An Artificial Intelligence approach to Dots-and-Boxes

- Master's thesis -

(Multi-Agent systems)

Gerben M. Blom, 1088777
blom@ai.rug.nl

Ambonstraat 24a
9715 HD Groningen

Supervisor:
dr. L.C. Verbrugge (Kunstmatige Intelligentie, R*u*G)
Second Evaluator:
prof. dr. L.R.B. Schomaker (Kunstmatige Intelligentie, R*u*G)

Kunstmatige Intelligentie
Rijks*universiteit* Groningen

Abstract

1 Introduction

The game of Dots-and-Boxes is a simple pen and pencil game, mostly played by bored children in the back of a classroom. Although the rules are simple, the game harbours a wide variety of ideas and possible strategies (see Berlekamp, Scott, Mang, Rhoads and Vardi). In this thesis I analysed and implemented a selection of these strategies. Dots-and-Boxes is a finite two-player zero-sum game of perfect information. Although it is a finite game, due to the scalable size of the playing field it is not possible to calculate all possible moves of a boardsize over 6×6 dots. Therefore I propose a human-like, rule-based approach to the game. The features of this human-like approach need to be robustness (the ability to apply rules in game positions of all boardsizes), speed (make up moves within a reasonable amount of time), an abstract representation of the game situation (description in terms of structures) and an abstract rule base (containing strategies for different types of game situations). These abstractions lead to a dynamic description of the way the game is played based on different stages in the game.

2 Dots-and-Boxes

The game of Dots-and-Boxes consists of a simple set of rules. It is played on a $n \times m$ grid of dots. The players take turns to connect two horizontally or vertically adjacent dots (an *edge*). If one of the players fills a *box* (makes the fourth connection in a square of dots), that player has to mark the box and he *must* make another connection. When all boxes are filled the game ends and the player who marked the most boxes wins.

In order to play the game in a human-like manner, the game-playing system must be able to 'see' the playing field at all times. Also it needs to have some sort of abstract interpretation of the playing field. Therefore the playing field is described in terms of *structures*. Structures are sequences of connected boxes (sharing a free edge) that have a valence (the number of free edges in a box) of two. The playing field is generally described in terms of the structures *chains* (connected boxes), *loops* (chains ending up in the same box) and *joints* (a box connecting multiple structures).

The strategies found in the analysis apply to three different game stages: the opening, the middle game and the endgame. The endgame covers the stage in which no free edges remain which do not give boxes away. In this stage the player in control (the one not giving up the first box) takes all but two (or four in case of a loop) from the offered structures to stay in control of the endgame. Unless giving up the control will win the game, the player in control is most likely to win since he has the ability to give up the control over the game. The middle game stage is covered by the Nimstring calculator which will try to gain the control over the endgame. There will be a heuristic monitoring the amount of time the Nimstring calculator

will use. In the opening stage the game playing system still makes random moves. Still, for opening play I presented a couple of ideas concerning machine learning as a tool for generating general rules for opening play in a Dots-and-Boxes game.

3 Implementation

The Dots-and-Boxes game playing system is implemented in Perl and the Graphical User Interface is implemented as a public website (`http://www.ai.rug.nl/~blom/dots/`) in *cgi-script* which is based on Perl. The Nimstring calculator implemented by Mang and optimalised by Rhoads has been integrated and slightly adjusted to give the desired output to the game playing system.

4 Results and Conclusions

The final implemented system, which can be challenged on a public website, is able to distinguish middle game and endgame positions and act upon them with the appropriate strategy. With the help of a heuristic function the system predicts the time the Nimstring calculator will take based on features in the playing field. This way the Nimstring calculator will only kick in if it will calculate for a reasonable amount of time. The implemented game playing system plays optimally in the way it is implemented. The only way to beat it is to use meta-strategies in the stage when the system only makes random moves. This is why it cannot win against the alpha-beta search based program Dabble, but it will beat most starting players for a very long time.

5 Discussion and Recommendations

The most interesting part of the game that can be hugely improved is the opening. With help of machine learning techniques it might be possible to create general rules for opening play on the base of abstract descriptions of the game situation. Abstract descriptions like present structures, taken boxes, the number of structures, the player to move and the number of moves on the side of the playing field seem better input features then just the edges of the playing field. Also the machine learning should distinguish between different stages in the game, it should not learn from the endgame that giving up boxes (which might be forced) is normal.

Contents

Chapter 1

Introduction

Due to my interest in logic and especially in applying logics to games, I wanted to try and find new insights in this area. I was very happy to find out about a game that can be described with a variety of techniques from Artificial Intelligence: the game of Dots-and-Boxes. Because the game is fairly easy to play (just a few rules and a simple setup) it is not so hard to calculate all possible moves. My interest starts when the size of the field increases, because then it takes too much time and memory to search the whole game tree. At this time only the winning strategies up to the 6×6 and 5×7 field have been completely calculated. This is why it is very interesting to look for possibilities to describe the game with general rules and start playing the game in an Artificially Intelligent way.

The game of Dots-and-Boxes is a relatively easy game played with paper and pencil. Most children learn the game in school when they are sitting bored in the back of the classroom. Just like 4-in-a-row and tic-tac-toe the game is about forming structures in an $n \times m$ 2-dimensional field, which, in the case of Dots-and-Boxes, is a rectangular grid of points which have to be connected. For a complete overview of the rules of Dots-and-Boxes see section 3.1.

During the course of the game it's possible to distinguish three phases in the game, the opening, the middle game and the endgame. The parts of the game that can come up with the most new information are the opening and the transition to the endgame (middle game). To learn about new insights in these parts we raise a couple of questions:

- How do we recognize an endgame?
- Is it possible to reach a winning endgame, and if so, how?
- How are existing endgame strategies applicable to earlier phases in the game?
- How is it possible to fool your opponent with meta-strategies?
- What is the influence of random opening moves on the game?
- Is it possible to get insight in new strategies with Machine Learning?

Besides the answers to these questions it is necessary to formalize existing theories on the game and work out a perfect handling of the endgame.

1.1 Research question

The main part of this thesis will focus on the analysis of the endgame and middle game of the game of Dots-and-Boxes. The object of this analysis is to generate a rule-based (human like) approach to playing the game of Dots-and-Boxes. Due to the human like objective of the system I state a couple of restrictions on the system. The system needs to be rule-based, indicating flexibility in different stages of the game, the system needs to make moves in a reasonable amount of time (say 5 seconds) and the system should be adapted to a playing field of any size (robustness). After the analysis I will present an implementation of a Dots-and-Boxes game playing system that in the first place will be able to recognize an endgame and handle it as well as possible, secondly it should be able to play in such a way that a good ending will be reached.

Besides the analysis and implementation of the endgame and middle game of Dots-and-Boxes I will present a couple of ideas about the way machine learning techniques can be used to create new insights in the way the game can be played in the opening stage.

To create such a system we need to find an answer to some questions and there needs to be a solution to some practical problems:

1. How can we formalize the endgame?
2. To what extent is it possible to analyze the problem at hand in to parts, and what is the optimal size of those parts?
3. Can strategies from the endgame be generalized over the whole game?
4. What kind of meta-strategies are possible to use?
5. How do we evaluate the performance of the system?

If we find solutions to these questions we are able to implement them into a game playing system.

1.2 Methods

The implementation of the Dots-and-Boxes game playing system will be done in Perl. To make it easy to test the final result I will implement some sort of graphical user interface in which it will be easy to make moves with mouse clicks. Because the user interface and the

gameplaying system have little interaction (just passing through the moves), the interaction will be integrated in the whole system through a public website implemented in cgi-script.

With help of available literature from Berlekamp a.o. [1, 2, 10, 19] I shall make a thorough analysis of the endgame of Dots-and-Boxes, after that the analysis will be formalized and implemented. The main goal of the implementation is to try to split the position dynamically and search through these abstract structures instead of trying to calculate all possible moves. This can be achieved by selecting possible future structures or by choosing different possible strategies.

From this analysis we have to subtract parts that can be useful to develop strategies to reach winning endgame positions. With the help of the Nimstring analogy ([12, 14]) the system will try to gain control over the endgame. At the same time I will formalize meta-strategies with help of the endgame analysis. Besides the formalization it is important to make an assessment about the reasonability of the amount time these strategies take to select a move.

I will make a selection from different machine learning techniques and show their features and performances in similar tasks. Based on that analysis I will present my opinion on the way machine learning techniques can help to find new rules and insights in the way the opening of Dots-and-Boxes can be played.

To make the game playing system as efficient as possible we shall look whether it is possible to split the reasoning process. How this splitting of the problem is done depends on current game positions in which one has to choose for the most optimal partition of the problem at hand.

At the end it is necessary to find a way to judge the game playing systems results. This can be done in different ways, by letting the system play to himself for a while, by testing the system against human players (i.e. on the internet), or by matching up against existing strategies like neural networks [11, 20], or parts of these strategies.

1.3 Goals

The goal of the implementation is that in the first place the final game playing system is able to play the endgame of any Dots-and-Boxes game perfectly. Secondly it is important that the system will be able to gain control over the endgame using the Nimstring analogy (within a reasonable amount of time).

1.4 Scientific relevance for Artificial Intelligence

The field of Multi-Agent Systems focuses for the main part on the possibilities of modelling and formalizing games. Through this approach we can get insight in human reasoning. So the approach in not just about calculating all possible moves and positions, but it is about searching for strategies humans might use to play games.

The most interesting new insights we are trying to reach from this research are the ones that should arise from the hybrid structure which is the basis of the gameplaying system. The combination of a connectionistic Machine Learning technique and a rule-based system should prove to be very interesting.

The ultimate goal is to make a robust implementation of a Dots-and-Boxes player which will play the endgame optimally and who possesses some relevant strategies to get through the opening moves without playing nonsense moves.

This research is in a certain sense a continuation of the work that has been done at the Artificial Intelligence department of this university on game theory. In the past a lot of people have done research on games like the research on Cluedo by Hans van Ditmarsch [4], the research on Kwartet (Happy Families) by Fiona Douma [6], and the work on games of imperfect information by Sjoerd Druiven [7]. On the side of machine learning this thesis links up with the work by Reindert-Jan Ekker [8] on reinforcement learning in the game of Go.

Chapter 2

Properties of intelligent game playing

In Artificial Intelligence there has been a lot of success in developing programs that are able to play games at an equally high (sometimes higher) level then human players. Some of these games, mostly the more simple ones like tic-tac-toe and four-in-a-row, have been *completely computed* during the research after these games. Because the *winning strategies* are known, this class of games is no longer of any interest to game researchers. The only use for this type of games is to serve as a test case when researches try new techniques to try to solve more difficult games, because all strategies are known. But still there is a class of games which includes i.e. checkers and chess, of which the best computer programs play only at grandmaster level. And in very difficult games like go, computers only play at amateur level.

2.1 Game properties

In this section I will go deeper into what a game is, how games are played and what differences arise between several classes of game that has influences on the way the game is played.

The set of games we know can be divided into several categories which distinguish between some very important properties. If games differ in some of these properties the approach of playing or solving may differ very much. In the following sections I will set out and explain some of the most important differences between several types and classes of games. After that, to make a clear distinction between what is allowed in a game (the rules) and what is the (best) way to play a game (the strategy) I will discuss their differences. Finally I will go into the consequences of these observations and the way I used these properties during this thesis.

2.1.1 *(Im)*perfect information

The first distinction between several classes of games we can make lies in the amount of information about the game that is available for all players. We make a distinction between two types of games:

1. **Games of *perfect* information**: all games where all players have access to all available information about the game. This type includes games like chess and four-in-a-row (both players see the board and all pieces each and every move).

2. **Games of *im*perfect information**: all games where all players do not have access to all information about the game. This type of games includes lots of card games like poker and bridge where players have their cards hidden for the other players and they need to make a guess about what the others are holding.

The difference between the way games of these different classes are played or decided has lots to do with chance. Strategies for games of imperfect information are in most cases focused on ways to collect information about the things you don't know. In games like bridge, players learn very much about the way the cards are dealt in the bidding stage. But still there will be lots of uncertainties about the game. In games like poker, players use the fact that nobody knows anything about how the cards are dealt by trying to bluff their opponents out of the game. On the other hand in games of perfect information all players see the complete game situation and they all know this; it is even common knowledge.

2.1.2 Zero-sum games

Another important distinction between types of games is the way the chance of winning is distributed over all players. In this distinction we also notice two classes of games:

1. **Zero-sum games**: the class of games in which the distribution of chances is equally distributed over all the players. This means not only that all players have the same initial chance of winning but the chance of losing is equally distributed over the opponents. Games included in this class feature chess and checkers.

2. **Non-zero-sum games**: the class of games in which the distribution of chances is not equally distributed. We can think of all sorts of casino games here where the bank (on average) always will make profit from playing a game, so one of the players will always gain more if the game is played often enough. Other types of non-zero-sum games include games where the win of one of the players does not implicate the loss of other player(s), here we can think of "win-win" situations in business negotiations.

Most games studied in game theory are zero-sum games. The interesting part of studying zero-sum games is that the chances of winning are equally distributed. Player have to find some clever strategy to win. This might be some meta-strategy or a psychological trick which might lure the opponent into a lost position.

2.1.3 Multi-player games

Developing strategies for certain games always takes into account the number of players that participate in that particular game. There are several types of games to distinguish here. The

single-player games are played alone and do not have to deal with possible moves of opponents. The multi-player games can be divided into games where everybody plays against each other like in poker, or team games where two or more players team up against another team or in special cases games where coalitions can form and dissolve dynamically. In these cases it obviously is important to evaluate the possibilities of opponent or partner players.

2.1.4 *(In)*finite games

The last distinction we make between games is based on the way games are decided or ended. There are types of games that have rules which would in theory allow them to play on forever without ever finishing. Chess used to be a good example of an *infinite* game since it is possible to make the same two moves over and over again. In modern chess however the moves are altered in such a way that the amount of repeating moves is limited.

The class of *finite* games includes games like four-in-a-row and tic-tac-toe. These are games with a finite amount of possible moves in their playing field. In the game of four-in-a-row every move fills one of the finite amount of holes so the game can not last longer then that (finite) amount of turns.

2.2 Finite two-player zero-sum games of perfect information

The class of finite two-player zero-sum games of perfect information is a subclass of the four different types of distinctions on games we can make. I consider this type of games to be the purest of all. In the beginning both players have the same chance to win, they will be able to know everything about the game during the complete match (common knowledge) and they will know that they both know this. This will lead to a clash of strategies and style approaches towards the game to be played in order to beat the opponent.

To go into this formally as described in [5] a game G, is a tuple, $G = (M, H, E, p, m, V)$. Here M is the set of all *possible moves* in the game, H the set of all legal move *histories* and $E \subseteq H$ the set of histories that can end up in a legal ending of the game. Then p is the *player function* indication which player is to move and the *move function* m the list of all legal moves after history H. Finally the *value function* V gives the resulting score for every history E. To determine the way to play we define a strategy for player p in game G as $s_p : H \rightarrow M$ which picks one of the moves out of m when player p is to move. If (as in a two-player game) both players follow their particular strategy they create a unique history $h(\mathrm{s})$ based on their pair of strategies $\mathrm{s} = \langle s_1, s_2 \rangle$. Based on this we can define a *solution* $\mathrm{s}^* = \langle s_1^*, s_2^* \rangle$ where there are no other strategies for both players which will increase their value for that game. In other words this solution defines the *optimal* strategy for both players.

