Simulation of Mathematical Games using Functional Programming

by

David Murphy

under the supervision of

Dr. Hugh Gibbons,
School of Computer Science and Statistics, TCD

A thesis submitted in partial fulfilment for the degree of MAI (Masters in Engineering)

in the
Faculty of Engineering, Mathematics and Science
School of Engineering

Submitted to the University of Dublin, Trinity College, May 2015

Declaration of Authorship

I, David Murphy, declare that this dissertation titled, 'Simulation of Mathematical Games using Functional Programming' and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a degree at this University.
- No part of this work has previously been submitted as an exercise for a degree, either in Trinity College Dublin, or in any other University.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- The library of Trinity College Dublin may lend or copy this work or any part thereof on request.

Signed: ______________________________

Date: ______________________________

"A computer once beat me at chess, but it was no match for me at kick boxing."

Emo Philips

Summary

The motivation of this project was to examine the practicality of functional programming when designing a program to play the combinatorial game Dots-and-Boxes at an advanced level. Specifically, it was intended that the program be able to analyse the state of the game at any time and to use the famous double-dealing move to good effect.

The realisation of such a program required the invention of sophisticated theory and strategies for advanced-level gameplay, the design of an architecture of data structures which would store game data in an efficient and utilisable way, the design and implementation of intricate algorithms for chain detection, game state analysis, and decision-making, as well as the development of an action policy which would enable the program to select actions consistent with the high-level strategies it was intended to follow. In addition to this, an interactive user interface required to be implemented in order to effectively test the program.

The design of the program centred around the representation of a game of Dots-and-Boxes as a mathematical graph. By doing this, the different types of nodes and chain structures could be categorised into different classes. The pattern-matching abilities of functional programming was used to identify these objects and perform actions accordingly.

It was found that functional programming was well-suited for this difficult task. The functional programming approach allowed complex and intricate functions to be constructed from simple, flexible, and powerful data structures and functions. Functional programming's support of higher-order functions and pattern-matching techniques, as well as its extensive toolbox of highly-utilisable data structures and functions allowed the program to be expressed in a concise and elegant way.

The completed program successfully achieved the goal of advanced-level play of the Dots-and-Boxes game and it did so with an elegance and finesse only achievable through the use of functional programming.

Acknowledgements

I wish to express my sincerest thanks to Dr. Hugh Gibbons for the invaluable guidance and encouragement which he extended to me over the course of this project. I am extremely grateful to him for sharing his time and expertise over the many weeks and months of this venture and also for introducing me to the fascinating world of combinatorial games.

I would also like to thank my family, friends, and my partner for their continuous support and encouragement during this long and arduous year. I would especially like to thank my parents for their unending support and guidance during this year and indeed my entire time spent at university. My gratitude to them for all that they have done for me cannot be expressed in so few words.

I also wish to offer thanks to ShareLaTeX [1] for enabling me to write this document without any prior experience with LaTeX, as well as to Miran Lipovača for teaching me functional programming through his book [2] which is free to read at `learnyouahaskell.com`.

Contents

Declaration of Authorship i

Summary iii

Acknowledgements iv

List of Figures ix

List of Tables xi

Author's Note xiii

1 Introduction 1
- 1.1 Combinatorial games . . . 1
  - 1.1.1 Nim . . . 2
- 1.2 Functional programming . . . 2
- 1.3 Aim . . . 3

2 The Dots-and-Boxes Game 4
- 2.1 Dots-and-Boxes . . . 4
- 2.2 Child's play? . . . 5
- 2.3 Complexity of the game . . . 5
- 2.4 Playing Dots-and-Boxes . . . 6
- 2.5 Mathematical theory of Dots-and-Boxes . . . 8
- 2.6 Creating a program to play Dots-and-Boxes . . . 9
- 2.7 Summary . . . 9

3 Strings-and-Coins 10
- 3.1 Strings-and-Coins . . . 10
- 3.2 Classification of nodes . . . 11
  - 3.2.1 The wall node . . . 12
- 3.3 Strings-and-Coins in functional programming . . . 13
  - 3.3.1 Node and edge data structures . . . 13
    - 3.3.1.1 Effectiveness of this architecture . . . 14

3.3.2 Traversing the graph . . . 14
3.4 Structures . . . 15

4 Chains **16**
4.1 Chains . . . 16
4.1.1 Length of a chain . . . 17
4.1.2 Categories of chain . . . 17
4.2 Open chains . . . 18
4.2.1 Opening chains . . . 18
4.3 Double-dealing . . . 19
4.4 The hard-hearted handout . . . 20
4.5 Short chains and long chains . . . 22
4.6 Control . . . 22
4.7 Detecting chains . . . 23
4.7.1 Chain data structures . . . 23
4.7.1.1 Chain data structures in functional programming . . . 24
4.7.2 The chain detection algorithm . . . 24
4.7.2.1 Chain detection in functional programming . . . 26
4.7.3 Gathering all the chains . . . 26
4.8 Summary . . . 27

5 Loops **28**
5.1 The loop chain . . . 28
5.2 Double-dealing with loops . . . 29
5.3 Detecting Loops . . . 30
5.3.1 Circular chain detection in functional programming . . . 32
5.4 Summary . . . 32

6 The Double-Dealing Decision **33**
6.1 Reward from the long chains . . . 34
6.2 Gaining control of the long chains . . . 36
6.3 Reward from the short chains . . . 37
6.3.1 Reward from 2-chains . . . 38
6.3.2 Allowing for 1-chains . . . 39
6.4 Decision time . . . 41
6.4.1 Cases when double-dealing is not performed . . . 44
6.4.2 The double-dealing decision in functional programming . . . 44
6.5 Summary . . . 45

7 Complex Chains **46**
7.1 Complex chains and complex structures . . . 47
7.2 Dependence . . . 47
7.2.1 Symmetry of dependence . . . 48
7.3 Dippers . . . 49
7.3.1 Detecting dipper loops . . . 50
7.3.2 Effective length . . . 51
7.3.2.1 Effective length in functional programming . . . 52
7.4 Detecting complex chains . . . 52

7.4.1 Complex chain data structures . . . 52
7.4.1.1 Complex chains in functional programming . . . 52
7.4.2 Complex chain detection algorithm . . . 53
7.4.2.1 Complex chain detection in functional programming . . . 54
7.5 Double-dealing and complex chains . . . 54
7.5.1 One-step-ahead analysis . . . 55
7.6 Summary . . . 57

8 Action policy **58**
8.1 Edge priorities . . . 58
8.2 Assigning edge priorities . . . 58
8.2.1 Further analysis of possible double-dealing opportunities . . . 60
8.2.1.1 Further analysis of edge type (B) . . . 60
8.2.1.2 Further analysis of edge type (G) . . . 61
8.2.1.3 Further analysis of edge type (H) . . . 62
8.3 Effectiveness of action policy design . . . 62
8.4 Summary . . . 63
8.5 Further work . . . 63

9 Conclusion **66**

A An Introduction to Functional Programming **68**
A.1 Functional Programming . . . 68
A.1.1 First-class functions . . . 68
A.1.2 Lazy Evaluation . . . 68
A.1.3 Higher Order Functions . . . 69
A.1.4 Currying . . . 69
A.2 Writing Functions in Haskell . . . 70
A.2.1 Type declarations . . . 70
A.2.2 Pattern matching . . . 70
A.2.3 Guards . . . 71
A.2.4 Wildcards . . . 71
A.2.5 Lambdas . . . 71
A.3 I/O with functional programming . . . 72

B Lists **73**
B.1 The List data structure . . . 73
B.2 Common List functions . . . 73

C Maps **76**
C.1 The Map data structure . . . 76
C.2 Common Map functions . . . 76

D Code **78**
D.1 Imported modules . . . 78
D.2 General functions . . . 78
D.3 Strings-and-Coins architecture . . . 79

D.3.1 Data types for nodes and edges 79
D.3.1.1 Edge and node identifiers 79
D.3.1.2 Edge structures 79
D.3.1.3 Node structures 80
D.3.1.4 NodeTuple objects 80
D.3.1.5 NodeMaps and EdgeMaps 80
D.3.1.6 Database objects 80
D.3.2 Node functions 80
D.3.3 Edge functions 82
D.3.4 Functions for graph traversal 83
D.4 Chains 84
D.4.1 The chain data type 84
D.4.2 Chain functions 84
D.4.2.1 General chain functions 84
D.4.2.2 Chain detection functions 86
D.5 Action policy functions 88
D.5.1 Double-dealing decision functions 88
D.5.2 Edge priority functions 90
D.5.3 The move selection function 93
D.6 Update functions 93
D.7 Other functions and data types 94

Bibliography **95**

List of Figures

2.1 A large game of Dots-and-Boxes as seen on the cover of [3] 4
2.2 A 4x4 grid of dots 5
2.3 An incomplete game of Dots-and-Boxes played on a 4x4 grid 7
2.4 Two possible endings to the game in figure 2.3 8

3.1 A game of Dots-and-Boxes shown with its corresponding game of Strings-and-Coins 10
3.2 A game of Dots-and-Boxes expressed in Strings-and-Coins representation 11
3.3 The five different classes of node used as part of this project 12

4.1 A series of chains 16
4.2 The Strings-and-Coins representation of figure 4.1 17
4.3 An open chain 18
4.4 Taking the coins from an open chain 18
4.5 Opening a chain 19
4.6 A winning position 19
4.7 Sacrificing some coins in order to force the opponent to open a longer chain 20
4.8 A double-dealing move 20
4.9 A closed 2-chain 20
4.10 The three possible ways to open the 2-chain (Dots-and-Boxes) 21
4.11 The three possible ways to open the 2-chain (Strings-and-Coins) 21
4.12 A double-dealing opportunity and a series of long chains 22
4.13 Detecting a simple chain 23

5.1 A game of Dots-and-Boxes with a loop 28
5.2 A loop chain 29
5.3 A double-dealing opportunity 29
5.5 An open loop composed of three nodes 29
5.4 Opening a loop chain 30
5.6 Double-dealing on an open loop 30
5.7 Loop detection 31

6.1 A difficult decision 33
6.2 A 4x4 grid of coins with four long chains 34
6.3 Gaining control of the long chains 36
6.4 Using a double-deal to gain control when the number of short chains is even 37
6.5 Distribution of coins from short chains when N_s is odd 38
6.6 A 4x4 grid of coins with both 1-chains and 2-chains 39

6.7 Board in figure 6.1 displayed in Strings-and-Coins format 42
6.8 List of chains contained within figure 6.7 43

7.1 A game of Dots-and-Boxes with several complex chains 46
7.2 A complex structure 47
7.3 Dependence between complex chains 48
7.4 An example of how dependence between chains can be asymmetric 48
7.5 A dipper structure and the complex chains contained within 49
7.6 Opening the non-circular region of a dipper 49
7.7 Opening a dipper 50
7.8 A dipper structure 51
7.9 A series of complex chains and a double-dealing opportunity 54
7.10 Two states which are equivalent from a double-dealing decision point of view 55

8.1 All the possible node pairs to which an edge can connect 59
8.2 An open simple 2-chain 60
8.3 An open loop chain containing four coins 61
8.4 An open complex 2-chain 62

List of Tables

6.1 Reward for gaining control of the long chains 41

8.1 Summary of function *getPriority* 64
8.2 Summary of function *getPriorityB* 65
8.3 Summary of function *getPriorityG* 65
8.4 Summary of function *getPriorityH* 65

Dedicated to my parents, Pauline and Peter Murphy

Thank you for everything

Author's Note

When analysing a state of a game of Dots-and-Boxes, the convention which will be used is as follows:

The player to whom the turn belongs is referred to simply as the "player". The other player is referred to as the "opponent".

Chapter 1

Introduction

People have always been intrigued by games and puzzles. The feeling of accomplishment that is felt when a long and arduous puzzle is finally solved is surpassed only by that of defeating a fellow player in a hard-fought game of chess or draughts. It is in our nature to always strive for victory over others and it is for this reason that games such as chess are so popular. This work is interested in a special family of games and puzzles known as *combinatorial games*.

1.1 Combinatorial games

Combinatorial games are typically two-player, turn based games with perfect information and no stochastic elements. Well-known examples of combinatorial games include chess, checkers, and tic-tac-toe. This family of games is not restricted to two player games, however; it also includes single-player puzzles and even zero-player games such as Conway's Game of Life. Most combinatorial games have very large state spaces and are NP-hard or NP-complete, making optimal play very difficult to achieve.

As well as being naturally appealing, the study of combinatorial games has applications in various areas including complexity analysis, logic, artificial intelligence, graph and matroid theory, and error correction codes [4]. One of the pioneering works on combinatorial game theory is *Winning Ways for your Mathematical Plays* [5], a book written by mathematicians Elwyn Berlekamp, John Conway (creator of the aforementioned Game of Life), and Richard Guy.

In [4], Fraenkel et al. provide a detailed explanation of combinatorial games and their properties as well as provide a large bibliography of popular combinatorial games and puzzles.

1.1.1 Nim

The game of Nim is one of the most important games in combinatorial game theory. Nim is a two-player game in which players take turns removing objects from separate *heaps*. During each turn, a player must take at least one object from a distinct heap. There is no limit to the number of objects which can be taken during a turn, provided that they all come from the same heap. In a normal game of Nim, the player who takes the last object is deemed the winner.

The game of Nim has been mathematically solved and the complete theory of the game was developed by Charles L. Bouton in [6]. A number which describes the value of a nim heap is known as a *Grundy number* or *nimber*. Nimbers have their own special addition and multiplication operations and are of great importance in the field of combinatorial game theory.

The Sprague-Grundy theorem [7], [8], discovered by mathematicians R. P. Sprague and P. M. Carmelo Grundy and later developed into the field of combinatorial game theory by Berlekamp et al. [5], states that every impartial game under the normal play convention has a corresponding nimber value. This does not mean that all games which meet this criterion have been mathematically solved, although it does mean that nimber arithmetic can be used in the their analysis.

1.2 Functional programming

Functional programming is a programming paradigm which models computation as the evaluation of mathematical functions. The output of such a function depends only on its list of arguments and their associated values. Unlike imperative programming (C, Java, etc.), there is no dependence on any global state.

The nature of functional programs greatly facilitates the difficult task of formal verification, i.e. proving that a program behaves as intended under any possible circumstance

using formal methods. The same cannot be said about imperative programs, however; the sequential and state-based nature of such programs makes this task much more difficult.

Functional programming languages have many capabilities which make them very useful for problem solving and analysis. In [9], Bird presents a program to solve the popular combinatorial puzzle Sudoku using functional programming. Bird's program solves the problem in an extremely elegant and sophisticated way and is a testament to the effectiveness and finesse of functional programming.

Function programming languages come with a toolbox of flexible and highly-utilisable functions, such as the famous *map* and *fold* functions, which greatly facilitate the development of concise and elegant programs. A detailed introduction to functional programming and its many capabilities is provided in appendix A.

1.3 Aim

The aim of this work was to create a program capable of playing the combinatorial game Dots-and-Boxes at an advanced level using functional programming in order to demonstrate the effectiveness of the functional programming approach to solving such problems. Specifically, the program was required to be able to perform a sophisticated action known as the *double-dealing* move and use it good effect.

The rest of this document is structured as follows: First, an overview of the Dots-and-Boxes game and its properties is given. Then the game of Strings-and-Coins is introduced and it is shown how this game influenced the design of the architecture of the program. In chapters 4-7, chain structures and their properties, advanced gameplay strategies, as well as various methods performing sophisticated analysis of the game using functional programming are discussed in detail. Finally, it is shown how the theory and strategies presented in this work are sufficiently summarised into the action policy of the program.

Chapter 2

The Dots-and-Boxes Game

The combinatorial game which was studied as part of this project is called Dots-and-Boxes. This game has a very interesting mathematical theory which makes achieving optimal play a difficult and nontrivial problem.

