The model will become less precise, but faster. Before this is done it is essential that careful considerations are done as to how precise compared to how fast you need the model to be. A compromise between these two should then be found and added if needed.
Another possible optimization is to divide the grid into larger tiles in areas where great detail is not required. The longer away, from the observer with the Bayesian Network, the less detail about the precise position is needed. This would not only give less possible tiles to take into consideration, but also allow for better scalability of the model. If we enlarged the board to a 64×64 it would practically kill the model, as the six input nodes will be eight times bigger. Then the children of these nodes will become larger, and so on - ending up with an unusable model that is far too time consuming and not adaptable. In this case the grid-optimization would offer better scalability at the cost of some precision, but as the sheep in our model does not react unless something is within its short radius this would actually be a plausible optimization. An example of a grid-model of the board can be seen on Figure 4.12.
There can be more optimizations that could be applied in exchange for precision and adaptability. As we do not yet know of how much ambiguity is present in our model we will not implement the optimizations described in this section. However, it is an important consideration, and especially in games, where everything is to be as optimal as possible.
With our models designed, and our prior considerations about it described we are ready to test the

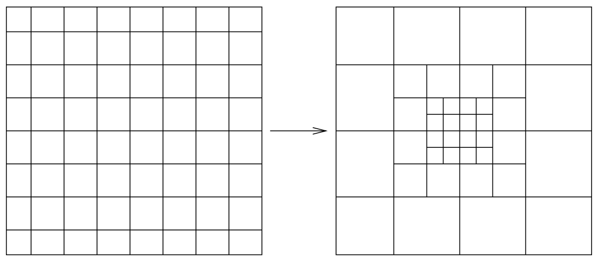


Figure 4.12: The grid-model. The board is divided into larger grids the longer away from the observer with the Bayesian Network. The original grid model is shown on the left, and the optimized on the right.

model in our implemented game. In the next chapter the strategies of the tests we have chosen to make are described, followed by the results and a discussion of these.

Chapter 5

Training And Testing The Network

The Bayesian Network is designed, and the example game is ready to be used for testing the agents. As described earlier several strategies exist for training the networks. Before testing can begin a training strategy must be chosen, ensuring efficient training of the networks. A test strategy is important to ensure that the correct aspects are tested. Considerations concerning the choice of these strategies will be described in the following.

5.1 Strategies for Training and Testing

In order to make the networks suitable for playing the example game they need to be trained with a set of training cases. Given a suitable training strategy the network should be able to learn their opponent's strategies through the training cases. However comparing a fully trained network to a network that has seen no training cases at all does not show the learning process of the Bayesian Network. To overcome this problem the network should be tested during the training. Before any tests are done a test strategy must be made, in which the different tests and their expected results will be described. This will ensure that the aspects we want to examine is tested and the change in parameter is kept under control. To avoid confusion concerning which type of agent is used we have decided to name the agent using the original, but simple deterministic strategy, *DFA-agent* and the other, using the Bayesian Network, *BN-agent*.

5.1.1 Strategy for Training

Before we start training the network we have to find an appropriate sampling method for the training. Two such methods were described in Section 2.2.1. The first method was to run a number of games using a random start location for each actor in each game until the network was trained sufficiently. The other method was to run through all board configurations in a systematic manner, saving the outcome of each, and feeding this training data to the network. We decided to try both, since as described earlier, the first method based on random start location training is, closer to how the network would be trained if it was to adapt to a human player, while the systematic approach would give a good indication of the expected result. How the two different sampling methods were adapted to train the models in our game is described in the following.

Random-based Sampling

The random-based sampling uses a random location for each player in each game. As past events do not interfere with the simple algorithms of the sheep and the caveman any given board configuration should be reachable with random start locations. However, we wanted the BN-agent's Bayesian Network, used by the caveman, to learn how different actions caused the sheep to react. This would not be accomplished by using the simple caveman algorithm already present in the game. We decided to let the caveman make a random move 20% of the time. This would allow situations where the caveman causes the sheep to make a choice it would not have taken. Another effect that needed to be taken into account - the possibility of loops in a game. This is especially a problem when all agents are deterministic. If the caveman returns to a configuration he has already been in he will continue selecting the same action in this case, thus resulting in a loop through the same configurations. Even though the caveman did his random choice 20% of the time the game could still end in loop 4 out of 5 times. The problem with the loop is that once we are through the loop once it does not add any new knowledge, when training it again. Therefore we decided to end the game, using timeout, after 16 turns, which is the maximum number of steps required to go from one end of the board to the opposite. There are 249984 possible board configurations available, but the random generator might give us the same start location more than once. To counter this effect, so that the network trained with this type of sampling could be compared with the network trained with the systematic approach, we decided to run close to double the amount of training cases on the network. In other words we will consider the network fully trained when it has seen 540000 training cases generated as described here. The training cases are to be saved in a file, and the network trained using batch-learning as described earlier.

Systematic Sampling

The reason for using the systematic approach, which may seem like cheating, is to be able to evaluate our model. The systematic approach runs through all the possible board configurations in a systematic manner, and for each configuration it makes a training case with the choice made by the other agents. This gives you perfect knowledge, and is therefore somewhat to cheat, but it also shows you the best expected result of the model. It is an interesting thing to test when compared with the training based on using random start locations. Like the random-based sampling the training cases are saved in a file, and the network trained later using batch-learning.

Training

With two sets of training data available the network can now be trained. We decided to use batch-learning instead of sequential learning. The reason for this choice was that we knew exactly how long the game would last, and how many training cases we wanted. In order to see the performance gain of adding new training cases to the network we will test the network when it has seen some of the training cases. It should be tested at different milestones. Hugin offers the feature of EM-learning directly from a file containing the training cases, which was used to train the network. One thing which should be noted concerning how we trained, is that in order to gain complete data we chose to set *DirectionX* and *DirectionY* even when *Step* was in “Stay”. This was to avoid having incomplete training data, making training take a very long time. The problem is that if the training set is not complete, then the missing variables have to be evaluated by propagating through the network, before the training can begin.

5.1.2 The Test Strategy

We need to measure how well the designed Bayesian Network model will perform in the example game. Compared with the effort used to model it, it should give some indication of the strengths and possible weaknesses of Bayesian Networks. In order to see the difference in performance between the DFA-agent used originally and the BN-agent a series of tests have been designed. For these tests the following parameters are used:

- 500 games: Each test would last for 500 games. Allowing us to measure the performance, but still within the time frame available as some of the test could prove rather time consuming.
- Timeout at 200 turns: In order to avoid loops, and thereby only seeing how well it performs within this loop, we used a timeout after 200 turns. If the cavemen do not capture the sheep within 200 turns, the sheep will win. Of course a good agent would then have as few as possible of this situation. There is however a chance that the sheep might be caught after 200 turns, but we have never experienced such a case. The maximum number of turns seen that ended without a loop, was 98. This should indicate that 200 is a relatively safe margin. It is worth noticing though that the timeout has an effect on the average length of the game. Had we chosen a timeout at 100 we would get a lower average length, but a smaller margin of error. As the average length is only used to compare cases it should not make a difference to choose 200. It should be noted that when we test for prediction correctness we only run 24 turns, since we want to minimize the effect of cycles. If we run into a cycle early on we would be getting the same few results many times, resulting in the answers for the looping cases to have very high weight in the prediction correctness.
- Random Start locations: To test different scenarios we will use random start locations for each agent in every game of the 500 games played. This could cause a problem, as the cavemen could by chance start right next to the sheep or already have the sheep lured into a corner with no escape left. This would add some cases in the tests that do not show the real performance of the agents. However, as it exists in all the tests the effect should be minimal as the difference between the tests would be the same.

