Intelligent game agents

Developing an adaptive fuzzy controlled backgammon agent

Author: Mikael Heinze

Educational institute: Aalborg University Esbjerg

Group: CIS4-E04-1

Date: 21st December 2004

Supervisor: Daniel Ortiz Arroyo

_____

Mikael Heinze

Preface

This paper is part of a master thesis on the subject of intelligent game agents on the 9th and 10th semester of Master of Science – Software Engineering.

This is the second and final part of the thesis, and is produced on the 10th semester at Aalborg University Esbjerg in the period 2nd September 2004 to 21st December 2004, and is continuation of work started on the 9th semester in the period 2nd February 2004 to 28th May 2004.

The paper has been composed by Mikael Heinze. Supervisor was Daniel Ortiz Arroyo.

The content of the enclosed CD is described in the appendix section 10.1.

Abstract

The topic of getting machines to play classical board games at the same level as top ranking human players, is a research area of artificial intelligence that has received a considerably amount of attention from researchers during the years. But although the area to this point has received quite a lot of attention, and there exist examples of computers that has defeated top ranking human players, it is still an area that intrigues researches, and where much research still can be done.

This paper presents a new way on how computers can be programmed to play classical board games. The model is based on a fuzzy controller with self calibrating membership functions, and it has been implemented and tested in the domain of backgammon with success. The paper also presents a model for automatically generating a fuzzy control rulebase. But even though this model shows signs of being able to generate such a rulebase, has it not become effective enough to match what a human control expert is able to.

An important aspect to the thesis is existing research in the areas of artificial intelligence, game theory and computer programs that play games. A large part the paper will therefore focus on these areas. This research has been done, as investigation of these areas was considered to be of high importance before work on a personal approach could begin.

Thesis Structure

Besides some general introduction presented on this and previous pages, is the structure of the paper based on eight main chapters not including references and appendix.

Chapter one is an introduction chapter defining the thesis. Chapter two through four will cover existing research and theories related to artificial intelligence and games. Chapter five will cover existing research on how fuzzy controllers can be created with the ability to learn. Chapter six will present the backgammon domain and describe the implementation of a backgammon application. Chapter seven will describe the implementation of an adaptive fuzzy controller able to play the game of backgammon, while chapter eight will evaluate and conclude the thesis.

In chapter nine, a list of references can be found. Throughout the paper references to one of these references will be done by the syntax [x] where x has been replaced with a number identifying the exact reference. At the end of the paper, chapter ten will hold the appendix of the paper. This appendix holds among other things a brief description of some other classical board games then backgammon mentioned throughout the paper.

Content

1. Project foundation

The object of this chapter is simply to introduce the thesis and this associated paper to the reader.

The following three sections are contained in this chapter:

- Introduction
- Problem definition
- Approach

1.1 Introduction

This paper is created in the 10th semester of Master of Science – Software engineering at Aalborg University Esbjerg as the last and second part of a master thesis on artificial intelligence (AI) and machine learning where the problem domain is classical board games.

This paper is a continuation of work done on the 9th semester. 9th and 10th semester can be viewed as two separate phases of the same project. The end of the 9th semester was however considered a major milestone, ending with an examination.

For this second part of the thesis, approximately half of the paper from last semester has been preserved, while the other half has been removed. This was done to obtain the best possible balance between keeping the paper at a reasonable size, while still maintain what is considered relevant for this second part.

1.1.1 Overview of first part

Part one of this master thesis focused a great deal on general theory of artificial intelligence, machine learning and game theory. This was both to obtain knowledge and understanding of this research area, as it for the author was a completely new topic, and to gather inspiration from work already done in these fields. Additionally two prototype agents with the aim of learning the simple game of tic-tac-toe was implemented, tested and evaluated.

The theoretical part of the previous semester was initiated with a study about artificial intelligence in general. Besides an attempt of trying to establish what artificial intelligence is, the following techniques related to artificial intelligence were also covered: Searching, fuzzy logic, fuzzy controllers, genetic algorithms, neural networks and Bayesian networks. The theoretical examination was continued with a study of basic game theory covering classification of games, game trees and agents. Finally the study was ended with a study of four highly successful attempts of getting machines to play classical board games. This part covered Deep Blue, Chinook, Evolutionary checkers and TD-Gammon. The sections related to this study have all been preserved in this second part of the thesis.

After the study of artificial intelligence and games, work on a personal solution model was started. Initial a section explaining the author's personal view on intelligence was given. This section is the last that has been preserved in the second part of the paper,

and is now presented as the very first section in the chapter related to artificial intelligence.

From that point requirements for an adaptive agent able to learn the game of tic-tac-toe was established, and a thesis for learning was identified. The identified solution model used a reinforced learning style where the outcome of games was used to learn the game of tic-tac-toe by classifying identified and played board patterns using a graded value between -1 and 1. After some modification of the initial solution model, did it turn out to be extremely effective in the domain of tic-tac-toe, and the agent became able to play the game very effective. To have some form of comparison an additional agent relying on neural networks and trained using genetic algorithms was also developed. Although this agent did show some small indications of learning, did it never turned out to be successful.

All sections related to the design, implementation, test and evaluation of these agents and the tic-tac-toe game has been removed from this second part of the paper. This is done as this part of the thesis will approach the problem of getting machines to learn and play games in a very different manner, and in a much more complex domain. The previous work can then mainly be considered history from which the author has gained a lot of experience. But as this work has nothing directly in common with the problem of this second part of the thesis, has it been removed.

For a better overview of what has been preserved and removed in this second part the reader can refer to Figure 1, which is an overview of the final paper after the first part. From this figure can it be seen that that the artificial intelligence, game theory and related work chapters has been preserved, as well as the intelligence section of the analysis chapter. Everything else has been removed.

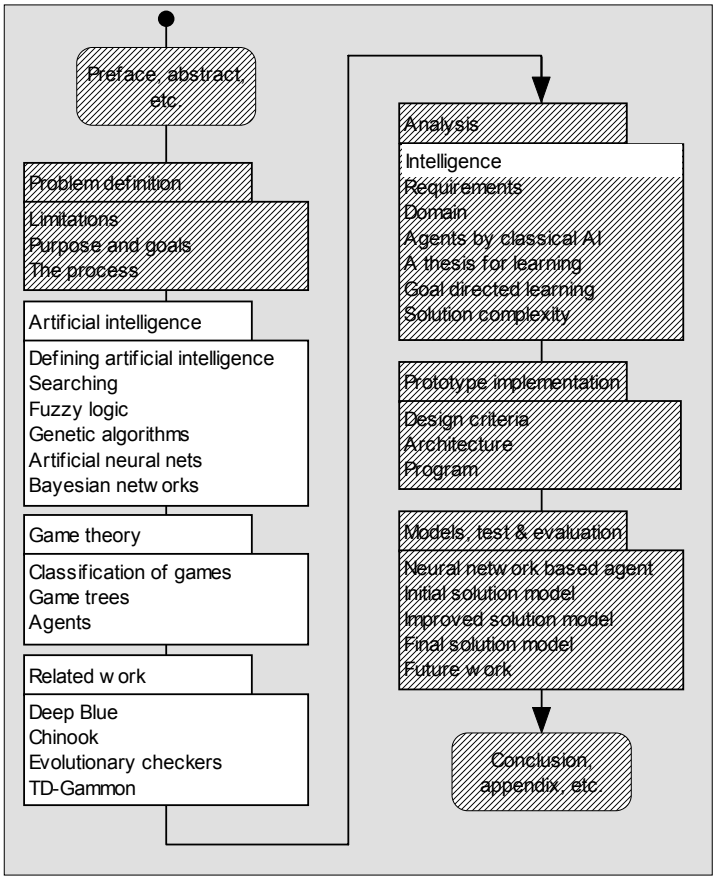

Figure 1: Preserved and removed sections of first part paper

1.1.2 Lesson learned

During the first part of this thesis, has it by the author been experienced that artificial intelligence in relation to games mainly seems to be about board pattern classification in one form or another.

The two typical approaches is to either use a database that will hold a huge amount of different boards from which a score indicating how good a board is can be obtained, or to use neural networks. If neural networks are used, the typical method is to feed the network with board patterns as input, and then obtain some score as output, indicating how good a board is estimated to be.

The challenges of AI research in relation to games is then to find ways on how these databases can be used effective by an agent, or maybe even automatically build. If neural networks are used, the object is to find some effective way of training it to classify board patterns.

Some examples include the use of existing databases holding large amount of games. These databases are then used to extract knowledge about the strength of boards. Such existing databases have also been used to train neural networks using supervised learning. Another typical approach is known as rote learning. In this approach the agent itself build a database of encountered boards while playing, and then mark them with some score depending on, for example, the outcome of the

games. Another used approach when using the outcome of games, is to use a neural network where each encountered board is feed as input. The network is then trained such that the output to each board pattern is a value based on the outcome of the game. Finally genetic algorithms have also been used to evolve neural networks, able to classify board patterns.

Although the above mentioned approaches as turned out to be effective in a many cases, it in the authors opinion has some drawbacks.

- Using existing game databases to create an evaluation function does not give the agent any form of learning or adaptive behavior. It simply has no possible way of learning from additional games played and mistakes done. The same is true if using genetic algorithms. After the evolution has been complete nothing more can be learned. The type of play in a given situation will in other words always be the same.
- For complex games it is an extremely heavy process to classify the amount of board patterns required to be able to have a well playing agent.
- Even though the agent learns by training, the amount of adaptive behavior is very limited. The amount of different boards in a complex domain is simply so large that a specific board is only rarely played. So even though the evaluated score of a board is changed, the adaptive behavior will first be visible the next time this board has the possibility of being played, and that might not necessarily happen soon after.
- Should the agent be trapped in an obscure an extremely rarely happening situation, it might not be able to respond in a clever way simply because the situation has never been encountered before.
- In relation to other branches of AI it is not a very general and usable approach to learning.
- It does intuitively not match how humans learn.

The work done on the first part of this thesis, has however contributed with a lot of knowledge that has given inspiration in trying to solve the problem of creating an intelligent and adaptive game agent in a quite different way.

1.2 Problem definition

The object of this thesis is to create and develop adaptive fuzzy controller. The selected problem domain is the classical board game of backgammon where the object, besides of creating a fully working backgammon game, is to create a computer agent based on a fuzzy controller, able to play and improve its play in general and possibly even against specific opponents with unique playing styles.

This fuzzy controller will throughout this project be refereed to as Fuzzeval. This is a contraction of Fuzzy Evaluator, and has been selected because the task for the fuzzy controller will be to effectively evaluate the strength of different board layouts by means of fuzzy logic, thereby hopefully develop an acceptable playing style.

More specifically then the learning issues related to a fuzzy controller and thereby Fuzzeval, relates to how membership functions can be adjusted, and how rules can

be generated and added (eventually removed) from the rulebase to improve and change behavior. In the most optimal circumstances must this be done automatically while Fuzzeval runs within its environment, which in this case is the game of backgammon.

The hope is that a new paradigm for learning and adaptive behavior can be developed. A paradigm that does not learn by “remembering” every situation encountered, but a paradigm that instead tries to somehow “understand” the basic strategies of the problem it is faced with, and then learns how to quantify the importance of each strategy against each other, thereby developing a successful playing style.

To evaluate whether Fuzzeval is able to develop a successful playing style, a benchmark backgammon player known as Pubeval is to be used. Pubeval is a quite capable backgammon player that can be used to test and compare the playing strength of different backgammon programs. For a slightly more detailed explanation of Pubeval one can refer to section 6.4.

Although fuzzy controllers often are associated with hardware control, does computer games in the author’s opinion makes an excellent environment for AI research. Games offer a combination of simulation and reality. The environment in which a computer player interacts is virtual, but the environment is not a simulation of a problem domain, it is the problem domain. Issues such as noisy sensor data, hardware dependencies and other hardware related problems can therefore be ignored, while still addressing a realistic problem.

Thesis
Show if fuzzy controllers is a useful way of making artificial intelligence with the ability to learn as well as demonstrate adaptive behavior in a game domain

1.2.1 Purpose and goals

To have a reference point throughout the thesis, purpose and goals are defined.

Purpose

- Demonstrate an understanding of current research in the field of artificial intelligence both in general and related to games.
- Contribute to the area of artificial intelligence, machine learning, games and fuzzy controllers with new thoughts, ideas and methods.
- Study if/how fuzzy controllers can be design to solve a task and improve in doing that.

Goals

- A backgammon game with an adaptive fuzzy control based computer opponent (an agent) able to play and learn the game.
- This paper documenting the above purposes and goal as well as phases, aspects and considerations of the thesis.

1.3 Approach

In Figure 2 an overview of how this thesis is to be approached is given. This is not to be confused with the structure of this paper. It should be noted that some overlap of the phases of course might be unavoidable.

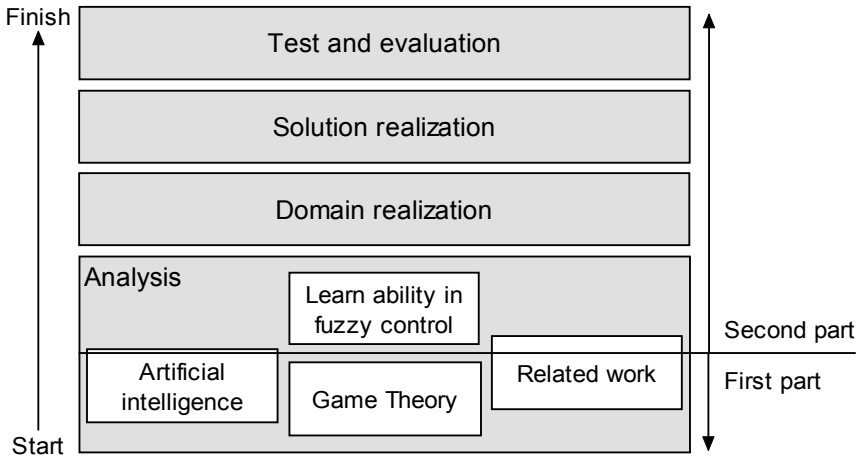

Figure 2: Approach of thesis

Analysis is the first step and has already been started during the first part of this thesis. The analysis of game theory is considered complete although some very minor adjustments still might be done. The analysis of artificial intelligence is for most parts also considered complete, except for the section related to fuzzy controllers, which might be extended and rewritten to some degree. Much analysis associated to the related work section is also considered complete. During the first part of this thesis, the author did however come across a range of additional attempts of making both adaptive and non adaptive game agents, where a description did not make it into the paper. A very brief overview of some of these attempts will therefore be added to the related work section. The study of TD-Gammon made during the first part will also be made slightly more detailed. The main analysis in this second part will however focus on fuzzy controllers, an exiting work related to how they can obtain the ability to learn. This is to gather inspiration when designing a fuzzy controller able to play and learn the game of backgammon.

Domain realization is the design and implementation of the domain Fuzzeval is to work within. In this case is this a backgammon game. This is a rather big part, without any real relation to the main objectives of this thesis. It is however still related to the derived objective of creating a backgammon program, and is a necessary phase to complete before the development of a solution model for Fuzzeval can begin.

Solution realization is the core and main part of this thesis. Here design and implementation of Fuzzeval are to be done. Here requirements will be defined and solution models will be presented, implemented, tested and improved.

Test and evaluation is the final part of this thesis. Here a detailed test and evaluation of the final solution model for Fuzzeval will be done, and conclude whether a successful model has been identified.

2. Artificial intelligence

This chapter is to give an overview of the area of artificial intelligence. Its intention is to give the author of this report insight into this field of artificial intelligence, first and foremost to get an understanding of what artificial intelligence is, but also to gather inspiration from the field. For the reader of this paper, it is to work as an introduction to the field of artificial intelligence.

Before going to deep into the field of artificial intelligence, is the first section (2.1) however to establish the author's personal view of intelligence in general, as well as what can be considered a realistic view on artificial intelligence in relation to this thesis.

This chapter consists of the following main sections:

- Intelligence
- Defining artificial intelligence
- Searching
- Fuzzy logic
- Fuzzy controllers
- Genetic algorithms
- Artificial neural nets
- Bayesian networks
- Conclusion on artificial intelligence

2.1 Intelligence

This section will take look on what intelligence is from the author's personal perspective. But before going to deep into intelligence and discussing the capabilities for an intelligent system, it should be determined if it is at all possible to create intelligence on a machine. Although most should agree that at has not been successfully done yet, it still needs to be determined if it is at all possible. The answer to this question lies very much in ones own view of intelligence. Two views can be taken on intelligence, a mechanistic or romantic [2].

In the mechanistic view intelligence is believed to be an algorithm. This means that there is a recipe for intelligence, which can be broken down into a finite step by step instruction. The invention of human like artificial intelligence is just a matter of understanding it, and implementing it. This does not mean that intelligence is something humans currently is able to create, but it is believe to be possible to create intelligent machines in the future, as it is just a matter of understanding it. One supporter of this belief is Bill Gates which as said: "I don't think there's anything unique about human intelligence. All the neurons in the brain that make up perceptions and emotions operate in a binary fashion."

In the romantic world view intelligence is believed to be something very unique. It is not governed by any rules, and can as a consequence not be implemented as an algorithm. All supporters of this belief do not necessarily believe that humans not will be able to create intelligence. They just believe that it has to be created in some other

way than rules implemented on machines. This means that artificial intelligence is believed to have a greater chance of success, in for example a scientific branch like biology.

Depending on what view one supports, should it lay down the fundament for what the goal is, and requirements are from an intelligent agent. As a result, the view that is to govern this thesis is the romantic. This is because catching the essence of intelligence in an algorithm, and creating a human like intelligent system, as supporters of the mechanistic view should find possible, is not seen as a realistic goal. Instead the goal is, like most existing research in AI, to find ways to simulate intelligence.

To obtain an initial understanding of intelligence and what is to be simulated, let us highlight some of the interesting terms when looking up intelligence, and its synonym *mind* on Dictionary.com:

Intelligence
1. a. The capacity to **acquire and apply knowledge**. b. The faculty of **thought** and **reason**. c. Superior powers of mind. *See Synonyms at mind.* 2. …

Synonyms at **mind**
1. The human consciousness that originates in the brain and is manifested especially in **thought**, **perception**, **emotion**, **will**, **memory**, and **imagination**. 2. The collective conscious and unconscious processes in a sentient organism that direct and **influence mental and physical behavior**. 3. The principle of intelligence; the spirit of **consciousness** regarded as an aspect of reality. 4. The faculty of **thinking**, **reasoning**, and **applying knowledge**: *Follow your mind, not your heart.* 5. …

So intelligence gives, according to its definition, the following capabilities:

- Having thoughts, perceptions, emotions, will, memory, imagination and consciousness.
- Ability to obtain knowledge
- Thinking and reasoning
- Influencing the mental and physical behavior.

These capabilities are however not to be seen as isolated from each other, but instead as capabilities that cooperate, to create what can be classified as intelligence. A way to illustrate this can be seen in Figure 3.

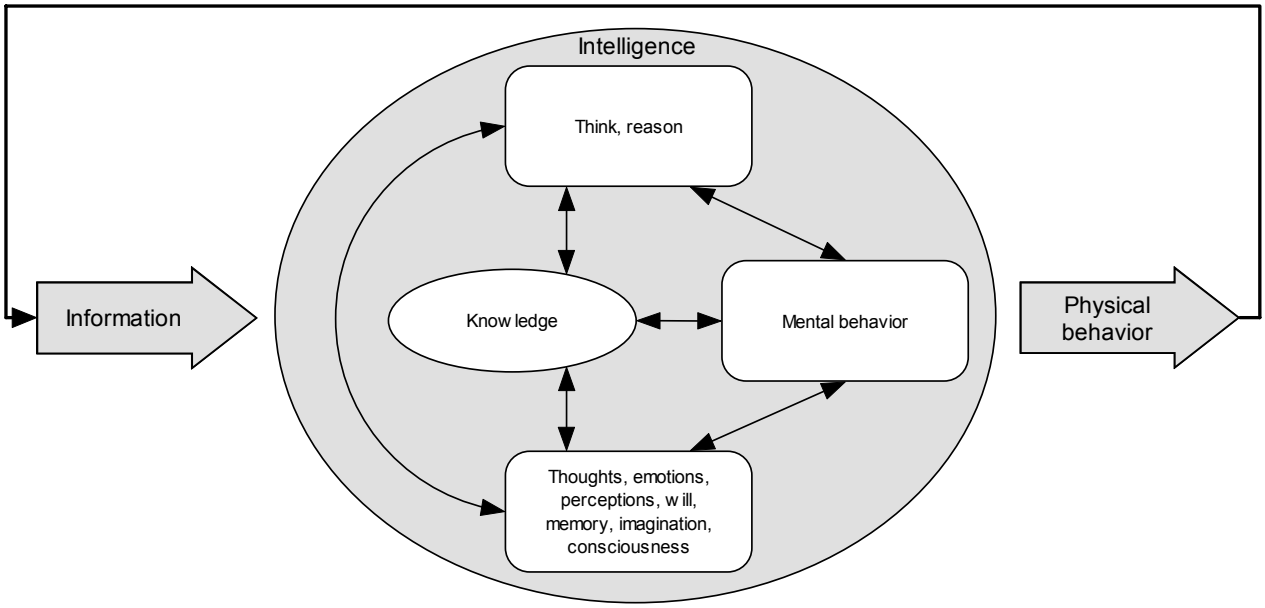

Figure 3: Illustration of intelligence

Information is continually given to the mind. This information is then if understood, and someone's intelligence is capable of it, converted to knowledge by use of existing knowledge and the capabilities represented by the rounded rectangles. All the capabilities mutually influence each other and ultimately govern the physical behavior which is how the intelligent entity acts. This physical behavior will ultimately create new information that is given to the mind.

Intelligence can for the entity possessing it, be viewed as a rather complex process. But the processes within the mind are of minor interest for entities observing other intelligent entities. The only part other is capable of observing is the information (or stimuli) given and the response obtained. It is as a consequence extremely important that the physical behavior seems logical in relation to the information given. Because we human has defined our self as intelligent, we measure intelligence against our own definition of intelligence, and that is how it to some degree has been the norm for human's to act, given that information. However what is interesting is that previous information is able to govern future actions; this can also be referred to as learning. Intelligent entities are aware of that and as a consequence knowledge and learning should play a major role in artificial intelligence. Many of the other capabilities are of minor importance. Emotions can for example not be directly observed. Emotions might trigger someone to be sad, that again may result in one crying. But it is the physical behavior of crying that can be observed, not the emotion itself. As such can the oval in Figure 3, be considered a black box, and the challenge in artificial intelligence is then to create this black box known as intelligence.

The physical behavior, that is the output from an intelligent entity, can be categorized in one of two categories, which are reflexes and deliberate actions. Deliberate actions, which are the one governed by the mind, and the interesting in this context, will in most (if not all) cases have the purpose of helping the entity reaching a desired goal. That could be win a game, obtain more knowledge, have fun, get from A to B, solve

an equation, etc. and with greater intelligence comes the ability to reach more complex goals. Reaching a goal is however not to be distinguished with luck. The goal might be reached, but if the intelligent entity has no idea on how it was reached, and is unable to repeat it, can it not be defined as intelligence.

In essence AI is to create some system that is feed with information. Given this information should it be able to obtain knowledge and thereby learn about the domain from which the information is given. Given some information the system should, if necessary, decide on an output, which is the response to the problem at hand and a suggested solution to how the entity will try to reach its goal. To be able to learn, is it necessary for the intelligent entity to receive feedback on its ability to reach the goal, such the entity is able to distinguish good decisions from bad. Eventually the system is then to become better and better at solving a given task and to reach the desired goal. Ultimately intelligence can then be defined simply as an entities ability to achieve goals.

2.2 Defining artificial intelligence

Most people have a general idea about what artificial intelligence is. Asking someone and the answer will almost certainly be something like: “it is making machines or computers think”. According to Dictionary.com this answer should be rather good as it can be seen that the two words are defined as:

Artificial	Intelligence
1. Made by humans; produced rather than natural. 2. Brought about or caused by sociopolitical or other human-generated forces or influences: *set up artificial barriers against women and minorities; an artificial economic boom.*	1. The capacity to acquire and apply knowledge. 2. The faculty of thought and reason. 3. Superior powers of mind. *See Synonyms at mind*.

So the study of artificial intelligence is about producing some human made device, with the ability to acquire knowledge, and using this knowledge to reason rational. But instead of just accepting this rather loose definition, lets take a more in depth look at some of the definitions of artificial intelligence, in the hope that a more exact definition can be establish.

Artificial intelligence goes back to 1950 where Allan Turing wrote a paper [5] in which he stated the question “can machines think?” In this paper he proposes a game that can test if a machine is intelligent. Today this test is known as the Turing test.

The proposed game is in its original form known as the imitation game. In this game there are three people, one male, one female and one interrogator. The interrogator is isolated from the male and female and all communication happens in written form. The goal for the interrogator is then to distinguish the male from the female simply by

asking questions, while those being questioned should try to fool the interrogator. The Turing test is based on replacing one of participants with a computer. The goal for the interrogator is then to distinguish the human from the computer. If the computer can fool the interrogator, then is it said to be intelligent. For the Turing test to be successful the computer would need to poses the following capabilities:

- Natural language processing
- Knowledge representation
- Reasoning abilities
- Machine learning

Some has argued that the Turing test is a good test because to pass it, a wide range of knowledge about the world is needed. Further more a great deal of flexibility in dealing with new situations is needed as well. However it has also been criticized for only taking a behavioral rather then psychological view of intelligence. The computer does in other words not need to be intelligent; it just has to give an impression of having it.

The reason why people still debate the validity of the Turing test is most likely because a broader definition of what exactly artificial intelligence is, still not has been found. In [3] the author has collected eight definitions made from other authors, and made an interesting observation. That is, the definition of artificial intelligence can be organized into four categories. The four categories are:

Systems that think like humans	Systems that think rationally
Systems that act like humans	Systems that act rationally

Table 1: Four main categories for the definition of AI

First comparing the two categories at the top against the two categories at the bottom the difference is that in the topmost, the definition of artificial intelligence deals with thoughts and reasoning. Artificial intelligence should actually be able to think, whereas the goal in the two categories at the bottom is to make a system that behaves as if it is intelligent, although it might have no intelligence at all.

Now comparing the left column against the right one, success in the left column is measured in human performance. While in the right column, success is measured against an ideal concept of intelligence, which the author has called rationality. This deviation comes from the fact that human sometimes do irrational actions. The meaning of irrational action in this context is simply the fact that humans sometimes are making mistakes unintentionally, or (having the power to) acting against all logic in a given situation.

An exact definition of artificial intelligence does unfortunately not exist. Although the term is widely used, does it does not attempt to give a specific definition. Almost every book on the subject of artificial intelligence, seem to have its own definition. The problem is then to find the definition that appeal to one. The following definition is made by Amit Konar in [1].

> Artificial intelligence can be defined as the simulation of human intelligence on a machine, so as to make the machine efficient to identify and use the right piece of “knowledge”.

Which in my opinion is in the same category as the Turing test, and that is systems that act like humans. Whether one agree with this definition or not, it should be intuitively clear, that artificial intelligence is about getting machines to find solutions to problems in ways somewhat equal to that of humans, and this involves borrowing characteristics from human intelligence, and applying them to machines.

2.3 Searching

Search approaches are described in [1]. One way of looking at artificial intelligence, is as search. One has a set of examples and a set of rules. The learning process is simply to apply the set of rules to the examples and start looking for a solution. In many cases the task of AI can simply be reduced to the problem of performing a search within a given search space. The search space is the collection of all possible and reachable solutions, and the search for a solution is then the problem solving.

2.3.1 Generate and test

The most straightforward approach is simply to test every possible solution to a problem and then picking the best one. However as all but the simplest problem either has an infinite, or finite but very large search space, is this approach impractical. It will simply be impossible to search every solution as it will take to long or even forever. One common use of this approach is in game playing programs. The computer will try to look as far ahead in the game as possible, and the consequently selecting the best move. This form of search is known as game trees, and is explained in section 3.2.

For searching large or infinite search spaces, the simplest alternative is to do a random search. Random solutions are generated until a satisfying solution is found or a sufficient large search space has been tested. There is no guaranty that this approach will find a useful solution unless allowed to run in an infinite time. If the search space contains very few non bad solutions, random search are unlikely to be of any use.

2.3.2 Hill climbing

This approach starts by generating a random solution which is to act as the starting state. But in contrast to the random search, is this solution used to generate one or more close variants. If a better variant is found, this is instead used to guide the further search and so on. If after several iterations no improvement is found, the search terminates. However in the case that all neighborhood states are equal or worse than the current best solution, and the best solution is not a satisfying goal, then the search algorithm is said to be trapped at a hillock or local extrema. One way to overcome this is by randomly selecting a new starting state, and then continuing the search process.

2.3.3 Heuristic search

Heuristic means rule of thumb. The general approach in heuristic search is to use one or more heuristic functions to determine the best candidate among a collection of legal states that can be generated from the current state. The difficult task in heuristic search, is selecting a satisfying heuristic function, as the heuristic function has to be intuitively selected or created.

2.3.4 Means and ends analysis

This search method attempts to reduce the gab between the current state and the goal state. One common way of doing this is by measuring the distance between the current state and the goal state, and then applying some operator that will reduce the gap. In mathematical theorem-proving, means and ends analysis is often used.

2.4 Fuzzy logic

Fuzzy logic [1], [29], [32] and [33], deals with the area of artificial intelligence known as uncertainty handling. The primary point in fuzzy logic is to define a way to deal with imprecision. The point is that in real life problems, the answer to a question is only rarely of the form true or false. Fuzzy logic is an extension to traditional two valued logic.

Traditional two valued logic uses the binary pair {0, 1} to represent the two truth values false and true. It is however argued by Lotfi Zadeh, the father of fuzzy logic, that many real world sets of today are defined by non sharp boundaries. Example of such a set could be height. This is because height of an object often simply cannot be answered as either being high (true) or not high (false). Instead an object can be high to a degree. Fuzzy logic uses the whole interval between [0, 1]. This gives the ability to model a continuous change from false to truth and thereby giving someone ability to be somewhat high instead of either just high or not high.

2.4.1 Fuzzy sets

A set is a collection or class of objects that shares a property. A set is specified by its members (items or elements). An example could be the set of integers greater than 0 and smaller then 5 specified by A = {1, 2, 3, 4}.

In a fuzzy set each member has a membership degree between 0 and 1. The higher the number is, the higher the membership is. The elements of a fuzzy set are taken from a universe. The universe holds all the elements that come into consideration when determine the fuzzy sets members. Every element in the universe is member of the fuzzy set to some degree. This membership degree could be zero. The set of elements that have a non-zero membership is called the support. The function that ties each element from a universe to a fuzzy set is called a membership function and is often denoted by the letter μ. Figure 4 is an illustration of a membership function that defines to what degree someone is tall.

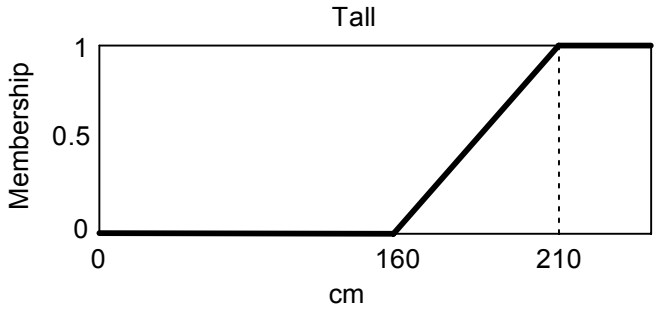

Figure 4: Illustration showing to what degree a person is tall

From Figure 4 can it be seen that if a person is below 160 cm that person is not tall at all, if the person instead is above 210 cm, the person is fully tall. The interesting part is between the intervals 160 and 210 cm where the degree to which someone is tall is a continuous change from 0 (false) to 1 (true). A membership function can also be written mathematically as below:

$$\mu_{Tall}(x) = \begin{cases} 0 \quad for \quad x \le 160 \\ 1 \quad for \quad x \ge 210 \\ \frac{1}{50} \cdot x - 3.2 \quad otherwise \end{cases}$$

Let X be a nonempty universe. The fuzzy subset A in X is characterized by its membership function:

$$\mu_A: X \rightarrow [0, 1]$$

and $\mu_A(x)$ is the element x's degree of membership in A. This is illustrated in Figure 5.

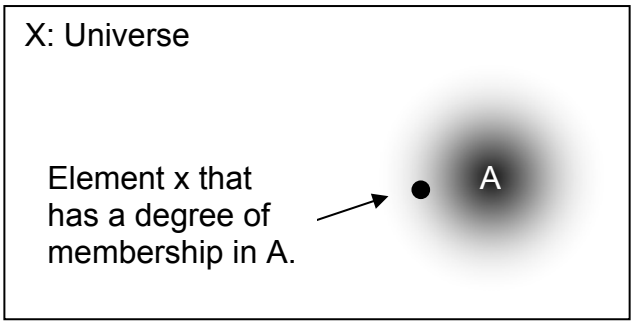

Figure 5: Element x is member of the fuzzy subset A in universe X

A fuzzy set can simply be viewed as a collection of ordered pairs:

$$A = \{(x, \mu(x))\}$$

Item x is a member of A and μ(x) is its membership. If A is a finite fuzzy set then an often used notation is:

$$A = \mu_1/x_1 + ... + \mu_n/x_n$$

to illustrate each member and its corresponding membership grade. A single pair (x, μ(x)) is called a singleton.

2.4.2 Operations on fuzzy sets

In traditional Boolean logic there are the three classical operators union (or), intersection (and), and the not operator. These operators exists in fuzzy logic too, but are defined differently.

- A or B = Max($\mu_A(x)$, $\mu_B(x)$)
- A and B = Min($\mu_A(x)$, $\mu_B(x)$)
- not A = 1 - $\mu_A(x)$

Fuzzy set operations creates a new fuzzy set from one or several existing fuzzy sets. Besides these classical operators a whole range of other operators exists. Fuzzy set operations can be divided into three groups. That is t-norms, averaging operators and t-conorms. Figure 6 illustrate how the three groups of operators are related.

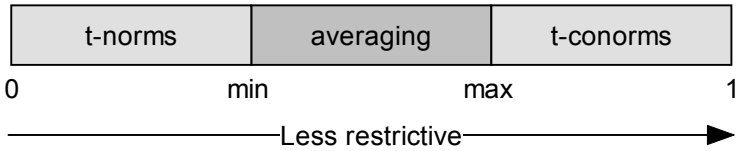

Figure 6: How the three operator types are related

The boundary between t-norms and averaging operators is defined by the classical *and* operator, also known as the *min* operator, or the intersection. The boundary between averaging operators and t-conorms is defined by the classical *or* operator, also known as the *max* operator, or the union. Figure 7 is an illustration of the three primitive set operations.

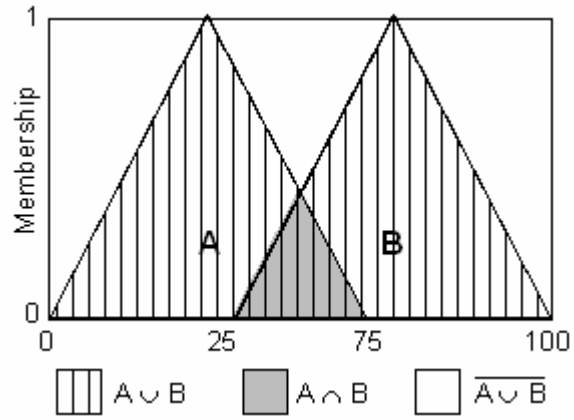

Figure 7: Illustration of the three primitive set operations

2.4.3 Fuzzy relations

A fuzzy relation is a way to describe relationship among objects. Graphically it can be illustrated as shown in Figure 8. From this illustration can it be seen that B resembles A to a degree of 0.8. Then if an operation involving B is requested, but no B exists,

then doing the operation involving A might be a better solution than not do doing the operation at all.

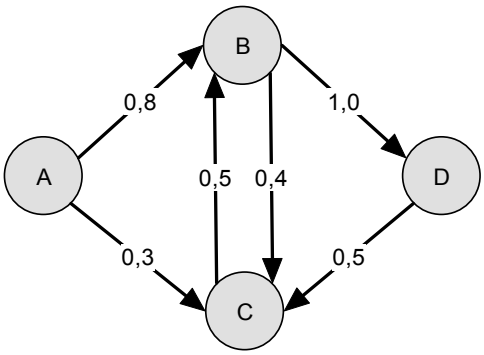

Figure 8: Graphical illustration of a fuzzy relation

To explain this model in relation to the human mind, and example could be a person asking another if he has a pick-up truck he can borrow, instead of just answering with a “no”, he could say “no but I have a trailer”. Here trailer is seen to be a pick-up truck to some degree and having some relation to pick-up truck in the given context, and using a trailer might be a reasonable solution.

Another way to illustrate a fuzzy relation is in matrix form as illustrated in Table 2.

↗	**A**	**B**	**C**	**D**
A	1	0,8	0,3	0
B	0	1	0,4	1
C	0	0,5	1	0
D	0	0	0,5	1

Table 2: Matrix representation a fuzzy relation

Although no explicit connection might exist between two objects, there still might be a connection through other objects. In the above example it can be seen that D has a connection to B through C. One way to find these connections, and the strength of them, is by use of the max product closure.

2.4.4 Linguistic variables

To represent the measurements of Fuzzy logic, linguistic variables are used. A linguistic variable takes word or sentences as values. Each value is called a linguistic term. A linguistic term has been defined over a base variable. In short: linguistic variables → linguistic term → base variable.

Example of a linguistic variable could be speed. Values of this linguistic variable, which is a linguistic term, could be slow, fast, very fast, quite slow or moderate. Each linguistic term is defined over a base variable that in this example could scale from 0 to 250 km/hour. Another example can be seen in Table 3.

Type	Example
Linguistic variable	Age
Linguistic term	young, very old, not so old
Base variable	0 to 100 years

Table 3: A linguistic variable and its fuzzy and base variable

A linguistic modifier is an operation that changes the meaning of a linguistic term. For example in the sentence “much larger than 0”, the word *much* modifies “larger than 0”. Although the exact effect of a modifier can be hard to define, they can often be approximated by operations as illustrated below. Where *a* is the membership function of a linguistic term.

$$\text{Very } a = a^2$$
$$\text{Extremely } a = a^3$$
$$\text{Slightly } a = a^{1/3}$$

2.4.5 Approximate reasoning

Approximate reasoning is the process to reach a fuzzy conclusion given some fuzzy input. In approximate reasoning we deal with what is called propositions; that is assigning graded truth values in the range from 0 (absolutely false) to 1 (completely true). A proposition can have the following form:

X is P

- X: The subject of the proposition
- P: The predicate

Example of a proposition could be:

Leaves are green

It can be seen the predicate is a fuzzy variable, as it can be argued that something can be green to a degree. Also the subject of a proposition can be a proposition. An example can be seen below:

(Leaves are green) is true

Propositions can be modified by use of linguistic modifiers. As a result the above proposition can be modified to something like:

All (leaves are green is not true)

Here *All* is called a quantifier and *not* is called a predicate modifier.

One of the most useful ways to do approximate reasoning is called the generalized modus ponens. This can be represented as:

$$\frac{\begin{array}{c}\text{If x is A then y is B} \\ \text{x is A}\end{array}}{\text{y is B}}$$

Which means that if it is given that *If x is A then y is B* and *x is A to some degree* then we can conclude that *y is B to some degree*. A more concrete example can be seen below:

If wind is blowing then window is closed
Wind is somewhat blowing

Window is somewhat closed

What exactly *somewhat* is defined as is problem dependent, but it could translate to something like the window should be open one third of what it is capable of.

2.4.6 Fuzzy possibility theory

There might be cases where the exact value for an element is not known, and as a consequence is the elements exact membership degree to a fuzzy set not known. Consider the membership function previously presented in Figure 4 and re-presented in Figure 9 for clarity.

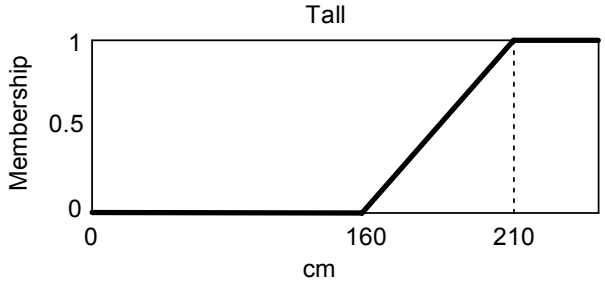

Figure 9: Illustration of the membership function to what degree someone is tall

And remember the corresponding membership function defined as:

$$\mu_{Tall}(x) = \begin{cases} 0 & for \quad x \leq 160 \\ 1 & for \quad x \geq 210 \\ \frac{1}{50} \cdot x - 3.2 & otherwise \end{cases}$$

Consider a case where we only know that someone's height is between 190 and 200 cm (Let us call the person *x*). The possibility distribution can then be illustrated as seen in Figure 10.

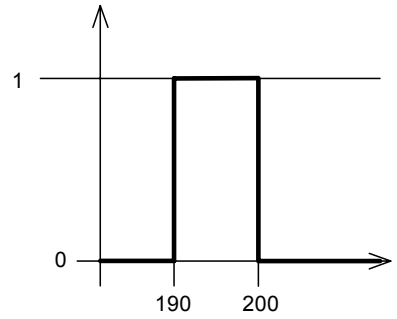

Figure 10: Illustration of possibility distribution for x's height

The answer to the question: "To what degree is the statement '*x* is tall' true?" can then be answered by possibility and necessity defined as:

$$\Pi(A \mid B) = \sup_{x \in X} \min(\mu_A(x), \pi_B(x)) \qquad \text{(Possibility)}$$

$$N(A \mid B) = \inf_{x \in X} \max(\mu_A(x), 1 - \pi_B(x)) \qquad \text{(Necessity)}$$

- μ_A : Membership function
- π_B : possibility distribution

Possibility and necessity can in *x's* case be illustrated as in Figure 11:

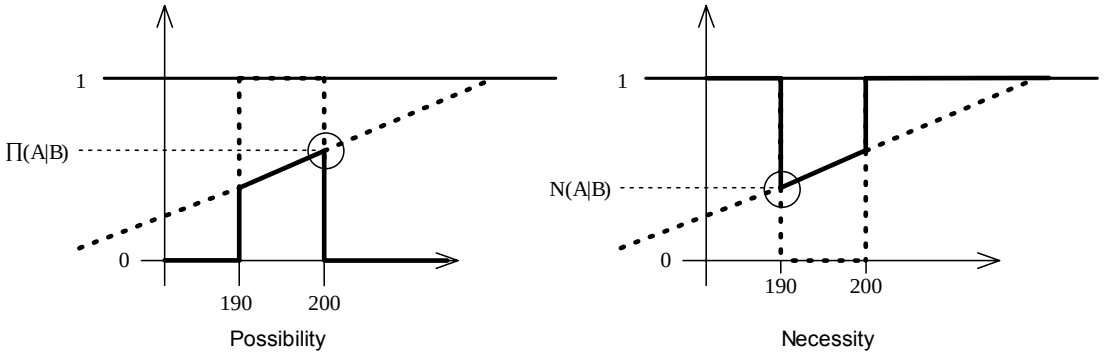

Figure 11: Possibility and necessity for x is tall

This can also be calculated as below:

$$N(Tall \mid [190,200]) = \frac{190}{50} - 3.2 = 0.6$$

$$\Pi(Høj \mid [190,200]) = \frac{200}{50} - 3.2 = 0.8$$

The result is that *x* is possibly tall to the degree 0.8 but necessarily tall to the degree 0.6.

2.5 Fuzzy controllers

Fuzzy controllers are described in [31]. Although fuzzy controllers often are associated with hardware, is there is no reason why the principles cannot be applied to other domains. Fuzzy controllers are also sometimes called fuzzy expert systems. The term fuzzy controller is generally used when the system is to control some hardware or electrical products such as washing machines, video cameras and rice cookers, or industrial equipment such as trains and industrial robots. Fuzzy expert system is a more general term referring to computer programs that by use of fuzzy logic, is capable of solving problems by use of a knowledge base generally crafted by human experts.

The main idea of fuzzy controllers is to build a model of a human control expert who is able of controlling something on behalf of rules specified in form of linguistic variables. A fuzzy controller can include empirical rules. The collection of rules is called a rulebase. The rules are in if-then format. The if-side is called the condition and the then-side is called the conclusion. Examples of typical fuzzy controller rules are given below.