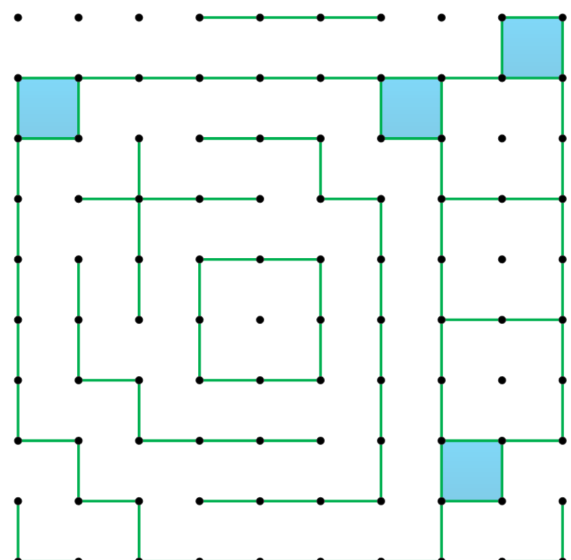


FIGURE 2.1: A large game of Dots-and-Boxes as seen on the cover of [3]

2.1 Dots-and-Boxes

Dots-and-Boxes is a simple, two-player, turn-based game traditionally played using pencil and paper. The game begins with a rectangular grid of dots enclosing a rectangular array of boxes such that each dot forms the vertex of one or more boxes. The players

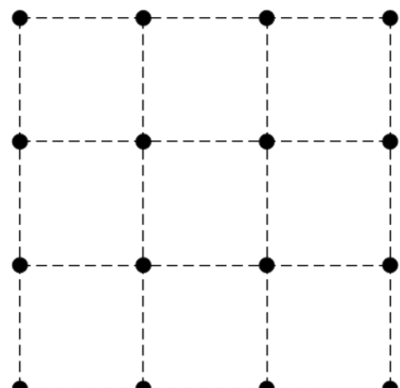


FIGURE 2.2: A 4x4 grid of dots

take turns connecting two vertically or horizontally adjacent dots with lines. Each time the final side of a box has been filled in, the player who drew the line is awarded with the box and another turn. It is compulsory that this additional turn be taken. The game ends once all of possible lines have been drawn in and the player who has earned the most boxes is deemed the winner.

A 4x4 grid of dots containing a 3x3 rectangular array of boxes is shown in figure 2.2.

2.2 Child's play?

Although traditionally played by children, advanced play of Dots-and-Boxes requires complex mathematical analysis. Mathematician Elwyn Berlekamp, recognised as one of the founders of combinatorial game theory, described Dots-and-Boxes as "*The mathematically richest popular child's game in the world, by a substantial margin*" [3].

2.3 Complexity of the game

The total number of lines contained within an $n \times m$ grid of dots is

$$N_e = n(m-1) + m(n-1) \tag{2.1}$$

It can be shown using basic combinatorial mathematics that the total number of possible states containing exactly i lines is $\binom{N_e}{i}$. Thus, the total number of possible states that the board can be in is

$$N_s = \sum_{i=0}^{N_e} \binom{N_e}{i} = 2^{N_e} \tag{2.2}$$

See that the state space of the game increases exponentially with the dimensions of the board. Thus, for large boards, achieving optimal play by a brute-force approach is simply not practical.

2.4 Playing Dots-and-Boxes

An incomplete game of Dots-and-Boxes played between a red player and a blue player is shown in figure 2.3. Recall that drawing the fourth edge of a box awards a player with a box and an additional move. For this reason, an Dots-and-Boxestive strategy would be to avoid filling in the third edge of a box so as not to offer the box to an opponent. See from figure 2.3 that both players follow this Dots-and-Boxestive strategy until a state is reached in panel 14 where it is impossible to make a move which does not offer a box to the opponent. Such a state is known as a state of *gridlock*.

When a state of gridlock has been reached, the board becomes divided into a series of structures called *chains*. Chain structures are strings of boxes with the special property which allows a player to take all the boxes contained within the chain once the third edge of any of the contained boxes has been filled. For this reason, when forced to open a chain, one should aim to open the shortest possible chain so as to offer the least amount of boxes to the opponent.

See from figure 2.3 that, when a state of gridlock is reached in panel 14, the board becomes divided into two chain structures, one containing four boxes and the other containing five. At this point, the red player is forced to open a chain. See in panel 15 that the red player opens the chain containing four boxes, i.e. the shortest possible chain.

The blue player is now free to take all of the boxes contained within the newly-opened chain. Figure 2.4a shows what would happen if the blue player were to take all the boxes on offer. See that, once all the boxes have been taken, the blue player will be forced to open the longer chain to the red player and thus allow the red player to win the game.

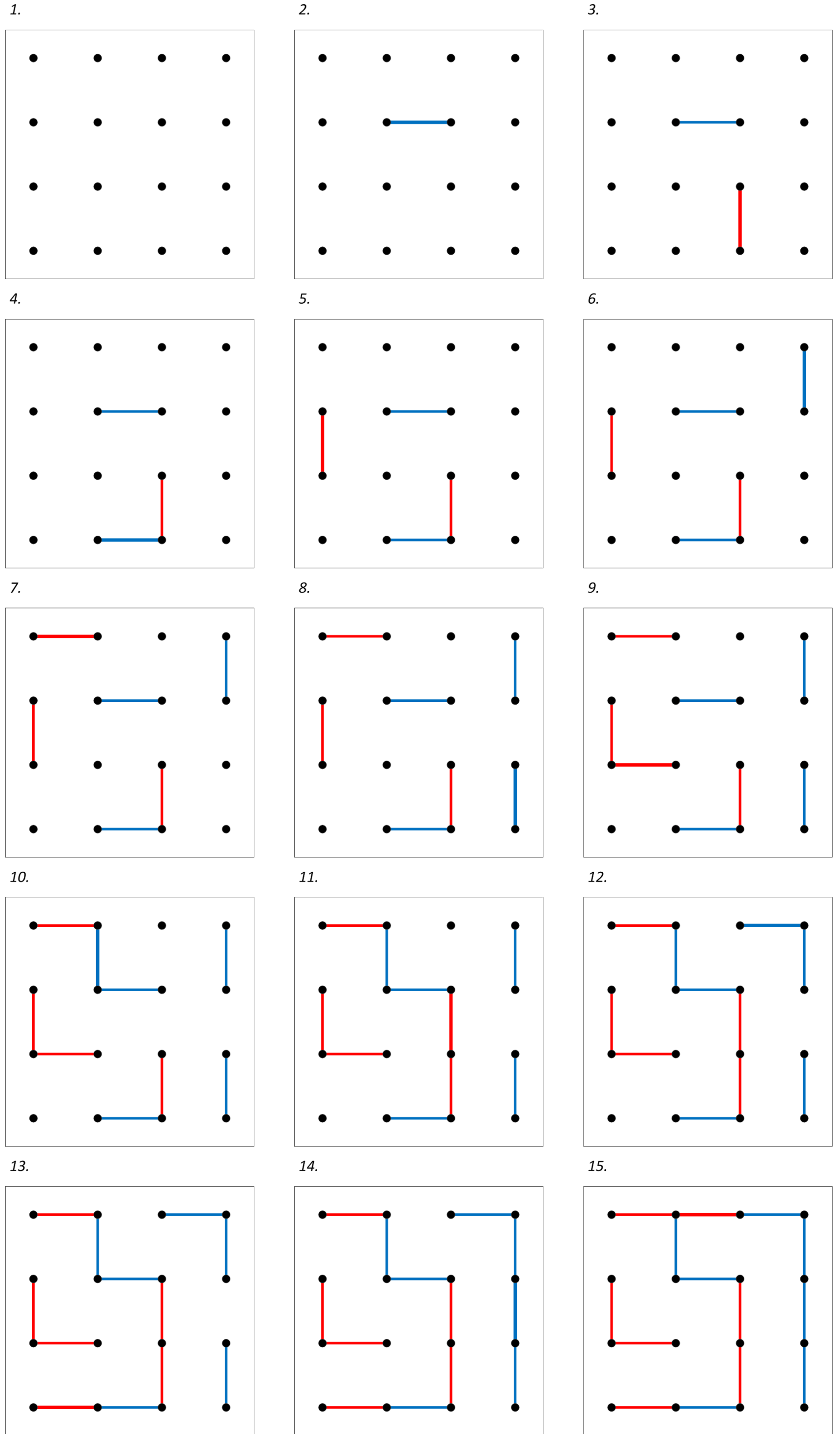
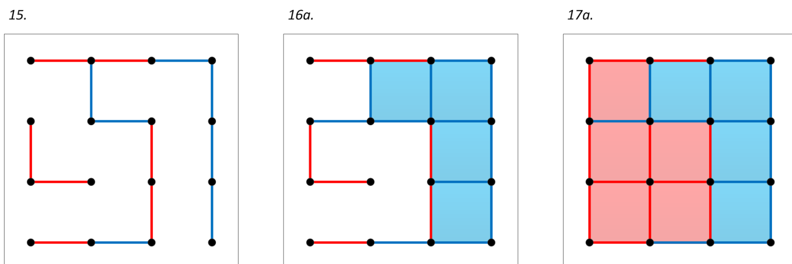
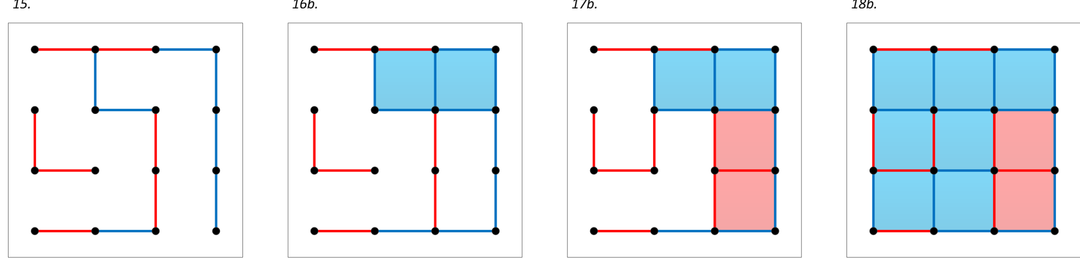


FIGURE 2.3: An incomplete game of Dots-and-Boxes played on a 4x4 grid



(A) The red player wins as a result of the blue player being greedy



(B) The blue player is clever and forces a win using a double-dealing move

FIGURE 2.4: Two possible endings to the game in figure 2.3

Alternatively, the blue player could perform the actions shown in figure 2.4b. See that, when taking the available boxes in the open chain, the blue player declines the last two boxes and hands them over to the red player instead of opening the longer chain. This action is known as a *double-dealing* move. The red player now has no option other than to open the longer chain to the blue player and thus the blue player is able to win the game.

2.5 Mathematical theory of Dots-and-Boxes

Dots-and-Boxes is an impartial game, meaning that, by the Sprague-Grundy theorem, any state of a game of Dots-and-Boxes has a corresponding nimber value. Thus, the theory of Nim can be used in the analysis of the game. Berlekamp describes the use of nimber arithmetic to play Dots-and-Boxes in [3]. The use of nimber arithmetic in the analysis of the game is beyond the scope of this project; however, the theory and architecture presented in this work is flexible enough to support the implementation of higher-level strategies which perform such analysis.

2.6 Creating a program to play Dots-and-Boxes

The concepts of chain structures and double-dealing moves are of paramount importance in advanced play of the Dots-and-Boxes game and it is for this reason that the main focus of this project was to create a program able to detect and analyse chains and to use the double-dealing move to good effect. Such a program requires an architecture which stores information in an effective an utilisable way which facilitates efficient and intricate analysis of the game. The program which was designed to play Dots-and-Boxes at an advanced level was implemented using the functional programming language Haskell. The implementation of this program required the design and realisation of the following modules:

- An architecture of functional data structures which facilitate the analysis of game and player data
- A set of functions capable of performing actions such as chain and pattern detection, data analysis, decision making, etc.
- An action policy which enables the program to select actions that are consistent with the advanced gameplay strategies which will be discussed later in this document
- An interactive user interface to enable testing of the program

The functional data structures which were used in the program are discussed in chapter 3. The source code for this program as well as information on the various functions and data structures is included in appendix D.

2.7 Summary

In this chapter the combinatorial game Dots-and-Boxes was introduced as well as its complexity and theory. In the next chapter, a game which is isomorphic to Dots-and-Boxes, the game of Strings-and-Coins, is introduced.

Chapter 3

Strings-and-Coins

In this chapter, a different way to represent a game of Dots-and-Boxes is presented. This new representation facilitates the analysis and understanding of the game by allowing any state of the game to be expressed as a mathematical graph.

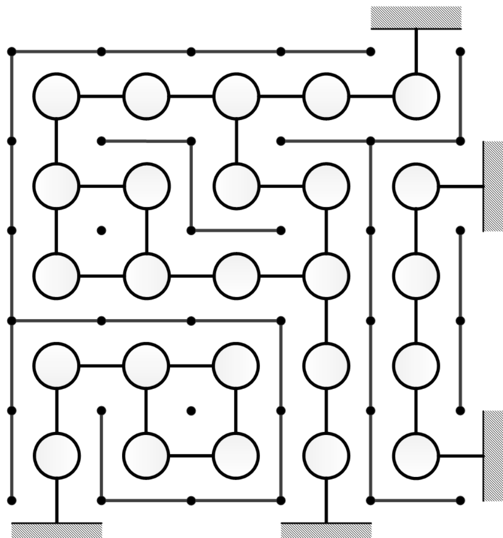


FIGURE 3.1: A game of Dots-and-Boxes shown with its corresponding game of Strings-and-Coins

3.1 Strings-and-Coins

Consider a set of coins interconnected with strings such that each string either connects two coins together or connects a coin to a wall. Now consider a game in which two players take turns cutting these strings. Once a coin has all of its strings removed, the

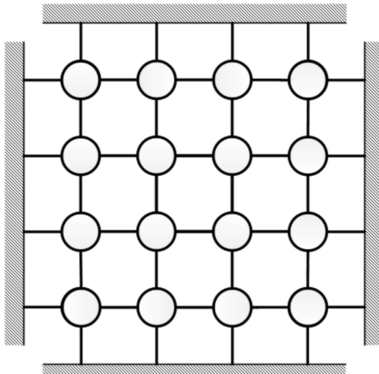


FIGURE 3.2: A game of Dots-and-Boxes expressed in Strings-and-Coins representation

player who made the final cut is awarded with the coin and an additional turn. The game ends once all of the strings have been cut and the player who has collected the most coins is deemed the winner. This game is called Strings-and-Coins and was invented by Elwyn Berlekamp.

A game of Dots-and-Boxes is in fact mathematically equivalent to a special case of Strings-and-Coins where each coin begins with exactly four attached strings and the set of coins is arranged in a rectangular array bordered by walls as shown in figure 3.2. Any arbitrary game of Dots-and-Boxes therefore has a corresponding Strings-and-Coins representation. A game of Dots-and-Boxes is shown with its corresponding game of Strings-and-Coins is shown in figure 3.1.

The great benefit of the Strings-and-Coins representation of a game of Dots-and-Boxes is that it allows the game to be expressed as a mathematical graph where the coins and walls form the set of nodes and the strings form the set of edges. This graph representation makes patterns and structures much easier to recognise and thus facilitates the overall understanding of the game and its concepts.

3.2 Classification of nodes

At this point it is beneficial to categorise the many different types of node which can occur during a game of Strings-and-Coins. First, each node should be classified as either a coin node or a wall node. Coin nodes and wall nodes are defined as follows:

Definition 3.1 (Coin)**.** A coin is a node which, once all its edges have been removed, awards the player who removed the final edge a point and an additional turn.

Definition 3.2 (Wall)**.** A wall is a node with a single edge connecting it to a coin node. Unlike a coin, removing all the attached edges of a wall has no additional effect.

The following classes of node were defined as part of this project. It was found that the categorization of nodes into these classes was of great use in the analysis of the game.

Definition 3.3 (Singleton)**.** A singleton is a coin with a single edge connected to it.

Definition 3.4 (Binode)**.** A binode is a coin which has exactly two edges connected to it.

Definition 3.5 (Multinode)**.** A multinode is a coin which has at least three edges connected to it.

Each class of node has a distinct set of associated properties which will be discussed later in this document. The symbols which hereby will be used to graphically represent these objects are shown in figure 3.3 below.

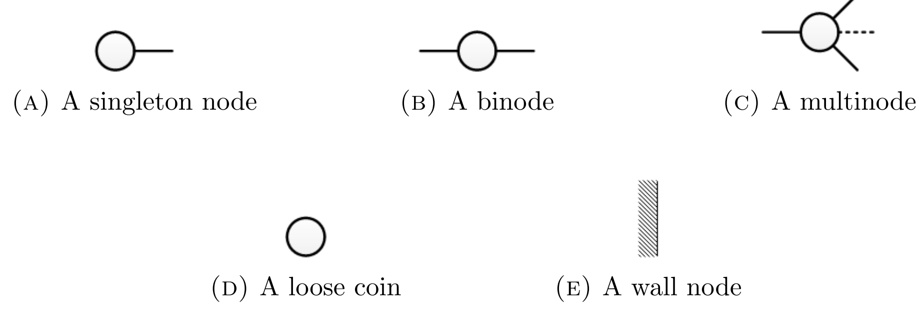


FIGURE 3.3: The five different classes of node used as part of this project

3.2.1 The wall node

Wall nodes play a special role in the game of Strings-and-Coins. An edge which connects two coins together will have a direct effect on both coins when removed. However, an edge which connects a coin to a wall node will effect only that coin when removed. Wall nodes play an important role in the strategy of the game and are a key component in structures called *simple chains*, which will be discussed in the next chapter.