Over the years several techniques like Minimax and Alpha-Beta pruning [15] have been developed to find solutions for all sorts of games. The drawback of this approach is that those

techniques need the complete histories H of the game they are investigating. In some cases this history is too large to calculate in a reasonable amount of time. This is also the case in a Dots-and-Boxes game larger than 6×6 or 5×7 boxes. Games smaller than these proportions have been *completely calculated*, a solution s^* has been found, indicating that all possible move histories ending up in a legal ending E have been evaluated and for both players their best strategies s_p^* have been determined based on the best possible moves of the opponent. In the case of a completely calculated game there is no need for general descriptions of game play since it is easier to save and follow all game histories with only the opponent's moves as variable.

Throughout this thesis I will not try to find the solution function for the game of Dots-and-Boxes since that will take to much time too calculate. In this case I focus on the way humans play games and try to make observations on the board which can be useful to create strategies that perform almost as well as the solution strategies observed by exhaustive calculation. Throughout this thesis the term *strategy* will not necessarily refer to the function over the game position and the game history or the strategies generated by complete calculation, but since the goal is to make a general description of the game a strategy will be a general description about how to play is certain sorts of game situations.

For simple or short games like tic-tac-toe almost everybody knows the one or two tricks that are possible in the game so most games will end up in a draw. In more difficult games, especially those which have not been completely calculated yet eventually the player with the best strategy will win. When both players use the same or complementary strategy they will need to search for a new plan to tackle the opponent's game plan. This battle for a win has many features and possibilities which I like to explore. I will do this by concentrating on the game of Dots-and-Boxes throughout this thesis. All rules and strategic features that the game harbours are discussed in the next chapter.

2.3 This thesis

One of the games that falls in the category of finite two-player zero-sum games of perfect information is the game of Dots-and-Boxes. It is a two player game played on a finite $n \times m$ grid of boxes on which players in their turns must make fill edges of the boxes until the grid is filled up completely. In the following chapters I will explain the game, suggest strategies about how to play it, and discuss an implementation of a Dots-and-Boxes game playing system.

The next chapter will deal with the properties and ideas about the game. In section 3.1 I will give a thorough explanation of all specific rules of Dots-and-Boxes. Then I will line out all properties the game holds from the way it is built up up to all different structures that will appear while playing a game of Dots-and-Boxes. After the explanation of all terminology I will discuss some ways in which we can use these structures to develop a strategy to play the game of Dots-and-Boxes through three different stages in the game. The way all properties and strategies are implemented in the system is explained in chapter 4 after which I will go deeper into the performance of the system and some recommendations for the future.

Chapter 3

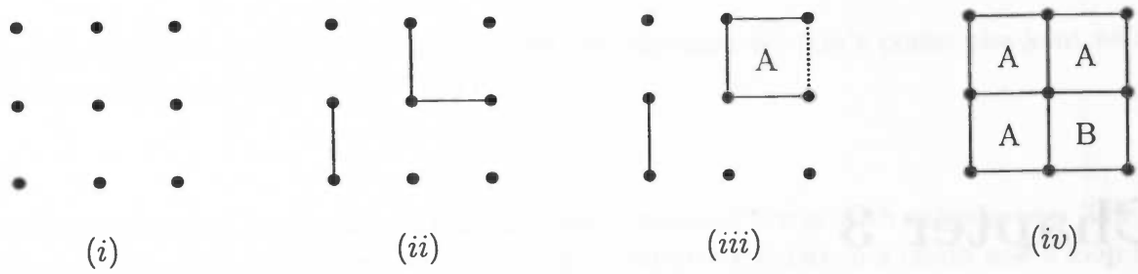
Dots-and-Boxes

The game of Dots-and-Boxes has a few very straightforward rules: connect all the dots, and try to gain as many boxes as possible by connecting the dots of the fourth side of a box. This game of perfect information (both players see everything there is about the game) is ideal for applying deep-search algorithms. The only problem is that when the size of the playing field increases, also the search space increases dramatically. So until today, only the playing fields up to 6×6 and 5×7 boxes have been completely calculated and the optimal strategy has been extracted [22]. Besides the complete search analysis, there has been some research in playing the game with the help of some neural network that should be able to generalize over some rules to find strategies which won't take as much space as the search programs do [20, 11].

Up till now the most interesting research on Dots-and-Boxes has been on he part of the game that is most easy to describe: the Endgame [1, 2, 10, 19]. Most researchers use parts of game theory in trying to describe the game as accurate as possible. In [1] a similar game to Dots-and-Boxes is described. This Nimstring game and the solutions to it give extra insights on the way to play the game of Dots-and-Boxes as smart as possible. These observations and theories are on game situations which arise in the latter stages of the game, which are called 'loony' positions. These arise when one of the players is in a position that he is forced to give up one or more boxes. I will take a deeper look on the endgame and loony positions in section 3.3

The research I'm proposing consists mainly of finding methods to realize winning endgame positions. I will try to generalize existing strategic rules about the game and when these rules are not applicable we will try to form new rules to reach better positions. See section 3.4 for a complete insight in the used methods and strategies. In section 3.6 I will discuss some of my thoughts about possibilities for opening play in the Dots-and-Boxes game.

Eventually the outcome of the complete analysis will be implemented and the implemented game playing device should be able to play a game on different board sizes. Therefore there needs to be a lot of generality in the strategic rules, so the system becomes robust and is able to find regularities in the different domains of different sizes.



(*i*) (*ii*) (*iii*) (*iv*)

Figure 3.1: Four typical 2×2 Dots-and-Boxes situations: (*i*) the startposition; (*ii*) the position after three moves; (*iii*) by playing the dotted line player 'A' takes the box in the upper-right corner; and (*iv*) the game is finished, player 'A' wins 3-1.

3.1 Rules

Before we can take a deeper look at the strategies available to play the game of Dots-and-Boxes we have to get familiar with the rules of the game. For an extensive explanation of the terminology used in this thesis I refer to section 3.2.1.

The playing field of Dots-and-Boxes is a pre chosen grid which consists of $n \times m$ dots (See figure 3.1 (*i*) for an example of a game of 3×3 dots). The players take turns and they have to connect two horizontally or vertically adjacent dots. (See figure 3.1 (*ii*) for an example where three connections already have been made). If a player fills a box (makes the fourth connection of a square), that player has to mark the box with his initial, and then he *must* make another connection (See figure 3.1 (*iii*) for an example in which player A marks the box in the upper-right corner after completing the square). If all boxes are filled with initials the game ends and the player with the most marked boxes wins (See figure 3.1 (*iv*) for an example in which player A has three boxes and player B only one).

3.2 Properties of Dots-and-Boxes

Before we take a closer look at the game of Dots-and-Boxes it is necessary to give an overview of all specific properties of the game. If we have a good understanding of all specifics and possibilities, it should be easier to make choices in later phases of the research concerning these basics.

3.2.1 Structures in Dots-and-Boxes

Because we don't want to make mistakes in the future to mix up different terms for different parts I shall present an overview of the used terminology throughout this thesis.

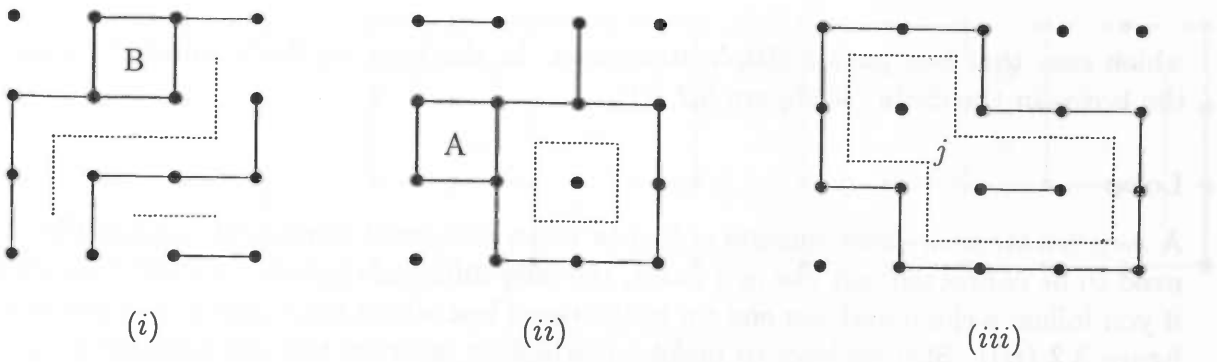


Figure 3.2: A couple of common structures in Dots-and-Boxes: (*i*) a Chain, (*ii*) a Loop and (*iii*) a Twin; the dotted lines show the connected boxes in the structure

Playing field

A *playing field* consists of a pre-chosen $n \times m$ grid of *dots* (see figure 3.1 (*i*)). These dots need to be connected during the game. In the end, all adjacent dots are connected and the game is over.

Box

A *Box* is a square of four dots which can be connected. Therefore, the playing field can also be described in terms of boxes (an $n \times m$ dots grid is the same as a $(n-1) \times (m-1)$ boxes grid). The *valence* of a box is the number of open edges in the square where it is still possible to connect two adjacent dots. A box with valence three or four, in which there is enough room for making a move, are called *joints*. What we see a lot in the endgame is that there exist joints in the playing field in which it is not possible to make a move without giving up a box (your opponent completes the last free edge). In these cases it is most likely that that joint connects multiple structures. We will now distinguish a few different kinds of structures.

Structure

A *structure* is a sequence of connected boxes with valence two. Two boxes are *connected* if they both have a valence of two (or more in case of a joint) and they share a free edge. The most common structures are *chains* and *loops*. Also several combinations of chains and loops are considered structures, some of them are mentioned here as well.

Chain

A *chain* is a structure that consists of three or more *connected* boxes with valence two, with the head and tail being a different box. The only restriction is that two boxes are connected if the edge that both boxes share is empty. A chain can end in a box that has valence 3 or 4 in

which case that box joins multiple structures. In this case we don't count the joint as one of the boxes in the chain (see figure 3.2 (*i*)).

Loop

A *loop* is a structure that consists of four or more connected boxes with valence two. The boxes need to be connected just like in a chain, the only difference between a chain and a loop is that if you follow a chain and you end up in the same box where you started, you are in a loop (see figure 3.2 (*ii*)). Still we have to make a distinction between the two because we will see that loops behave differently in game situations. If one of the boxes in a loop has valence three, the structure will not be counted as a loop but still as a chain which might end up in a loop.

Twin

A *twin* is a structure consisting of two loops that share a joint (see figure 3.2 (*iii*)). In general this structure will not completely disappear once one of the players gets the opportunity to take several boxes. It is most likely the first loop will be played like a chain and then only the second loop remains.

Dipper

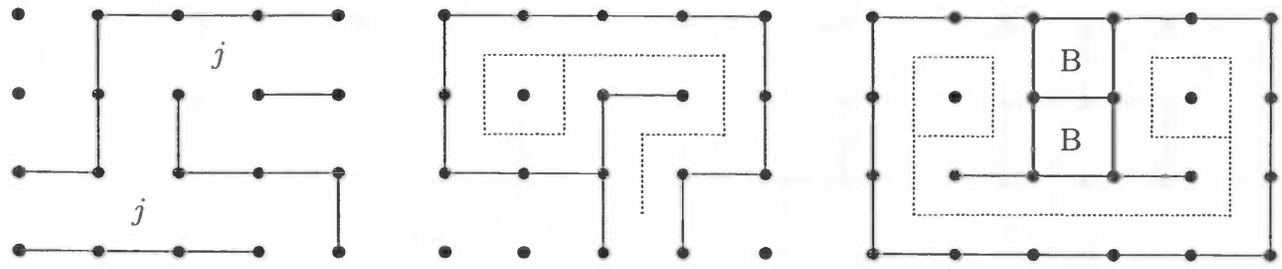
A *dipper* is a structure that consists of a loop connected with a chain (see figure 3.3 (*ii*)). Both structures are connected to a joint which in most cases will become part of the loop. The only way to avoid that this structure will disappear at once is to play the chain first after which the loop will remain.

Earmuffs

An *earmuff* is a structure consisting of two loops connected by a chain (see figure 3.3 (*iii*)). Since, as we shall see later, the player having to make the first move in a structure will gain the most from loops the earmuffs structure is most likely to be played as a chain and two single loops.

3.2.2 Types of moves

A move in Dots-and-Boxes is made by connecting two horizontally or vertically adjacent dots. Sometimes making a move has a direct consequence on the evaluation of the resulting game position. Here I provide an overview of all possible important special moves.



(*i*) (*ii*) (*ii*)

Figure 3.3: An example of (*i*) a joint that can't be played, (*ii*) a Dipper and (*iii*) an Earmuffs

Doublecross

The *doublecross* is a move which takes two boxes with only one move. This is possible after a double-dealing move (see figure 3.4 (*i*)). The double-dealing moves are usually played in the endgame to stay in control of the game (see section 3.3.3 for all insights on endgame strategies).

Handout

The *handout* is a move which fills (whether or not on purpose) the third free edge in a box, it hands out the box to the opponent who only has to fill the last edge to claim the box. Because in most cases one move has influence on multiple boxes (in most cases one edge belongs to two boxes or boxes are part of a structure) we sometimes have to deal with a double handout, or a series of handouts. We can distinguish three types of multiple handouts:

1. **Double-dealing move**: a move in a chain of length two which offers both boxes to the other player. Both boxes can be taken by making one single move, a doublecross (see figure 3.4 (*i*)).

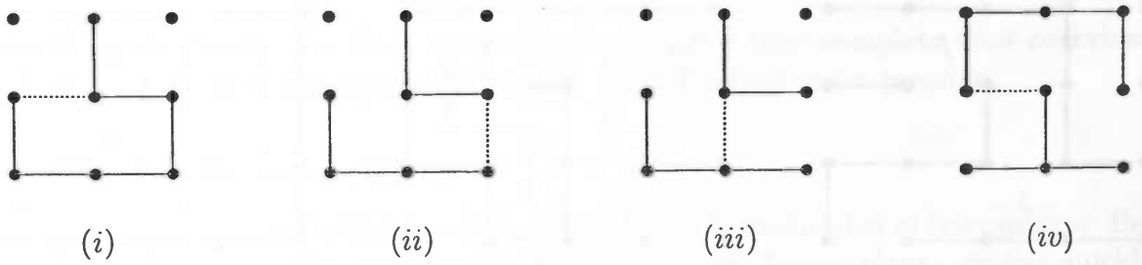
2. **Half-hearted handout**: a move in a chain of length two which offers both boxes to the other player (see figure 3.4 (*ii*)). The opponent now has the option of taking both boxes, or play the double-dealing move like in figure 3.4 (*i*).

3. **Hard-hearted handout**: a move in a two-chain which offers both boxes to the other player (see figure 3.4 (*iii*)). With a hard-hearted handout you prevent your opponent from making a double-dealing move the next turn.

There is one other very important handout move which we call 'loony'; I will define this type of handout now.

Loony move

Because in principle it is strange to give away boxes we call certain types of handouts *loony*. We distinguish three types of loony moves:



(*i*) (*ii*) (*iii*) (*iv*)

Figure 3.4: Four typical handout situations; (*i*) the double-dealing move (dashed), (*ii*) the half-hearted handout (dashed), (*iii*) the hard-hearted handout (dashed), (*iv*) a loony move (dashed).