1. If error is Neg and change in error is Neg then output is NB
2. If error is Neg and change in error is Zero then output is NM
3. …

Neg, *NB* and *NM* are linguistic terms and stands for Negative, Negative Big and Negative Medium. The controller is then able to execute the rules and compute a control signal depending of the inputs error and change in error. In a rulebase, the control strategy is stored in a somewhat natural language. This makes the controller easier to intuitively understand and maintain for humans. In the next parts an overview of the structure of fuzzy controller is first given, and then a concrete example of how a fuzzy controller works is given.

2.5.1 Fuzzy controller structure

Figure 12 illustrates the different blocks in a typical fuzzy controller. The following section will explain the different blocks in the diagram.

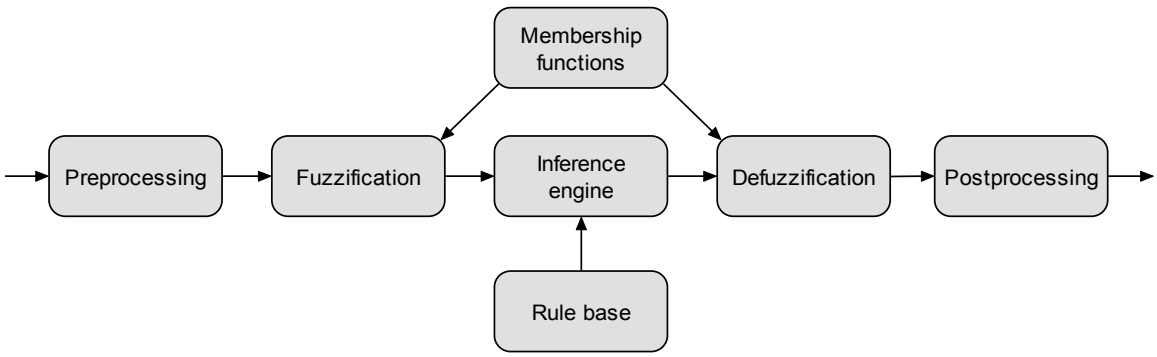

Figure 12: The different blocks of a fuzzy controller

Preprocessing

Inputs to a fuzzy controller are often hard or crisp data from measuring devices. A preprocessor makes some initial adoption of the data before it enters the actual fuzzy controller. Some examples of preprocessing could be:

- Rounding from floating points to integers.
- Normalization or scaling into a standard range.
- Filtering to remove noise.
- Averaging to obtain a tendency.
- Combining of measurements to obtain key indicators.

Membership functions

All input values to a fuzzy controller have associated linguistic terms. All these linguistic terms needs to have an associated membership function. These membership functions are used when calculating to what degree an input value fulfils the requirement for a linguistic term. This block holds all linguistic terms and their associated membership function.

Fuzzification

As the fuzzy controller takes input from the real world, this data has to be converted to fuzzy values. This process is called fuzzification. This is done by assigning a membership degree to each linguistic term for that input.

Rulebase

The rulebase is where all rules for the fuzzy controller are stored. Basically rules are the fuzzy controller's linguistic terms, which are connected in an if-then fashion, but thy can be represented in different ways. Rules are typically connected with the words *and*, *or*, *if-then*, *if and only if* and the modifier *not*.

Inference Engine

The inference engine is responsible for calculating the resulting fuzzy set. The fuzzy set is the possible output values and their calculated membership degree to the input. First each rules grade of fulfillment is calculated, this is also often called firing strength. This is done by aggregating each value in the condition. After each rules grade of fulfillment has been calculated, the fulfillment of each possible output can be found.

Defuzzification

The resulting fuzzy set must be converted to some kind of control signal that can be sent to the object there is to be controlled. This final control signal can be calculated in a number of different ways. Depending of the problem domain a suitable method can be selected.

Postprocessing

If necessary some final adoption of the result can be done in the postprocessing part.

2.5.2 Fuzzy controller example

A fuzzy controller is however probably best explained by giving a step by step example. Consider a fuzzy controller who takes two input values distance and load and puts out one value speed. It could be some kind of device that was transporting gods between load and unloading points. The fuzzy controllers linguistic input and output variables along with their defined linguistic terms and membership functions can be seen in Figure 13.

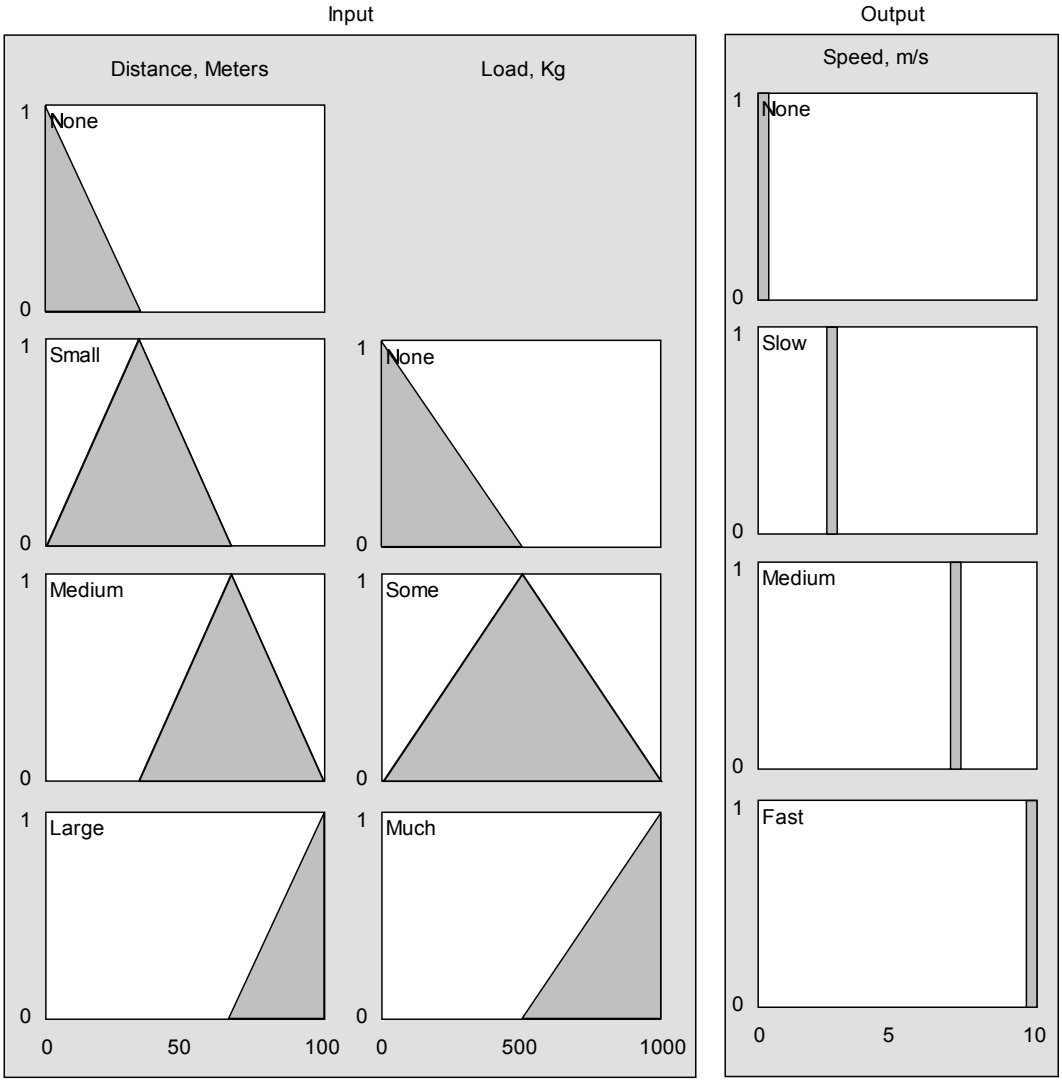

Figure 13: Membership functions for input variables Distance, Load and output variable Speed

The defined rulebase could look something like:

1. If Distance is None and Load is None or Some or Much then output None
2. If Distance is Small and Load is None or Some or Much then output Slow
3. If Distance is Medium and Load is None or Some output Medium
4. If Distance is Medium and Load is Much output Slow
5. If Distance is Large and Load is None or Some output Fast
6. If Distance is Large and Load is Much output Medium

For a better overview and to check for completeness of rules such the controller will not get into undefined states, a matrix representation of the rulebase can be seen in Table 4.

		Distance			
		None	Small	Medium	Large
Load	None	*None*	*Slow*	*Medium*	*Fast*
	Some	*None*	*Slow*	*Medium*	*Fast*
	Much	*None*	*Slow*	*Slow*	*Medium*

Table 4: Fuzzy controller rulebase in matrix form

Now consider an example where this controller receives the following input.

Input	
Distance	50 meters
Load	250 kg

Table 5: Fuzzy controller input

The goal is then to decide the control signal, which is the speed of the device this controller is responsible for.

Preprocessing

First the input values enter the preprocessor which makes some initial filtering or adoption of the data. In this example it could simply be rounding from floats to integers or a change of the input value distance to 100 meters in the event that it was above that value. In this example we will however not make any preprocessing of the input.

Fuzzification

In the fuzzification block, the inputs membership degrees to each of the linguistic terms have to be calculated. This is done by a lookup in the membership functions that can be seen in Figure 13. The membership to each linguistic term can be seen in Table 6.

Distance

Linguistic term	Membership
None	*0*
Small	*0.5*
Medium	*0.5*
Large	*0*

Load

Linguistic term	Membership
None	*0.5*
Some	*0.5*
Much	*0*

Table 6: The inputs membership to each associated linguistic term

Inference engine

In the inference engine the fulfillment for each rule in the rulebase is calculated. This is done by aggregating the variables in the conditional part. Typically the fuzzy *min* operator will be used as *and* operator, while the fuzzy *max* operator will be used as *or* operator. A relatively often used alternative is instead to use the fuzzy algebraic product or sum defined as $(a \cdot b)$ and $(a + b) - (a \cdot b)$ as *and* and *or* operators. In this example the standard *min* and *max* operators are however used. Each rules grade of

fulfillment can be seen in Table 7 while we for clarity exemplifies the calculation of the second rule as Min(0.5; Max(0.5; 0.5; 0)) = 0.5.

Rule	Fulfillment
If Distance is None and Load is None or Some or Much then output None	0
If Distance is Small and Load is None or Some or Much then output Slow	0.5
If Distance is Medium and Load is None or Some output Medium	0.5
If Distance is Medium and Load is Much output Slow	0
If Distance is Large and Load is None or Some output Fast	0
If Distance is Large and Load is Much output Medium	0

Table 7: Each rules grade of fulfillment

Rules in a rulebase can furthermore be weighted by a value in the range from 0 to 1. In this case the fulfillment of a rule will be changed to

$$\alpha^* = w * \alpha$$

where α* is the new grade of fulfillment, w is the weight for that rule and α is the original calculated grade of fulfillment. This is however ignored in this example.

After all the rules are calculated, we have the fulfillment of each rule. A problem then arises as there might be several rules with similar conditional output, but with different fulfillment degrees. In Table 7 it can for example be seen that there exist two rules outputting *slow* and two rules outputting *medium* both with different grades of fulfillment. Such a situation will result in different fulfillment values for the same output. These identical output terms are then typically aggregated. A typical way to aggregate is to use the fuzzy *max* operator which will result in the following fuzzy set.

$$\text{Speed} = 0/\text{None} + 0.5/\text{Slow} + 0.5/\text{Medium} + 0/\text{Fast}$$

Alternatively aggregation can be done using any other fuzzy operator one might see fit. Examples include such operators as bounded sum, or an averaging operator such as the fuzzy arithmetic mean. Another approach is also simply to forward all identical outputs with different fulfillment grades to the defuzzifier and then use all the output values when defuzzification is done. For the rest of this example we however assume the *max* operator has been used.

As it is the case in this example, the conditional part does not necessarily have to output linguistic terms. Actual values are also in some cases used. Even the use of functions as output can be used. Such rules are refereed to as Takagi and Sugeno rules. An example is illustrated below:

If distance is large and angle is small then output $(a \cdot \text{distance} + b \cdot \text{angle} + c)$

Where a, b and c are constants.

Defuzzification

The resulting fuzzy set has to be defuzzified into a single valued control signal. There is no defined way to do this, but a popular method is centre of gravity defined as:

$$u = \frac{\sum \mu(x_i)\ x_i}{\sum \mu(x_i)}$$

Here x is the actual value of the linguistic terms in the output set. These values are illustrated in Figure 13 and summarized in Table 8 for clarity.

Linguistic term	Value
None	0 m/s
Slow	3 m/s
Medium	7 m/s
Fast	10 m/s

Table 8: Actual values of output linguistic terms

The calculation of the control signal is then as follows:

$$\frac{(0\cdot 0) + (0.5\cdot 3) + (0.5\cdot 7) + (0\cdot 10)}{0 + 0.5 + 0.5 + 0} = 5$$

The control signal would be the value 5. This would be the calculated speed the object to control is to be assigned.

Postprocessing

If necessary a final adoption of the output could be done. This is however ignored in this example.

2.6 Genetic algorithms

Genetic algorithms [35] and [36], deals with the area of artificial intelligence known as machine learning. In short genetic algorithms use the principles of Charles Darwin's theory of evolution to search for solutions to a problem. A problem is encoded such that it can be represented as a vector. This vector is called a chromosome. To start with a population of random chromosomes is created and tested up against the given problem. Depending on how well a given chromosome solves this problem, a fitness grade is assigned to the chromosome. Then a new population of chromosomes is generated by "mating" the chromosomes in the current population with each other. The process of mating is not completely random as the fittest chromosomes have a greater change of mating then the not so fit ones. If everything goes well the next generation of solutions should become fitter. This evolutionary process then continues over many generations until a satisfying solution is evolved.

2.6.1 Steps in genetic algorithms

Before a genetic algorithm can be used, a way to encode the problem must be found. This encoding could for example be a vector of real numbers or integers. It could also be a string, or a more typical case, a binary string. The way to encode the problem at

hand effectively is probably the hardest part of designing a genetic algorithm. The encoding of a problem is called a chromosome and each value in a chromosome is often referred to as a gene and may in the binary case for example look like:

10001001110010010

To start with a population of random chromosomes is created. Each of these chromosomes represents a solution to the problem. Then the following steps are repeated until a satisfying solution is found:

1. Calculate the fitness score for each chromosome in the current population.
2. Chose two parent chromosomes from the current population using fitness proportional selection.
3. Depending on crossover rate, crossover the parent chromosomes at a random point to generate their offspring.
4. Depending on the mutation rate, mutate each bit in the offspring chromosome.
5. Go to step 2 until population is complete.
6. Replace current population with the new population.
7. Go to step 1 until a satisfying solution is found.

The fitness score

The fitness score is a score assigned to each chromosome that indicates how well this chromosome is at solving the problem at hand. Deciding on an effective fitness algorithm can be all from easy to hard depending on the problem at hand. An example of a straightforward fitness algorithm, if the problem was the traveling sales person, is to calculate the distance. The lower the distance, the more fit the solution is.

Chromosome selection

The fitness proportional selection is a way to select members from the population in a way such that chromosomes with the best fitness value have a better chance of being selected. It does not guarantee that the fittest members are selected, just that their chances of being selected are rather good. Imagine a population of five chromosomes where the fitness for each chromosome, is a percentage of the total fitness in the population. This could be the population shown in Table 9. Then each chromosomes change of selection is the same as its fitness value.

Chromosome	Fitness
1	8.8%
2	30.8%
3	28.4%
4	5.6%
5	26.4%

Table 9: Population of five chromosomes and their fitness

This gives chromosome 2, 3 and 5 a rather good change for selection. While the not so fit chromosome 1 and 4 have a much smaller change of influencing the evolvement of the next generation.

Crossover

Next step after parent chromosomes have been selected, is to produce their offspring. The most widely used method for this is called single point crossover. This is done by swapping the two parent chromosomes bit at a random chosen location. This is probably best explained by looking at Figure 14.

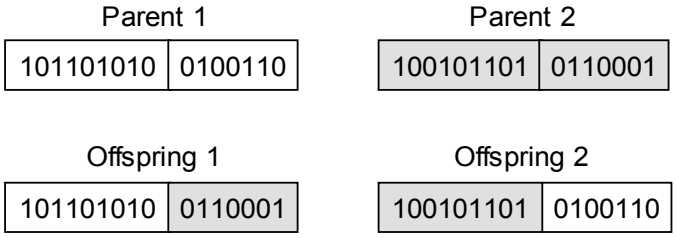

Figure 14: Crossover of two chromosomes and their offspring

Sometimes crossover does not occur. In this case the parent nodes are copied directly to the new population. The chance for a crossover to occur is usually around 70%.

Mutation

After chromosome selection and crossover, a new population of chromosomes exists. Some of them are exact copy of chromosomes from the previous population and some of them are produced by crossover. To ensure that not all chromosomes are the same, there is a small chance that a gene (position) will mutate. Depending on how the chromosomes are encoded mutation can either happen by changing a value, or by flipping a bit. The chance for mutation is typically 1 tenths of a percent. Mutation is a fairly simple process of simply stepping through each chromosome; each gene in a chromosome then has a small chance of changing value.

2.7 Artificial neural nets

Artificial neural nets (ANN) [1], [34], [37] and [41] is a way of trying to simulate a biological brain electronically in software or hardware. Artificial neural nets are also often just called neural nets, although artificial neural nets is a more correct term, as a neural net is a biological term refereeing to the network within brains. Artificial neural nets are mainly designed to work with pattern classification. The network can take a vector of numbers and then classify that vector and thereby give a desired output. For a concrete example of how to apply a neural network to a simple problem one can reefer to the appendix section 10.2. This example also exemplifies how genetic algorithms described in the previous section can be applied to the problem of learning.

A brain is made up of billions of tiny cells called neurons. Each neuron is connected to other neurons and communicates with them through connections made up of what is called synapses and dendrites. A neuron continuously receives signals from other neurons through these inputs. It then performs some operation on the input to find a value called the activation. If this activation value is higher than a given threshold, the

neuron outputs a signal. This is typically called a step function and an example can be seen in Figure 15.

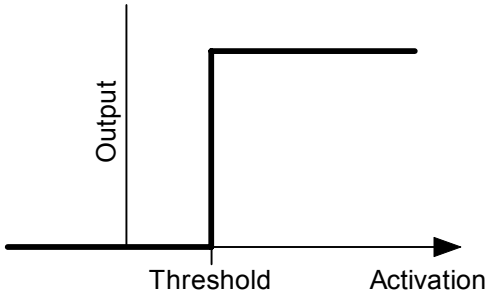

Figure 15: A step function that outputs a signal when the activation is above a given threshold

This output is then forwarded to other neurons that will receive this output signal as their input.

2.7.1 The artificial neuron

Artificial neural nets are made up of a collection of artificial neurons. There is no specific rule for the amount of neurons used in a neural net. Depending on the task, it can differ between as few as three up to millions. Figure 16 is an illustration of an artificial neuron.

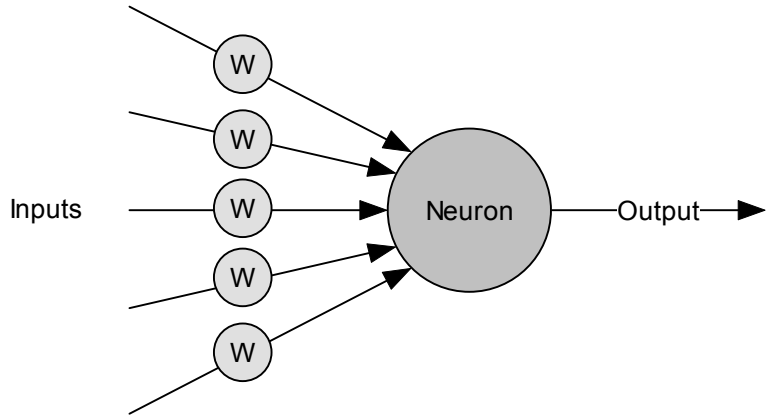

Figure 16: Artificial neuron with five inputs. Each input has an associated weight W

An artificial neuron can have any amount of inputs. Each input has an associated weight which is simply a floating point value. In most cases can this weight be both positive and negative. It is these weights that are adjusted, as a neural network learns. The inputs may be represented as x_1, x_2, x_3, …, x_n and the related weights for each input as w_1, w_2, w_3, …, w_n. The activation value for a neuron can then be found by:

$$activation = \sum_{i=0}^{i=n} w_i x_i$$

However in an artificial neuron the activation value is not put through a step function. This would limit the output to a binary output. Instead of using a simple step function, a function able to soften the output is used. One of the most popular is the sigmoid function defined as:

$$output = \frac{1}{1 + e^{-a/\rho}}$$

- Where *e* means the exponent, which is a constant approximated to 2.7183.
- a is the activation into the neuron.
- And *p* is a value controlling the shape of the function typically set to 1.0.

This function will produce and output of the form seen in Figure 17.

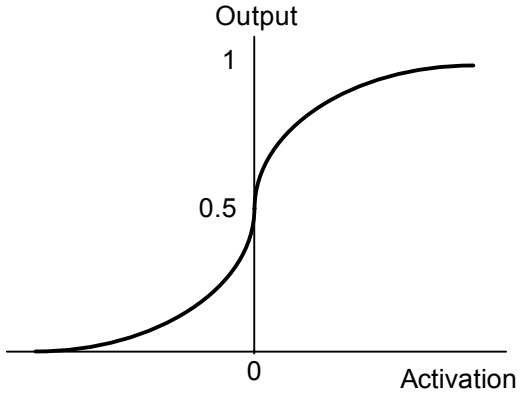

Figure 17: Output of a sigmoid function

Therefore to get a graded output greater then 0 and smaller then 1 from an artificial neuron, the activation value just has to be run through this sigmoid function.

2.7.2 Artificial neural net architecture

To create an artificial neural net, a collection of neurons, also called nodes or units, has to be connected somehow. There are different types of networks; they do all however have a standard network architecture consisting of several layers, an input layer, hidden layers, and an output layer. The input layer takes the input and sends to the hidden layers. The hidden layers then do the necessary computation before sent to the output layer, which outputs the data to the user.

The most popular architecture is called a feed-forward network. An example of a feed-forward network can be seen in Figure 18. This network gets its name because each neuron feed its output forward to the next layer until a final output is obtained.

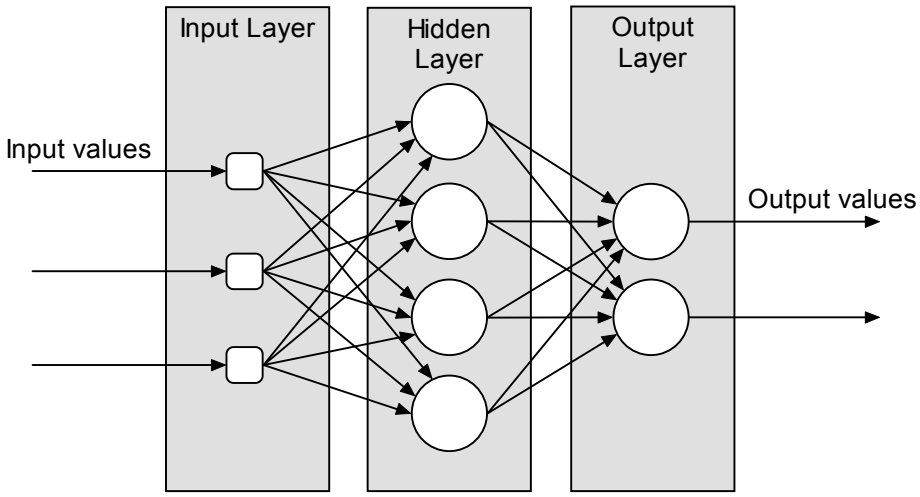

Figure 18: Example of a feed-forward network

Each input is sent to every neuron in the hidden layer and each hidden layer's neuron is connected to every neuron in the next layer. There can be any number of inputs to, and outputs from a feed-forward neural net. Also the number of neurons in each layer, as well as the number of hidden layers can be adjusted as needed, although one hidden layer typically suffices for most problems.

2.7.3 Training and memory of artificial neural nets

At its initial state a neural network is not very interesting as it has not been trained to solve any specific problem. As a result if you feed an artificial neural net with some input to start with, is the output not satisfying. That is because the weights associated with each neurons input are initially in a random state. All the weights associated with each connection to a neuron, therefore needs to be adjusted, such that the final net should be able to solve a specific problem. There are basically four ways an artificial neural net can get into a state where it starts to perform as desired.

- Supervised training
- Unsupervised training
- Reinforced learning
- Evolution

Supervised training is the most popular method for training neural networks. In supervised training an input and its associated desired output is provided. The network then processes the input, and compares its output against the provided and desired output. The connection weights within the network are then adjusted accordingly. The entire set of training data is shown to the network many times and the weights are adjusted each time, until the artificial neural net behaves as desired. One of the most popular algorithms for training neural networks are known as the back-propagation algorithm and explained in section 2.7.4.

In unsupervised training the network is provided with input but no desired output. The system itself must then decide on some features it will use to group the input data. In

the unsupervised learning process the network will attempt to generate a unique set of output for one particular type of input patterns.

Reinforced learning uses success, failure, reward, and punishment as indicators of the result. In reinforced learning the network does some action on the environment and gets some feedback from the environment. The learning system is then responsible for calculating the desired output for some input. Based on this are the network adjusted using the same method as in supervised training.

In the evolutionary approach genetic algorithms explained in section 2.6 is used to evolve the weights within the network. It is a requirement that the neural networks live in an environment with other networks. A fitness score is then calculated for each network depending on how well it is at solving the problem at hand. This fitness score is then used in the genetic algorithm when all the neural networks evolve to the next generation.

2.7.4 Back-propagation training algorithm

Probably the most popular methods for training a neural network, is the back-propagation algorithm, also known as the generalized delta-rule. It is a supervised training algorithm and therefore requires that both the input and its desired output are provided. The most significant strength of this algorithm is its ability to train multilayered feed-forward networks. Before the invention of this algorithm training was only possible for single layered feed-forward neural nets.

The adjustment of the input weights for each neuron is as follow:

$$\Delta w_i = \eta \cdot input_i \cdot \delta$$

- i : Index of neuron input
- w_i : Array of input weighs to neuron
- η : Learning rate (constant)
- $input_i$: Array of input values to neuron
- δ : Error term for neuron (see below)

If neuron is in output layer, its error term is calculated by:

$$\delta = output\ (1 - output)\ (target - output)$$

- output : Output from the neuron
- target : Desired output from the neuron

If neuron is in hidden layer, its error term is calculated by:

$$\delta = output\ (1 - output)\ (\Sigma\ \delta_k \cdot w_{ik})$$

- k : Designates a successor neuron
- δ_k : Error term for successor neurons

- w_{ik} : Array of input weights from current neuron to successor neurons

For each set of training data do:

1. Feed the network with input data to obtain its output.
2. Compute δ for each of the neurons in the output layer.
3. Compute δ for each of the neurons in the hidden layer(s) using δ from the next layer.
4. Calculate Δw for each neuron input and adjust weights accordingly.
5. Repeat step 1 until error at output layer is within desired margin.

2.8 Bayesian networks

A Bayesian network (BN) [3] and [6] is a graphical model that can be used for probabilistic reasoning that in relation to AI can aid the process of decision making. BN encodes probabilistic relationships among variables and are also sometimes called belief networks. More formally a Bayesian network is a directed acyclic graph with the following characteristics:

- A set of random variables makes up the nodes of the network.
- A set of directed links represented by arrows, representing normally causal relationships, connecting the nodes.
- Each node has a conditional probability table that quantifies its effect.

Consider an example where we have a burglar alarm. This alarm can be triggered either, by a burglary or an earthquake. If the alarm is triggered, your neighbor John has promised to call you. This can be illustrated as in Figure 19.

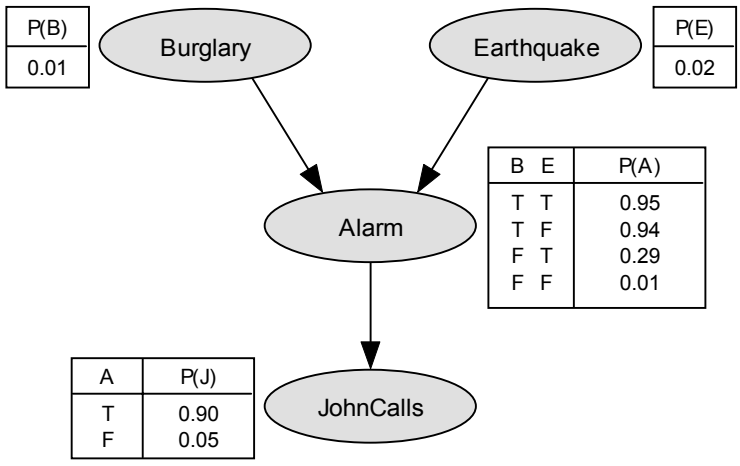

Figure 19: Illustration of a Bayesian network

Arrows are drawn from influencing nodes to influenced nodes. Influencing nodes are called parents of the influenced nodes. Influenced nodes are called children of the influencing nodes. The tables next to the nodes are called the conditional probability tables and represent the probability for the event represented by that node to happen. If the node has no parents it is called a root node and the conditional probability table simply represents the probability for that event to happen on its own. If the node is

children to some nodes, the conditional probability table holds the probability for that event to happen for each possible combination of outcome of its parents.

In the above example can it for example be seen that the probability for a burglary is estimated to 1%. If there is a burglary but no earthquake the probability for the alarm to get activated is 94%, while the chance for the alarm to get activated on its own is 1%. Should the alarm be activated there is 90% chance John calls. Although not explicitly illustrated, it can also be seen that the probability for John not to call, even though the alarm is sounding, is 10%.

2.8.1 Bayes theorem

Before continuing on Bayesian networks, let's look at some basic probability theory involving Bayes theorem. This is a rather simple but powerful equation for probabilistic calculations. Bayes theorem is as follow:

$$P(H \mid E) = \frac{P(E \mid H)\ P(H)}{P(E)}$$

- *P(H | E):* Given E, probability for H to happen
- *P(E | H):* Given H, probability for E to happen
- *P(H):* probability of the H to happen
- *P(E):* probability of the E to happen

P(E) can be calculated by the formula:

$$P(E) = P(E \mid H)\ P(H) + P(E \mid \neg H)\ P(1 - P(H))$$

- *P(E | ¬H)* is the probability for E to happen given H has not happened (¬H is called the false alarm rate)

Consider an example where someone is considering investing some money into a new company. The person knows that only 20% of all start-up companies succeed. The uncertainty can be reduced by asking for an expert opinion. A known fact about the expert is that of the companies that eventually succeeds, 40% is judged good prospects, 40% is judged moderate, and 20% is judged poor. It is also known that of all start-up companies that fails, 10% is judged to be good prospects, 30% to be moderate prospects, and 60% to be poor prospects.

Let's say we have the evidence that the expert judges the company to be good. We then wish to find the probability for the company to succeed. First the probability for the evidence good (*P(E)* in Bayes theorem) needs to be calculated, this is done as:

$$P(good = T) = 0.4 \cdot 0.2 + 0.1 \cdot (1 - 0.2) = 0.16$$

This can also be considered the probability that the expert will judge any new random company good.

We can now find the probability for the company to succeed using Bayes theorem as:

$$P(success = T \mid good = T) = \frac{0.4 \cdot 0.2}{0.16} = 0.5$$

So it is given that 20% of all start up companies succeeds. Our expert judge the company as good, 40% of all companies judged good succeed, and 10% fails. The probability for the company to succeed is then 50%.

The above example can be illustrated by the Bayesian network seen in Figure 20.

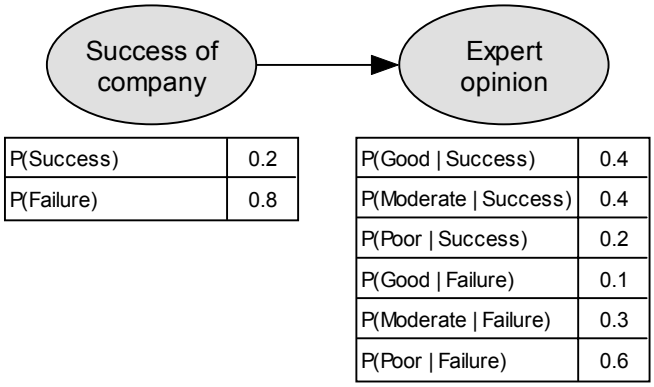

Figure 20: A simple Bayesian network

2.8.2 Joint probability distribution

The joint probability distribution (JPD) is the probability for a combination of specific assignments to each variable in the Bayesian network and it is defined as:

$$P(x_1,...,x_n) = \prod_{i=1}^{n} P(x \mid Parents(X_i))$$

Returning to Figure 19 is it for example possible to find the probability that John will call because the alarm has sounded, but neither a burglary nor an earthquake has occurred. The calculation can be seen below:

P(¬Burglary, ¬Earthquake, Alarm, JohnCalls)
= P(¬Burglary)P(¬Earthquake)P(Alarm | ¬Burglary, ¬Earthquake)P(JohnCalls | Alarm)
= 0.99 * 0.98 * 0.01 * 0.90 = 0.0087318

The Joint probability distribution for all combinations in the Bayesian net illustrated in Figure 19, can be seen in Table 10.

Row	Burglary	Earthquake	Alarm	JohnCalls	JPD
1	True	True	True	True	0.000171
2	True	True	False	True	0.0000005
3	True	False	True	True	0.0082908
4	True	False	False	True	0.0000294
5	False	True	True	True	0.0051678
6	False	True	False	True	0.0007029
7	False	False	True	True	0.0087318
8	False	False	False	True	0.0480249
9	True	True	True	False	0.000019
10	True	True	False	False	0.0000095
11	True	False	True	False	0.0009212
12	True	False	False	False	0.0005586
13	False	True	True	False	0.0005742
14	False	True	False	False	0.0133551
15	False	False	True	False	0.00009702
16	False	False	False	False	0.92076831

Table 10: Joint probability distributions for all combination in Figure 19

2.8.3 Inference

Inference is the computational method for deriving answers to queries in a Bayesian network. An example could be we simply want to know the probability that John will call *P(JohnCall)*. This query can be answered by taking the sum of all the rows where JohnCalls is true. We sum row 1-8 and obtain:

$$0.000171 + ... + 0.0480249 = 0.071119$$

This can be illustrated by the following equation (Only the first letter of a variable name is used. As an example does the letter J detonates the variable JohnCalls):

$$P(J = True) = \sum_{Burglary, Alarm, Earthquake} P(B) \cdot P(E) \cdot P(A \mid B, E) \cdot P(J = True \mid A)$$

Two other common types of inference are:

- **Casual inference** from causes to effects. Given a Burglary what is the probability that John calls *P(JohnCalls* | *Burglary)*.

- **Diagnostic inference** from effects to causes. Given John Calls what is the probability for a burglary *P(Burglary* | *JohnCalls)*.

Casual inference example

Using the joint probability distribution in Table 10, is it possible to answer the query *P(JohnCalls* | *Burglary)* which is the probability that John will actually call when a burglary is being made. The computation can be written as:

$$P(J = True \mid B = True) = \frac{\sum_{Alarm,\, Earthquake} P(B = True) \cdot P(E) \cdot P(A \mid B = True, E) \cdot P(J = True \mid A)}{\sum_{JohnCalls,\, Alarm,\, Earthquake} P(B = True) \cdot P(E) \cdot P(A \mid B = True, E) \cdot P(J \mid A)}$$

That is the sum of the rows in Table 10 where Burglary and JohnCalls is true divided by the rows where Burglary is true. This is the sum of row 1-4 divided by the sum of row 1-4 and 9-12. The result is:

$$\frac{0.000171 + ... + 0.0000294}{0.000171 + ... + 0.0000294 + 0.000019 + ... + 0.0005586} = 0.84917$$

So there is around 85% chances that john will actually call when a burglary is being committed.

Diagnostic inference example

Diagnostic inference can be calculated in the same way as above. It is possible to answer the query *P(Burglary | JohnCalls)* which is the probability for a burglary is being done given that John calls. The computation can be written as:

$$P(B = True \mid J = True) = \frac{\sum_{Alarm, Earthquake} P(B = True) \cdot P(E) \cdot P(A \mid B = True, E) \cdot P(J = True \mid A)}{\sum_{Burglary, Alarm, Earthquake} P(B) \cdot P(E) \cdot P(A \mid B, E) \cdot P(J = True \mid A)}$$

Which is the sum of the cells where Burglary and JohnCalls is true, divided by the cells where JohnCalls is true. This is the sum of row 1-4 divided by the sum of row 1-8. The result is:

$$\frac{0.000171 + ... + 0.0000294}{0.000171 + ... + 0.0480249} = 0.119401$$

So receiving a phone call from john, the probability for a burglary is just slightly below 12%.

2.8.4 Inference by set-factoring

As it can be seen in the above examples, the calculation of a joint probability table quickly becomes a cumbersome task as the size of a Bayesian network increases. If another node with two possible values for example were added to the network, the size of the JPD table would increase to twice its size and contain 32 values. The added node could be another neighbor that also has promised to call if the alarm was sounding. Unfortunately the kind of inference presented above has been proved to be NP-hard, and as a consequence are other methods for calculation needed if inference is to be done on Bayesian networks of reasonable size. A whole range of exact and approximate inference algorithms exists in the literature. Here a heuristic algorithm that performs exact inference known as set-factoring [6] will be presented.

Set factoring works by taking pairs of nodes in the Bayesian network and combining them. It begins with an initial set of node combinations, selects a pair to combine, removes the pair from the set and places the result back in the set. The process

continues until there is only one node left that holds the answer to the query. The set to combine in each step is the one with the smallest result table. If there is a tie the pair to combine is the one with the fewest parameters represented in the query and other candidate pairs.

Let us consider the example where we want to know the probability for John to call.

In the first step we have four nodes, which give six candidate pairs. Three of them share at least one parameter. The three pairs that share parameters are:

1. P(Earthquake) * P(Alarm | Burglary, Earthquake) => P'(Alarm | Burglary)
2. P(Burglary) * P(Alarm | Burglary, Earthquake) => P'(Alarm | Earthquake)
3. P(Alarm | Burglary, Earthquake) * P(JohnCalls | Alarm) => P'(JohnCalls | Burglary, Earthquake)

Which one of the candidate pairs 1 or 2 is combined, does not matter. Below we combine candidate pair 1 to obtain the new conditional probability table for *P'(Alarm | Burglary)*, calculations are included for clarity:

Burglary	**P(Alarm)**
True	Earthquake = True: (0.02 * 0.95) + Earthquake = False: (0.98 * 0.94) = 0.9402
False	Earthquake = True: (0.02 * 0.29) + Earthquake = False: (0.98 * 0.01) = 0.0156

Table 11: Calculation of conditional probability table for alarm when eliminating earthquake

The Earthquake node has been eliminated, and we instead have an alarm node, where in the case of a burglary, the chance that the alarm will be activated is 94.02%, and a 1.56% chance that the alarm will be activated when there is no burglary.

There are three nodes left in the Bayesian network, which give three candidate pairs. Two of them share at lest one parameter. The two are:

1. P(Burglary) * P'(Alarm | Burglary) => P'(Alarm)
2. P'(Alarm | Burglary) * P(JohnCalls | Alarm) => P'(JohnCalls | Burglary)

Again pair 1 is combined to obtain *P'(Alarm)*:

P(Alarm)
Burglary = True: (0.01 * 0.9402) + Burglary = False: (0.99 * 0.0156) = 0.024846

Table 12: Calculation of conditional probability table for alarm when eliminating burglary

The burglary node has been eliminated and a new alarm node where the probability for the alarm to be activated is 24.846%, is replacing the old alarm node.

There are now two nodes left in the network which give the pair to combine as seen below:

1. P'(Alarm) * P(JohnCalls | Alarm) => P'(JohnCalls)

The probability that John will call can be obtained by combining the last two nodes to obtain the conditional probability table for simply for john to call:

P(JohnCalls)				
Alarm = True		Alarm = False		
(0.024846 * 0.9)	+	(0.975154 * 0.05)	=	0.0711191

Table 13: Final calculation of probability for John to call

This result matches the result obtained in section 2.8.3 for the exact same query but the amount of calculations is far fewer as the calculation of a conditional probability table is skipped. The whole process is illustrated in Figure 21.

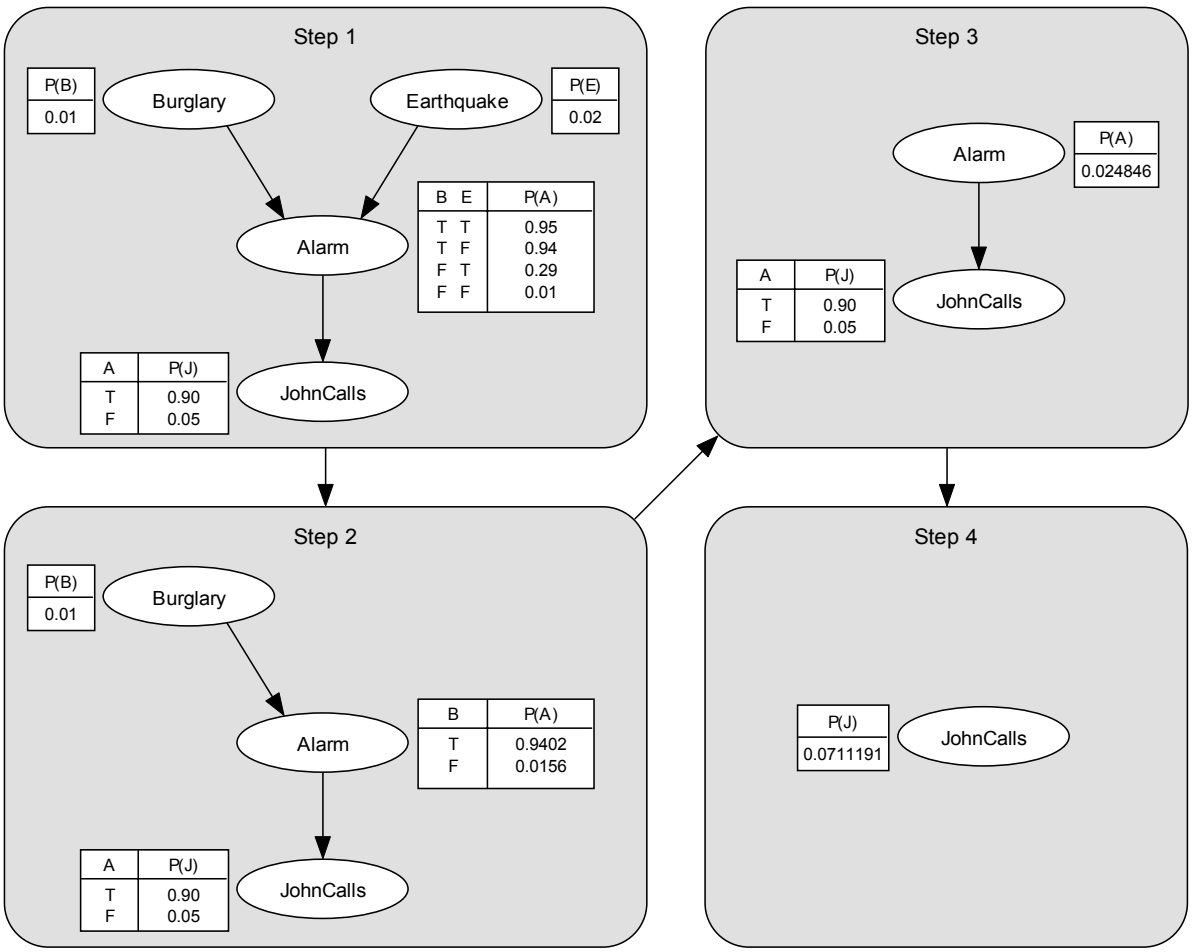

Figure 21: Illustrated example of set-factoring

2.9 Conclusion on artificial intelligence

Artificial intelligence is not about inventing the brain. It does not look like the creation of a human mind in software is a serious research area. That could be because people still see it as impossible and accept there is something about the human mind that with our understanding currently cannot be duplicated, something that perhaps can be called a soul.

Artificial intelligence can be viewed as a way to solve problems that do not have a straightforward solution. AI can also in many ways be viewed as a way that seeks to code human knowledge into software. The success seems to depend a great deal on the programmer obtaining (or helping the program to obtain) expert information, and making it correctly available to the program.

The methods used in AI spans over a range of different techniques, but they basically seem to be in at least one of two main categories. That is the category of learning or the category of supporting decision making in either a more human like or rational way.

