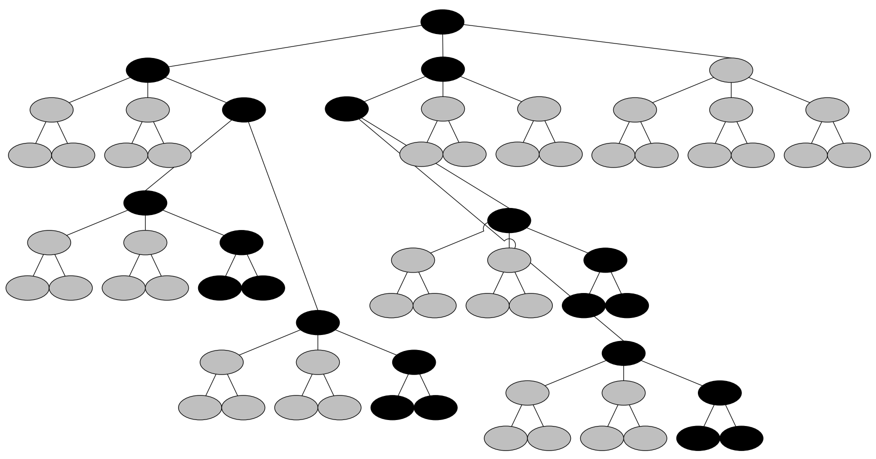
Department of Computer Science
Aalborg University, Denmark
July 31st, 2008



Master's Theses

Asymmetric Game Trees using Dynamic Bayesian Networks

Author:
Esben Skov Pedersen

Supervisor:
Manfred Jaeger

AALBORG UNIVERSITY

Department of Computer Science
Selma Lagerlöfsvej 300
http://www.cs.aau.dk

Title:
Asymmetric Game Tree using a Dynamic Bayesian Network.

Theme:
Machine Intelligence

Semester:
Dat6, 1st of February - 31th of July 2008

Group:
d632b

Members:
Esben Skov Pedersen

Supervisor:
Manfred Jaeger

Copies: 3

Report - pages: 36

Appendices: 0

DVD: 1

Total pages: 44

Abstract:

This project we improve mini max search by prediciting which move the opponent is most likely to take. This prediction is performed by using a dynamic bayesian network. This network has nodes that represents a number of abstract features extracted from the game. The network is trained using EM-learning based on data from a number of comptuter players playing against each other. These computer players are defined as the weighted sum of the features mentioned before. These weights are found using a genetic approach. The system is able to improve performance of the worst strategies without increasing the search space.

Contents

Part I

Introduction

Chapter 1

Introduction

In game theory the minimax algorithm can be considered the corner stone of zero-sum games. Assuming unlimited computing resources it can solve a game simply by exploring all possibilities until a terminal state is found. In practice the exponential nature of minimax leads to the need of optimizations and less than optimal results. The solution is often to define heuristic value functions which act as an approximation of the true value function.

A heuristic value function can be seen as the strategy of a player. If we assume there is only a number of different value functions which works in practice we can define a hypothesis space consisting of these value functions. The challenge is to map the observed moves of the opponent into this hypothesis space. If we know enough about the hypothesis space we can hopefully gain some knowledge about the next move of the opponent.

When the computer plays against an opponent it observes the board, but it has no information about the strategy of the opponent. We call the different strategies for the game the hypothesis space of strategies. We are interested in a mapping from the observations to the hypothesis. By observing a number of games played out by simple strategies, we can train bayesian networks to recognize the different strategies. When the past is inserted as evidence, the future can also be predicted. We use such a prediction as a guide line as to which paths of a game tree should be explored to a greater depth.

What we are trying to develop is a function to tell us which children of a given board position are most likely for the opponent to choose. With data from simulated tournaments, we train a dynamic bayesian network with expectation maximation learning based on this data. Later, we use the dynamic bayesian network as a function that will tell us which moves are most likely, and explore the descendants of those board positions to a greater depth.

The prediction of the hypothesis space is not bullet proof, so a backup plan might be a good idea. Say we have a prediction of the opponent choosing a given state but the prediction is wrong. The solution to this problem is to not simply prune the branch starting at that point, but to explore a small depth down this path.

As a test bed for this system we use a game we refer to as Pawns first introduced by Dimitris Kalles in [Kal07] The rules are explained in Chapter 4. The purpose is to bring one of four pawns from one corner of a chess board to the opposite before the opponent does the same. Our course of action in this

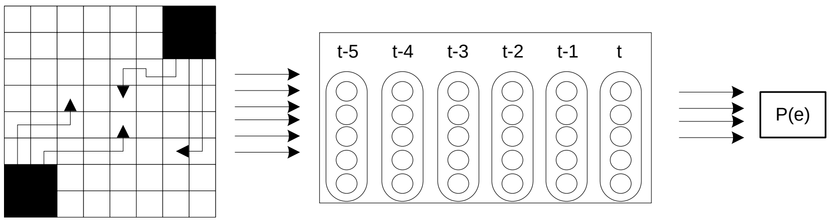


Figure 1.1: Purpose of the Project. Take the Moves and Try to predict the Next

project is to create a number of AI-players for this game. These AI-players are implemented using the MiniMax algorithm with a number of heuristic value functions. We refer to such a heuristic value function as a strategy of a player.

We want to create a hypothesis space of these strategies. In to learn more about how these strategies interact with each other we let them play a number of tournaments against each other. The results of these tournaments are saved, and we refer to these results as game data.

The ultimate goal of this project is to improve the performance of some of the strategies. We do that by creating an asymmetric game tree based on the prediction of a given node in the tree. To create an asymmetric game tree we must know more about which moves an opponent is likely to take. In order to get a prediction of the next move of the opponent we use use a Dynamic Bayesian network(DBN). This asymmetric game tree can be used to improve the performance of some of the strategies by allowing them to search the game tree to a greater depth without increasing the search space significantly.

We need to input the current state of the game into to the DBN. Using an exact representation of the board would probably not yield very good results since the likelihood of encountering the exact same position is small. So instead of using an exact representation we choose a number of features from the board used as input. An example of an abstract feature is the sum of moves to the opponent base.

To learn the conditional probabilities of the DBN, we use the previously described game data, and train the DBN to give us a relative probability of a board configuration given the configurations that led to it.

The purpose of the game data is to learn a model based on this data which can accurately predict the next move of the opponent. In order to do that we implement the Pawns game in a Windows Forms GUI and implement the minimax algorithm along with a number of value functions. A tournament structure is set up to show which heuristics has the best performance.

In Figure 1.1 we see the crucial part of this project, which is finding a relative probability of a given node in a game tree. This relative probability is denoted $P(e)$ and it is found by extracting the feature scores from a node in the game tree, and inserting those into the DBN, which given us relative probability of the given node.

Now we know what we need to complete the project we should get started on the the background theory.

Part II

Background

Chapter 2

Bayesian Networks

In this project we use a dynamic bayesian network(DBN) to model the opponent's move in a board game. A dynamic bayesian network is a bayesian network(BN) where certain parts are repeated a number of times. First we introduce bayesian networks [Yud].

A BN is a graphical model capable of performing probabilistic reasoning based on predefined conditional probabilities combined with a causality graph.