1. Half-hearted handouts (see figure 3.4 (*ii*))
2. A move offering a long chain (at least 3 boxes, see figure 3.4 (*iv*))
3. A move offering a long loop (at least 4 boxes)

We call these types of moves loony because they give the opponent the opportunity to collect a certain amount of boxes and give the opponent the possibility to stay in control of the rest of the game. The latter is in sharp contrast with the hard-hearted handout which prevents the opponent from making a double-dealing move to stay in control of the game.

Although there are stages in the game where a player is *forced* to make a loony move we will still address to these moves as being loony.

A nice feature of loony moves is that loony moves in large structures are *equivalent* which means that an other move in the same structure would have the same effect and same outcome. As we see in figure 3.4 (*iv*) other possible moves besides the dashed move played do not change the situation or prevent the opponent from making a hard-hearted handout. In short structures this equivalence is not always present (look at the difference between the moves in figure 3.4 (*ii*) and (*iii*), the difference between a hard- and half-hearted handout in a 2-chain).

3.2.3 Isomorphism

The playing field of a Dots-and-Boxes game is an $n \times m$ grid of dots. This gives the game an important extra property, isomorphism. Because of the symmetry in the field, certain different positions are in fact equal and can be handled in the same way in future calculations. As we can see in figure 3.5 we can describe all possible mirrored configurations of a Dots-and-Boxes position with three simple rules. In the case of an $n \times m$ grid there are three other positions (figure 3.5 2-4) which are equivalent to the original configuration (figure 3.5 1). These equivalent positions can be obtained by the first two rules:

1. **Mirror in x**: a position is equivalent to the original position if the playing field mirrored in the x-axis is the same as the original.

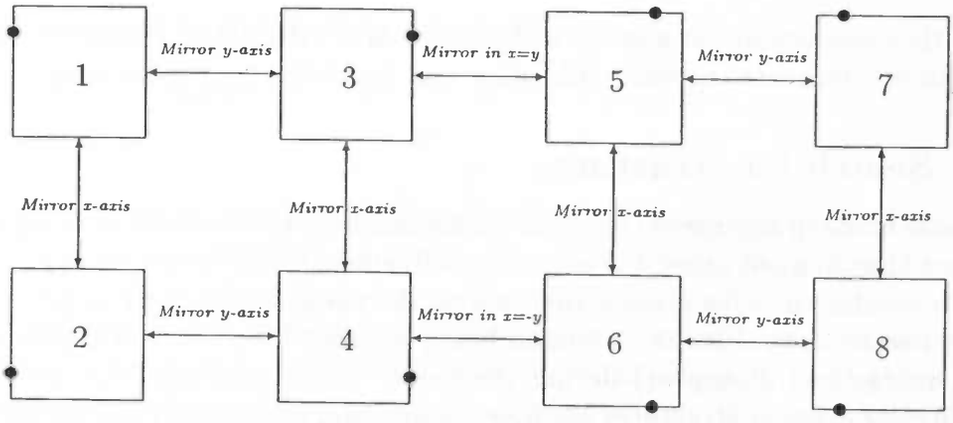


Figure 3.5: Representation of the mirror features in the Dots-and-Boxes grid.

2. **Mirror in y**: a position is equivalent to the original position if the playing field mirrored in the y-axis is the same as the original.

The first two equivalent positions (figure 3.5 2 and 3) can be found using either of the two rules, the third (figure 3.5 4) is found by applying both rules (it doesn't matter in which order). If we take a close look at the left half of figure 3.5 we see the three other configurations occur after applying the rules described above, as we see for the third configuration that it does not matter in which order the rules are applied, they give the same result.

In the case of a $n \times n$ grid we see a double amount of equivalent positions. Now we need a third rule to describe this class of equivalent positions:

3. **Mirror in x=y**: a position is equivalent to the original position if the playing field mirrored in the (x=y)-axis is the same as the original.

As we see in figure 3.5 the four new positions (figure 3.5 5-8) are the same as the four we already obtained for the $n \times m$ grid but mirrored in the x=y axis.

This is of course only possible if the n and m dimension are the same, if $n \neq m$ the grid can not be mirrored in the x=y axis.

In the end we can use the property of isomorphism to cut off leaves and branches from search trees if we find positions which are isomorphically equivalent to already found positions.

3.3 Endgame

The endgame of a Dots-and-Boxes game is very important for the eventual outcome of the game. Most people when they first play the game will try to make as few mistakes as possible (not play the third edge in a box) until absolutely necessary. In the endgame that arises players will offer the smallest structure, the other one will collect all available boxes and offer the next smallest structure until the complete playing field is filled. Because of the possibility of the

whether they are already in a loony endgame or not. After they complete their overview they can weight all structures against each other and make the best move possible.

3.3.4 Search for structures

Every move made in a game of Dots-and-Boxes can alter the evaluation of the position. Because of the fact that in most cases one move has influence on two boxes, things change quickly in a game. To be able to make general rules about the endgame we need to get a good overview of all present structures. Just like a human being we must keep track of the structures (how they develop, merge and disappear) during the game. After each move all possible consequences on the already existing structures are investigated and if necessary the structures are updated. This gives players the right information about the game at any time during the game (not only when it is their own move).

We can divide the originated structures into five main categories:

1. Chains
2. Loops
3. Joints
4. Handouts
5. Double Handouts

The structures in the handout category are not always complete structures, but the category is intended to keep track of all available handouts in the game. All possible doublecrosses are in fact two separate handouts, but because both boxes will disappear with a single move we keep track of them in a special double handouts category, which consists of all possible doublecrosses.

Because of the way certain positions need to be played we have divided the Chain category into four extra sub-categories:

1. 1-chains (chains of length one)
2. 2-chains (chains of two connected boxes)
3. long chains (chains longer than two connected boxes)
4. possible loops (chains of which head and tail are connected to the same (other) box, in most cases a joint).

To keep the game overview up to date it is most efficient to evaluate the direct consequences of a certain move on the existing structures. To recognize these consequences the boxes and the moves made on them contain several useful features:

1. New valence of the box(es) a move is made on
2. Part of existing structure
3. Connections

The *new valence* of boxes feature gives a nice indication of the way the game situation develops. After a move is made, the possible new valences are 3, 2, 1 or 0. In the case the new valence is 0, obviously the box is filled (there are no free edges left) and this particular box can be removed from all structures it was part of. This can be done without any consequences since the filled box was a handout before the last move. It could only be present at the head or tail of a structure (a box in the middle of a structure must always have valence > 1). Therefore it is not possible that the structure the box was part of is split up after the completion of the box. When the handout was part of a joint structure with two other chains attached to it the joint will disappear, but this will be done when the joint-box is updated (since both boxes connected to the edge are updated separately). The joint-box will receive a new valence of 2. See the rest of the explanation for what happens with boxes that receive a new valence of 2. If the new valence becomes 1, there is a creation of a handout. This has influence on the handout categories and sometimes a few structures the box was part of. If the valence becomes 2, it is upgraded to a structure. If the free edges are not connected to any other structure, the box becomes a 1-chain, otherwise it will be added to an existing structure. If the box is connected to two structures, these three structures are most likely to merge into one new large one. On the other hand the box could be the end of a joint which lost one of the connected structures. In all other cases boxes have valence three or four. These boxes can only be part of structures which contain joints, in which case they themselves act as the joints.

After the choice is made in which category of valence the move is, the structures to which the box now belongs need to be updated or created which sometimes makes it necessary for former structures to be deleted.

A very powerful feature is that of the available connections boxes have. Because it is still not possible to work in parallel, the two boxes in which an edge is filled need to be updated sequentially. That is why we have to be aware of moves which add both boxes to the same structure(s). If we don't take this possibility into account it might be possible for single boxes to be left out of structures they should belong to. Also we need to look at connections that might disappear after a move is made. The disappearance of joints can have huge impact on the evaluation of the game situation because the joint node can join two single chains and merge into one single longer chain. This can have some major consequences on the control of the game.

Once all structures are described, we can make a complete evaluation of the game position. Because there are no 'free' boxes left in an endgame players will have to give away or take the boxes which will gain them as much as possible in the end. The complete description of all structures is important because in general you will not give away single boxes, but by giving up one box, you are offering complete structures.

3.3.5 Evaluate gain *(in control)*

The evaluation of a (loony) endgame is represented by a single integer value $V(G)$, the *value* V of endgame *graph* G. We can calculate this value in the following way: suppose player A is in control, which means player B is out of control, then $V(G)$ is equal to the predicted number of boxes player A is going to take in graph G minus the predicted number of boxes player B will take. To make a good prediction about the number of boxes both players will take we need to look at the strategies both players will adopt during the endgame. When there are still 1- and 2-chains left the control of the game is most likely to switch from time to time. That's why we will focus on loony endgames here, where the player in control can choose to stay in control.

As we saw earlier the player in control of the endgame will try to stay in control so he can make a lot of profit from the long chains and loops available in the game. On the other hand, being out of control also has its benefits. Each chain gets the player out of control two boxes and every loop will increase his total by four. For short loops and short chains the total gain of player B is higher if player A wants to stay in control of the game. So in an endgame with very few long loops available the player out of control might win if the player in control tries to stay in control.

Now we can see all sides of the story, the player in control has the benefit of taking a lot of boxes from long chains, but the player out of control has the opportunity of making the choice which boxes are played first. So player B will play moves in structures which will gain him some points and prevents player A from scoring heavily (i.e. short chains or short loops). This way player B might be able to build up a lead in points and might not be caught up by his opponent. Still player A has the opportunity to give up the control if being out of control has a higher gain. The most important criterion for player A to decide whether he stays in control is the outcome of the predicted number of points he will receive if he stays in control, and how many he will get when he takes all he can and gives up the control over the game.

How does this work in practice? Every time player B makes the first move in a structure player A needs to weight two possibilities: how many boxes do I score if I stay in control and how many do I score when I take all remaining capturable boxes and give away the control. In general the total score of player A ($S(A)$) if he stays in control is the total number of boxes b left minus the score of player B ($S(B)$):

$$S(A) = b - S(B)$$

The total score of player B is the total number of loops l times 4 plus the total number of chains c times 2:

$$S(B) = 4l + 2c$$

Because chains are sometimes attached to joints j and these joints often merge into single chains with other connected structures, we need to calculate the total score a bit differently.

There can be two types of structures attached to joints. In principle only chains are attached to the joints, but some of these chains end up at the same joint. These types of chain are called

possible loops (see section 3.3.4), like the ones we find in dippers and earmuffs. As we shall see in the next section, players who are out of control will play joints with possible loops attached to them in a way that the possible loops will in fact become real loops, this way player B will gain the most from the given situation. So if a joint is connected to a possible loop and one or two chains, the chains will be played first and the loop last, which implies that these structures behave as isolated chains and loops. If a joint is connected to two possible loops, this can only be played as a chain and a loop, this way player B will gain two points less than if there were two independent loops. For every joint that has only chains attached to it one chain will disappear when only two chains would have remained. The two remaining chains will merge into one single chain including the joints they where connected to, which will reduce the player's score by two. During this calculation we have to make sure we only add structures which have not been included already (e.g. a chain which is connected to two different joints).

The last correction on the score of player B comes in the final stage of the game. When player B opens the last structure, player A will not play a double-dealing move in the end, but will collect all remaining boxes as a *bonus* since after taking the last box the game is over. As we will see in the next section in general this last structure will be a long chain which will reduce the score of player B by two points, but sometimes in case of i.e. a remaining dipper, player B is forced to play the loop as the last structure which will reduce his score by four points.

Now we can say the total score of player B in a loony endgame is the total number of chains c times two plus the total number of loops l (including possible loops) times four minus two times the number of joints j not connected to one possible loop minus the boxes player A will collect by filling up the last structure completely:

$$S(B) = 4l + 2c - 2j - bonus$$

3.3.6 Evaluate gain *(out of control)*

In the last section we have seen how the player in control can make the choice whether he wants to stay in control or not. Now we want to investigate what options the player out of control has to stand a chance in the endgame.

Of course the only thing the player out of control can do is making doublecrosses and opening new structures for his opponent. But he still has the choice *which* structure he wants to give up first.

As explained in [17] and [2], player B will gain the most if he delays the opening of profitable structures for player A, because otherwise player A will collect most of those structures and give up the control to collect the profitable structures for the player out of control. In practice this means that player B will choose structures which give player A a few boxes if he wants to stay in control, like short loops and short chains, or structures which give player B a lot of boxes if player A chooses to stay in control, like all types of loops (player A must leave four boxes to player B in a loop if he wants to stay in control).

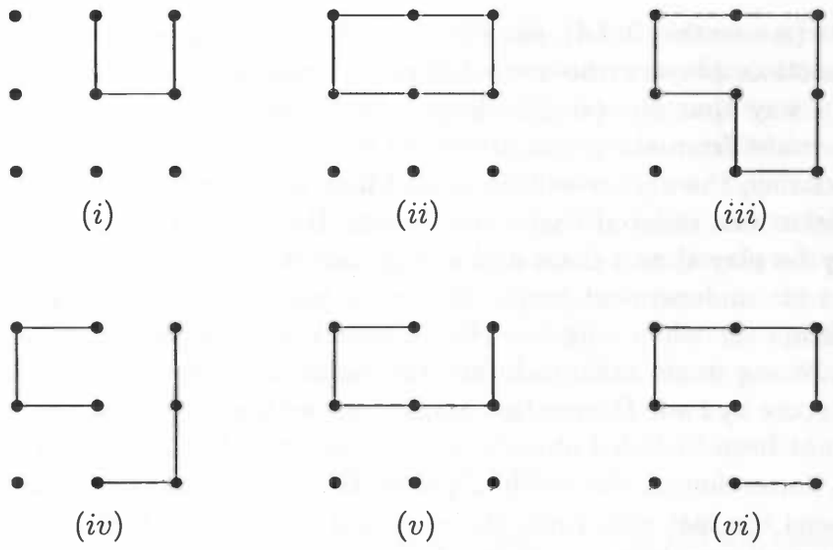


Figure 3.7: The six possible capture types; (i) a free box at the edge of the field, (ii) two free boxes with a doublecross, (iii) three free boxes and a doublecross, (iv) a free box in the middle of the field, (v) two free boxes at the edge of the field (half-hearted handout) (vi) two free boxes in the middle of the field (half-hearted handout)

1. Boxes that can be captured for free. There is no use in leaving these types of handouts to your opponent and another move will not force your opponent out of control (hard-hearted handouts, see figure 3.7 $(i) - (iv)$).

2. Boxes that can be captured but might offer the change to force your opponent out of control (half-hearted handouts, see figure 3.7 $(v) - (vi)$).

What we learn from this distinction is that taking boxes of the first category has no influence on the winning strategy. If there is a winning strategy for the rest of the game then you can take all handed-out boxes from the first category and follow that strategy. If you are in a position that your opponent has a winning strategy in the remaining playing field, then it is possible to *steal* his strategy when there are boxes available from the second category by making a double-dealing move.