3. Game theory

This chapter will give an overview of basic game theory. Much theory of games concerns itself with what is known as the Nash equilibrium defined as a situation in which each player makes its best response (maximizes its score), given the actions of rival players. The section will also try to define what games actually are, and how games theoretically can be solved. It should also give an overview of why most games cannot be solved in practice.

The following sections are contained in this chapter:

- Classification of games
- Game trees
- Agents
- Conclusion on game theory

3.1 Classification of games

Before classifying games, is it necessary to define what a game is. In [40] the following definition is found:

> A game is defined as a decision problem with two or more decision makers – players – where the outcome for each player may depend on the decisions made by all players. Each player evaluates possible outcomes in terms of his own utility function, and works to maximize his own expected utility only.

At a high level, games can be divided into three main categories as illustrated in Figure 22. This is however only the main categories as hybrid games also exist. Football can be considered a game requiring some of all elements.

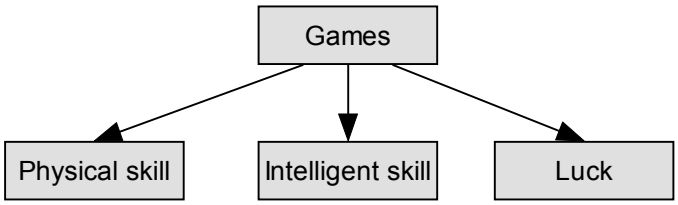

Figure 22: Main classification of games

Games interesting for artificial intelligence can mostly be categorized in the intelligent games category. These types of games are normally subdivided along two dimensions of stochasticity and information [40].

Stochastic games are games involving some form of random variable, chance or probability. Games involving a dice and many card games are in this category. The opposite is called deterministic games, and they do not contain any random effect.

The other dimension classifies games as either perfect-information or imperfect-information games. In a perfect information game all players have the same knowledge about the current game state including its history. In perfect information

games there is one best way to play for each player. This will not necessarily guarantee a win, but will minimize the defeat. The trouble with many perfect-information games is however that the number of strategies is so large, that it is not possible to determine the best one. In imperfect-information games, players cannot know the result of their actions as the whole state of the game is not known. Often this occurs because players play simultaneously.

Table 14 is summarizing the classification of games along with some example games.

	Deterministic	**Stochastic**
Perfect-information	Chess	Backgammon
Imperfect-information	Paper-rock-scissor	Poker

Table 14: Classification of example games

There exists further classification of games known as Zero-Sum and Non-Zero-Sum games. Section 3.1.1 and 3.1.2 will explain these classifications.

3.1.1 Zero-Sum games

In a zero-sum game [30] no wealth or points is created or destroyed. That means that in a two player zero-sum game, what one looses the other win. The players do therefore not share any common interest. In a zero-sum game an optimal solution for a player always exists. In zero-sum games, is it sometimes possible to reach a state where there exist an optimal strategy as explained below.

Dominant Strategy

A dominant strategy is a strategy that will always be the best, no matter what the other opponents do.

Consider a simple two player game played with nickel and quarters. In each round they both select one coin to play. If at least one player plays a nickel, player *A* win both coins, otherwise player *B* win both. Using the payoff matrix illustrated in Table 15, it is possible to find the dominant strategy for player *A*.

		Player B	
		Nickel	Quarter
Player A	Nickel	5	25
	Quarter	5	-25

Table 15: Payoff matrix for player A

It can be seen that that the largest payoff possible for player *A* if playing a quarter is 5. This is equal to the smallest possible payoff if playing a Nickel. In other words, playing a nickel is always at least as good as playing a quarter. So playing a nickel is called the dominant strategy.

Saddle point

Saddle point is a strategy, such that a player has no motivation for changing it because a change by any player would lead the player to possibly earn less than if she remained with the current strategy.

Changing the rules from the previous game such that if player *A* plays a nickel, Player *B* gives player *A* a nickel. If player *B* plays a nickel and player *A*, plays a quarter, player *A* gets a quarter. If both play quarters, player *B* gets a quarter from player *A*. The payoff matrix for player *A* can be seen in Table 16.

		Player B	
		Nickel	Quarter
Player A	Nickel	5	5
	Quarter	25	-25

Table 16: Player A payoff matrix

This game does not have a dominant strategy for player *A*, so another approach is needed to find the best strategy to play. To find the strategy that guaranties the largest payoff for player *A*, the minimum value is to be found in each row. In the example the lowest value in the first row is 5 while the lowest value in the second row is -25. The best strategy to play is then taking the row with the maximum of the minimum values. So the best strategy for player *A*, is always playing a nickel. This is called the saddle point.

3.1.2 Non-Zero-Sum games

Non-zero-sum games [30] are games where one person gain necessarily not comes at expense of someone else's. In zero-sum games, one person must lose for another to win. In non-zero-sum games the supply of wealth or points, is not fixed or limited. It is a case of building a resource rather than dividing it. In some non-zero-sum situations, a person can only benefit from a situation when other benefits as well. A classical example illustrating a non-zero-sum game is the prisoner's dilemma proposed by Merrill Flood in 1951.

Prisoner's dilemma

Two suspects are arrested by the police. The police do not have enough evidence for a conviction. The suspected has been separated and each of them is offered the same deal: If you confess and your partner does not, he gets a 15 year sentence and you go free. If he confesses and you do not, you get a 15 year sentence and he goes free. If none of you confess we can only give both of you 1 year for a minor charge. If you both confess, you both get 5 years. The dilemma is illustrated in Table 17.

	He confess	**He denies**
You confess	Both get 5 years	He gets 15 years, you go free
You deny	He goes free, You get 15 years	Both gets 1 year

Table 17: The classical prisoner's dilemma

What should each prisoner do? If we assume there is no honor among criminals, each prisoner will typically reason that it is best to confess, as he does not know what the other will do. But although each of them has done what seemed to be best for them, they could have gained a better result by remaining silent.

3.2 Game trees

Game trees are described in [27] and [42]. One straight forward way to make a computer solve a given problem is simply to let it search for the best solution. This method is known as brute force, and can in games normally be represented as an n-ary tree where each node represents a status of the game. Such a tree is normally called a game tree. The root node represents the current state of the game. The root nodes children are all the possible moves from the current position. The children to these nodes are then the possible next moves for these game states. This is continued all the way down to the leaves of the game tree. Each leave represents a terminal position where the outcome of the game is decided.

3.2.1 Min-Max search trees

A Min-max search tree can be used to analyze the best possible next move for a player. Each leave must have a score. For example could 1 be associated with a win, 0 with a draw and -1 with a loss. Min-max works simply by examining the entire tree, and picking up the best path that can be forced.

To explain a Min-max game tree, an example using the game of NIM will be given. The rules of NIM are as follow: The game board consists of piles of sticks. The players take turns removing any number of sticks from a single pile. The person removing the last stick looses. Figure 23 illustrates a game of NIM by use of matches. This game has three piles of sticks. In the left most one stick is present. In the middle and right most pile, there are two sticks. This can also be represented as (1, 2, 2).

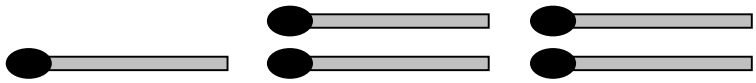

Figure 23: A game of NIM consisting of one pile with one stick and two piles with two sticks

Using a Min-max search tree is it possible to analyze every move. The Min-max search tree for the game of NIM with the (1, 2, 2) configuration, and two players, can be seen in Figure 24. At the leaves a victory for the current player, is illustrated by the value 1, while a loss is illustrated by a 0. The two emphasized paths in the central branch are the ones that can force a victory for the player about to make a move.

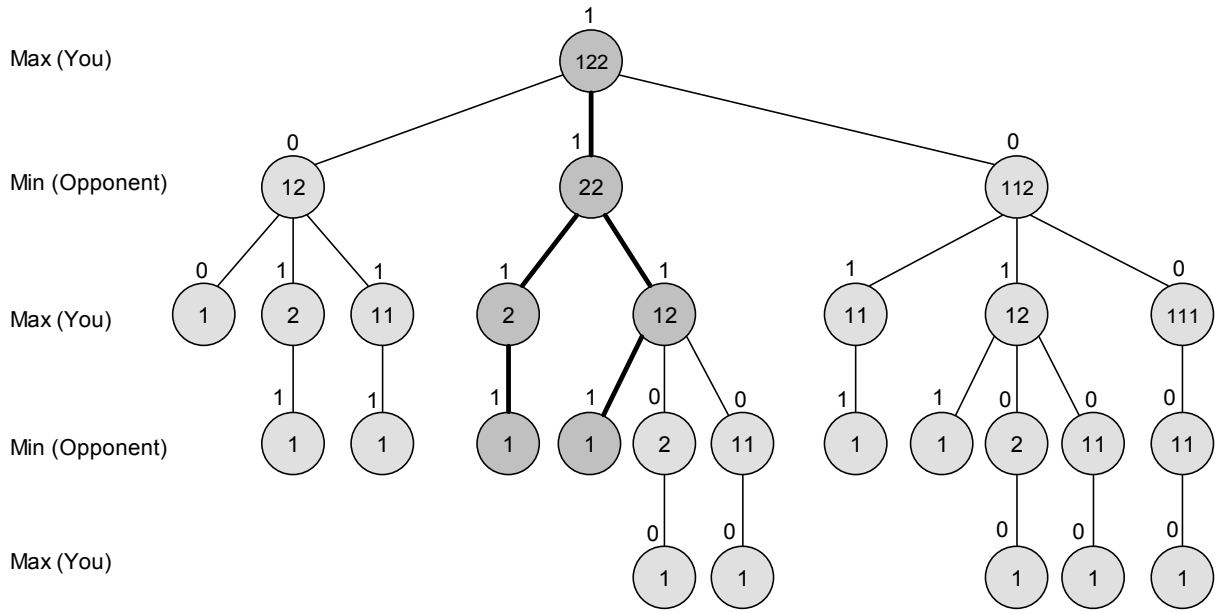

Figure 24: Game tree for a (1, 2, 2) game of NIM

The algorithm works by recursively traverse downwards the tree. In a two player game, all nodes positioned at an odd depth is called *Max* nodes, nodes at an even depth are called *Min* nodes. If the traversed node is a *Max* node its value becomes the value of the largest of its children. If the traversed node is a *Min* node its value becomes the smallest value of its children. The goal is then in each turn to take the branch with the highest value. Pseudo code for the algorithm can be seen below.

```
MinMax( Node node )
{
   if ( node is a leaf  )
      return score of node;
   generate legal next moves;
   if ( node is a min )
      for ( all children of node: c1, .. cn )
         return Min{ MinMax( c1 ), .. , MinMax( cn ) };
   else if ( node is a max )
      for ( all children of node: c1, .. cn )
         return Max{ MinMax( c1 ), .. , MinMax( cn ) };
}
```

3.2.2 Bounded look ahead

The problem with the Min-max approach and other brute force methods in general, is that their time complexity often grows exponentially.

For example in chess it has been estimated that each position in average has 35 legal moves. So if min-max is used to search one move ahead, 35 positions are to be examined. To look two moves ahead 35^2 positions are to be examined. As it can be seen in Table 18 requires a six move look ahead, almost two billion positions to be examined. It is in other words impossible to search all nodes in a large search tree.

Moves ahead to look	Number of boards to examine
1	35
2	1,225
3	42,875
4	1,500,625
5	52,521,875
6	1,838,265,625
7	64,339,296,875
8	2,251,875,390,625
9	78,815,638,671,875
10	2,758,547,353,515,625

Table 18: Estimated amount of boards to examine in a game of chess

One solution to this problem is simply to trim the tree at a given level. Depending on the time available a three, four or five move look ahead could be possible in a game of chess. Each of the nodes at that level is then evaluated by some heuristic function.

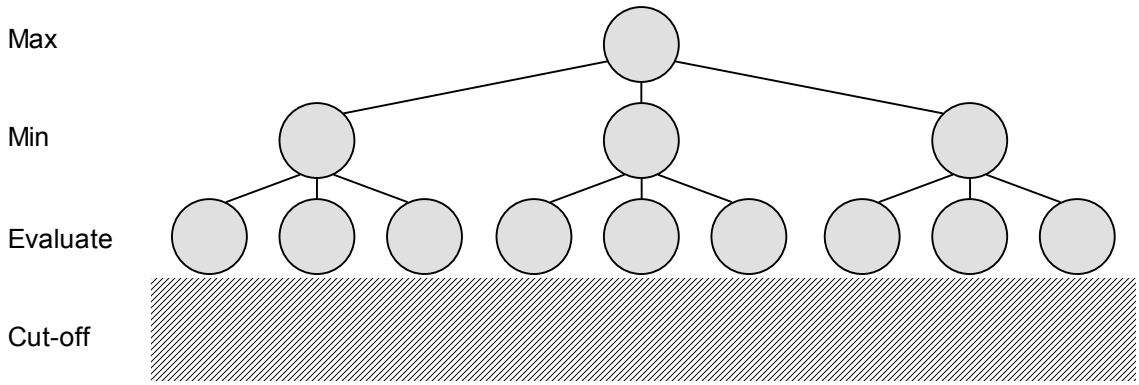

Figure 25: Bounded look ahead in a game tree

The evaluation function is constructed by an expert user and based on insight and experience. The evaluated nodes are then to function as the leaves. For all the nodes above this level, the standard Min-max approach still applies. The problem with bounded look ahead, is that the evaluation function is not exact; it is just a heuristic function making an educated guess. Pseudo code for the algorithm can be seen below.

```
BoundedMinMax( Node node, Int depth )
{
   if ( depth is zero or node is a leaf  )
      return evaluated score of node;
   generate legal next moves;
   if ( node is a min )
      for ( all children of node: c1, .. cn )
         return Min{BoundedMinMax( c1, depth - 1 ), .., BoundedMinMax( cn, depth - 1 )};
   else if ( node is a max )
      for ( all children of node: c1, .. cn )
         return Max{BoundedMinMax( c1, depth - 1 ), .., BoundedMinMax( cn, depth - 1 )};
}
```

3.2.3 Alpha-Beta search

If the goal is to create the strongest possible chess player, is it important to search as deep as possible. To search deeper the branching factor must somehow be reduced. This can be done by use of an Alpha-beta search. The Alpha-beta search relies on the idea that if a choice is known to be bad for the player, then other choices are examined instead. Just like in the Min-max algorithm, is the tree recursively traversed downwards. But in the Alpha-beta search, two scores are passed around in the search. That is the alpha value and the beta value.

The alpha value is the score that by any means can be forced. Anything that has a value equal to or less than this can be ignored. That is because there already is a known strategy that can produce the same or better result.

The beta value is the worst situation for the opponent. If the search finds something that produces a score equal to or better than beta for the opponent, it is too good. So the object for the player is not to make a move in that direction, and thereby deny the opponent the chance of using this strategy.

So if a move results in a score greater than alpha but less than beta, this move is to be searched further. During the search the values of alpha and beta are continually updated. If none of the legal moves produces a result that fulfill these requirements, is it avoided by making a different choice somewhere above in the search tree. In Figure 26 an Alpha-beta search is illustrated.

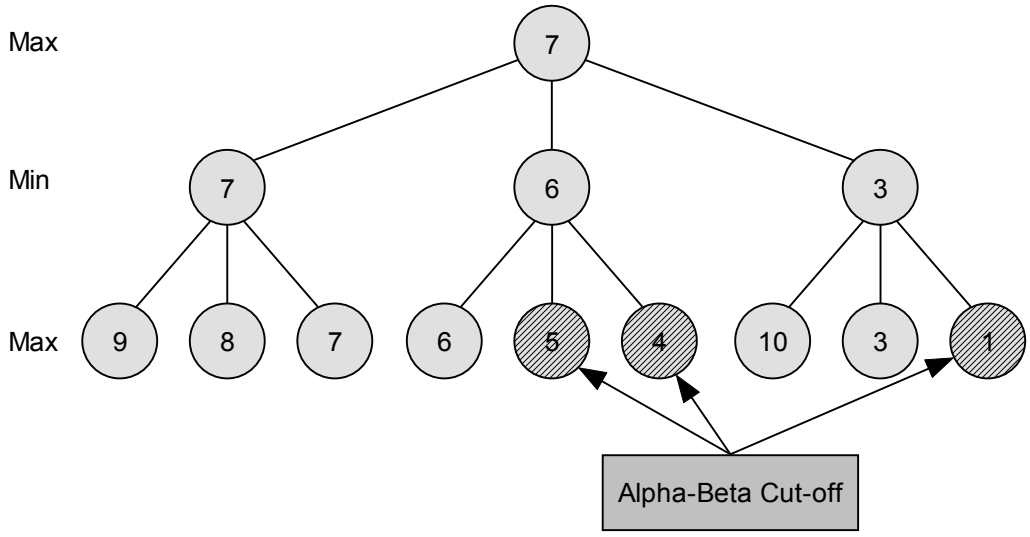

Figure 26: Alpha-beta search in a game tree

It can be seen that the nodes containing the values 5, 4 and 1 can be ignored. This is because already by examining the nodes containing the values 6 and 3 is it clear that the left-most branch will be a better move. To put it another way: if we follow the middle or right most branches, we know our opponent can force us to end the game with either 6 or 3 points. However taking the left most branches, the opponent is only capable of forcing a score of 7 points for us.

The problem with the Alpha-beta algorithm is that its performance depends on the order in witch moves are searched. If it always is the worst move that is examined first, a cut-off will never be achieved, and the algorithm works exactly in the same way as the Min-max algorithm.

However in the most favorable circumstances where the best move is always picked first, the expected branching factor is approximately reduced by the square root. This means that in the most favorable case the branching factor of chess can be reduced from 35 to 6. This will allow the algorithm to search almost twice as deep as the Min-max algorithm. Pseudo code for the algorithm can be seen below.

```
AlphaBeta( Node node, Int beta )
{
   if ( node is a leaf  )
      return score of node;
   generate legal next moves;
   if ( node is a min )
   {
      alpha = +infinity;
      for ( all children c of node )
      {
         value = AlphaBeta( c, beta );
         alpha = Min{ value, beta }
         if ( alpha <= beta )
            exit loop;
      }
      return alpha;
   }
   else if ( node is a max )
   {
      alpha = -infinity;
      for ( all children c of node )
      {
         value = AlphaBeta( c, beta );
         alpha = Max{ value, alpha }
         if ( alpha >= beta )
            exit loop;
      }
      return alpha;
   }
}
```

3.3 Agents

Before defining what an agent is in game theory a definition on the more general term computer agent, should be presented first. In [22] it is however argued that the term agent is in danger of becoming a noisy term that is subject for both confusion and misuse without an exact definition. They have however tried to come up with a weak definition of the term agent.

A general way to define the term agent is computer system that has the following properties [22]:

- **Autonomy**: Agents operate without the direct intervention of humans or others, and have some kind of control over their actions and internal state.
- **Social ability**: Agents interact with other agents (and possibly humans) via some kind of agent-communication language.
- **Reactivity**: Agents perceive their environment, (which may be the physical world, a user via a graphical user interface, a collection of other agents, the internet, or perhaps all of these combined), and respond in a timely fashion to changes that occur in it.
- **Pro-activeness**: Agents do not simply act in response to their environment; they are able to exhibit goal-directed behavior by taking the initiative.

It is however further argued that in the artificial intelligence community the following stronger definition also applies:

“An agent is to be a computer system that, in addition to having the properties identified above, is either conceptualized or implemented using concepts that are more usually applied to humans. For example, it is quite common in AI to characterize an agent using mentalistic notions, such as knowledge, belief, intention, and obligation. Some AI researchers have gone further, and considered emotional agents.” [22]

3.3.1 Game agent

In the context of games an agent can be defined as an entity with goals, plans of action and preferences that can interact with other agents. An agent can be conceptualized as a person, animal, firm, country, etc. An agent has the ability to decide on a sequence of actions that hopefully are more correct then actions made by other agents. Agents involved in games are referred to as players and assumed to be rational when selection actions necessary to reach its goal.

In the context of this thesis the term agent referees to the part of the backgammon application to be developed, that is responsible for actually deciding the moves and thereby playing the game.

3.4 Conclusion on game theory

Game theory can in many ways be viewed as the theory of social situations. Some situations can be solved by simply finding the solution, while other problems are unique and requires overview and political skills to be dealt with in the best way. As in the real world, depending on the situation different skill is needed. Just like different games require different skill.

The ability to predict situations, as game trees can be used for, is a reasonable ability in AI related problems. A human also possesses the ability to predict situations. However where humans have an ability to select some limited but interesting situations for a more deep analysis, the computer instead has to select a much larger amount for analysis. In either way it is limited how many steps ahead it is possible to look for both humans and computers. As such brute force and thereby the ability to

predict, can be viewed simply as yet and ability for intelligence. However relying on it alone will make it impossible to solve many real life situations.

As also illustrated in the chapter, not all problems are of a type where prediction is the key to success. Other factors such as randomness, probability and luck as well as decisions made by other, also has a key role in many situations. As such ability to learn and reason is to be important properties in many cases, if the goal is to develop an intelligent game agent.

4. Related work

This chapter will try to establish an overview of what AI in relation to games is. The chapter will first and foremost look at four very successful but also very different efforts. At the end the chapter will also hold a section giving a brief overview of some additional methods that has been used when creating different programs able to play board games.

This chapter consists of the following sections:

- Deep Blue
- Chinook
- Evolutionary Checkers
- TD-Gammon
- Survey of some additional related work
- Conclusion on related work

4.1 Deep Blue

Deep Blue [21] and [26] is a chess (brief description in appendix section 10.3.1) computer developed by IBM researchers that in 1996 and 1997 played against the world champion of chess Garry Kasparov. While it in 1996 was defeated, it chocked the chess world by winning in 1997. Although IBM themselves said that the power behind Deep Blue is its ability to calculate 200 million positions in a second, is its foundation for success not to be found in its computing power alone. This can be seen by the fact that in the year between the two matches, did Deep Blues computing power only increase by a factor of two. Noticeable advances were instead obtained in Deep Blue's chess knowledge. This knowledge is gained from two databases. One called the opening book and one called the extended book.

The Databases

The opening book contains information about the opening moves that has been estimated to be best. An opening book is a set of positions along with recommended best moves. An opening book is often used to guide the early stages of a chess game.

The extended book is a database of positions that has arisen in around 700,000 real grandmaster games. For each of the positions contained in this database, is it able to compute a value for the moves. A move that seems good is assigned a higher value then moves that seems less good. Different factors contribute to the final score of a move.

- The number of times a move has been played.
- Relative number of times a move has been played. If move A has been played more then Move B, move A is expected to be better.
- Strength of the player that play the move.
- Recentness of a move. Recently played moves are likely to be better as possible flaws are yet to be discovered.
- Result of the move.

- Commentary on the move. Good moves are often annotated with “!” while weak moves are annotated with “?”.
- Game moves vs. commentary moves. Annotators of chess games sometimes suggest alternate moves. Game moves are considered more reliable.

Making a move

Before making a move Deep Blue first checks if a move is available in the opening book. If it finds a move it is played immediately to save time for use later in the game. If a move is not found, it consults the extended book. If the position is found there, it starts to award bonuses to the available moves. If a move with a high enough score is found, the move is carried out immediately, again to save time for later use in the game. If no specific move with a special high score is found, Deep Blue carries out a narrow search biased on the best scoring moves in the extended book. This is because there is the possibility that a stronger move has been overlooked. Deep Blue then has the chance of discovering it. The final move is then found by combining search results, and the results calculated from the extended book. In the event that a move can not be identified in any of the two databases, Deep Blue relies solely on its own search and evaluation algorithm. This is why human chess experts when playing computers, often at some point makes a move that for a human observer can be considered bizarre. The hope is to trap the chess computer in a situation where it only has its search algorithm to rely on. This strategy did however obvious fail for Kasparov.

4.2 Chinook

Chinook [18] is a checkers (brief description in appendix section 10.3.2) program developed at the University of Alberta, Canada. Although it might not be as well known as Deep Blue, is its performance is by no mean less remarkable. Currently Chinook holds the world championship title of Checkers, however if Chinook really is the best checkers player of our time, remains unknown. Chinook was set to play for the world title August 15, 1994. The match was against the seemingly unbeatable World Checker Champion Marion Tinsley. Tinsley was considered to be the ultimate checkers player, far superior to other human competitors. It was due to the lack of any real human challengers, Chinook was allowed to play for the title. Tinsley simply wanted to play someone able to give him proper resistance. After six games the match was drawn. However before the seventh match Tinsley resigned due to health concerns. In accordance with the rules was Chinook declared world champion. One week later Tinsley was diagnosed with cancer. He died April 3, 1995.

Chinook consists of four aspects:

- Search algorithms for traversing through the $O(10^{20})$ search space.
- Evaluation function to decide how good a position is.
- Endgame database holding perfect information for which endgame positions are won, lost and drawn
- An opening book database containing good openings to start a game with.

Search

Chinook uses a parallel alpha-beta search algorithm with minimum search depth of 21 moves. This ability has been obtainable because Chinook uses a search algorithm that has been adjusted such that it can select interesting lines of play for a much deeper search, while ignoring other less interesting lines of play.

Evaluation

The evaluation function consists of around 24 components, each containing several heuristic weights. Attempts to tune these weights using neural nets and genetic algorithms have been tried. They experience is however that the results are less successful then a hand tuned evaluation function. The evaluation function has been the biggest source of errors in Chinook. Several bugs have been discovered over the time, also during tournament play.

Endgame database

Chinook includes a complete 8 piece endgame database. Every possible combination of play with 8 or less checkers left on the board has been completed. This database holds 440 billion positions stored in a 6 gigabyte database.

Opening book

The opening book is a database containing the opening plays that has been published in the literature over the years. After the initial database had been build was several months used to let Chinook check the positions for errors. As a result where several hundred positions adjusted due to discovered errors.

4.3 Evolutionary Checkers

Evolutionary checkers [8], [9] and [19] is a checkers game developed bye Kumar Chellapilla and David Fogel at the University of California. Although it does not hold the world championship of computer checkers, it has turned out to be quite successful. The approach taken in creating the checkers player differs from the approach taken in creating Chinook and Deep Blue. They argue that Chinook and Deep Blue can not be considered intelligent as everything they know, it knows because they ware explicitly told. Chinook and Deep Blue learn nothing on its own, and how can a system that never learns be considered intelligent.

The challenge when developing evolutionary checkers was to create an algorithm that instead of relying on human expertise, explicitly put into the program, had the ability to learn the game simply by playing it. The hope was that after several games a competent algorithm capable of playing checkers would emerge.

The experiment

The approach taken was to use evolutionary computing to evolve neural networks that would represent strategies for the game of checkers. 30 neural networks where initialized in a random state, and then set to compete against each other using co-evolution for 250 generations. In each generation the neural net played a series of

games. The 15 best scoring neural networks where then maintained as parents for the next generation.

The neural network used was a standard feed forward network with 32 input nodes, corresponding to the 32 possible positions of a board. It had two hidden layers consisting of 40 and 10 nodes and one output. The input to the neural net consisted of 32 values that could have the values from {-K, -1, 0, 1, K}. K represents a king (initially set to 2.0), 1 a checker and 0 an empty square. Positive indicates the piece belongs to a player, negative that it belongs to the opponent. A move was determined by feeding the neural net with possible new board positions from the current one. The output was then a value in the range of -1 to 1. The higher the value, the better the board was estimated to be. The highest scoring board was then selected as the move.

Results

The best evolved network was used to play against human opponents on www.zone.com. Human player's login and then play against other human opponents. Initially new players are assigned a rating of 1600. The rating then increase or decrease depending on a player's performance against other players. Over a two week period, 90 games where played, none of the opponents where told they where playing with a computer. The final rating for the neural network was 1901.98 rating it as a class 'A' player. It easily defeated players rated below 1800, was playing even with players rated between 1800 and 1900, but was not able to match players rated more than 2000, which was in the expert and master categories.

4.4 TD-Gammon

TD-Gammon [15] is a backgammon (rules are described in section 6.2) program developed by G. Tesauro at IBM research. Backgammon programs based on the principles used in TD-Gammon is by far the most successful backgammon programs developed. As evolutionary checkers, TD-Gammon is based on a neural network based evaluation function. It does however take a different approach when learning.

Architecture

First of all TD-Gammon is not based on one, but two neural networks. One for the phase of the game where there still is player contact, and one for the phase called race. The neural networks architecture is feed forward networks which as input takes the number of white and black checkers at each location. The neural nets have four output values indicating whether white or black is estimated to win a normal victory or a gammon. Due to its rarity, are backgammon victories ignored.

TD-Gammon does however not rely solely on a neural network to play. Explicitly programmed features are also put into the program. This includes such features as evaluation of the strength of a blockade, probability of being hit, and a search algorithm looking up to three moves ahead.

Training

Where evolutionary checkers relied on evolution to create a capable player, TD-Gammon relies on a reinforced learning style known as temporal difference. Temporal

difference works in such a way that the current value of the previous state is adjusted to be closer to the value of the newly created state using the following formula:

$$V(s) = V(s) + \alpha[\, V(s') - V(s) \,]$$

- s detonates the state before the move that has just been done.
- s' is the current state the move has brought us in.
- α is a small positive value called step-size parameter which influence the learning rate.

Initially all boards are evaluated random except for the boards where the outcome of the game is decided. These boards are evaluated to either 1 (win) or 0 (loss).

Let us consider an example where we have just played a board with the evaluated score of 0.2 and where the previously played board before that was evaluated to a score of 0.4. If the step-size parameter then is 0.5 the new score for the previous board can be calculated as:

$$0.3 = 0.4 + 0.5(0.2 - 0.4)$$

This is done after each move and the weights within the neural network are adjusted using the back propagation algorithm explained in section 2.7.4, such the evaluation of the previous board is changed from 0.4 to 0.3, which is a little closer to the subsequent move evaluated to 0.2. At the end of each game a final adjustment is done to all played boards. During training the neural net is used to select moves on both sides.

Results

TD-Gammon has been tested against top world players in a series of exhibition games at the world cup of backgammon. Rather surprisingly, the general experience was that the top players of the world were only able to pull some very narrow victories. The general opinion is that TD-Gammon plays at a strong master level where most commercial programs play at a weak intermediate level. It is even believed that in long game sessions or tournaments the game could win as it does not get tired as humans do. Today the worlds two strongest available backgammon programs BGBlitz (shareware) and Gnu-backgammon (Open source and freeware) are both based directly on the same principles as TD-Gammon. Temporal difference learning has however been tested in a range of other domains such as chess and checkers without much success.

4.5 Survey of some additional related work

This section will try to give a very brief overview of some additional techniques, not mentioned in the previously four sections, that has been used when creating intelligent game agents.

The pioneer in the field of artificial intelligence and games is Arthur L. Samuel [4]. In 1947 he decided to build a checkers program, challenge the world champion and beat him. For more then twenty years he worked on the program and produced a master

level checker player. Other researchers have however later questioned its real strength and indicating that its strength was overrated. Never the less has the program pioneered an important technique like alpha-beta search. It also featured the first successful attempt of automatic evaluation function tuning. The checker program did also remember positions it encountered. This is known as rote learning and allowed it to save time when performing searches. It did also implemented a scheme for forgetting positions again, it rarely encountered.

An often used method to govern the early stages of a game is by the use of an opening book. Samuel did already use an opening book in his checkers program. There might however be cases where the opening book has more then one alternative in a given situation. The problem is then to learn which opening variation is most successful against a certain opponent. Robert Hyatt [24] has solved this problem in his chess program known as Crafty by the use of its evaluation function. After leaving the book the ten subsequent moves are evaluated. Depending on how these boards are evaluated is a score associated with the played opening moves adjusted, giving the program the ability to detect opening moves that might be more successful the others.

The above method relies on a manually constructed opening book. Such information might be widely available for a popular game like chess. For less popular games it might however not be as widely available and an opening book must be build from scratch. Michael Buro [20] has described a technique used in many strong Othello (brief description in appendix section 10.3.3) programs. It incrementally builds an opening book by adding every position encountered by the program to a move tree. The method evaluates every position as it goes, and adds not only the move played to the tree, but also the move that has been evaluated to be the best alternative. Each leaf position is either evaluated by the outcome of the game or the internal evaluation function in case it was a suggested alternative. During play a possible move is (if possible) located in the tree and a search is done through the sub-tree to calculate a score for the move.

A logical approach for learning is to avoid repeating mistakes already done. This can for example be done by remembering each position where a mistake was made, such that the playing strategy can be changed when the position is encountered again. Susan L. Epstein [25] has used such a method for a game-playing system known as Hoyle that she claims should be able to learn a wide variety of games. After each game the last position where an alternative move could have been made by the loser is identified. From this position a search is carried out to determine if an acceptable alternative can be identified. If the search succeeds the position is marked as “significant” and the alternative move is stored for further encounters with this position. If the search fails the position is marked as “dangerous” and it should be avoided in subsequent games.

A straight forward solution for learning weights of and evaluation function is to provide the program with example positions for which the exact values of the evaluation function is known. The program then tries to adjust the weights in a way that minimizes the error of the evaluation function on these positions. Typically this is done

by a technique such as back-propagation training for neural networks. The first evaluation function developed by G. Tesauro [16] for the program neurogammon that later became TD-Gammon, was trained this way using thousands of example positions from different sources. Tesauro rated the best and the worst moves in the position. He emphasizes that it is important to having examples of bad moves to avoid the program from rating every move as good. A slightly different approach for training an evaluation function is known as comparison training. This method was also introduced by G. Tesauro. Here the learner is not given exact evaluation values for the possible moves, but just presented with the order in which they are preferred.

In games like chess and checkers one of the most important features of creating a strong computer player is searching. Moriarty and Miikkulainen [10] has instead of training a neural network to become an evaluation function, evolved a neural network with a genetic algorithm to cut off unpromising branches of a search tree in Othello. Experiments show that this selective search can be done successfully as their program maintains the same level of play as it did with a full search.

4.6 Conclusion on related work

This section has by no mean tried to cover everything done in the field of artificial intelligence and games. It should however have given and overview of a range of different techniques as well as given insight into how problems typically are approached in this field, which typically are by some mean of board classification scheme.

The game programs presented in this section rely on different approaches. In one category Deep Blue and Chinook can be placed. In a second category Evolutionary checkers can be placed. TD-Gammon can be considered a hybrid that can be placed a little in both categories. Although all agents have been highly successful, does it seems that Deep Blue and Chinook have been the most successful. However if these can be called intelligent is at discussion. They can probably better be classified as expert systems. All their knowledge has been explicitly put into the program by human experts; they do not learn anything on their own. Evolutionary checkers and TD-Gammon on the other hand, has been able to learn the games by playing them, and can better be classified in the category of intelligent game agent.

5. Learn ability in fuzzy control

This chapter will give an overview of some existing work done in the field of creating fuzzy controllers able to learn and hopefully improve in performing a given task. This is to obtain inspiration from this field, which hopefully can become useful when a backgammon playing fuzzy controller referred to as Fuzzeval is to be designed and implemented.

When starting the study of fuzzy logic, fuzzy control and such systems ability to learn, it turned out that much research in this area seems to be about creating some neural network architecture making use of fuzzy logic. These kinds of systems are refereed to as neuro fuzzy systems. For many of these systems is the relation to fuzzy controllers in the author's opinion hard to find. As a consequence material that at first looked relevant was later discarded. Left were five methods of making fuzzy controllers, with the ability to learn. Four of these five methods are relying on classical AI methods such as genetic algorithms and neural networks. The final method refereed to as NEFCON is however a different and highly interesting approach for creating fuzzy controllers that can learn.

This chapter holds the following sections:

- Membership optimization using genetic algorithms
- Membership tuning by a neural network
- Fuzzy rules by neural networks
- Rulebase learning using genetic algorithms
- NEFCON
- Conclusion on adaptive fuzzy control

5.1 Membership optimization using genetic algorithms

In [23] genetic algorithms for optimization of membership functions in fuzzy controllers are covered. The interesting topic is how membership functions can be encoded for optimal results as other issues such as population size, crossover rate, mutation rate and fitness function design is problem dependent and hard to generalize.

The membership functions used in all examples are standard symmetrical triangle functions that can be determined by three parameters. Such a function is illustrated in Figure 27.

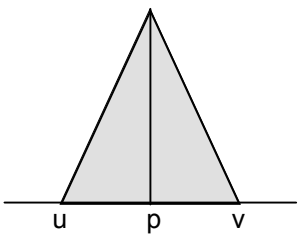

Figure 27: Triangular membership function determined three parameters

The three parameters are the maximum and center point which is called *p*, the left base point called *u* and the right base point called *v*. Two of these three parameters

are enough to determine the shape of each symmetrical triangular membership function.

Base point encoding

An encoding method claimed to be used much in recent research is illustrated in Figure 28.

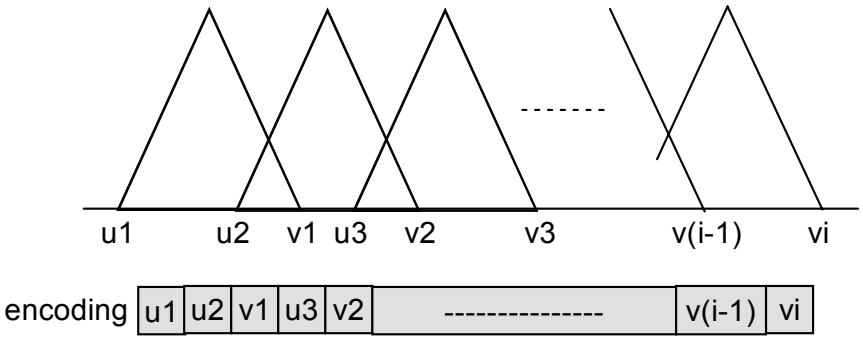

Figure 28: Base point encoding of membership functions

Every base point of every membership function is determined by a specific gene in a chromosome determined by every base point's ideal position in relation to each other. The step of generating new membership functions can then be illustrated as in Figure 29. Two parent chromosomes are randomly selected based on their fitness score. If a crossover is to be performed, is it done at some random point. Finally every gene in the chromosome has a small chance of being mutated. A mutation is performed simply by decrement or increment the value by one.

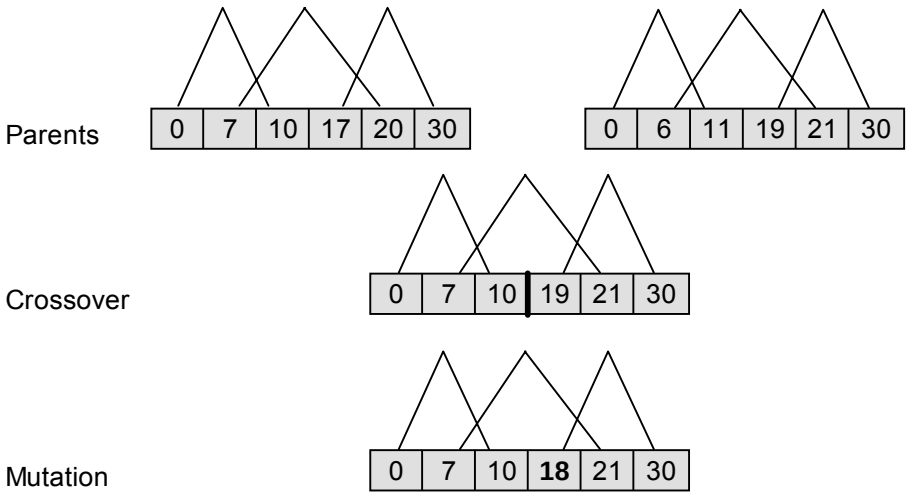

Figure 29: Crossover and mutation of membership functions

This method does however have the disadvantage that after a crossover and a mutation the order of

$$u1 < u2 < v1 < u3 < v2 < \ldots < v(i\text{-}1) < vi$$

may fail to hold. As a result a chromosome having one or more of the bad characteristics as the one illustrated in Figure 30 might be evolved.

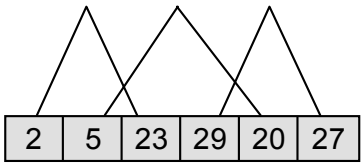

Figure 30: Very bad chromosome representation of membership functions

This is an extremely poor chromosome for several reasons.

- The middle membership function is completely contained “inside” the leftmost membership function
- The rightmost membership is simply illegal as its left base point has a higher value than the right base point. This could however be fixed by switching the two values.
- Still even if the rightmost membership function is considered legal, having its left base point at 27. There exists a gap in the range from 23 to 27 where no membership function is defined.
- If the whole range the membership functions is to cover goes from 0 to 30 as in Figure 29, there exist gaps both in the lower and higher and of the range.

These problems do not make the method useless. The assignment of fitness scores to chromosomes should ensure that chromosomes having one or more of these characteristics only have a minimal chance of evolving further. The overall performance of the genetic algorithm will however suffer due to the continued creation of these useless membership functions penalize the total fitness of each generated population.

One solution to this problem is however to rearrange all the genes after its creation such

$$u1 < u2 < v1 < u3 < v2 < \ldots < v(i-1) < vi$$

again holds. The chromosome illustrated in Figure 30 will then in its rearranged form look like the one illustrated in Figure 31.

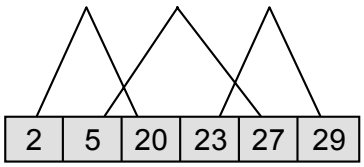

Figure 31: Rearranged chromosome

Although this method solves some of the above mentioned problems, does it suffer from the bad side effect of causing not just small, but severe changes to evolved chromosomes.

Relative increment encoding

Another solution able to solve some of the problems mentioned in the previous method, is to use relative increment encoding as illustrated in Figure 32.

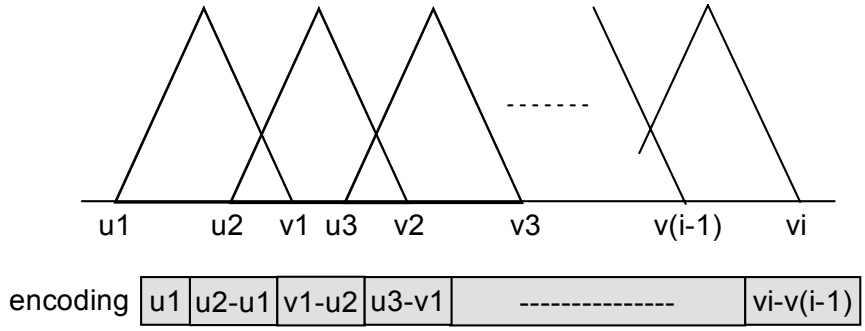

Figure 32: Relative increment encoding of membership functions

Every base point's desired position in relation to other base points, governs the order of the related gene in the chromosome representation. Each chromosome gene does however no more hold an absolute value of a base point position. Instead every chromosome gene holds the position relative to the previous base point. An example can be seen in Figure 33.

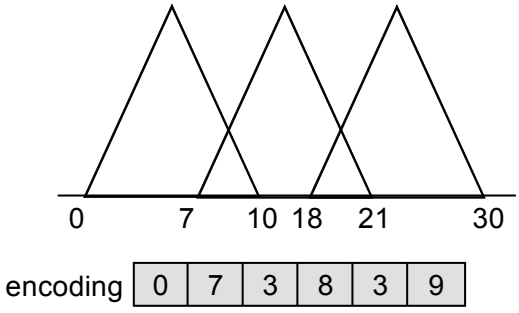

Figure 33: Example of relative increment encoding

Although this method minimizes some of the problems related to the previously mentioned base point encoding without rearranging, does it introduce a new unfortunate feature. That is that the modification of a gene value will have impact on all membership functions determined by subsequent gene values. The change of a value will in other words no longer just modify one of the membership functions, but in worst case all of them.

Center point and width encoding

The relative increment encoding as well as the rearranged base point encoding, does still suffer from a problem that depending on the problem, might not be acceptable. That is that a point on the x axis can never be defined by more than two membership functions which are the case in Figure 34.

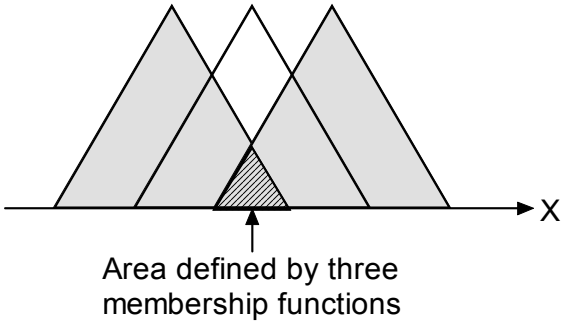

Figure 34: Three overlapping membership functions

An encoding style solving this problem is center point and width encoding illustrated in Figure 35.