In Figure 2.1 we see an example of a simple BN. The figure represents this scenario: A large barrel is filled with eggs. The eggs can contain a pearl. The color of the egg can be observed. It can not be observed if an egg contains a pearl. We know that 40% of the eggs contains pearls. We know that 30% of the eggs containing pearls are painted blue. 10% of the eggs not containing pearls are painted blue. The bayesian network tells us about the causality of the problem, but to figure out the likelyhood of a blue egg containing a pearl we need Bayes rule.

Definition 2.0.1. Let A and B be probability distributions. Then the probability of A given B is defined as:

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)} \tag{2.1}$$

If we apply Bayes rule to our example we can calculate the probability of a blue egg containing a pearl. First we formalize the example in the same notatation as Definition 2.0.1.

$$P(pearl) = 40\% \tag{2.2}$$
$$P(blue|pearl) = 30\% \tag{2.3}$$
$$P(blue|{\sim}pearl) = 10\% \tag{2.4}$$

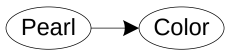


Figure 2.1: Simple Bayesian Network

Now it is time to explain Bayes rule. It finds the probability of A given B. From our example this is $P(pearl|blue)$. We can observe the color of the egg, so we need to calculate the probability of it containing a pearl.

$$P(pearl|blue) = \frac{P(blue|pearl)P(pearl)}{P(blue)} \tag{2.5}$$

Now we applied Bayes rule on our example. The only problem is $P(blue)$ is not given in the example so we have to calculate this.

The probability of the egg containing a pearl is .4% so the probability of the egg not containing a pearl is .6%. We need this number for calculating $P(blue)$. We can find this by saying $P(blue) = P(blue|pearl)P(pearl) + P(blue|\sim pearl)P(\sim pearl)$

This technique is called marginalization and can be generally described as

$$P(A) = \sum_B P(A, B) \tag{2.6}$$

Where $P(A, B)$ is the probability of A and B.

Inserting the numbers we get

$$P(pearl|blue) = \frac{.3 \cdot .4}{.3 \cdot .4 + .6 \cdot .1} = \frac{2}{3} \tag{2.7}$$

2.1 Expectation Maximation Algorithm

In the example above it was given that the chance of egg containing a pearl was 60%. In the real world these priors are not always given. When they are not given we can calculate them from data. If the data does not contain missing values we can simply let the priors be the frequencies of a given state occouring for a variable. If there are missing values we iterate towards approximate parameters. This process is known as the expection maximation algorithm.

Say we have a database with variables A,B,C. To get the maximum likelihood of $P(A = a|B = b, C = c)$ we count the number of times this occurrence has been observed in the data. This example is from [JN07].

$$\frac{N(A = a, B = b, C = c)}{N(B = b, C = c)} \tag{2.8}$$

The notation N is the number of times a given value assignment occurs in the database. The priors are found by repeating this calculation for each combination of parameters.

When the data contains missing values we perform an estimation of the missing values by using the current expectation.

Example

In Table 2.1 we see an example of a pregnancy test performed on cows. Some values are missing. The number of cases where $P(Pr = yes)$ can now be calculated by counting the non-missing values and adding the chance of the

Cases	**Pr**	**Bt**	**Ut**
1	?	pos	pos
2	yes	neg	pos
3	yes	pos	?
4	yes	pos	neg
5	?	neg	?

Table 2.1: An example of a pregnancy test for cattle. We see 5 cases. A variable for pregnancy(Pr), A blood test(Bt), and an urine test(Ut).

missing values where $P(Pr = yes)$. The non-missing values are $3 \times yes = 3$. If we assume an even distribution in the first iteration of the algorithm, missing values contribute to $P(Pr = yes)$ with .5 So to sum up we have the expected number of counts of $Pr = yes$ denoted by $E[N(Pr = yes)]$

$$\begin{aligned} E[N(Pr = yes)] &= P(Pr = yes|Bt = pos, Ut = pos) \\ +3 \cdot 1 + P(Pr = yes|Bt = neg) \\ &= .5 + 3 + .5 \\ &= 4 \end{aligned}$$

Similarly

$$\begin{aligned} E[N(Pr = no)] &= P(Pr = yes|Bt = pos, Ut = pos) \\ +3 \cdot 0 + P(Pr = yes|Bt = neg) \\ &= .5 + .5 \\ &= 1 \end{aligned}$$

In the first iteration of the algorithm the conditional probability of $P(Pr = yes)$ is

$$P(Pr = yes) = \frac{4}{5} = .8$$

In general for $P(Pr = yes)$ this looks like

$$E[N(Pr = yes)] = \sum_{d \in D} P_0(Pr = yes|d)$$

Where E is the expectation of a variable. D is the dataset and P_0 is the initial conditional probability.

This is done for all parameters. The procedure is repeated until a stopping criteria is met.

2.2 Dynamic Networks

We have now shown how to model causality, but for a domain that evolves over time, our previous models are not strong enough. To model temporal properties in a bayesian network, we introduce dynamic bayesian networks [JN07].

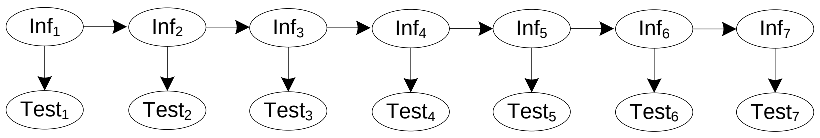


Figure 2.2: Example of a Dynamic Bayesian Network with Seven Time Slices

A dynamic bayesian network is a bayesian network, where a certain part of the network is repeated a number of times. Such a part of the network is known as a time slice. In Figure 2.2 we see an example of such a network[JN07]. The model is a classifier for infected milk, where the nodes Inf_i is an infection at day i, and $Test_i$ is the test result at day i. The logic behind such a network is that the past affects the future.

2.3 Object Oriented Bayesian Networks

Now that we have introduced a way of modelling temporal properties in a bayesian network, we can go on to a more modelling trick employed for making more logical model, where we hide information much like object oriented programming(OOP). This technique is called object oriented bayesian networks(OOBN).

In OOBNs[JN07] we can abstract away from internal representation by only exposing the input, and output nodes of a domain. When building oobns we have the option of marking a node as input, or output(or no marking which is what we do when not building OOBNs).

When we mark a node as input, the node acts as a placeholder of a node in another instance, just like a reference in most OOP languages. We can describe the input node as the formal parameter, and the node connecting to it the actual parameter.

Nodes can be marked as output nodes. When a node is marked as output, we can use the node as the actual parameter of input nodes. When connection an output node to an input node, we require that the nodes are of the same type, and have the same states.

In Figure 2.3 we see a bayesian network of the the driving characteristics of a car. The example is from [JN07]. The interesting part is the three nodes in the square. That part handles the grip of the car. The three nodes in the square speficies the interface of a separate network. Such a separate network can be seen in Figure 2.4, which is the implementation of the interface. One advantage of this approach is any implementation can be used as long as it satisfies the interface. Also note how the network in Figure 2.4 contains additional nodes not part of the interface, which allows for arbitrarily complex implementations of an interface.