The outcome is measured in *success rate* (“How many times did the sheep escape?”) and *speed* (“How many rounds were used?”). For some of the tests we will also measure the time used. This is important when comparing effect with the cost. With the tests that used Bayesian Networks we will display results at different stages in the training, allowing us to measure the progress and the amount of cases needed to perform well. With the common parameters set we will now describe the different tests we are going to conduct, and what parameter is changed compared to the last test.

- First Test: Two DFA-agents play against the sheep to find how well they perform before we start changing the parameters. We will be testing their success rate and speed as a basis of comparison with the rest of the cases.
- Second Test: The correctness of the predictions of the trained networks is tested. The network tries to predict the next move of the sheep and the fellow caveman given a board configuration in a game. This is done with both a network trained with random games (with measurements at 50000, 100000, 200000 and 540000 games), and a network trained systematically through all the possible board configurations. The reason for the last training method is to test the optimal result for the network.

- Third Test: One BN-agent and one DFA-agent tries to capture the sheep. The BN-agent uses his Bayesian Network for predicting the movement of the sheep, and uses this prediction to estimate the next placement of the sheep. The measures will be the same as in the first test. Lookahead will be used to search for captures of the sheep in more than one move.

- Fourth Test: Different Lookahead is tested in order to see the cost of using more computing on lookahead, meaning more time for prediction versus the effect of capturing the sheep more often and faster. Lookahead of 3,2 and 1 should be tested for this Cost Versus Effect. The time used to play the 500 games are measured as well as success rate and speed.

- Fifth Test: See how the BN-agent performs when the sheep's strategy is changed, like for example changing the radius and priority of dangers. How much will this effect the caveman? How many games does it take him to adapt to this change in the environment?

- Sixth Test: The second caveman is now also a BN-agent. The test will determine: if the network for predicting the caveman is sufficient or whether a new network must be constructed and trained in order to adapt to this change. We will make this assessment based on how well they are doing with respect to success rate and speed.

The tests described in our strategy will provide a measure of effectiveness in adaptability, lookahead advantages and time usage compared to the original caveman.

5.2 Test Results

In this section we will show the results of testing the use of a Bayesian Network compared to the algorithm already present, the prediction percentages reached by the network at different stages in its training, the effect and cost of using lookahead, the adaptiveness of the model we have chosen and a discussion of the results provided by these tests.

Each test is as described in the test strategy based on 500 games played, where random start locations are used to place the agents. The sheep is set to win if it can escape capture for 200 turns. We will start with the first test, which will act as a basis for comparisons later.

5.2.1 Two DFA-agents

The effectiveness of using the Bayesian Network can only be measured if we know how well the agent performs without using it. The first test measures this. We play 500 games, using random start positions, and measure how many times the sheep gets away. The test setup is two DFA-Agents using the simple algorithm described in Figure 3.1, and the sheep uses its algorithm described in Figure 3.2. This is the first test described in our test strategy in Section 5.1.2. The results of the test can be seen in Table 5.1.

Sheep Escaped	Average Length
332	136

Table 5.1: The results of the cavemen, where none is using a Bayesian Network - based on 500 games.

As the results clearly show the sheep wins most of the matches. Its algorithm is by far superior to that of the cavemen. It wins 66.4% of the games. However, as mentioneed earlier it is possible that a few of the victories could be the "easy" win situtations where the cavemen start right next to the sheep.

5.2.2 Results with random-based training

This test show the development of the correctness in the prediction percentage during training with the model trained with random sampling. These percentages should be compared to the next test, where we do the same for the systematic sampling method, which goes through all the possible configurations. This test is the first part of the second test described in Section 5.1.2. The second part can be seen in the next section. The results of the test can be seen in Table 5.2.

Turns Trained	Prediction Correctness
50000	58.5%
100000	58.1%
200000	61.0%
540000	66.4%

Table 5.2: The results of using random-based sampling in the training data - based on 500 games.

The result that is worth noticing in the table is that after training 540000 training cases it reaches 66.4%, which is quite close to the result of the network trained with systematic sampling. The result of this test shows that it is possible to train a network using other approaches than the systematic approach. It is not always possible to train a network with as many training cases as we have done in this test. This is especially the case if a human trainer, like a player, is used to monitor the training.

5.2.3 Result with systematic approach

The systematic approach goes through all the possible configurations and makes a training case for each, which is learned by the network. This will make a close to optimal set of training data, and allow us to measure what level of prediction we should be able to end up with. It will allow us to see how well the model fits the problem. The cases it predicts incorrectly will be either the result of ambiguity in the model, or the result of incorrect assumptions during the modeling. The results of the test can be seen in Table 5.3.

Turns Trained	Prediction Correctness
249984	67.5%

Table 5.3: The results of using systematic training data - based on 500 games.

As can be seen clearly from the results the model does not fit the problem particularly well. To a certain degree this was expected, as described in Section 4.1. There are two possible reasons for this. Either the model leaves room for ambiguity in some places, such as the problem we explained with the *Step* node, or some of the assumptions we have made, such as the independence between the directions and step, are too strong. Also we have found that there is a problem with way we have trained the network. As we have described we decided to set the values of *DirectionX* and *DirectionY* even in the case where they had no value (in the case of *Step* = “Stay”). When running the adaptation test we realized that this is a problem, since there are many configurations where *Step* is “Stay”, and the corresponding parent configuration of one of the Direction nodes matches a non “Stay” *Step* value. This leads to the network being trained to expect the value we have set in the case of “Stay” rather than the values which the training should have given the node. An example of how this works can be seen in Figure 5.1.

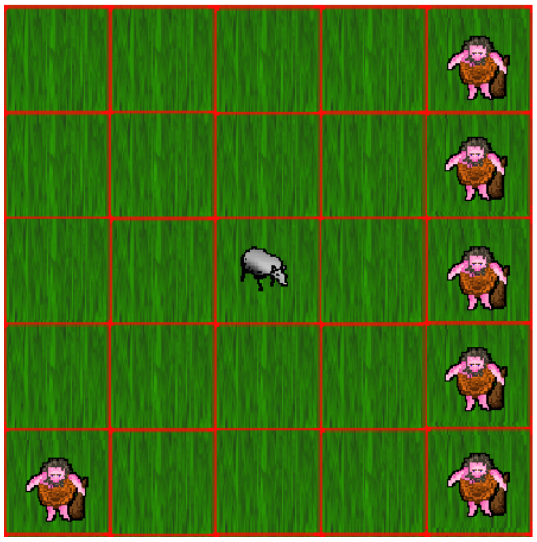


Figure 5.1: The problem with Step=“Stay” the caveman in the bottom leftmost corner is the first caveman, and the column of cavemen represents the different positions of the second caveman, in which the parent configuration of *DirectionX* is the same.

This shows how a caveman can have the same parent configuration for *DirectionX* but still both have a number of “Stay” configurations and only a single actual move. This could lead to the caveman predicting the sheep wrongly if the value set in the case of “Stay” was different from the one in the middle case. Unfortunately this problem was discovered too late for it to be possible to fix. However we have two solutions which could be used to fix the problem. The first solution is simply to accept the incomplete data. However this would be a problem since this would make training in Hugin take a very long time. The other solution is to split the network into two parts, a Step network and a Direction network, and then only train the Direction network when *Step* is not “Stay”.

5.2.4 Training prediction of the DFA-agent

Apart from the model for the sheep, we also have to train the model for the predicting the other caveman. We decided to focus the most of our tests on the training of the sheep, since this model is the most interesting one. However, we still need to make sure that the model for predicting the caveman will work sufficiently well for our goal. The method we are going to use for training the caveman model is similar to the method we used for learning the sheep model. That is, we are going to run a number of games with random starting configurations making a data file with the moves of the caveman, and then use EM-learning on the data in the file. The reason for this choice is the same as it was for the sheep, using EM-learning is faster, and since we will have to run a large number of games anyway there is no real reason for using sequential learning.