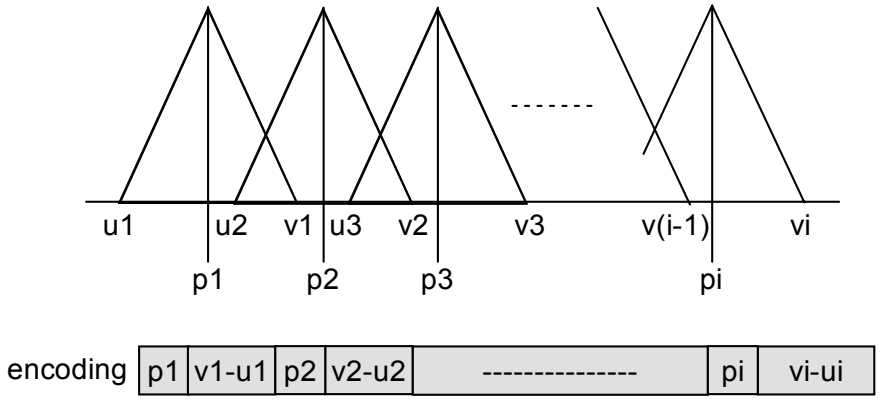

Figure 35: Center point and width encoding of membership functions

Every two gene holds the position of a membership functions center point while a subsequent gene holds the width of the membership function. An example is illustrated in Figure 36.

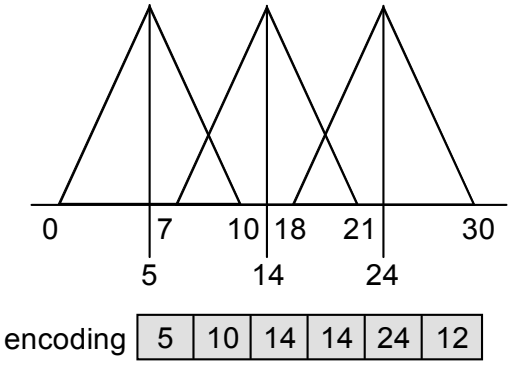

Figure 36: Example of center point and width encoding

Still this method is by no means perfect; gaps where no membership function is defined can exist, and a membership function still has the possibility of be contained “inside” other membership functions. Depending on the problem this could of course be acceptable features. In most cases is it however problems that is to be avoided.

As it should be evident, then the encoding of membership functions to chromosomes is by no mean a completely trivial task, if the best possible performance is to be achieved. The three covered methods do however show that genetic algorithm is a possibility when the object is to create self tuning fuzzy controller.

5.2 Membership tuning by a neural network

Membership function tuning in fuzzy controllers by neural networks is explained in [11]. When designing a fuzzy controller a human control expert is responsible for defining the linguistic variables and terms in the fuzzy controller, as well as the rulebase. Additionally is it necessary to define membership functions for the linguistic terms. The design of correct membership functions is however quite arbitrary.

For example consider the linguistic term approximately zero. Of course the membership function should reach its maximum at zero. But neither the range of the function or its shape, which could be triangular or Gaussian, can be determined by approximately zero. Typically a control expert will have some idea about the function, but he would not be able to determine exactly what is better when considering small changes in the range initially specified. As a consequence the tuning of membership functions has become an important issue in fuzzy control.

An approach is to assume a certain shape for the membership functions. This shape then depends on different parameters that can be learned by a neural network. It can for example be assumed that the membership functions are symmetrical triangular functions depending on two parameters determining the functions width and maximum.

The approach then requires a set of training data in the form of correct fuzzy controller input, as well as its associated correct fuzzy controller output. The entire set of input data is then feed to both the fuzzy controller as well as the neural network as input. Then if the output from the fuzzy controller is not within an acceptable range of what is desired, some adjustment is made to the neural network, such the membership functions will be adjusted in a way that should move the actual output from the fuzzy controller closer to the desired output.

In the typical case the neural network will initially be configured or trained to give neutral output to all input. The meaning of neutral output is that the membership function will not differ from the initial definition of the membership functions. For example if the membership functions are adjusted by adding an output value from the network, then an output of 0 is a neutral. Then to begin with a control expert defines all membership functions by intuition. If the output from the controller for a certain input is not satisfying or can be optimized, the neural network is trained to change its output for this specific input in a way that will adjust all membership functions in a way that will optimize the controllers output.

The principle of this type of fuzzy controller is illustrated in Figure 37. In this example a fuzzy controller is responsible for deciding a break pressure depending on some speed and distance.

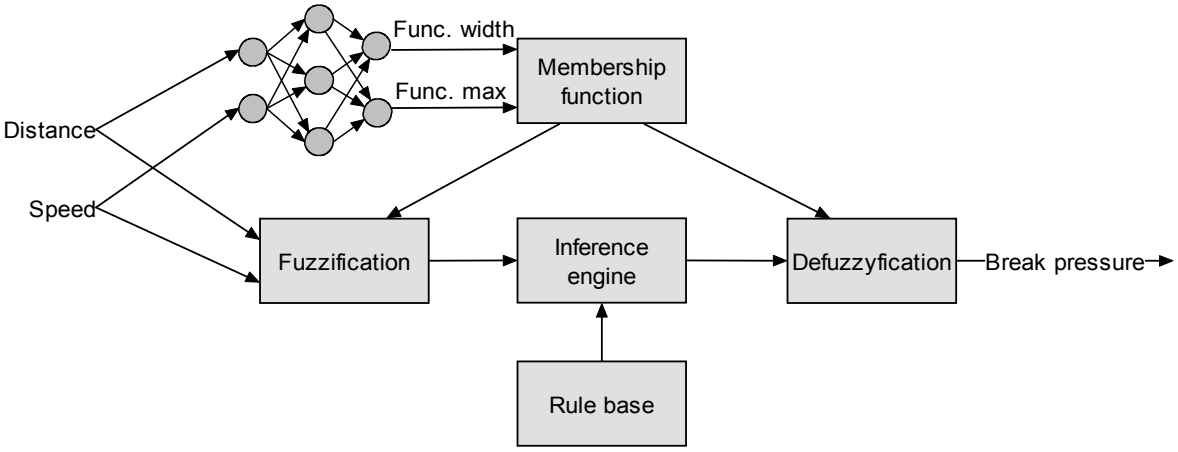

Figure 37: Fuzzy controller with membership functions adjusted by a neural network

What should be understood is that in this method will the membership functions very depending on the input to the controller. For example with an input where distance is 100 and speed is 10 the output from the neural network can be different than if the input to distance is 50 and speed is 25, and as a consequence might the membership functions have slightly different maximum points, as well as width.

The resulting effect is that the membership function to a linguistic term like for example *long* will differ depending on the input, and this will be the case for all membership functions. The entire collection of membership functions is in other words dynamically changing depending on the input. However during training the neural network has tuned the parameters of all the membership functions in such a way that correct output is obtain from the fuzzy controller, given some input.

5.3 Fuzzy rules by neural networks

In the method explained in section 5.2, the rulebase was still made up of basic if-then statements and the learning ability was located in the fuzzification part of the controller. Another approach is explained in [43]. Here the method is to replace the fuzzy controller's if-then rules with neural networks. The rules are in other words no longer specified by if-then statements, but by neural networks where the conditional part has become the input to a neural network, and the consequence part the output from a neural network. The replacement of rules with neural networks can be viewed as a way of importance weight rules.

Consider a block of fuzzy rule of the form:

If speed is X and distance is Y then break pressure is Z

In the rule, *X*, *Y* and *Z* are to be replaced with actual linguistic terms. X could for example be replaced by linguistic terms such as *slow* and *fast*, and *Y* by linguistic terms such as *short* and *long*. Each rule of this type can then be understood as a training pattern for a neural network, where the conditional part of the rule is the input, and the consequence part of the rule is the desired output of the neural network. An illustration of a simple network able to replace the above type of rule can be seen in

Figure 38. (Note that input neurons do not change the input signals, so their output is the same as their input.)

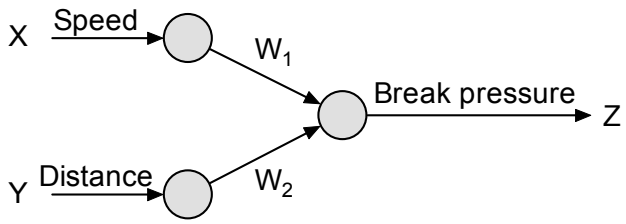

Figure 38: Neural network able to replace basic fuzzy rule

The networks the rules are replaced with are fuzzy neural networks. Fuzzy neural networks use fuzzy operations such as t-norm and t-conorms to combine incoming data, thereby creating a neural network architecture based on fuzzy operations. The combination of the incoming data to the neuron in Figure 38 can for example be done as Z = Min(Max(X, W_1), Max(Y, W_2)) for a fuzzy *and* operation, and as Z = Max(Min(X, W_1), Min(Y, W_2)) for a fuzzy *or* operation.

5.3.1 Example

Let us consider an example where we start out with having a regular fuzzy controller with a regular rulebase without any form of neural networks involved. The rulebase could have the following rules where we for simplicity use actual values as output instead of linguistic terms.

1. If speed is slow and distance is long then break pressure is 0
2. If speed is fast and distance is long then break pressure is 10
3. If speed is slow and distance is short then break pressure is 25
4. If speed is fast and distance is short then break pressure is 75

Let us in this example skip the fuzzification of some actual input and just say that the membership degree for each linguistic term has already been calculated to the values listed in Table 19.

Linguistic term	Membership
Slow	0.7
Fast	0.3
Long	0.6
Short	0.4

Table 19: Linguistic terms and membership degrees

Then using the fuzzy *max* operator as *and* operator, the fulfillment of each rule can be calculated to the values illustrated in Table 20.

Rule	Fulfillment
If speed is slow and distance is long then break pressure is 0	0.6
If speed is fast and distance is long then break pressure is 10	0.3
If speed is slow and distance is short then break pressure is 25	0.4
If speed is fast and distance is short then break pressure is 75	0.3

Table 20: Fulfillment of each rulebase rule

Using the center of gravity formula, the output from the controller can be calculated the following way:

$$\frac{(0 \cdot 0.6) + (10 \cdot 0.3) + (25 \cdot 0.4) + (75 \cdot 0.3)}{0.6 + 0.3 + 0.4 + 0.3} = 22.1875$$

Replacing rulebase with neural networks

Let now the rulebase be replaced by a collection of neural networks. For this example we keep it simple and simply use networks consisting of one neuron with two inputs. Each rule is replaced by its own network which as input is given the membership values illustrated in Table 19. The output from each network can then be considered the fulfillment of each rule. This is illustrated in Figure 39. It should however be noted that this example will not produce the same output results as the standard fuzzy controller exemplified before. For the specified input will it however give a result close by. Identical behavior could however have been obtained by setting all weights in all networks to zero.

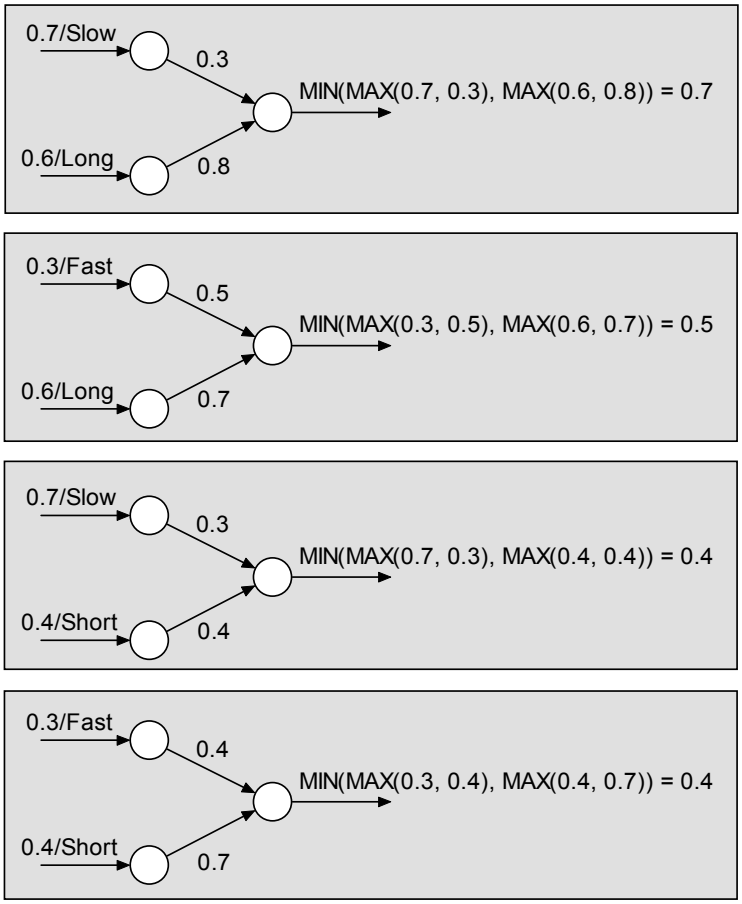

Figure 39: Neural network rulebase inference

Again the center of gravity can be used to calculate the final output from the controller for the exemplified input:

$$\frac{(0 \cdot 0.7) + (10 \cdot 0.5) + (25 \cdot 0.4) + (75 \cdot 0.4)}{0.7 + 0.5 + 0.4 + 0.4} = 22.5$$

Due to the extreme simplicity of all the networks in the above example, any serious tuning of the fuzzy controller will of course be near impossible. To overcome this, the networks can simply be extended with hidden layers containing a desired amount of neurons, giving each network the flexibility required, such a control expert will be able to tune each rule and thereby the controller.

5.4 Rulebase learning using genetic algorithms

Instead of optimize and tune already existing knowledge contained in a controller, in form of membership functions and rulebase rules, in [14] a method able to learn fuzzy control rules is proposed, using genetic algorithms. In this proposal are traditional genetic algorithms however not used; they instead use what they reefer to as *messy genetic algorithms*. Also the structure of the knowledge base is slightly different, as they use what is called a hierarchical prioritized structure to represent the rules. This hierarchical prioritized structure is not a requirement for the proposed method to work.

It is however an interesting solution to a common problem in fuzzy control, and will therefore be briefly explained.

5.4.1 Messy genetic algorithms

In classical genetic algorithms all chromosomes have a defined length and the meaning of a gene in the chromosome is completely defined by its position. This is to ensure that each “value” that is encoded into the chromosome occurs exactly once and always is located at the exact same position.

In messy genetic algorithms each gene in a chromosome is made up of two values. The first value determines the meaning of a gene and the other value determines its actual value. Using messy genetic algorithms, a binary string like [1001] can then be encoded as [(1, 1) (2, 0) (3, 0) (4, 1)]. As each gene holds a value indicating its meaning the order of the genes becomes irrelevant, meaning that the string [1001] just as well can be encoded [(3, 0) (1, 1) (4, 1) (2, 0)].

Messy chromosomes do not need to hold every gene type exactly once, as a result both [(2, 0) (1, 0) (4, 1)] and [(4, 0) (3, 0) (4, 1) (1, 1) (2, 1)] are valid, even though gene 3 is missing in the first example and gene 4 is represented twice in the second.

Also the crossover operation is slightly changed and is now refereed to as splice and cut. The cut operator cuts the two selected chromosomes at, for both chromosomes, random point. The splice operator is responsible for assemble the chromosomes again. This is illustrated in Figure 40.

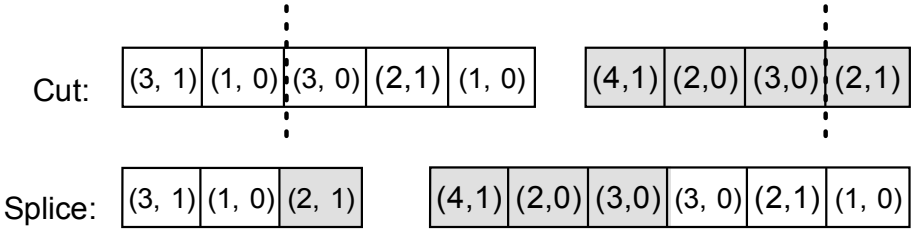

Figure 40: Splice and cut in messy genetic algorithms

The difference with splice and cut compared to basic crossover is that the position of the cuts can be chosen independently, where in classical genetic algorithms the crossover point must be identical.

5.4.2 Hierarchical fuzzy controller

Consider a rulebase made up of just two rules. One of the rules depends on a single variable covering the whole input space while the other rule is a specialized rule only to be triggered in special situations. An example can be seen below.

1. If pressure is high then valve is open
2. If pressure is high and undefined_pressure_loss is high then valve is closed

The problem is that if the slightest amount of pressure exists the valve will fail to close completely, even though *undefined_pressure_loss* might be completely fulfilled. This

is because the more general rule will contribute to the final output as well. A human control expert would be able to ignore the general rule and only consider the special rule in such a case. A regular fuzzy controller will however not be able to do this.

A solution to overcome this problem is to use a rulebase with a hierarchical prioritized structure. The rulebase is organized in several layers with decreasing priority. Rules with highly specific conditions are attached to the levels of highest priority, while rules with more general conditions, are contained in the lower levels. The level, to which a rule is attached, depends on the number of clauses in the conditional part. In the lowest level all rules only have one variable in the conditional part. Rules made up of two variables are contained in the second lowest layer and so on. This is illustrated in Figure 41.

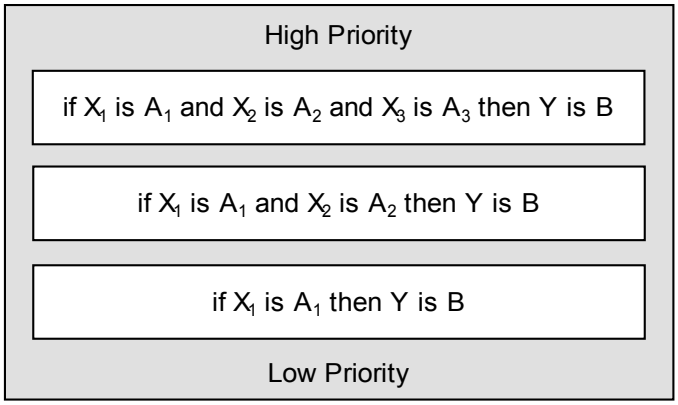

Figure 41: Hierarchical fuzzy controller rulebase

Then as input is to be processed by the rulebase, is it first run through the rules contained in the layer of highest priority. If at least one rule is fulfilled to a degree above zero, this level is used for inference while all lower levels are ignored. In the case this level does not have any rules that are fulfilled above zero, is the next highest level to be used instead. The controller will then continually try lower and lower levels of rules, until hitting a level with at least one rule that can be fulfilled to some degree.

5.4.3 Fuzzy rulebase coding

Before the capabilities of messy genetic algorithms can be used to encode information about a rulebase a designer must in advance define all linguistic input variables, as well as each variables associated linguistic terms. The first of the two values contained in a messy gene then determines the linguistic variable, while the second value determines the associated linguistic term. For example the gene (3, 1) could mean the third variable, maybe defined as height, and the first associated term, maybe defined as low.

Consider a controller having the three input variables distance, angle and height and one output variable speed, and where each variable has three associated linguistic terms. Each of the variables, as well as terms, then has an associated number as illustrated in Table 21.

	1	2	3	4
	Distance	**Angle**	**Height**	**Speed**
1	Short	Zero	Low	Slow
2	Intermediate	Medium	Middle	Average
3	Long	Large	High	Fast

Table 21: Four linguistic variables, each with three associated linguistic terms

The messy string [(3, 2) (1, 3) (4, 2) (2, 1)] would then correspond to the following rule:

If Distance is Long and Angle is Zero and Height is Middle then Speed is Average

At the lowest level a value pair such as (3, 2) can be viewed as a gene while a messy string encoding a rule such as [(3, 2) (1, 3) (4, 2) (2, 1)] can be viewed as a chromosome. A chromosome should however not just represent one rule but a whole rule base. The messy strings encoding rules are therefore concatenated in one larger chromosome representing the whole rulebase. This is illustrated in Figure 42.

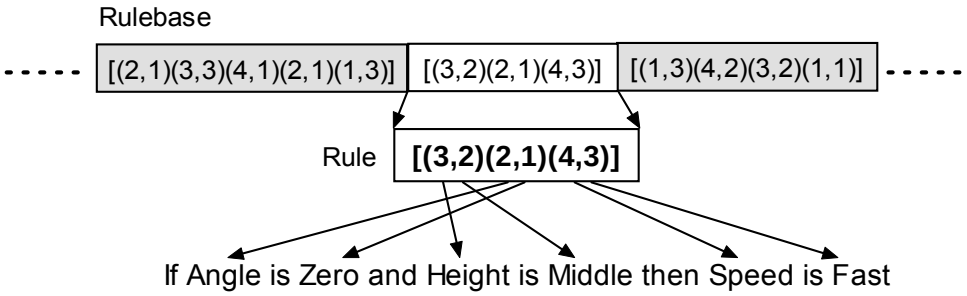

Figure 42: Rulebase encoding in messy genetic algorithms

During decoding of a rulebase each rule is then, depending on the number of variables in the conditional part, placed in the corresponding levels in the hierarchical fuzzy controller.

There exist cases where illegal rules can be created. In cases where the same variable occurs more than once in a messy string, only the left most is used in the final rule. For example in the messy string [(3, 0) (1, 1) (4, 1) (3, 3)], (3, 3) gets suppressed by (3, 0). Another problem occurs in situations where a messy string with no variable representing a conclusion is created. The simple solution would simply be to skip this rule. A repair mechanism which inserts a random chosen conclusion variable is however proposed as better.

Then depending on how well an evolved rulebase is at solving the problem at hand, the higher fitness score is it assigned. Every evolved rulebase in a whole population therefore needs to be tested in a fuzzy controller before evolution of the next generation can be started.

5.5 NEFCON

NEFCON [12] and [13] is short for NEuro Fuzzy CONtrol and is a model developed by Detlef Nauck and Rudolf Kruse. The name might indicate that NEFCON is yet another attempt of combining neural networks and fuzzy controllers. This is however not the case as NEFCON makes no use of neural networks in any way. Instead the NEFCON architecture is simply a fuzzy control architecture inspired by neural networks.

5.5.1 Basic structure

NEFCON can be viewed as a model containing three layers, the input layer, hidden layer and output layer. The input layer consists of some input nodes where the actual input to the controller is given, just as it is the case with neural networks. The output layer can also be compared to the output layer of a neural network, as this layer will output the controllers control signal to some input. The hidden layer is however quite different as this layer represents the rulebase of the fuzzy controller.

To better understand this structure consider a fuzzy controller having two input variables speed and distance, an output deciding some break pressure and a rulebase consisting of the following two rules:

1. If Speed is Slow and Distance is Long then Break pressure is Little
2. If Speed is Fast and Distance is Short then Break pressure is Much

Then such a fuzzy controller can be illustrated as in Figure 43.

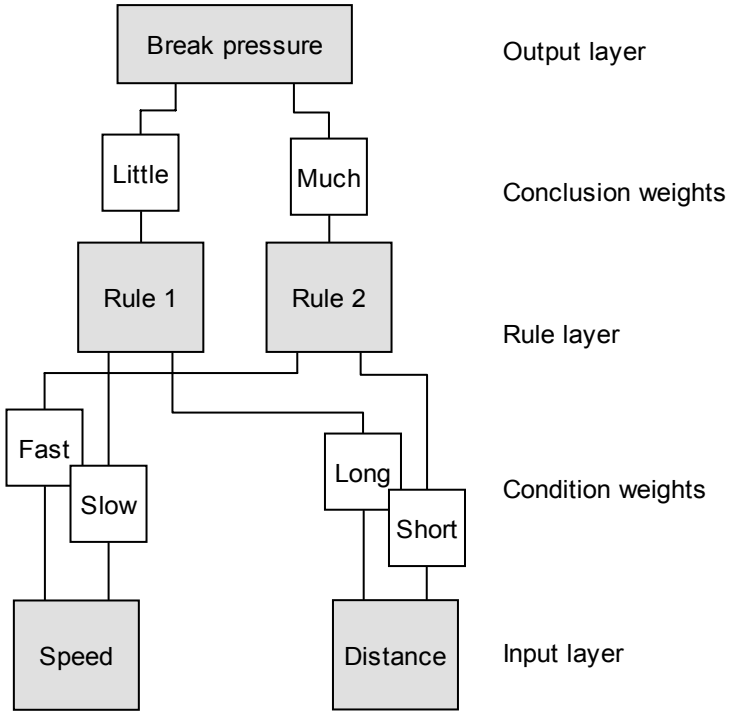

Figure 43: Concrete example of a NEFCON structure

While the larger gray squares can be considered nodes belonging to their respective layer, the smaller white squares can be considered the adjustable weights within the

NEFCON fuzzy controller. In reality these weights are the membership functions of the fuzzy controller.

It should be noted that although the conclusion weights in the above example only have one input and one output, do these weights have multiple inputs and outputs, in cases where there exist multiple rules with the same conclusion as output. If for example there exist three rules having the same term as output, will there be three connections into as well as out from a weight representing this term.

The flow of the controller is not that different than in an ordinary controller. First the input nodes are given the crisp and actual values from the environment. These values are then simply sent through all connections to the rule layer. When these values pass the conditional weights, are the crisp input values fuzzified to membership degrees. These membership degrees are then received by the rule nodes in the rule layer which simply is responsible for calculating the fulfillment of each rule by aggregate all incoming values using the fuzzy *min* operator, or some other t-norm operator. The value indicating the fulfillment of each rule is then sent on to the output layer. When a value passes a conclusion weight, is it converted to a value pair holding both the fulfillment of the rule and the actual value of the output term. Finally are all these value pairs combined to an actual control value in the output layer.

The NEFCON model does however have some requirements to the membership functions it uses in its weights. They must be of a type refereed to as Tsukamoto's monotonic membership functions. Such a function is characterized by two points *a* and *b* where *μ(a)* = 0 and *μ(b)* = 1, and is defined as:

$$\mu(x) = \begin{cases} \dfrac{-x+a}{a-b} & if\ (x \in [a,b] \wedge a \leq b) \vee (x \in [b,a] \wedge a > b) \\ 0 & Otherwise \end{cases}$$

A membership function must in other words be a single straight line as illustrated in Figure 44.

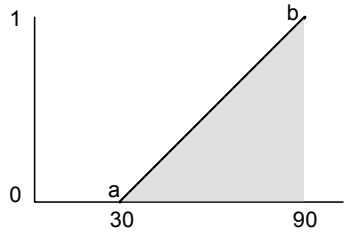

Figure 44: Monotonic membership function

Then using this membership function, the fuzzy membership for the value 52 can be calculated as:

$$\frac{-52+30}{30-90} = 0.367$$

Also the defuzzification is carried out a little different in NEFCON than in typical fuzzy controllers. In a typical fuzzy controller the output terms of the rules, typically have

some associated actual value. Then using these values, as well as the calculated fulfillment of each rule, is the center of gravity used to obtain the actual output from the controller.

In NEFCON each conclusion term has an associated membership function from which an actual value can be obtained using the following formula:

$$x = -y(a - b) + a$$

To exemplify this we might consider some linguistic conclusion term having the associated membership function previously illustrated in Figure 44. Then if this linguistic term has been calculated by the rulebase to have a fulfillment degree of 0.58, the actual and defuzzified value can be calculated as:

$$-0.58(30 - 90) + 30 = 64.8$$

Finally the actual output from the NEFCON fuzzy controller is calculated by the formula:

$$\frac{\sum_{i=1}^{n} r_i \cdot v(r_i)}{\sum_{i=n}^{n} r_i}$$

Where n is the number of rules, r_i the fulfillment of rule i and $v(r_i)$ is the defuzzified value of rule r_i's output. In reality this formula is identical to the center of gravity formula normally used.

5.5.2 Learning

A goal with NEFCON is to make an architecture where the membership functions can be tuned by a learning algorithm. This is done by defining a fuzzy error that is propagated back through the architecture. Based on this error, and the fulfillment of each rule in the rulebase, is the membership functions adjusted. The adjustment of the membership functions in the controller, is simply done by moving the point *a* on the x-axis in Tsukamoto's monotonic membership functions, while keeping the point *b*.

The first step when tuning the fuzzy controller is to obtain some fuzzy error of the output. How to obtain this error seems to depend on the problem that is considered by the controller, as no exact way seems to be explained. One way could then be to use the following formula:

$$E = 1 - \frac{Inf(c_{opt}, c_{out})}{Sup(c_{opt}, c_{out})}$$

Where c_{out} the output from the controller, c_{opt} the optimal or desired output and *Inf* and *Sup* are short for inferior and supreme meaning we are using the smallest and largest

of the values. If this corresponds to the intended way of calculating the error is unfortunately not completely clear.

After the fuzzy error has been calculated is each rules contribution to the error calculated by:

$$eR_i = \begin{cases} -r_i \cdot E & if\ c_i < c_{opt} \\ r_i \cdot E & if\ c_i > c_{opt} \end{cases}$$

Where r_i is the fulfillment of the rule and c_i is the defuzzified crisp output from this rule after it has been through the conclusion weights.

After the calculation of each rules contribution to the overall error, the weights within the controller must be adjusted. Remember that the weights within the controller is actually monotonic membership functions decided by the two points *a* and *b* where an adjustment simply is done by moving the point *a* on the x-axis.

For the conclusion membership functions, the adjustment is calculated as:

$$a_k^{new} = \begin{cases} a_k - \sigma \cdot eR_i \cdot |\, a_k - b_k \,| & if\,(a_k < b_k) \\ a_k + \sigma \cdot eR_i \cdot |\, a_k - b_k \,| & otherwise \end{cases}$$

Where σ is a learning rate factor and *k* is the weight or membership function under consideration. If *k* is shared by multiple connections from the rule layer to the output layer, this adjustment is done once for every connection.

For the conditional membership functions the adjustment is calculated as:

$$a_{jk}^{new} = \begin{cases} a_{jk} + \sigma \cdot eR_i \cdot |\, a_{jk} - b_{jk} \,| & if\,(a_{jk} < b_{jk}) \\ a_{jk} - \sigma \cdot eR_i \cdot |\, a_{jk} - b_{jk} \,| & otherwise \end{cases}$$

Where *k* is a weight or membership function and *j* is the input variable it is associated to. Again if *k* is shared by multiple connections from the input layer to the rule layer, this adjustment is done once for every connection.

5.5.3 Example

To better illustrate exactly how the NEFCON architecture works we will consider an example based on the controller illustrated in Figure 43. The weights or membership functions, can then for this example initially be defined as illustrated in Figure 45.

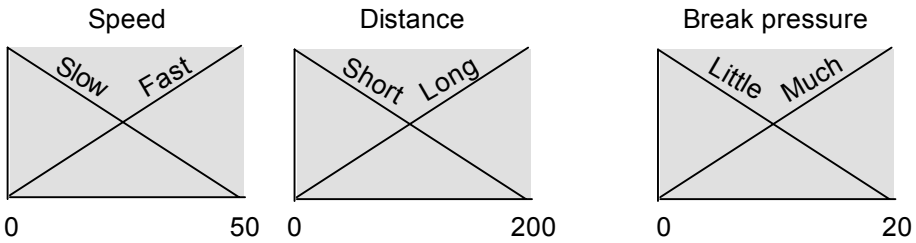

Figure 45: Initial monotonic membership functions used

The crisp input to the controller in this example is illustrated in Table 22.

Input	
Speed	30
Distance	96

Table 22: NEFCON controller input

The fist step is to fuzzify the input in the conditional weights. When this is done each node in the rule layer is responsible for aggregating its input and calculate the fulfillment the rule it represents. In this example both of these operations will be done in one step below.

First the calculation of the fulfillment of rule 1:

$$Min\left(\frac{-30+50}{50-0}=0.4,\ \frac{-96+0}{0-200}=0.48\right)=0.4$$

Then the calculation of the fulfillment of rule 2.

$$Min\left(\frac{-30+0}{0-50}=0.6,\ \frac{-96+200}{200-0}=0.52\right)=0.52$$

Running these values through the conclusion weights and each rules output gets defuzzified to:

Rule 1: -0.4(20 - 0) + 20 = 12
Rule 2: -0.52(0 - 20) + 0 = 10.4

Finally the output layer receives the two value pairs (0.4, 12) and (0.52, 10.4) from which it is able to calculate the output from the controller as:

$$\frac{(0.4\cdot 12)+(0.52\cdot 10.4)}{0.4+0.52}=11.096$$

Adjustment

If it turns out that this output not exactly matches the desired output, some adjustment is necessary. It could for example be that the controlled object is breaking just a

fraction too slow and the calculated break pressure should be slightly increased to the value illustrated in Table 23.

Desired output
11.65

Table 23: Desired controller output

First step is to calculate the fuzzy error. We do this as:

$$1 - \frac{11.096}{11.65} = 0.048$$

Then each of the rules error rate must be calculated.

Rule 1 error:	$0.4 \cdot 0.048 =$	0.019
Rule 2 error:	$-0.52 \cdot 0.048 =$	-0.025

Now the adjustment of all the membership functions, also refereed to as weights, can be calculated. This are done by calculating the new point for *a* in the monotonic membership functions. First this is done for the conclusion membership functions. For this example we will use a learning rate of 0.2.

Break pressure		
${}_{new}a_{little}$	$20 + 0.2 \cdot 0.019 \cdot 20 =$	20.076
${}_{new}a_{much}$	$0 - 0.2 \cdot (-0.025) \cdot 20 =$	0.1

Next we calculate the new point for *a* in the conditional membership functions.

Speed		
${}_{new}a_{slow}$	$50 - 0.2 \cdot 0.019 \cdot 50 =$	49.81
${}_{new}a_{fast}$	$0 + 0.2 \cdot (-0.025) \cdot 50 =$	-0.25
Distance		
${}_{new}a_{short}$	$200 - 0.2 \cdot (-0.025) \cdot 200 =$	201
${}_{new}a_{long}$	$0 + 0.2 \cdot 0.019 \cdot 200 =$	0.76

In Figure 46 it has been tried illustrated what exactly it is the learning algorithm has done to the membership functions. The arrows indicates how the point refereed to as *a* has been moved on the x-axis.

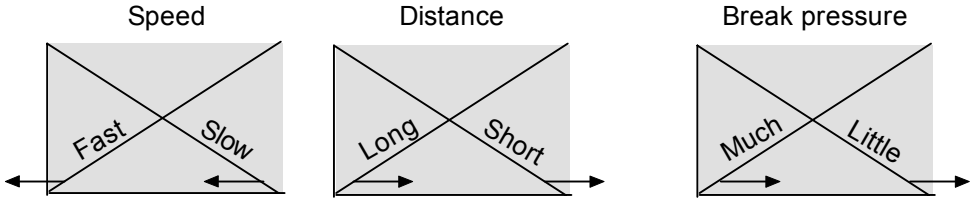

Figure 46: Arrows illustrating the adjustment of membership functions

As it can be seen then a value fuzzified with the *slow* or *long* membership function will now be assigned a slightly smaller membership degree, while values fuzzified with the *fast* or *short* membership function, now will be assigned a slightly higher membership degree.

Now remember that the defuzzified output of rule 1 was higher then the desired output, and the defuzzified output from rule 2, was smaller then the desired output. With these adjustments of the membership functions the fulfillment of rule 1 will be reduced, while the fulfillment of rule 2 will be increased.

The overall output from the controller will however still increase, as the adjustment of the two conclusion membership functions for *much* and *little*, will ensure that a fuzzy value now will be defuzzified to a slightly higher value on the output weights, making the overall output from the controller slightly higher.

Recalculation of output

With these adjustments the output from the controller can be recalculated.

Fulfillment of rule 1:

$$Min\left(\frac{-30+49.81}{49.81-0}=0.398,\ \frac{-96+0.76}{0.76-200}=0.478\right)=0.398$$

Fulfillment of rule 2:

$$Min\left(\frac{-30+(-0.25)}{-0.25-50}=0.602,\ \frac{-96+201}{201-0}=0.522\right)=0.522$$

Defuzzification in conclusion weights:

Rule 1: -0.398(20.076 - 0) + 20.076 = 12.086
Rule 2: -0.522(0.1 - 20) + 0.1 = 10.488

Calculation of output:

$$\frac{(0.398\cdot 12.086)+(0.522\cdot 10.488)}{0.398+0.522}=11.179$$

The adjustment has been responsible for adjustment the output to 11.179, where the output before the adjustment were 11.096. The output has in other words been moved closer to the desired output of 11.65. A larger adjustment could have been achieved by using a higher learning rate then the value 0.2 used in this example. Setting the learning rate to high is however not always advisable. This simply might do that the learning algorithm makes the controller overshoots the desired target, making it impossible to ever tune the controller to a desirable result within an exactable range of error. Deciding on a learning rate value is in other words a tradeoff between precision and learning speed.

5.5.4 NEFCON II

As it is apparent then NEFCON architecture is not able to learn completely from scratch as the rulebase has to be defined beforehand by a human control expert. To solve this problem an extension has been made, such the NEFCON architecture is able to learn its own rulebase. This extended architecture is refereed to as NEFCON II. Still with this extension a human control expert is of course responsible for defining the linguistic input and output variables as well as the amount of membership functions (linguistic terms) associated to each variable.

The method assumes that the rulebase necessary to control the object is not known. A network is then created holding every possible combination of the linguistic input and output terms, which in reality is every possible rule. During the learning process rules are then continually removed until a good rulebase has been created. The learning process in this model can be divided into three phases.

Phase 1: In this phase all rules that is producing counterproductive results is removed. This simply means that rules producing negative output value where a positive value is required are deleted right away and vice versa. Additionally every rule is marked with an associated counter that is decreased every time a rule is not fulfilled to a degree above zero. In the other case where a rule is fulfilled to some degree above zero, is this counter reset back to its maximum value. In the case a counter hits zero, its associated rule is removed.

Phase 2: In this phase every rule still has its associated counter value and still a rule is removed in the case its counter hits zero. Further more each rule now keeps track of its overall rate of error. Then if there at the end of this phase still are rules with identical conditional parts, all these rules except for the one with the smallest overall error are deleted.

Phase 3: In this final phase the performance of the architecture is enhanced by tuning the membership functions as already described.

5.6 Conclusion on adaptive fuzzy control

The study of adaptive fuzzy control did turn out to be somewhat of a mixed experience. Honestly one had hoped to find more innovative and useful ideas in the field that has been the case, with the NEFCON architecture as the great and positive exception.

The problem is that the use of for example genetic algorithms to evolve membership functions, as well as a rulebase, in the author's opinion hardly can be considered innovative. This idea is in all humbleness something most people in the field probably could come up with. This does not in any way mean that nothing has been learned by this study. The concept of messy genetic algorithms, as well as how they apply such algorithm to the problem of evolving a rulebase, is probably not an idea one self could come up with in the nearest of future. The study of genetic algorithms for the

evolvement of membership functions does also shed some light upon some encoding problems, one must consider before deciding on an exact encoding scheme.

The main problem with genetic algorithms is however not whether the idea is innovative or not. The main problem is that it is simply isn't a very good solution considering the aim of this thesis. The problem is, that from the moment a strong solution has been found, and is applied to a fuzzy controlled agent, nothing more is learned. The agent will in other words not be adaptive in any way. It will simply continue to have the same playing style over and over again no matter how many games it plays (and loses) against a human opponent.

In [14] a solution to the problem of handling exceptions in a rulebase is however given. This solution uses what is refereed to as a hierarchical structured rulebase. The use of such structure might, depending on the time available and identified solution models, be considered when designing Fuzzeval (the fuzzy control based backgammon agent).

Instead of relying on genetic algorithms, two other covered methods are beginning to cross into the field of neuro fuzzy systems and explains how neural networks can be used to both adjust membership functions, and to adjust a controller by replacing the rulebase with fuzzy neural networks. Both methods seem useful considering the aim of this thesis. It is however by the author considered an important property that the Fuzzeval is a "clean" fuzzy controller where all knowledge is interpretable by humans. Nonetheless must it be admitted, that the reason for discarding neural networks as a solution honestly is based more on a personal disliking of using neural networks in relation to making a fuzzy controller adaptive, rather then rational arguments.

NEFCON on the other hand, is a quite brilliant architecture in the author's opinion. The architecture ensures that some kind of reinforcement learning style such as temporal difference explained in section 4.4 theoretically can be used to continually adjust the controller while playing a human opponent. Also the architecture can be considered a "clean" fuzzy controller from which all encoded knowledge easily can be extracted. Exactly how much the final solution model for Fuzzeval will resample the NEFCON architecture is at this point of course unknown, but some resembles is not unlikely.

6. The backgammon domain

This chapter will present the domain of backgammon. The chapter will start with a general introduction of backgammon such as its history and its rules. It will then gradually become more technical with sections describing the implementation of a backgammon application, in which Fuzzeval can be implemented and tested.

The following sections are contained in this chapter:

- History of backgammon
- Rules of backgammon
- Skill vs. luck in backgammon
- Why backgammon as domain
- Domain requirements
- Initial design overview
- Implementation and final design
- Application overview
- Conclusion on the backgammon domain

6.1 History of backgammon

This section will try to present a short overview of the history of backgammon. Keeping this section short is however not an easy task as the history of the game is long, complicated, incomplete and quite interesting. The source is [38] which is a reprint of the 1970 edition of, "The backgammon Book" by Oswald Jacoby & John R. Crawford.

Backgammon is a board game making use of dices. Dice games seem to have developed all over the world. First tribal priests have probably rolled bones from animals to predict the future. But it probably did not take long before people were starting to roll dices and then bet on the outcome. Once the dice had been invented, the next step was probably to develop boards on which pieces could be moved around. Such games seem to have developed all over the world in ancient times, and some of them may be early visions of what later developed into backgammon.

The oldest possible ancestor of backgammon discovered, is around five thousand years old. It was discovered during the excavation of a royal Sumar cemetery. Sumar is and old civilization that existed in what is now Iraq. In one of the graves, game boards with layouts bearing some resemblance to the backgammon boards of today, was found. Under one of the boards two set of playing pieces as well as dices were discovered. Each player apparently had seven pieces and six dices.

There are also evidences suggesting that the Egyptian Pharos where playing a game resembling backgammon around 1500 year before Christ. Game layouts as well as wall paintings have been discovered in graves and tombs. The wall painting suggests that both the common folk as well as the aristocrats where playing the game.

Another reasonable guess from where the first version of backgammon originates from could be China or India, as this is where most games are inherited from.

Moreover is there sufficient similarity between backgammon and another ancient Indian game known as Parcheesi, to suggest the later as a possible remote ancestor

The first early version of a game resembling backgammon most likely reached Western Europe through the Mediterranean. It is known that the Greeks where playing this game a thousand years after the Egyptians where playing their version. The Greek mythical hero Palamedes has even been credited for inventing the game as well as the dice among other things. This games invention has however also been credited to the Lydians, a civilization ruling from 900 – 547 BC in a part of what is now Turkey. Also the Romans where known to play this the game.

An earliest version of backgammon as we know it today was however encountered by the crusaders among some Arabs. The game was named Nard-shir and the Arabs have learned the game from the Persians. The game was supposedly named after Ard-shir Babakan of the Sassanid dynasty of the ancient Persian Empire, who was said to have invented it. Various early versions this game, at that time known as Tables, seems to have been most popular in Britain. The game is mentioned in old English literature, where it is claimed, that it dates back to the Crusades.

It is also known that the game remained a favorite game among the upper class in the rest of Europe during the middle Ages, and then spread to the middle society. The earliest known use of the word backgammon is in 1645 where in a “History of Board Games Other Then Chess” is said that backgammon is the modern form of Tables invented in England in the early seventeenth century. The two main differences between Tables and backgammon are according to the book that dice doublets now are played twice, and the introduction of the triple win refereed to as a backgammon victory.

Although the game probably has been less popular in the United States, is it known to have been played since the seventeenth century. Thomas Jefferson played the game often during the three weeks before July 4, 1776 while he was working on the Declaration of Independence.

In Europe and the United States the interest in backgammon rapidly increased in the 1920s when some unknown player in an American club, came up with an innovative idea. He proposed that, at his turn to play, He could insist on doubling the stakes. His opponent would have the right to refuse, in which case the game would end and the player doubling would score the original stake. This same person, or perhaps a second one, then added the redoubling feature, which allows a player who has been doubled, to redouble his opponent in the same manner, whenever it is his turn to roll the dice.