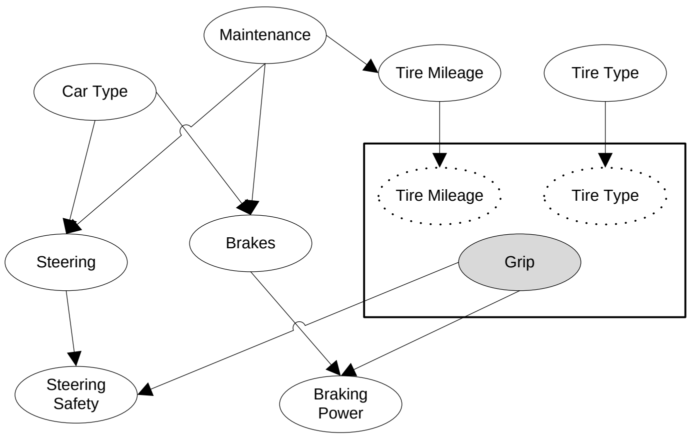


Figure 2.3: Example of an oobn

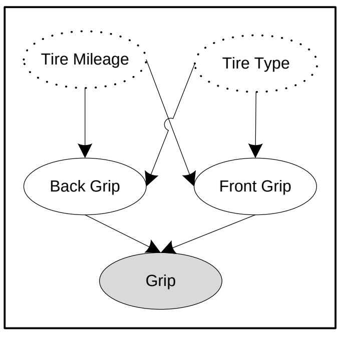


Figure 2.4: Implementation of the interface from Figure 2.3

Chapter 3

Minimax Algorithm

The Minimax algorithm can be considered the corner stone of zero sum two player games, where what is good for one play is bad for the other. In this chapter we review game trees, minimax and some optimization.

In a tree, let the nodes be legal board configurations, and let the children be the possible moves in a given configuration. Such a tree is called a game tree [Jun].

A game tree grows exponentially, meaning the search of a game tree usually requires a heuristic value function to be useful. In Figure 3.1 we see an example of a game tree with a cut-off depth of four plies. A heuristic value function has been applied at level 4.

With the concept of game trees in place we can move on to the minimax algorithm. The algorithm works by iterating the possible moves of a board configuration and backing the found values up to the root.

Lets start by looking at Figure 3.2 as an example. Of the two players, min and max, in Figure 3.1 it is now max's turn to move.

The minimax algorithm works by taking the maximum(minimum) value of each child if it is max's (min's) turn. In Figure 3.2 we see the values in the game tree after minimax have backed the values up to the root. When the algorithm reaches the cutoff depth, the node is evaluated using the heuristic evaluation function. The parent of a given node chooses then the best value. This is repeated until the root is reached.

Listing 3.1: Minimax Algorithm

```
function minimax(board, limit){
```

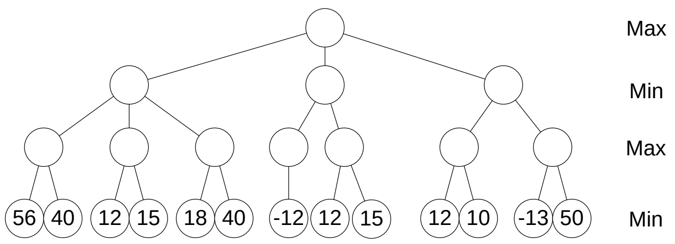


Figure 3.1: Example of an Initial Game Tree with Values at Level 4

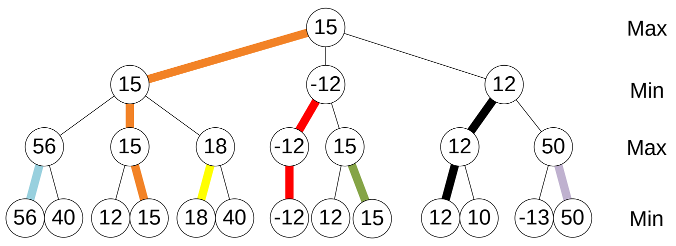


Figure 3.2: Game Tree where Values are Now Backed up to the Root

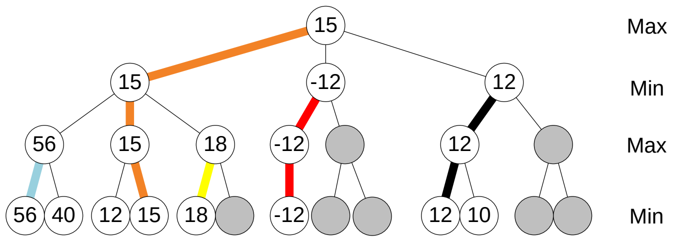


Figure 3.3: Example of Minimax Search using Alpha Beta Pruning

```
  if(limit = 0 or winner(board) <> none)
    value <- evaluate(board)

  else if(board.turn = min)
    value = infinity
    foreach(child in children(board))
      min(value, minimax(child, limit -1))
  else
    value = -infinity
    foreach(child in children(board))
      max(value, minimax(child, limit -1))

  return value
}
```

3.1 Alpha-Beta Pruning

Because minimax search has an exponential runtime, any optimization is welcome. One such optimization is alpha beta pruning. The intuition behind alpha beta pruning can be described as not putting one self in a situation, which is worse than what we already know we can archive [Lin]. In the examples children are evaluated left to right. Color traces are added s.t. it is easier to see which child a value is coming from.

In Figure 3.3 we have extended the previous example and implemented alpha beta pruning. The grayed out nodes are never evaluated.

Consider the left-most grayed out node in Figure 3.3. It does not have to be evaluated because the min player never will choose the node with 18 because it is guaranteed to get 15.

The next cluster of grayed out nodes from the left does not need be evaluated because the max player knows it can get a 15. But by going down the middle branch from the root, it gives away a -12 to the min-player. The last cluster follows the same concept as the middle. We know we can get a 15, which is why we can ignore any nodes where we are only guarenteen a 12.

With the intuition in place, we can now define the algorithm.

Listing 3.2: Minimax algorithm with alpha beta pruning

```
function minimaxab(board, limit, alpha, beta){
  if(limit = 0 or winner(board) <> none )
    value <- evaluate(board)
  else if(board.turn = min)
    foreach(child in children(board))
      value <- minimaxab(child, limit-1, alpha, beta)
      if(value < beta)
        beta <- value
      if alpha >= beta
        break

  else
    foreach(child in children(board))
      value <- minimaxab(child, limit-1, alpha, beta)
      if(value > alpha)
        beta <- value
      if alpha >= beta
        break

  return value
}
```

In listing 3.2 we see we have extended the minimax algorithm with two extra parameters: alpha and beta. The alpha(beta) represents the best value so far for the max(min) player. If alpha gets larger than beta we can stop the search at a given branch.

Notice also the order of the children is important for the number of pruned nodes.

Chapter 4

Introducing Pawns

We need a game that we can use as a test bed. For this we choose the game Pawns, first introduced by Dimitris Kalles in [Kal07]. It is a two-player board game played on a chess board. Each player has n pieces and one base.

In Figure 4.1 we see the board for the Pawns game. The bases are located in upper right corner and lower left. The size of each base is 2x2.

The game is over when one player moves a piece in to the opponents base or one player has no more pieces. If at any point a piece don't have any valid moves it will be removed from the board. A such piece will be refereed to as dead.

At any turn in the game a player can move one piece to an adjacent field at same or greater distance from it's base. The distance to the base is defined as the maximum of vertical and horizontal distance from the base.

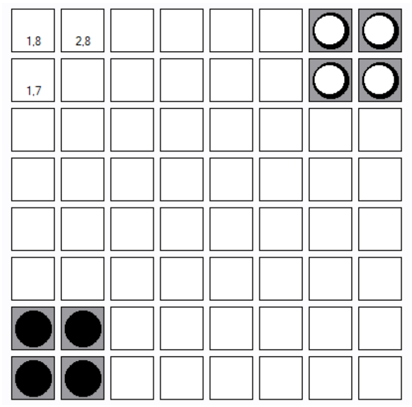


Figure 4.1: Pawns Board with 4 Pieces for Both Players

Chapter 5

Creating Strategies

A common approach when experimenting with new approaches in machine learning is to obtain data from human experts playing the game. This is practical in Go or Chess where tournaments are held. In Pawns not many tournaments(none!) are held, so we have to simulate some data by letting a number of different strategies play against each other.