Although there is no parallel to a wall node in a game of Dots-and-Boxes, their effects are summarised by the existence of lines which form the side of only one box.

3.3 Strings-and-Coins in functional programming

When designing a program to play Dots-and-Boxes, it was found that an architecture based on the game of Strings-and-Coins greatly eased the implementation and analysis of the game.

3.3.1 Node and edge data structures

```
data Node = Wall                      -- A wall node
          | EmptyNode                 -- Coin with no attached edges
          | Singleton EdgeId          -- Coin with a single attached edge
          | BiNode (EdgeId, EdgeId)   -- Coins with two attached edges
          | MultiNode [EdgeId]        -- Coins with three or more attached edges
```

The Haskell code above shows how node data structures were defined in the program. See that nodes are divided into the five categories shown in figure 3.3. See also that they are defined clearly and concisely; this is one of the many strengths of the functional programming approach.

Coin objects contain key values called *EdgeId*s which uniquely identify the edges which are connected to them. The edge objects themselves are stored in a special type of lookup table known as a *map*. The map data structure is another highly utilisable member of the Haskell toolkit; its $O(log\, n)$ search time allowed the program to be highly scalable. Using the appropriate *EdgeId*, any edge stored within the map can be accessed and modified. Like edge objects, nodes are also stored in a map structure and each node can be accessed using its associated key value called a *NodeId*.

Specifically, map objects which store node objects were called *NodeMap*s and map objects which store edge objects were called *EdgeMap*s. For convenience, a data structure called a *Database* was defined to store both a *NodeMap* and an *EdgeMap* in a single data structure. All the information about the current state of a game could be found in its associated *Database*. More information on *Map* data structures in Haskell can be found in appendix C.

type $Edge = (NodeId,\ NodeId,\ Bool)$

The above code shows how edge data structures were defined in the program. See that all edge objects contain three values: two key values which uniquely identify the nodes to which they connect, and a Boolean variable which indicates whether or not the edge has been cut.

More information on these node and edge objects can be found in section D.3.1.1.

3.3.1.1 Effectiveness of this architecture

Several different architectures were designed over the course of this project before the above architecture was reached. Due to its simplicity and flexibility, this architecture greatly facilitated the design of the functions and algorithms which analyse the current game state and it allowed the program to become more sophisticated without creating any unnecessary obstacles.

3.3.2 Traversing the graph

The architecture defined in the previous section allows the graph which expresses a game of Strings-and-Coins to be traversed with ease. Let i be a key value which uniquely identifies some coin n_i on the board. The value of i can be used to reference its associated coin stored in the *NodeMap* and thus it can be used to retrieve E_i, the set of all edges which connect to n_i. The program can hop to a neighbouring node by following any of the edges in E. As an edge object, e, contains the key values of the two nodes to which it connects, n_a, n_b, the key value of the neighbouring node which can be reached by following e is the only element in the set $\{a, b\}$ whose value is not equal to i. This key value can then be used to retrieve the neighbouring node from the *Database*. The $O(log\, n)$ search time of map data structures allows the graph to be traversed quickly and efficiently.

More information on graph traversal can be found in section D.3.4.

3.4 Structures

The term "structure" has been used loosely in this document thus far. Now that the Strings-and-Coins representation of the game has been discussed in detail, structures can be formally defined with respect to this representation. The definition of a structure is stated as follows:

Definition 3.6 (Structure)**.** A structure is a set of interconnected nodes S such that, for any pair of nodes, $n_i,\ n_j \in S,\ i \neq j$, there exists at least one path which connects them.

At the beginning of each game of Dots-and-Boxes, there exists at least one path connecting each arbitrary pair of nodes on the board. The entire board thus forms one large structure. However, as the game progresses and edges are removed, this structure will typically become divided into several independent structures. Recall that, once a state of gridlock has been reached, the board becomes divided into a series of structures known as chains. Such structures will be discussed in detail in the next chapter.

Chapter 4

Chains

The concept of a chain is the cornerstone of the analysis of the Dots-and-Boxes game and advanced play requires a deep and thorough understanding of chain structures and their properties. In this chapter, chain structures are studied in detail as well as the strategies which involve them.

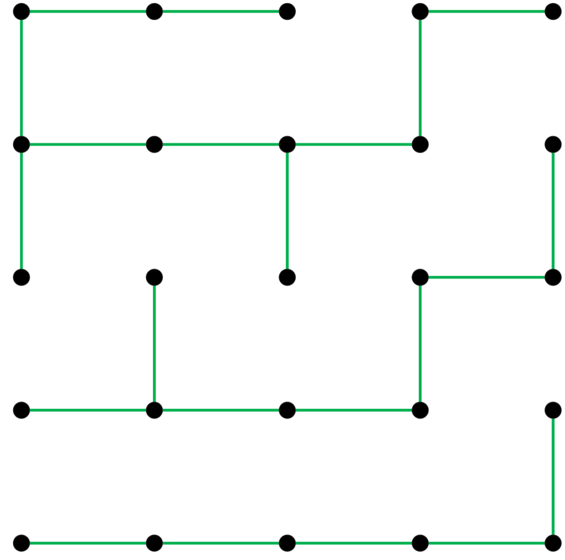


FIGURE 4.1: A series of chains

4.1 Chains

Recall that when a game of Dots-and-Boxes reaches a state of gridlock, the board becomes divided into a series of chains. Figure 4.1 shows an example of such a board. The

same board is shown in Strings-and-Coins representation in figure 4.2.

See that the chain structures are much easier to recognise in this figure. This is one of the great strengths of the Strings-and-Coins representation of the game.

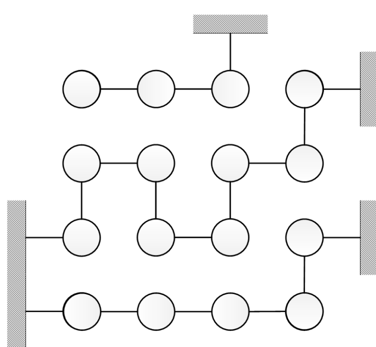


FIGURE 4.2: The Strings-and-Coins representation of figure 4.1

In addition to making chain structures easier to recognise, this representation also demonstrates clearly what is contained within a chain structure: a sequence of binodes. See that all the chains in figure 4.2 are made up of binodes and one of which contains a singleton node. Using this observation, a chain can be defined formally as follows:

Definition 4.1 (Chain)**.** A chain is a continuous string of singleton nodes or binodes.

4.1.1 Length of a chain

Since each chain contains a set of coins, it is useful to define a quantity which measures the cardinality of this set. This quantity is known as the *length* of the chain.

Definition 4.2 (Length of a chain)**.** The length of a chain is the number of coins contained within the chain.

In order to keep the notation concise, a chain of length n will be referred to as an n-chain. The definition of an n-chain is stated formally as follows:

Definition 4.3 (n-chain)**.** An n-chain, where $n \in \mathbb{N}$, is a chain which contains n coins.

4.1.2 Categories of chain

As part of this project, it was found that chains could be categorised into three distinct classes: simple chains, loop chains, and complex chains. Each class of chain has a different set of properties which affect the strategies of the game.

The first class of chain which will be examined is the *simple chain*. Simple chains are defined as follows:

Definition 4.4 (Simple chain)**.** A simple chain is a chain which is bounded by wall nodes.

All of the chains which will be looked at in this chapter are simple chains. The other types of chain will be discussed in the later chapters of this document.

4.2 Open chains

A chain containing a singleton node is known as an *open chain*. Figure 4.3 shows an example of such a chain. Open chains have a special property which enables a player to take all of the coins contained within the chain. The reason for this is that, once the singleton coin is freed from the end of the chain, a new open chain is produced and the player is awarded an additional turn. With this additional turn the player is able to take the coin from the end of the new open chain and this process repeats until all the coins in the chain are removed. This process is depicted in figure 4.4.

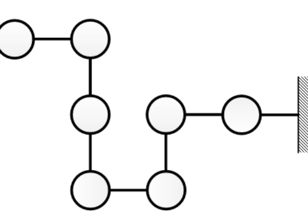


Figure 4.3: An open chain

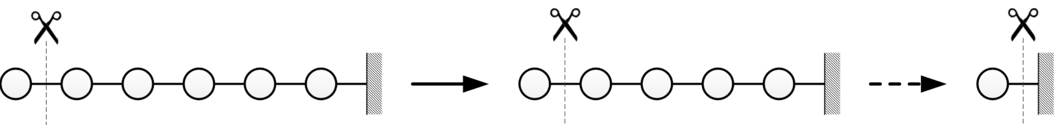
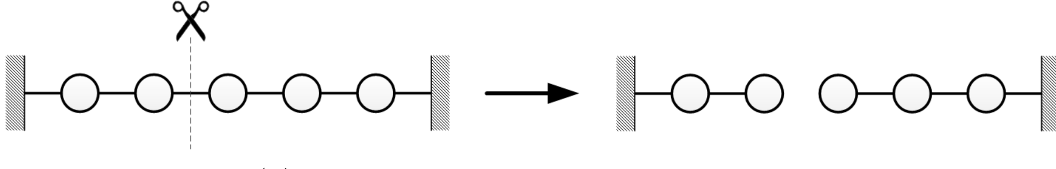


Figure 4.4: Taking the coins from an open chain

4.2.1 Opening chains

In contrast to an open chain, a *closed chain* is a chain which does not contain any singleton nodes and thus no coins can be taken from such a chain during the next move. However, if one of the edges contained within a closed chain is cut, the chain will be reduced to one or two open chains depending on where the cut was made. This can be seen clearly in figure 4.5.

Since opening a chain allows the opponent to take all of the coins contained within the chain, an intuitive strategy when forced to open a chain would be to open the smallest possible chain so as to minimise the number of coins offered to the opponent. This strategy is stated formally in strategy 4.1 as follows.



(A) Reducing a closed chain to two open chains

(B) Reducing a closed chain to a single open chain of the same length

Figure 4.5: Opening a chain

Strategy 4.1 (Opening chains)**.** *When forced to open a chain, one should open the shortest possible chain so that the smallest possible number of coins are offered to the opponent.*

4.3 Double-dealing

Consider the situation in figure 4.6. The board was in a state of gridlock and the opposing player was forced to open a chain. Following strategy 4.1, the opponent decided to open the shortest chain. The player is now free to take the coins contained within this chain. However, if the player were to be greedy and take all of the available coins, he/she would be forced to open the longer chain and thus allow the opponent to win the game. In this situation, a good strategy for the player would be to perform an action which passes the turn on to the opponent without opening the longer chain. There are two possible actions which would achieve this and these actions are shown in figure 4.7. See that performing either action in figure 4.7 would result in a scenario in which the opposing player would eventually have no choice but to open the long chain. Any move that the opponent can make either frees a coin and gains an additional move or opens the long chain and passes the turn to the player. Thus by the end of the opponent's turn the long chain will have been opened and the player will be able to take the available coins and win the game. The player is therefore in a winning position.

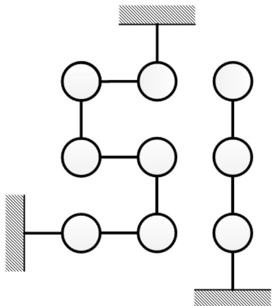
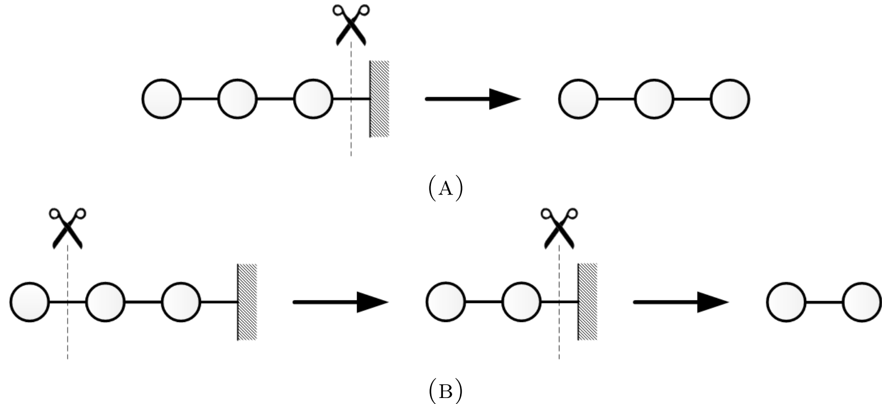


Figure 4.6: A winning position



(A)

(B)

FIGURE 4.7: Sacrificing some coins in order to force the opponent to open a longer chain

Although either action in figure 4.7 would result in a situation in which the player can win the game, the action in figure 4.7a is somewhat foolish; it sacrifices more coins than is necessary in order to force a win. The action in figure 4.7b is the best possible action in this situation; it reduces the open chain to an open 2-chain and then detaches the chain from the wall node to form a special chain composed of two singleton nodes sharing one common edge. In [3], Berlekamp refers to this structure as a *double-cross*, however in this document the structure will be referred to as a *dipole*.

A move which produces a dipole and passes the turn on to the opponent is known as a *double-cross* move or a *double-dealing* move. Double-dealing moves are of great importance in advanced play of the Dots-and-Boxes game and it is for this reason that the main focus of this work was to create a program which is able to use this move to good effect.

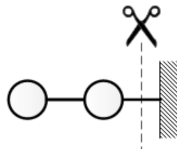


FIGURE 4.8: A double-dealing move

4.4 The hard-hearted handout

Note that in order to perform a double-dealing move, an open chain with a length of at least two nodes is required. A double-dealing move cannot be performed on an open 1-chain because the chain does not contain enough coins to produce a dipole.

FIGURE 4.9: A closed 2-chain

Consider the closed 2-chain shown in Dots-and-Boxes representation in figure 4.9. When forced to open such a chain, an inexperienced player might not give much thought into

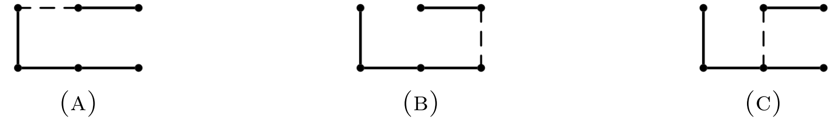


FIGURE 4.10: The three possible ways to open the 2-chain (Dots-and-Boxes)

choosing which edge to fill as either way the opponent will be given the opportunity to take the two boxes contained within. There are three possible moves which will open this chain and these moves are shown in figure 4.10. At first glance there may not seem to be a difference between these three actions; however, choosing the right move in this situation could be the difference between victory and defeat.

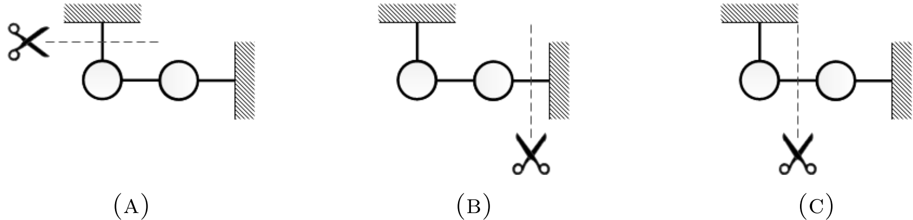


FIGURE 4.11: The three possible ways to open the 2-chain (Strings-and-Coins)

Figure 4.11 shows the same actions in Strings-and-Coins representation. In this representation the consequences of each action are much more obvious: actions (A) and (B) produce an open 2-chain whereas action (C) produces two open 1-chains. Recall that a double-dealing move requires an open chain containing at least two nodes. Thus the opponent will given a double-dealing opportunity if actions (A) or (B) are taken. With this opportunity the opponent may have the chance to force a win, and for this reason actions (A) and (B) can be considered to be very foolish moves. If action (C) is chosen, then no such opportunity is presented to the opponent, as double-dealing moves require open chains with lengths of at least two coins.

The move in action (C) is referred to as a *hard-hearted handout* [3]. Hard-hearted handouts are defined as follows:

Definition 4.5 (Hard-hearted handout)**.** A hard-hearted handout is an action which divides a closed 2-chain into two open 1-chains.