After the invention of doubling, the game increased rapidly in popularity. There was however still one great problem, no universally acceptable rules for the game existed. Depending on where the game was played small variations could exist. In 1931 Wheaton Vaughan, chairman of the club's Card and backgammon Committee, decided to gather a committee undertaking the task of writing the rules. These rules have remained the accepted rules until today.

The game we refer to as backgammon is not the only game played on the backgammon board as we know it. Many variations exist such as Russian and Persian backgammon, the Turkish Moultezim and Gioul, the Greek Plakoto and Eureika and other such as Acey-deucy which originates from the Middle East.

6.2 Rules of backgammon

This section is a slightly edited version of rules written by Tom Keith. The original version can be found at backgammon Galore http://www.bkgm.com/rules.html.

Backgammon is a stochastic perfect information game of skill (and a small amount of luck). The game is considered to be the king of racing games, in comparison to chess which is considered the king of fighting games. It is played on a board consisting of twenty-four narrow triangles called points. The triangles alternate in color and are grouped into four quadrants of six triangles each. The quadrants are referred to as a player's home board and outer board, and the opponent's home board and outer board. The home and outer boards are separated from each other by a ridge down the middle of the board called the bar (see Figure 47).

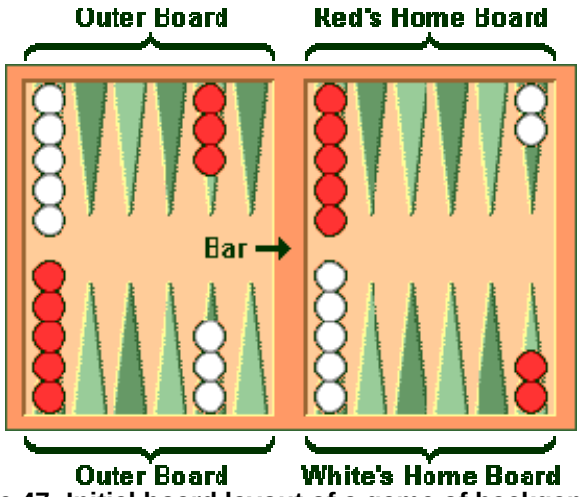

Figure 47: Initial board layout of a game of backgammon

The points are numbered for either player starting in that player's home board. The outermost point is the twenty-four point, which is also the opponent's one point. Each player has fifteen checkers of his own color. The initial arrangement of checkers is as illustrated in Figure 47 if white is to move counter clockwise.

Both players have their own pair of dice and a dice cup used for shaking. A doubling cube, with the numerals 2, 4, 8, 16, 32, and 64 on its faces, is used to keep track of the current stake of the game.

Object of the Game

The object of the game is for a player to move all of his checkers into his own home board and then bear them off. The first player to bear off all of his checkers wins the game.

Movement of the Checkers

To start the game, each player throws a single dice. This determines both the player to go first and the numbers to be played. If equal numbers come up, then both players roll again until they roll different numbers. The player throwing the higher number now moves his checkers according to the numbers showing on both dices. After the first roll, the players throw two dice and alternate turns. The roll of the dice indicates how many points, or pips, the player is to move his checkers. The checkers are always moved forward, to a lower-numbered point as illustrated in Figure 48, and a checker may only be moved to an open point that is not occupied by two or more opposing checkers.

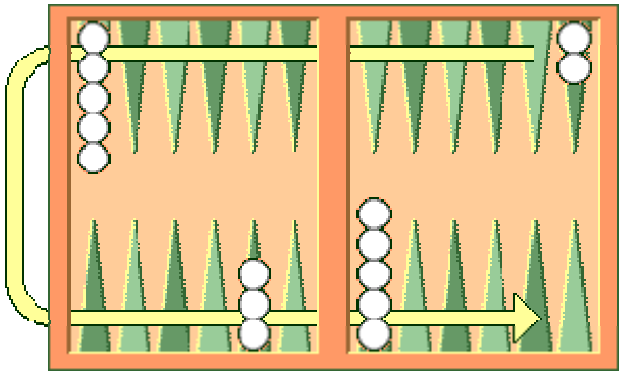

Figure 48: Move direction for white

The numbers on the two dice constitute separate moves. For example, if a player rolls 5 and 3, he may move one checker five spaces to an open point and another checker three spaces to an open point, or he may move the one checker a total of eight spaces to an open point, but only if the intermediate point (either three or five spaces from the starting point) is also open. This is illustrated in Figure 49.

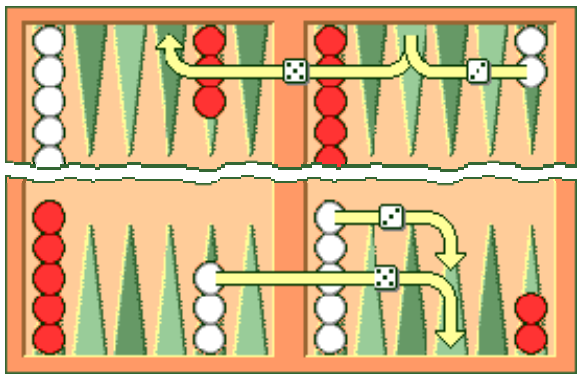

Figure 49: Two ways white can play a roll of 3 and 5

A player who rolls doubles plays the numbers shown on the dice twice. A roll of 6 and 6 means that the player has four sixes to use, and he may move any combination of checkers he feels appropriate to complete this requirement.

A player must use both numbers of a roll if this is legally possible (and all four numbers of a double). When only one number can be played, the player must play

that number. Or if either number can be played but not both, the player must play the larger one. When neither number can be used, the player loses his turn. In the case of doubles, when all four numbers cannot be played, the player must play as many numbers as he can.

Hitting and Entering

A point occupied by a single checker of either color is called a blot. If an opposing checker lands on a blot, the blot is hit and placed on the bar. Any time a player has one or more checkers on the bar, his first obligation is to enter those checker(s) into the opposing home board. A checker is entered by moving it to an open point corresponding to one of the numbers on the rolled dice.

For example, if a player rolls 4 and 6, he may enter a checker onto either the opponent's four point or six points, so long as the prospective point is not occupied by two or more of the opponent's checkers. An example entering can be seen in Figure 50.

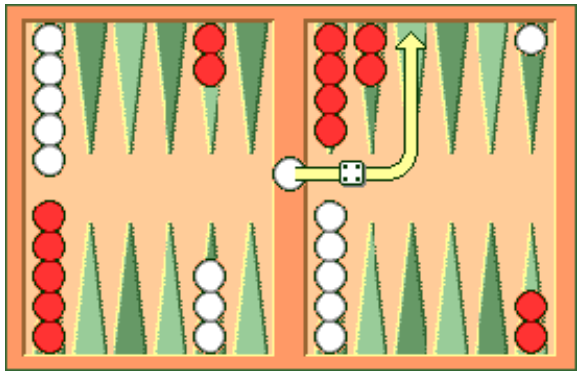

Figure 50: If white have rolled 6 and 4 the first move done must be the one illustrated

If neither of the points is open, the player loses his turn. If a player is able to enter some but not all of his checkers, he must enter as many as he can and then forfeit the remainder of his turn. After the last of a player's checkers has been entered, any unused numbers on the dice must be played as normally.

Bearing Off

Once a player has moved all of his fifteen checkers into his home board, he may begin bearing off. A player bears off a checker by rolling a number that corresponds to the point on which the checker resides, and then removing that checker from the board. Thus, rolling 6 permits the player to remove a checker from the point six.

If there is no checker on the point indicated by the roll, the player must make a legal move using a checker on a higher-numbered point. If there are no checkers on higher-numbered points, the player is permitted (and required) to remove a checker from the highest point on which one of his checkers resides. A player is under no obligation to bear off if he can make an otherwise legal move. An example illustrating a legal bear off can be seen in Figure 51.

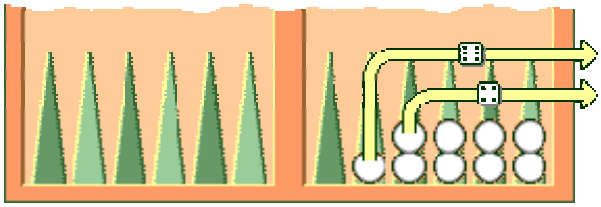

Figure 51: Legal bear off by white if 6 and 4 has been rolled

A player must have all of his active checkers in his home board in order to bear off. If a checker is hit during the bear-off process, the player must bring that checker back to his home board before continuing to bear off. The first player to bear off all fifteen checkers wins the game.

Doubling

Backgammon is played for an agreed stake per point. Each game starts at one point. During the course of the game, a player who feels he has a sufficient advantage may propose doubling the stakes. He may do this only at the start of his own turn and before he has rolled the dice. A player who is offered a double may refuse, in which case he concedes the game and pays one point. Otherwise, he must accept the double and play on for the new higher stakes. A player who accepts a double becomes the owner of the cube and only he may make the next double.

Subsequent doubles in the same game are called redoubles. If a player refuses a redouble, he must pay the number of points that were at stake prior to the redouble. Otherwise, he becomes the new owner of the cube and the game continues at twice the previous stakes. There is no limit to the number of redoubles in a game.

Gammons and backgammons

At the end of the game, if the losing player has borne off at least one checker, he loses only the value showing on the doubling cube (one point, if there have been no doubles). However, if the loser has not borne off any of his checkers, he is gammoned and loses twice the value of the doubling cube. Or, worse, if the loser has not borne off any of his checkers and still has a checker on the bar or in the winner's home board, he is backgammon and loses three times the value of the doubling cube.

6.3 Skill vs. luck in backgammon

In [39] and [44] is it discussed whether backgammon is a game of luck or skill. Backgammon combines strategic and mathematical skill with elements of chance. One of the fascinating elements of the game is that a rather poor player, if having the right amount of luck, occasionally can beat a better player. Therefore multiple matches are necessary to fairly evaluate a player’s skill. But even though the game is a dice game, is it wrong to think that the luck element have that much influence on the game. Everybody that has played the game to some degree, know that the rules hide a sophisticated game of skill. In the long run, one will simply begin to notice that it usually is the best player that is the luckiest.

Of course one will at one time or another experience a game where one is way ahead and still end up loosing because the opponent suddenly rolls three double sixes in a

row. But this is not a fair argument for claiming that backgammon is a game decided mostly by luck. How can it for example be that GNU-backgammon (a very strong backgammon program based on the same principles as G. Tesauro's TD-Gammon described in section 4.4) almost constantly beats up average players (including the author of this paper) on the harder levels?

The element that makes the better player the luckiest one is the fact that this player is better at playing the odds. This aspect of the game is the biggest contrast to a game like chess. If one makes a good move in chess one is rewarded with a good position, where in backgammon you are rewarded with better odds. The better odds mean that one is more likely to obtain the victory at the end of the game. The better odds does however also mean, even though it is less likely, that one may end up being panelized for this good move. This difference is something that makes many chess players very uncomfortable with the game of backgammon.

The way backgammon then eliminates the luck factor is by requiring multiple wins in a tournament. A typical backgammon tournament one will require 7, 9 or 13 points to achieve a victory. This is equal to playing one game in a chess tournament. As a result one may get lucky in the first, or the second, or even the third one as well. But one will probably not end up being lucky forever. By requiring multiple matches to decide the outcome of a game, the luck factor gets evened out, and the better player ends up winning.

But whether backgammon is a game decided by skill or luck has actually been settled by a court. The history started in 1981 where the Portland police in Oregon, arrested a backgammon tournament director. He was accused for promoting gambling as his tournaments offered cash prices and required entry fees. According to the statutes of Oregon, New York, and other states, gambling is defined as risking something of value upon the outcome of a contest of chance. Instead of simply accepting the charges, the director decided to fight the matter in court, hoping the court would rule in his favor as previously done by the Alabama Supreme Court in 1976. For his defense he among others got help from the former World backgammon champion Paul Magriel. The main issue to be decided was the effect of the dice in games. But Magriel told a packed courthouse during his expert testimony that "Even after rolling, you may have as many as 30 or more options" and "The decision where to move your pieces after the dice have been cast - that is the essence of the game. Chance is not a material factor." The judge agreed that the director was not promoting gambling, and that backgammon is a game decided by skill just as chess and bridge, and not by chance.

6.4 Why backgammon as domain

One may wonder why backgammon has been chosen as the domain considering the fact that G. Tesauro already has created a model, able to play the game at the same level as top ranking human players. This model does however have at least one of the characteristics, one exactly is trying to eliminate with a solution model hopefully identified at the end of this thesis. That characteristic is that training is a relatively heavy process. TD-Gammon first stopped improving after it had played around 1.500.000 games against itself. One can easily imagine that even though TD-

Gammon still learns when playing humans, its rate of learning and thereby adaptive behavior will be close to none existing. The TD-Gammon model is in other words not an already existing solution to the problem trying to be solved in this thesis, as the goal in this thesis is to explore ways to develop adaptive behavior by use of fuzzy logic, possibly even to develop a player that learns to exploit specific opponent's weaknesses, instead of relying on an overall strategy considered to be the best in general when playing a range of different players.

The development of TD-Gammon is in fact a major advantages and actually a positive contribution to the final decision of selecting backgammon as domain. One of the problems one would be faced with after the development of computer playing agent is to decide how to evaluate its strength. This exact problem has in fact been solved by G. Tesauro as he has trained a benchmark evaluator known as Pubeval [28]. Pubeval is not TD-Gammon trained to its final strength. Pubeval is an evaluator of intermediate strength (not too strong, not too weak) that has a reasonable style of play and does not create bizarre looking board positions. The source code has been made public available and is relatively simple to use. Despite its simplicity, Pubeval should according to G. Tesauro himself, play at a decent intermediate level. He guesses that it is better than many, if not most commercial programs. Tests indicate it wins 57% against the Sun program gammontool, and 75% against the Unix program btlgammon. Not only does the existence of Pubeval ensure a decent opponent, it is in fact an opponent having a deterministic playing style with no form of adaptive behavior. This is the ideal opponent for testing whether Fuzzeval will be able to play backgammon reasonable well. The only real limitation with Pubeval is that it only supports move evaluation, and no cube doubling evaluation. Backgammon does also have other advantages over many other games such as for example chess. One is that every game is guarantied to produce a winner and a looser. Another is that the dices will ensure variation in the play. One of the main problem developers of machine learning models, learning by self play, is faced with, is ensuring variation in the play. Often the agent will simply get stuck in playing the same drawn game over and over again. This is a problem that one self experienced during the development of a tic-tac-toe agent during the first part of this thesis. Different methods have been proposed to solve this problem, where the most popular is making a certain percentage of the moves be randomly selected. The vary nature of backgammon does however ensure that this problem can be neglected. Yet another advantage is that the dices are ensuring that the problem of creating a capable player not gets reduced to developing a highly effective search algorithm. The branching factor of backgammon is also of such a scale that even a probability based search is impossible beyond three moves.

The most important contribution to the decision of selecting backgammon as domain is however the fact that one self is able to play this game reasonably well. The game is well known by the author and has been played for several years. When playing against Pubeval one seems to be slightly below its level of play. Although one has never been playing in tournament play, it is also the apparent feeling from playing against human opponents that one is certainly not in the worst half of the average skilled player.

6.5 Domain requirements

This section will describe the requirements to the backgammon application at a very general level.

Besides the obvious requirements that the application must work, the most important requirement is that the backgammon application as a whole should provide Fuzzeval with a reliable, flexible and effective domain, in which test and evaluation can be performed. That means that first and foremost must human opponents be able to play the game of backgammon against Fuzzeval on fair terms. That means that the application must provide a user interface that can be easily comprehended by humans. A command line interface with text output “explaining” the layout of the board is not acceptable. The interface must graphically illustrate the board in just the same way as if a human was playing another human on a real backgammon board. The application must further more ensure that only legal play is done by humans as well as agents playing within the application.

Another requirement is that the application as a whole must work as test domain against a reference evaluator. Play against this reference evaluator has to be done fast and effective as a large range of games might be needed to conclude the strength of Fuzzeval, as well as if it is able to learn. This reference evaluator has been selected to be Pubeval. This seems as the only logical choice since both source code is available and that it has become the de facto standard for backgammon program benchmarking.

Even with the ability to evaluate Fuzzeval against humans and Pubeval, is it also a requirement that Fuzzeval can be tested against other backgammon programs. This can be achieved by letting a human work as interface between Fuzzeval running in the backgammon application to be developed, and another backgammon program. The only requirement needed to fulfill this, is the ability to manually set the dice values before each roll, thereby making sure a human is able to copy rolls, as well as moves from another program. Using a human as interface in this way even ensures that Fuzzeval can play against other human opponents on internet servers, without them knowing they are playing a computer program.

To generalize this a little then the requirement is simply to develop a fully working backgammon application offering enough flexibility to test Fuzzeval primarily against Pubeval, but also against other backgammon programs as well as humans.

6.6 Initial design overview

This section will present the initial planned design of the backgammon game that is to work as domain for a backgammon playing fuzzy controller. The design is not so detailed that every exact method is defined. The design is instead done on a more overall level where the different modules, their possible classes and their functionality is identified and explained. As the application only is to be developed by one person, was this considered to be the best solution between having a general idea on how the development are to be approached, while still not wasting time on making a design that anyway would be changed due to better understanding when making the actual

implementation. A later section will then give a more detailed explanation of the final design and actual implementation.

The overall application is to be developed as modules. Each module is to have a clearly defined purpose. The application is to be developed in the .Net (dotNet) environment using the C# (C sharp) language. In .Net a module is refereed to as an Assembly. An Assembly in .Net can be compared to a package in the Java language or simply a dynamic link library (dll) file in basic windows programming.

Each module may contain multiple classes. Some may be public classes that can be accessed by other modules, while other classes may be private, meaning they can only be accessed and used by other classes within the same module.

General overview

The overall idea is that the application is to be made up of three main modules as well as what will be refereed to as decision modules. The main modules are a module representing the game of backgammon, a module representing a computer agent able to play the game of backgammon, and then finally the graphical user interface of the application. Additionally the decision modules are the part of the application that in reality are to decide what move the agent is to play in a given situation. Fuzzeval as well as Pubeval is then to be implemented as decision modules. A graphical representation can be seen in Figure 52.

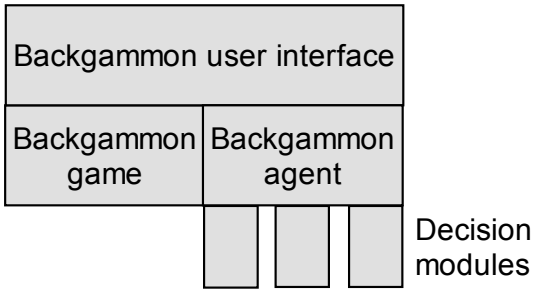

Figure 52: Module overview and their relationship

Backgammon game

The first thing to develop is the actual representation of the backgammon game itself. This representation is not the application as a whole with graphical user interface, computer opponents, etc. It is the part of the application that is the representation of a backgammon game.

This module is to have one class acting as the interface to the backgammon game. This class should have all methods required to play the game of backgammon. This should include methods to start a new game, end a turn, roll the dices, make a move, etc. Other methods to obtain the current status of the game such as the current board layout, dice values, the winner, whose turn it is to play, etc. are also to be available from this interface. The module is to be responsible for all the game logic, meaning it should insure that invalid moves can not be made, a turn can not be ended before all valid moves are used, correct switching of the board etc. The module should in other word be responsible for everything from representing the game to ensure valid play.

One additional requirement to this module is the ability to obtain a history of a game. Although the exact solution model of Fuzzeval is not determined yet, is it likely the learning scheme will be based on some kind of analysis of the whole game after the outcome has been decided (reinforced learning). The agent should therefore, after a game, be able to obtain a history of all the boards that has been played during a game.

An initial UML description of the whole module can be seen in Figure 53

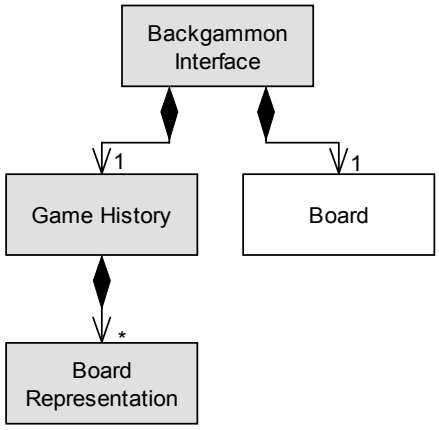

Figure 53: Initial UML of the backgammon module

Backgammon Interface is the most important class of this module. It is to hold all the methods required to play the game of backgammon. From this class it is possible to obtain a history of the current game. This history will hold a range of objects where each of them expresses a board layout in some simple form that has come up during the game. *Board* is a private class the *backgammon interface* most probably will require to keep track of the current board layout in a game in progress.

The whole backgammon module might in its final form hold additional classes if necessary. This will however most probably be relatively simple helper classes. An example could be a class representing the dices.

Backgammon agent

This modules main class is a class representing a computer player playing backgammon. The overall idea is that during instantiation of the agent, it obtains an association to the backgammon game. During each turn the agent then checks weather it is its turn to play. In the case it is, will it generate a list of the possible moves, and thereby a list of the possible boards that can be created. This list of possible boards is then sent to all the decision modules that are responsible for assigning a score to each board, depending on how good that board is estimated to be. The move that has received the highest overall score from all the decision modules is to be played. After the game has ended the game history is obtained from the backgammon game and presented to all the decision modules. A decision module then has the opportunity to learn from the game that it has just been playing.

A requirement to the decision modules is that they should be relatively easy to add and remove without recompilation of an agent. It is therefore desired that the decision modules used by an agent, can be configured in a simple text file. Then when an agent during runtime is instantiated, it simply dynamically loads all the decision modules it has been configured to use.

An initial representation of the module as a whole and its possible classes can be seen in Figure 54.

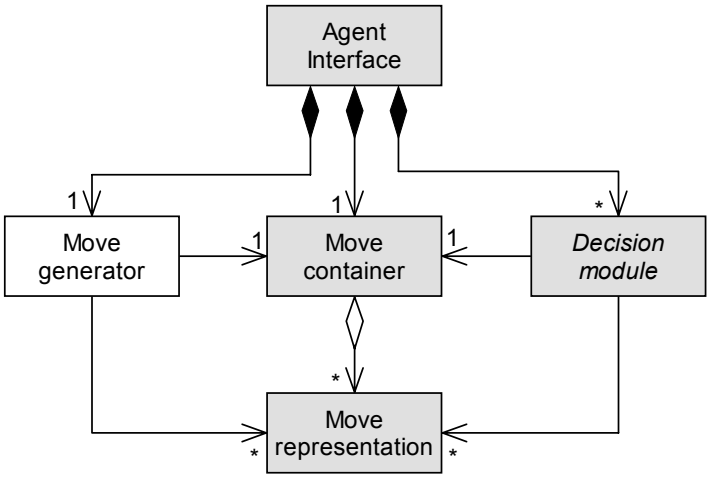

Figure 54: Initial UML of the agent module

Agent interface is the public class representing the interface to an agent able to play backgammon. During instantiation it should receive a reference to the backgammon game contained in the backgammon module. *Move generator* is a private class intended to have the responsibility of generating all the possible moves from the given situation in the backgammon game. Each possible move is then represented in some simple form by an object of the *move representation* class. The *move container* class then holds all these possible moves. The *decision module* is an abstract class every decision module must inherit from. *Move container*, *move representation* and the *decision module* interface are all public, as they are to be used and accessed by the actual decision modules. In its final form this module may end up containing additional classes if necessary.

Backgammon user interface

The user interface is expected to hold the entry point of the whole application, and will as a consequence be responsible for instantiating the *backgammon interface* class in the backgammon module, as well as two instances of the *agent interface* class contained in the agent module. Two instances are required to enable one agent using the Fuzzeval decision module to play another agent using the Pubeval decision module. Besides representing the view into the state of the backgammon game, the user interface is responsible for obtaining requests such as a player move or a dice roll from the user of the application, and forward into the backgammon game.

The user interface module will most likely consist of one main class making up the main interface as well as some additional classes representing necessary dialog boxes and etc.

Decision modules

The decision modules are the part of the application responsible for making the application able to play backgammon. They are to inherit from the *Decision module* interface in the agent module. Decision modules will be dynamically loaded by an agent when it is instantiated at run time. During play a decision module will at each turn receive a list of possible boards from the agent. It is then a decision modules responsibility to grade the strength of each board. At the end of a game each decision module are to receive a history of the game that has just been played. This history can then be used to learn from.

Multiple decision modules should of course also be able to co-operate within the same agent if desired, as this simply is a matter of letting each decision module grade a board and then sum all decision modules grade. A requirement is of course then, that all decision modules score must be normalized to a specific range. An agent based on multiple evaluators such as Fuzzeval, Pubeval, an opening book, a bear off database an even other modules should therefore become a possibility. It is simply a matter of develop a specific module, and then configure an agent to also include this module in its decision making.

6.7 Implementation and final design

Based on the design presented in section 6.6, has the backgammon application, as well as at this point, one decision module based on Pubeval been implemented. This section will try to give a more detailed presentation of the final architecture of the backgammon application, and how it has been implemented. The section will present all the classes making up the whole application by use of UML. Public classes within a module will be represented by the color gray, while private classes will be white. Public classes within a module will additionally be presented with an interface description. It should be noted that some names of the classes initial presented in section 6.6 has been changed.

6.7.1 GammonGame

GammonGame is the name of the module making up the representation of a backgammon game. A complete UML description of this module can be seen in Figure 55.

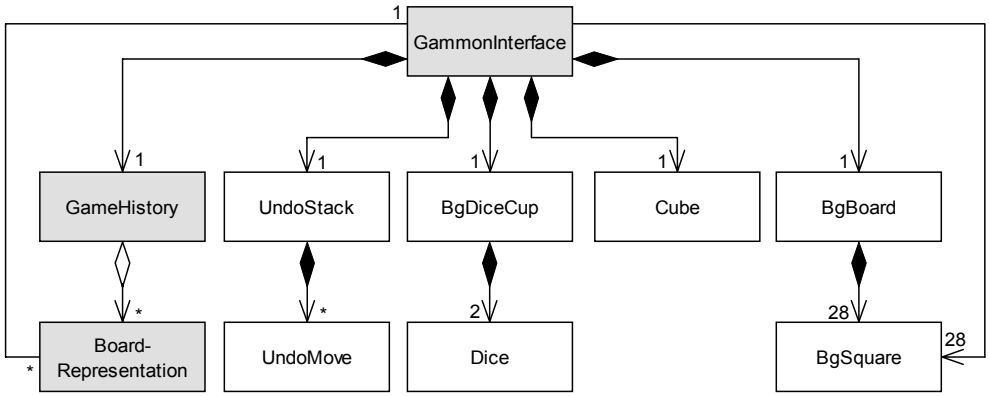

Figure 55: UML description of the module GammonGame

It can be seen that some classes has been added to the initial design presented in section 6.6. These classes are however all private classes with rather limited functionality. *GammonInterface*, *GameHistory* and *BoardRepresentation* are therefore still the only classes accessible outside this module. It is *GammonInterface* that is responsible for instantiating objects of the type *BoardRepresentation* but it turns the ownership of these objects over to *GameHistory* right after this instantiation.

Additionally this module also holds a public enumeration defined as *BgPlayer* with the three possible values *None*, *Dark* and *Light*. This enumeration is used throughout the application. But it is not included in the UML diagrams because an enumeration can be viewed as a simple data type.

UndoStack keeps track of the moves done in each turn. This ensures the functionality required such a move can be undone. A move can however only be undone until the dices get rolled again and the turn is ending. The number of *UndoMoves* hold by *UndoStack* will therefore in reality never be more then four as it gets emptied at each end of a turn.

BgDiceCup represents as the name indicates the dice cup or shaker. In reality is it just a very simple class holding two objects of the type *Dice* that is a wrapping of a randomizer in the range one to six.

The class *Cube* is representation of the doubling cube. This class is also rather simple without much functionality.

BgBoard is the representation of the current board layout, of the game in progress. It holds 28 objects of the type *BgSquare*. A *BgSqure* represents a board location. A board location can either be one of the 24 standard locations, one of the two bar locations or one of the two bear off locations. An object of *BgSquare* keeps track of the number and color of checkers it holds, while *BgBoard* assemble all these *BgSquare's* to represent a board.

GammonInterface

GammonInterface is the most interesting class of this module. This class holds all the methods required to play a valid game of backgammon. All public methods are presented below.

GammonInterface()	*//Class constructor*
void NewGame()	*//Starts a new game*
bool MakeMove(int from, int to)	*//Makes a move (returns false if invalid)*
bool EndTurn()	*//Ends turn (switch player and roll dices)*
void DoubleCube(BgPlayer player)	*//Doubles the cube*
BgPlayer Winner()	*//Returns the winner if any*
int VictoryType()	*//Returns 1 (reg.), 2 (gam.), 3 (backgam.)*
void Undo()	*//Undo last move (if EndTurn not called yet)*
GameHistory GetHistory()	*//Returns the game history*
bool ValidateMove(int from, int to)	*//Returns true if move is valid*
BgPlayer CurrentPlayer()	*//Returns the current player to play*
BgPlayer CurrentOpponentPlayer()	*//Returns player not playing*
int BoardSquareCount()	*//Returns board square count (26)*
BgPlayer SquareOwner(int square)	*//Returns owner of a square*
int SquareCheckerCount(int square)	*//Returns amount of checker on square*
int PiecesRemoved(BgPlayer player)	*//Returns checkers beard off by player*
int GetDiceValue(int idx)	*//Returns value of a specific dice (1 or 2)*
int GetCubeValue()	*//Returns value of the cube*
int GetMove(int idx)	*//Gets a move (1-4) returns 0 if none*
int GetPips(BgPlayer player)	*//Gets the pips left for this player*
void SetDiceValues(int val1, int val2)	*//Sets the current value of the dices*
void SetNextDiceValues(int val1, int val2)	*//Sets the next dice values to roll*
void EditBoard()	*//Enable board edit*
void PutPiece(int square, BgPlayer player)	*//Place checker on square (If edit enabled)*

One important property of this class must be known. That is that after each turn, the board is switched around, such the squares (board locations) 1 – 24 always are viewed from the current player's point of view (current players pieces move in the negative direction from 24 to 1). Position 25 represents the current player's bar location, and element 0 represent opponent's bar location. This property has relevance when using the methods *MakeMove(…)*, *ValidateMove(…)*, *SquareOwner(…)* and *SquareCheckerCount(…)*. This is illustrated in Figure 56.

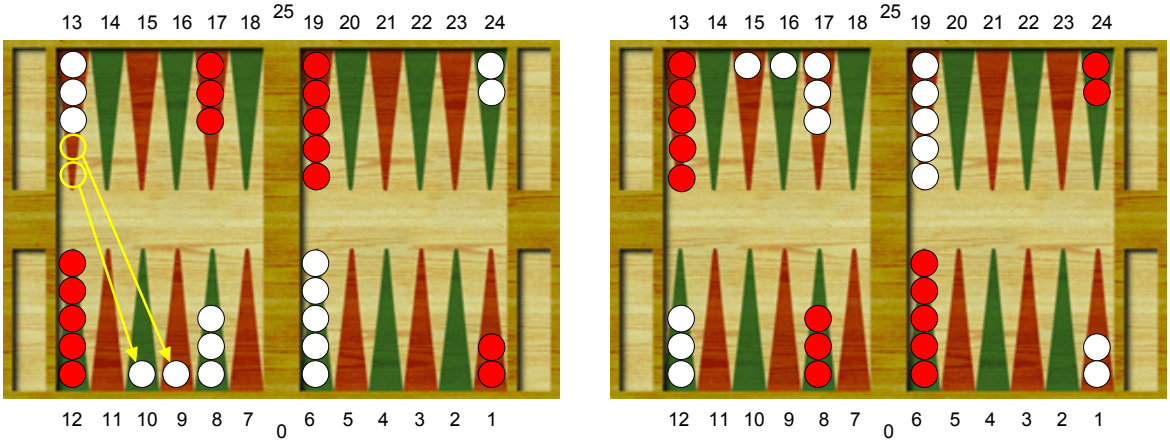

Figure 56: View of the same board before and after switch

It is whites turn and the view of the board is as illustrated in the left image. After white has performed its move, *EndTurn()* is called and the board get switch around to the one illustrated in the right most image. The current player that is now red will therefore see the board illustrated in this right image. This form of board switching ensures that an agent will always see an identical board in the same manner no matter playing side.

Many of the methods on the class should be self explaining but some of them still require a little more detailed explanation.

Winner() returns the winner of the game, if the game is not over yet, it simply returns *none*. *VictoryType()* returns the type of victory. 3 means a backgammon, 2 a gammon, 1 a regular win and finally 0 if the game is still in progress.

BoardSquareCount() will return the number of squares (board locations) on the board, this is simply always the value 26 (24 regular squares plus 2 bar squares). This method is useful when iterating over the board and accessing all squares with the methods *SquareOwner(…)* and *SquareCheckerCount(…)*. The bear off squares can not be directly accessed but *PiecesRemoved(…)* will return how many pieces a player has removed from the board.

GetDiceValue(…) will return the value of either dice 1 or 2, while *GetMove(…)* will return what moves still is available. This method might be a little hard to understand but it is in reality rather simple. Valid parameters for *GetMove(…)* is in the range 0 to 3. Then if a double six has been rolled, will it return the value six for all these four inputs. Then as some of these moves get used, will it start to return 0 where this used move was represented before. Another example could be the dice roll 4 and 2. Then the return value for *GetMove(…)* with 0 as parameter would be 4, with 1 as parameter 2, with 2 as parameter 0 and with 3 as parameter 0. After a piece has been moved two squares forward, will *GetMove(…)* instead return 0, when given the parameter 1.

EditBoard() enables the board to be edited. Specific situations or board layouts can then be created by placing pieces with the *PutPiece(…)* method.

GameHistory

GameHistory is a class that actually just encapsulates a dynamic array. The interface below should be pretty self explaining.

GameHistory()	*//Class constructor*
void Add(BoardRepresentation rep)	*//Adds BoardRepresentation as last element*
BoardRepresentation PeekFirst()	*//Returns first element*
BoardRepresentation PeekLast()	*//Returns last element*
BoardRepresentation RemoveFirst()	*//Returns and removes first element*
BoardRepresentation RemoveLast()	*//Returns and removes last element*
BoardRepresentation Peek(int idx)	*//Returns the element at position 'idx'*
BoardRepresentation Remove(int idx)	*//Returns and removes element at position 'idx'*
int Count()	*//Returns number of elements*
void Clear()	*//Removes all elements*
public BgPlayer Winner	*//Get and sets the winner of the game (dark/light)*
public int WinType	*//Get and sets the type of win*

The idea is that after a game has ended, an agent and its decision modules will get a reference to an object of this type. It can then iterate through all the elements and obtain a reference to an element of the type *BoardRepresentation*. The last element will describe the final board where one of the players has won because all pieces have been removed from the board. From this element every other element then represents a board played by the winner and the other half represents a board played by the loser. For example if the final board is positioned at index 52, will all wining boards be placed at even indexes (52, 50, 48, …), while losing boards will be positioned at odd indexes (51, 49, 47, …).

BoardRepresentation

BoardRepresentation will in a simple way describe a board layout that has been represented in the game.

BoardRepresentation(GammonInterface game)	*//Class constructor*
int SquareCount()	*//Returns number of squares (26)*
int GetPiecesAt(int square)	*//Returns pieces at index / square*
int BearOffCountCurrent()	*//Returns players removed pieces*
int BearOffCountOpponent()	*//Returns opponents removed pieces*
bool PatternMatches(BoardRepresentation repr)	*//Returns true if 'this' equals 'repr'*

A specific square (board location) can be obtained by *GetPiecesAt(…)*. The value 1 – 24 corresponds to square 1 – 24 from a current player's point of view (current players pieces move in the negative direction from 24 to 1). Current player's pieces are represented by positive integers, while opponent's pieces are represented by negative integers. Element 25 represents current players pieces on the bar (positive integer), and element 0 represents opponent's pieces on the bar (negative integer). The amount of pieces a player as removed from the board can be obtained by the methods *BearOffCountCurrent()* and *BearOffCountOpponent()*. *PatternMatches(…)* is simply a method that will returns true if this instance of *BoardRepresentation* equals the parameter instance *repr*.

6.7.2 GammonAgent

GammonAgent is the module in which an agent able to play backgammon is contained. The UML diagram illustrating the classes in this module can be seen in Figure 57. It should be noted that the three leftmost classes *GammonInterface*, *GameHistory* and *BoardRepresentation* are the public classes from the

GammonGame module. These three classes are not contained in the *GammonAgent* module but are included to illustrate their relation to classes contained in the *GammonAgent* module. The classes represented by a white color are the private classes of the module.

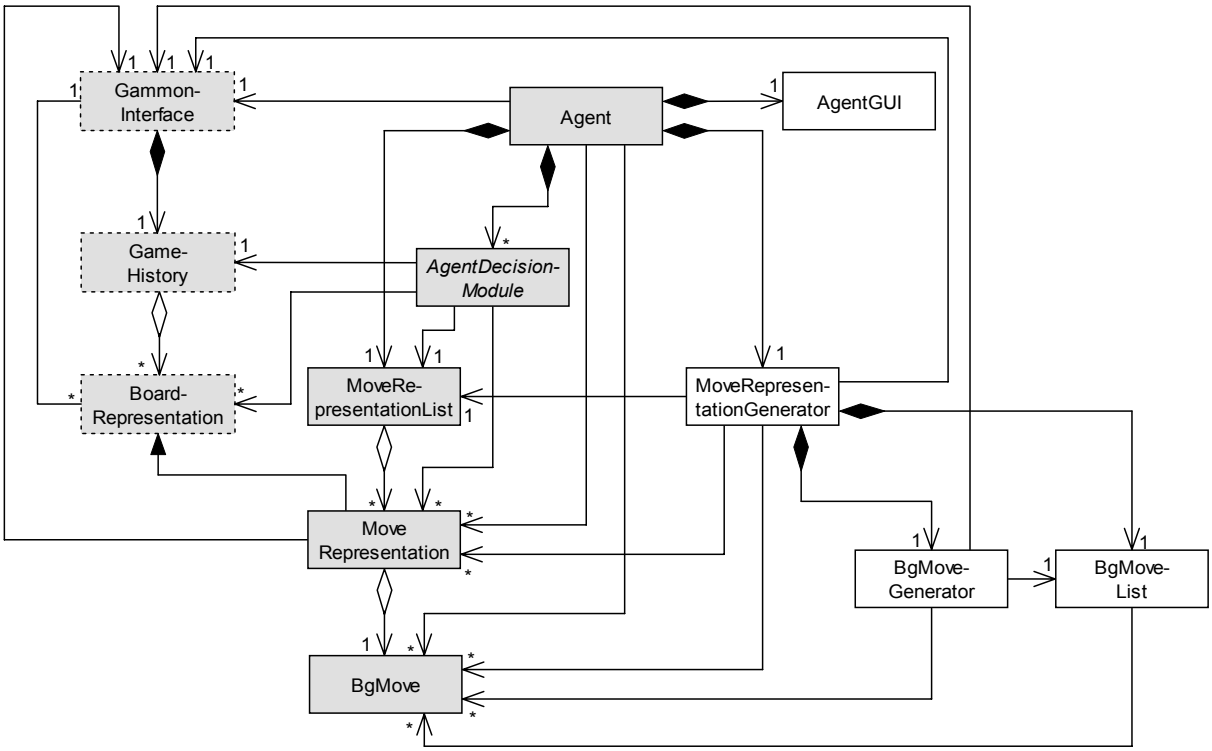

Figure 57: UML of GammonAgent, the three leftmost classes are from GammonGame

A diagram that might look complicated when looking at it. *Agent* is the central class in this module. When instantiated it receives a reference to *GammonInterface* which is the backgammon game running in the core of the whole application. *Agent* has its own graphical user interface only it can access. When an *Agent* is instantiated, it also loads and instantiates all its decision modules, as well as a *MoveRepresentationList* and a *MoveRepresentationGenerator*. The *MoveRepresentationGenerator* holds a *BgMoveGenerator* object responsible for generating all the valid moves, represented by the *BgMove* class, which can be performed in the backgammon game. These *BgMoves* are temporarily held by the *BgMoveList*. When all the valid moves that can be performed, has been generated, is it the *MoveRepresentationGenerator's* responsibility to create all the possible new board layouts, represented by the *MoveRepresentation* class, which can be created in the game. Each *MoveRepresentation* ends up holding the *BgMove*, initially created by the *BgMoveGenerator*, necessary to create this new board layout. After the creation of the *MoveRepresentationList,* is it send to all the decision modules loaded by the agent. It is then their responsibility to grade all the boards. If desired the agent can then as the final task perform the move associated with the *MoveRepresentation* assigned the highest score by the decision modules.

Agent

This is the class representing a computer player, which by help of its decision modules, is able to play the game of backgammon. Its interface can be seen below.

```
Agent( GammonInterface game, string caption, string confFile )      //Class constructor
bool IsAgentMoveing()                                  //Returns true if it is agents turn to move
bool IsAgentPlaying( BgPlayer player )                 //Returns true if agent is playing as 'player'
void ViewBoard()                                       //Generate and grade every possible board
BgMove CommitMove()                                    //Performs the move considered to be best
void Learn()                                           //Allows the agent to learn from game history
void ShowAgentGUI()                                    //Shows the agents user interface
bool OutputMovesAndScore                               //if true the GUI is updated with possible moves
```

When *Agent* is instantiated it receives a reference to an instance of the backgammon game. *Caption* is simply a text string such that multiple instantiated agents can be distinguished and *confFile* is the path to a file configuring what decision modules the agent should load and use.

ViewBoard() will make the agent generate all the possible moves that can be performed in the game, and then let the decision modules grade each of these possible boards. If it is the current agents turn to play *CommitMove()* will perform the move that has received the highest score.

When a game has ended, *Learn()* can be called to send the game history to all decision modules. A decision module then has the option of learning from this game history.

OutputMovesAndScore is a public variable that determines if the agent should update its own user interface with the possible moves when *ViewBoard()* is called. Setting this value to false will increase performance if the agent is playing a range of games again and again as fast as possible.

MoveRepresentationList

This class is a list holding all the board representations to be sent to the decision modules.

```
MoveRepresentationList()                                                    //Class constructor
void SortByScore()                                                          //Sort representations by score
void AddMovRep( MoveRepresentation MovRep, bool addIfExists )      //Adds representation
MoveRepresentation GetMoveRepresentation( int idx )                         //Returns representation at 'idx'
int Count()                                                                 //Returns representation count
void Clear()                                                                //Clears the list
```

When a move representation is added to the list the variable *addIfExists* determines if a move representation that already exist should be added. This is because different moves creating the same board might exist. Then if a move representation is added to the list, and another move representation having the same final outcome exist, this other move representation is discarded. *AddMovRep(…)* is in reality never called in the application with the *addIfExists* parameter set to true.

MoveRepresentation

MoveRepresentation does in a simple way represent a possible new board that can be created. It is these *MoveRepresentations* that is to be graded by all the decision modules (Fuzzeval and Pubeval). It inherits from the class *BoardRepresentation* contained in the *GammonGame* module.

```
    MoveRepresentation( GammonInterface game, BgMove move )     //Class constructor
BgMove GetMoves()                                   //Gets the move to create this board
void AddScore( double score )                       //Adds a score to this board
double GetScore()                                   //Gets the average score of the board

/*---inherited from BoardRepresentation ---*/
 int SquareCount()                                  //Returns number of squares (26)
int GetPiecesAt( int square )                       //Returns pieces at index / square
int BearOffCountCurrent()                           //Returns players removed pieces
int BearOffCountOpponent()                          //Returns opponents removed pieces
bool PatternMatches( BoardRepresentation repr )     //Returns true if 'this' equals 'repr'
```

When a *MoveRepresentation* is instantiated it receives a reference to the running backgammon game and a possible move that can be done. The access methods *GetPiecesAt(…)*, *BearOffCountCurrent()* and *BearOffCountOpponent()* then returns information corresponding to the new board.

AddScore(…) is the method all the decision modules are to call to give their score to this board. There is however the requirement that all decision modules must agree on the range of the score when calling this method. It is not desirable for example that one decision module gives a score in the range -1 to 1, while another decision module gives a score in the range 0 to 100. No forced control is given to solve this problem but the class *AgentDecisionModule* offers a method able to normalize the scores. There is however no explicit control forcing a concrete decision module to use this method.