5.1 Harmony Search

We could just create some strategies based on random assignment of a weight vector of a number of features. This chapter describes a general way of searching for a vector given some score function. This approach is a genetic algorithm known as harmony search[Gee]. In this project we will use harmony search for finding the strategies we will try to recognize and predict. For now, we can just think of a strategy as a vector of integers.

The algorithm can be expressed like this[Gee]:

1. Initialize memory, pick k random vectors: $x_1 ... x_k$
2. Make a new vector x' for each component x'_i
   - With probability p_{hmcr} pick the component from memory: $x'_i = x_i^{rand(k)}$
   - With probability 1 - p_{hmcr} pick a new random value in the allowed range.
3. Pitch adjustment: For each component x'_i:
   - with probability p_{par} change x'_i by a small amount, where x'_i can take the a valid value bw indexes from the current value.
4. If x' is better than the worst x_i in the memory, then replace x_i by x'
5. Repeat from step 2 until a maximum number of iterations has been performed

Where k, p_{hmcr}, p_{par}, and bw are constants, and k is the size of memory. Typical values of p_{hmcr} is .95, and p_{par} is in the range .3 : .99. The function rand(k) gives a random non-negative integer less than k.

Part III

Opponent Modelling

Chapter 6

Overview

With the background theory in place it is about time we provide an overview of this project.

Lets start by looking at Figure 6.1 which provides all the building blocks of this project. The first component is Harmony* search. We call it Harmony* because it is slightly altered from standard Harmony. The scoring function which we use is a relative measure of the memory, whereas standard harmony is an absolute measure. Read the definition of Harmony in Chapter 5.1 and the test setup in Chapter 10.

The purpose of Harmony* search is finding a number of vectors, that work as the weights for the weighted sum that defines a strategy. The strategies found by Harmony* are the strategies which we try to identify later.

The next point in Figure 6.1 is strategy. We want to be able to identify the next move given the past. For that purpose we have a latent variable in our network. This latent variable is the strategy of the opponent. We define the strategy as a heuristic value function, which we use in minimax search.

The next point is data. We can use the strategies to generate data. This data consists of all the board positions in a number of tournaments. A tournament consists of all strategies playing against all strategies. As randomness is involved in the implementation of the strategies, we play a large number of tournaments. Randomness is involved when two board positions have the same

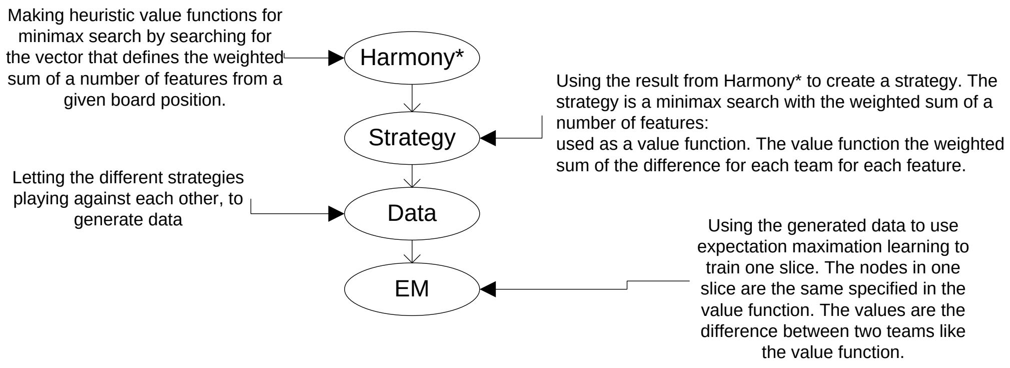

Making heuristic value functions for minimax search by searching for the vector that defines the weighted sum of a number of features from a given board position.
Harmony*
Using the result from Harmony* to create a strategy. The strategy is a minimax search with the weighted sum of a number of features:
used as a value function. The value function the weighted sum of the difference for each team for each feature.
Strategy
Letting the different strategies playing against each other, to generate data
Data
Using the generated data to use expectation maximation learning to train one slice. The nodes in one slice are the same specified in the value function. The values are the difference between two teams like the value function.
EM

Figure 6.1: How to Create the System

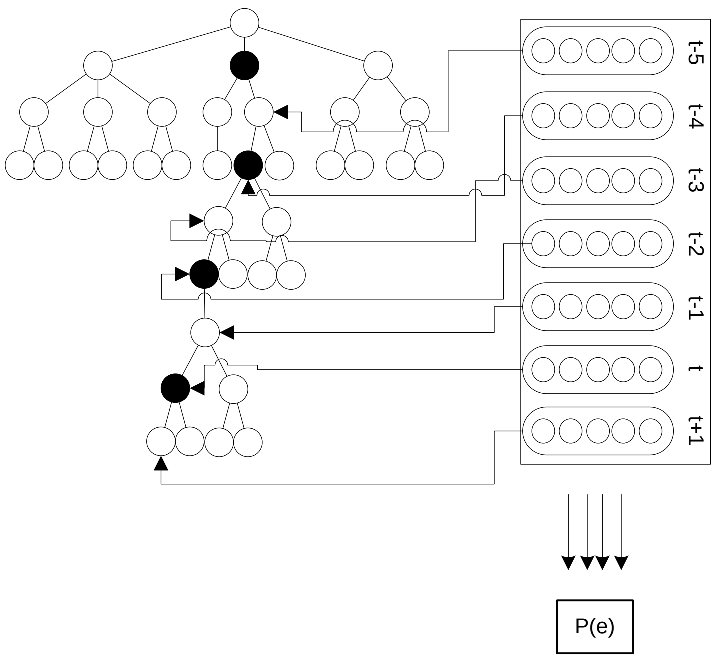


Figure 6.2: How to Use the System

value. We choose between them be always adding a small random number in the range $] - .5 : .5[$ to the value function when evaluating the board during minimax search.

When the data is generated we can use this as input for EM learning. The features are extracted, and saved in a file, ready to by used by hugin for EM learning. Each line in the file consists of the feature scores of the current board configuration, and the previous. The EM learning is used for training a slice for the DBN.

The core of the system is the DBN. The DBN consists of a number of slices. Each slice is an oobn consisting of the same features as the value functions, trained using EM learning. We will describe this more fully in Chapter 8, and refer to it simply as the DBN. The DBN is supposed to work with the game tree. This can be seen in Figure 6.2 where the parents of a node in the game tree is inserted into the DBN. This gives us $P(e)$ for a given node. This is repeated for all siblings. We can now compare $P(e)$ to get an estimate of which siblings are most likely.

The notion $P(e)$ is the probability of evidence. Evidence in this context is the feature scores from a number of board configurations up to and including the currect board.

The final point is improving the performance of a strategy, by choosing which branches in a game tree should be explored more, based on the result of the DBN. This is also described in Chapter 9.1 and is referred to as an asymmetric game tree

6.1 Board Games

Now that we have described how to build and use the DBN to improve performance. It is time to describe when this technique is useful.