Nimber

To find a winning strategy for the Nimstring game we can give any Nimstring game position a value. This value is a nonnegative integer number which we call a *nimber*. To start off we will state that the nimber of the empty playing field is equal to 0. As we have seen taking handouts of the types showed in figure 3.7 $(i) - (iv)$ does not change the strategy. So in a Nimstring playing field we call positions in which configurations of the first category appear

equivalent to the game situations in which those boxes have been taken. Since there is a *choice* for control in game positions where configurations shown in figure 3.7 $(v) - (vi)$ appear, we call those situations loony. In formal descriptions we will use the symbol $\mathfrak{D}$ for a loony position. These observations will lead to some sort of recursive description of a Nimstring game value. To complete the recursion we will use a function called the *mex* function

Mex function

Now we have the values of empty Nimstring positions and Nimstring game situations in which handouts appear. To calculate the value of an arbitrary Nimstring position we use the mex (minimal excludant) function. Given a list of nonnegative integer numbers, the mex function returns the smallest integer not in the list.

For example:

$$\text{mex}(5,12,23) = 0$$
$$\text{mex}(0,5,9) = 1$$
$$\text{mex}() = 0$$
$$\text{mex}(3,5,0,13,1,4) = 2$$

As we see in the third example the mex of the empty list is 0.

Since we have to deal with loony positions we also need to define de behaviour of the mex function with our special symbol $\mathfrak{D}$. When a $\mathfrak{D}$ appears in a list we can just ignore it since it leaves a choice to the player to move. As we see in figure 3.7 (v) and (vi) which have value $\mathfrak{D}$, there is always a winning strategy available since the player to move has the choice between taking the open boxes and moving in the remaining playing field and creating a hard-hearted handout with a double-dealing move to force the opponent to move in the remaining playing field. One of the choice of moves is the winning strategy. For example:

$$\text{mex}(3,5,0,\mathfrak{D}) = 1$$

Recursion

Now we can give the complete recursive definition of an arbitrary Nimstring position:

1. The nimber value of the fully filled field is 0.
2. The nimber value of a playing field containing one of the four handout configurations displayed in figure 3.7 $(i) - (iv)$ is equal to the subfield with the capturable boxes removed.
3. The nimber value of a playing field containing one of the two handout configurations displayed in figure 3.7 $(v) - (vi)$ is $\mathfrak{D}$.

But in large positions it is mostly impossible to calculate the nimber value within a reasonable amount of time. To tackle this problem we can state some rules which can help in the early stages of the game to gain some advantages. In this section we try to set out a few simple rules which might help to set up the game before things become calculable. These rules are partly based on the interaction taking place in almost every game between the strategies of both opponents. Strategies taking into account the way the opponent might play are called meta-strategies. This "taking into account" can be extended to higher levels since both players might base their strategies on the other so you are taking into account that your opponent is taking into account that you are taking into account what he is doing, ad infinitum.

The first thing is a mere observation as you can check that every possible Dots-and-Boxes game obeys the following formula, which we will adres to as **Rule 1**:

Rule 1: *# initial dots + # doublecrosses = # turns*

Make sure you notice the difference between the number of turns and the total number of moves made: a turn can consist of several moves and lasts until a non-taking move is made.

Most games of Dots-and-Boxes are won by the player who is in control. The player being in control will eventually win by taking most boxes of the long chains at the end of the game. This indicates that both players want to make the last move. This leads us to the next rules:

Rule 2a: The player who begins makes the odd turns, so in order to make the last move he wants the total number of *turns* to be *odd*.

Rule 3a: On his turn, the player who plays second makes all even moves and wants the total number of *turns* to be *even*.

This important feature is already important before the end, since when you are out of control of the game you don't have any influence on the amount of doublecrosses anymore. So in a game with an even number of initial dots the first player must create some sort of doublecross early in the game or try and prevent the opponent from making lots of long chains, making it less attractive to gain control over the game.

In most cases we see that the number of long chains present in the game is one more than the number of doublecrosses (since the player who takes the last long chain will not make a double-dealing move in the end). In this case the battle for control is won over the number of number of long chains, so players do not have to desperately search for a doublecross but may try to create or erase single long chains. So we can sharpen the rules a little bit and only apply them to creating chains instead of doublecrosses:

Rule 1: *# initial dots + # doublecrosses = # turns*

Rule 2b: The player who begins makes the odd turns, so in order to make the last move he wants the total number of *initial dots* + the total number of *long chains* to be *even*.

Rule 3b: On his turn, the player who plays second makes all even moves and wants the total number of *initial dots* + the total number of *long chains* to be *odd*.

These three rules can help players coming up with moves in stages of the game with only a few moves made. It is difficult to formalize these rules into specific actions at specific moments in the game. Usually it comes down to a battle round the centre. The player who wants the number of long chains to be odd will try to force a chain through the centre preventing other (large) structures to arise. On the other hand the player who wants to have an even number of long chains will try to split that long chain in two, maybe even by sacrificing a centre box early in the game, preventing the one long chain through the centre.

3.5 Machine Learning

Machine learning is an area which covers lots of different topics. In general it is a way of gaining information about topics which are very difficult to understand. The main idea behind machine learning is that it easy to let a computer program or *agent* try a certain task many times. Initially the program does not try to perform the task well, but based on the behaviour of the environment it tries to figure out *how* the tasked can be performed. In other words the program is *learning* how to perform the task. This method makes it possible to learn tasks very quickly, only focused on the topics researchers are interested in, with only a few guidelines about the way it should adapt itself to finally learn to perform the task. Machine learning is a very useful tool in very complex or changing environments in which it is difficult to describe the desired behaviour of the system directly in a few rules.

Important well-known machine learning techniques include neural networks and genetic algorithms. Neural networks are often used to generate a system which is able to generate over some features that are present in the environment. Genetic algorithms on the other hand use strings of features and try to find which feature set fits best in the environment it must work in.

3.5.1 Machine Learning in games

Since the development of machine learning techniques they have always been involved in game research. As described in [13] and [8] neural networks can be helpful in the research in two of the most difficult and interesting games in game theory research, chess and go. In these cases

neural networks are trained on a set of features present in the game. The training is based on the assumption that the chosen features all have a certain influence on the evaluation of a game position at any time in the game. The neural network is used to find out, based on the current game position, what the influence of each of those features is on the evaluation of the position. In the training stage the influence of each feature is adjusted based on the performance of the game playing system.

There have also been different machine learning approaches to the game of Dots-and-Boxes. In [11] we see a neural network with as the input all edges on the board and an extra node indicating who is to move, creating the preferred move in that position. Testing this setup with different sets of hidden layer nodes on a 3×3 Dots-and-Boxes playing field showed very poor results. Against simple strategies that only complete handouts besides making random moves, it never scored higher then 10% and against slightly smarter heuristics even worst.

In [20] we see a different approach in the same environment. Weaver and Bossomaier use a fitness function to evaluate the performance on three different levels of a simple feed forward network on a 3×3 playing field against the same strategies Kronfol [11] used. After the evaluation a new generation of networks is produced which will play the heuristics again, after which this generation is evaluated again and a new population is grown. As we saw in the neural network case of Kronfol, the genetic algorithm also never scored over 10% against the most simple heuristic.

The main problem with both previous machine learning approaches to the Dots-and-Boxes game is that they both use the edges of the playing field as the input layer. This approach has a few drawbacks. Because of the property of isomorphism lots of game positions are equivalent. This could cause the networks to generalize too much over the game since the networks might 'learn' differently from equivalent positions. Also since as we have seen the way Dots-and-Boxes should be played is not 'locally', based on individual edges. The way Dots-and-Boxes should be played is mostly based on the complete character of the game, the number of structures in the field and the way these structures are configured or interact.

A neural network approach based on heuristics about the available structures should be a lot more useful.

Besides the machine learning techniques there are other established search techniques like Minimax and Alpha-Beta pruning [15, 5] that can be useful, maybe in combination with machine learning, in the search for the best moves in games and especially in the case of Dots-and-Boxes in the research of opening play.

3.6 Opening

As said in the introduction of section 3.4, the transition from the opening to the middle game is somewhat vague. The strategies outlined in section 3.4.3 are in general also useful in the opening. The only difference is that there are lots of places on the playing field where very few moves are made, and it is difficult to make a judgement about the way that particular section

will evolve. One obvious solution is to try to calculate your way out of trouble. The main problem of a deep calculation in the opening stages is that almost all options and possibilities are open, so the search space will not be restricted or limited by obvious silly moves. The rules to be used in the opening need to be some sort of guideline how one tries to set up the game in such a way that one can profit towards the endgame.

Beside the rules described in section 3.4.3 it is very difficult to follow other rules about the way one should play in the early stages of the game. There is no evidence that it is useful to make a lot of moves in the centre or the complete opposite. The only really useful method is cutting off long chains if you are forced out of control and trying to extend chains when you are likely to gain control.

Now I will outline some ideas of mine which might be very interesting to implement and maybe will generate more general rules about the way the opening of a Dots-and-Boxes can be played.

3.6.1 Generating General Rules

From different proposed machine learning techniques in section 3.5 it might be possible to derive general rules about the way the opening of a Dots-and-Boxes game must be played. Because of the methods these techniques use, it is difficult to predict how these rules appear in the way such a technique chooses his (opening) moves. But as we have seen in section 3.5.1 we need to make some adjustments to the way these systems are implemented. We can think of two types of systems to solve our opening problem: a system completely based on a neural network solution or a system including search techniques.

By training a neural network on the opening of a Dots-and-Boxes game, using the computable evaluation of the endgame or end of the middle game as a reference, it might be possible to find new general rules for opening play. Still it would be very difficult to derive general rules from the way the neural networks behaves. Since it is a generalization over the complete game it would still be the question whether or not you really found a rule in particular behaviour of the trained network. But if it would win all the time by building the same sort of structures, or making moves on the outside of the field, who are we to question it! The main difference with [11] and [20] is that I suggest a method based on an evaluation at the start of the endgame, and not based on the outcome of the game. The evaluation at the start of the endgame (which is easy to check) is enough to distinguish between a 'good' and a 'bad' endgame. During play without any knowledge about the endgame there is to much that can go wrong in the endgame in the training stage of the network. Training on the endgame should be avoided while investigating the opening since the way the endgame is played is totally different from the way the opening is played. For example, when a player loses control he is *forced* to give away boxes in the endgame, but a system should not learn from the endgame that it is normal to give away boxes in general, let alone in the endgame.

We can think of two different types of networks here, those with only the most basic features of the game as input, one node per edge in the playing field to indicate whether that edge is

filled or not. On the other hand we can think of a more abstract representation of the game in terms of chosen characteristics that might occur in the playing field (e.g. # filled edges in centre, on the side, # structures, score etc). I would say that a choice for a network based on the characteristics of the game would perform a lot better, since there exist a lot of equivalent parts in the game so the game is not played 'locally' but most decisions are made based on how structures look like, how many there are etc.

Another approach is to start a calculation on the current opening position and evaluate the calculated positions of *all* possible moves after some reasonable amount of time. This evaluation is of course not easy to make since there is very little known about the game with very few moves made.

One way to create an evaluation of a difficult position is to create a heuristic function which takes into account all sorts of chosen characteristics of possible opening positions which might have influence on the outcome of the game. For example, how many long chains are there around, how many of them can be created, what is the current score or how many doublecrosses have been made, all sorts of characteristics which are important in strategies outlined in previous sections about the middle game (see section 3.4).

Using this somewhat barbaric approach, it would not be possible to generate general rules from the behaviour the system, since none of its actions are recorded and evaluated. Neither can the system be smart by disregarding 'stupid' moves immediately since handouts can sometimes be very useful in the opening of a Dots-and-Boxes game. The only way the search space can be reduced is by taking the property of isomorphism into account, and leave all mirrored positions out.

It is very time consuming to set the proportion of the effect that present characteristics have on the outcome of the heuristic evaluation function. Therefore it can be a good idea to make a machine learning system that trains to learn the optimal proportions for each chosen characteristic that has effect on the heuristic evaluation of the position. This way it should not be too difficult to set the values of each characteristic in the heuristic function. To evaluate the performance of the network it should be enough to look at the evaluation of the endgame since the network should only be trained on the opening. The tactics in the endgame are too different from the way the rest of the game is played that the network should not be misled in the training stage. Taking the evaluation of the endgame is a huge improvement over the approach used by [20] and [11] in training neural networks for playing Dots-and-Boxes. Still it needs a lot of research to find out which characteristics can play a useful role in the heuristic evaluation of a position.

The most important feature of the game of Dots-and-Boxes that must always be kept in mind is that the game has different stages which require different strategies. Therefore it is not said that one game theory approach is enough to describe the game. Maybe this game requires different approaches in different stages to describe the complete game. I think that a system consisting of a good search technique such as alpha-beta pruning with a trained evaluation function based on characteristics found in the game must be able to play a good opening game. After the opening, the Nimstring theory and endgame rules cover the rest of the game. All ideas

covered in this section about applying Machine Learning techniques on the game of Dots-and-Boxes will remain ideas for now. I had not enough time to implement any of those fascinating ideas that whirled through my head, so I kept the focus on the human like behaviour of the game playing system.

Chapter 4

Implementation

In chapter 3 the theoretical outline of how the game must be played was presented. In this chapter I will discuss the way I implemented all theoretical strategic rules in a Dots-and-Boxes playing system. The program is written in Perl. This is a programming language mainly based on scripts but with almost all possibilities modern program languages have. The main reason for choosing Perl is that the data structures used in Perl are very useful for manipulating lists and especially the special associated list data structures. Take a look at section 4.1 for a short introduction.

The implemented system consists of several different features that, combined together, create the game playing device. First of all there is the playing field that needs to be represented in some way. There needs to be a referee functionality that looks after the way the game is played: it monitors whose turn it is and keeps the score. And then there is the cognitive active part which needs to make up moves, and hopefully good ones. This part consists of several subparts. First there is a controlling part, which I will refer to in the future as `UpdateStructures`, which keeps track of every event in the field and updates the current knowledge as it is changed over time. It keeps a list of all available structures that arise during the game. This device is working all the time: just like a human being it stays working while the opponent is thinking about his next move. Then there is a part which is based on the number of boxes that are not included in a structure yet, an array `@free_boxes`, on the basis of which the decision is made which stage the game is in. Since this decision is easily made and only useful when the game playing system must select its move, it is a component of the last part. The last part will be called `SelectMove`; this part follows the strategic rules appropriate for the current stage of the game in order to come up with the next move.

In the following sections I will discuss all decisions that are made by the game playing system in different stages of the game. Since one can define the different stages in the game pretty clearly, this categorization as introduced in chapter 3 is also implemented in the system, and I will address the stages in different parts of this chapter.

4.1 Perl

The program language Perl [16] provides an environment somewhere between scripting languages and the large program environments like C and Java. Like the execution of scripts, Perl programs do not need to be compiled into an executable. But when the Perl program is executed it calls Perl in the first line (`#!/usr/bin/perl -w`), which compiles the program and then executes it.

For simplicity, Perl has only three different data types. The basic data type that can hold information are *scalar variables*. Scalar variables can hold any string of characters including numbers. So there is no difference between special data types like `int` or `float`: everything is stored in the same sort of data string. Variables containing scalar data are preceded by `$` (e.g. `$count`). The second data type are *array variables*. An array is a list which can store several elements of scalar data and the array variable is preceded by an `@` (e.g. `@list`). These elements are stored in a specific order. The array: `@list = ("Top","Right","Left","Bottom")` stores four scalar elements which can be accessed by their entry value. The access values in an array start with 0, so the scalar `$list[0]` holds the value `"Top"`, just as `$list[2]` is equal to `"Left"`. The last data type is the associated array or *hash variable* which stores data in its *value* that can be accessed by a *key* corresponding with the stored value. This is just like an array with the difference that elements can be accessed by a chosen key and not by a number that is set. For example: in a telephone book, all phone numbers and the person it belongs to are stored in an ordered fashion. If someone want to find someone's number one can look up that person's name and get the number one wants. You can say that the phone numbers are the values of the phone book hash, with the names corresponding to the phone numbers the keys that give access to the wanted value. The hash variable is preceded by a `%` and the keys are given between curly brackets (e.g. `%hash` can contain the (scalar) element `$hash{key}`, or like in the phone book example `$phonebook{name}== phonenumber`).