BgMove

This is the class that represents a backgammon move. *BgMove* is actually a linked list with a references to other instances of it self. This is because a move in backgammon actually can exist of up to four single moves.

```
    BgMove( int from, int to, BgMove precedingMove )    //Class constructor
int From                                                //Square id to move from
int To                                                  //Square id to move to
BgMove FirstMove                                        //The first move
BgMove NextMove                                         //The next move (null if 'this' is last)
static BgMove CloneFromFirst( BgMove move )             //Clone 'move' to new instance
```

BgMove is simple class consisting of only one constructor, one static method and four public variables. It is the *BgMoveGenerator* that is responsible for instantiating these *BgMoves*, but it is a *MoveRepresentation* that ends up owning them. *CloneFromFirst(…)* is a method used by the *BgMoveGenerator* to create a new instance of a move that is equal to the move given as parameter.

AgentDecisionModule

This is an abstract class all decision modules must inherit from. Its interface can be seen below.

```
abstract string NameId()                                  //Returns name of module
abstract void GradeBoards( MoveRepresentationList list )  //Grade all boards in list
abstract void Learn( GameHistory history )                //Learns from the game history
protected void ScoreNormalizer( double[] scores )         //Offer normalization of scores
```

NameId() is a simple method a concrete decision module must implement to return a name that identifies it.

ScoreNormalizer(…) is a protected method a decision module can chose to use to normalize the scores it assigns to all the move representations. A decision module can then temporarily store all scores in a array of doubles and let the *ScoreNormalizer(…)* method normalize the scores, before they finally are assigned to all the move representations. This normalization is done such that the highest scoring board will get the value 1 assigned, while the lowest scoring board will get the value -1 assigned. Intermediate scoring boards will get a score between 1 and -1 with the same relative "distance" to the lowest and highest scoring board as before the normalization.

To exemplify this consider the case where a decision module has graded four boards with the following four scores:

{-3.4; -0.1; 2.8; 5.3}

These scores can then be considered x-points on a straight line in a coordinate system where the y-points are the normalized score. The y-point for the highest scoring board should then correspond to 1 and for the lowest scoring board correspond to -1. This is illustrated in Figure 58.

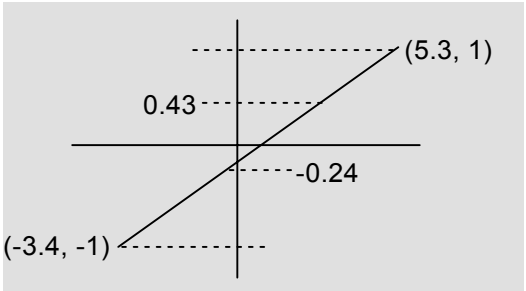

Figure 58: Score normalization

The resulting normalized score is then {-1; -0.24; 0.43; 1}.

Another more simple solution would simply be to have the requirement that a decision module just had to grade a board in the range -1 to 1. The drawback with this slightly simpler method is that one decision module might grade every board relatively high, while another decision module might grade every board relatively low. The decision module grading boards high would then have a higher influence on the final decision.

This method does however solve this problem by normalizing into a concrete interval identical to all decision modules, ensuring all decision modules having the same influence on the final decision.

6.7.3 GammonGUI

This module only holds one class of interest named *MainForm*. That is the class making up the whole graphical user interface of the application. This class is also the entrance point of the whole application and is responsible for instantiating two agents and the backgammon game thereby melting it all seamlessly together. The whole module does hold some additional classes that are dialog boxes. These classes are however omitted from the UML diagram illustrated in Figure 59. This diagram illustrates the user interfaces relation to other classes in the whole application. It should be noted that only public classes are included from the *GammonGame* and *GammonAgent* module.

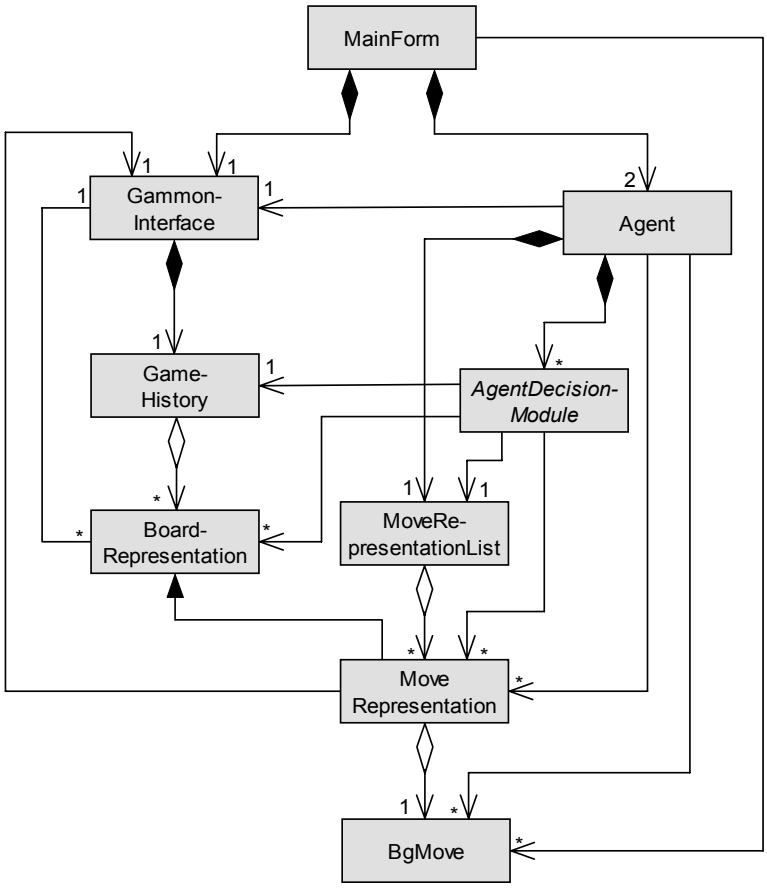

Figure 59: MainForm's relation to public classes in GammonGame and GammonAgent

The only thing to comment in this diagram is the association from *MainForm* to *BgMove*. The reason for this association is that the agent returns a reference to the move it performs. This reference is then used when the *MainForm* prints the move performed on its user interface.

6.7.4 PubevalWrapDM and PubevalWin32

PubevalWrapDM is the name of the first decision module implemented. It works as a wrapper interface to the standard Win32 dynamic link library (dll) named *PubevalWin32* implementing the Pubeval evaluator by G. Tesauro. An initial fear before starting the implementation of the Pubeval decision module was that the whole source code originally written by G. Tesauro had to be rewritten into managed code (managed code is code written and compiled to run in the Microsoft .Net framework/platform, unmanaged code is code written and compiled to run on the standard Windows platform without the control of a virtual machine). This was however fortunately not the case as the source code, without any trouble, was able to compile to a standard unmanaged Win32 dynamic link library under the windows platform. Only required alteration of the source code was that two of the methods had to be marked as dll exports.

PybevalWrapDM is the decision module that from the backgammon applications point of view is implementing Pubeval. It consists of two classes named *PubevalWrapper* and *PubevalDM*. Their relation to other classes can be seen in Figure 60. Classes with a doted edge are classes contained in other modules and the white box *PubevalWin32* is in reality not a class but the dynamic link library actually implementing Pubeval.

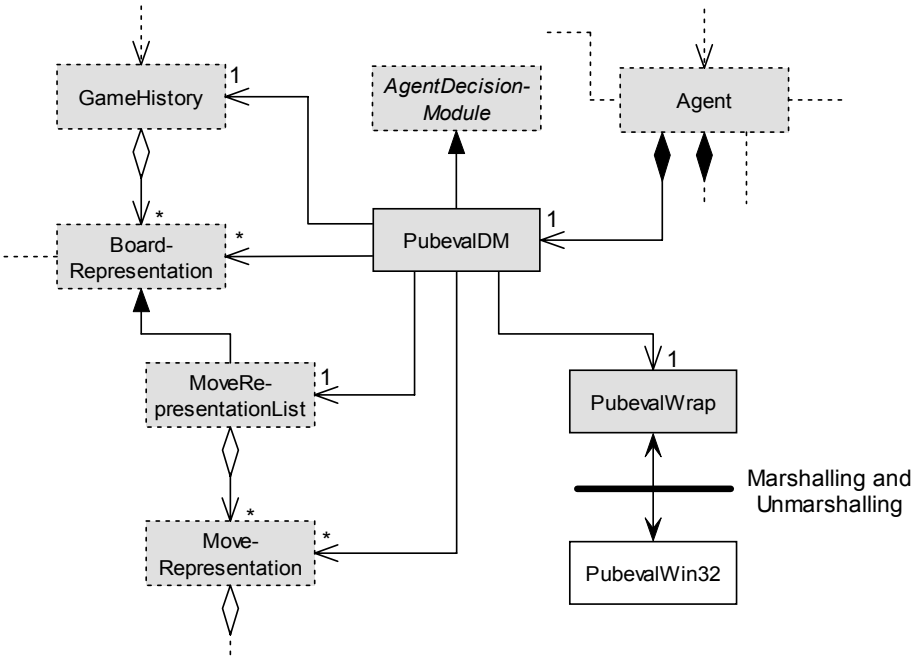

Figure 60: UML describing the implementation of Pubeval

PubevalWrapper is the class wrapping the two exported functions from *PubevalWin32* into managed code. *PubevalWrap* has two methods exactly identical in return value and parameter list, as the two exported functions from the *PubevalWin32* dll. These two methods can be called just as regular methods by *PubevalDM*. *PubevalWrap* will then in a completely transparent manner call the methods in *PubevalWin32* while ensuring a correct marshalling between the managed and unmanaged code. *PubevalDM's* responsibility is then primarily to convert every *MoveRepresentation* into

the format taken in the *PubevalWrap* methods and afterwards normalize all obtained scores before returning them to the agent.

6.7.5 Flow diagram

The presentation of the implemented backgammon application has to this point been rather technical and possibly a little hard to grasp. Figure 61 is a little less complicated diagram trying to illustrate the overall flow between the core of the application that is the representation of a backgammon game, and the agents and their decision modules. The user interface is however omitted in this diagram.

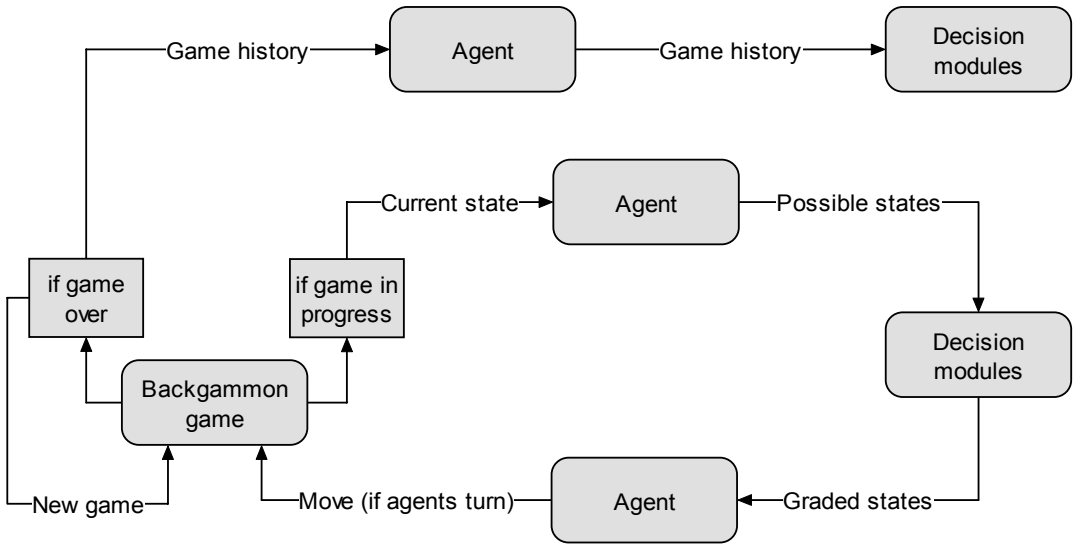

Figure 61: Backgammon application flow

6.8 Application overview

This section will present the overall backgammon program and its capabilities in a more understandable manner. The section will take basis in the graphical user interface of the application, thereby giving an explanation of the domain Fuzzeval is to be tested in.

When launching the program the interface presented in Figure 62 will show up.

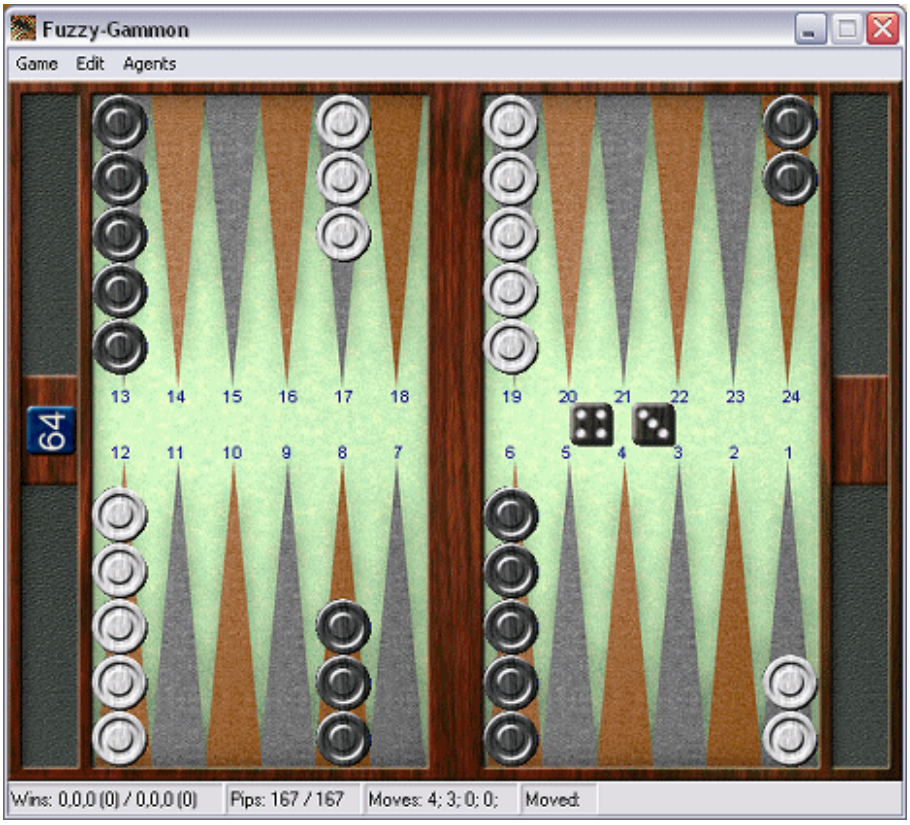

Figure 62: User interface of the backgammon application

The backgammon board on which the game is played takes up most of the user interface. When the application is launched, is the light playing side assigned to an agent. The dark playing side is at this point not assigned to any of the agents and is therefore playable by a human. A checker can be moved by clicking the mouse on the board location from which the checker is to be moved from, and then afterwards clicking the board location the checker is to be moved to. If the move performed is invalid nothing happens. When all valid moves have been done, the turn is ended by a mouse click on the dices. In the case more valid moves exist nothing happens.

Although the backgammon board internally is switched around as explained in section 6.7.1, is this not done on the user interface. This is because the user interface counter switches the board such that a human player always will view the board from the perspective of playing dark. The numbers identifying the different board locations will however be switched around at each turn to correctly identify a location as the same value as done internally in the game.

At the bottom of the user interface a status bar like the one illustrated in Figure 63 provides one with some general information about the status of the backgammon game(s).

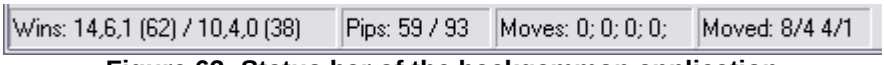

Figure 63: Status bar of the backgammon application

The left most panel is the only panel not related to the current situation in the game in progress. This panel outputs how many victories and type of victory, each of the players have as well as a number indicating the percentage of victories between the players. The numbers left of the slash shows the victories obtained by dark and the numbers right of the slash shows the victories obtained by light. It can in this example be seen that dark has fourteen regular victories, six gammon victories and one backgammon victory, while light has ten regular, four gammons and no backgammon victories. The numbers in the parentheses, indicating the percentage of victories, are calculated such that a backgammon victory is equal to three regular victories, while a gammon victory is equal to two. This means that if dark has two regular victories and light has a regular and a gammon victory, will the distribution not be 50/50 but 40/60 in lights favor. This is an important property to remember and all testing throughout the last part of this paper will be done by these calculations and not by the raw amount of games won or lost. This panel can be reset to zero by right clicking it with the mouse.

The middle left panel shows how many pips each player has left. Pips can be explained as the “distance” a player has left until every piece is removed from the board and the game is won. Pips can be used as a general indicator of the current status of the game, and which players that seems to be ahead. The value left of the slash is dark players remaining pips, while the value to the right is light players remaining pips.

The middle right shows the remaining moves a player have in his turn. In this example all the move has been used, but in Figure 62 can it be seen at both moves is still available.

The final panel is the right most. This panel outputs the moves done by an agent. In this example can it bee seen that the agent first has moved a piece from location 8 to location 4 and then moved the same checker from location 4 to location 1. That these moves have been done by an agent, explains why the middle right panel indicates that no more moves are available.

At the top of the application the main menu is located. The items contained in this menu can be seen in Figure 64.

Figure 64: Main menu items

In the *Game* menu is it possible to start a new game. Some general info about the application is also available, as well as the possibility of exiting the application. The

Edit menu holds three items useful for manipulating the game. *Undo* will undo the last move done by a human player in a turn. *Undo* can not undo a move done by an agent. If enabling *Manual Dices*, is the dialog box show in Figure 65, going to be shown at each dice roll.

Figure 65: Dialog box to set the next dice roll

One can then explicit set the dice values to roll, and enable one to copy dice rolls from another backgammon program thereby letting Fuzzeval play another backgammon application where a human is working as interface between them. If one is suspecting that the application cheats by looking ahead in the dice rolls, one can also test that this is not the case, and that the playing strength remains the same when dice rolls are manually entered. Last item in the *Edit* menu is *Edit Board*. Enabling this item will allow one to manually place the pieces around the board, thereby enabling on to create a specific board layout.

The *Agent* menu holds five items. Selecting either *Show Agent 1* or *Show Agent 2* will show the user interface of one of the agents. An agents user interface looks like the one illustrated in Figure 66.

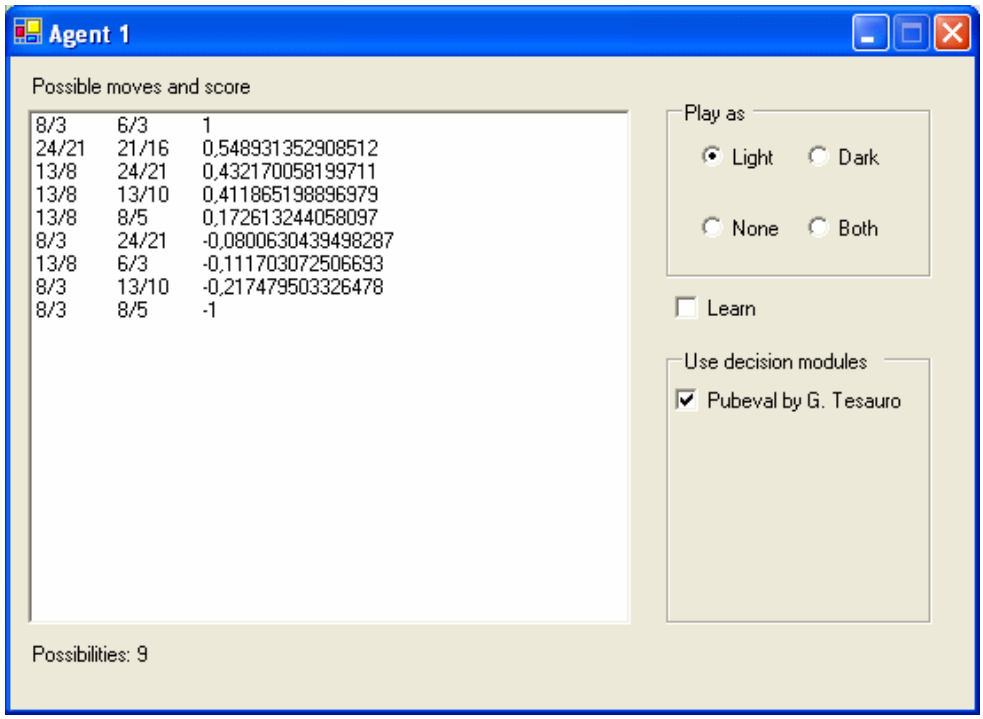

Figure 66: Agent user interface

This user interface will for a given situation in the game output the move possibilities, as well as the associated score that has been assigned by the agent's decision modules. It is also possible from this user interface to set what piece the agent should play as. In this example this specific agent is playing light. It can also be set if the agent calls the *Learn(…)* method on checked (used) decision modules when a game has been completed, such a decision module has the ability to learn from the history of a completed game. Finally is it possible to set what possible decision modules the agent should use. Possible modules can be configured in an associated configuration file. A checkbox will then automatically be added for this decision module. A decision module can then be checked or unchecked depending on if the agent should use this decision module in its decision making.

The next two elements in the *Agents* menu are *Auto Play (Animated)* and *Auto Play (Fast)*. In the case that both playing side is assigned to an agent, will one of these two menu items let the agent(s) play game after game without human intervention. This is continued until interrupted by clicking the last menu item named *Stop Auto Play*. The difference between the two is that *Auto Play (Fast)* is much faster, but one will not be able to actually view the games played. *Auto Play (Animated)* is quite slow; one will however be able to view each game as progressing.

An alternatively if one really wants to study the play done by two agents, is simply to assign and agent to both playing sides and then manually click the dices at the end of each turn.

6.9 Conclusion on the backgammon domain

A backgammon application capable of working as domain for Fuzzeval has been implemented. The application fulfils the requirement needed to work as test domain. The design is flexible enough such that different board evaluators (decision modules) easily can be added to and removed from the application.

The application supports the required features such an agent relying on Fuzzeval is able to play humans, as well as other agents relying on other decision modules such as Pubeval. An agent can additionally be set to play against itself thereby selecting moves for both sides.

During each game the history of the game is recorded. After completion of a game this history is shown to both agents and their loaded decision modules.

In itself then the implementation of the backgammon application and the Pubeval module, completes the task of developing a backgammon application offering an actually quite good opponent. The application does however have the limitation that cube doubling for a human user as well as an agent is disabled. In the core of the application where the backgammon game is represented, is cube doubling however still supported.

Extensive testing has been done against GNU-Backgammon to ensure that the developed backgammon application behaves in a correct manner and only offers valid play. This extensive testing has however not revealed any flaws with the move and board generators or Pubeval. Generated moves for tested situations are identical in the two applications. The play done by Pubeval is also identical in the two applications.

7. Developing Fuzzeval

This chapter will describe the development of the backgammon playing fuzzy controller named Fuzzeval. The development process has been an iterative process where a changes and additions continually have been done. A hundred percent exact description of Fuzzeval in its final form is as a consequence difficult to give. The chapter should however still give a pretty accurate description of both the fuzzy controller contained in the hart of Fuzzeva,l and the calibration methods responsible for learning.

The sections contained in this chapter are:

- How to model a backgammon playing fuzzy controller
- Supporting advices taking
- Initial design overview
- Fuzzy controller implementation
- Function calibration
- Rulebase learning
- Conclusion on developing Fuzzeval

7.1 How to model a backgammon playing fuzzy controller

This section will exemplify how a fuzzy control architecture that plays backgammon can be created. The idea is that the fuzzy controller receives a board as input, and then outputs a value indicating the goodness or strength of that board. At each turn of play, all the possible boards from a given situation, is then generated and evaluated by the fuzzy controller. The board that receives the highest score is then to be played.

The first step when creating a fuzzy controller is to identify and define the linguistic input and output variables, and their associated linguistic terms. For the game of backgammon then these linguistic input variables can be the different rules of thumb one as human considers before making a move. Some examples could be the amount of single standing pieces, the amount of opponent pieces hit back on the bar, the amount of own pieces moved into ones own home board, the length of a consecutive blockade, the amount of locations blocked and etc. For this example we will consider the three linguistic input variables and their terms presented in Table 24.

Linguistic variable	Linguistic terms	Base variable
SinglePieces	Few, Many	0 to 15 pieces
OpponentOnBar	Few, Many	0 to 15 pieces
SinglePiecesInHome	Few, Many	0 to 6 pieces

Table 24: Linguistic input variables and terms considered by fuzzy controller

And the linguistic output from the controller is presented in Table 25.

Linguistic variable	Linguistic terms	Base variable
BoardStrength	Weak, Good	-1 and 1

Table 25: Linguistic output

The next step is to define membership functions for the different linguistic input variables and associated terms. Example membership functions for the input variables are illustrated graphically in Figure 67.

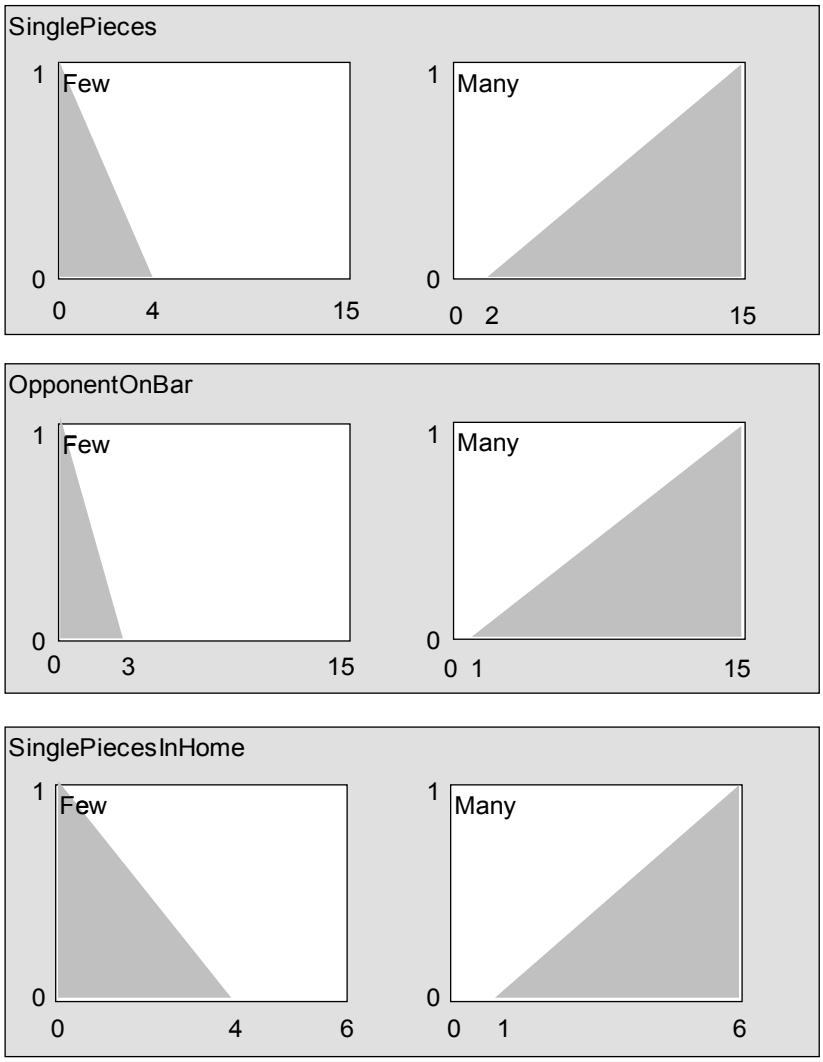

Figure 67: Example of membership functions

The last step is to define the rulebase the fuzzy controller is to use. A relatively simple rulebase is presented below.

1. If *SinglePieces* is *Many* then output *Weak*
2. If *SinglePieces* is *Few* then output *Good*
3. If *OpponentOnBar* is *Few* then output *Weak*
4. If *OpponentOnBar* is *Many* then output *Good*
5. If *SinglePiecesInHome* is *Many* then output *Weak*
6. If *SinglePiecesInHome* is *Few* then output *Good*
7. if *OpponentsOnBar* is *Many* and *SinglePiecesInHome* is *Many* then output *Weak*

7.1.1 Example

Consider an example where red is about to calculate the board strength for the board illustrated in Figure 68.

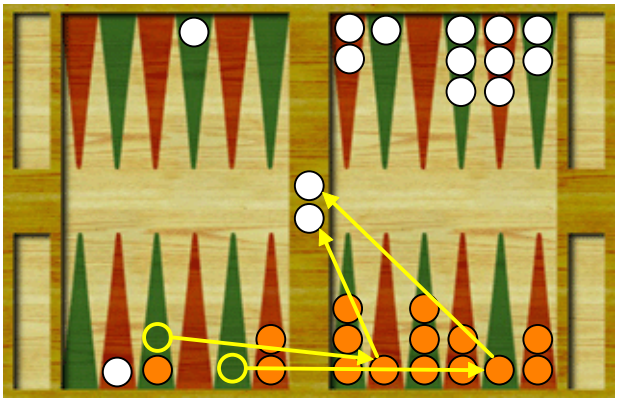

Figure 68: Board fuzzy controller is to calculate board strength for

The first step is to present this board to the fuzzy controllers preprocessor. The preprocessor is then responsible for creating the crisp input to the controller for each of the linguistic input variables considered. The crisp input for each linguistic input variable is presented in Table 26.

Linguistic variable	Crisp value
SinglePieces	3
OpponentOnBar	2
SinglePiecesInHome	2

Table 26: Crisp input for each linguistic variable

Next step is to convert this crisp input to fuzzy membership degrees. This is done for each different combination of linguistic variables and terms. Using the membership functions illustrated in Figure 67 the membership degrees becomes approximately the ones illustrated in Table 27.

Linguistic variable	Linguistic term	Membership
SinglePieces	Few	0.25
	Many	0.077
OpponentOnBar	Few	0.33
	Many	0.071
SinglePiecesInHome	Few	0.5
	Many	0.2

Table 27: Fuzzified membership degrees for each linguistic variable and term pair

The fulfillment or firing strength can now be calculated for each of the seven rules. The resulting fulfillment for each of the rules is illustrated in Table 28.

Rule	Fulfillment
If *SinglePieces* is *Many* then output *Weak*	0.077
If *SinglePieces* is *Few* then output *Good*	0.25
If *OpponentOnBar* is *Few* then output *Weak*	0.33
If *OpponentOnBar* is *Many* then output *Good*	0.071
If *SinglePiecesInHome* is *Many* then output *Weak*	0.2
If *SinglePiecesInHome* is *Few* then output *Good*	0.5
if *OpponentsOnBar* is *Many* and *SinglePiecesInHome* is *Many* then output *Weak*	0.071

Table 28: Fulfillment of the seven rules

The final step before the board strength can be obtained is to defuzzifie the output from each rule into one single value. This can be done using the center of gravity formula previously described in section 2.5.2. The defuzzification is carried out as illustrated below and we obtain a board score of approximately 0.095.

$$\frac{(-1\cdot 0.077)+(1\cdot 0.25)+(-1\cdot 0.33)+(1\cdot 0.071)+(-1\cdot 0.2)+(1\cdot 0.5)+(-1\cdot 0.071)}{0.077+0.25+0.33+0.071+0.2+0.5+0.071}=0.095$$

7.2 Supporting advices taking

One of the strength with fuzzy controllers is that most of their configuration on how to work is contained in a collection of rules, and a collection of membership functions that can be modified by a human control expert. One desirable goal when developing Fuzzeval is to make it completely configurable without having to modify the source code. It should in other words, by use of only text files, be possible to easily add and remove linguistic variables to consider, define associated membership functions and to maintaining the rulebase.

The object is to develop Fuzzeval such a human is able to “explain” how to play backgammon, without having to modify the source code and recompiling it. This is in AI research is refereed to as *learning by taking advice* [17] and is an approach that previously has been tried in different game playing programs. One of the more successful is the chess program NIMZO [7] developed by C. Donninger where a users can express advices using a graphical user interface.

To exemplify this, a human might discover that Fuzzeval seems to have some difficulties with understanding the concept of building and maintaining a consecutive blockade. Building and maintaining blockades is one of the most important properties in backgammon. This is because such blockades are hard to pass by an opponent. A very strong consecutive blockade impossible to pass by white is illustrated in Figure 69.

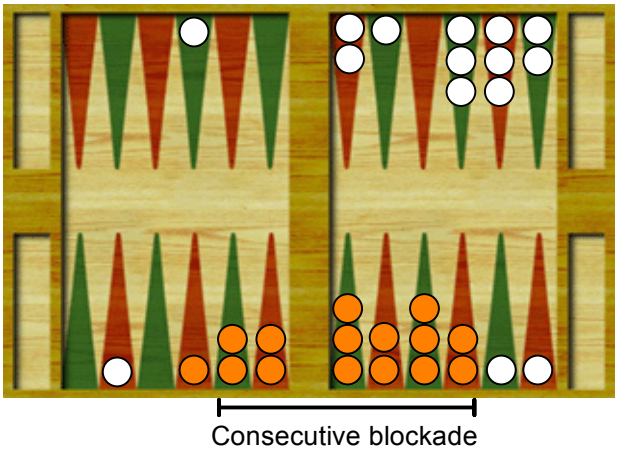

Figure 69: Red player having a strong consecutive blockade

To allow Fuzzeval to understand this concept, one should be able to define a linguistic variable representing the concept of a consecutive blockade. This variable could for example be named *ConsecutiveBlockade.*

One should then be able to define how to preprocess a board and converting it into a crisp value for the input variable *ConsecutiveBlockade*. For the example in Figure 69, the board should be converted to the crisp input value 6 for the variable *ConsecutiveBlockade*. Obtaining this value can simply be done by traversing over the board and then counting the maximum number of consecutive board locations with two or more pieces of its own kind. The challenge will however be to create a script language flexible enough to allow one to explain how to preprocess a board and create the crisp input for all the linguistic variables considered.

Next Fuzzeval should allow one, in a simple manner, to define the associated membership functions. This definition of membership functions can simply be done by defining the left and right base point and the maximum center point for associated triangular membership functions. One could then for example chose to define two associated membership functions named *Long* and *Short*.

Finally one could then add the two following rules to the rulebase.

1. If *ConsecutiveBlockade* is *Short* then output *Weak*
2. If *ConsecutiveBlockade* is *Long* then output *Good*

Where the conclusion terms *Weak* and *Good* and their associated crisp value of course also would have to be defined.

The advice of building and maintaining consecutive blockades has thereby been given to Fuzzeval, and boards with a consecutive blockade should as a result be evaluated higher, thereby changing Fuzzeval's playing style towards one where it has become better at building and maintaining such blockades. At this point the challenge for a human is then to adjust the membership functions to a level where optimal and hopefully improved play is achieved. The membership functions should in other words

be adjusted to a level where they do not have to much influence over the other concepts to evaluate, while still having some influence. A specific goal in this thesis is however to hopefully identify a model that will be able to perform this membership function tuning automatically.

7.3 Initial design overview

This section will present the initial planned design of the Fuzzeval decision module. The design presented here is not so detailed that every class and method that ultimately are to exist is presented. The section will instead on a more overall level describe how Fuzzeval is planned to be implemented.

One of the first challenges to solve is to determine how to let Fuzzeval support advice taking as explained in section 7.2, thereby allowing Fuzzeval to be completely reconfigurable without changing the source code and recompiling it. The challenge is to find a solution such different ways to preprocess a board can be defined and associated to different linguistic input variables. One way to solve this is to develop a script language. This is however a quite cumbersome task and not the focus of this thesis and is therefore not a desirable solution. The whole .Net framework does however ship with the C# compiler, and as the .Net framework is requirement for running the backgammon application, is this compiler also known to be present on systems where the application is to run. A solution is then to contain definitions on how to preprocess a board in a single C# source file. This file is then when Fuzzeval is initialized compiled into a module that is dynamically loaded by Fuzzeval. This process is illustrated in Figure 70.

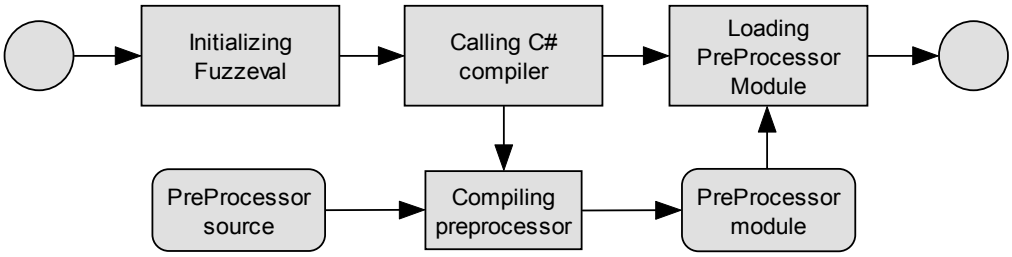

Figure 70: Run time compilation and loading of the Fuzzeval preprocessor

Allowing a human to maintain a collection of membership functions and a rulebase should be easier to support. This is just a question of making Fuzzeval load and save membership functions and rulebase rules from and to text files.

The core in Fuzzeval will be the fuzzy control structure responsible for grading the strength or goodness of different backgammon boards. Additional Fuzzeval should include functionality able to calibrate the fuzzy controller's membership functions and rulebase. As it might be necessary to implement and test a range of different solution models for solving these two issues, is it important that such solution models is easy to implement, maintain and test with minimum impact on the fuzzy controller itself. It is therefore desirable to locate such calibrators outside the fuzzy controller and then let the fuzzy controller implement necessary methods such the calibrators are able to

obtain references to the rulebase and the collection of membership functions. A preliminary UML description of Fuzzeval can be seen in Figure 71.

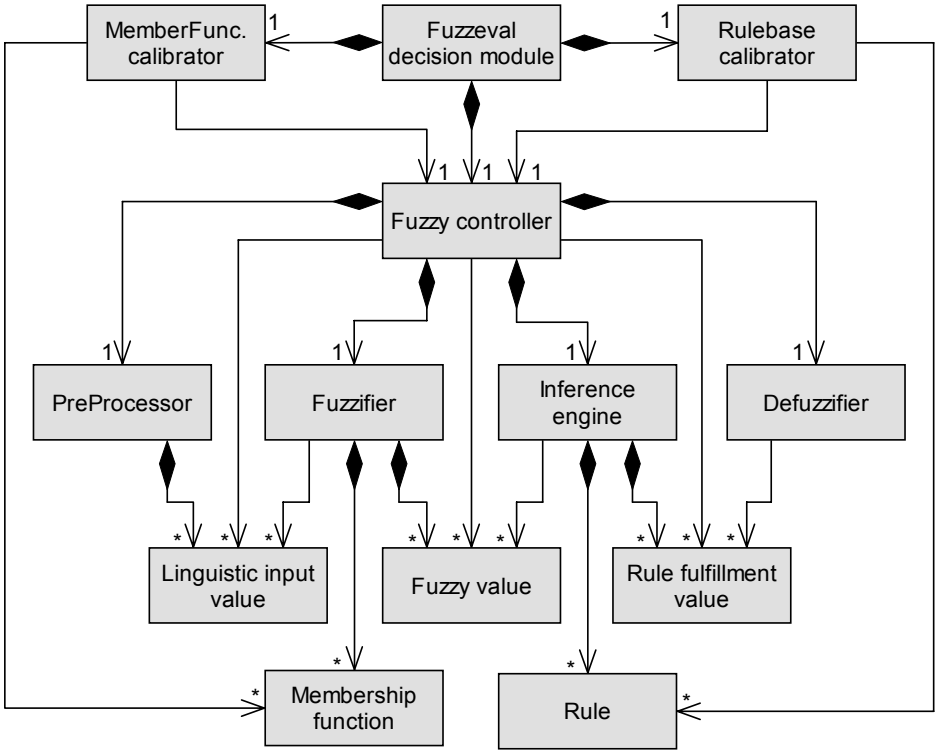

Figure 71: Initial UML description of Fuzzeval

Additional classes might end up exciting in the final implementation of Fuzzeval. Especially the two calibrators might end up consisting of more then just one class each. The diagram should however give a general impression on how Fuzzeval is thought to be designed and implemented. The *PreProcessor* class is the class that is intended to be compiled and dynamically loaded during the initialization of Fuzzeval, while the different membership functions and rules are to be loaded from basic text files. The solution models for making Fuzzeval adaptive by automatically adjusting membership functions and modify the rulebase are then to be located in the calibrator classes outside the Fuzzy controller itself. This is to ensure minimal changes to the actual fuzzy controller when different solution models is implemented and tested.

7.4 Fuzzy controller implementation

The implementation of Fuzzeval as a whole will be an evolving process where ideas continually will be implemented, tested, removed or improved until the deadline for the thesis has been reached. A 100% accurate, in depth and final description of the architecture and implementation of Fuzzeval can therefore as a consequence not be given at this point. The core of Fuzzeval will however be the fuzzy controller responsible for grading the different boards. The implementation of this fuzzy controller has been completed and its architecture will be presented here. It should be noted that small changes and additions to the architecture presented here, at a later point might be done. This could be to either support features required by one of the solution models responsible for calibrate the membership functions or the rulebase, or

simply to add features that in general will improve the performance of the controller. A UML diagram of the current architecture of the fuzzy controller is illustrated in Figure 72.

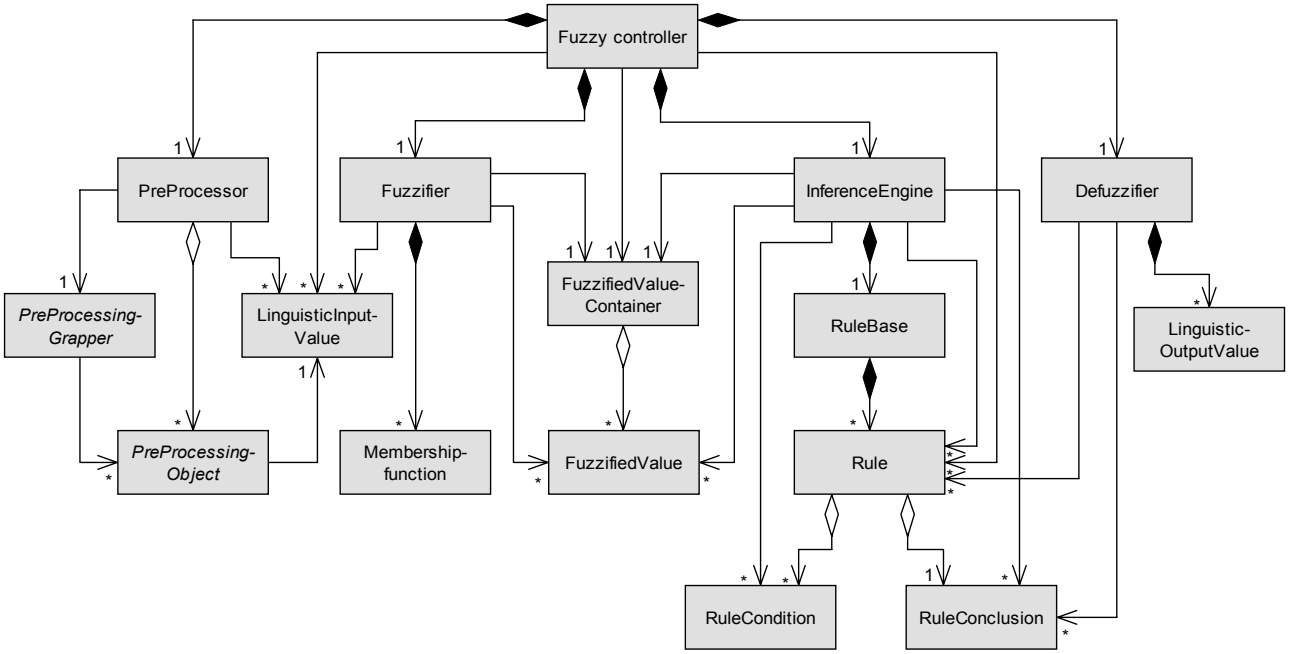

Figure 72: UML of the fuzzy controller contained in the core of Fuzzeval

An architecture that contains quite a lot of classes. Many of the classes are however quite simple and consists only of a value and a name responsible for identifying to which linguistic variable and term the value is associated. An example is *FuzzifiedValue,* for which the whole implementation is illustrated below.

```
class FuzzifiedValue
{
        public string   VariableName;
        public string   TermName;
        public double   FuzzyValue;

        public FuzzifiedValue( string varName, string termName, double fuzzyVal )
        {
                VariableName = varName;
                TermName = termName;
                FuzzyValue = fuzzyVal;
        }
}
```

LinguisticInputValue, *RuleCondition*, *RuleConclusion* and *LinguisticOutputValue* are equally simple although their exact member variables might be slightly different.