5.2.5 Testing prediction of the DFA-Agent

Since the model does not include many assumptions[1] it is expected that the model will be fairly good at predicting the caveman's moves. The method used for testing the ability to predict the caveman, is the same as the one used for testing the prediction of the sheep. That is, we run 500 games making a prediction for each move, then evaluating whether the prediction was correct. Finally we sum up all the right answers, and calculate how many answers out of the total were correct. The result can be seen in Table 5.4. The prediction of the caveman is much better than that for the sheep. This was expected since this model only makes a single assumption; namely that the actions of the caveman is independent of the actions of the other caveman. This is mostly correct, except when the other caveman is in the way, in which case the caveman cannot move in that direction. The number however, seems rather large for these cases so there may be other factors which have a role to play, however we have not been able to establish which. It is quite possible that we have used more training samples than necessary. However, we decided to use an amount of training samples where it would certainly be fully trained.

Turns Trained	Prediction Correctness
540000	92.3%

Table 5.4: Prediction correctness for the caveman model. Tested in 500 games.

5.2.6 DFA-agent and BN-agent results

Now that we have tested how well the DFA-agent performs and how well the BN-agent with his Bayesian Network predicts, it is time to see how much improvement is added when the Bayesian Network is used by the first caveman. We have chosen to use the random-based trained network, as it is a rare situation in game development to be able to define all possible configurations and train the network with these. It would also be to "cheat" somewhat. To utilize the Bayesian Network we will be using Lookahead, as described earlier. We will test the different lookahead to see the effect of increasing the lookahead. This effect should be compared to the time used to compute that lookahead. This is the third and fourth test of the tests described in Section 5.1.2. The results of the test can be seen in Table 5.5. The result for running the BN-agent with a look-ahead of zero would be the same as running an extra DFA-agent, since the part of the code, which is run when no result comes from the lookahead, is the same as the DFA-agent's algorithm. These results can seen in Table 5.1.

Lookahead	Sheep Escaped	Average Length	Time Used
1	45	33	Few Hours
2	35	34	Many Hours
3	27	30	Days

Table 5.5: The results of using lookahead - each based on 500 games.

To get an overview of the effect of lookahead the results of using a larger lookahead as compared with the performance of the cavemen is shown on Figure 5.2.

[1]the only assumption is that the move of the caveman is independent of the fellow caveman's location.

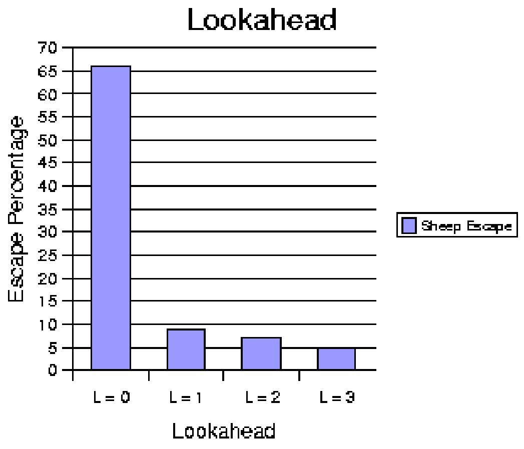


Figure 5.2: The effect of a greater lookahead in the BN-agent compared to the DFA-agent measured in the performance of the cavemen.

It is quite clear from the results that using a BN-agent with lookahead is far better than using the simple DFA-agent. However, it is not as easy to say how much lookahead is enough to use. We have chosen not to be exact about the time spent, but only show the magnitude. The choice of which lookahead to use for a specific application depends heavily on the application. If we were playing chess we could for some situations allow the use of minutes, especially if we were not playing against time as well. However, a human player would not be satisfied with waiting hours for his computer opponent to make a move each time. Chess is however turn-based, instead imagine a realtime game. Then anything needs to be faster, and lookahead might not be possible at all. It should be noticed though that we have not implemented any pruning of the possible paths in the lookahead tree. This should be done, and could reduce the time complexity considerably. Even though it would be able to compute more within the same time with such optimization it is clear that a tradeoff between *time spent* and *effect required* is necessary when choosing a lookahead approach.

5.2.7 Two BN-agents

Since we have found it too time consuming to train a network for predicting the caveman which uses Bayesian Networks, we have instead decided to test how well the original model captures the actions of the BN-agent. This was done using two BN-agents and letting them play the game 500 times, after which we summed up how many of the games ended with the sheep getting away, and how long the average game lasted, the results of this test can be seen in Table 5.6. This setup proved not to yield very good results, as the sheep got away 233 times out of the 500 games. However, the bad result does not come as a surprise. After all using the model made for the DFA-agent caveman is much too simple for modeling the BN-agent. Also the training used is the one for the DFA-agent, since training for the BN-agent would be very time consuming. The average game length, being 102 rounds, is slightly better than for two DFA-agents, and since the number of times the sheep got away was smaller this does not come as a surprise either. What can be seen from this test is that even though the network used for predicting the fellow caveman does not fit properly, having the model for the sheep is still a gain for the caveman.

Sheep Escaped	Percentage	Average Length
233	46.6%	102

Table 5.6: The results of the cavemen, where both are using a Bayesian Network - based on 500 games.

5.2.8 Adaptiveness Test

The next test was a test for the adaptiveness of the model. This was done using a trained model (540.000 cases) and then changing the way the sheep behaved. The main purpose of this test was to see how much effect fading has on the learning of a changing strategy, as well as testing the adaptiveness of the model. Therefore the training is being run both using fading and not using fading. The learning method used is sequential learning since EM-learning does not make sense when using fading.
The changes made to the sheep algorithm were the following:

- The sheep no longer waits for the cavemen to come in range, but instead constantly checks for the move which results in the greatest distance from the caveman and the corner.
- The sheep weights the second cavemen twice as high as the first.

These changes makes for a great change in the way the sheep behaves, and thus the caveman has to learn the model once again in order to make sure the sheep can be predicted using the old model.
The training used is running 50.000 games with random start positions, with the new sheep. We have decided to try using two large fading factors, since we still have a lot of new training data so it is not necessary to have the old training expire very fast. The fading factors we have decided to use are 0.9 and 0.95.
The test is performed in the same the other tests were conducted. That is, we run 500 games with random starting configurations, and afterwards count how many percent of the predictions were correct. The results of the adaptation tests can be seen in Table 5.8, while the result of running with the unadapted network can be seen in Table 5.7.

	Prediction correctness
No new training	16.2%

Table 5.7: The networks ability to predict the new sheep, before new training.

Fading	Prediction correctness
No fading	20.0%
0.9	76.1%
0.95	75.5%

Table 5.8: The results of the adaptiveness test, with 50.000 training games and tested on 500 games.

The result of this test is both surprising and not. It is not surprising to see a gain from using fading nor that there is a slight gain from adding new training data without fading. However, the extent of the gain of using fading is. We see a prediction correctness higher than the one from the fully trained model in Section 5.2.2. There are two possible explanations for this, either the strategy used by sheep has a tendency of only covering a few of the board's configurations, thus making it easier to predict the sheep. The other is the training method we use. In the case of "Stay" we still set *DirectionX* and *DirectionY*. The large number of stay configurations in the original sheep algorithm could result in the training for these nodes being biased towards the values we set in case of stay, instead of the values gained from the sheep actually moving. We also see that there is not much difference between the result gained with a fading factor of 0.9 and one of 0.95 actually the result with the second fading factor is a little smaller than the with first one, however the results are so close that it would be reasonable to see them as identical.