In contrast, actions (A) and (B) are referred to as *half-hearted handouts.* A double-dealing opportunity should never be presented to an opponent free of charge and thus the following strategy should always be followed when opening 2-chains:

Strategy 4.2. *[Opening 2-chains] When opening a 2-chain, a hard-hearted handout should always be performed.*

4.5 Short chains and long chains

Under the assumption that both players follow strategy 4.2, then opening a 2-chain will never result in a double-dealing opportunity. There is no way to open a 3-chain such that the opponent will not be given such an opportunity, however. In fact, opening a chain of length greater than 2 nodes will always result in a double-dealing opportunity. For this reason, chains of length three or more coins are referred to as *long chains* and chains shorter than three coins are referred to as *short chains*. Short chains and long chains are defined formally in definitions 4.6 and 4.7 respectively.

Definition 4.6 (Short chain)**.** A short chain is a chain containing less than three coins.

Definition 4.7 (Long chain)**.** A long chain is a chain containing at least three coins.

4.6 Control

Consider a situation where the board has become divided into a large set of long chains and an open 2-chain as shown in figure 4.13. See that, if the player were to perform a double-dealing move on the open 2-chain, the opponent would be forced to open one of the long chains. Since an open long chain will always present a double-dealing opportunity, the player is able to force the opponent to open each long chain by repeatedly double-dealing on each open long chain. If the player is in a situation in which the opponent can be forced to open each long chain by double-dealing appropriately, then the player is considered to have *control* over the opponent.

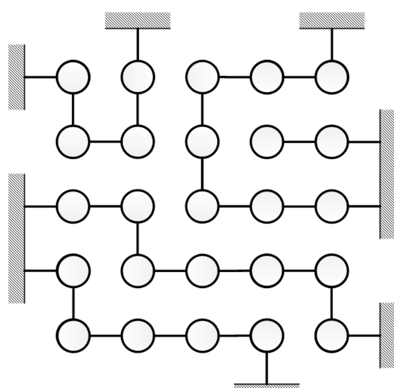


FIGURE 4.12: A double-dealing opportunity and a series of long chains

In the situation in figure 4.12, the player is actually in a winning position. By repeatedly double-dealing and forcing the opponent to open each long chain, the player is able to

take enough coins to win the game. The cost of two coins per double-cross is a small price to pay for maintaining control over the opponent. Note that a double-dealing move need not be performed on the last open chain as the game will end once all the coins in this chain are taken. By following this strategy of repeatedly double-dealing in order to maintain control over the opponent, the player should win the game with a score of 17 coins to 8.

4.7 Detecting chains

One of the main requirements for a computer program to play Dots-and-Boxes at an advanced level is that it should be able to recognise chain structures. See that, from definition 4.1, each binode has an associated chain. Therefore, with the exception of open 1-chains and dipoles, all chains on the board can be detected by taking each binode and following the path in either direction until the bordering nodes are reached. This is illustrated in figure 4.13.

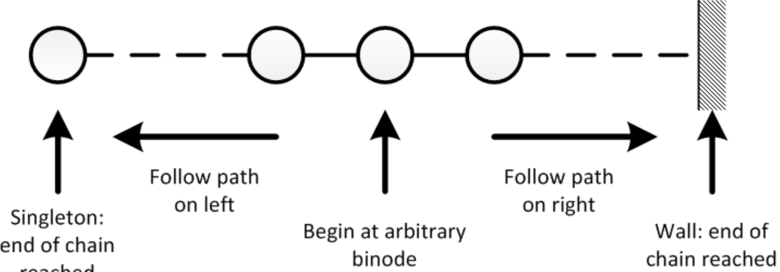


FIGURE 4.13: Detecting a simple chain

Note that, in this context, a path is a string of binodes which ends once any other type of node is reached.

4.7.1 Chain data structures

In order to make use of a detected chain, the information must be stored in an appropriate data structure. When designing the program, it was found that it was sufficient to express a simple chain as a set of coin nodes with a label indicating the chain type (open, closed). A chain structure can thus be represented with the following notation: $\langle S \rangle_x$, where S is the set of coins contained within the chain and x is the label.

This notation is very useful for describing chain structures as it is simple, flexible, and can be easily represented by a functional data structure.

4.7.1.1 Chain data structures in functional programming

type *Chain* = ([*NodeId*], *Label*)
data *Label* = *Open* | *Closed*

The above code shows how such chain structures can be defined using functional programming. See that a chain data structure is composed of a list of *NodeId*s which represent the coins which are contained within the chain as well as a label which indicates if the chain is open or closed. These structures are simple, concise, yet powerful and they closely follow the mathematical notation from the previous section.

4.7.2 The chain detection algorithm

As a chain is modelled as a set of coins and a label, a chain detection algorithm must work out which coins are contained in such a set and also the label which describes the chain. The algorithm which was designed to complete these tasks is stated in algorithm 4.1. As the program was written using functional programming, algorithm 4.1 is presented as a mathematical function.

Algorithm 4.1 (Detecting simple chains)**.** *Given a binode b with neighbours n_1, n_2, the chain which contains b is the output of $f(n_2, f(n_1, \langle S_0\rangle_c))$ where $S_0 = \{b\}$ and the function $f(\cdot)$ is defined as follows:*

$$f(n, \langle S\rangle_x) = \begin{cases} f(n', \langle S \cup \{n\}\rangle_x), & isBinode(n) \\ \langle S \cup \{n\}\rangle_o\,, & isSingleton(n) \\ \langle S\rangle_x\,, & otherwise \end{cases}$$

where n' is the next neighbour of a binode n, $\langle\cdot\rangle_c$ denotes a closed chain, and $\langle\cdot\rangle_o$ denotes an open chain.

See that the algorithm is centred around the function $f(\cdot)$. $f(\cdot)$ is a recursive function which takes both a node and a chain structure as arguments and returns a chain structure

as an output. The type of the input node n determines the operation of the function. If the end of the chain has been reached, i.e. if n is not a binode, then the function will terminate; otherwise the function will recurse and examine the next node.

If n is a singleton, then the algorithm recognises that the end of the chain has been reached. It adds n to the set S and returns an open chain containing S. The chain is labelled as an open chain because any chain containing a singleton node is open.

If n is neither a singleton nor a binode, the algorithm concludes that a wall node or a multinode has been reached and returns the same chain structure which was input.

If n is a binode then the end of the chain has not been reached so the algorithm adds n to the set S and then calls itself with n' and S as arguments where n' is the node obtained by following the next edge of n.

See that the algorithm calls the function $f(\cdot)$ two times, once for each path (see figure 4.13). The output of the first call will be a chain structure containing the initial binode b and the sub-chain worked out by following the first path. This chain is then input to the second call of $f(\cdot)$ along with the second neighbour of b. The final output of the algorithm will be a chain structure containing a set $S = \{b\} \cup S_1 \cup S_2$ where S_1 is the set of coins from the first path and S_2 is the set of coins from the second path. Note that the chain structure which was input to the first call of $f(\cdot)$ is labelled as a closed chain. If either call of $f(\cdot)$ is terminated by a singleton node then the final output will be an open chain; otherwise the chain will remain a closed chain.

4.7.2.1 Chain detection in functional programming

Algorithm 4.1 can be expressed in Haskell code as follows:

$$
\begin{aligned}
&chainDetect\ (b,\ BiNode\ (e_1,\ e_2))\ =\ foldl(\lambda\ acc\ n\ \rightarrow\ f\ n\ acc) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad ([b], Closed)\ [n_1, n_2] \\
&\qquad\qquad \textbf{where} \\
&\qquad\qquad\quad [n_1, n_2]\ =\ getNeighbours\ [e_1, e_2]\ b \\
&f\ n\ c@(ns,\ x) \\
&\qquad\qquad |\ isBiNode\ n\ =\ f\ (nextNode\ n)\ (n : ns,\ x) \\
&\qquad\qquad |\ isSingleton\ n\ =\ (n : ns,\ Open) \\
&\qquad\qquad |\ otherwise\ =\ c
\end{aligned}
$$

A *fold* function in functional programming takes some accumulator value and modifies it for each element of some input list. In this case the accumulator is the chain structure being calculated and the input list is the list of neighbours of the binode b. See that the chain structure is initialised as $([b], Closed)$, i.e. a closed chain containing the node b.

The above code implements algorithm 4.1 in a concise and elegant way. One of the great strengths of functional programming is its toolkit of flexible and highly utilisable functions such as *foldl* which keep the code short and concise. The implementation of algorithm in 4.1 in an imperative language would typically be longer and less elegant.

4.7.3 Gathering all the chains

Recall that algorithm 4.1 takes a binode b and its two neighbouring nodes n_1, n_2 and returns the chain which contains b. Thus the set of all chains on the board, C, can be obtained by calling algorithm 4.1 for every binode. However, as a chain can contain several binodes, calling algorithm 4.1 for each binode would result in the set C containing duplicates. Therefore, before calling algorithm 4.1 for a binode b, one must first ensure that b is not contained in any of the previously detected chains in C. If so, then b can be discarded; otherwise, the chain containing b should be worked out and added to the set C. The function *getChains* performs this action. See section D.4.2.2 for more details.

Recall that open 1-chains and dipoles will not be detected by this process. This is not a problem, however, as these types of chain are not needed when analysing the current state of the board. This is discussed further in chapter 8.

4.8 Summary

In this chapter, simple chain structures and their properties were examined in detail. Strategies involving such chains were discussed and an algorithm to detect these chains was presented. The double-dealing move, one of the key concepts in this project, was introduced and it was shown how this move could be used to gain control over the opponent.

Chapter 5

Loops

In this chapter a new type of chain, called the loop chain, is presented. Loop chains have certain properties which add a whole new level of complexity to the game and its analysis.

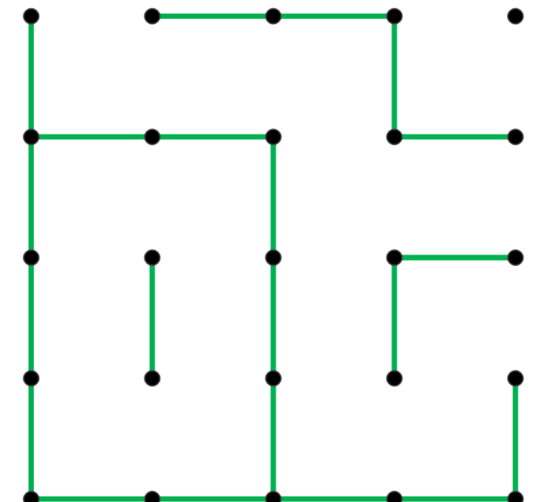


FIGURE 5.1: A game of Dots-and-Boxes with a loop

5.1 The loop chain

All of the chains which have appeared so far in this document have been simple chains, i.e. chains bounded by wall nodes. These structures are, however, not the only type of chain which can appear in a game of Dots-and-Boxes.

Consider a chain which is not bounded by endpoints. Instead the chain forms a closed loop of binodes. Such chains are known as *circular chains* or *loops* and are a common occurrence in large grids. Loop chains are defined formally as follows:

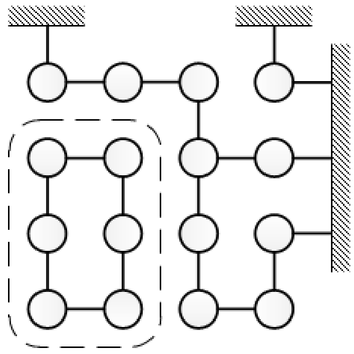


FIGURE 5.2: A loop chain

Definition 5.1 (Loop chain)**.** A loop chain is a structure composed entirely of binodes which forms a closed loop.

Figure 5.1 shows an example of a game of Dots-and-Boxes which contains a loop. The same game is shown in Strings-and-Coins representation in figure 5.2.

Not only does the circular nature of loop chains increase the complexity of processes such as chain detection, it also has a profound effect on the gameplay. This effect is illustrated in the next section.

5.2 Double-dealing with loops

Consider the typical double-dealing opportunity involving an open 2-chain shown in figure 5.3. This structure is composed of a singleton node and a wall node on either side of a binode. When the double-dealing move is performed, no coins are freed and thus the turn is passed to the opponent. Since a loop chain is not bounded by endpoints, the structure which is formed as a result of opening such a chain will never resemble the structure in figure 5.3. Because a loop chain is composed entirely of binodes, opening a such a chain will result in an open chain composed of a string of binodes bounded by two singleton nodes, as shown in figure 5.4 This has major implications with regards to a double-dealing strategy.

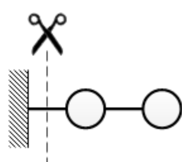


FIGURE 5.3: A double-dealing opportunity

Consider the open chain in figure 5.5 which resulted from opening a loop and removing a number of nodes. In this situation, is it possible to perform an action which would result in passing the turn to the opponent? Upon inspection one should conclude that it is not possible. The reason for this is that, unlike the situation in figure 5.3, it is

FIGURE 5.5: An open loop composed of three nodes

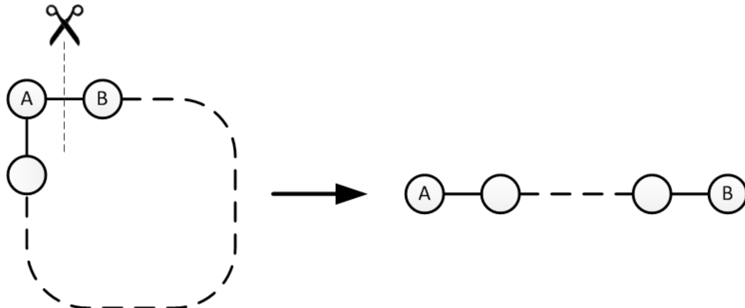


FIGURE 5.4: Opening a loop chain

impossible to perform a move which would not result in freeing a coin, thus earning an additional move. It *is* possible to double-deal on open loop chains, but it requires that the open loop consists of at least four nodes.

Figure 5.6 shows a double-dealing move being performed on an open loop. Like the double-dealing move in figure 5.3 this move would not free any coins and thus the turn would be passed to the opponent, however it is much more costly; such a move would result in a handout of four coins to the opponent, double that of a typical double-dealing move. The reason for this is that, when a loop chain is opened, the result is an open chain with two *loose ends*. The term *loose end* is defined in definition 5.2. Such is the nature of a loose end that in order to perform a double-dealing move on an open chain, it requires a handout of one dipole for each loose end. For this reason, it requires that an open loop must contain at least four nodes in order for a double-dealing move to be possible. Due to the rectangular nature of the Dots-and-Boxes game, all loop chains will contain at least four nodes and thus a double-dealing move can be performed on any loop chain. Open simple chains, such as the one shown in figure 5.3, contain exactly one loose end and thus a double-dealing move would require a handout of a single dipole.

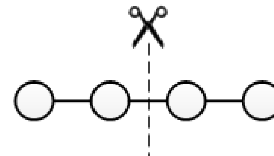


FIGURE 5.6: Double-dealing on an open loop

Definition 5.2 (Loose end)**.** A loose end is a singleton node which forms the end of an open chain.

5.3 Detecting Loops

Due to the circular nature of loop chains, the chain detection algorithm stated in 4.1 would enter an infinite recursive loop if it were to attempt to detect one. Thus to

allow for loop chains, algorithm 4.1 should be modified so that it is able to recognise nodes which have already been encountered. An algorithm with such a modification is presented in algorithm 5.1.

Algorithm 5.1 (Chain detection with loops)**.** *Given a binode b with neighbours n_1, n_2, the chain which contains b is the output of $f(n_2, f(n_1, \langle S_0 \rangle_c))$ where $S_0 = \{b\}$ and the function $f(\cdot)$ is defined as follows:*

$$f(n, \langle S \rangle_x) = \begin{cases} \langle S \rangle_l, & n \in S \\ f(n', \langle S \cup \{n\} \rangle_x), & isBinode(n) \quad and \quad n \notin S \\ \langle S \cup \{n\} \rangle_o, & isSingleton(n) \quad and \quad n \notin S \\ \langle S \rangle_x, & otherwise \end{cases}$$

where n' is the next neighbour of a binode n, $\langle \cdot \rangle_c$ denotes a closed chain, $\langle \cdot \rangle_o$ denotes an open chain, and $\langle \cdot \rangle_l$ denotes a loop chain.

Algorithm 5.1 works similarly to algorithm 4.1 only that the chain detection function $f(\cdot)$ has been modified to allow for loop chains. It does this by checking to see if the input node n is already contained within the input chain structure $\langle S \rangle$. If so, then the node n has been visited before in a previous call to $f(\cdot)$ and thus the chain structure must be circular in nature. This is illustrated visually in figure 5.7.