The technique we describe here, referred to as an asymmetric game tree is useful when ever one wants to imprive an AI for a board game.

Lets say we have a board game GB and a heuristic value function GB_v. We also have a set of games played between two players. Each element of the set consists of a sequence of legal board positions representing the moves made by the players. These games could be human experts playing against each other, or human experts playing against an AI, or like in this project: AIs playing against AIs.

The last element in the construction of the system is selecting a number of features from the representation of a board. The big question now is: which features should be selected. It is not an easy question, but we will try to answer. We want features that distinguish the different strategies. We then try to match the features to something that can distinguish the different strategies of the game. This could be how far from initial or goal position the pieces are. This can be combined with min(), max(), avg() and sum(). Anything that can be quantified can be used as a feature.

Say the game in question is chess: Example features are:

- Number of pawns not in initial position.
- Each type of piece: the distance to initial position.
- Distance to the king
- (Weighted) Sum of pieces.

These are just some of the features one could choose. All the distances can of course be combined with min/max/avg/sum.

We now proceed to define the features that we use in the pawns game.

Chapter 7

Features

We now define the features used in the pawns game. These are used for predicting the next move, as well as defining strategies as the weighted sum of the features scores. In this project, black is always the max player, and white is always the min player.

Definition 7.0.1. Let p be a set of pieces of the same team. Then we define a feature as a function f that returns a number $n \in N$ with p as a parameter: $f(p)$

A feature can be anything we can measure by the positions of the pieces. An example is the number of pieces.

Definition 7.0.2. Let b be a set of black pieces, and w be a set of white pieces. A feature score of a feature f is defined as:

$$f(b) - f(w) \tag{7.1}$$

An example of a feature is the number of pieces. e.g. we prefer an AI not to lose it's pieces. Now that feature scores are defined, we can define the value of a board as the weighted sum of a number of feature scores. Defining an AI is now reduced to choosing the weighted sum we want to use in a given AI.

The features we choose are:

Definition 7.0.3. Let board be a representation of a board and board.Black be a list of black pieces and board.White be a set of white pieces, we define the feature score Player Diff(PD) as:

```
PlayerDiff(board) {
  return board.Black.Count() - board.White.Count();
}
```

The logic behind PlayerDiff (PD) is to give a reward if a player has more pieces than the opponent, and by the same logic punish a player for being in a situation with fewer pieces than the opponent.

Definition 7.0.4. Let board be a representation of a board and board.Black be a set of black pieces and board.White be a set of white pieces, we define the feature score DistSum (DS) as:

```
DistSum(board)
{
return
 (from b in board.Black
        from b2 in board.Black
        select b.Distance(b2)).Sum()
      -
      (from b in board.White
       from b2 in board.White
       select b.Distance(b2)).Sum();
}
```

The purpose of DistSum (DSum) is to measure the sum of the distance of all the pieces of a player. DistSum is not necessarily a bad thing. Having pieces close together can help trap opponent pieces. On the other hand having pieces far away can be harder to defend against for the opponent.

Last, we have four feature scores based on the distance to the base, or distance to the opponent base.

- Opponent Base Min (OBMin, drive one piece to the opponent base)
- Opponent Base Sum (OBSum, drive all pieces to the opponent base)
- Base Max (BMax, drive one piece away from home base)
- Base Sum (BSum, driver all pieces away from home base)

These should be self-explanatory. The purpose is summarized in parenthesis)

7.1 Usage

Now the definitions have been defined we can talk about the usage. There are two usage scenarios for the abstract properties.

In the first case the feature scores are used as a value function in a minimax search. A strategy is defined as a vector of weights.

Definition 7.1.1. Let V_f be a vector where the elements are the chosen feature scores. Let V_s be a vector of integers. We then then define the value of a board as

$$V_f \cdot V_s$$

V_s is the strategy and when we refer to a strategy we are talking about a vector V_s

The second use case is where we are observing the moves of the opponent, and set evidence in our model. The model can then tell us how likely a given board position is, and we can choose to ignore some of the most unlikely moves from the opponent when searching the game tree.

Chapter 8

DBN

In order to be able to predict what the opponent is doing we build a DBN which can tell us what the opponent is most likely to do next. This DBN is valuable when performing minimax search s.t. our search space is reduced. For doing that we need some data. The data which we use comes from a tournament played out between the different strategies. This chapter describes how we can turn this data into a classifier that can say how likely a given sequence of moves is.

The slice in Figure 8.1 represents one board configuration. It has a number of features ($F_1 ... F_n$) and two strategy nodes. Nodes with a dotted line are input nodes, meaning they are not real nodes in the sense that they are to be replaced with a real node when the network is compiled. For all slices except the first, it holds that slice t is connected with slice $t - 1$ meaning the previous slice representing the previous move from the same player.

8.1 Connecting

In the previous section we saw how we used simulated data to learn one slice of our DBN. In this section, we will show how the slices are connected.

In Figure 8.2 we see the structure of the network. It is the slice from Figure 8.1 repeated a number of times. The variable t is the time of the current slice. The value in each node is the feature scores. Since feature scores are defined as the difference between features from black and white it makes sense to use the

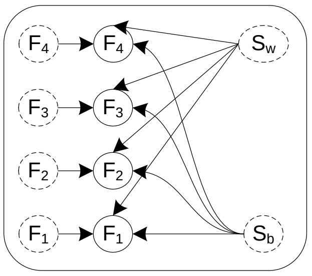


Figure 8.1: One slice from the network

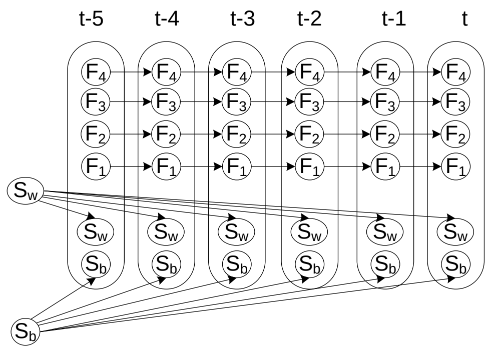


Figure 8.2: Connecting the slices

same slice for both black and white moves. We also have two strategy nodes: S_b, and S_w. A slice represents the feature scores after a given move. The slice with time t is always white who is about to move. Slices before and after represents the moves that led to the configuration and slices after represents a descendant of time t.

The slices for the two players are identical. Even though one slice represents a move from one player, we choose to include the data from both players in one slice. Remember that the input to the DBN is the feature scores from the board. Since feature scores are defined as the difference of a given function of the black player's pieces minus the white player's pieces, this means that the values between two slices are almost identical, since one player can only move one piece to an adjacent field. Another approach would be to train a slice only for the black moves, and another for the white moves. We do not adopt that approach since this is a much simpler solution which we can expect to have equal or better performance than the two-slice approach.

When designing bayesian networks it is important the model makes sense. When playing both players react to each other. This results in a given board position. This board position is abstracted and inserted as evidence in the network. This can hopefully help predicting the next move of the opponent. If we assume both players are rational the position of the white player will depend on the moves of the black player. The moves of the black player depends on the strategy of the black player. The strategy of the black player is inserted as evidence in the network. The states corrosponding to the moves which have been seen most often during training will have a higher prior probability. Since the states have a higher probability, they can help predict the moves of the opponent.

Recall that we built the network using oobn's, this allows for a more flexible approach when experimenting with the number of slices appropriate for making a prediction.