5.2.9 Discussion of the test results

One of the things the tests have shown is that for the model we have used, quite a few learning cases are needed. This is both the case when it comes to learning the network from scratch, and when then network has to adapt to a new strategy. This is a problem, if the model was to be used for modeling the behavior of a human player, since it is not likely to have thousands of games at your disposal in this case. So a model for learning a human player's strategy would have to have a much smaller CPT in order to be usable.
We have also seen that the degree of lookahead used in the improved caveman has an effect on how well it captures the sheep. This was to be expected since this gives the caveman a much larger area from which it can predict the winning configurations. However it is surprising that the use of lookahead with a model having a relatively low correctness in predictions still is much better than the DFA-agents. This just goes to show that even small improvements in strategy can yield great results. However, we have also seen that the cost of increasing the lookahead is much larger than the gain from using a greater lookahead. When increasing the lookahead from two to three the time used for running 500 games changed from hours to days, thus making the greater lookahead practically useless.
The last thing we have seen is that it is not sufficient to merely predict the sheep correctly. The prediction of the other caveman must also be correct in order to gain any major improvement by using lookahead. This is no surprise, since the prediction of the other caveman's actions is also used for the lookahead.

The tests have shown some strengths and some weaknesses of Bayesian Networks. The main problems discovered were the size of the CPT, when considering adaptation, and the time used to compile the network and compute lookahead. If it were to be used in commercial games it would most likely have to consist of a range of small networks, each giving decisions on different aspects. The problem of being able to divide the network into many small networks, is that independence of the variables are required in order to be precise. A variable cannot affect other variables if it is not included in the model that measures the effect. Games tend to have many variables, and not that many of these are often independent. When this is not the case other optimizations are required. Some of the these can be seen in the section where we construct our model, but other methods exists that have not been described in this report. To conclude; the tests offered us a chance to see that Bayesian Networks could actually be used to offer an improvement of the decisions made by the caveman, and this result was attained.
The tests were the last part of the implementation of the example game, and its agents. During the period of designing, modeling, implementing and testing we have experienced many situations that is

not covered in the many articles dealing with the potential of Bayesian Networks in games. In order to give the full picture of advantages and disadvantages of using Bayesian Networks for game development we will in the following chapter describe what we learned from our use in the example game.

Chapter 6

Potential for Bayesian Network Agents in Games

After having tested Bayesian Networks for our example game we will now summarize advantages and disadvantages we have experience while using Bayesian Networks. It is important to notice that this is only what we have observed during this process - not always the case. Applications of better usage of the Bayesian Networks might be present, other ways to model our problem, other ways to implement it - but it should still offer a good insight into working with Bayesian Networks.

6.1 Advantages of Bayesian Networks

In this section we will not describe the possible advantages, as is seen in many of the articles written from people in the industry, who finds this technology interesting. Instead we will describe the advantages we have found during the use in our example game. Had the example been more complex or closer to commercial standards the agents would most likely had been more complex, and the list longer. The advantages we describe here are compared to the alternatives already in use today.

- *Structural Bounds:* You can define structural bounds. This is something you often wish to do when working with Neural Networks. When you define causal directions in Bayesian Networks you at the same time define structural bounds. In order to make the causal directions a lot of analysis is required, as you need to assure that conditional independence exist among the variables you choose not to connect. It can prove difficult to classify situations if the conditional independence of the model is incorrect. This problem was seen with our model, where unrealistic conditional independence assumptions were made.
- *Adaptiveness:* The Bayesian Network technology enables you to take advantage of adaptation, when building agents. This can offer a longer lasting gameplay. Combined with the concept of fading you get an agent that automatically does many things you would see a human player do. This ability is not present in technologies such as DFA-, Fuzzy Logic- or Dempster-Shafer agents.
- *Controlled Complexity:* Bayesian Networks do not grow beyond control. It can often be a problem when working with Neural Networks, that overfitting and complexity is introduced at a later stage in the development. A controlled complexity is also a nice attribute to have in a technology when you need to debug a model. If you have a fully trained Neural Network,

where all nodes are interconnected it can often prove difficult. You can, if you need structural changes, add structural learning to the Bayesian Network. This would not be feasible in game, but perhaps when the agent is not active. Structural learning gives you the strength to change the network's structure, but can be difficult to keep under control if no boundaries are specified.

6.2 Disadvantages of Bayesian Networks

There are several reasons that the use of new technology is feared by people in the industry. We offer the problems we ran into during the use of Bayesian Networks in our example game. Some of the them could be overcome by working with Bayesian Networks for a longer period of time, as experience can help you in many ways, when searching for a better way of doing it.

- *Modeling:* The main problem with Bayesian Networks is the modeling process. It requires a lot of prior analysis of the problem area, and often also a lot of assumptions in order to be efficient. As was clearly seen in our example assumptions are dangerous, and can lead to inefficient models. It can also be difficult to predict how good your assumptions are before you test it afterwards. This means that modeling a Bayesian Network model is tedious and highly iterate work, as you need to test your assumptions. If these do not work you need to think of a new way of modeling it, and test it. This process continues until you are satisfied with the model. The reason this is a problem is that it can take a good while where you do not see results. If you are used to work with technologies such as DFA-agents you would expect to progress fast and see results early in the process. This is not the case with Bayesian Networks, as it can be difficult to make a model with good coverage, which is still efficient.
- *Coverage:* As mentioned briefly in Modeling, it can be very difficult to find a model that which fits the problem. This is an important notice for AI Designers, as they are often faced with many variables to cover, and many possible states of outcome.
- *Computing Power:* As always when talking about game development it is important to weigh the effect against the cost. As was shown in the test, using a Bayesian Network with lookahead without further optimization would not be usable in many games. Compared to technologies such as DFA-agents and Fuzzy Logic agents you need to weigh the need for advanced agents against the cost in computing power. If the agent does not need to do advanced reasoning or adapt to changing environments it would often be a waste of cpu-cycles and memory, and could be done faster, easier and more efficient with DFA- or Fuzzy Logic Agent.

6.3 Summarizing Benefits and Costs of using Bayesian Networks

It is always important to measure the cost against the effect, especially in an area like game development, where the technology is moving so fast. Bayesian Networks undoubtedly offers great advantages, where adaptation, fading and structural bounds are among our favorites. However it can be a large task to analyze the problem and then create a correct model, which is also efficient. In some cases it could prove faster than normal method, as you can change the model and small parts of the code controlling it, and then you have changed your network to work in a different setting. This is not always as easy, when working with scripts. A good model could also be used again and again, as you can just interpret the input and the output if the cases it needs to reason about is sufficiently close.

Modeling is the main problem, aside from the cost in computing power. The problem of cost in computing power is also seen in other agent types that are slowly being accepted by the industry, such as Neural Networks. Bayesian Networks offers a greater degree of control compared to Neural Networks in many ways, but is also more difficult to create efficient models in.
To recall our step-approach, described in Section 3.2, our practical test of Bayesian Networks covered all the described steps until the second part of the fifth step. However the sixth step was not reached, and therefore will be dealt with in greater detail in Chapter 7, Future Work, where we will lay the groundwork so that others can test this step in the future.

Chapter 7

Future Work

The test results have provided us with a large insight into the workings of Bayesian Networks. Even so, there are many other aspects left to examine. The possible applications of Bayesian Networks in games are many, and we have only been dealing with a small part. In this chapter we will try to shed light on other aspects that might be examined by others in the future.