Another very useful tool in Perl are *regular expressions*. Regular expressions are very useful to search for patterns in scalar date without describing them to strictly (e.g. if all chains are given a name `Chain_#`, it is possible to access all chains by searching all structure names, filtering all patterns matching `Chain`). This is useful when one wants to search for a certain theme. This way one can store anything in the same place and sort or select them later. For the way regular expressions work I refer to [16].

4.2 Program overview

In the previous chapter we have seen how the game of Dots-and-Boxes must be played. Now I will outline how I implemented the suggested theory in a game playing system. In the program we recognise two parts. The *representation* of the board and all observed elements in the game are discussed in section 4.2.1. The way things are observed (`UpdateStructures()`), calculated (`SelectMove()`) and the actions taken on different observations or calculations are discussed

in section 4.2.2.

4.2.1 Data structures

The game playing system contains several representations for different features in the game. First of all there must be something that makes sure that both players obey the rules and make their moves when they are supposed to. Also there must be something to keep track of the moves made on the board to check for illegal moves and to keep track of the score of both players. This is enough to play a complete game of Dots-and-Boxes, obeying the rules as described in section 3.1, but since we also want to implement a game playing device, we need to keep track of a few other features of the game. The parts of the game that are common knowledge, the structures that arise during the game, must be represented in some way after each move. Just like any human player this is not something that needs to be updated just before the computer's next turn, but every time a move is made since everybody thinks all the time, not only when it's one's turn. The way these things are implemented is discussed in the following subsections.

Game representation

A game of Dots-and-Boxes is controlled by the `PlayGame()` function, that keeps track whose turn it is in the game and keeps the score. At the beginning of the game it prints the empty board and asks the first player to make his move. After the first move, it assigns the turn to the computer and asks its move after which it switches turns again. Now it checks after every move whether the turns must be switched and assigns the move to the player to move next. For both players the same protocol is used every turn. First the current playing field is printed (`PrintGrid()`), then the move is made. After the move is made it is added to the move list (`MakeMove()`), and updated in the playing field (`MarkEdgeOccupied()`) after which the structure list is updated (`UpdateStructures()`) by adjusting all structures to the new situation.

Playing field representation

The playing field consists of $n \times m$ boxes numbered $0 \ldots (nm-1)$. The playing field is stored in one hash variable `%Grid` with as keys the number of the box. This way the hash is used as an array, since more hashes are used as the values of the `%Grid` hash, it will be easier to work with when it is defining as a hash. The values in the `%Grid` keys are hashes which represent the edges of each box. The keys store the location of the edge in the box, `T`, `L`, `R` and `B` for top, left, right and bottom respectively. As we see in figure 4.1, this way some edges are stored twice since some edges are shared by two boxes (the middle edge is represented as `$GridName{"0"}{"R"} == "1"` and `$GridName{"1"}{"L"} == "A"`). This seems double work, but I found it useful to treat the edge for each box in a separate way. The values of these hashes store two types of

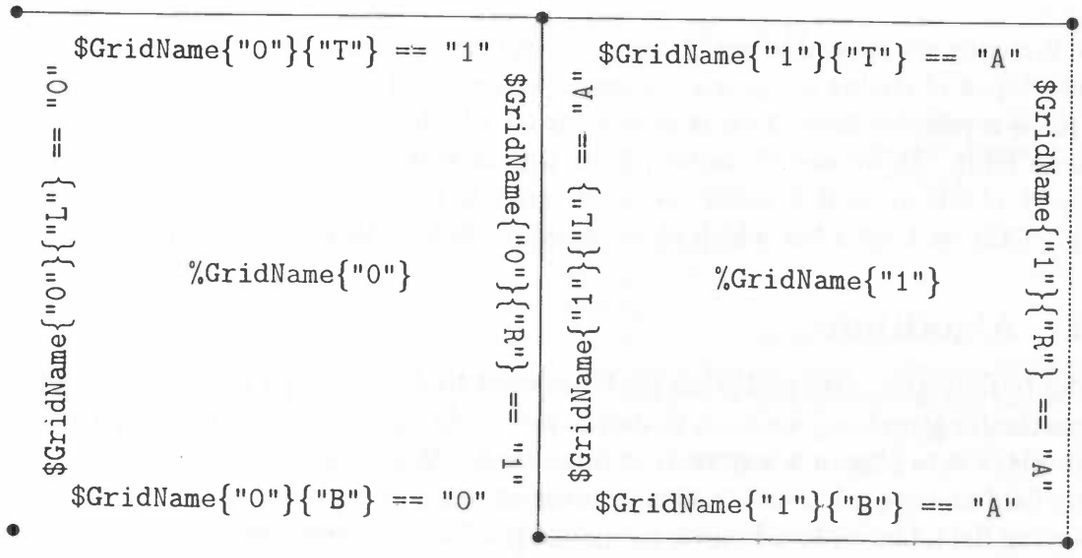


Figure 4.1: Representation of the playing field GridName after the last move by player A (dashed).

information. When the edge is *free*, it has value 0 and when it is filled it gets value 1, see the box labelled number 1 in figure 4.1. When the complete box is filled, all edges of that box get the value of the player who completed the box (see figure 4.1, all edges in box number 1 have value "A"). This way the box completion does not involve the edges of other boxes since the other boxes have an edge themselves. The advantage of this sometimes double representation is that all boxes have their own functionality. Moreover, the valence of the different boxes is a very important feature in the game and plays a major part in developing strategies.

Game structure representation

All structures that are discovered during the game are stored in a single hash list. The keys of this list are the different kinds of structures we distinguish. Each new structure gets its own key entry consisting of the name of the kind of structure followed by an index number that keeps track of the total number of current and past structures (i.e. Chain_13). We keep a list of the following different structures: chains of all lengths, loops of all lengths, joints, handouts and double handouts (see the list in section 3.3.4). The values that these keys correspond with are arrays containing the boxes that form that particular structure. In the case of a joint the array contains the joint box and the names of the structures that are connected to that joint. These are the major distinctions we find in a Dots-and-Boxes grid.

Because structures of particular lengths and shapes have important properties which have influence on the strategies we need to follow, we make a small refinement if we want to check for these strategies. The refinement is made in the Chain section (see the second list in section 3.3.4). Almost all chains behave in the same way, but there are three distinctions to make. The

1- and 2-chains are special types of chains since they can be offered as a hard-hearted handout, so these types of chains are stored in separate arrays. The third refinement is the structure I refer to as a *possible loop*. This is a chain from which the first and last box are connected to the same joint. As we saw in section 3.3.5 this means that such structures are most likely to be played as a loop, so it is useful to count possible loops as loops and not as chains. For each type of chain we keep a list which chain numbers belong to which type of chain.

4.2.2 Algorithms

In order to find structures and other parts we want to discover in a Dots-and-Boxes playing field or a particular structure, we need to define protocols. The object of the way the game playing system plays is to play in a way such as humans do. When people play a game they look at the playing field and see coherent structures automatically. Also, when they notice small changes in the playing field, like a played move, people do not look through the complete grid to reassemble all structures in the field, they only update the structures that are influenced by the change in the field. Besides this people have the natural ability to see isomorphism immediately, they are able to assign rotated playing fields as the same. Therefore these properties are common knowledge and must be available throughout the game. In the following subsections I will discuss the way the mirror function and the structure searcher are implemented.

Implementation of isomorphism

To check whether two playing fields are equivalent by means of isomorphism (see section 3.2.3) we need to develop a protocol which can fulfil such a task. The mechanism that checks for isomorphism is called `Compare_Mirror()` and is designed to compare two separate playing fields. The subroutine returns a value of 0 or 1 to give insight about whether both fields are equivalent. Since Dots-and-Boxes can be played on an $n \times m$ grid we have to make a distinction between two types of playing fields. The subroutine `Compare_Mirror()` we see on the top left in table 4.1 first checks the dimensions of the original playing field; if the number of boxes horizontally differs from the number of vertical boxes the subroutine `Compare_n_m()` is called (see the bottom left in table 4.1).

The subroutine `Compare_n_x_m()` deals with the fields of $n \times m$ dots with $n \neq m$. This type of fields has only four isomorphic fields for the same game position since the field can not be rotated (then the dimensions of the field are no longer the same). So from the field to be compared we find the three isomporphic fields by mirroring the field in the x-axis (`(%mir_x) = Mirror_in_x($name2);`), the y-axis (`(%mir_y) = Mirror_in_y($name2);`) and by applying both mirrors (`(%mir_x_y) = Mirror_in_x("mir_y")`). The `Mirror_in_n()` subroutines create a grid that is the mirrored version of the grid given as an argument in the direction the name suggests (either _x or _y) as shown in figure 3.5. These isomorphic (and original) fields are then compared with the current playing field (`CompareGrid($name1,...)`, bottom right in table 4.1) and if one of them is equivalent the function will assign both fields as equivalent.

```
sub Compare_Mirror {
  my($name1) = $_[0];
  my($name2) = $_[1];
  if($nmb_x == $nmb_y) {
    return Compare_n_x_n($name1,$name2);
  } else {
    return Compare_n_x_m($name1,$name2);
  }
}
```

```
sub Compare_n_x_m {
  my($name1) = $_[0];
  my($name2) = $_[1];
  local (%mir_x) = Mirror_in_x($name2);
  local (%mir_y) = Mirror_in_y($name2);
  local (%mir_x_y) = Mirror_in_x("mir_y");

  return (CompareGrid($name1,$name2)  or
    CompareGrid($name1,"mir_x") or
    CompareGrid($name1,"mir_y") or
    CompareGrid($name1,"mir_x_y"));
}
```

```
sub Compare_n_x_n {
  my($name1) = $_[0];
  my($name2) = $_[1];
  local(%rot) = Rotate_90_de($name2);
  return (Compare_n_x_m($name1,$name2) or
    Compare_n_x_m($name1,"rot"));
}
```

```
sub CompareGrid {
  $name1 = $_[0];
  $name2 = $_[1];
  foreach $box (keys %$name1) {
    foreach $edge (keys %{$$name1{$box}}) {
      if ((!exists($$name2{$box}{$edge})) or
          (($$name1{$box}{$edge}) ne
           ($$name2{$box}{$edge}))) {
        return 0;
      }
    }
  }
  return 1;
}
```

Table 4.1: Implementation of isomorphism

When the horizontal and vertical dimensions are the same, the `Compare_n_x_n()` subroutine is called. This routine deals with all fields of $n \times m$ dots with $n = m$. In this case the orientation of the field does not matter at all, so if we rotate the field by 90° the dimensions are still the same, and as we have seen in the previous chapter the amount of isomorphic fields doubles. So in case of a $n \times n$ field the `Compare_n_x_n()` (see the top right in table 4.1) compares next to the four isomorphic fields, the four rotated versions. These are obtained by `(%rot) = Rotate_90_de($name2)` as shown in figure 3.5. Thus, `Compare_n_x_m($name1,"rot"))` and `Compare_n_x_m($name1,$name2)` compare all eight fields with the original and will assign both fields equal if that's the case.

Implementation search for structures

One of the most important features in Dots-and-Boxes are the structures that arise during a game. The way the structures are arranged and the way they are built says a lot about the situation of the game and adding or deleting single boxes can change the evaluation of a position.

Therefore, it is important to have a complete overview of all structures at any time at any stage in the game. The way I implemented this overview is not really a 'search' for structures but more an update of all existing and new structures (see subsection 3.3.4 for the definition of the different structures).

Every time a player makes a move, the one or two boxes that contain the played edge are

updated in the playing field. For the overview of all structures present in the playing field this also happens at this point.

The way structures are updated depends on the nature of the move made. The structures are updated after each move, therefore the subroutine only has to look at structures that contain the played edge, or add the new structures which might be created by the played move. For a box in which an edge is filled we distinguish four possible outcomes after the move played. This outcome is closely related to the new valence, which has decreased by one, of the box. When a move is made the `UpdateStructures($GridName,$ListName,$box_nr,$edge_nr)` subroutine is called which updates the current structure list `$ListName` based on the current playing field `$GridName` and the played edge `$edge_nr` in box `$box_nr`. First it checks the valence of the box in which an edge is played (`@valence = FindValence($GridName,$box_nr)`), where the `@valence` array contains all free edges of the box `$box_nr` in playing field `$GridName`. Based on the current valence it is possible to predict the behaviour of the current structures, based on the new move. This gives the possibility to update the structure list without having to check for *all* possible changes in the structures.

A box has four edges so there are four different outcomes after an edge is played in a box (valence = 0...3). Each of these outcomes have different implications on the changes in the structure list. Here I present the protocol for the game playing system to follow when the new valence is known. Note that `$#array_name` gives the index number of the last entry in the array `@array_name` and in case of the empty list the value -1.

Valence = 0: When the valence of a box becomes 0, the box is filled. Therefore it is to be removed from all structures it was part of. There is no need to search for structures which might have been broken in multiple pieces. Since the box was a handout before the move played it is only possible that the box was at the end of a structure (in the case of a single or double handout the structures are emptied).

```
if ($#valence == -1) {
  foreach $structure (keys %{$ListName}) {
    if (InList(@{$$ListName{$structure}}, $box_nr)) {
      @{$$ListName{$structure}} =
        RemoveFromList(@{$$ListName{$structure}}, $box_nr);
    }
  }
}
```

Table 4.2: Implementation of Valence = 0

Valence = 1: A box which becomes of valence 1 becomes a handout. Now there are three ways structures can change due to the creation of a new handout. The most distinguishing property to check for is whether the free edge is a *stop* (either a side edge of the playing field or connected to a joint).

```
if ($#valence == 0) {
  $open_edge = pop(@valence);
  $next_box = AdjecentBox($box_nr,$open_edge);
  if (($next_box eq "F") or (IsJoint($nextbox))) {
    @{$$ListName{"Hand_Out_$nmb_handout"}} =
      AddToList(@{$$ListName{"Hand_Out_$nmb_handout"}},$box_nr);
    foreach $structure (InStructure($ListName,$box_nr)) {
      if ($structure !~ /Hand_Out/) {
        @{$$ListName{$structure}} =
          RemoveFromList(@{$$ListName{$structure}},$box_nr);
      }
    }
    last;
  } else {
```

Table 4.3: Implementation of Valence = 1 (i)

As we see in table 4.3, if it is the case that the box is a stop (`($next_box eq "F")`, there is no box connected to the free edge, or, `(IsJoint($nextbox))`, the box connected to the free edge is a joint) we distinguish a single handout. At that point a new `Hand_Out_n` structure is created with the next index number. After that all presences of the box in *other* structures are removed (most likely from a joint).

When the box is not a stop we move forward after the else-statement in the last line of table 4.3 to the next test, which we see in table 4.4. We know there is another box connected to the handout box. This leaves us two possibilities: the next box is a handout or the connected box is not a handout. So the next step is to check for the valence of the adjacent box (`@tmpvalence = FindValence($GridName,$next_box)`).