FuzzifiedValueContainer, *RuleBase* and *Rule* do not hold much functionality either. They are basically just container classes. *FuzzifiedValueContainer* has been created to improve performance when the *InferenceEngine* is to calculate the fulfillment of

each rule. This is done by wrapping a hash table of *FuzzifiedValues* instead of just passing an array.

PreProcessor, *Fuzzifier*, *InferenceEngine* and *Defuzzifier* are the classes responsible for each of the steps in the fuzzy controller. The *PreProcessor* is responsible for converting a board into a range of crisp linguistic input values. This could for example be the number of single standing pieces. The *Fuzzifier* will then use its membership functions to convert this crisp input into membership degrees for each linguistic input variable and term. These membership degrees are then passed on to the *InferenceEngine* that is responsible for calculating the fulfillment of each rule. The *Defuzzifier* is then as the final step responsible for calculating the defuzzified output from the fuzzy controller using center of gravity.

A thing to note about the fuzzy controller is that the rulebase in its current form not is hierarchical. A hierarchical rulebase is described in section 5.4.2. Changing the fuzzy controller to use a hierarchical rulebase can however be relatively easy done by modifying the *InferenceEngine* such that it aggregates multiple instances of the *RuleBase* class. There is however no reason to experiment with if a hierarchical controller can improve performance until some success has been achieved with a regular rulebase structure.

The goal of implementing the fuzzy controller such that it is completely configurable without having access and modify the source code has also been completed. As a result is Fuzzeval completely supporting advice taking as described in section 7.2. Membership functions are loaded from and saved to a text file. Membership functions are triangular and defined by its left and right base point and its maximum center point. A membership function can additionally be defined such it is unable to be adjust by the function calibrator that is to be developed. This is done by putting a * sign behind the rule. An example of a file defining three membership functions where the second rule is locked such calibration is impossible, can be seen below.

```
OwnSinglePieces Few     0   0    15
OwnSinglePieces Some    0   7.5  15  *
OwnSinglePieces Many    0   15   15
```

Also the rulebase is loaded from, and saved to a regular text file and can be completely edited by a human. An example of a rulebase can be seen below.

```
if OwnSinglePieces is Few then output Good
if OwnSinglePieces is Some then output Fairly
if OwnSinglePieces is Many then output Weak
```

The output terms *Good*, *Fairly* and *Weak* and their associated crisp value must also have been defined for the rules to be valid. This is also done in an ordinary text file that might look like the one below.

```
Good      1
Fairly    0
Weak      -1
```

The challenge was to find out how the preprocessing of a board to a crisp linguistic input values was to be defined. This has been solved by the method presented in section 7.3 where a file is compiled and then dynamically loaded when the fuzzy controller is instantiated. It is however not the whole preprocessor that is compiled. Instead each linguistic input is to inherit from the abstract class *PreProcessingObject* and then implements one method defining how the preprocessing of a board to this linguistic input value is to be done. An example of this file holding a definition for how the crisp input for the linguistic input variable *OwnSinglePieces* is to be obtained is presented below.

```
using System;
using GammonAgent;
using GammonGame;
using Fuzzeval;

public class OwnSinglePieces : PreProcessingObject
{
   public override LinguisticInputValue CreateLinguisticInputValue( BoardRepresentation board )
   {
      LinguisticInputValue result = new LinguisticInputValue( GetName(), 0 );
         for ( int i = 1; i < 25; i++ )
            if ( board.GetPiecesAt( i ) == 1 )
               result.CrispValue++;
      return result;
   }
}

public class Grapper : PreProcessingGrapper
{
   public PreProcessingObject[] GetObjects()
   {
      PreProcessingObject[] result = {
                  new OwnSinglePieces()
            };
      return result;
   }
}
```

The file holds two classes. The first class is named *OwnSinglePieces* and inherits from the abstract class *PreProcessingObject*. *OwnSinglePieces* implements one method that is the one responsible for creating a linguistic input value. The second class *Grapper* inherits from *PreProcessingGrapper* and is the class that insures that the *PreProcessor* class in the fuzzy controller is able to obtain a reference to all the different *PreProcessingObjects* one decides to define in this file.

If one then wants to add a new linguistic variable to consider by the fuzzy controller, one simply creates a new class that inherits from *PreProcessingObject* and then implements the *CreateLinguisticInputValue(…)* method. To enable the preprocessor to obtain a reference to this newly defined *PreProcessingObject,* one also has to add an instance of it to the array returned by the *GetObjects(…)* method in the *Grapper* class.

7.5 Function calibration

The first learning related issue to solve in Fuzzeval is the automatic calibration of the membership functions. The development of a method to solve this specific problem has been done in an iterative way where an initial solution model first has been identified, implemented and tested. This model has then been further improved and tested. The final solution model is however different then these initial solution models. All three models are however presented in this section to give an overview of the whole process of developing the function calibrator model.

Before presenting the different solution models there is however some requirements one was to keep in mind when the solution model is to be developed.

- The adjustment must somehow be based on the outcome of games. A method where the membership functions for example are tuned using genetic algorithm and then locked when playing a human is not acceptable. This is because such a model won't allow Fuzzeval to be adaptive when playing humans.
- It should be able to learn and adjust its membership functions relatively fast. It is not desirable to have a model that requires thousand of games just to make small calibrating adjustments to the membership functions. The adjustment should be done in a rate where an actual change of play might happen based on just one, five or ten games. The lesser games the better.
- If possible it should somehow be a method that should be able to identify a successful style of play and then either improve, or stick to this style. It should in other words not only learn by how the opponent plays but also from itself in the case that it has identified a successful type of play. This can also be expressed as the ability to find and exploit a specific opponents playing style.

7.5.1 Initial solution model

The initial solution model is to some degree inspired by the model used in NEFCON described in section 5.5. In NEFCON a fuzzy error is calculated based on the actual output from the controller and the desired output. Based on this error, an error rate is calculated for each rule. Each membership function connected to a rule is then adjusted in a way that will move the actual output closer to the desired output.

The initial idea in Fuzzeval is, as in NEFCON, to calculate an error rate for a rule. The way to calculate this error, and then afterwards make the adjustment of the involved membership functions, does however differ from the way it is done in NEFCON. Additionally the idea is as in NEFCON to use monotonic membership functions. It should be noted here that the fuzzy controller in the hart of Fuzzeval do support regular triangular membership functions, but that monotonic membership functions can be supported simply by making the center max point identical to either the left or right base point.

Basic idea

The basic idea is then after a game has ended to run all the winning boards through the fuzzy controller and extract an average fulfillment value for each rule in the rulebase. The average fulfillment value for a rule like

If *SinglePieces* is *Few* then output *Good*

could for all the winning boards for example be 0.87. Afterwards all the loosing boards are run through the controller and an average fulfillment value for all the rules are calculated. For the above rule the average fulfillment value for all the loosing boards could for example be 0.69. When comparing the two values is it evident that the most successful player is the one that has been best at fulfilling this rule. As we as humans are able to interpret the rule, can we even conclude that the better player is the one that has been best at avoiding isolated or single standing pieces. The goal is now to adjust the membership functions related to this rule in a way that generally will let this rule be more fulfilled in a subsequent game. As monotonic membership functions are used, is it just a matter of increasing the distance between the base points as illustrated in Figure 73.

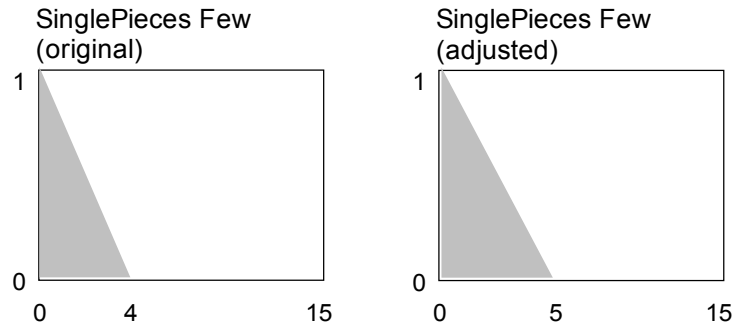

Figure 73: Membership function before and after adjustment

In the opposite case where the average fulfillment value for the winning player instead had been smaller, are the adjustment to be done such the distance between the base points are decreased.

Intuitively this may seems strange as boards with more single standing pieces now suddenly starts to be graded better. But what should be noted is that boards with for example only one single standing piece also will be graded better with the adjusted membership function then before. The aspect of having as few single standing pieces as possible, has in other words been quantified as more important compared to other membership function that does not cover as large an interval.

Calculating adjustment

Before adjusting the membership functions, and error rate is to be calculated for each rule. This error rate is the amount a membership function is to be adjusted. The first step in doing this is simply to calculate the average fulfillment of each of the rules for the winning boards and for the loosing boards. The next step after the average fulfillment of each rule has been calculated is to calculate the error rate for each rule. This error rate for a rule is calculated by the following formula.

$$R_nError = 1 - \frac{Inf(R_navg_{winner}, R_navg_{loser})}{Sup(R_navg_{winner}, R_navg_{loser})}$$

- R_nError is the error rate for a rule
- R_navg_{winner} is the average fulfillment of a rule for the winning boards
- R_navg_{loser} is the average fulfillment of a rule for the loser boards
- *Inf* is short for inferior and means the smallest of the values are used
- *Sup* is short for superior and means the largest of the values are used

The calculated error rate for each rule is then finally used to determine the amount a membership function connected to a rule, is to be adjusted. In the case where the average fulfillment of a rule is largest for the winning boards, is the distance between the base points increased by the error rate. In the opposite case where the average fulfillment of a rule is smallest for the winning boards, is the distance between the base points decreased by the error rate.

Example

Consider the case where the average fulfillment value for the rule

If *SinglePieces* is *Few* then output *Good*

For the winning boards are 0.87 and for the losing boards are 0.69. The associated membership function might look like the one illustrated in Figure 74.

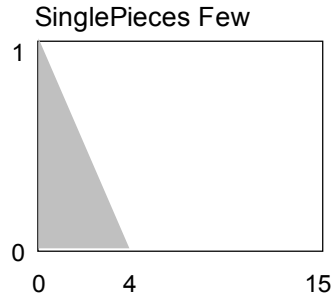

Figure 74: Membership function to be adjusted

The error rate for the rule is then calculated as below.

$$1 - \frac{0.69}{0.87} = 0.207$$

As the average fulfillment value for the rule is highest for the winner, are the associated membership function to be adjusted such the distance between the base points are increased. The adjusted membership function then becomes the one illustrated in Figure 75.

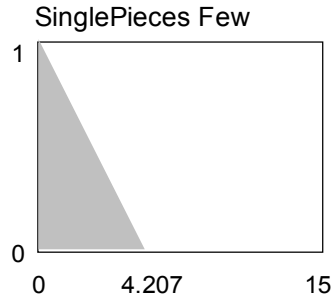

Figure 75: Membership function after adjustment

Impression

Initially the model was merely tested as a brief idea without any hope of any real success. Rather surprisingly the model did however improve the performance of Fuzzeval against Pubeval noticeably compared to the initial manually tuned membership functions. Any real evaluation will however not be given at this point as the linguistic input to consider, and the defined rulebase at this point is on a relatively simple level.

7.5.2 Improved initial solution model

Although the initial solution model did seem to work, ideas for further improvements were still considered. One of the drawbacks with the initial solution model is that membership functions possibly are associated with multiple rules, where a rule then again can be associated to multiple membership functions. A membership function associated with multiple rules will in other words be adjusted multiple times. If some of these rules further more are associated with other membership functions, are all these other membership functions also influencing the adjustment of a membership function. This might introduce some kind of noise that will do that a membership function not is adjusted to the optimal level.

Change

A change where then made to the solution model such the error rate instead were calculated for each membership function, instead of for each rule. The formula for calculating the error are in reality still the same, it is however the average fulfillment values for the membership functions that is now used, and not the average fulfillment of the rules. The modified formula can be seen below.

$$MF_n Error = 1 - \frac{Inf(MF_n avg_{winner}, MF_n avg_{loser})}{Sup(MF_n avg_{winner}, MF_n avg_{loser})}$$

- $MF_n Error$ is the error rate for a membership function
- $MF_n avg_{winner}$ is the average fulfillment of a membership function for the winning boards
- $MF_n avg_{loser}$ is the average fulfillment of a membership function for the loser boards
- *Inf* is short for inferior and means the smallest of the values are used
- *Sup* is short for superior and means the largest of the values are used

The adjustment of a membership function is still done as in the initial solution model. The advantage is however that the error rate is calculated directly on the membership function, and that each membership function now only is adjusted once.

Impression

The impression when testing this improved solution model, compared to the initial solution model, is that the performance of Fuzzeval seems to have improved slightly. This should indicate a better calibration of the membership functions. Again no real evaluation will be given at this point as the linguistic input to consider, as well as the rulebase, still is at a simple level.

7.5.3 Final solution model

The final solution model is quite different from the initial and improved solution models explained in section 7.5.1 and 7.5.2. In the final solution model the idea is simply to calculate the average crisp input to the fuzzy controller for all the winning boards and then adjust all associated membership functions to “center” around this value. This solution model is also no longer limited to only use monotonic membership functions.

A typical linguistic input to consider by the fuzzy controller within Fuzzeval could for example be *OwnSinglePieces.* A crisp input to this value would then typically be something between 0 and 4, where three, four and above would be increasingly rare if the goal is to play the game successfully. A typical average input to this linguistic variable could possibly be around 1.75. The method is then to adjust the membership functions to “center” around the value 1.75 as illustrated in Figure 76.

Figure 76: Example of adjusted membership functions

The adjustment is done such that after each completed game, is the average crisp input to each linguistic variable calculated for the winning boards. The point to adjust on a membership function is then moved slightly closer to the target value matching the average crisp input. For a basic triangular function is the point to be adjusted the center max point, while it for shouldered triangular membership functions are either the left or right base point.

The amount a point to adjust is to be moved closer to a target point, is calculated by the formula below.

$$AmountToAdjust = rate \cdot \Delta x$$

- Δx is the difference between the target point to "center" a membership function around, and the current location of the adjustable point of a membership function.
- *rate* is a learning rate and is at this point set to 0.1.

Example

Let us now consider a case where a game just has ended. The first step is then to calculate the average crisp input to each linguistic variable for the winning boards. For the linguistic input variable *OwnSinglePieces* could this average input for example be 1.5. If the current adjustments of the membership functions then are as illustrated in Figure 76, some adjustment is to be made. The next step is then to calculate the amount the membership functions are to be adjusted. This is simply done as:

$$0.1 \cdot (1.75 - 1.5) = 0.025$$

All the membership functions are then adjusted accordingly, and the new point they centers around is changed from 1.75 to 1.725.

Impression

The impression of this model leaves no doubt that it is performing better than the previous two models explained in section 7.5.1 and 7.5.2. After the implementation of this model some time went into further improving the rulebase of Fuzzeval, and the linguistic input to consider. With this extended, but still rather simple rulebase, has Fuzzeval become able to win an impressive 15% – 20% of the games against Pubeval. Comparing this to btlgammon that according to G. Tesauro in section 6.4 is claimed to win 25%, is this in the author's opinion quite impressive. The actually quite simple rulebase used to obtain this result is presented below.

if OwnSinglePieces is Few then output Good
if OwnSinglePieces is Many then output Weak
if OwnSinglePieces is Some then output Fairly

if OwnSquaresOwned is Few then output Weak
if OwnSquaresOwned is Many then output Good
if OwnSquaresOwned is Some then output Fairly

if OwnStrongestConsecutiveBlockade is Short then output Weak
if OwnStrongestConsecutiveBlockade is Long then output Good
if OwnStrongestConsecutiveBlockade is Moderate then output Fairly

if OwnPiecesNotInHome is Few then output Good
if OwnPiecesNotInHome is Many then output Weak
if OwnPiecesNotInHome is Some then output Fairly

if OwnLastPiecePos is Front then output Good
if OwnLastPiecePos is Back then output Weak
if OwnLastPiecePos is Middle then output Fairly

if OpponentsOnBar is Few then output Weak
if OpponentsOnBar is Many then output Good
if OpponentsOnBar is Some then output Fairly

if OwnBeardOff is Few then output Weak
if OwnBeardOff is Many then output Good
if OwnBeardOff is Some then output Fairly

The requirements given in the start of this section are also fulfilled with this model. The adjustment of the membership functions are done after each game, based on the playing style of the winner. The adaptive behavior that is a requirement to the solution model is therefore obtained. The requirement that Fuzzeval must learn fast is also fulfilled. Only around 75 to 100 games against Pubeval are required before Fuzzeval with initial unadjusted membership functions, starts to win around 15% - 20% of the games. The final requirement that Fuzzeval, if possible, should be able to fulfill, is the exploitation of a specific opponents playing style. This requirement is a little hard to test. The way the model however tries to accomplish this goal, is by adjusting the membership functions according to the playing style of the winner that actually might be Fuzzeval self. Fuzzeval will then try to calibrate the membership functions such the successful playing style just played is more likely to be repeated in a subsequent game.

A more extensive testing and evaluation of Fuzzeval as a whole and thereby also the solution model used for function calibration will be given section 8.1.

7.5.4 Redefined membership functions

Redefined membership functions are not another solution model, but a solution to a problem that was discovered during the development of a solution model able to calibrate the membership functions. A problem with all the presented calibration method arises at the point where a linguistic input variable such as *OpponentHomeBlockade* is defined. This is a reasonable linguistic input to consider and might be used in a rule like.

If *OpponentHomeBlockade* is *Dense* and *OwnSinglePieces* is *Many* then output Weak

This is reasonable rule used to express that if the opponent has a dense blockade in his home, then single pieces is extra bad, because if hit back on the bar, is it difficult to get back on the board. The problem is that *OpponentHomeBlockade* is a linguistic input based on board layout created by the loser, and calibrating a membership function based on play done by the loser is not desirable.

The solution to this problem is refereed to as redefined membership functions. A redefined membership function is a membership function that actually is based on another membership function, and can't be calibrated by the previously described models. It is instead at all times equal to the membership function on which it is based. One can then for example define that *OpponentHomeBlockade* is to be based on another linguistic input such as *OwnHomeBlockade*. If *OwnHomeBlockade* then has two associated membership functions such as *Dense* and *Sparse*, is two associated membership functions named *Dense* and *Sparse* also created for *OpponentHomeBlockade.* As *OwnHomeBlockade*s associated membership functions then continually are calibrated, is *OpponentHomeBlockades* membership functions also continually adjusted to match.

Redefined membership functions are simple to define. It is simply done in a basic text file as in the following example where *OpponentHomeBlockade* is defined to be based on *OwnHomeBlockade*.

```
OpponentHomeBlockade   OwnHomeBlockade
```

7.6 Rulebase learning

With a successful working function calibrator, is the next goal to develop an extension to Fuzzeval that will allow it to modify or learn its own rulebase. An initial developed methods relying on genetic algorithms where first developed. The hope was that the history of a game could be used for the assignment of fitness scores, and then evolves the rulebase for one generation after each game. Results obtained with this method were however of such poor results that a further description not will be given.

The next idea was instead to build what will be refereed to as a condition connection matrix and then use this condition connection matrix to identify rules made up of multiple conditions. A condition connection matrix can be compared to a fuzzy relation that is a way to describe relationship among objects. A fuzzy relation, or from now on refereed to as a condition connection matrix, can be graphically illustrated as in Figure 77.

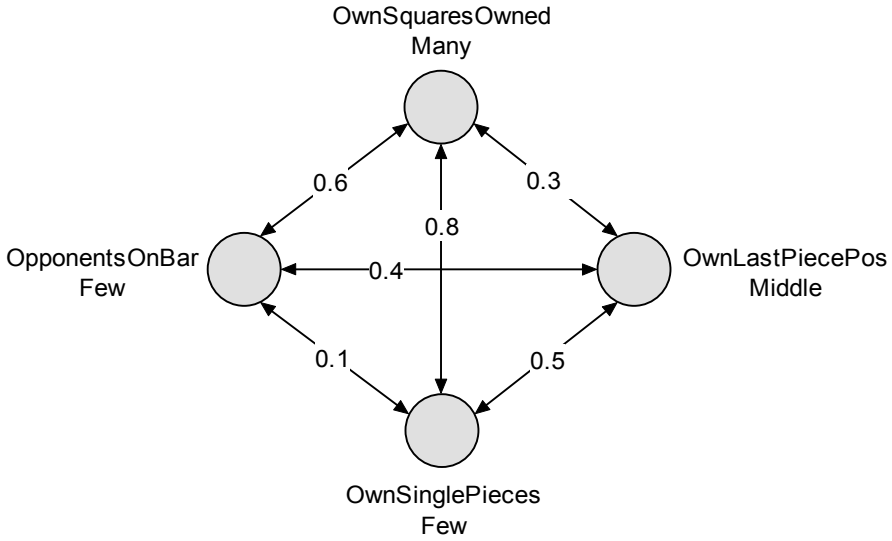

Figure 77: Graphical illustration of condition connection matrix

A condition connection matrix is in other words a collection of the possible conditions, which in reality are all the defined membership functions, that has been defined in Fuzzeval. Every possible condition has a connection of some strength to other conditions. Another way to illustrate this condition connection matrix is in matrix form as in Table 29.

	OpponentsOnBar Few	OwnSquaresOwned Many	OwnSinglePieces Few	OwnLastPiecePos Middle
OpponentsOnBar Few	1	0.6	0.1	0.4
OwnSquaresOwned Many	0.6	1	0.8	0.3
OwnSinglePieces Few	0.1	0.8	1	0.5
OwnLastPiecePos Middle	0.4	0.3	0.5	1

Table 29: Matrix representation of condition connection matrix

The idea is then after a game has ended, to identify conditions with strong connections above some given threshold, and then generate them as possible candidate rules. If this threshold for example is set at 0.75, a candidate condition in the above illustrations would be:

If *OwnSquaresOwned* is *Many* and *OwnSinglePieces* is *Few* then output …

This candidate condition is then used to generate the possible rules made up of this condition. If the defined linguistic output from the controller for example is defined as the two variables *Weak* and *Good,* are the possible rules:

If *OwnSquaresOwned* is *Many* and *OwnSinglePieces* is *Few* then output *Weak*
If *OwnSquaresOwned* is *Many* and *OwnSinglePieces* is *Few* then output *Good*

With two candidate rules identified, is the next step to test whether one of these rules will improve the performance of Fuzzeval. One way to do this would simply be to play a whole range of games where one of these rules have been added to the rulebase and then measure if the performance of Fuzzeval has increased against Pubeval. This is however not a desirable solutions as this wont allow Fuzzeval to learn a good rule against a random opponent in a limited series of games. Even if candidate rules still were to be tested against random opponents, would it not be a desirable solution. First of all is this extensive testing a heavy process, but the continued testing of possible candidate rules would most likely penalize the overall performance of Fuzzeval against for example a human opponent, and that is not desirable. As a consequence a different way of testing whether a possible candidate rule is good is required.

The idea for testing the goodness of a possible identified candidate rule is therefore, after a game has ended, to use the history of the completed game to measure if the candidate rule will improve the average evaluated score for the winning boards more then the average evaluated score for the losing boards. This is refereed to as the improvement rating. To exemplify this consider the case where a game just has ended. All winning boards and all losing boards are then run through the fuzzy controller and an average evaluated score for the winning boards and for the losing boards is obtained. This average evaluated score could for example be 0.2 for the winning boards and -0.1 for the losing boards. The candidate rule is then added to the rulebase of the fuzzy controller and the winning and losing boards are once again run through the fuzzy controller. The obtained average output from the fuzzy controller could then for the winning boards for example have changed to 0.19, while it for the losing boards could have changed to -0.18. The next step is then to measure how much the relative evaluated difference from the losing boards up to the winning boards has increased / decreased. In the case the difference has increased, a possible good rule might have been identified. This is the case in this example where the difference before the rule was added were 0.2 – (-0.1) = 0.3, while it after the rule was add were 0.19 – (-0.18) = 0.37. The improvement rating for that rule can therefore be calculated as 0.37 – 0.3 = 0.07. If this increase is larger then a given threshold, is the rule possibly a good rule.

Building the condition connection matrix

One of the challenges with this solution model is to develop a method that can be used to build the condition connection matrix, such possible candidate conditions and thereby candidate rules can be identified.

The method to build this condition connection matrix is once again to use the history of a game after it has been completed. In this case is it however only the winning boards that is to be used. The idea is then for every winning board to run it through the fuzzy controller and obtain the fulfillment degree for every membership function for every board. The fulfillment degree for every membership function for every board is then to be kept track of.

Let us consider a case where a game just has been completed and where the number of winning boards in the history for this example is just five, and where the two membership functions listed in Table 30 are defined.

OwnSquaresOwned Many
OwnSinglePieces Few

Table 30: Example membership functions

The fulfillment degree for each of the two membership functions for each of the five boards can then for example be as in Table 31.

	Board 1	Board 2	Board 3	Board 4	Board 5
OwnSquaresOwned Many	0.5	0.6	0.9	0.5	0.6
OwnSinglePieces Few	1	0.8	0.6	0.9	0.7

Table 31: Membership function fulfillment degree for each board

From this table can it for example be seen that the third winning board in the history, has a membership degree of 0.7 to the membership function *OwnSquaresOwned Many*.

When all the membership functions membership degrees for all boards has been calculated, is the next step to calculate the connection degree between two membership functions or conditions. The first step in doing this is for every membership function to identify the boards that can be viewed to be exceptions in fulfilling this membership function to the optimal point. These boards are found by calculating the average fulfillment degree for a membership function, and then identify the boards where their fulfillment values are above or below the average. The boards that are to be considered exceptions are the boards where fewest exist above or below the calculated average.

To exemplify this, is the average fulfillment value for *OwnSquaresOwned Many* first calculated to 0.62. Four of the boards have a membership degree below this value while board 3 has a membership degree above this value. The board that can be considered the exception is therefore board number 3. Next the boards to be considered exceptions in fulfilling *OwnSinglePieces Few* are to be found in the same way. This average value is then calculated to 0.8 and the boards that can be considered the exceptions are board 3 and board 5.

The final step before the condition connection matrix can be updated is then to calculate the relationship between *OwnSquaresOwned Many* and *OwnSinglePieces Few.* This relationship is simply calculated by taking the amount of boards considered exceptions that matches for both membership functions and dividing it by the amount of exceptions for the membership function that has most exceptions. In this case is the relationship calculated as 1/2 = 0.5. This is because the membership functions have one board that matches, and the membership function with most exceptions has two exceptions.

The argument for calculating the relationship between membership functions or conditions in this way is that if a membership function suddenly starts to be fulfilled to a degree that is different then what is normal, it might be because some other condition suddenly is fulfilled as well. If two conditions then have identical boards where they are fulfilled to a degree different the normal, there might be a connection between them.

7.6.1 Applying the solution model

First thing to notice is that the condition connection matrix fails to consider rules that are made up of just one condition. Two obvious ways of handling such rules can be considered. One method is to test all possible single condition rules after a game, in the same way as candidate multi condition rules. This is a method that theoretically should allow a rulebase to be learned completely from scratch. This is however a heavy process to do after each game, and a human control expert would have quite easy with simply identifying these rules as the control expert under any circumstances has to define them as linguistic input with associated membership functions. As a result are all single condition rules to be added manually to the rulebase by the human control expert, and the rulebase learner will as a result only consider multi condition rules.

Next step is to decide the life cycle of the condition connection matrix. One possibility is to build it from scratch after each game, identify candidate rules and then discard it. Another, and preferred method is however to preserve it game after game and then update it after each game. This will allow the relationship between conditions to settle on an average value.

Final thing to consider is how and when the rulebase is to be updated with new rules. For maximum adaptability one approach would be after each game to remove every rule with multiple conditions, and then again identify all the candidate rules that seems to increase performance and then add them to the rulebase. A different and preferred approach is however instead to preserve candidate rules in yet another structure with a lifecycle where it is preserved game upon game. This structure is then just like the condition connection matrix updated with candidate rules and their calculated improvement rating. Rules will then also in this structure be able to settle on an average for how good it is measured to be. If a rule and its improvement rating then drop below a certain minimum, is it to be removed from this structure. This structure will then ultimately end up holding the rules that potentially are good in a range of games, and not just only the recent completed game.

7.6.2 Implementation details

The implementation details will not be given such every class and method is explained. The implementation details will instead be given in a step by step fashion where it is explained what happens in the different steps when the solution model is represented within a computer program. A diagram illustrating the steps can be seen in Figure 78.

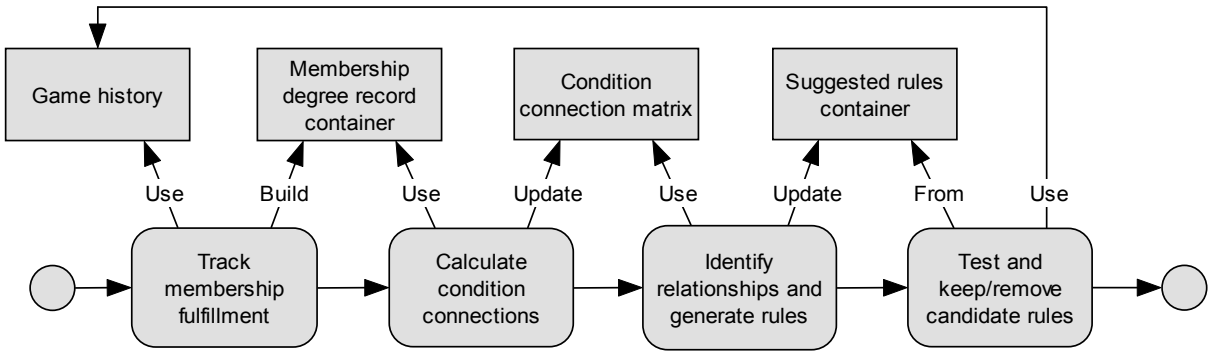

Figure 78: Flow of the process of generating rules

- The first step not having anything to do with the solution model is to complete one game of backgammon. After the completion of this game a method *Learn(…)* is called within the Fuzzeval decision module. This *Learn(…)* method then calls the method on the rulebase learner responsible for running the whole process of the solution model. This method as parameter receives two arrays. One containing all the winning boards and one containing all the losing boards. From that point the whole process of identifying potentially good rules is begun.
- Step one is to track the fulfillment degree for every membership function for every board. This is done using what is called a *MembershipDegreeRecordContainer* that contains *MembershipDegreeRecords*. Every defined membership function in Fuzzeval has a related *MembershipDegreeRecord* that is updated with all the fulfillment degrees. When this step is complete, is it possible to step through every *MembershipDegreeRecord* and extract boards that are considered exceptions in fulfilling a membership function to an optimal degree.
- Step two is to compare all the *MembershipDegreeRecords* and the boards that are considered exceptions. This process calculates the relationship between all membership functions and updates the *ConditionConnectionMatrix* accordingly. The lifecycle of the *ConditionConnectionMatrix* is that it exist game after game, so unless the game is the first completed, is the update just a matter of adjusting all the connection strengths to new calculated average values.
- Step three is to identify conditions in the *ConditionConnectionMatrix* with a relationship above some threshold. Based on these relationships are all potential rules generated.
- Step four is to test all generated rules one by one in the fuzzy controller's rulebase. When a rule then is added to the rulebase, are all winning boards and all losing boards run though the fuzzy controller and the improvement rating are obtained. All rules and their improvement rating are then added to a *SugestedRuleContainer*. The lifecycle of this *SugestedRuleContainer* is that it exist game upon game. If an added rule already exists in the *SugestedRuleContainer,* is the improvement rating calculated to a new average. If it does not exist, is it simply to be added.
- Step five is to remove all rules from the *SugestedRuleContainer* where the improvement rating is below a certain threshold. The rules left are the rules that potentially are identified as good rules, and might be able to increase the performance of Fuzzeval if permanently added to the rulebase.

7.6.3 Impression

Depending on ones own requirements for success the impression may differ. The model is in no way perfect, but it is certainly not bad either. It is evident that a range of rules that do panelize the overall performance of Fuzzeval is generated. It is therefore not desirable to let the model itself be responsible for continually updating the rulebase of Fuzzeval, if the best possible performance is to be achieved at all times. A preferred solution could then instead be to use this model for giving hints to a human control expert, about rules that might be worth to consider.

That being said is it actually able to identify rules that look amazingly sound. Some of the rules identified in an early test runs can be seen below. It should be noted that the solution model in its current form only builds and consider rules made up of two conditions.

if IsContact is False and OwnBeardOff is Many then output Good
if OwnPiecesNotInHome is Few and OwnLastPiecePos is Front then output Good
if OwnPiecesNotInHome is Many and OwnLastPiecePos is Back then output Weak
if OwnStrongestConsecutiveBlockade is Short and OwnBeardOff is Many then output Good
if OwnStrongestConsecutiveBlockade is Short and OwnBeardOff is Few then output Weak

In the case that these rules where extracted from a collection holding a many poor rules, would the result not be that impressive. This is however not the case. The threshold values for both the condition relations and the improvement rating are for this example set relatively high, and the complete amount of suggested rules are in this case only eleven. Exactly if the rules will increase the performance Fuzzeval is not tested at this point. It is however quite hard for a human expert to intuitively discard them as bad without any further testing. Another quite interesting thing to notice about the rules is that rule two and three as well as rule four and five seems to be rule pairs expressing almost the same in opposite ways. A more in depth evaluation of the solution model will be given in section 8.1.

7.7 Conclusion on developing Fuzzeval

Fuzzeval has been implemented and brief initial testing seems promising. Extensive work on making Fuzzeval play at maximum possible performance has however not been done yet.

Three solution models for calibrating the membership functions has been developed and briefly tested. The final of these solution models are able to improve the performance of Fuzzeval quite noticeably, compared to a version of Fuzzeval with manually defined membership functions. Learning or calibration of these membership functions are even done extremely fast and only a limited amount of games seems to be required before optimal play, given a rulebase, is achieved. Exactly how adaptive this calibrating method makes Fuzzeval is on of the things to be tested. The goal is to find a calibrating rate that will give the best possibly performance without making its playing style to deterministic against an opponent.

A completely working method for Fuzzeval to be able to learn or modify its own rulebase for improved performance, without human intervention, does unfortunately not seems to have been identified. Keeping in mind the complexity of this problem, is this not complete letdown. If the solution then instead is able to “hint” possible good multi conditional rules to a human control expert, remains to be tested. An intuitive evaluation of some of the proposed rules, do however indicate that it actually might be able to propose quite promising and clever rules.

An AI related issue referred to as advice taking is also fully supported in Fuzzeval. A human control expert is able, only by the use of text files, to completely configure Fuzzeval. Everything such as linguistic input to consider, membership functions and their definition, the rulebase and linguistic output variables can as a result be defined and used to “explain” how Fuzzeval is to play backgammon, without modifying Fuzzeval itself.

8. Evaluations

This chapter's purpose is simply to evaluate and conclude the whole thesis.

The sections of the chapter are:

- Testing and evaluating Fuzzeval
- Beating Pubeval
- Conclusion

8.1 Testing and evaluating Fuzzeval

This section will describe the test and evaluation of Fuzzeval. This test and evaluation will be performed in three main steps.

In step one the object is do define what linguistic input Fuzzeval should consider, and what associated membership functions this linguistic input should have. Once linguistic input and membership functions have been identified, is a human crafted rulebase to be built. The object is to test how well Fuzzeval, with a human crafted rulebase, and automatically calibrated membership functions will be able to perform against the benchmark player Pubeval.

Step two will then be to test the rulebase learning solution. Is it able to identify rules that look sound, and will these rules eventually be able to improve the performance of Fuzzeval.

Step three will be a more objective evaluation of Fuzzeval where the authors overall opinion and impression is given.

8.1.1 Defining input and building rulebase

First the linguistic input Fuzzeval are to consider is to be identified. These linguistic inputs are parameters a human backgammon expert might consider relevant to evaluate before a move is to be done. Identified linguistic variables are presented in Table 32.

Linguistic variable	Explanation
Contact	An input that is to function as a Boolean value indicating if there still is contact between the two player's pieces.
OwnSinglePieces	The current player amount of single standing or isolated pieces.
OwnSinglePiecesInHome	The current player's amount of single standing or isolated pieces in ones home area.
OwnSquaresOwned	The current player's amount of board locations occupied with two or more pieces.
OwnSquaresOwnedHome	The current player's amount of board locations occupied with two or more pieces in ones home area.
OwnStrongestConsecutiveBlockade	The length of the current player's longest blockade.
OwnPiecesNotInHome	The current player's amount of pieces not moved into ones home area.
OwnLastPiecePos	The board location of the current player's piece farthest behind.
OwnPieceCountAtLastPos	The amount of pieces the current player has located on the location farthest behind.
OpponentsOnBar	The amount of opponent pieces hit back on the bar.
OwnBeardOff	The amount of pieces the current player has removed off the board.
OwnPipAdvantage	The advantage or disadvantage the current player has in distance compared to the opponent, before every piece has been removed from the board.
OpponentSquaresOwnedHome	The opponent player's amount of board locations occupied with two or more pieces in his home area.
OpponentSinglePiecesInHome	The opponent player's amount of single standing or isolated pieces in his home area.
OpponentBeardOff	The amount of pieces the opponent player has removed off the board.

Table 32: Linguistic input considered by Fuzzeval

There is however no guarantee that everything that is worth considering is included in this list; it should however hopefully suffice for giving Fuzzeval a reasonably style of play. If obvious flaws in the playing style of Fuzzeval then later are identified, can such flaws hopefully be a help to identify additional linguistic input to consider.

Next all the membership functions associated to these linguistic variables are to be defined. Regular defined membership functions are illustrated in Table 33.

Linguistic variable	Membership functions	Base range
Contact	True, False	0 – 1
OwnSinglePieces	Few, Some, Many	0 – 15
OwnSinglePiecesInHome	Few, Some, Many	0 – 6
OwnSquaresOwned	Few, Some, Many	0 – 7
OwnSquaresOwnedHome	Few, Some, Many	0 – 6
OwnStrongestConsecutiveBlockade	Short, Moderate, Long	0 – 7
OwnPiecesNotInHome	Few, Some, Many	0 – 15
OwnLastPiecePos	Back, Middle, Front	0 – 25
OwnPieceCountAtLastPos	Few, Some, Many	0 – 15
OpponentsOnBar	Few, Some, Many	0 – 15
OwnBeardOff	Few, Some, Many	0 – 15
OwnPipAdvantage	Low, High	-375 – 375

Table 33: Defined membership functions in Fuzzeval

Cantact and *OwnPipAdvantage* are thought to be locked membership functions, meaning that they can not be adjusted by the membership function calibrator. The point on the x-axis where both membership functions related to *Contact* is to be zero is at 0.5. Exactly at what point on the x-axis the membership functions related to *OwnPipAdvantage* will reach zero, is to be determined when it can be tested where best performance is achieved.

Three of the linguistic input variables do however rely on the redefined membership functions seen in Table 34. Redefined membership functions are explained in section 7.5.4.

Linguistic variable	Redefined based on
OpponentSquaresOwnedHome	OwnSquaresOwnedHome
OpponentSinglePiecesInHome	OwnSinglePiecesInHome
OpponentBeardOff	OwnBeardOff

Table 34: Redefined membership functions in Fuzzeval

Before building a rulebase the final step is to define the linguistic output terms of the controller. The output terms are the possible conclusions of the rules. They can be seen in Table 35.

Linguistic output term	Crisp value
Good	1
Fairly	0
Weak	-1

Table 35: Defined linguistic output terms in Fuzzeval

Now that the linguistic variables and terms have been identified, is the final step to build a rulebase. The building of a rulebase is an iterative process where rules continually are added and removed to achieve the best possible performance. The whole process of adding and removing rules from the rulebase to improve performance will not be described here. It should instead simply be noted that pushing Fuzzeval to a win percentage of around 20% is fairly easy. It then becomes

increasingly hard to identify rules able to further improve the performance of Fuzzeval. One of the great problems is that it then becomes extremely hard to judge if a rule actually does improve performance. In one game session where around thousand games are played against Pubeval, might the win percentage stabilize at a higher win percentage then before the rule was added. While it then in a subsequent game sequence of around thousand games might stabilize at a lower win percentage. This behavior seems in the author's opinion very strange considering the fact that when Pubeval plays itself for thousand games, will the amount of wins quite early begin to lay fairly stable at between 49% and 51% for both sides. Thousand games should in other words be beyond what is required to eliminate the luck factor of the dice.

Nevertheless has one been able to push Fuzzeval to an impressive win of around 34% to 35% against Pubeval. At occasions will the win even be as high as 38% to 39%, while it in other occasions may be as low as 30% to 31%. Without actually knowing what courses this behavior, could a fair guess be that the more games Fuzzeval by some chance win early in the session, the more influence will the playing style of Fuzzeval in these early games, have on the calibration of its own membership functions. If these early game wins then are coursed by a playing style Pubeval is having a hard time handling, will Fuzzeval continue to win more games, and as a consequence will Fuzzeval own playing style continue to have more impact on the function calibration. If that actually is the case is of course unknown and hard to test. The rulebase designed to achieve these results are presented below.

if OwnSinglePieces is Few and Contact is True then output Good
if OwnSinglePieces is Some and Contact is True then output Fairly
if OwnSinglePieces is Many and Contact is True then output Weak

if OwnSinglePiecesInHome is Few and Contact is True then output Good
if OwnSinglePiecesInHome is Some and Contact is True then output Fairly
if OwnSinglePiecesInHome is Many and Contact is True then output Weak

if OwnSquaresOwned is Few and Contact is True then output Weak
if OwnSquaresOwned is Some and Contact is True then output Fairly
if OwnSquaresOwned is Many and Contact is True then output Good

if OwnSquaresOwnedHome is Few and Contact is True then output Weak
if OwnSquaresOwnedHome is Some and Contact is True then output Fairly
if OwnSquaresOwnedHome is Many and Contact is True then output Good

if OwnStrongestConsecutiveBlockade is Short and Contact is True then output Weak
if OwnStrongestConsecutiveBlockade is Moderate and Contact is True then output Fairly
if OwnStrongestConsecutiveBlockade is Long and Contact is True then output Good

if OwnPiecesNotInHome is Few then output Good
if OwnPiecesNotInHome is Some then output Fairly
if OwnPiecesNotInHome is Many then output Weak

if OwnLastPiecePos is Front then output Good
if OwnLastPiecePos is Middle then output Fairly
if OwnLastPiecePos is Back then output Weak

if OwnPieceCountAtLastPos is Few and Contact is True then output Good
if OwnPieceCountAtLastPos is Some and Contact is True then output Fairly
if OwnPieceCountAtLastPos is Many and Contact is True then output Weak

if OpponentsOnBar is Few then output Weak
if OpponentsOnBar is Some then output Fairly
if OpponentsOnBar is Many then output Good

if OwnBeardOff is Few then output Weak
if OwnBeardOff is Some then output Fairly
if OwnBeardOff is Many then output Good

if OwnBeardOff is Few and Contact is False then output Weak
if OwnBeardOff is Many and Contact is False then output Good

if OwnPipAdvantage is Low then output Weak
if OwnPipAdvantage is High then output Good

if OpponentSquaresOwnedHome is Many and OwnSinglePieces is Many and Contact is True then output Weak

Given more time might it even be possible to achieve even better performance. The time needed to refine the rulebase for improving results, is however increasing rapidly and given the current tradeoff between the time that have been used for building the rulebase, and the results achieved, does this seem as a good place to stop refining and move on.

8.1.2 Testing and evaluating the rulebase learning model

With a version of Fuzzeval that is able to play backgammon using a human crafted rulebase, is the next step to test if the rulebase learner is able to suggest rules actually able to improve the performance of Fuzzeval.

Stableness

The first thing to test is simply if it is stable in suggesting the same rules game upon game, or if the suggested rules will change dramatically from one game to a subsequent game. To test this we set the threshold values for the condition connections and the improvement rating relatively high. A couple of games will then be played once at a time, and the suggested rules will then be compared from one completed game to another. To get as accurate result as possible is this test started with calibrated membership functions. The opponent is once again Pubeval.