Up until now we have shown that Bayesian Networks can be applied in game agents. It adds features, not present in the majority of todays games, such as adaptiveness and reasoning after observation. However there is another area in which Bayesian Networks might prove well suited - *Communication*. It is often a problem in games that the virtual worlds created appear too static, as they are only affected directly or indirectly by the player's actions. If the computer controlled agents were allowed to communicate, using a predefined protocol, it would add the sensation of a more dynamic world. Bayesian Networks are already set to work on observed input, and make decisions given only its observations - much like an agent in a dynamic world would be required to do. To incorporate this, some parts of the virtual world needs to be unknown to the agent. It will not be useful in a world where agents have perfect knowledge, such as our example game provides. The aspect of communication becomes even more powerful if combined with the aspect of *Trust*. If agents were programmed to lie occasionally to promote own benefits, other agents would be forced to assess the credibility of the agents it interacted with. This aspect should also be handled quite well with Bayesian Networks, as they allow you to change your beliefs when new observations occur, like when an agent lies to you.

The two aspects described here, *Communication* and *Trust* will be examined in greater detail in this chapter, as will a few other areas that we think would prove interesting. However no actual tests will be conducted - this is left as future work for other interested persons.

In order to show the potential of these aspects we have decided to describe an addition to our current example game, *Caveman's Struggle of Life*. The addition will add communication to the network, and later in the chapter also the aspect of trust. However to have the cavemen of our game act as independent decision makers they need to able to learn their fellow caveman's strategy. This has not yet been implemented and could cause some difficulty. If the model is not well described and limited you might end up in a situation where one caveman tries to learn the other's strategy, which depends on the first caveman's strategy. This way they will each incorporate the other's strategy, which will then expand as it also contains your own strategy. This may never stop, and hence they may not learn efficiently. If this problem could be overcome we could make some additions to the game rules, that would allow us to see the communication work.

7.1 New Rules

In order to have something to communicate about we need to add the following restrictions and additional rules to the game:

- Line of Sight: The caveman should have a radius, which is the limit of his eyesight. This will add situations where the caveman cannot see the location of his fellow caveman, or situations where he cannot see the location of the sheep.

- Communication: The cavemen should be able to request information from the fellow caveman. This should be done before any decisions are made, and the information returned should be received before any move-decisions are made. To limit the communication in order for it to be simple to implement, we have decided that only one piece of information may be requested and given in each round. The following pieces of information might be shared/requested:

  - Sheep's Current Position: If a caveman cannot see the sheep currently, he might request this information from his fellow caveman.
  - Fellow Caveman's Position: If a caveman needs to trap or predict the sheep it is vital to him, that he knows where his fellow caveman is placed, since this affects the sheep's decision. Therefore he might request this piece of information.
  - Predicted Sheep's next Position: The caveman might not have enough variables observed to be able to predict where the sheep is going to move to next, so he might request this information.
  - Fellow Caveman's next Position: The caveman might need to know how to lure the sheep into a corner, using his fellow caveman as help in the trapping procedure. Therefore he might request this information.

With this set of rules the two cavemen should be able to cooperate to capture the sheep - a common goal they share. Had our model been able to act independently, adding these new rules would not change anything, except in the code for the game, which would be responsible for exchanging the given evidence between the two cavemen.

7.2 Introducing Communication

Once the two cavemen have been put into the game, using the new rules, it becomes apparent that communication is going to be a great help. With the new rules, both cavemen may not have complete information of the game, and thus the cavemen may have to communicate both their wishes and their knowledge in order to reach their goal. In this section we are going to explain both how it could be decided which information to request, and how the new knowledge could be used in order to render decisions.

The first element concerns how the caveman should decide which part of the other caveman's knowledge he should request. The first thing which should be noted, is that the other caveman may not have the information you request. In this case you would receive a "don't know" message, and would have wasted your chance of getting additional information. Otherwise you would get the information you requested, and may use it to make your own decisions.

In order to decide which information should be requested we recommend the use of influence diagrams. As an example of how the diagram could be modeled we have included one in Figure 7.1.

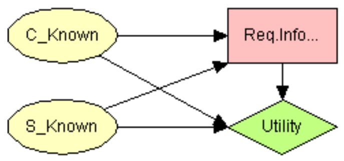


Figure 7.1: Influence diagram for requesting information. The *C_Known* node expresses whether the position of the other caveman is known, while *S_Known* states whether the position of the sheep is known.

```
if(C_known == "Unknown" or S_Known == "Unknown",
  if(C_known == "Unknown" or S_Known == "Known",
    if(Req.Info == "C_Pos", 1, 0),
    if (C_known == "Known" and S_Known == "Unknown",
      if (Req.Info == "S_Pos", 1, 0),
      if (Req.Info == "S_Pred", 1, 0))),
  if(Req.Info == "C_Mov", 1, 0))
```

Figure 7.2: Hugin Expression for the Utilily function.

The main idea is that, given that we know what information we have available, we have a decision with at number of choices:

- *C_Pos?* Request the position of the other caveman
- *S_Pos?* Request the position if the sheep
- *S_Pred?* Request the other caveman's prediction of the sheep
- *C_Mov?* Request where the other caveman will go to next

Depending on what knowledge we already have about the state of the game, we have to make a decision about which information to request. The utility function could be made as a Hugin expression, such as the one seen in Figure 7.2. The main idea is that if we have the information we need, we ask for the one we are missing. Else if we have neither, we ask the other caveman to share his prediction of the sheep's move. Finally, if we have all the board positions, we may ask the other caveman to share where he is going so that we can start our lookahead with more certain knowledge. The way the diagram together with the expression works, can be seen in the following example. We assume the position of the other caveman is known ($C\_Known = “Known''$), while the position of the sheep is unknown ($S\_Known = “Unknown''$). This will result in the highest utility being given to the decision where $Req.Info = “S\_Pos''$.
Once the request has been made and responded to, it is time to decide how the newly gained information should be put to use. In case we did not gain any new information (the other caveman did not have the information we requested), we simply make our decision based on the information we already have. In case we do not know the position of the sheep we will have to make a move, which seems reasonable, such as moving towards the field which gives us the greatest overview of the game board.

In case we got the position we needed, we can run our lookahead function, using this new information as evidence, and predict where the tile with the highest gain is. If the requested information was where the sheep is predicted to go to (for example because we did neither know the other caveman's nor the sheep's location), we should decide where to go using this information. Since we cannot run a prediction of the sheep (we do not know the other caveman's location) we will have to make do with a simple method, such as optimizing the distance to the sheep's next position. Finally in case we ask for the caveman's next position (this could be useful if we have both positions, since then we would not have to use our prediction of him for the first move, and thus would gain certain knowledge for this part.) we use this information in order to plan an ambush on the sheep.

7.3 Introducing Trust

The second aspect we will be dealing with is *trust*. We are going to use the same model as we made for the communication aspect, but extend further to take conflicting goals into consideration. It is important to have conflicting goals for trust to be interesting. In the communication aspect the two cavemen shared a common goal - they wanted to capture the sheep for themselves. In order to get a conflicting goal, while still keeping a mutual goal we will add another rule to the game's rules.

- Capturer Wins: The caveman who captures the sheep wins, the other dies of starvation.

This goal makes it valuable for a caveman to be the one who captures the sheep. However he still needs the other caveman to lure the sheep into corners to be able to capture it. This way they have both conflicting and mutual goals, making it worth it to both cooperate and plot against the another. To show the benefits of trust we will allow the cavemen to *lie* when giving information to the fellow caveman. This will be incorporated into the Bayesian Network, which will then allow the caveman over time to assess when his fellow caveman is telling the truth and when he lies to capture the sheep. This is an interesting aspect as it adds something very realistic to the game. You will not often find that all agents in a game have goals that offers mutual benefit to all others. The player might have several enemies, that are also mutual enemies, but facing the player they choose to cooperate towards a greater mutual benefit. The player could also try to make one part of the enemies believe the other part is lying to them, thereby undermining their mutual benefit.