8.2 Learning

We use EM learning described in Section 2.1 to learn one slice in our dynamic bayesian network. One slice represents one move.

$F_{1,(t-2)}$	$F_{2,(t-2)}$	$F_{3,(t-2)}$	$F_{1,t}$	$F_{2,t}$	$F_{3,t}$	S_b	S_w
$V_{1,1}$	$V_{2,1}$	$V_{3,1}$	$V_{1,3}$	$V_{2,3}$	$V_{3,3}$	sid	?
$V_{1,3}$	$V_{2,3}$	$V_{3,3}$	$V_{1,5}$	$V_{2,5}$	$V_{3,5}$	sid	?

Table 8.1: Data for EM learning

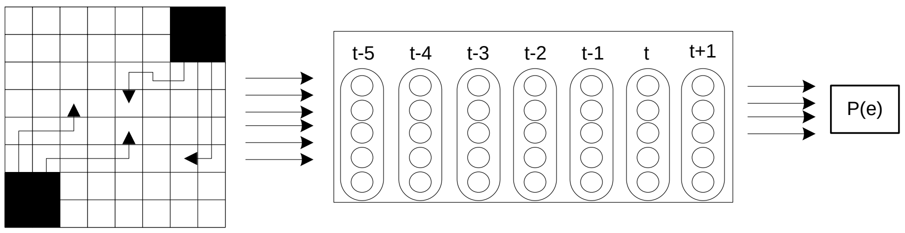


Figure 8.3: The Slice at t+1 is a child of the slice at Time t

In the process of EM learning we are interested in learning one slice at a time. For that we create input in the form of a .dat file. This file is created from the tournament data.

The slice has two strategy nodes. One for us, and one for the opponent. We know our own strategy, and the slice has been trained with evidence in that node. In most real world scenarious we don't know anything about the opponent's strategy, so we treat that as a latent variable.

The format of the .dat(for features ($F_1 ... F_3$) and strategy S with values ($V_1 ... V_3$) for time (1...5) can be seen in Table 8.1. We use the notation $V_{i,t}$ for the value corresponding with F_i at time t. The field S_b is the strategy id of the black player. The field S_w is a symbol representing a missing value.

8.3 Predicting Next Move

We would like to be able to predict the next move of an opponent. We use the DBN for predicting such a move, and this section describes how we make such a prediction. The notion "‘predicting next move"’ is slightly incorrect. We are not really predicting the next move, but more predicting the probability of the feature scores of the available moves in the given context. This is due to the decision of not making an exact representation of the board in order to increase the likelihood of getting a reasonable answer even though the exact board position never showed up during training. This approach has the consequence that we are likely to have multiple nodes with the same $P(e)$ value, so it is not completely trivial to figure out how to choose which nodes to pursue. This we will investigate further in the results part of the report.

Recall the introduction, where we have a figure much like Figure 8.3. We have now extended the figure s.t. a slice with $t + 1$ is now present. The slice with time $t + 1$ is a child of the t slice.

In Figure 8.3 we also see the notion $P(e)$ which means probability of evidence. We insert evidence in all slices, and we get the number $P(e)$. This number is crucial in determining which child we should focus the search on. Predicting the next move is now reduced to a matter of comparing the $P(e)$ corresponding to each child.

Chapter 9

Search Heuristic

We have the DBN which can give us a number $P(e) \in R$. This number is a positive number close to zero. The higher the number, the more likely a given sequence is.

We use this property in a modified minimax search. We call this an asymmetric game tree.

9.1 Asymmetric Game Tree

We assume we are the black player as always. Say it is now black's turn to move and we have a list of all previous moves. We now insert this list as evidence in the classifier along with one child of the current node. This is done for all children. We can now assign a value of $P(e)$ to all the children. We use this value to assign a maximum search depth for the children. Normally in minimax search we have a constant search depth, e.g. 8. In the asymmetric game tree we assign the maximum search depth based on $P(e)$ s.t. the branch under the children with the highest value will have a higher limit.

In Figure 9.1 we see an example of an asymmetric game tree. The solid painted nodes represent the nodes the DBN deemed as most likely, and thus their descendants are investigated further. In the example given we assume a default search depth of 4, which is the depth which would be searched if no search heuristic is applied. With the search heuristic we can hopefully continue the search beneath this limit with a linear increase in the nodes searched.

9.1.1 Example

We now create an example to show how the asymmetric game tree works.

In Figure 9.1 we see an asymmetric game tree where the square boxes symbolize a prediction. The value in the prediction box is $P(e)$ of the past and current nodes. The number in the nodes is the limit of the minimax search for descendants of the given node.

Let's explain Figure 9.1 from the top. We use standard minimax search. In right side a label explains if the nodes at a given level is a min or max node. As always we take the perspective of the max node. We try to predict the move of the opponent. The first time the opponent has to move is in level 2. We use

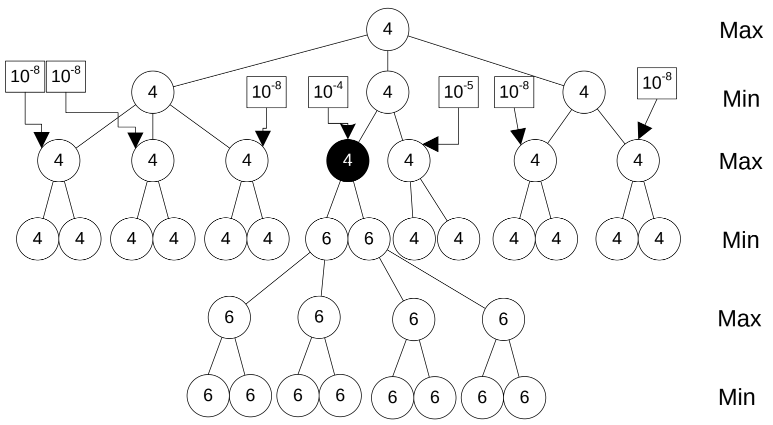


Figure 9.1: Game Tree with Search Heuristic Applied. Black Nodes are the Most Likely

the DBN to give the probability of a node. A black node represents a node we have predicted as most likely. In this example we use the simple heuristic of choosing the best $P(e)$ and setting that as the most likely. This is not a good approach and it is only used in this example for sake of simplicity.

In this example we see a number of values in a square box. These are a relative propability of the opponent choosing that particular node. In the middle is a node colored black. This indicates the node has a higher probability than all it's siblings. Because of that we increase the search depth of descendants of that node.

9.1.2 Algorithm

It now time to write the algorithm for asymmetric game trees than previously described. We use a pseudo code inspired by C# and java.

```
next(board)
{
  return search(board, 0, -INFINITY, INFINITY)
}


search(board, depth, alpha, beta){
//creates an asymmetric game tree by utilizing
//a DBN to predict which moves are most likely
//this is a modification of the minimax algorithm
//with alpha beta pruning
  if(depth >= board.depth){
  // board.depth is dependent of how likely
  // the ancestors are
    board.value = heuristic.evaluate(board);
    board.bestvalue = board.value;
    return board;
  }

  if(board.turn == white && board.trypredict){
    //all the ancestors are deemed most likely by the DBN
    moves = children of board where each child has been assigned P(e)
    foreach (move in moves){
      if(mostlikely(move)){
        //increase the search depth for descendants of this node
        move.depth = MAXDEPTH + 2
        move.trypredict = true
      }
      else{
        move.depth = MAXDEPTH
        move.trypredict = false
      }
  }else{
    moves = children of board
    foreach(move in moves){
      //we either have a black node or a white node
      //which is not likely
      move.trypredict = board.trypredict
      move.depth = board.depth
  }

  foreach(move in moves){ // standard minimax
    Board tmp = search(move, depth+1, alpha, beta)
    if(tmp.turn == white  && tmp.bestvalue < beta)
      beta = tmp.bestvalue
      best = tmp
    }
    if(tmp.turn black && tmp.bestvalue > alpha){
      alpha = val.bestvalue
      best = tmp
    }
    if(alpha >= beta)
      break
    }

    if(tmp.turn == white)
      board.bestvalue = beta
    else
      board.bestvalue = alpha
    return best;
  }

  abstract mostlikely(node){
    // see results part of the report for this method
  }
}
```

The code is explained in the comments. See Section 11.1.1 for more information of the mostlikely() method.