The first sample game used for this example is won as a regular victory by Pubeval, and the following six rules are suggested as new candidate rules.

if OwnPieceCountAtLastPos is Some and OwnBeardOff is Some then output Good
if OwnSquaresOwned is Many and OwnBeardOff is Few then output Good
if OwnPieceCountAtLastPos is Few and OwnBeardOff is Some then output Good
if OpponentsOnBar is Many and OwnBeardOff is Few then output Good
if OwnSquaresOwned is Many and OpponentsOnBar is Many then output Good
if OpponentSquaresOwnedHome is Many and OpponentBeardOff is Few then output Weak

We will not comment on the intuitive goodness of these rules put simply use them for comparison with suggested rules in a subsequent games. The subsequent game in this experiment is this time won by Fuzzeval and the suggested rules are after this game the following seven rules.

if OwnPieceCountAtLastPos is Some and OwnBeardOff is Some then output Good
if OwnSquaresOwned is Many and OwnBeardOff is Few then output Good
if OwnPieceCountAtLastPos is Few and OwnBeardOff is Some then output Good
if OwnPiecesNotInHome is Many and OwnLastPiecePos is Back then output Weak
if Contact is False and OwnBeardOff is Many then output Good
if OwnSquaresOwned is Many and OpponentsOnBar is Many then output Good
if OpponentSquaresOwnedHome is Many and OpponentBeardOff is Few then output Weak

The rules 1, 2, 3, 6 and 7 are survivors while one rule has been removed and replaced with two other rules.

Game three is in a very close match won by Pubeval and the suggested rules are now the six rules listed below.

if OwnPieceCountAtLastPos is Some and OwnBeardOff is Some then output Good
if OwnSquaresOwned is Many and OwnBeardOff is Few then output Good
if Contact is True and OwnBeardOff is Many then output Weak
if Contact is False and OwnBeardOff is Many then output Good
if OwnPieceCountAtLastPos is Few and OwnBeardOff is Some then output Good
if OpponentSquaresOwnedHome is Many and OpponentBeardOff is Few then output Weak

Of these suggested rules are 1, 2, 5 and 6 still survivors from the first game, while rule 4 has survived from the second. Rule number 3 is however new.

We now play seven games in a row with out keeping track of the suggested rules until the completion of game tenth. At that point has the suggested rules become the ones listed below.

if OwnPieceCountAtLastPos is Some and OwnBeardOff is Some then output Good
if OwnSquaresOwnedHome is Few and OwnStrongestConsecutiveBlockade is Short then output Good
if OwnPieceCountAtLastPos is Few and OwnBeardOff is Some then output Good
if Contact is False and OwnBeardOff is Many then output Good
if OwnSquaresOwned is Many and OwnBeardOff is Few then output Good

Surprisingly enough are rule 1 and 3 still survivors from game one, while rule 4 has survived from game two.

We now play ninety games so the total amount of games becomes one hundred. The only suggested rules are now the two listed below.

if OwnSquaresOwned is Few and OwnBeardOff is Many then output Good
if Contact is False and OwnBeardOff is Many then output Good

Both rules are new and have not previously been suggested. It is however worth noticing that rule number 2 actually is an exact copy of a rule already existing in the rulebase. No check is done to see if suggested rules already exist. This rule is in other words in the authors opinion one that with out a doubt can be considered highly relevant, as it previously has been manually added.

But what now happens if a large range of additional games are played, is that two rules continually are suggested. Playing even more games does not seem to change that. The two rules are listed below.

if Contact is True and OwnBeardOff is Many then output Good
if Contact is False and OwnBeardOff is Many then output Good

The continued suggestion of these two rules might however be rooted in the fact that they both involves the condition *OwnBeardOff Many,* which always, without exceptions, will be more fulfilled for the winner then the loser, and as a consequence will the improvement rating for this rule always be positive. This however still requires that the condition connection matrix must keep a relatively strong connection between the conditions.

What can be concluded by this little test is that the model does not seem to be able to identify rules that most definitely will improve play in any circumstances over a longer series of games. Instead the rules seem to depend a great deal on the specific flow of games. This do however indicates a model that in the sort run is highly adaptive and is taking recent games into account. But the test also shows that the model is not so adaptive that it is suggesting completely new rules from one game to another in a way

that looks to be mostly random. This stableness is however not of such kind that Fuzzeval can be set to play thousand of games, and left are five perfect rules ready to be added to the rulebase. A human expert must instead at regular intervals check for rules that intuitively looks good and might be able to improve performance overall. This is an important property to keep in mind when potentially good rules are to be extracted among suggested rules. Lowering the threshold values might however do that more rules are surviving game upon game.

Testing rules

What now becomes interesting is to see if any of the suggested rules then actually are able to improve the performance of Fuzzeval. Some natural rules to start out with are the rules listed below.

if OwnPieceCountAtLastPos is Some and OwnBeardOff is Some then output Good
if OwnPieceCountAtLastPos is Few and OwnBeardOff is Some then output Good
if Contact is False and OwnBeardOff is Many then output Good

if OwnSquaresOwned is Few and OwnBeardOff is Many then output Good
if Contact is False and OwnBeardOff is Many then output Good

if Contact is True and OwnBeardOff is Many then output Good

This is some of the rules suggested in the previous test. The rules are the ones suggested at game seven that has survived from game one and two, the rules that where suggested at game number one hundred and the additional rule that continually are suggested now matter how many games that seems to be played. It is not that these rules intuitively look especially good, but they do not look bad either.

Adding all rules at once to the rulebase, did however not have much influence on Fuzzeval's performance against Pubeval. This first impression was in fact very positive, as it was a fear that performance would suffer. Next step was then to remove one rule at a time. This was to identify rules that influenced Fuzzeval in a negative direction. The hope was that the unchanged performance was a consequence of new rules that evened each other out. As hoped, but really not really believed, did the striking result appear at the removal of the last rule. With the removal of this rule does it undoubtedly seem like Fuzzeval has increased in performance with about 2% against Pubeval. 2% might not sound as much but considering the trouble one had at pushing Fuzzeval from a winning around 30% to 35%, is this in fact considered a major achievement. One can simply not deny the fact that Fuzzeval now seems to win around 36% to 37% of the games. It evens occasionally wins around 40%, while only winning in the lower half of 30%, seems to have become slightly more rare.

With such a result is the next step to test if even more rules that improve performance can be identified and added to the rulebase with success. This test has unfortunately not turned out any significant results, and further improvement of Fuzzeval has as consequence unfortunately not been achieved.

Learning entire rulebase

The final and ultimate test how well the model will perform when a rulebase has to be learned completely from scratch. This test is done by starting with a completely empty rulebase. Ten games are then played against Pubeval and all suggested rules are then simply added to the rulebase without any form of human influence. With these rules added are ten more games played and the rules then suggested are also added. This process is continued until no more noticeably increase in performance is observed, or no more rules are suggested. While ten games are used for identifying new rules, are several hundred games played to determine if performance has improved. This test is then to indicate how well the rulebase learner, without human intervention actually is. It should be noted that the model of course will suffer from the fact that it in its current form only can generate and consider rules with two conditions. It should however still give an indication of how good the model seems to perform as plenty of good, as well as bad, rules can be generated with two conditions.

First must it be determined how well Fuzzeval performs without any rulebase at all, and that is of course as expected not especially good, but it still in some strange way manage to win about 2% of the games. Ten games are then played and twenty six suggested rules are added to the rulebase. Testing leaves no doubt that the performance of Fuzzeval with these rules added, has increased as it now wins around 7% of the games. This is however still quite low, and a little disappointing. Ten games are then again played and this time are twenty four suggested rules added to the rulebase. Adding these rules does unfortunately not increase performance and Fuzzeval now only wins 5% of the games. Ten more games are now played and five suggested rules are generated. Adding these rules to the rulebase, remarkably enough does that Fuzzeval now wins 9% of the games. Playing ten more games now only result in 3 suggested rules. Adding these three rules, does however has the devastating effect of dropping Fuzzeval down to a catastrophic 1% wins.

It is evident the rulebase learner on its own isn't able to produce a rulebase of a quality even remotely close of what a human expert is able to. Still (if ignoring the last three rules added), is the test indicating a model that is able to generate rules of at least some quality. That is after all also an achievement. It should after all be remembered that the total amount of bad rules in the domain of backgammon, most definitely is of a much higher count then the amount of good rules. And still, in this jungle of mostly bad rules, being able to suggest rules that overall actually mostly improve performance, must after all be considered to be a success to some degree. This can however not deny the fact that the solution model on its own is unable to contribute positive to the goal of creating the strongest possible backgammon player. It might however be a solution model that might be worth to study more, possibly in a less complex domain then backgammon.

8.1.3 Objective evaluation of Fuzzeval

The object of creating a solution model for generating fuzzy control rules for the game of backgammon has unfortunately not been as successful as hoped. This does however not in any way mean that the object of this thesis is unfulfilled as the more overall main goal of creating a capable and adaptive backgammon agent by fuzzy control in the author's opinions has been reached. That Fuzzeval not is as strong or

better then Pubeval is by no means considered to be a failure. A final win percentage against Pubeval of 36% – 37% is in the author's opinion pretty good compared to the UNIX program btlgammon that only is able to win around 25%. It is also worth noticing that the SUN program gammontool that wins around 43% of the games against Pubeval was considered to be among the stronger Backgammon programs available before G. Tesauro created TD-Gammon and Pubeval. It is also worth remembering that Fuzzeval's playing strength has been reached without the help of other methods such as an opening book and a bear off database or rote learning mechanisms where generalized and typical board patterns are stored in a database.

When simply observing the playing style of Fuzzeval is it also evident that it plays reasonable rational and is able to give humans with even some backgammon skill resistance. It keeps a good balance between playing safe and hitting the opponent. It also understands the importance of building consecutive blockades as well as building blockades within ones own home board. Fuzzeval does however have one obvious weakness, and that is that it fails to understand the concept of probability of being hit when it has to choose between different pieces to isolate as plots (single standing pieces). Fuzzeval is simply unaware of the fact that an isolated piece in front of a collection of opponent pieces is more devastating then an isolated piece located at the back of a relatively long blockade.

The object of making Fuzzeval adaptive has also been reached. The probably best way to observe this is in its opening play. One example is when Fuzzeval must make an opening move on the dice roll 5 and 2. In this case might it some times play 13/11 11/6, while it in other cases might play 13/8 8/6. Other cases are on the roll 5 and 1 where the moves have been observed to alternate between 24/23 13/8 and 24/23 23/18, as well as a dice roll of 4 and 3 where different moves such as 13/10 10/6 and 13/9 9/6 has been played. But what also is worth noticing is that it is not so adaptive that it's suddenly decides to play completely irrational in a subsequent game. Its adaptive behavior is still limited to alternating between rational playing styles. When entering the middle game and the boards become more complicated should this adaptive behavior be even more apparent, as more factors then starts to influence the evaluation of boards.

Fuzzeval is also able to learn and calibrate its membership functions extremely fast. No more then around 50 to 100 games are required before Fuzzeval with initially unadjusted membership functions is able to play close to its maximum level. This adjustment can even be done when Fuzzeval play itself, it does in other words not needs a master teacher.

To summarize then Fuzzeval is not the world's strongest computer backgammon player, but it not weak either. It is a player that seems able to match other backgammon programs in playing strength. Fuzzeval does further more have some advantages such as being adaptive as well as allowing people to manually configure the playing style of Fuzzeval. This might be a property some potential users might find appealing. The modification of Fuzzeval is simply done by modifying the rulebase, adding linguistic variables and terms, etc. This is all relatively easy done as the hardest step in this process, which is to quantify a new property according to existing

properties, is done automatically by Fuzzeval using its membership function calibration mechanism.

8.2 Beating Pubeval

Even though Fuzzeval in the author's opinion has reached a playing strength that is very satisfying, can it not be denied that creating a backgammon player able to beat Pubeval is something that is highly desirable. It is also quite possible that the performance of Fuzzeval can be increased further by putting more work into it. This could for example be by further refining linguistic input to consider as well as the rulebase. It could also be that hierarchical fuzzy controllers described in section 5.4.2 would be able to improve the performance. The test of hierarchical fuzzy controllers has however not been possible due to time concerns. Exactly how much of a performance increase one is able to achieve by these methods can however be questioned, and it is honestly not truly believed that these methods is able to increase the playing strength to the level needed to beat Pubeval.

If beginning to relying on methods not especially related to Fuzzeval, could another solution be to include some form of probability based search into Fuzzeval. So instead of simply evaluating the possible boards in a given situation, would Fuzzeval instead look one or two moves ahead and then somehow combine the evaluated score of these further boards with some form of probability of the dices. This is a method that has been used in TD-Gammon and other of the strongest backgammon programs with success. Yet another approach could be to extend Fuzzeval with some rote learning scheme where a database are used to remember important board patterns in some generalized form.

This is all valid solutions but another approach in beating Pubeval is however taken. The approach is simply to copy the method G. Tesauro used when he created TD-Gammon, and then implement this method in a new decision module. Not only would this hopefully allow one to achieve the desired goal of beating Pubeval, but it would also allow one to get a first hand impression of the method such as the training time needed, adaptability and a general overview on how hard it is to get working.

Even though this experiment might sound as a cumbersome task to perform, is it actually not the case. This is because the implementation of a neural network trainable by the back-propagation training algorithm has been done during the first part of this thesis. The actual training of the neural networks can additionally simply be done over a period of time, without any form of human intervention.

8.2.1 Setting up the neural networks

G. Tesauro used two neural networks in TD-Gammon, one for the phase of the game where there still is contact between the two players, and one for the phase known as the race. Two networks are therefore also used in this experiment. Both networks are made up of an input layer of 196 units, a hidden layer of 80 units and an output layer of 5 units. The selection of 80 units in the hidden layer is the same amount as G. Tesauro used in some of the middle versions of TD-Gammon. Some earlier versions used 40 hidden units while later versions uses up to 160 hidden units. 80 have been

selected as it is hoped that this gives the best tradeoff between training time and strong play. More hidden units can however according to G. Tesauro ultimately create a better player, the training time needed to reach equal playing strength is however greater with more hidden units.

The coding scheme for a board at the input layer is as follows. Each location on the backgammon board has 8 associated input units. 4 of the units indicate the number of the current player's pieces on that location, while the other 4 indicates the number of opponent pieces on the location. If there are no pieces on the location then all input units takes the value 0. If there is one piece, then the first unit takes the value 1. If there are two pieces, then both the first and the second unit take the value 1. If there are three or more pieces on the location, then all of the first three units take the value 1. If there are more than three pieces, the fourth unit takes the number of additional pieces beyond three divided by two. With four units for both sides, is the total amount of units 192. Two additional units then take the number of pieces on the bar divided by two, while two other units takes the number of pieces removed from the board divided by 15, making the total amount of input units 196. It should be noted the G. Tesauro used two additional units that in a binary fashion indicated whether it was white or black to move.

The coding scheme at the output layer is the estimated probabilities for winning or losing with one of the possible outcomes. The coding scheme at the output layer can be illustrated as (P_1, P_2, P_3, P_4, P_5) where P_1 = regular win, P_2 = gammon win, P_3 = backgammon win, P_4 = gammon loss, P_5 = backgammon loss. The estimated score or grade for a board is then calculated as $P_1 + 2P_2 + 3P_3 - 2P_4 - 3P_5$. This is also a little different then done by G. Tesauro, as he used four output units and did not include backgammon outcomes.

Training is done by self play in the same manner as done with TD-Gammon. Opposed to TD-Gammon where training was done at each move, is training of the networks in this experiment done after the completion of each game. Adjustment is first done for the final winning board, then the final losing board, then second last winning board and second last losing board and etc until the first played board is adjusted as the last one. Adjustment of the networks is done as a combination of the standard back propagation algorithm explained in section 2.7.4 and temporal difference explained in section 4.4. Such a combination makes temporal difference responsible for calculating the desired target output for a board pattern, before the back propagation algorithm then makes the actual adjustment of the network weights such actual and desired output matches. In the final experiment is the step-size parameter in the temporal difference calculation set to 0.1 and the networks are during training adjusted to a maximum error of between 0.0035 and -0.0035 from the desired target output.

It should be noted that some early experiments where run with higher values for the training precision error and step-size parameter. This is because higher values will initially increase speed and learning early one, while sacrificing later possible playing strength. These experiments where also run such that Pubeval was playing as opponent instead of letting the networks play them self. These early experiments also only used 40 hidden units instead of the final 80. These experiments where used to

test the networks and training methods for bugs and errors. When confident that everything worked, were the networks set to play 300.000 games. This amount of games has been selected as this is equal to the lowest number of games played in experiments where G. Tesauro has published successful results. Other and improved results have been published with networks of different sizes trained for 800.000 and 1.500.000 games.

8.2.2 Results

Playing 300.000 games did take a little less then 3 whole days and nights of almost constant play. Even though this might be considered to be quite some time, is it less then initially feared.

The first test was done already around 10.000 trained games. At that time was it evident that the TD trained networks had learned basic strategies such as hitting the opponent and playing safe. In a test against Pubeval were the networks rather surprisingly already able to achieve a win of almost 20%. Next test was done after around 75.000 training games and the TD trained networks where now able to achieve a win of almost 40%. At 150.000 played training games, was the goal of beating Pubeval almost reached, as the TD trained networks now was playing almost even with Pubeval. It did however still seemed as if Pubeval was having the edge and playing slightly better. It was evident that the improvement in play from now on was happening slowly, as any increase in playing strength was hard to observe at 160.000 training games. At 250.000 training game, was the goal of beating Pubeval however reached. The TD trained networks was now beating Pubeval with a win of around 57%. Training for further 50.000 games and thereby making the total amount of training games 300.000, improved this by roughly 1%, making the TD trained networks able to achieve a win rating around 58%.

It must be admitted that the above results leaves one with some mixed feelings. In one way is one glad that the goal of beating Pubeval has been achieved, and that one is, given enough time, able to create a backgammon program able to compete against the best. It is however also somewhat of a letdown that the method in raw playing strength seems to be so far superior to the personal developed solution model used in Fuzzeval, and that the model actually is rather simple to get working.

8.3 Conclusion

This second part on a master thesis on intelligent game agents had the objective of showing if fuzzy controllers could be a usable solution when creating adaptive and intelligent game agents. The objective of creating a backgammon application implementing an agent based on fuzzy logic has been fulfilled. The fuzzy controlled agent named Fuzzeval has a reasonable style of play and seems able to play at the same level as other average backgammon applications. It is however far from being able to compete against the best.

The issues related to Fuzzeval, was to make it adaptive and to give it shifting style of play. This goal has been achieved by continually adjusting the membership functions. This continued adjustment of the membership functions further more makes it much

easier for a human expert to define what Fuzzeval should consider when playing, as the step of deciding how to quantifying a new concept to consider, can be skipped.

An additional strength with Fuzzeval is that this definition of what Fuzzeval should consider can be done without having to modify and recompiling the source code. Fuzzeval does in other words completely support what is referred to as advice taking, making one able to “explain” Fuzzeval how to play backgammon.

A solution model for Fuzzeval to be able to modify its own rulebase has also been developed. This model has however not turned out to be as successful as hoped, even though the model has been able to propose a few of rules able to improve performance. Test has also been done to see if the model is able to generate a decent rulebase from scratch. This test indicates that the model is able to generate a rulebase able to improve performance. The performance of the generated rulebase is however far from a human crafted rulebase. If putting more time into the solution model and applying it to a less complicated domain, might it not be impossible that better results can be achieved.

The developed backgammon application as a whole has also worked flawlessly as both test domain for Fuzzeval and backgammon program in general. Continued testing of Fuzzeval has been done effective and precise due to the applications ability to run a large amount of games between two agents and their decisions modules fast and automatic. The use of decision modules has further more made the application very flexible such different board evaluators easily can be removed and added. The final creation and training of a decision module based on the same principles as TD-Gammon has further more done that the backgammon application offers an opponent playing at a strong level, able to give even experienced players a struggle. This opponent does however not include any form of visible adaptive behavior.

9. References

Books

[1]. A. Konar, “Artificial Intelligence and Soft Computing: Behavioural and Cognitive Modelling of the Human Brain,” ISBN 0-8493-1385-6, 1999

[2]. B. Dahlbom, L. Mathiassen, “Computers in Context: The Philosophy and Practice of Systems Design,” ISBN 1-55786-405-5

[3]. S. Russell, P. Norvig, “Artificial intelligence: a modern approach,” ISBN 0-13-103805-2, 1995

Papers

[4]. A. L. Samuel, “Some studies in machine learning using the game of checkers,” *IBM Journal of research and development* **44**(1/2): 206 – 226, 2000

[5]. A. M. Turing, “Intelligence machinery,” *Machine intelligence* **5**: 2 – 23, 1950

[6]. B. D'Ambrosio, “Inference in Bayesian Networks,” *AI Magazine* **20**(2): 21 – 36, 1999

[7]. C. Donninger, “A graphical language for expressing chess knowledge,” *International Computer Chess Association Journal* **19**(4): 234 – 241, 1996

[8]. D. B. Fogel, “Evolutionary Entertainment with Intelligent Agents,” *IEEE Computer* **36**(6): 94 – 96, 2003

[9]. D. B. Fogel, “Evolving a checkers player without relying on human experience,” *Intelligence* **11**(2): 20 – 27, 2000

[10]. D. Moriarty, R Miikkulainen, “Evolving neural networks to focus minimax search,” *Proceedings of 12th national conference on artificial intelligence* 1371 – 1377, 1994

[11]. D. Nauck, F. Klawonn, R. Kruse, “Combining neural networks and fuzzy controllers,” *Fuzzy Logic in Artificial Intelligence* 35-46, 1993

[12]. D. Nauck, R. Kruse, “A fuzzy neural network learning fuzzy control rules and membership functions by fuzzy error back propagation,” *IEEE International Conference on Neural Networks ICNN'93* 1022 – 1027, 1993

[13]. D. Nauck, R. Kruse, “A neural fuzzy controller learning by fuzzy error propagation,” *North American Fuzzy Information Processing NAFIP’92* 388 – 397, 1992.

[14]. F. Hoffmann, G. Pfister, “A New Learning Method for the Design of Hierarchical Fuzzy Controllers Using Messy Genetic Algorithms,” *In Proceedings Sixth International Fuzzy Systems Association World Congress* **1**: 249 – 252, 1995

[15]. G. Tesauro, “Temporal Difference Learning and TD-Gammon,” *Communications of the ACM* **38**(3): 58 – 68, 1995

[16]. G. Tesauro, T. J. Sejnowski, “A parallel network that learns to play backgammon,” *Artificial intelligence* **39**(3): 357 – 390, 1989

[17]. J. Fürnkranz, “Machine Learning in Games: A Survey,” *Machines that learn to play games*: 11 – 59, 2001

[18]. J. Schaeffer, R. Lake, P. Lu, M. Bryant, ” Chinook: The world man-machine checkers champion,” *AI Magazine* **17**(1): 21 – 29, 1996

[19]. K. Chellapilla, D. B. Fogel, “Evolving neural networks to play checkers without relying on expert knowledge,” *IEEE Transaction on Neural Networks* **10**(6): 1382 – 1391, 1999

[20]. M. Buro, “Towards opening book learning,” *International computer chess association journal* **22**(2): 98 – 102, 1999

[21]. M. Campbell, “Knowledge Discovery in Deep Blue,” *Communications of the ACM* **42**(11): 65 – 67, 1999.

[22]. M. Wooldridge, N. R. Jennings, “Intelligent Agents: Theory and Practice,” *The Knowledge Engineering Review* **10**(2): 115 – 152, 1995

[23]. Q. Lu, Z. Peng, F. Chu, J. Huang, “Design of Fuzzy Controller for Smart Structures Using Genetic Algorithms,**"** *Smart Materials and Structures* **12**: 979 – 986, 2003

[24]. R. M. Hyatt, “Book learning – a methodology to tune an opening book automatically,” *International computer chess association journal* **22**(1): 3 – 12, 1999

[25]. S. L. Epstein, “Learning to play expertly: A tutorial on Hoyle,” *Machines that Learn to Play Games* 153 – 178, (2001)

[26]. Y. Seirawan, H. A. Simon, T. Munakata, “The implications of Kasparov vs. Deep Blue,” *Communications of the ACM* **40**(8): 21 – 25, 1997

Web links (Validated December 14, 2004)

[27]. B. Moreland, “Alpha-beta search,” http://www.seanet.com/~brucemo/topics/alphabeta.htm

[28]. G. Tesauro, “Benchmark player,” http://www.bkgm.com/rgb/rgb.cgi?view+610

[29]. I. Goldberg, “Falcon: Fuzzy Adaptive Learning Control Network,” http://www.generation5.org/content/2001/falcon.asp

[30]. J. Chen, S. Lu, D. Vekhter, “Game theory,” http://cse.stanford.edu/classes/sophomore-college/projects-98/game-theory

[31]. J. Jantzen, “Design of Fuzzy Controllers,” http://www.iau.dtu.dk/~jj/pubs/design.pdf

[32]. J. Jantzen, “Tutorial on Fuzzy Logic,” http://www.iau.dtu.dk/~jj/pubs/logic.pdf

[33]. J. Matthews, “An Introduction to Fuzzy Logic,” http://www.generation5.org/content/1999/fuzzyintro.asp

[34]. J. Matthews, “An Introduction to Neural Networks,” http://www.generation5.org/content/2000/nnintro.asp

[35]. J. Matthews, “How do Genetic Algorithms Work,” http://www.generation5.org/content/2001/ga_math.asp

[36]. M. Buckland, “Genetic Algorithms in Plain English,” http://www.ai-junkie.com/ga/intro/gat1.html

[37]. M. Buckland, “Neural Networks in Plain English,” http://www.ai-junkie.com/ann/evolved/nnt1.html

[38]. M. Crane, “History of backgammon,” http://www.msoworld.com/mindzine/news/classic/bg/history1.html

[39]. O. Guner, “Newsgroup posting in rec.games.backgammon made July 3. 1999,” http://groups.google.com/groups?hl=en&lr=&threadm=7lkd3o%24pd%241%40nnrp1.deja.com&rnum=1&prev=/groups%3Fq%3Dbackgammon%2Bbetter%2Bplayer%2Bluck%26hl%3Den%26lr%3D%26selm%3D7lkd3o%2524pd%25241%2540nnrp1.deja.com%26rnum%3D1

[40]. O. M. Halck, F. A. Dahl, “On Classification of Games and Evaluation of Players – with some sweeping generalizations about the literature,” http://www.ai.univie.ac.at/icml-99-ws-games/papers/halck.ps.gz

[41]. P. Crochat, D. Franklin, “Back-Progagation Neural Network Tutorial,” http://pcrochat.online.fr/webus/tutorial/BPN_tutorial.html

[42]. P. Y. Chan, H. Y. Choi, Z. Xiao, “Game trees. Alpha-beta search,” http://www.cs.mcgill.ca/~cs251/OldCourses/1997/topic11

[43]. R. Fullér, “Implementing fuzzy IF-THEN rules by trainable neural nets,” *Hybrid systems I*, http://www.abo.fi/~rfuller/nfs13.pdf

[44]. Unknown, “A backgammon Gamble Pays Off,” *Reprint from the September 1982 issue of GAMES Magazine*, http://www.edcollins.com/backgammon/backgamb.htm

10. Appendix

This is the appendix for the thesis.

The sections are:

- Content of enclosed CD
- Neural net tic-tac-toe example
- Game overviews

10.1 Content of enclosed CD

A screenshot of the content of the enclosed CD can be seen in Figure 79.

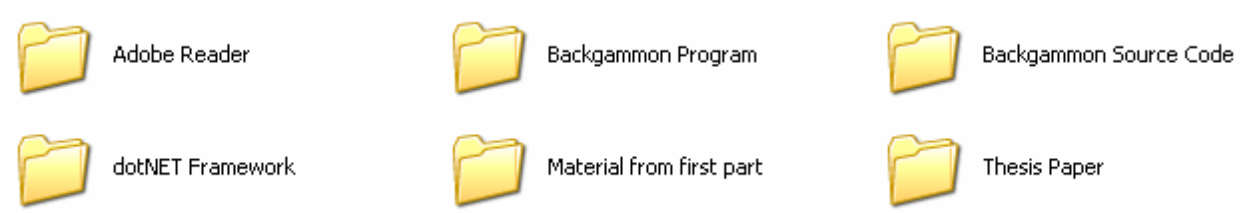

Figure 79: Screenshot of enclosed CD content

Adobe Reader: This directory contains an installer for Adobe Reader 6.01 in English and Danish

Backgammon Program: This directory contains a compiled and ready to use version of the developed backgammon program using Fuzzeval and other decision modules. It requires the Microsoft .NET Framework 1.1 to be able to run.

Backgammon Source Code: This directory contains all the source code for the backgammon program, Fuzzeval and other decision modules.

dotNET Framework: This directory contains the Microsoft .NET Framework 1.1 portable installer in English and Danish. This is required for the backgammon program to work.

Material from first part: This directory contains material from first part of this thesis. This includes the final paper after first part as well as the Tic-Tac-Toe prototype program developed during the first part.

Thesis Paper: This directory contains this paper in Microsoft Word 2003 format (DOC), in Adobe Portable Document Format (PDF) and in Rich Text Format (RTF).

10.2 Neural net tic-tac-toe example

This section will exemplify how neural network can be used to create a tic-tac-toe player. The training methods will rely on supervised learning and genetic algorithms.

10.2.1 Overall architecture

The neural network architecture can be a standard feed-forward network with 9 inputs and 9 outputs, one for each square on the board. The amount of hidden layers and node in each hidden layer is harder to decide without ability to test the networks ability to learn. So for this description we decide in one hidden layer with 18 nodes. Each input is connected to every node in the hidden layer and all nodes in the hidden layer are connected to every node in the output layer. With the decided network architecture that gives a total amount of 324 adjustable weights. For the step function we decide on a standard sigmoid function. This ensures the output from each node and the network itself, will be in the range greater than 0 and lower than 1.

The input to the network is the current representation of the board. The value 1.0 represents an O and the value 0.5 represents X. 0.0 indicates an empty square. The output from the network is 9 values in the range greater than 0 and lower than 1. The output with the highest value, and where the square is empty, is where the agent decides to place its piece. Figure 80 illustrates the network. It should be noted that for simplicity the hidden layer is illustrated as one “super” node instead of 18 separate nodes. Further more are the output values rounded at one decimal.

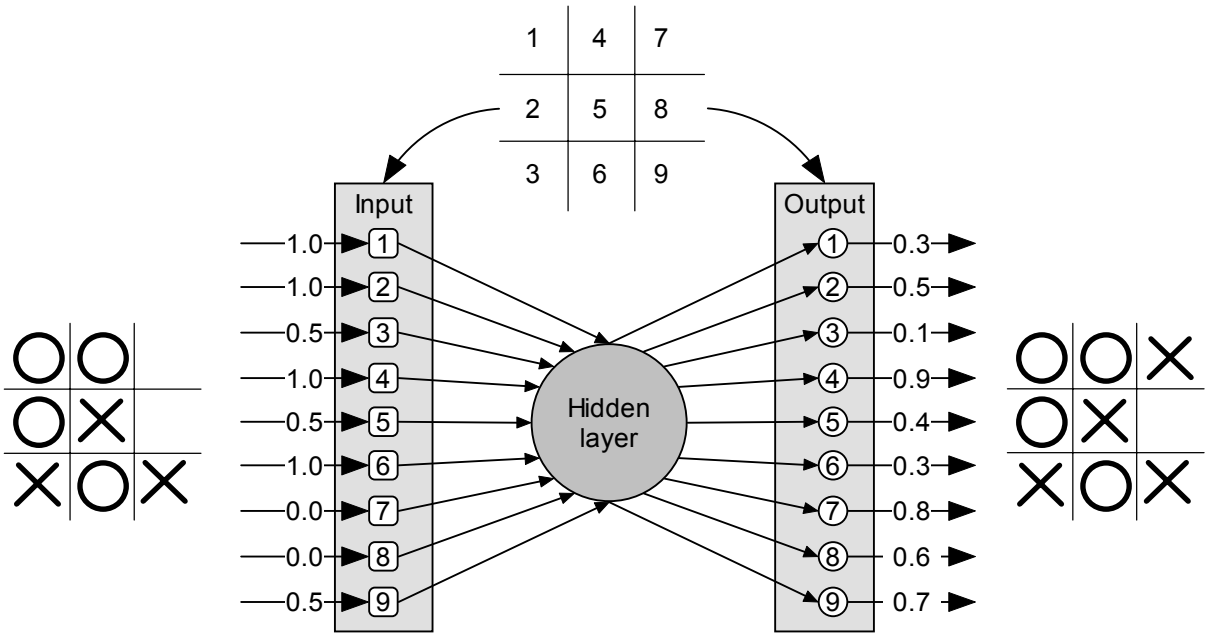

Figure 80: Neural network capable of playing tic-tac-toe

As illustrated the board layout is converted to its corresponding input values. Values will then flow through the network to obtain an output. In the above case, output number 4 has the highest output of 0.9. This square is however already taken and as

a result the square where the agent decides to put its piece is 7. This is the free square with the highest output value.

10.2.2 Supervised learning

For the agent to be able to play tic-tac-toe it is necessary for it to learn it. This can be done when playing humans by use of the back-propagation training algorithm described in section 2.7.4.

Consider the board layout illustrated in Figure 81. The human player plays X and it is X's turn.

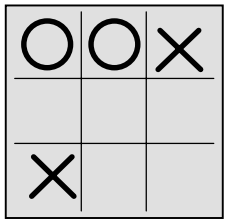

Figure 81: X's turn

X should clearly put its piece in the center square to win the game. To train the network to learn this strategy the network has to be feed with this board. The obtained output is then compared to the desirable output, which is obtained by the move the human player plays. The network is then trained by the back-propagation algorithm. This process is explained below.

First step is to switch the board such the agent sees the situation as its own. This is illustrated in Figure 82.

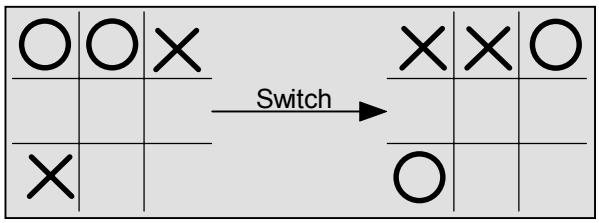

Figure 82: Switching board layout to match opponents view

The switch board is then feed as input when training the network to obtain an output. As the human player (hopefully) would put the piece in the center square the desirable output can be identified as, as low values as possible in all outputs except for the selected center square, where the output value should be as high as possible. This is illustrated in Figure 83. Remember that because the sigmoid function is used as step function, an output value can never be exactly 0 or 1; as a result the training values should never be exactly one of these values.

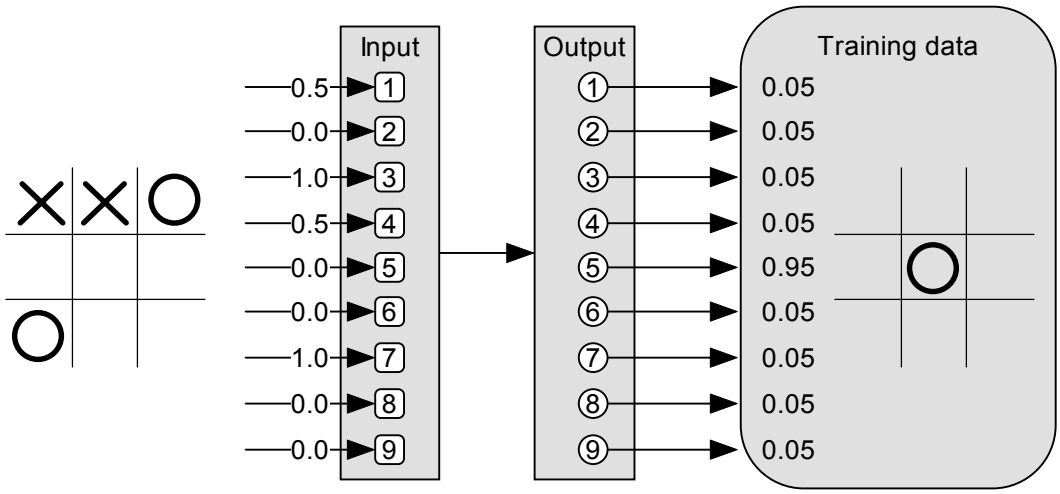

Figure 83: Training of neural network

As the desirable output for a specific input can be obtained from the player move, the network can be trained with the back-propagation algorithm. This process is then repeated every time the human player makes a move. Then as more games are played and the network is trained with board patterns, is the network slowly learning the game.

10.2.3 Learning by co-evolution

To learn in co-evolution genetic algorithms can be used to evolve the weights within the network. An amount of agents can then be initialized and set to compete against each other in a series of games. The fitness score used in the genetic algorithm evolving the network weights could then be calculated depending on how well an agent is performing. 3 points could for example be assigned for a victory, 1 point for a draw, and 0 point for a loss. The process is illustrated in Figure 84.

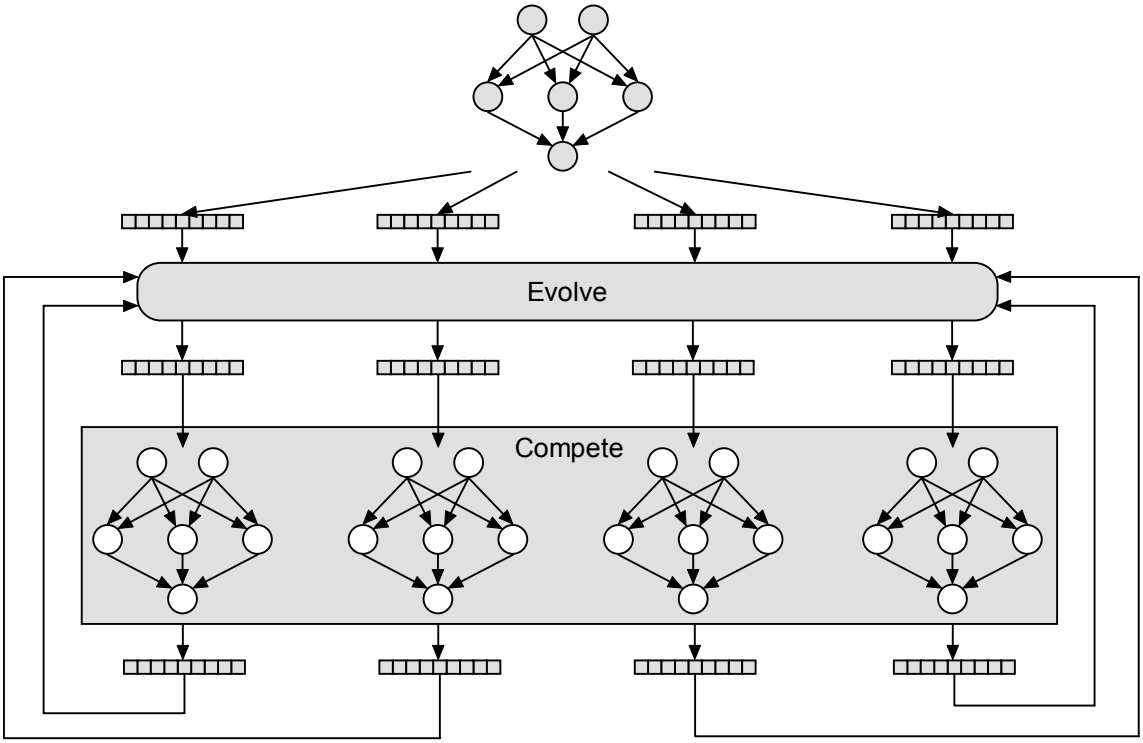

Figure 84: Tic-tac-toe training by use of a genetic algorithm

The agent's network weights are to begin with copied into a series of vectors that are to be the chromosomes in the genetic algorithm. Because all these chromosomes at first are the exact same we start by evolving them. This is to make sure the mutation step in the genetic algorithm will make a slightly alteration in some of the chromosomes. These evolved chromosomes or vectors of network weights, are then copied back into a series of neural networks. The networks are then set to compete against each other in a series of games. After the competition the network weights are again copied into a series of vectors. Depending on how well a network was performing, a fitness score is assigned to the chromosomes. After this process has been repeated a series of times, the best performing network in the final competition, is selected as the agent.

10.3 Game overviews

This section will give a brief overview of classical board games mentioned throughout this paper.

10.3.1 Chess

Chess is a game played by two players on a square board comprised of 64 alternating light and dark squares in eight rows of eight squares each. Each of the players begins the game with eight pawns and eight "major" pieces: one king, one queen, two rooks (sometimes known as "castles"), two bishops, and two knights. The two players are labeled "White" and "Black" regardless of the actual colors of the pieces. The initial board setup is illustrated in Figure 85.

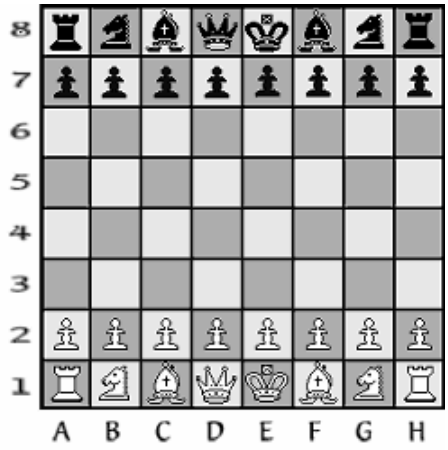

Figure 85: Chess board with initial setup

White always moves first, and the player's alternate turns. Different pieces may be moved in different ways. A player moves by transferring a piece to another square that is free or occupied by an opponent's piece. If the square is occupied, the moving player removes, or captures the opponent's piece and replaces it with the capturing piece. The chess match is won when one player maneuvers his/her pieces to checkmate the opponent's king.

10.3.2 Checkers

Checkers are just like chess a game played by two players on a square board comprised of 64 alternating light and dark squares in eight rows of eight squares each. Black always moves first. Figure 86 is a screenshot of the initial board position of a checkers game about to start.

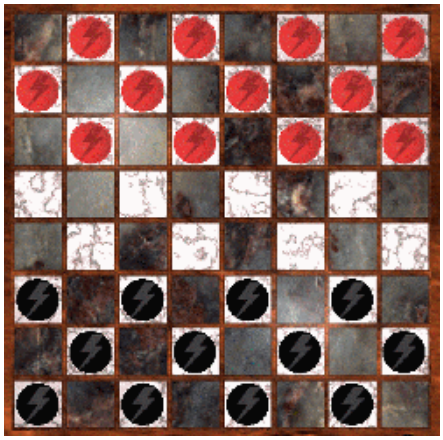

Figure 86: Checkers board with initial setup

Pieces move diagonally forward. Black starts at the bottom of the board and moves up towards Red; Red starts at the top of the board and moves down towards Black. Captures are enforced. If you can capture a piece, you must do so. If you have more then one possible capture, you may choose which to make. To capture an opponent he must be in one of the squares diagonally in front of you, and the square beyond him must be empty. The captured pieces are removed from the game. If the piece can make another capture in its new location, it must continue to jump until it either

reaches the last row or has no more possible jumps. When a piece reaches the last row, it is promoted to a King. Kings can move and capture backward as well as forward. The game is over when a player can no longer make a move either because he has no more pieces, or his opponent has him totally blocked and he has no valid moves.

10.3.3 Othello and Reversi

Figure 87 illustrates the initial board layout in Othello. In Reversi mode play starts on an empty board and the players first take turns to fill up the 4 central squares. Disregarding rotations and reflections, there are two possible outcomes to start the actual game from. The rules from that point on are the same as in Othello mode.

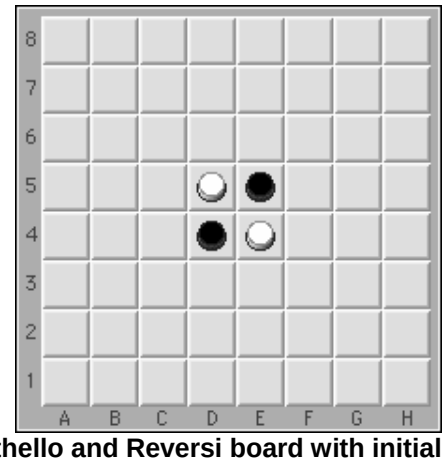

Figure 87: Othello and Reversi board with initial Othello setup

Black starts. Players must move on their turn, unless they cannot legally move. In that case the turn goes back to the opponent. If neither can move legally, the game ends. The players share 64 bi-colored stones - black one side, white the other. A move must be a “custodian capture”: the stone played must trap at least one opponent's stone or unbroken row of stones, between itself and an already present stone of like color. It can do so in up to eight straight and diagonal directions simultaneously. Captured stones are reversed immediately

The game ends by one player's resignation or if both must pass on successive turns. The winner is now the player with the most territory, that is: the highest number of stones on the board