Returning to our example we see that we need to measure how much we trust the other caveman. This could be measured on how many times he tells the truth compared to how many times he lies. We would also need to know when it is in our own interest not to tell him the truth. An example of such a situation is when he could capture the sheep shortly after, but does not know our location, which he needs to be able to predict the sheep's next move. Here it would serve our own quest for survival to feed him with an incorrect position, letting him believe that we affect the sheep to go in an entirely different direction.
An example of how this could be added to the model, which already has communication, can be seen on Figure 7.3.
In the figure you have the *GeneralTrust*-node, which states how much the fellow caveman can be trusted. An example is that there is no point in asking him for any information at all if he lies more than 90% of the times he has been asked. The *GeneralTrust* node is effected by a number of trust nodes, one for each type of information which can be requested. These values could be gained through experience, and thus the agent would have to learn the trust worthiness of the other caveman. This gives the overall trust given the possibility of the opponent lying on different requests (he may

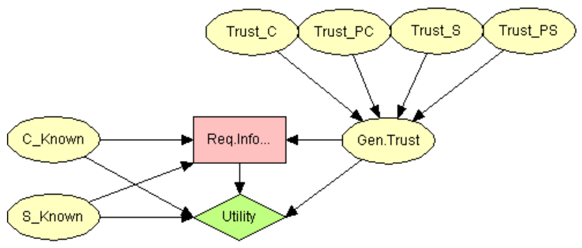


Figure 7.3: The addition of trust in the communications network. We have added the *GeneralTrust*-node, which is used to consider the overall trust to the fellow caveman. For each information we might request of the fellow caveman a trust node is added to assess how much we can trust him to tell the truth when this particular type of information is concerned.

have more tendency to lie in some cases rather than others). We also need to extend the *Decision*-node with another state.

- *DontAsk!* Do not request any information.

This state is used, when there is no gain in asking the fellow caveman about anything, as he lies too much.

```
if(Gen.Trust < Threshold,
  if(C_known == "Unknown" or S_Known == "Unknown",
    if(C_known == "Unknown" or S_Known == "Known",
      if(Req.Info == "C_Pos", 1, 0),
      if (C_known == "Known" and S_Known == "Unknown",
        if (Req.Info == "S_Pos", 1, 0),
        if (Req.Info == "S_Pred", 1, 0))),
    if(Req.Info == "C_Mov", 1, 0),
  if (Req.Info == "DontAsk!", 1, 0))
)
```

Figure 7.4: Hugin Expression for utility function, containing trust.

The Hugin expression in Figure 7.4 first evaluates if it is worth asking the fellow caveman at all. If he lies too much, the trust in him will be below a certain threshold, and there is no point in asking him about anything. The rest of the expression is the same as the original expression in the decision node, seen in Figure 7.2. The Threshold variable is a predetermined threshold for the amount of trust you require in order to request the information. It can be different for each type of information, as you might weigh some information higher than others.
The approach of keeping account of trust could be done in another way. You could also use *soft evidence* as opposed to hard evidence, when receiving information from your fellow caveman. This would allow past personal observation to be weighted against the evidence received from you fellow caveman.
The aspect of trust could add another dimension to many games. Your actions causes an effect, also on agents you have not yet met in the games. You will have to prove to other agents that they can

trust you through your actions.

7.4 Other Areas

Communication and Trust could be great additions in the scene of commercial games, but other areas of Bayesian Networks could be examined in future work. During the tests and the examination of the potential of Bayesian Networks we ran into aspects that could be examined by others in the future. It became apparent that Bayesian Networks would not be usable in the way we chose to make low-level decisions. Had the example game had time-critical restrictions, which should be expected in commercial games, it would not be a usable solution. Realtime games have time-critical restrictions. Bayesian Networks should still be usable, but at a higher abstraction level. We will briefly examine this aspect in this section. Another strength of Bayesian Networks, which was shortly introduced in the examination of the potential of Bayesian Networks was structural learning. Could this strength be utilized in commercial game development, or combined with already known techniques within the industry? This aspect is introduced in this section as well.

7.4.1 Realtime Games

In this report we have focused on a simple turn-based game. However, the majority of the games being published are realtime games. Thus, it should be considered which aspects of Bayesian Networks could be used in this kind of games.

It is clear that the approach we have used, is too ineffective for this kind of games, so either more efficient models must be made, or Bayesian Networks should be applied to different parts of the agent. For example Influence Diagrams have been used for selecting which strategy should be used in Unreal Tournament [RBBN02]. Influence Diagrams have also been used in a minor part of the artificial intelligence for Black & White [AIGPW02]. The possibility of using influence diagrams in realtime games makes it reasonable to expect that Bayesian Networks could also be a possible way of making some parts of the agent. For example predicting the actions of the opponent would be a place where it would be reasonable to use them. However, it should be noted that in order for Bayesian Networks to be useful in this case, it would take a very high level of abstraction, or the network would be too complex and thus unusable.

7.4.2 Structural Learning

A strength in Bayesian networks we have not gone into details with is structural learning. We have already seen that it can be a problem to find the structure of the network. This would minimize the problem of finding the structure. It could, in our opinion, prove a possible addition to the *GoCap* system. The GoCap system is a method in which the actions of a human player is monitored, and then used as training cases for training the agent. This method is described by Thor Alexander from Hard Coded Games in [GEMS3]. He is one of the AI Designers that sees great potential in technologies containing Machine Learning. The main problem of using structural learning with Bayesian Network for game agents, would be to find the proper variables. After all structural learning does not work by finding out which variables are useful, but finding out how the already given nodes should be connected. Another problem could be that the network resulting from the learning could prove to be too large and ineffective. Even though we have a relatively small number of nodes it is

possible that the connections between them can make the network useless[1]. An example of this is the simple network we constructed. It had only seven nodes. However, the way the nodes were connected resulted in the network CPT size being essentially too large compared to our problem. There is no guarantee that this would not be the network gained from running structural learning. The conclusion of this must be that structural learning sounds like a nice and easy way of using Bayesian Networks, however in practice it may not be quite as simple. The main problems are both finding the relevant nodes, and the complexity of the network learned.
This could be done as future work, as it would essentially remove one of the greatest disadvantages with Bayesian Networks - difficulty in making a good model. It would be a great step forward for Bayesian Networks if someone devised a way to combine the ideas of GoCap with structural learning without suffering from the setback of getting a too complex network. Even AI Designers who choose DFA-agents because they are easy to develop would be forced to reconsider, as technologies with machine learning actually speeds up other parts of the process. This can be seen in [GEMS3].

'From a production standpoint, machine learning will bypass need for the thousands of lines of brittle, special-case AI logic that is used in many of today's games."

- *Thor Alexander, Hard Coded Games*

Bayesian Networks would offer a fast agent-development technology if a GoCap system existed which could offer structural learning without the possible flaws that is inherent in the method today. You would have fast access to High-Level Artificial Intelligence, which also gives advantages such as adaptation and efficient debugging.