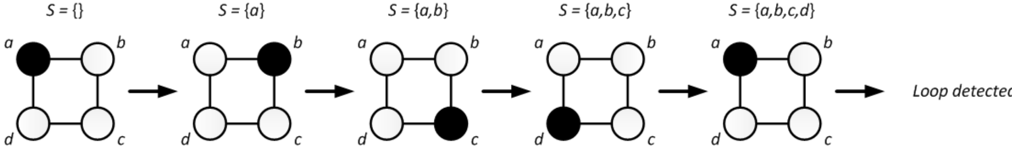


FIGURE 5.7: Loop detection

Note that $f(\cdot)$ is called for the first neighbouring node n_1 and then the output of this function is fed into another call of $f(\cdot)$ alongside the second neighbouring node n_2. If the chain in question is circular, then the entire chain will have been recognised after the first call of $f(\cdot)$ and thus second call of $f(\cdot)$ will return the same chain which was input.

If one were to think of a chain structure as a trail of breadcrumbs, then algorithm 5.1 can be thought of as a process which picks up these breadcrumbs and places them into a

basket. The process ends once all the breadcrumbs have been collected and the resulting basket of breadcrumbs is the output chain structure.

5.3.1 Circular chain detection in functional programming

Algorithm 5.1 can be expressed in Haskell code as follows:

$$\begin{aligned} \mathit{chainDetect}\ (b,\ \mathit{BiNode}\ (e_1,\ e_2)) \ &= \ \mathit{foldl}(\lambda\ \mathit{acc}\ n\ \rightarrow\ f\ n\ \mathit{acc}) \\ &\qquad\qquad\qquad ([b], \mathit{Closed})\ [n_1, n_2] \end{aligned}$$

$$\begin{aligned} &\textbf{where} \\ &\quad [n_1, n_2] \ = \ \mathit{getNeighbours}\ [e_1, e_2]\ b \end{aligned}$$

$$\begin{aligned} &f\ n\ c@(\mathit{ns},\ x) \\ &\qquad \mid\ \mathit{elem}\ n\ \mathit{ns}\ =\ (\mathit{ns},\ \mathit{Loop}) \\ &\qquad \mid\ \mathit{isBiNode}\ n\ =\ f\ (\mathit{nextNode}\ n)\ (n : \mathit{ns},\ x) \\ &\qquad \mid\ \mathit{isSingleton}\ n\ =\ (n : \mathit{ns},\ \mathit{Open}) \\ &\qquad \mid\ \mathit{otherwise}\ =\ c \end{aligned}$$

See that only a slight adjustment is required to allow for circular chains. The function *elem* is from the Haskell toolkit and returns a Boolean value indicating whether or not an object is a member of some input list.

5.4 Summary

In this chapter, loop chains and their properties were examined in detail. The effect of loop chains on double-dealing moves was discussed and algorithm 4.1 was modified to allow for such chains.

Chapter 6

The Double-Dealing Decision

The increased handout for open loop chains adds a new level of complexity to the decision of whether or not to double-deal if given the opportunity. The cost of two coins per double-deal was considered a small price compared to the sizeable reward which could be gained from retaining control over an opponent. The presence of loop chains, however, increases this cost to such an extent that it is often more profitable to hand control over to the opponent than to retain it.

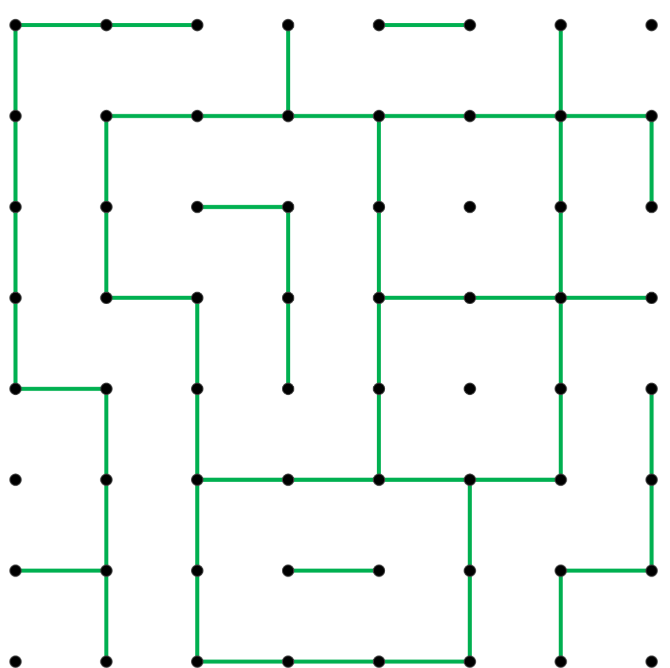


FIGURE 6.1: A difficult decision

Consider the situation in figure 6.1. A decision of whether or not to double-deal has presented itself (see the open 2-chain in the top right corner). The question is: is it more

profitable to perform the double-dealing move or to take the coins and open the next chain? This is a difficult question and cannot be answered with certainty without prior knowledge of the opponent's strategy. An informed decision can be made, however, based on the minimum achievable reward which can be gained by applying a certain strategy.

6.1 Reward from the long chains

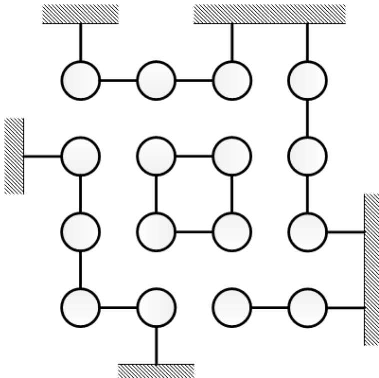


FIGURE 6.2: A 4x4 grid of coins with four long chains

Before this decision can be made, the situation must be analysed correctly. Such an analysis requires that different features of the game board be examined individually. The first feature which will be investigated is that of the long chains. If a player's opponent has been forced to open the first long chain, then the player has gained control of the long chains. See that double-dealing on the open 2-chain in figure 6.2 would result in a gain of control. Recall that, when a player is in control of the long chains, the player can force the opponent to open each long chain by double-dealing appropriately. Gaining control of the long chains yields a certain reward, which will be denoted as r_l. Under certain assumptions, the value of this reward can be predicted. The first assumption which will be made is that the opponent plays at a high level and won't give away coins needlessly. The player must therefore adopt a strategy which guarantees a minimum yield from the long chains, given that the player has control. This strategy is stated as follows:

Strategy 6.1 (Control of long chains)**.** *When presented with an open, long chain, one should take as much coins as possible before double-dealing and forcing the opponent to*

open the next long chain. If, however, the final long chain has been opened, one should take all the coins contained within the chain.

The strategy which the opponent is assumed to use is as follows:

Strategy 6.2 (Without control of long chains)**.** *When forced to open a long chain, the shortest available chain should be opened. Priority should be given to loop chains over non-circular chains of the same length.*

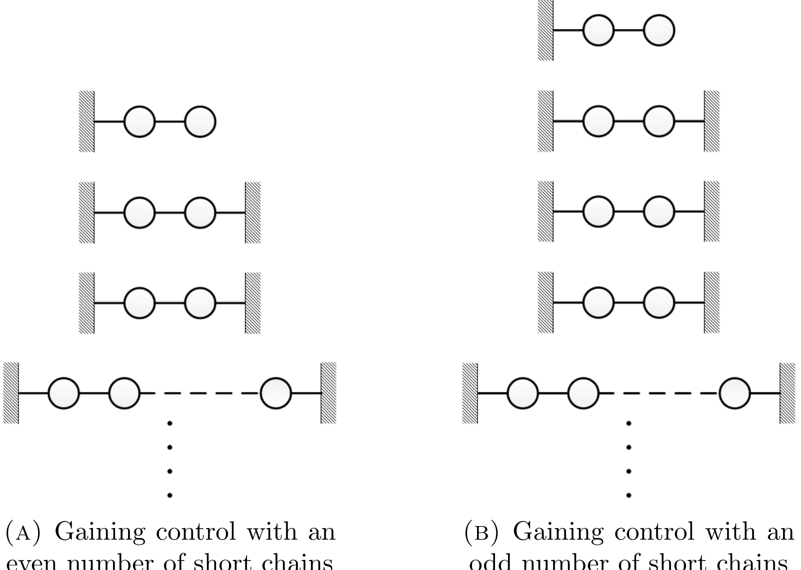
The reason why loop chains are given priority over non-circular chains of the same length is because loop chains yield a greater reward if the opponent were to double-deal. This is particularly important in the case where two such chains are the only chains remaining because the opponent does not need to double-deal on the final long chain.

Under the assumptions stated above, it is possible to calculate the minimum achievable reward which can be gained by adopting strategy 6.1. The reward from each long chain except the last is equal to the length of the chain minus the associated handout for double-dealing. The reward from the last chain is simply equal to its length. The total reward from the long chains can be summarised by the following expression:

$$r_l = \sum_{i=1,\ i \neq i^*}^{N_l} \Big(|C_i| - h_i \Big) + |C_{i^*}| \tag{6.1}$$

where N_l is the total number of long chains, $|C_i|$ is the length of the ith chain, h_i is the handout of the ith chain, and i^* is the last long chain, i.e. the longest chain in the set with priority given to loop chains. If n_l is the total number of coins contained within the set of long chains under inspection, then the number of coins handed to the opponent is $n_l - r_l$.

Recall that double-dealing would result in the gain of control of the long chains in figure 6.2. There are four long chains on this board: two non-circular 3-chains, one non-circular 4-chain, and one circular 4-chain. Assuming that the player and opponent would adopt strategies 6.1 and 6.2 respectively if the player had gained control of the long chains, then, from (6.1), the player would gain a reward of 6 coins from the long chains. Since the long chains contain a total of 14 coins, it follows that the opponent would gain a reward of 8 coins in addition to a reward of 2 coins gained from the double-deal. It



(A) Gaining control with an even number of short chains

(B) Gaining control with an odd number of short chains

FIGURE 6.3: Gaining control of the long chains

would therefore be unwise for the player to double-deal in this situation, as the minimum guaranteed yield is only 6 coins out of the total of 16 available.

6.2 Gaining control of the long chains

r_l is the expected yield from the long chains given that control of the long chains has been gained. In this section, the associated cost/reward of gaining control is explored. Consider the situations in figure 6.3. If one were trying to gain control of the long chains, should one perform the double-dealing move or not? Or, perhaps, is it not certain that either decision would lead to the acquisition of control? To answer these questions, certain assumptions must be made in a similar to manner to section 6.1. When opening 2-chains, it is assumed that both players follow strategy 4.2, i.e. that a hard-hearted handouts are always used. Recall that a hard-hearted handout divides a closed 2-chain into two open 1-chains. Such a strategy prevents the opponent from getting the opportunity to double-deal and switch the parity of the situation. By applying this strategy and thus preventing the opponent from double-dealing, the question of whether or not performing the double-dealing move would result in a loss or gain of control becomes a certainty and the answer is determined by the parity of the short chains. Note that the open chain is not counted as one of the short chains in this context.

See from figure 6.3a that, under the above assumptions, double-dealing will result in a gain in control. A double-deal would force the opponent to open the first short chain. The player would then take the coins in the opened chain and open the second short chain with a hard-hearted handout. The opponent is now left with no choice but to open the first long chain and thus the player has gained control over the long chains. This can be seen graphically in figure 6.4.

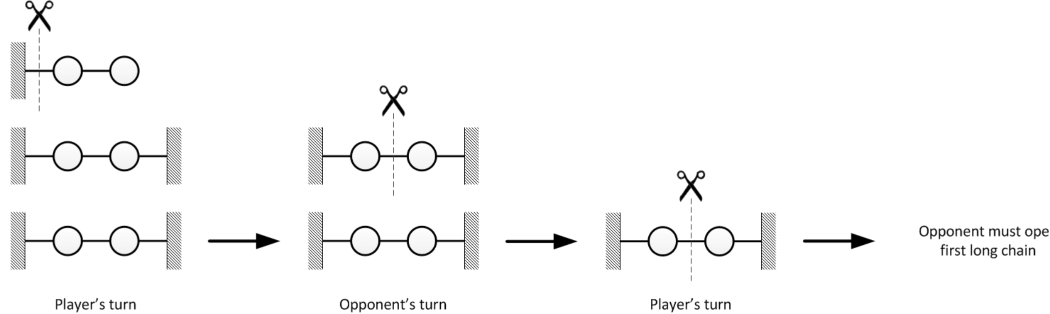


FIGURE 6.4: Using a double-deal to gain control when the number of short chains is even

Figure 6.3b shows a similar scenario except with an extra short chain. See that in this case, double-dealing will result in a loss in control. Instead, if the player were to take all the coins in the open chain and then open short chains following strategy 4.2, the opponent would be forced to open the first long chain and thus control would be gained. In general, given a double-dealing opportunity, double-dealing can be used to gain control if the number of short chains is even; otherwise, control can be gained by taking all the coins in the open chain and opening the first short chain. This will hereby be referred to as the *short chain rule*.

Rule (Short chain rule)**.** *Given a double-dealing opportunity, if the number of short chains is even then control of the long chains can be gained by double-dealing and employing strategy 4.2; if the number of short chains is odd, then control of the long chains can be gained by taking all the coins in the open chain and employing strategy 4.2.*

6.3 Reward from the short chains

The process of gaining control of the long chains has an associated reward, and the value of this reward depends on the number short chains and their lengths.

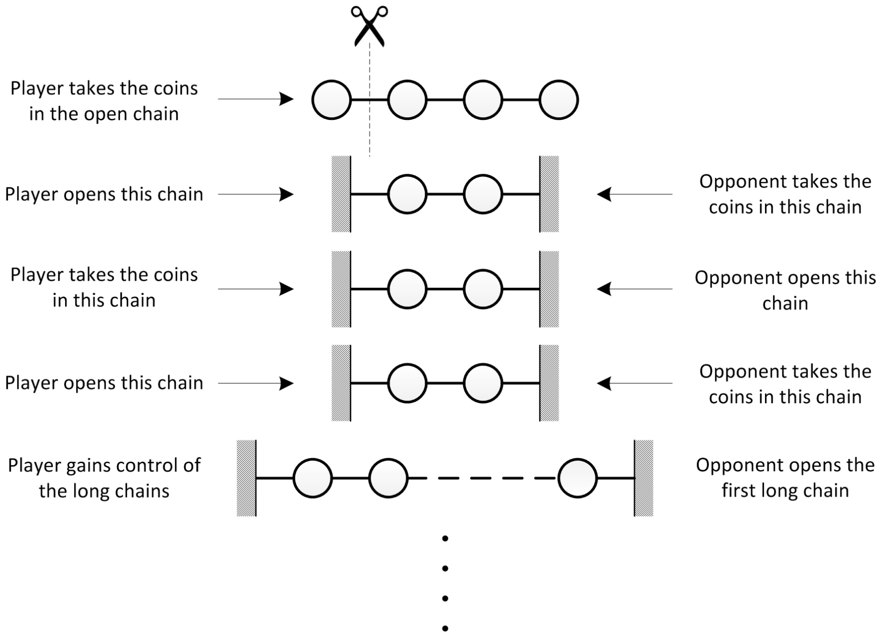


FIGURE 6.5: Distribution of coins from short chains when N_s is odd

6.3.1 Reward from 2-chains

Let N_s, N_1, N_2 be the number of short chains, 1-chains, and 2-chains respectively. It follows that

$$N_s = N_1 + N_2 \tag{6.2}$$

Assume that $N_s = N_2$, i.e. that there are no 1-chains. In the case where N_s is odd, control can be gained by taking the coins in the open chain and opening the first short chain. It is expected that the opponent would take all the coins in this newly opened chain and open the next short chain. The players will take turns collecting coins from the short chains until the first long chain is opened. Since the number of short chains is odd, the opponent will receive a greater reward from the short chains. This is illustrated in figure 6.5. The reward associated with gaining control in a situation where N_s is odd can be summarised by the following expression:

$$r_{s,\,odd} = 2\left\lfloor \frac{N_s}{2} \right\rfloor + |C_o| \tag{6.3}$$

where $|C_o|$ is the number of coins in the open chain and $\lfloor\cdot\rfloor$ is the flooring operator. In the case where N_s is even, control can be gained by double-dealing and the associated

reward can be summarised by the following expression:

$$r_{s,\,even} = 2\left\lceil \frac{N_s}{2} \right\rceil \tag{6.4}$$

where $\lceil\cdot\rceil$ is the ceiling operator. Although unnecessary, the ceiling operation was included for consistency. Note that in the cases above, it is assumed that all short chains are 2-chains.