Part IV

Results

Chapter 10

Harmony Search

In Chapter 5.1 we described how harmony search can be used in order to find a vector that minimizes some cost function. In Chapter 7.1 we defined a strategy for the game as a weighted sum of the feature scores described in Chapter 7.1. This chapter describes in detail how we found the strategies that we re trying to recognize.

The purpose of this search is finding a number of strategies which are not too similar. Very similar strategies could prove hard for the DBN to recognize. We will search for strategies which perform well.

First a harmony search with 4 random vectors were set to play against each other. Using the cost function described earlier, where all strategies play against each other, we count the number of wins, loses and draws. The cost is $NW - W$ (not win and win respectively).

We use a modification of Harmony search. We want a relative measure of performance, because we want strategies which performs well, and a good way to find such strategies is to let them play against each other. Because this cost function is a relative measure we refer to the algorithm as Harmony*.

This algorithm can be described as this:

1. Initialize memory, pick k random vectors: $x_1 ... x_k$
2. Make a new vector x' for each component x'_i
   - With probability p_{hmcr} pick the component from memory: $x'_i = x_i^{rand(k)}$
   - With probability 1 - p_{hmcr} pick a new random value in the allowed range.
3. Pitch adjustment: For each component x'_i:
   - with probability p_{par} change x'_i by a small amount, where x'_i can take the a valid value bw indexes from the current value.
4. Let all the strategies represented by the vectors play a tournament, and find the score: $NW - W$ (specific to Harmony*).
5. If x' is better than the worst x_i in the memory according to $NW - W$, then replace x_i by x'

Strategy ID	BMax	BSum	DSum	OBMin	OBSum	PD
1	0	10000	0	-1000	-10	0
2	10	100	0	-10000	-100	0
3	0	10	0	-10000	0	0
4	100	1000	0	-10000	0	10

Table 10.1: Chosen Strategies

		White			
		1	2	3	4
Black	1	67%	0%	67%	0%
	2	67%	33%(67%)	33%	0%
	3	67%	33%(33%)	33%	33%
	4	100%	100%	67%	100%

Table 10.2: Balance of Power Between Chosen Strategies

6. Repeat from step 2 until a maximum number of iterations has been performed

After each iteration of the Harmony* search we remove the worst performing vector from memory. The size of the memory(k) is 4. incidentally we also search for 4 different strategies so we perform the search 4 times. These numbers are unrelated and the similarity is completely incidental. The algorithm has 4 parameters. In this setup we use $p_{hmcr} = .95$, $p_{par} = .5$, $bw = 2$ and $k = 4$

The strategies in memory tend to be quite similar, so just taking the memory of one search and using those for the project is not feasible. The solution is to perform 4 searches, and take the best from each search, then see if the power is balanced, s.t. they all are able to win against some other strategy.

We are now ready to run the search. We let it run for 100 iterations. The strategies found are seen in Table 10.1

These four chosen strategies are what we try to recognize. The power of balance between these can be described as the chance of a given strategy winner. This is seen in Table 10.2, where the number represents the ratio of won games for the black player. The data is from 144 games(9 tournaments). The number in parenthesises is the percentage of games ending in a draw. In this sample a game is declared a draw if no winner is found within 200 moves.

The setup is as follows: Each strategy plays against all other strategies(including itself). Table 10.2 shows the black player on the vertical direction. The black player always starts first. We use minimax with a search depth of 6 for this experiment.

From the data it can be concluded that all of the strategies can beat at least two of the other strategies, but two strategies perform very poorly against one specific opponent.

Chapter 11

Performance

In this chapter we show how the strategies benefit from using an asymmetric game tree.

11.1 Asymmetric Game Tree

This section describes if the asymmetric game tree improves performance to an AI due to increased look-ahead in the same number of nodes. The black player is using an asymmetric game tree which can predict the next move of the opponent to a certain degree.

11.1.1 Pessimistic Ratio

We need some measure to determine if the prediction can be considered one of the most likely moves at a certain point. One important criteria to have in mind is the case when all children have the same $P(e)$. Lets say we have a set of $P(e)$ of the siblings, and sort these descending we could get a list like $[10^{-6}, 10^{-8}, 10^{-10}]$. That might be a reasonable approach unless all siblings have the same $P(e)$. We need a more pessimistic approach so to say. A solution to this problem is something which we call the pessimistic ratio.

Definition 11.1.1. Let L be a multi set which represents $P(e)$ for all the siblings at a given point in time, eg. $P(e_i) = L_i$ where $L_i \in R$ and represents the $P(e)$ of sibling i. The pessimistic ratio, $pr(L, L_i)$ of a number L_i is then

$$pr(L, L_i) = |\{p \mid p \in L \wedge p >= L_i\}|/|L| \tag{11.1}$$

We use *pr* for determining which children can be considered for further exploration. The number pr is well suited for this purpose because the case where all siblings are deemed equally likely, they will have a pr of 1, if a single child is the only one with the highest $P(e)$ it will have a low number: e.g: $\frac{1}{10}$ in the case of 10 children. During the search in the asymmetric game, we need to find a constant MAX_PR which pr must be less than to have their search depth increased.

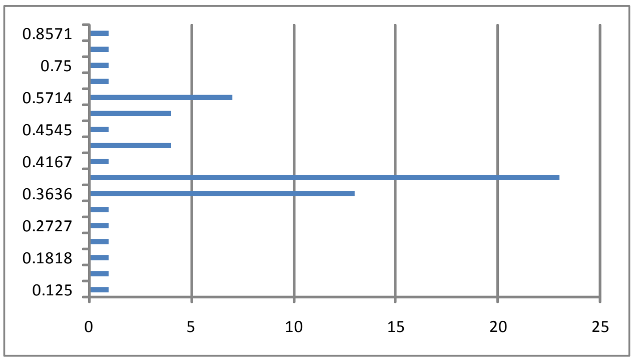


Figure 11.1: Histogram of the Number of Times a Certain *pr* has been Played by the Opponent

Using Pessimistic Ratio

A naive approach to using the pessimistic ratio is to require the ratio is below a certain constant pm_{limit} to be considered most likely. Comparing this with an approach where $P(e)$ from the DBN is substituted for a random number shows that the DBN performs worse when comparing the move actually chosen from the opponent. This shows that a more dynamic approach is needed.

In search of a more dynamic approach we observe the *pr* of the move actually chosen from the opponent. In Figure 11.1 we see a histogram where the count of each pr value is plotted. Excluded from the chart is the value 1, which have a count of 75 in this test run, recall a *pr* value of 1 means all nodes have the same $P(e)$, which means the feature scores of the board all belong in the same state in the DBN. We see that some values are much more frequent than others. A reasonable approach is to assume the opponent is going to choose a move which have a *pr* value already seen before.