7.4.3 Summarizing Future Work

Many games today appear too static. The only actor who can change the game, unless scripted down to the very last detail, is the player. Agents do not interact, and they do not react realistically if you act against them or lie to them repeatedly. Bayesian Networks have the basis to change this, but in order for it to end up used in actual games, serious considerations and testing is required in this area. We have given some of the considerations in this chapter, but the application in this area is vast, and should be explored in much greater detail by others in the future.
Another problem is the application areas of Bayesian Networks within different types of games. Realtime games offers a challenge, as they have strict requirements for the time it takes to render a decision. Bayesian Networks requires some computing power, and therefore considerations needs to be made, as to how low a level of control a Bayesian Network controlled agent must have, or if the agent could consist of other technologies as well. Often the best AI solution is found by using different agent technologies in different areas of the decision making within the agent. An example of this approach can be seen in [AIGPW02], where the agents used in the game "Black & White" was divided into different agent technologies to attain different strengths and counter different weaknesses.
Bayesian Networks also have abilities which we have not dealt with in great detail. One of these is structural learning, which offers the possibility of changing the network's structure, based on learning. This can be a positive strength, but unfortunately suffers from issues that needs to be dealt with before the method would attract great attention among AI Designers.

[1]We actually tried using structural learning with one of the random training sets, and got a network with far too many connections between the nodes, as well as a structure completely different from the one we are using.

To conclude this chapter - there is much work left in this area, but it seems that certain of the aspects that game developers might find interesting are ready to be utilized through Bayesian Networks.

Chapter 8

Conclusion

There can be many reasons for AI Designers to change their choice of technology. Even though there are many different technologies to choose from and new aspects are found each year, there are strict requirements to such choices. In the analysis of the problem area it became apparent that there is still more focus on efficiency, debugability and control than on advancing the agents. Much of what can be found in commercial games today appear to be old fashioned and not taking advantage of technologies which have been accepted in many other areas of computer applications. However, in later years the reasons for not choosing a more advanced technology has been weighed against the need for longer lasting gameplay. As graphics and sound have gone through revolutionary changes over the last ten years, Artificial Intelligence has seen little in the field of game development. Today a change in this mentality is seen in the discussion between AI Designers, and they now look for new technologies to amaze their audience in new ways. The question then remains - *which new technologies should they adopt?*

The majority of the work, written by people in the industry, indicates a growing interest in the potential advantages of Bayesian Networks, as a new technology in agents. It offers a human-like reasoning, adaptability and a degree of control over the structure of dependencies. These are attributes that make the technology interesting, but there are still strict requirements it must apply to. How difficult is it to use compared to writing a script? How much more computing power does it require? How easy is it to debug? These questions need answering in order for the industry to seriously consider the actual use of such a technology.

The best way to prove that it would work and offer the advantages shown from its potential would be to use it in a real game development process. There are however not many who would risk this, as the process of making games is always on a tight schedule and does not hold room for great experimenting.

Another approach is to make an example game and then describe the process of working with the technology from beginning to end, showing both advantages and disadvantages during this process. This report is the result of such an approach.

It offers an insight that articles introducing the possible advantages of using this technology rarely contain. Even though it is shown using an example game this should not be used to restrict the application areas of the technology. There are many other types of games, which could have been used for this purpose. The example game is representative. It offers the agent information about the environment and what outcome these situations render. The agent can then learn other agents' strategies from these situations. This approach is generic, and therefore many of the observations done in this example game could be used in others as well.

The expected outcome of this project was described in the problem analysis, and consisted of several goals. A functional game was to be constructed, and used for showing weaknesses and strengths in using Bayesian Networks as artificial intelligence technology. The game, “Caveman’s Struggle of Life” was successfully implemented, and used for a series of test on the implemented agents. The game, as described earlier, provided the basis for showing the problems one could encounter during modeling, training and deployment of agents using Bayesian Networks. The description and evaluation of these problems, and their solution, was one of the main focus areas of this report. In order to see strengths and weaknesses in a perspective they were compared to strengths and weaknesses of other agent technologies. Altogether this offered an insight into what situations Bayesian Networks could be used, and which pitfalls that might then be encountered. Even though many other situations and pitfalls could be encountered, it could indicate in which situations AI Designers might introduce this technology.

Through the modeling process a network model was constructed. Although it at first appeared to cover the problem it was designed to, its performance was far from optimal. This was not unexpected, but not to the degree seen in the test results. The main cause of this was that it can be difficult to describe a specific implemented algorithm in Bayesian Network through modeling without adding too much bias. It was important from our point of view not to have much bias in this model, as this technology could as easy be used in predicting human behavior. In this case little bias can applied as humans do rarely reason and act in the same way.
The model was trained in two ways. One offered a measure for the degree of performance we could expect, the other offered a more realistic result. The prediction percentage of the model was in no way impressive, but combined with lookahead it still proved a powerful opponent to the sheep agent. The time used to calculate the lookahead were in many cases too long. However, it should be noted that no optimizations were implemented, and therefore the time required for the calculations could be reduced to a certain degree.
Even though the model was not large concerning the number of nodes present in the network, it suffered from another complexity problem. The CPT of the adaptable nodes were simply too large. This resulted in a model that did not adapt very fast. Adding fading improved the result greatly, but in commercial games such CPT sizes should be avoided. The example game offered the luxury of as many training cases as required. This is possible because all agents were computer controlled actors, as playing against a human player is something else. Adaptation is one of the attributes AI Designers are looking for in their choice of new technologies. Combined with fading the game offers a longer lifespan and is therefore worth considering. However, if the problems which needs to be modeled in games are larger than the problem of the sheep the size of the CPT would not allow for a realistic adaptation.

The two main problems found in using Bayesian Networks was *modeling the problem* and *CPT size*. The example game did not offer many actors or a complex environment, but still posed a problem to describe sufficiently. In commercial grade games the number of actors and the complexity in the environment would often be much higher. This yields both the problem of modeling the agent, but also of reducing the size of CPT in order to be able to adapt. Overcoming this problem is not easily done. Combining Bayesian Networks with other technologies or use it to make decisions at a higher abstraction level seem to be realistic ways of avoiding the problem. If the problem could be overcome Bayesian Networks could offer additional advantages as described in future work. Aspects such as communication and trust offers a more realistic agent behavior, which is what many AI

Designers look for. Even though there is still much work left to be done in this field, and many problems still needs to be overcome, there is no question that the technology of Bayesian Networks has a future in commercial game development. The results of this report offers an insight into working with Bayesian Networks for this purpose, the advantages and disadvantages encountered during this process, and an evaluation of its practical use. The introduction of new technologies is always a risk for a company, and costs and benefits needs to be compared. This report offers a basis for such an evaluation.

As with all other technologies in this field Bayesian Networks offers both strengths and weaknesses. It is not a technology an AI Designer can sit down and start implementing, but requires thorough analysis of the problem area. In exchange for this analysis it offers attributes that would appear appealing to many AI Designers. The question of using Bayesian Networks or not must be based on the requirements for these attributes in the game.