If the valence of the adjacent box also has value 1 we delete both boxes from all structures they where part of and add both boxes to a new double handout structure.

If the current box still fails this test we are left with a box with valence 1 (`$box_nr`) connected to a box with a valence other than 0 or 1 (`$next_box`). As we see in table 4.5, the system now does two things. First, it adds the handout box to the structure list as a single handout. Secondly, it creates a new chain structure, puts the handout box in it and adds as many connected boxes with valence = 2 to update the chain the handout is part of. There is one exception, namely chains larger than two boxes which begin and end with handouts. In that case the last box of valence = 1 is also included in the chain and is the last box in the structure in particular.

After the new chain is formed, older occurrences of (parts of) the chain are erased and when the chain has length 1 it is not saved either. In case a move splits a chain, both parts are accounted for, since the structure updates are done for all boxes in which an edge is filled. So the boxes on both sides of the filled edge will add all boxes on their side

```
@tmpvalence = FindValence($GridName,$next_box);
if ($#tmpvalence == 0) {
  foreach $structure (InStructure($ListName,$box_nr)) {
    @{$$ListName{$structure}} =
      RemoveFromList(@{$$ListName{$structure}},$box_nr);
  }
  foreach $structure (InStructure($ListName,$next_box)) {
    @{$$ListName{$structure}} =
      RemoveFromList(@{$$ListName{$structure}},$next_box);
  }
  @{$$ListName{"D_Hand_Out_$nmb_d_handout"}} =
    AddToList(@{$$ListName{"D_Hand_Out_$nmb_d_handout"}},$box_nr,$nextbox);
  last;
} else {
```

Table 4.4: Implementation of Valence = 1 (*ii*)

```
@{$$ListName{"Hand_Out_$nmb_handout"}} =
  AddToList(@{$$ListName{"Hand_Out_$nmb_handout"}},$box_nr);
@{$$ListName{"Chain_$nmb_chain"}} =
  AddToList(@{$$ListName{"Chain_$nmb_chain"}},$box_nr);
while (($next_box ne "F") &&
       (@tmpvalence2 = FindValence($GridName,$next_box))) {
  if ($#tmpvalence2 == 1) {
    @{$$ListName{"Chain_$nmb_chain"}} =
      AddToList(@{$$ListName{"Chain_$nmb_chain"}},$next_box);
    @tmpvalence2 = RemoveFromList(@tmpvalence2,SwitchEdge($open_edge));
    $open_edge = pop(@tmpvalence2);
    $next_box = AdjecentBox($next_box,$open_edge);
  } elsif ($#tmpvalence2 == 0) {
    @{$$ListName{"Chain_$nmb_chain"}} =
      AddToList(@{$$ListName{"Chain_$nmb_chain"}},$next_box);
    $next_box = "F";
  } else {
    $next_box = "F";
  }
}
```

Table 4.5: Implementation of Valence = 1 (*iii*)

of the old structure to their respective new structures.

Valence = 2: A box which gets a valence of 2 becomes a new structure. As we see in table 4.6, first the box is removed from the `@free_boxes` list since another move made in the box would leave a handout. After that all structures that are connected to either free edge are added to the new structure. At last, all structures that are added to the new

```
if ($#valence == 1) {
  @free_boxes = RemoveFromList(@free_boxes,$box_nr);
  foreach $edge (@valence) {
    $n_box = AdjecentBox($box_nr,$edge);
    if ($n_box ne "F") {
      push(@strlist,InStructure($ListName,$n_box));
    }
  }
  @{$$ListName{"Chain_$nmb_chain"}} =
    AddToList(@{$$ListName{"Chain_$nmb_chain"}},$box_nr);
  foreach $str (@strlist) {
    unless($str =~ /Hand_Out/) {
      @{$$ListName{"Chain_$nmb_chain"}} =
        AddToList(@{$$ListName{"Chain_$nmb_chain"}},@{$$ListName{$str}});
      @{$$ListName{$str}} = EmptyStructure(@{$$ListName{$str}});
    }
  }
  @strlist = ();
}
```

Table 4.6: Implementation of Valence = 2

structures are removed from the structure list since they are merged into a new structure.

Valence = 3: If a box gets a valence of 3 after a move is played it is treated as if it is a free box in which never a move is played.

After a move is made and its direct consequences are updated in the structure list, the list of joints and free boxes needs to be updated. Boxes in which no move has been made can become joints when (all) neighbouring boxes are part of a structure. After the selection on the valence of the played box is done, the joint list is emptied, after which all previous joints together with all free boxes are checked for being a joint. After this, all present chains are checked whether they are a loop (`if (($stru !~ /Joint/) && (CheckLoop(@{$$ListName{$stru}})))`) or maybe a possible loop (`$cpl = CheckPosLoop(@{$StrList{$j}}[0],@{$StrList{$ch}})`).

Now all possible new configurations of the structures are accounted for after a single move is made. The structure list is updated and ready to use by anyone.

4.3 Implementation of the endgame

Now that all parts of the game that are common knowledge are found and available all the time during a game, it is time to take a closer look at the way the game playing system makes its decision about which move to make next. The `PlayGame()` subroutine that controls the game calls the `SelectMove($GridName)` routine that has to come up with a box `$box_nr` in which to

```
if ($#d_handouts >= 0) {
  $box_nr = ${$StrList{$d_handouts[0]}}[0];
  @tmpval = FindValence($Gridname,$box_nr);
  $edge_nr = pop(@tmpval);
  return ($box_nr,$edge_nr);
}
```

Table 4.7: Implementation of Doublecrosses

fill the edge $edge_nr. The SelectMove() routine first checks whether there are free boxes left or whether the game finds itself in an endgame (if ($#free_boxes >= 0)). If there are no free boxes left, the subroutine moves to the endgame part covered in the following subsections. The part about, the case in which there still exist free moves is discussed in section 4.4.

4.3.1 First stage of the endgame

The game playing system will not have any problem finding out that the endgame stage is reached. When the @free_boxes array is empty there are no boxes left that are not part of any structure, therefore there are no edges left that can be played without leaving a handout which indicates that the endgame stage is reached.

The first thing the SelectMove looks for when the game is in the endgame stage, is whether there are doublecrosses to make in the current playing field (see table 4.7).

If there are multiple doublecrosses, it does not matter which to fill first since the player is to move again to fill the others. This is what we see in the second line in table 4.7: the first available doublecross is selected, the valence of one of the two boxes is found and the edge that is in the valence array is the only edge available in the doublecross structure and is the one returned to be played. When there are multiple doublecrosses available this rule will be applied again when the next move needs to be selected.

If there are no doublecrosses left in the playing field the next rule is applied which is concerned with the other available handouts. First all possible configurations in which the handout must be filled are worked out. When none of these rules can be applied there is one option left, indicating that the only way to gain control is to give away boxes, thus returning the double dealing move.

First, all present handouts that can be taken safely are filled. This is the case when the handout is present in only one structure or when there are not more then 3 moves left (if ($#tmplist == 0) or (($nmb_edge - $move_number) < 4)). Also when the sum of 1- and 2-chains in the playing field is even and greater than 0, all available handouts must be taken. When the sum of 1- and 2-chains is even and a player is to move with a handout present in the playing field, this player will gain control if he takes all boxes in the structure, and plays a hard-hearted handout in one of the 1- or 2-chains. When there are 1- or 2-chains connected to joints there needs to be a refinement ($refine) for joints connected to more then two 1- or 2-

```
if ($#handouts  >= 0) {
  foreach $ho (@handouts) {
    $tmpbox = ${$StrList{$ho}}[0];
    @tmplist = InStructure("StrList",$tmpbox);
    if (($#tmplist == 0) or (($nmb_edge - $move_number) < 4) or
      ((($#one_chains + $#two_chains + 2) > 0) and
      ((($#one_chains + $#two_chains + 2 - $refine) % 2) == 0))) {
      $box_nr = $tmpbox;
      @tmpval = FindValence($Gridname,$tmpbox);
      $edge_nr = pop(@tmpval);
      return ($box_nr,$edge_nr);
    }
  }
}
```

Table 4.8: Implementation of taking a handout in the First Stage

chains. Then one of the short chains will merge into a new structure when the joint disappears (`$refine = 1`), and in a special case two short chains will disappear (`$refine = 2`). Since there is no sense in denying these types of handouts, the open edge is returned as the move to be played (see table 4.8).

These rules cover all possibilities in the first part of the endgame. The next subsection will explain how to play during the loony endgame.

4.3.2 Loony Endgame

When none of the rules from the previous subsection are applicable to the current playing field (e.g. there are no 1- or 2-chains left) the system recognizes the loony endgame. In a loony endgame we notice two types of positions: positions with and positions without handouts present in the playing field. We can safely assume that when there are single handouts around, the player to move is in control and when there are no handouts present in the playing field, the player to move is out of control. For the systems rule base this distinction is not that strict since in both situations the 'make a doublecross' rule will always be applied first (if possible) before anything else happens. In the following subsections I discuss the way the system makes its choice in situations with (in control) and without (out of control) handouts in the current playing field.

In Control of the Loony Endgame

When there are handouts available in a loony endgame we can claim that the player to move is in control. Therefore, now all other possible playing field configurations with present handouts are checked (see table 4.9). First it is checked whether there is more than one handout available. If so, and when there are exactly two handouts available the first available box is

```
foreach $ho (@handouts) {
  $tmpbox = ${$StrList{$ho}}[0];
  @tmplist = InStructure("StrList",$tmpbox);
  if ($#handouts > 0) {
    if ($#handouts == 1) {
      $tmpbox1 = ${$StrList{$handouts[0]}}[0];
      $tmpbox2 = ${$StrList{$handouts[1]}}[0];
      $samestr = InSameStructure("StrList",$tmpbox1,$tmpbox2);
      if (($samestr ne "F") && ($#{$StrList{$samestr}} == 3)) {
        if (FindControlValue($samestr,4)) {
          return Take Handout
        } else {
          return Double-Dealing Move
        }
      }
    }
    return Take Handout
  }
```

Table 4.9: Implementation of taking a handout with multiple handouts in a loony endgame

filled since the double dealing move to keep control over the endgame can be made in the other structure. This happens unless both handouts are present in the same structure (`$samestr = InSameStructure("StrList",$tmpbox1,$tmpbox2);`). This indicates the remainings of a loop, and therefore the double double-dealing move must be made at the end of the turn leaving four boxes for the opponent to keep the control over the game. The double-dealing move only has to be made when there are four boxes left in the structure, until then all open boxes must be filled. Before the double-dealing move is played, there is made room for one exception. Since the system is in the position at this point to make a double-dealing move it is in control. Also, the only way to keep control is to give away four boxes, which is a lot. So when taking the four boxes would guarantee a decisive lead, the system, instead of making the double-dealing move, takes all four remaining boxes (`if (FindControlValue($samestr,4)) {`). In all other cases with two handouts around, one of the handouts is filled (see the end of table 4.9).

At this point it is still possible to move beyond this point in the rule base. When there is exactly one handout left in a structure containing more than one box, none of the tests to access the previous tests is passed. Now the following happens: if the structure containing the handout is a 2-chain, a handout is produced unless the taking of both remaining boxes will guarantee a decisive lead (see table 4.10). At last, in all cases with a handout in the playing field that are not covered by any rule, the handout is taken (see the end of table 4.10).

```
foreach $s (@tmplist) {
  if (InList(@two_chains,$s)) {
    if (FindControlValue($s,2)) {
      return Take Handout
    } else {
      return Double-Dealing Move
    }
  }
}
return Take Handout
```

Table 4.10: Implementation of taking a handout in a single chain in a loony endgame

Out of Control of the Loony Endgame

When the system is out of control and needs to come up with a new move it follows the protocol given in section 3.3.6. Since we assumed the system to be out of control when there are no handouts in the playing field, I also covered the first stage of the endgame here. This led to a small adjustment to the protocol. Instead of starting to give up loops the system starts with giving up 1- and 2-chains with a hard-hearted handout. This might be more profitable because this gives the opponent the possibility to make a mistake. After this the smallest loops are offered following the initial protocol.

If we take a look at the code sample from the implementation table 4.11, we see that this protocol is strictly followed. Before any action is taken there is a check which structure is the best to give away in the order of the protocol. In the first place the 1-chains are given away. If there is a 1-chain in the playing field, one of the open edges of this structure is selected to play. The selection of the edge uses one of the most important features of Dots-and-Boxes in an endgame: when there is no possibility of a hard-hearted handout all moves in a structure are equivalent. When there is no 1-chain present, the system looks for 2-chains. If there is a 2-chain present in the playing field the system chooses the hard-hearted handout, filling the edge between the two boxes in the structure (`$edge_nr = GiveCommonEdge($box_nr, ${$StrList{$two_chains[0]}}[1]);`).

With neither 1-chains or 2-chains available in the playing field the system looks at the loops. It selects the smallest loop (`if ( $#{$StrList{$loop}} < $#{$StrList{$minloop}}) {`) and selects one of the edges in it to play.

After this, the rule that selects possible loops might come in action when there aren't any loops left. It selects if available a possible loop and selects the move that finishes the loop (`($box_nr,$edge_nr) = split(/:/,$tmpedge);`), in order to disconnect it from anything attached to it. The loop that arises will be dealt with during the next move and will give the player out of control some points.

When none of the rules above can be applied, indicating that there are only long chains available, the system searches for the smallest long chain (`if ($#{$StrList{$chain}} <`

```
if ($#one_chains >= 0) {
  $box_nr = ${$StrList{$one_chains[0]}}[0];
  @tmpval = FindValence($Gridname,$box_nr);
  $edge_nr = pop(@tmpval);
  return ($box_nr,$edge_nr);
} elsif ($#two_chains >= 0) {
  $box_nr = ${$StrList{$two_chains[0]}}[0];
  $edge_nr = GiveCommonEdge($box_nr,${$StrList{$two_chains[0]}}[1]);
  return ($box_nr,$edge_nr);
} elsif ($#loops >= 0) {
  my $minloop = $loops[0];
  foreach my $loop (@loops) {
    if ($#{$StrList{$loop}} < $#{$StrList{$minloop}}) {
      $minloop = $loop;
    }
  }
  $box_nr = ${$StrList{$minloop}}[0];
  @tmpval = FindValence($Gridname,$box_nr);
  $edge_nr = pop(@tmpval);
  return ($box_nr,$edge_nr);
} elsif ($#posloops >=0) {
  $tmpedge = pop(@posloopmoves);
  ($box_nr,$edge_nr) = split(/:/,$tmpedge);
  return ($box_nr,$edge_nr);
} else {
  my $minchain = $L_chains[0];
  foreach my $chain (@L_chains) {
    if ($#{$StrList{$chain}} < $#{$StrList{$minchain}}) {
      $minchain = $chain;
    }
  }
  $box_nr = ${$StrList{$minchain}}[0];
  @tmpval = FindValence($Gridname,$box_nr);
  $edge_nr = pop(@tmpval);
  return ($box_nr,$edge_nr);
}
```

Table 4.11: Implementation of selecting a move being out of control in a loony endgame

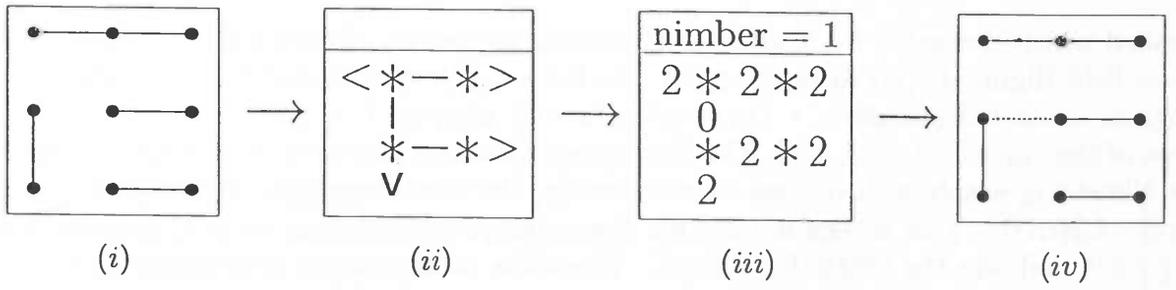


Figure 4.2: Steps in nimber move calculation. (*i*) The original Dots-and-Boxes playing field; (*ii*) the corresponding Nimstring field obtained by `PrintNimPos($Gridname)`; (*iii*) the nimbers of all possible moves obtained by the nimstring calculator; (*iii*) the next move to play (dashed) obtained by `Select_Nimber_Zero()`.