6.3.2 Allowing for 1-chains

We now examine the case where the set of short chains is composed of both 1-chains and 2-chains. Figure 6.6 shows an example of such a case. The short chain rule still applies in this context and thus, since N_s is even, control of the long chains can be gained by double-dealing. The presence of 1-chains has an effect on the expected reward from the short chains when gaining control and thus (6.3) and (6.4) must be modified to allow for this.

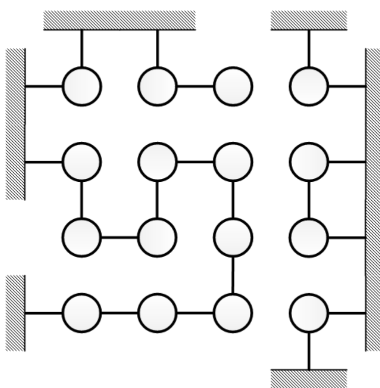


FIGURE 6.6: A 4x4 grid of coins with both 1-chains and 2-chains

Consider a set of short chains where all chains are of equal length. If a player were to open the first chain in the set, the opposing player would take the available coins and open the next chain. This would continue in a similar manner to figure 6.5. Note that if the number of chains is even, both players would gain an equal number of coins from the chains; however, if the number of chains is odd, then the player who opened the first chain would receive a smaller reward because the player would have opened more chains than the opponent, as was the case in figure 6.5. In general, the player who opens the first chain would receive a reward of $l\left\lfloor \frac{N_c}{2} \right\rfloor$ coins, where l is the length of each chain and N_c is the number of chains in the set. The opposing player would therefore receive a reward of $l\left\lceil \frac{N_c}{2} \right\rceil$ coins.

In order to calculate the number of coins that each player will take from the short chains, it requires prior knowledge of the chains that each player will open. Obviously, such information cannot be obtained with certainty; however, it is reasonable to assume that each player will follow strategy 4.1, i.e. 1-chains will be opened before 2-chains. Under

this assumption and the assumptions from section 6.2, the number of coins obtained by each player from the short chains can be evaluated if knowledge of both the player who opens the first 1-chain and the player who opens the first 2-chain is obtained. With such knowledge, the theory from the above paragraph can be used to calculate the number of coins that each player receives from both the 1-chains and the 2-chains.

Let P_1 be the player who opens the first 1-chain and let P_2 be the opposing player. The player who opens the first 2-chain is determined by the parity of N_1, the number of 1-chains. If N_1 is even, then P_1 will also open the first 2-chain and thus the number of coins which P_1 will receive is:

$$r_{P_1,\, even} = \left\lfloor \frac{N_1}{2} \right\rfloor + 2 \left\lfloor \frac{N_2}{2} \right\rfloor \tag{6.5}$$

The corresponding reward which P_2 will receive is:

$$r_{P_2,\, even} = \left\lceil \frac{N_1}{2} \right\rceil + 2 \left\lceil \frac{N_2}{2} \right\rceil \tag{6.6}$$

If, on the other hand, N_1 is odd, then P_2 will open the first 2-chain and thus the number of coins which P_1 will receive is:

$$r_{P_1,\, odd} = \left\lfloor \frac{N_1}{2} \right\rfloor + 2 \left\lceil \frac{N_2}{2} \right\rceil \tag{6.7}$$

The corresponding reward for P_2 in this case is:

$$r_{P_2,\, odd} = \left\lceil \frac{N_1}{2} \right\rceil + 2 \left\lfloor \frac{N_2}{2} \right\rfloor \tag{6.8}$$

Finally, the reward for gaining control of the long chains can be calculated. As the value of this reward depends on the parities of both N_s and N_1, there are four possible cases to consider. These four cases are summarised in table 6.1. Note that from (6.2), if the parities of both N_s and N_1 are known then the parity of N_2 is also known.

See that in figure 6.6, $N_s = 4$ and $N_1 = 3$. Using table 6.1, the expected reward for gaining control of the long chains evaluates to 2 coins.

N_s	N_1	control gained by	opens first 1-chain	opens first 2-chain	reward r_c
even	even	double-dealing	opponent	opponent	$\left\lceil \frac{N_1}{2} \right\rceil + 2\left\lceil \frac{N_2}{2} \right\rceil$
even	odd	double-dealing	opponent	player	$\left\lceil \frac{N_1}{2} \right\rceil + 2\left\lfloor \frac{N_2}{2} \right\rfloor$
odd	even	not double-dealing	player	player	$\left\lfloor \frac{N_1}{2} \right\rfloor + 2\left\lfloor \frac{N_2}{2} \right\rfloor + \lvert C_0 \rvert$
odd	odd	not double-dealing	player	opponent	$\left\lfloor \frac{N_1}{2} \right\rfloor + 2\left\lceil \frac{N_2}{2} \right\rceil + \lvert C_0 \rvert$

TABLE 6.1: Reward for gaining control of the long chains

6.4 Decision time

The theory covered in the previous sections can now be used to make an informed decision on whether or not to perform a double-dealing move. Define r_e to be the minimum expected reward from gaining and retaining control over the long chains. Thus, r_e can be calculated with the following expression:

$$r_e = r_c + r_l \tag{6.9}$$

where r_l is calculated using (6.1) and r_c is calculated using table 6.1. Let n be the total number of coins contained within the set of chains under examination, including the open chain. A reasonable policy for deciding whether or not to double-deal would be to base the decision on the ratio of r_e to n. If $r_e > n/2$ then gaining control would yield a reward greater than the reward which the opponent would receive and thus the player should choose the option which would result in the gain of control. If $r_e < n/2$ then gaining control would yield a reward less than what the opponent would receive and thus the player should choose the option which results in a loss of control. It should be noted, however, that the minimum expected reward for handing control over to the opponent is not simply equal to $n - r_e$, although often this is the case. When control has been handed over to the opponent, they may decide not to double-deal and thus hand control back to the player, if it seems more profitable to do so. Thus, for the case where $r_e = n/2$ the player should choose the option which results in the gain of control so as not to give any power to the opponent. This strategy can be summarised as follows:

Strategy 6.3 (Double-dealing)**.** *When given a double-dealing opportunity, if performing the double-dealing move results in a gain of control, then the move should be performed if r_e is greater than or equal to $n/2$. If, on the other hand, performing the double-dealing*

move results in a loss of control, then the move should be performed if r_e is less than $n/2$. If neither of the above conditions are met, then the double-dealing move should not be performed.

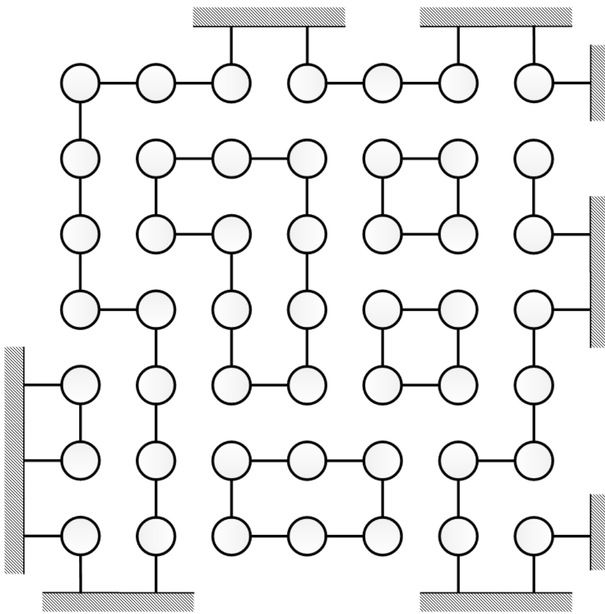


FIGURE 6.7: Board in figure 6.1 displayed in Strings-and-Coins format

Strategy 6.3 can be used to decide whether or not to perform the double-dealing move in figure 6.1. The board is shown in Strings-and-Coins format in figure 6.7. See that the board is divided into a series of chains and that there is an open chain with which a double-dealing move can be performed. Often it is easier to analyse the chains on the board if they are compiled into a chain list as shown in figure 6.8. The chains at the top of the list are the chains which are expected to be opened first and the chains at the bottom are those expected to be opened last. See that $N_s = 4$ and that $N_1 = 3$. Thus, by table 6.1, control of the long chains can be gained by double-dealing and the associated reward is $r_c = 2$ coins. From (6.1), the expected reward from the long chains evaluates to $r_l = 22$ coins, where C_{i^*} is the non-circular 10-chain. The total expected reward is therefore $r_e = 24$ coins. As the total number of coins contained within the chains is $n = 49$ coins, strategy 6.3 states that the double-dealing move should *not* be performed in this situation.

FIGURE 6.8: List of chains contained within figure 6.7

6.4.1 Cases when double-dealing is not performed

Note that this decision process assumes that there is only one open chain on the board. A double-dealing move should never be performed when there is more than one open chain as performing such a move would not only offer the opponent the corresponding handout from double-dealing on the chain but also all of the coins contained within the other chain(s). This would be a very foolish move and, although somewhat obvious to a human player, a computer could easily perform this action in error.

In addition to this case, the program will not perform a double-dealing move if any of the following conditions are met:

- There are no long chains
- The board is in a pre-gridlock state
- There are chains which are not independent of each other

Note that, if the first condition is met, then there are no long chains and thus a double-dealing move cannot be used to gain control of the long chains. If the second condition is met, it means that the program can end its turn without offering any coins to its opponent. If this is the case, the program will greedily take all the available coins and then make a move which does not offer any coins to the opponent. The third case refers to the existence of a special type of chain called a *complex chain*. These chains will be discussed in detail in the next chapter.

6.4.2 The double-dealing decision in functional programming

The function which was written to make the double-dealing decision, *doubleDeal*, can be found in section D.5.1. The function takes in a list of chains as well as a list of all nodes on the board and then makes the decision by employing strategy 6.3.

The presence of complex chains or moves which can end a turn without offering any coins to the opponent can be detected simply by checking if there are any multinodes on the board. If this is the case, *doubleDeal* will return *False*.

6.5 Summary

In this chapter, theory and framework for deciding whether or not to perform a double-dealing move when given the opportunity were presented. By making certain assumptions about the action policies of the opponent, formulas for the minimum expected reward for gaining control over the long chains were derived and a formula for the minimum expected reward from maintaining control over the long chains was presented. These reward values were then used along with the total number of coins contained within the chains in order to make an informed decision of whether or not to perform a double-dealing move.

The underlying assumption throughout this chapter was that all the chains on the board were *independent*, i.e the chains were either simple chains or loop chains. In the next chapter, a new type of chain will be introduced. These new chains do not have this independence property and thus significantly increase the complexity of the game.

Chapter 7

Complex Chains

So far in this document, only *independent* chains have been studied. Independent chains are chains which either form a closed loop or are bounded at both ends by wall nodes. Opening such chains and removing their contained nodes will have no effect on the other chains on the board and this greatly simplifies analysis and prediction. In this chapter, chains without this independence property as well as the difficulties associated with them will be investigated.

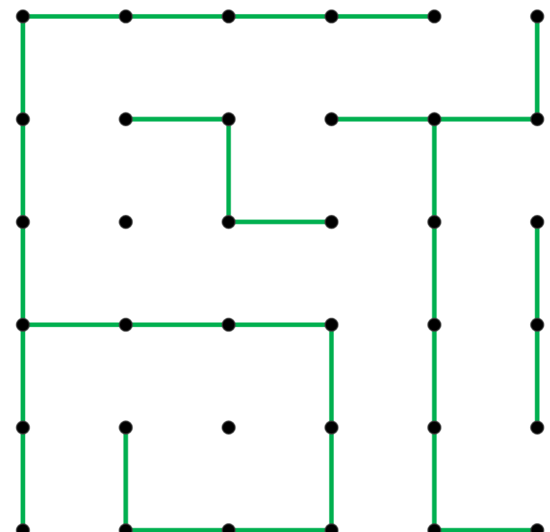


FIGURE 7.1: A game of Dots-and-Boxes with several complex chains

7.1 Complex chains and complex structures

Consider the structure in figure 7.2. This structure contains three sets of nodes which adhere to definition 4.1, i.e. the definition of a chain. However, these chains do not adhere to definition 4.4 because they are each bounded by a multinode and thus cannot be classified as simple chains.

Recall that, as part of this project, chain structures were categorised into three classes: simple chains, loop chains, and complex chains. The chains in figure 7.2 are examples of complex chains. The strict definition of a complex chain is stated below.

Definition 7.1 (Complex chain)**.** A complex chain is a non-circular chain bounded by at least one multinode.

A structure composed of several interconnected complex chains, such as the structure in figure 7.2, will hereby be referred to as a *complex structure*. A complex structure is defined as follows:

Definition 7.2 (Complex structure)**.** A complex structure is a single structure composed of complex chains and the nodes which bound them.

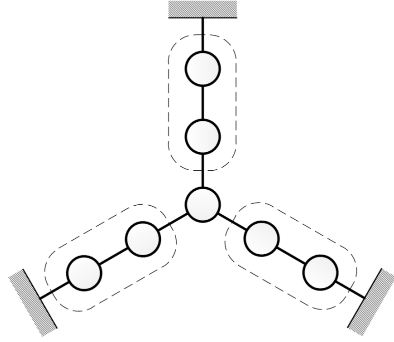


FIGURE 7.2: A complex structure

7.2 Dependence

What would happen if one of the complex chains in figure 7.2 were to be opened and its coins were to be removed? Figure 7.3 shows an example of such an action. See that, once the complex chain is opened and removed, the two remaining complex chains and the node connecting them merge together to form a single long chain. This is an example of how opening a complex chain in such a structure can have a direct and significant effect on the other complex chains in the structure. These complex chains are thus considered to be *dependent* on each other.

Not all neighboring chains in complex structures are dependant, however. A necessary requirement for one complex chain to be dependent on another is that opening the other chain and removing all of its coins must result in the shared multinode being reduced

FIGURE 7.3: Dependence between complex chains

to a binode or singleton node. See in figure 7.3 that when the top chain is opened and its coins are removed, the multinode connecting the three chains is reduced to a binode and thus forms a link which merges the other two chains into a single long chain. See that, due to the nature of this complex structure, opening any of these chains would result in the other two merging together. Thus each complex chain is considered to be dependent on the other. A strict definition of dependence is stated below.

Definition 7.3 (Dependence)**.** Chain A is dependent on chain B if opening chain B and removing all of its coins alters the composition of chain A.

This dependence property makes analysis of complex chains and structures significantly more difficult than that of simple chains and loops.

7.2.1 Symmetry of dependence

Consider the complex structure shown in figure 7.4. See that the complex chain labelled b is bounded at both ends by the same multinode. Therefore, if this chain were to be opened and its associated coins removed, the multinode would be reduced to a binode and chains a and c would be merged together. Thus, chains a and c are dependent on chain b. However, if either chain a or c were to be cut and their associated coins removed, the multinode would not be reduced to a binode.

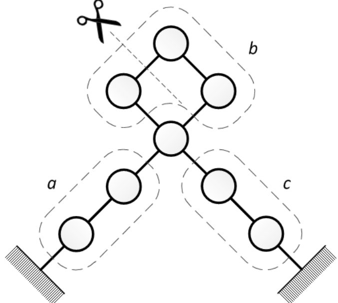


FIGURE 7.4: An example of how dependence between chains can be asymmetric

This is an example of how dependence between chains is not always symmetric. In this example, chains a and c are dependent on chain b but chain b is neither dependent on chain a nor chain c.

7.3 Dippers

An interesting structure which arises as a result of two dependent complex chains is the *dipper*. A dipper, which gets its name from the Big Dipper asterism, is a complex structure which is part circular and part non-circular [3]. Dipper structures contain a trinode (multinode of valency 3) with one edge forming part of a non-circular chain bounded by either a wall or another multinode and the other two edges forming a closed loop.

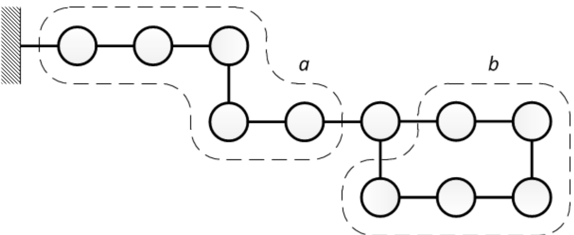


FIGURE 7.5: A dipper structure and the complex chains contained within

Figure 7.5 shows a dipper structure. See that the structure contains two complex chains connected by a trinode. The first chain, labelled with an a, is bounded by a wall and a trinode and thus forms the non-circular part of the structure. The second chain, labelled with a b, begins and ends at the same trinode and thus forms the circular part of the structure. This is not a circular chain, however, as it does not adhere to definition 5.1.