Appendix A

Appendix

A.1 Source code for the sheep

```
/* The main Sheep Deterministic Decision Function -- It first focuses
on staying away from the cavemen once they get within range for it to
 sense them, then it also focuses on staying away from corners ...
COMMENT : It needs to take borders into account as well */

pair<int, int> Sheep::decision(int id, CaveManGame *game) {

  if (caveman_in_range(game->get_board(), 0)) {
    // Find each caveman within range
    vector<pair<int, int> > cavies = other_locations(TYPE_SHEEP,
                                                     TYPE_CAVEMAN,
                                                     game->get_board(),
                                                     id);

    vector<pair<int, int> > pos_mov = possible_moves(TYPE_SHEEP, id, game);
    int longest_distance_caveman = 0;
    int longest_distance_corner = 0;
    int pos_index = 0;

    // For each possible position to move to
    for (int i=0; i < pos_mov.size(); i++) {
      // if there is no caveman on it
      if (TYPE_CAVEMAN != game->get_board()[pos_mov[i].first]
                                          [pos_mov[i].second].type) {
        int temp_distance = 0;
        // for each caveman on map

        for (int j=0; j < cavies.size(); j++) {
          // Add the New York distance of the cavemen
          temp_distance +=
              calculate_new_yorker_distance(pos_mov[i].first,
                                           pos_mov[i].second,
                                           cavies[j].first,
                                           cavies[j].second) ;
          }

          temp_distance += get_nearest_corner(pos_mov[i],
                                            game->get_board().size());
          // If the joint New York distance is larger
          // than previous seen

          if (temp_distance > longest_distance_caveman) {
            longest_distance_caveman = temp_distance;
            // Choose this position
            pos_index = i;
            // If distance is equal - calculate corner distance
          }
        }
      }
      // Returns a new position the pierce wants to be
      // in accordance with possible moves
      return pos_mov[pos_index];
    }

    // If no caveman is within range just stay at the current position
    return own_location(TYPE_SHEEP, game->get_board(), id);
  }
```

A.2 Source code for the DFA-agent (Caveman)

```
/* The main Caveman Deterministic Decision Function -- Blindly follows
the sheep to capture it - run entirely by its need for food */
pair<int, int> CaveMan::decision(int id, CaveManGame *game) {

  // Find possible moves
  vector<pair<int, int> >pos_mov = possible_moves(TYPE_CAVEMAN, id, game);

  // Get the locations of sheep (prepared for more sheep
  vector<pair<int, int> > sheepies = other_locations(TYPE_CAVEMAN,
                                               TYPE_SHEEP,
                                               game->get_board(),
                                               id);
  int index_found =0;
  int lowest_found = game->get_board().size();
  for (int i=0; i < pos_mov.size(); i++)
    {
      // Among the possible moves find the one with shortest
      // New York distance to the sheep
      if(lowest_found > calculate_new_york_distance(pos_mov[i].first,
                                                     pos_mov[i].second,
                                                     sheepies[0].first,
                                                     sheepies[0].second)) {
        lowest_found = calculate_new_york_distance(pos_mov[i].first,
                                                     pos_mov[i].second,
                                                     sheepies[0].first,
                                                    sheepies[0].second);
        index_found = i;
      }

    }
  // Returns a new position the pierce wants to be
  // in accordance with possible moves
  return pos_mov[index_found];
}
```

A.3 Loading and compiling a domain in the Hugin API

```
h_domain_t d;

void init_bn() {
  // Load the domain
  if ((d = h_load_domain("InBetween2.hkb")) == NULL) {
    cout << h_error_name(h_error_code()) << endl;
    exit(0);
  }

#ifdef MULTI
  h_domain_set_concurrency_level(d, 2);
  pthread_setconcurrency(2);
#endif

  FILE *log = fopen("model.log", "w");
  h_domain_set_log_file(d, log);

  // Triangulating the domain
  h_domain_triangulate(d, h_tm_clique_weight);

  // Compile the domain
  h_domain_compile(d);

  h_domain_save_to_memory(d);
}
```

A.4 Source code for the BN-agent with a lookahead of two.

```
enum Dir {
  N,
  S,
  E,
  W,
  STAY
};

// Returns the move which puts the caveman closest to the sheep
// Algorithm similar to the deterministic cavemans
pair<int, int> goto_sheep(CaveManGame *game, int id) {

  pair<int, int> sheeppos = predict_sheep(game->get_board());

  vector<pair<int, int> > moves = possible_moves(TYPE_CAVEMAN, id, game);
  int dist = 200;
  int res;

  for(int i = 0; i < moves.size(); i++) {
    if (dist > calculate_euclidian_distance(sheeppos.first, sheeppos.second,
                                            moves[i].first, moves[i].second)) {
      res = i;
      dist = calculate_euclidian_distance(sheeppos.first, sheeppos.second,
                                          moves[i].first, moves[i].second);
    }
  }
  cout << moves[res].first << " " << moves[res].second << endl;
  return moves[res];
}

// Given a direction return the tile to go to
pair<int, int> goto_dir(Dir dir, pair<int, int> pos, int id) {
  switch(dir) {
  case N:
    pos.second++;
    break;
  case S:
    pos.second--;
    break;
  case E:
    pos.first++;
    break;
  case W:
    pos.first--;
    break;
  case STAY:
    break;
  }
  return pos;
}

// Calculates lookahead, and evaluates if the lookahead results in
// a winning configuration
Dir look_ahead(vector<vector<Tile> >board, bool &success, int id) {
  pair<int, int> spos = predict_sheep(board);
  pair<int, int> c2pos = predict_caveman(board, !id);
  pair<int, int> c1pos = own_location(TYPE_CAVEMAN, board, id);

  vector<Dir> step2;
  vector<Dir> step3;

  for (Dir d = N; d <= STAY; (int) d += 1) {
    pair<int, int> new_pos = goto_dir(d, c1pos, id);
    if (calculate_euclidian_distance(new_pos.first, new_pos.second,
             spos.first, spos.second) <= 1) {
      success = true;
      cout << "LOOK 1" << endl;
      return d;
    }

    for(Dir d2 = N; d2 <= STAY;  (int)d2 += 1) {
      pair<int, int> spos2 = predict_sheep(spos, c2pos, new_pos);
      pair<int, int> c2pos2 = predict_caveman(spos, c2pos, !id);
      pair<int, int> c1pos2 = goto_dir(d2, new_pos, id);
        if(calculate_euclidian_distance(c1pos2.first, c1pos2.second,
                spos2.first, spos2.second) <= 1) {
                  success = true;
                  step2.push_back(d);
                  }
    }
  }
  if (step2.size()) {
    cout << "LOOK 2" << endl;
    return step2[0];
  }
  else {
    cout << "MAX LOOK" << endl;
```

```
        success = false;
        return N;
      }
    }

    // Uses lookahead in order to try and predict the optimal move
    // Else it moves closer to the sheep
    pair<int, int> Learner::decision(int id, CaveManGame *game){

        cout << "LEARNER" << endl;

        bool success = false;
        pair<int, int> own_pos = own_location(TYPE_CAVEMAN,
                                              game->get_board(), id);


        Dir win = look_ahead(game->get_board(), success, id);

        if (!success)
          return goto_sheep(game, id);

        else
          return goto_dir(win, own_pos, id);
    }
```

Bibliography

[AIGPW02] Editor: Steve Rabin, *AI Game Programming Wisdom*, Charles River Media Inc., 2002

[AIUGDN99] Scott M. Thomson, *AI Uncertainty*, http://www.gamedev.net/reference/articles/article197.asp, 1999

[BNDG01] Finn V. Jensen, *Bayesian Networks and Decision Graphs*, Springer-Verlag New York Inc., 2001

[GEMS1] Editor: Mark DeLoura, *Game Programming Gems*, Charles River Media Inc., 2002

[GEMS3] Editor: Dante Treglia, *Game Programming Gems 3*, Charles River Media Inc., 2002

[GEN5] Editor: James Matthews, *http://www.Generation5.org*, 1998-2003

[HAPI6] Hugin Expert A/S, *HUGIN API Reference Manual Version 6.0*, Hugin Expert A/S, 2003

[ML97] Tom M. Mitchell, *Machine Learning*, McGraw-Hill, 1997

[RBBN02] Rimantas Benetis, *RBOT (Intelligent agent in Unreal game environment)*, Aalborg University, 2002

[UIAI13] Editor: Dan Geiger & Prakash Pundalik Shenoy, *Uncertainty in Artificial Intelligence - Proceedings of the Thirteenth Conference*, Morgan Kaufmann Publishers, 1997

[VUGF03] Vivendi Universal Fact Sheet, *http://www.vivendiuniversal.com/vu2/en/what_we_do/factsheet_publishing.cfm*, Updated March 6th 2003