`$#{$StrList{$minchain}})` {) and selects an arbitrary edge in that structure.

Being out of control leaves the system nothing but giving up structures in a useful order, so these rules cover the way the loony endgame must be played being out of control.

4.4 Implementation of the middle game

For the middle game there exists the strategy of applying the Nimstring calculator when it takes a reasonable amount of time to calculate the rest of the game (see section 3.4.2. Before that the game playing system makes random moves since it is very difficult to define heuristic and basic strategies for the early parts of the game.

At the end of the middle game when no *free* moves remain the game transposes into the endgame which will be completely played by the simple rules described in section 4.3.

4.4.1 Nimstring

The theory of the Nimstring game that has been thoroughly explained in section 3.4.2 has been implemented by Freddy Mang [12] and improved by Glenn Rhoads [14] and made available for everyday use. The program reads a file in which it looks for a representation of a Nimstring field for which the nimber must be calculated. The name of the file needs to be given as an argument when the program is started. I altered the program in such a way that it returns the nimber value as the return value of the program so that it can be immediately read by the Perl game playing device (see table 4.12 for the way the next move is selected).

The main problem in implementing the nimber calculator in the game playing system, is the difference in representation between both programs. In figure 4.2 we see the steps needed to find the nimber of a particular Nimstring position. Since the Nimstring calculator is implemented for a Nimstring game, it expects a Nimstring graph as input to make its calculations on. So before the Nimstring calculator is called to do its job the `PrintNimPos($Gridname)` subroutine

is called which generates a file with the Nimstring equivalent (figure 4.2(*ii*)) of the Dots-and-Boxes field (figure 4.2(*i*)) to be analysed (see table 4.12). Now everything is in place and the program is started (`$nimber = (system("./a.out nimpos") / 256)`) returning the nimber value of the current playing field. The Nimstring calculator also creates another file containing the Nimstring graph with instead of each string, the nimber values of that particular move (figure 4.2(*iii*)). This makes it easy for the game playing device to pick possible 0-moves to go out and win the battle for control. When the nimber value is different from 0 or 100 (indicating a loony position), there is a move available that makes the nimber 0 to gain control of the game. Therefore if the nimber is different from 0 or 100 (`if (($nimber != 0) and ($nimber != 100))`) the system searches for the moves which make the nimber 0 (`@zero_moves = Select_Nimber_Zero()`) that can be found in the file that the Nimstring calculator created. Now the first move which does not leave a handout is selected to be played. If all possible 0-moves leave a handout, the move in the smallest structure is selected to be played (figure 4.2(*iv*)). Since all 0-moves will gain control over the endgame there is no sense in giving up more boxes than necessary.

4.4.2 Critical value

Since the object of this thesis is to create a game playing device based on the way human players play games, it would be unfair to let some sort of brute force calculating mechanism do all the work. So the object is refined to creating moves in a reasonable amount of time. The Nimstring theory is a typical brute-force technique but also very intuitive when it comes to smaller playing fields and fields that are divided in parts by the available structures. This is the reason we will still use it in the evaluation of a position and the choice of moves.

Still we need to use the Nimstring calculation in an intuitive fashion. One way to do this is to create a critical value which decides whether the calculator may be used in the move making process. Furthermore we have to measure in some way when the calculator will take a reasonable amount of time to calculate the nimber of the current game position. The way I choose here is to make a heuristic based on the *density* of the game position. We saw in section 3.4.2 that the search space of the nimber function decreases if there are a lot of loony moves available since then the value of those positions is immediately made. We can argue that loony moves are most likely to arise if there are many structures available in which loony moves arise all the time. We can also argue that structures are more likely to arise if there are lots of moves made in the center of the playing field since single moves there have influence on two boxes at the same time. And thirdly we can say, that when there a lot of completed boxes early on, the chance of a loony position will decrease. We will define the density of a Dots-and-Boxes playing field as the degree of structure forming in that particular field. In this case this density is based on the three factors mentioned above, the complete valence of a playing field (the sum of all individual valences of the boxes), the number of structures present in the playing field and the number of completed boxes.

The reason for choosing the total valence of a playing field as an indication of the density

```
if (($GridValence <=
  ($nmb_edge + $#two_chains + $#L_chains + $#loops + 5)) or
  (($nmb_edge - $move_number) < 30)) {
  PrintNimPos($Gridname);
  $nimber = (system("./a.out nimpos") / 256);
  if (($nimber != 0) and ($nimber != 100)) {
    @zero_moves = Select_Nimber_Zero();
    foreach $move (@zero_moves) {
      ($b,$e) = split(/:/,$move);
      $adbox = AdjecentBox($b,$e);
      @val1 = FindValence($Gridname,$b);
      ($adbox ne "F") and (@val2 = FindValence($Gridname,$adbox));
      if (($#val1 > 1) and (($adbox eq "F") or ($#val2 > 1))) {
        return ($b,$e);
      }
    }
    foreach $move (@zero_moves) {
      ($b,$e) = split(/:/,$move);
      $adbox = AdjecentBox($b,$e);
      foreach $s (InStructure("StrList",$b)) {
        if ($adbox ne "F") {
          foreach $st (InStructure("StrList",$adbox)) {
            if ($min > $#{$StrList{$st}}) {
              $box_nr = $b; $edge_nr = $e;
            }
          }
        }
        if ($min > $#{$StrList{$s}}) {
          $box_nr = $b; $edge_nr = $e;
        }
      }
    }
    return ($b,$e);
  }
}
```

Table 4.12: Implementation of selecting the next move by nimber calculation

of a playing field instead of the number of moves made up to that point is because the total valence gives a better indication about how many edges are filled. The total valence of a playing field can differ dramatically after a certain number of moves if all moves are made at the edge of the field since a move at the edge decreases the valence of a field by only one. So moves in the center will lower the total valence of a playing field more quickly and limit the options on the field, indicating more possibilities of loony positions.

The total valence of a playing field is the best indication of the density of a playing field but to make the heuristic a bit more accurate we can use the presence of structures in it. We can say that the presence of a structure in a playing field causes a decrease in the search space of the nimber of about one move.

In table 4.12 we see how the critical value is used in the game playing system. It is used as a test to check whether the Nimstring calculator can be used, whether it will take a reasonable amount of time to calculate the nimber. For small playing fields (up to 3×3) the Nimstring calculator can be used during the complete game, so when there are less than 30 moves left but the critical value is not reached yet, the Nimstring calculator is used anyway. When both tests fail, the system moves on to the other rules available in the system until one is applicable.

4.5 Implementation of the opening

In section 3.6 I outlined some ideas about the way the opening can be played by a game playing system. These ideas are mostly machine learning solutions. Due to the human like character of this thesis, I did not see a reason and chance to implement all those fascinating ideas, described in section 3.6, swirling through my head.

Therefore, I chose to model the way Dots-and-Boxes is played when none of the rules are applicable, described in the previous sections on the middle game and endgame, as random moves. It might be possible that no rule is applicable when the current game situation does not allow any of the possible rules to come into action. As we see in table 4.13 the system picks a random box out of the `@free_boxes` array (`$box_nr = $free_boxes[int(rand($#free_boxes + 1))]`) and picks one of its free edges (`$edge_nr = $val[int(rand($#val + 1))];`). When this edge is also free to be played from the perspective of the adjacent box (`while(!(InList(@free_boxes, $nb)) and !($nb eq "F"))`) the selected edge is returned to be played.

I still believe that with the help of several machine learning techniques it is possible to find specific rules in the way the machines solve the game. These rules, containing some abstract form of information about the game, can then be used to play the game of Dots-and-Boxes in such a way that it is only necessary to have a general understanding of how to play the game instead of calculating all possible moves and select the best.

```
while(!(InList(@free_boxes,$nb)) and !($nb eq "F")) {
  $box_nr = $free_boxes[int(rand($#free_boxes + 1))];
  @val = FindValence($Gridname,$box_nr);
  $edge_nr = $val[int(rand($#val + 1))];
  $nb = AdjecentBox($box_nr,$edge_nr);
}
return ($box_nr,$edge_nr);
```

Table 4.13: Implementation of making a random move

4.6 Graphical User Interface

In section 4.2.1 it has been described how the Dots-and-Boxes game playing system monitors the game. However, nothing is mentioned about the way the moves of the opponent of the game playing system are fetched. To give the human opponent a chance to interact with the game playing system as easily as possible and to give the opponent a good overview of the game situation, I created a Graphical User Interface (GUI) which gives a colorful display of the current game situation, highlights the last move made, and allows the player to click on the edge one wants to play.

The GUI is implemented as a (public) website (see section 4.6.1). Since the only goal for the GUI is to be used as a front-end of the game playing system, it does not have to be very sophisticated, and also websites are easy to maintain and accessible. This way it gives anybody the chance to take on the system and is it easy to test. Still the object was to create a website which only changes the moves made instead of reloading the complete page when something changes. This caused some technical difficulties related to client- and server side scripting. The solutions to this problem are presented in section 4.6.2.

4.6.1 Choice of GUI

The website were the game can be played is made by a *cgi-script*. This gives the user the flexibility to set the size of the playing field and the choice of who goes first. The first page containing these choices can be found at `http://www.ai.rug.nl/~blom/dots/index.php` (see figure 4.3).

After the user has entered his preferences and clicked the 'Go' button, the game interface is loaded. Based on the size entered by the user a cgi script creates the wanted playing field and loads the Dots-and-Boxes game playing system on the side (see figure 4.4). When it is the user's turn to make a move, it says 'A is to move' below the playing field, and the user can click on an empty (yellow) edge to select his move. The selected move now turns blue indicating that is was the last move made. Now the text indicating who is to move underneath the playing field may change, depending on the game situation, but the user has to wait until it changes back to 'A is to move', indicating it's his turn. When the game playing system is

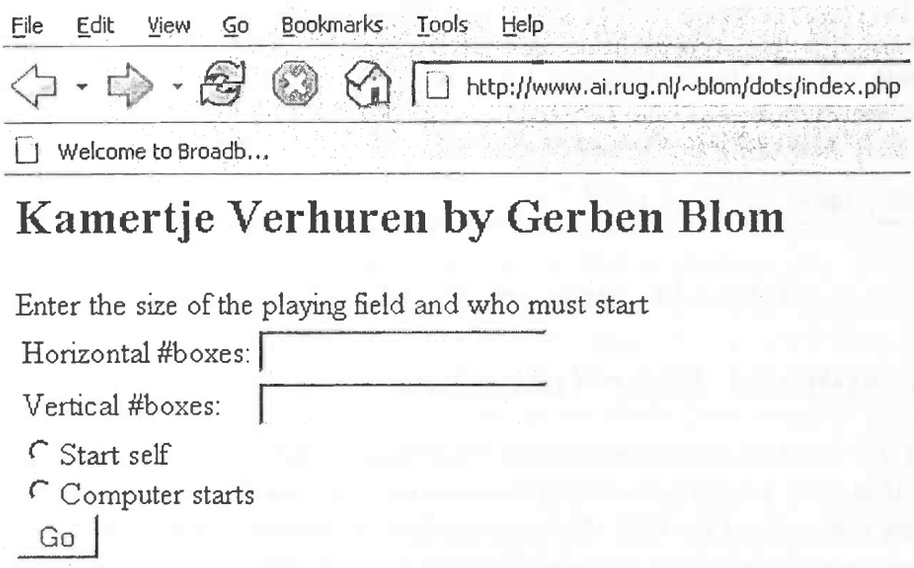


Figure 4.3: Screenshot of the Dots-and-Boxes index.php page

to move and has made up its 'mind', it colors its preferred edge blue, indicating that it made its move. When all edges are filled the score is presented on the bottom of the screen, together with an invitation to play again!

4.6.2 Interaction between GUI and Perl

Since the Dots-and-Boxes game playing system was implemented in Perl it was fairly easy to load the system in an extra frame on the game playing system's website. Using the standard cgi headings at the start of the program, the game playing system now prints all kinds of information to the side of the site (see figure 4.5).

Now everything is in place, a pre-chosen starting grid by the user (created by `main.cgi`) and a started game playing system on the side (`KamertjeVerhuren.cgi`), both systems need to know how to interact. The game playing system needs to know which and when moves are made by the user in order to make the next move and the user must see which move is made by the game playing system. This interaction causes some difficulties. Since the game playing system runs on the server on which the website is located (server side) and the website on which the user selects his moves is located on his own computer (client side), there is a huge gap to to bridge to get the selected moves back from client to server (see [3] for more insights on client-server interaction). The other way is obviously not a problem since most information is sent from servers to clients. One way to overcome this problem is by asking the server for a new page (by clicking on one of the edges) and adding to this request the selected move. This way the server can provide the game playing system with the move made. The

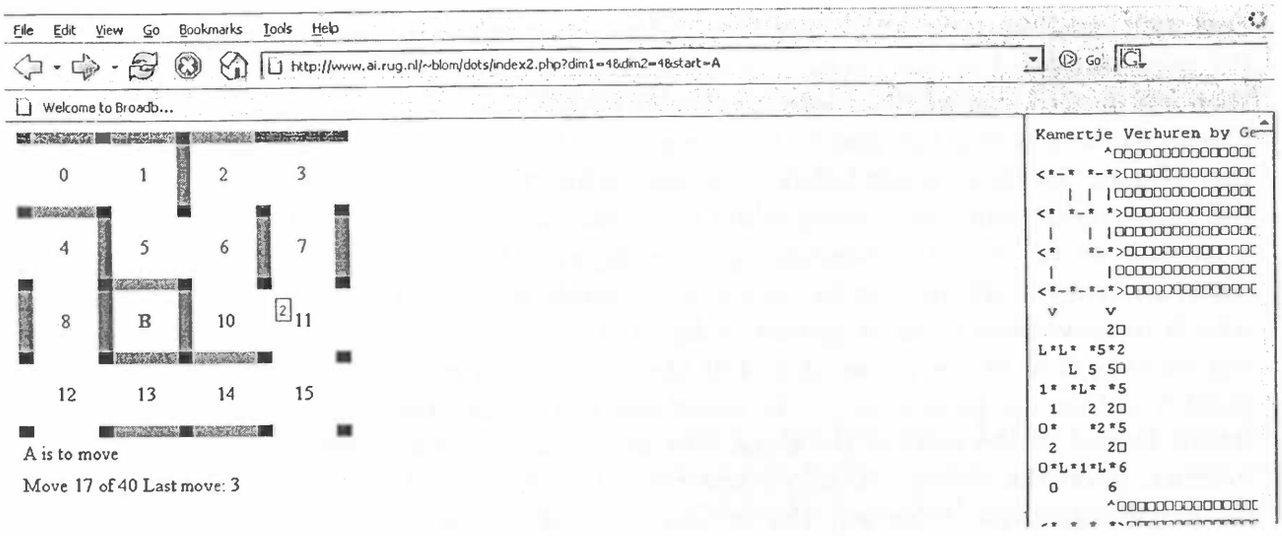


Figure 4.4: Screenshot of the Dots-and-Boxes game play interface. The yellow (invisible) edges indicate free edges, the edges indicate all played edges, and the blue one on the top of box number 3 highlights the last move made.