Upon inspection, one should conclude that these two chains are dependent on each other. Figure 7.6 shows the consequences of opening the chain marked a and removing the available coins. See that the chain marked b is transformed into a closed loop chain and has gained an extra coin as a direct consequence of this action.

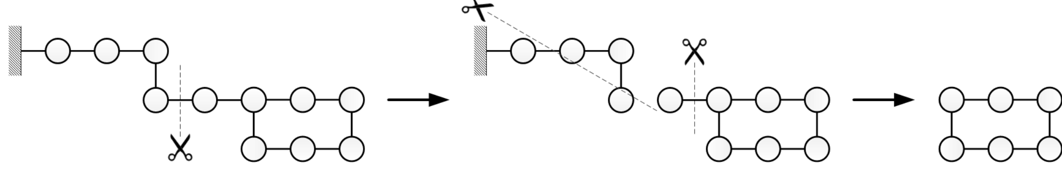


FIGURE 7.6: Opening the non-circular region of a dipper

Figure 7.7 shows the consequences of opening the chain marked b. See that, since the circular region contains two of the three edges of the trinode, opening this region and taking the available coins will reduce the trinode to a singleton and thus allow the rest of the coins in the dipper structure to be taken. There are two ways in which such a region can be opened. The first way involves opening one of the two edges connected to

(A)

(B)

Figure 7.7: Opening a dipper

the trinode, as can be seen in 7.7a. See that this action causes the dipper structure to become a single, large open chain. The second way involves opening one of the edges not connected to the trinode, as can be seen in 7.7b. Such an action splits the chain into two open chains which, when removed, will reduce the multinode to a singleton and thus allow the rest of the coins in the dipper to be taken. Although each action has a different effect on the dipper structure, either way would result in offering both a double-dealing opportunity and all the coins contained within the dipper to the opponent and thus the actions can be considered equivalent with regards to their overall effect on the game.

The process of opening the circular region of a dipper and thus offering all the coins contained within the dipper to the opponent is known as *opening the dipper*. It is a risky move and only in rare cases could performing such a move be considered wise. A complex chain which, when opened, will cause a dipper to open will hereby be referred to as a *dipper loop*.

7.3.1 Detecting dipper loops

Figure 7.8 shows a dipper structure where the non-circular region contains a significantly larger amount of coins than the circular region. The chain-detection procedure defined in algorithm 5.1 would detect a closed 12-chain and a closed 3-chain from the two regions

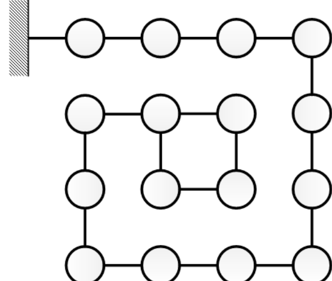


FIGURE 7.8: A dipper structure

of the dipper. A computer program following strategy 4.1 would choose to open the 3-chain first due to its smaller length. Such a move would in fact offer 16 coins to the opponent instead of the assumed three. Thus it is important that the program is able to recognise such an action to avoid performing it in error.

Dipper loops can easily be identified by the fact that they are bounded at both ends by the same trinode. Algorithm 7.1 below determines if a complex chain is a dipper loop.

Algorithm 7.1 (Detecting dipper loops)**.** *Given a complex chain bounded by two end-nodes, n_i and n_j, if n_i and n_j are both trinodes and $i = j$, i.e. the indices which uniquely identify each node are the same, then the chain should be classified as a dipper loop; otherwise it should not.*

The function which was written Haskell to determine if a chain is a dipper loop using algorithm 7.1 is called *isDipperLoop*. The source code for this function can be found in section D.4.2.1.

7.3.2 Effective length

Since opening a dipper loop offers more coins to the opponent than was initially contained within the chain, it is useful to define a quantity which measures exactly the number of coins which is offered to an opponent upon opening a certain chain. This quantity is defined to be the *effective length* of the chain.

Definition 7.4 (Effective length)**.** The effective length of a chain is the number of coins which are offered to an opponent once the chain has been opened.

The effective length of a dipper chain can be calculated using the following expression:

$$l_e = |C_1| + |C_2| + 1 \tag{7.1}$$

where $|C_1|$ is the number of coins in the dipper loop and $|C_2|$ is the number of coins in the non-circular region of the dipper.

Using the concept of effective length, strategy 4.1 can be adjusted to allow for dipper loops as follows:

Strategy 7.1 (Opening chains and dippers)**.** *When forced to open a chain, the chain with the smallest effective length should be opened so that the smallest possible number of coins are offered to the opponent.*

7.3.2.1 Effective length in functional programming

The function which was written to determine the effective length of a chain using (7.1) is called *effLen*. The source code for this function and an explantion of how it works can be found in section D.4.2.1.

7.4 Detecting complex chains

7.4.1 Complex chain data structures

In order to facilitate the detection of dependence between complex chains, the bounding multinodes of a complex chain should also be stored in the chain data structure. Thus the notation used to represent a chain data structure should be modified to the following: $\langle S \rangle_x^Y$ where S is the set of coins contained within the chain, x is the label, and Y is the set of multinodes which bound the chain. If the chain is not complex, then Y should be an empty set.

7.4.1.1 Complex chains in functional programming

The following code shows how complex chain data structures can be defined using functional programming:

type *Chain* = ([*NodeId*], [*NodeTuple*], *Label*)

data *Label* = *Closed* | *Open* | *Loop*

This was how chain data structures were defined in the source code of the program. See that the chain data structure contains a set of *NodeId*s which represent the set of coins contained within the chain, a set of *NodeTuple*s which represent the set of multinodes which bound the chain, and a label which indicates whether the chain is open, closed or a loop chain. Note that a *NodeTuple* object is a data structure containing a node and its associated key value. It was found to be more convenient to use *NodeTuple* objects to store information about the bounding multinodes of a complex chain. Again, these structures are defined in a simple, concise, and highly utilisable way which can only be achieved with the functional programming approach.

See section D.4 for more information about the implementation of chains in functional programming.

7.4.2 Complex chain detection algorithm

As complex chains can be easily identified by the multinodes which bound them, algorithm 5.1 need only be modified slightly in order to detect them. Algorithm 7.2 below is an chain-detection algorithm able to recognise such chains.

Algorithm 7.2 (Chain detection with loops)**.** *Given a binode b with neighbours n_1, n_2, the chain which contains b is the output of $f(n_2, f(n_1, \langle S_0\rangle_c^{Y_0}))$ where $S_0 = \{b\}$, $Y_0 = \{\}$, and the function $f(\cdot)$ is defined as follows:*

$$f(n, \langle S\rangle_x^Y) = \begin{cases} \langle S\rangle_l^Y, & n \in S \\ f(n', \langle S \cup \{n\}\rangle_x^Y), & isBinode(n) \quad and \quad n \notin S \\ \langle S \cup \{n\}\rangle_o^Y, & isSingleton(n) \quad and \quad n \notin S \\ \langle S\rangle_x^{Y \cup \{n\}}, & isMultinode(n) \quad and \quad n \notin S \\ \langle S\rangle_x^Y, & otherwise \end{cases}$$

where n' is the next neighbour of a binode n, $\langle\cdot\rangle_c$ denotes a closed chain, $\langle\cdot\rangle_o$ denotes an open chain, and $\langle\cdot\rangle_l$ denotes a loop chain.

See that algorithm 7.2 behaves almost exactly like algorithm 5.1 except that it facilitates the modified chain data structure from section 7.4.1. A new case has been added: the case where n is a multinode. In this case, the algorithm recognises that the end of a chain has been reached and thus terminates. It also adds the multinode n to the set Y. As the set Y is initially an empty set, the output chain from algorithm 7.2 can easily be recognised as a complex chain if the number of elements in Y is nonzero. If the chain in question is not complex, then the set Y will remain empty throughout the execution of the algorithm.

7.4.2.1 Complex chain detection in functional programming

The function which was written to detect a complex chain using algorithm 7.2 is called *toChain*. The source code for this function can be found in section D.4.2.2.

7.5 Double-dealing and complex chains

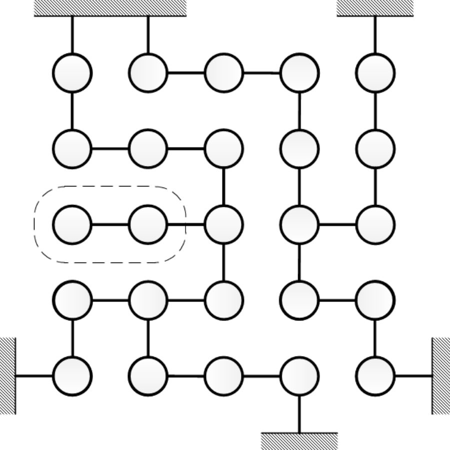


FIGURE 7.9: A series of complex chains and a double-dealing opportunity

Consider the grid of coins in figure 7.9. The board is in a post-gridlock state and is divided into a series of chains, one of which provides a double-dealing opportunity. Can the strategies discussed in chapter 6 be used to make an informed decision on whether or not to perform the double-dealing move? Upon inspection one should conclude that they cannot. The reason for this is that the strategies in chapter 6 work on the assumption that the board is divided into a series of *independent* chains. This independence property makes it possible to accurately predict an opponent's actions and thus predict

the distribution of chains on the board several turns in advance. The minimum expected reward from making a double-dealing decision is thus reduced to a few simple formulas. In order to make an informed decision of whether or not to double-deal in this situation, it requires the use of other decision-making techniques such as the minimax algorithm or nimber analysis. The use of such techniques was beyond the scope of this project and thus it was decided that the program would not perform a double-dealing move in such a situation. Instead, the program would take the coins on offer and open the next chain following strategy 7.1.

The function which was written to decide whether or not to double-deal, *doubleDeal*, will return *False* if there are any multinodes contained within the board. The presence of multinodes indicates that the board is either in a pre-gridlock state or in a state which contains complex chains and structures. Double-dealing moves will not be performed under such conditions. See section D.5.1 for more information on the function *doubleDeal*.

7.5.1 One-step-ahead analysis

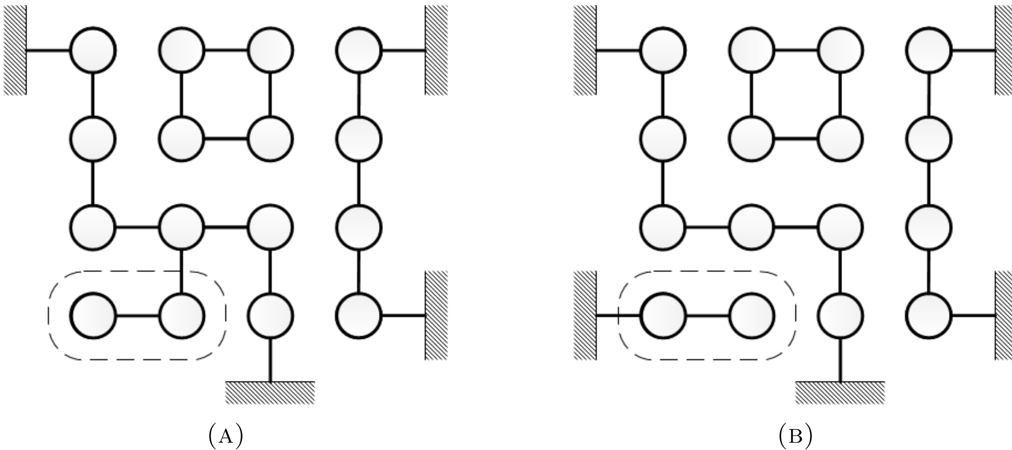


FIGURE 7.10: Two states which are equivalent from a double-dealing decision point of view

The presence of complex chains on the board does not always cause the formulas in chapter 6 to be inaccurate. Consider the board in figure 7.10a. See that there is a complex structure composed of three complex chains and that one of these chains is an open 2-chain. If either a double-dealing move is performed on the open 2-chain or the two coins in the chain are removed, the multinode which borders all three chains will be reduced to a binode and the two remaining chains will merge together. Notice

that once either action is carried out, the complex structure will be reduced to a single simple chain and all the complex chains on the board will have been removed. As the board becomes divided into a set of independent chains after either action is carried out, the theory from chapter 6 can be used to make an informed decision about the double-dealing move. In fact, from a double-dealing decision point of view, the board can be considered equivalent to the board in figure 7.10b, where the board is divided into a series of independent chains before the double-dealing decision has been made.

The program which was written to play Dots-and-Boxes is able to make an informed decision on whether or not to double-deal in the case of figure 7.10a simply by *pretending* that it is the case of figure 7.10b and calling the function *doubleDeal* to decide if the move should be made or not. The function *doubleDeal* takes in a list of chain data structures and a *Database* object containing information about the board and returns a Boolean value indicating whether or not a double-dealing move should be performed. Thus the program can pretend that the state in figure 7.10a is the state in figure 7.10b by appropriately modifying the list of chains and the *Database* object.

```
doubleDealComplex :: [Chain] → EdgeId → Database → Bool
doubleDealComplex cs e (nm, em)
    | (1 ≠) $ length opens = False
    | f c = let n = fst $ head $ getBoundingNodes c
                cs' = filter (not . boundedBy n) cs
                a = (fromChain c, [], Open)
                nm' = M.map (λ n → removeEdge n e) nm
                b = toChain (n, getElem n nm') (nm', em)
            in doubleDeal (a : b : cs') (nm', em)
    | otherwise = False
  where
    opens = filter (λ x → and [isOpenChain x, isComplex x]) cs
    c = head opens
    f (_, ys, _) = any (isTriNode . snd) ys
```

The above code shows the function *doubleDealComplex*, which can be found in appendix D.5.1. This function was written to indicate whether or not to perform a double-dealing move on an open complex 2-chain. In the above code, the variable c represents the open complex chain. If c is bounded by a trinode, then removing c would reduce this

trinode to a binode and thus any chains dependent on c would be merged together. This is exactly the case of the open 2-chain in figure 7.10a. If such conditions are met, *doubleDealComplex* will modify the list of chains and the *Database* and then input these objects as arguments to the double-dealing decision function, *doubleDeal*. The modifications are performed with the following steps:

- Get the trinode n which bounds c.
- Remove all chains which are bounded by n from the list of chains, cs.
- Disguise c as an open simple chain and represent it by the variable a.
- Reduce n to a binode by removing the edge which connects c to n and by updating the database.
- Use the chain detection function, *toChain*, on the binode n to get the new chain b to which it belongs.
- Add the chains a and b to the list of chains cs and input this list as well as the updated database to the function *doubleDeal*.

Note that if there are still complex chains remaining after the modifications were made, then *doubleDeal* will notice these and return *False*.

7.6 Summary

In this chapter, complex chains and their properties were examined in detail and an algorithm to detect such chains was presented. The concept of dependence between two complex chains and the effect this has on the analysis of Dots-and-Boxes was discussed. Furthermore, the implications that complex chains have on the double-dealing decision was discussed and a one-step-ahead analysis strategy for double-dealing with complex chains was presented.

In the next chapter, the action policy which the program used in order to follow the strategies discussed in this document is presented.

Chapter 8

Action policy

In order for a computer to play a game, it requires an action policy which allows it to decide which move to make next. The action policy implemented for this project needed to be consistent with the strategies and theory discussed in the previous chapters. The design and implementation of this action policy is discussed in this chapter.

8.1 Edge priorities

In order to choose an action, the program was implemented so that it analyses the current state of the board and assigns a priority value to each available edge. The program then chooses the edge with the highest priority value as the next edge to cut. If more than one edge has the highest priority value, then the program chooses one of these edges at random. The function which was written to perform these actions is called *makeMove* and the Haskell code for this function can be found in section D.5.3.

8.2 Assigning edge priorities

The priority of an edge is determined by the effect that cutting the edge would have on the state of the game. An edge which, when removed, will result in a gain in coins for the player would have a high priority value; on the other hand an edge which, when removed, will result in offering a number of coins to the opponent would typically have a low priority value. For example, a computer playing Dots-and-Boxes at a low level

might assign a priority value of +1 to an edge which would result in a gain of a coin, a priority value of −1 to an edge which would result in offering a number of coins to the opponent, and a value of 0 to an edge which would not result in any player gaining coins. Higher level programs would require more sophisticated policies for assigning a priority value to an edge.