By keeping track of which values have been selected previously by the opponent we can create a heuristic to use for determining whether a node can be deemed as most likely. We refer to this as the "‘seen before heuristic"’ as a function seen_before(pr), where $pr \in \{p \mid p \in R \wedge p > 0 \wedge p <= 1\}$ that returns true if *pr* has been selected by the opponent at least one time before, and *pr* is smaller than 1. This is the implementation of the mostlikely() method described in the asymmetric game tree algorithm.

Tests show that using seen_before() have about 50% that the chance of prediction is the correct move and on average approximately 5 nodes satisify seen_before() when the prediction was correct. In this game the branching factor is usually in the range 10 : 11, so eliminating the descendants of 5 nodes from the search space is a significant reduction.

11.2 Improving Performance

Now we put all the pieces together and test the performance of the asymmetric game tree. We make the same tournament as described in Table 10.2, but now the black player is using the asymmetric game tree. The asymmetric game

		White			
		1	2	3	4
Black	1	67%	67%(33%)	0%	0%
	2	33%	0%(67%)	0%	33%
	3	67%	100%	33%	67%
	4	100%	100%	0%	67%

Table 11.1: Balance of Power Between Chosen Strategies

black	white	black wins (minimax)	black wins (asym)
1	2	0%	67%(33%)
2	4	0%	33%
3	1	67%	100%
3	2	33%(33%)	100%

Table 11.2: Strategies which have Improved Performance by using Asymmetric Game Tree

tree is tuned s.t. the number of nodes searched is roughly the same as the vanilla minimax algorithm. Also recall from Section 3.1 that the order or the nodes is very important for minimax search. To eliminate the arbitrary nature of the ordering of the children of a given node we choose to sort the children according the to value of a board according to the heuristic value function before proceeding to another depth in the search tree. This sort greatly reduces the search space.

The DBN in this test has 4 slices, because initial tests have shown that the number of slices has almost no effect on the number of correct predictions. Some of the values from the feature scores have many different values in our test data. DSum e.g has over 100 different values. We chose to specify the node representing DSum, OBSum, and BSum as interval nodes instead of numbered nodes as the rest. For the tests we have chosen a number close to 16 as the number of states for the interval nodes.

In Figure 11.1 we see the results after the asymmetric game tree has been applied. This can be compared with Table 10.2. What is interesting to see is some of the strategies with really poor performance has improved quite a lot. Unfortunately some of the really good has decreased in performance.

The strategies which have increased performance can be seen in Table 11.2. What we measure is the percentage of black wins, with draws in parenthesis as usual. The third column was when playing the tournament with minimax.

black	white	black wins (minimax)	black wins (asym)
1	3	67%	0%
2	1	67%	33%
2	2	33%(67%)	0%(67%)
2	3	33%	0%
4	3	67%	0%
4	4	100%	67%

Table 11.3: Strategies which have Worse Performance by using Asymmetric Game Tree

In the fourth column we have applied the asymmetric game tree for the black player.

The first interesting result is for black=1, white=2 which went from a 0% wins to now a 67% win(33% draw). Which meant it didn't lose a single match against the strategy which consistently won over it. The next line also improves an 100% losing match, to a win by a 33% chance. The results pretty much speak for them selves.

Additional Stats

From the same data set from extract some additional interesting statistics.

Lets start by looking at how many nodes was in the search space when using the asymmetric minimax. Surprisingly the counts were really close. Using a default depth of 6: The minimax with alpha beta pruning had an average of 2670 nodes for one search. The asymmetric game tree also with alpha beta pruning used slightly less, that is 2673. This was a surprising result given we newer search below the default depth, unless alpha beta pruning prunes some branches.

When looking at the time used the results are less encouraging. Minimax with alpha beta pruning used on average 0.197 seconds to search to depth 6. The asymmetric version used on average 0.873.

The number of correct predictions was 2007 out of a total 5520 predictions. Sometimes the algorithm is not able to make a prediction. This happens when all children has the same relative probability, and in the beginning of the game, where no *pr* values are seen.

Chapter 12

Conclusion

In this project we have worked with AI players in a simple board game. We wanted to see if we were able to predict some of the moves made by the opponent. We used the minimax search and a weighted sum of a number of feature scores to implement the AI players.

We wanted to recognize the strategies played by the opponent. To have some strategies we first searched for those using a genetic algorithm which was able to find us four different strategies which we try to predict the next move in an online game, based on observations of the behavior of these strategies from a number of offline games. Of the four found strategies, none of them are total losers, although strategy four seems stronger than the rest.

We tried predicting a move from the opponent. Roughly 50% of the times, the prediction was correct, and often we were able to search to depth 8 in an asymmetric game tree as opposed to depth 6 in a normal game tree. The nodes to search in depth 8 was determined by predictions based on observations from offline games.

The number of nodes searched using asymmetric game tree were smaller than regular game tree on average, even though we also searched to depth 8 on occasions and never explicitly decreased the search depth. If there is no prediction available for a given set of siblings or they all have the same relative likelihood, we don't even try to predict. This means it can't be the explanation of the smaller search space. What is likely to happen is that a node found at depth 8 has a value s.t. alpha beta pruning is able to prune some more branches it would not have been able to otherwise.

We saw how the asymmetric game tree improved performance for 2 out of 3 of the players which had an opponent over which a win was unlikely. We also saw how the some strategies suffered from worse performance when using the asymmetric game tree. This is especially true for strategy 4, which is really strong. Even though we never explicitly reduce the search depth of the game tree, we assume that the move of the opponent is in a given set. If this assumption is false, we would likely have taken a different move. Over all the asymmetric game tree improves performance significantly for poor performing strategies.

Chapter 13

Future Work

In this project we chose the slice of the oobn to be identical for both black and white player. A different approach is to use a different slice for both players. Two slices with the same structure, but trained with different values during EM-learning. Such an approach could possibly improve performance.

A comparison of the following approaches would also be interesting.

- One-slice-histogram. The approach we used in this project.
- Two-slice-histogram. Like one-slice histogram, but with different probabilities for the slice representing black and white player.
- One-slice-threshold. Using the same slice for black and white, but requiring *pr* to be below some constant MAX_PR
- Two-slice-threashold. Like above, but using two slices.
- Random. Instead of using $P(e)$, use a random number.

In theory the most likely board configurations will also have a higher frequency of high *pr* values. A direct comparison of the five methods listed, and the *pr* actually selected by the opponent is also interesting.

Bibliography

[Gee] Zong Woo Geem. Music-inspired optimization algorithm harmony search. http://www.hydroteq.com/HS_Intro.pdf.

[JN07] Finn V. Jensen and Thomas D. Nielsen. *Bayesian Networks and Decision Graphs*. Springer, 2007.

[Jun] Fore June. Game trees. http://www.webkinesia.com/games/gametree.php.

[Kal07] Dimitris Kalles. Measuring expert impact on learning how to play a board game. 2007.

[Lin] Yosen Lin. Game trees. http://www.ocf.berkeley.edu/~yosenl/extras/alphabeta/alphabeta.html.

[Yud] Eliezer Yudkowsky. An intuitive explanation of bayesian reasoning. http://yudkowsky.net/bayes/bayes.html.

Listings

List of Figures

List of Tables