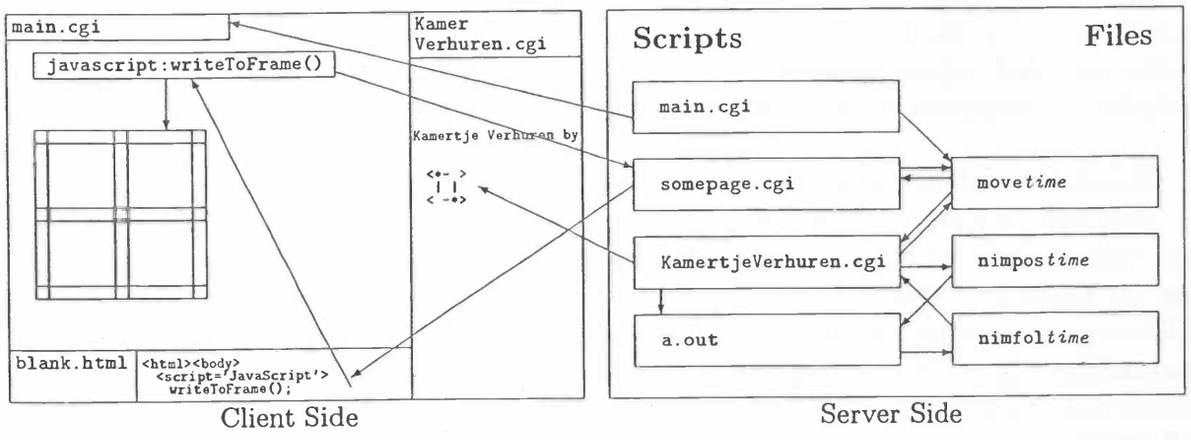


Figure 4.5: Overview inner structures of the Dots-and-Boxes website. On the left the parts that work on the user side, on the right everything that runs on the server.

drawback behind this method is that every time one selects a move, the complete page must be reloaded. The solution to this problem seems somewhat laborious, but it is quite neat. To deal with small updates on the page locally on the client side, the site uses javascript. When the page is created on the client's screen, Javascript loads two functions that it can use when they are needed. So when an edge is clicked by the user, a function is called. This function checks whether it was the user to move and then it updates the color of the clicked edge and the color of the move made before that one. After that it decides who is to move (this way the user cannot continue clicking while the game playing system is thinking) and it calls for a website on the server (`somepage.cgi` (see figure 4.5)). Along with this call, the function sends some extra information to the server. It sends who made a move (user or system [A,B]), who is to move next (user or system [A,B]), the move made (the number of the edge played), the starting time of the game (stored at the start of the game) and the last move made (the number of the edge played last). The `somepage.cgi` script prints its output to the `blank.html` frame, loaded at the start of the game, but not visible. The `somepage.cgi` script handles the interaction on the server. It writes and reads the user's and system's moves to and from a file named `move`*time* (this way the system can distinguish between different games). When the script has read a move by the game playing system, it writes the javascript function to the hidden frame (`blank.html`). This way the playing field will be updated with the system's move and all other features, and the player to move can make the next. This process continues until the last move is made after which an extra javascript function is called which prints out the score of the completed game and invites the user to play again. When the last move is made, also the `KamertjeVerhuren.cgi` script stops running and the help files `move`*time*, `nimpos`*time* and `nimfol`*time* are removed from the server.

Chapter 5

Results

To evaluate the Dots-and-Boxes game playing system I implemented, I tested its desired behaviour in all sorts of situations. I played it myself, made it play against a very good program called Dabble [9] which uses alpha-beta pruning, and finally I watched people perform against the game playing system on the public website.

The results are as expected. When in control, the game playing system collects all offered boxes except the boxes it needs to give away to keep the control. When giving up the control would gain enough to win on the other hand, the system takes all offered boxes, taking a lead that cannot be overcome. If the game playing system is out of control it gives away the available structures following the protocol described in section 3.3.6. This keeps the opponent in control, while the system gains as many boxes as possible.

The performance of the critical value heuristic might need some tuning. So far it has been working great. Due to a slight adjustment made during the testing phase the Nimstring calculator never really starts calculating annoyingly long. The testing phase uncovered a hidden type of game situations, situations in which a lot of moves have been played but with very few existing structures (typically when a lot of boxes are taken early on in the game), in which the Nimstring calculator took too much time to think.

The game playing system still has its weak parts, situations which it does not handle very well. One weak point of the system is its behaviour in games with few or no large structures available. In that case neither player is able to keep the control by giving away the last boxes of a structure. In that case it needs a bit of good fortune. The biggest problem for the game playing system is obviously the opening stage in which it only makes random moves until the nimstring calculator kicks in. The best chance of beating the system is to use meta-strategies in this stage of the game, keeping the game playing system from finding 0-moves in the outcome of the Nimstring calculator at the end of the middle game. This is why the system loses all the time from the alpha-beta pruning program Dabble. Dabble calculates all the time, and based on its heuristics it has better predictions earlier on about the control of the endgame.

Chapter 6

Conclusions

The game of Dots-and-Boxes seems to be a very interesting research subject for Artificially Intelligent game playing. Because of the possibilities of extension of the playing field, the game can be used to test a variety of different approaches to Artificial Intelligent game playing such as search, machine learning or rule-based techniques.

Concerning the research proposed in section 1.1 and the goals set in section 1.3, I can say I have found answers and solutions to most of the questions raised. As we have seen in the Results chapter (see chapter 5), I have created the intended implementation of a robust, fast, human-like game playing system. The system plays the endgame in the best possible way: it reaches the desired control over the endgame in most cases with the help of the Nimstring calculator and makes moves within a reasonable amount of time due to a heuristic that gives an indication of the Nimstring calculation time.

As for the extra questions raised (see section 1.1) in the introduction I can say I found some partial solutions. The formalization of the endgame has been completely analysed and described (see section 3.3). I find this to be the starting point of the way one plays the game of Dots-and-Boxes. When one becomes familiar with the subtleties hidden in the endgame, one becomes aware of the importance of control. I think that this is the starting point of any research into intelligent reasoning about the game of Dots-and-Boxes.

The answer to the question about how the game can be analysed in parts is partly answered throughout this thesis. Obviously there have been partial solutions to different stages in the game. Whether splitting the game in parts in opening play, e.g. study clusters of edges or analyse the horizontal/vertical distribution of played edges, helps in finding general rules for opening game play will remain subject for future research for now.

The third question concerning general strategies for the whole game is also partly answered. We have found evidence that the complete game focuses on gaining control over the endgame. From that point of view we can say that endgame strategies are generalized over the whole game. This, however, is not a necessity for a Dots-and-Boxes game playing system to find rules for perfect play.

On the question of meta-strategies there is still a lot room for improvement. I have not

implemented any ideas about the way opponents might approach the game. Such reasoning about opponents (see also [5]) and the rules outlined in section 3.4.3 might offer a very useful contribution to the research for opening play in the game.

The evaluation of the game playing system is described in chapter 5. Still there is room for discussion about the way one *should* evaluate game playing systems in general. In this case I have chosen for a human-like system and evaluated it on the behaviour I intended it to have. On the other hand, many first-time users may evaluate the system poorly due to the strength of the system, since only a few human subject were able to win a few games. On little playing fields up to 4×4 boxes, human players were able to create positions from which the game playing system was not able to win. In larger playing fields human players have to practise a lot before they will win a game.

Chapter 7

Discussion

To give some insight in the way this research links up with other work, I shall distinguish two categories of research that can be connected with this thesis. Firstly there is the work done on the analysis and implementation of the game of Dots-and-Boxes and secondly we can compare this work with research on other human-like approaches to Artificially Intelligent game playing.

The implementation of the Dots-and-Boxes game playing system is mainly based on the work by Berlekamp and Scott [1, 2, 17] with the useful help of the work on the Nimstring Calculator by Mang and Rhoads [12, 14]. With the help of their work I made the analysis on Dots-and-Boxes and made an implementation of a human-like game playing system. In comparison with other approaches such as the machine learning devices by Kronfol [11] and Weaver [20], the human-like game playing system is able to distinguish different stages in a Dots-and-Boxes game, giving it the ability to use different strategies at different times. The advantage of using different strategies becomes visible when we see machine learning techniques generate over complete games, thus learning that giving away boxes is in fact a good thing (even though it might be forced). The most promising other approach is the alpha-beta search method implemented in Dabble [9]. Using heuristics to evaluate a Dots-and-Boxes position can proof to be very useful. Still heuristics need a lot of senseless searching which loses the human-like character of the game-playing system.

The human-like aspect mainly links up with work like its done by Pieter Spronck [18] on "dynamic scripting" which uses a dynamic rule base which consists of different rules in different stages of a game. This dynamical approach gives a game playing system the ability to adapt to different stages in a game quickly. This way a game playing system is not forced to focus on one general aspect of the game, but now it has the option, like a chameleon, to adjust itself to the changing environment. This is in complete agreement with the fundamentals of multi-agent systems research, trying to describe a dynamic environment from different (expert) perspectives creating a complete working system.

Concerning machine learning to generate general rules in different games, the ideas described in section 3.6.1 link up with the work done on using reinforcement learning in Chess by Henk Mannen [13] and in Go by Reindert-Jan Ekker [8]. Like the research by Mannen and Ekker, I

propose machine learning techniques that use abstract features in game situations. Also these ideas link with the work by Erik van der Werf on Artificial Intelligence techniques in Go [21], where he uses theories from the field of pattern recognition among other things. This research might be of huge interest in the search for general rules in the opening of Dots-and-Boxes, since as in Go, the playing field can be described in terms of the structures that arise during a game.

Chapter 8

Recommendations for future research

The complete base for a rule-based approach to the game of Dots-and-Boxes is presented throughout this thesis. In both theory and implementation of the way the endgame and middle game must be played there is not much room for improvement. Still there are a few points that need some attention. The calculation for the critical value from which the nimstring calculator can be used might need some tuning. Also a heuristic which can distinguish between good or better "nimber = 0" moves might come in handy at some point. On the user end of the system there is room for improvement on the website. You can think of playing against different levels of the game playing system (random only, not giving away boxes or taking handouts etc.), better color schemes, coloring the background of a filled box and maybe try to save some games in an ordered fashion and try to create some high score list.

Besides these small adjustments there is room for the implementation of new suggested strategies (see section 3.6). The suggested methods, learning from abstract features in the playing field such as present structures, taken boxes, the number of structures, the player to move and the number of moves on the side of the playing field, need lots of investigation before we are able to derive concrete rules from them about how to play in the opening of Dots-and-Boxes. Still they seem better input variables into a machine learning network then just the edges of the playing field. One idea to implement the meta-strategy rules (see section 3.4.3) might be to split the free boxes list in two parts categorizing the free edges in a "connected to structure" and a "not connected to structure" category. This way the system can make random moves, stretching existing structures or tries to create new ones, depending on the current situation.

One of the most important things we have learned from this thesis is the fact that games like Dots-and-Boxes are not homogeneous enough to be described in one general way. The distinction between different stages in a game is necessary since a game playing system should not learn that giving up boxes, which might be forced in some stage in the game, is normal. Instead of trying to create one generalization, we need to find a way to describe every different stage independently.

Bibliography

[1] E.R. Berlekamp. *The Dots-and-Boxes Game: Sophisticated Child's Play.* AK Peters, Ltd. 2000

[2] E.R. Berlekamp and K. Scott. Forcing your opponent to stay in control of a loony Dots-and-Boxes endgame. *More Games of No Chance*, Vol. 42, MSRI Publications, 2002, pp. 317-330.

[3] D.E. Comer. *Computer Networks and Internets with Internet Applications (fourth edition).* Prentice Hall, Upper Saddle River, NJ, 2004, ISBN 0-13-143351-2

[4] H.P. van Ditmarsch. *Knowledge Games.* Proefschrift University of Groningen, ILLC, Amsterdam, 2000

[5] H.H.L.M. Donkers. *Nosce Hostem, Searching with Opponent Models.* Proefschrift University of Maastricht, Universitaire Pers Maastricht, 2003

[6] F.E. Douma. *Kwartetten: Logica en Strategie voor het Kaartspel.* Scriptie KI Groningen, 2002

[7] S. Druiven. *Knowledge Development in Games of Imperfect Information.* Scriptie KI Groningen, 2002

[8] R-J. Ekker. *Reinforcement Learning and Games. Temporal-Difference algorithms for Gameplay and their performance on playing 5x5 Go.* Scriptie KI Groningen, 2003

[9] J.P. Grossman. *Dabble.*
`http://www.ai.mit.edu/~jpg/dabble/`

[10] J.C. Holladay. A note on the game of dots. *The American Mathematical Monthly*, Vol. 73, No. 7, 1966, pp. 717-720.

[11] Z. Kronfol. Dots-and-Boxes: a neural networks approach. Princeton University Computer Science Department, 2000

[12] F.Y.C. Mang. *A Nimstring Calculator.*
`http://www-cad.eecs.berkeley.edu/~fmang/nimstring/index.html`

[13] H. Mannen. *Learning to play chess using reinforcement learning with database games.* Scriptie CKI Utrecht, 2003

[14] G.C. Rhoads. *Nimstring Calculator.*
`http://paul.rutgers.edu/~rhoads/ind.html`

[15] S.J. Russell and P.I. Norvig. *Artificial Intelligence: a Modern Approach.* Prentice-Hall, 1995

[16] R. Schwartz, T. Christiansen and L. Wall. *Learning Perl.* O'Reilly, 1997

[17] K. Scott. *Loony dots and boxes endgame.*
`http://www.msri.org/publications/ln/msri/2000/gametheory/scott/1/`

[18] P. Spronck, I. Sprinkhuizen-Kuyper and E. Postma. Online adaptation of game opponent AI with dynamic scripting. *International Journal of Intelligent Games and Simulation* (eds. N.E. Gough and Q.H. Mehdi), University of Wolverhampton and EUROSIS, March/April 2004, Vol.3, No.1, ISSN: 1477-2043, pp. 45-53.

[19] I. Vardi. *Mathter of the Game.* `http://cf.geocities.com/ilanpi/dots.html`

[20] L. Weaver and T. Bossomaier. Evolution of neural networks to play the game of Dots-and-Boxes. In: *Alife V: Poster Presentations*, 1996, pp. 43-50

[21] E.C.D. van der Werf. *AI Techniques for the Game of Go.* Proefschrift University of Maastricht, Universitaire Pers Maastricht, 2005

[22] D. Wilson. *Dots-and-Boxes Analysis Index.* `http://www.cae.wisc.edu/~dwilson/boxes/`