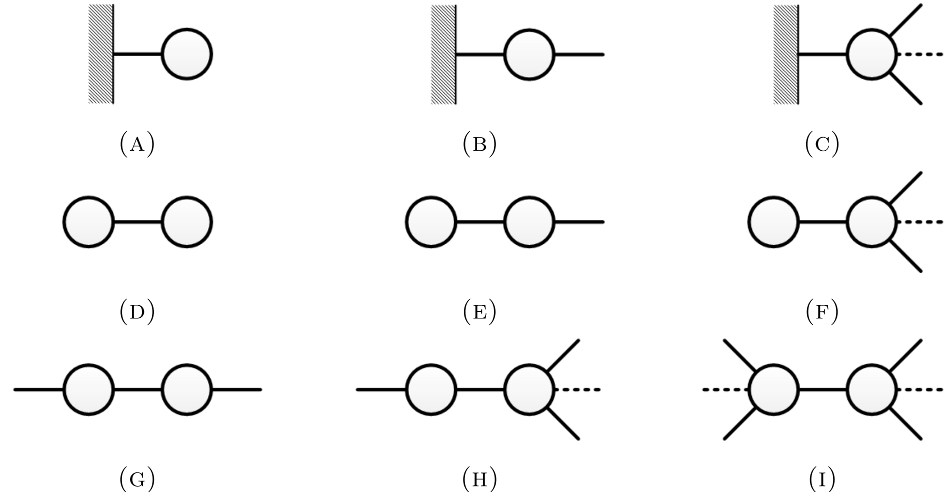


FIGURE 8.1: All the possible node pairs to which an edge can connect

The effect that removing an edge would have on the game can be determined by examining the two nodes to which the edge connects. The function *getPriority* was written to assign a priority value to an edge by pattern matching on its associated nodes. This function can be found in section D.5.2.

Figure 8.1 shows all the possible combinations of node pairs to which an edge can connect in a game of Dots-and-Boxes. See that edge types (A), (D), and (F) should have the highest priority values because removing them would award the player with with coins and an additional turn without worsening the player's strategic position. In other words, removing these edges will award the player *free coins*. *getPriority* assigns a priority value of 3 to such edges. Recall that the chain detection algorithms described in this document do not detect open 1-chains and dipoles. These chains contain only free coins and are detected by pattern matching in *getPriority*.

The same cannot be said for the edge in figure 8.1e, however, because removing it may result in the loss of a double-dealing opportunity. Nevertheless, removing the edge would result in the gain of a coin and thus the move should be given a high priority, albeit not

as high as a free coin or a double-dealing move used to good effect. *getPriority* assigns a priority value of 1 to these edges.

See that removing the edges in figures 8.1c and 8.1i would neither result in a gain of coins for the player nor an offering of coins to the opponent. These edges can thus be considered to have medium priority values. *getPriority* assigns a priority value of 0 to these edges.

The edges types (B), (G), and (H) are connected to binodes but not to singleton nodes and thus removing them would result in offering a number of coins to the opponent. As all these edge types are connected to binodes, it follows that they form a link in a chain structure. The exact number of coins offered to the opponent when the edge removed is therefore equal to the effective length of the edge's associated chain. In order to implement an action policy which follows strategy 7.4, the priority value of such an edge should be proportional to the effective length of its associated chain such that chains with a larger effective length would have a lower priority. However, if removing such an edge results in a double-dealing move, then this edge may have a high priority value. Further analysis is therefore needed. The further analysis of the edge types (B), (G), and (H) is performed in the functions *getPriorityB*, *getPriorityG*, and *getPriorityH* respectively. These functions can be found in section D.5.2.

8.2.1 Further analysis of possible double-dealing opportunities

In order to perform further analysis on edge types (B), (G), and (H), it requires that other nodes be examined. These nodes of interest are the next neighbouring nodes of each binode under analysis. Edge types (B) and (H) connect to a single binode and thus only one additional node needs to be examined. Edge type (G) connects two binodes together and thus two additional nodes are required.

8.2.1.1 Further analysis of edge type (B)

The function *getPriorityB* determines the priority value of the edge in figure 8.1b by examining the next neighbouring node of the binode. If this node is a singleton node, then the resulting structure would be the structure in figure 8.2. This structure is

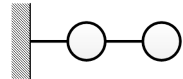


FIGURE 8.2: An open simple 2-chain

an open simple 2-chain and thus cutting the edge would result in performing a double-dealing move. If the double-dealing decision function, *doubleDeal*, indicates that the double-dealing move should be performed, then *getPriorityB* will return a value of 2. See that the priority value of a double-dealing move is less than that of a free coin but greater than that of the move in figure 8.1e. If it is deemed that the double-dealing move should not be performed, then *getPriorityB* will return a value of -4 (-2 times the number of coins offered to the opponent).

If the node of interest is not a singleton, then removing the edge would result in opening a chain and thus *getPriorityB* returns a value of -2 times the effective length of the associated chain. The reason why the effective length is multiplied by -2 is that priority values are stored as integers in the program.

8.2.1.2 Further analysis of edge type (G)

The priority value of the edge type (G) is calculated using the function *getPriorityG*. This function examines the neighbouring nodes of each of the binodes in order to determine the edge's probability. See that, if both these nodes are singleton nodes, then the resulting structure would be an open 4-chain with two loose ends shown in figure 8.3. Recall that removing this edge would be a special case of the double-dealing move where two dipoles are produced. In a similar manner to *getPriorityB*, a priority value of 2 is returned if it is decided that the double-dealing move should be performed; otherwise, a value of -8 is returned.

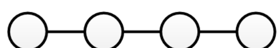


FIGURE 8.3: An open loop chain containing four coins

If both of the nodes under inspection are either wall nodes or multinodes, then the resulting structure is a closed 2-chain. Cutting the edge would therefore produce a hard-hearted handout. Recall that strategy 4.2 states that a hard-hearted handout should always be performed when opening a 2-chain. In order to follow this strategy, this edge should have a higher priority value than one which produces a half-hearted handout, i.e. a move which opens a 2-chain and creates a double-dealing opportunity. *getPriorityG* assigns a priority value of -3 for hard-hearted handouts. See that this priority value is greater than that a half-hearted handout (-4) yet smaller than that of an edge which would open a closed 1-chain (-2).

If none of the above cases are met then the structure under inspection must be a long chain and thus removing the edge would result in offering a number of coins to the opponent equal to the effective length of the chain. If this chain is non-circular, then *getPriorityG* returns a value of $-2l_e$, where l_e is the chain's effective length; if the chain is a loop chain, *getPriorityG* returns a value of $2l_e + 1$ in order to give priority to loop chains over non-circular chains of the same length. This is consistent with strategy 6.2.

8.2.1.3 Further analysis of edge type (H)

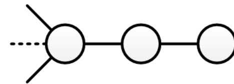


FIGURE 8.4: An open complex 2-chain

The edge type in figure 8.1h connects a binode to a multinode and thus forms a link in a complex chain. If the next neighbouring node of the binode is a singleton node, then the resulting structure would be an open complex 2-chain as can be seen in figure 8.4. Removing the edge would therefore result in a double-dealing move. Recall from chapter 7 that, since detailed analysis of complex structures is beyond the scope of this project, double-dealing moves typically won't be performed except perhaps if the board were to become divided into a series of independent chains as a result of removing the open complex chain. *getPriorityH* thus uses the function *doubleDealComplex* in order to check whether or not to perform the double-dealing move. If *doubleDealComplex* returns *True*, then *getPriorityH* will return a priority value of 2; otherwise it will return a priority value of -4.

If the next neighbouring node is not a singleton, *getPriorityH* will return a priority value of -2 times the effective length of the associated chain.

8.3 Effectiveness of action policy design

The priority value of an edge was calculated by pattern matching on the two nodes to which the edge connects. This approach enabled the strategies and theory from the previous chapters to be integrated into the action policy and as a result the program was able to play the game at an advanced level.

The pattern recognition approach was greatly facilitated by functional programming because functional programming languages have the ability to pattern match on data structures and values.

```
isEdgeTypeA :: Node → Node → Bool
isEdgeTypeA Wall (Singleton _) = True
isEdgeTypeA (Singleton _) Wall = True
isEdgeTypeA _ _                = False
```

The Haskell code above shows a function which detects an edge of type (A) by pattern matching on the two nodes which are joined by a certain edge. See that if one of the nodes is a *Wall* and the other is a *Singleton*, then the function will recognise the pattern and conclude that the edge connecting the nodes is of type (A). The use of pattern matching is a concise, effective, and elegant way to reach this conclusion.

Furthermore, in [3], Berlekamp shows how a nimber value can be assigned to a structure by recognising certain patterns contained within the structure. The pattern matching capabilities of function programming languages would therefore be of great use when designing a program to perform nimber analysis on a game of Dots-and-Boxes.

8.4 Summary

The process of assigning a priority value to an edge is summarised by tables 8.2, 8.3, and 8.4. See that this process allows a program to follow the strategies which were discussed throughout this document. A program following the action policy discussed in this chapter should be able to consistently defeat a naïve player who does not have an advanced-level understanding of the game.

8.5 Further work

The program created during this project is one which is able to play Dots-and-Boxes at an advanced level and is able to use techniques such as chain counting and double-dealing moves to good effect. However, this program is not yet able to play at the highest level and thus further work is required in order to increase its sophistication.

The next step for this program would be to enable it to perform advanced analysis on complex structures using techniques such as alpha-beta pruning and nimber analysis. This should allow the program to make better decisions in both pre-gridlock and post-gridlock situations. The data structures and functions implemented during this project are general and flexible and thus should facilitate further development of the program.

Edge type	Pattern	Comment	Priority
(A)		Free coin	3
(B)		More analysis required	Refer to table 8.2
(C)		No obvious consequence	0
(D)		Free coins	3
(E)		Gain a coin	1
(F)		Free coin	3
(G)		More analysis required	Refer to table 8.3
(H)		More analysis required	Refer to table 8.4
(I)		No obvious consequence	0

TABLE 8.1: Summary of function *getPriority*

Pattern	*doubleDeal*	Comment	Priority
	True	Perform double-dealing move	2
	False	Do not perform double-dealing move	-4
Otherwise	n/a	Offer a number of coins to the opponent	$-2l_e$

Table 8.2: Summary of function *getPriorityB*

Pattern	*doubleDeal*	*isLoop*	Comment	Priority
	True		Perform double-dealing move	2
	False		Do not perform double-dealing move	-8
			Hard-hearted handout	-3
			Hard-hearted handout	-3
			Hard-hearted handout	-3
Otherwise		*False*	Offer a number of coins to the opponent	$-2l_e$
Otherwise		*True*	Give priority to loops	$-2l_e + 1$

Table 8.3: Summary of function *getPriorityG*

Pattern	*doubleDealComplex*	Comment	Priority
	True	Perform double-dealing move	2
	False	Do not perform double-dealing move	-4
Otherwise	n/a	Offer a number of coins to the opponent	$-2l_e$

Table 8.4: Summary of function *getPriorityH*

Chapter 9

Conclusion

The aim of this project was to create a program capable of playing the combinatorial game Dots-and-Boxes using functional programming. The program was required to play the game at an advanced level such that it could use the famous double-dealing move to good effect. In order to achieve this, a thorough understanding of the game and its properties was needed. In this work, theory and framework for the Dots-and-Boxes game were established and high-level strategies for playing the game were presented.

The program which was implemented to play the Dots-and-Boxes game required an architecture of data structures which would store information in an efficient and utilisable way, functions capable of performing actions such as chain detection, lookahead analysis, and decision making, an action policy which would enable it to employ high-level gameplay strategies, as well as an interactive user interface with which the game could be played and tested. These were all realised through the use of functional programming.

The use of functional programming proved to be extremely beneficial in the design of this program. Its capability to support the use of functional data structures, pattern matching, and higher-order functions, as well as its toolkit of flexible and highly-utilisable functions made it extremely suitable for tackling the difficult task of achieving advanced-level play of the Dots-and-Boxes game. Furthermore, the use of functional programming allowed the program to be elegant and concise while achieving a sophisticated level of functionality. The functional data structures used to express nodes, edges, and chains

were simple, concise, yet extremely powerful; they allowed the program to perform complex analysis in an efficient and precise way and they are also flexible enough to allow for further development of the program.

A similar program could be realised through the use of imperative programming languages, however it would most likely lack the finesse and concision which can be achieved using the functional programming approach.

Appendix A

An Introduction to Functional Programming

A.1 Functional Programming

Functional programming is a programming paradigm which models computation as the evaluation of mathematical functions. The output of such a function depends only on its list of arguments and their associated values. Imperative programming languages, on the other hand, model computation as the sequential execution of statements which change a global state. In functional programming there is no dependence on any global state variable.

A.1.1 First-class functions

Functions in functional programming are required to be *first-class*, meaning that a function and a list of arguments can be treated as a *value* prior to evaluation. This useful property allows for the use of techniques such as higher-order functions, currying of functions, lazy evaluation etc.

A.1.2 Lazy Evaluation

The requirement of functions to be first-class allows functional programming languages make use of the evaluation strategy known as *lazy evaluation*. With lazy evaluation, the evaluation of a function is not performed until absolutely necessary. This avoids the unnecessary evaluation of expressions and thus can lead to more efficient use of computational resources.

Lazy evaluation allows infinite data structures to be defined and used. An infinite data structure which is often used in functional programming is the infinite list. For example, the Haskell code '*take* 5 $[1, 2..]$' will return $[1, 2, 3, 4, 5]$. In this example, $[1, 2..]$ is an infinite list of consecutive integers beginning at 1. This infinite list will not be completely evaluated, however, as only the first five elements are requested. As lazy evaluation performs calculations on a need-to-know basis, evaluation of the infinite list will cease after the fifth element is calculated.

A.1.3 Higher Order Functions

One of the great strengths of functional programming is its ability to use higher-order functions, i.e. functions which do at least one of the following:

- Take a function as one of its arguments
- Return a function as an output

An example of a higher order function from mathematics is the differentiation operator, $\frac{d}{dx}$. This function takes a function $f(x)$ as its input and returns a function $f'(x)$ as its output. The use of higher order functions allows for the implementation of flexible and highly utilizable functions such as the famous *map*, *fold*, and *filter* functions.

A.1.4 Currying

Functional programming languages evaluate functions using a technique called *currying*. This technique translates a function of several arguments into a sequence of unary functions.

Consider the function $f(x, y) = 2x + y$. A curried version of this function is the function $\alpha(x) = x \mapsto f(x, y)$. The function $\alpha(x)$ takes a single argument and returns a function $\beta(y)$ as its output. $f(1, 2)$ can thus be evaluated using currying as

$$f(1, 2) = \beta(2),$$

where

$$\beta(y) = \alpha(1) = f(x, y)|_{x=1} = 2 + y$$

All multivariable functions in functional programming can thus be thought of as higher order functions due to the use of currying.

A.2 Writing Functions in Haskell

All the source code for this project was written using the functional programming language Haskell. This section discusses many common techniques used when programming in Haskell in order to facilitate the understanding of the source code presented in appendix D. A detailed and enjoyable tutorial for Haskell can be found in [2].

A.2.1 Type declarations

The function $f(x, y)$ from section A.1.4 can be implemented in Haskell as follows:

$$f \;::\; Num\; a \Rightarrow a \rightarrow a \rightarrow a$$
$$f\; x\; y \;=\; 2 * x + y$$

The line $f \;::\; Num\; a \Rightarrow a \rightarrow a \rightarrow a$ declares the type of the function. In this example, f is a function which takes two arguments, each of type a where a is a member of the Num typeclass (meaning that a must be a number), and returns a value of the same type. The part $a \rightarrow a \rightarrow a$ can be rewritten as a $a \rightarrow (a \rightarrow a)$ and literally means "a function which maps a type a to a function which maps a type a to a type a". The reason for this is that all multivariable functions in Haskell are curried. For all intents and purposes, the last value in the type declaration can be thought of as the return value of the function and all other values as the arguments.

A.2.2 Pattern matching

A useful property of functions in Haskell is their ability to use a technique called *pattern matching*. Pattern matching allows certain patterns in the input arguments to determine the behaviour of the function.

$$g \;::\; Maybe\; a \rightarrow Bool$$
$$g\; (Just\; x) = True$$
$$g\; x \qquad = False$$

Consider the function g written in Haskell above. The input variable is of the *Maybe* type. The *Maybe* type has two possible values: *Nothing* and *Just* a, where a is some type (*Int*, *Bool* etc.). The function g uses pattern matching on this input variable and returns different values depending on the pattern which the variable matches. The first pattern is *Just* x and any input with the value *Just* a will match this type. The second pattern is the most general pattern that all inputs will match. However, if the input matches the second pattern it means that it did not match the first pattern and thus cannot have the value of *Just* a. Thus it must have the value *Nothing*.

A.2.3 Guards

Guards work in a similar way to pattern matching, however, instead of the patterns of the input arguments, the behaviour of the function is determined by whether or not the arguments satisfy certain predicates.

$$
\begin{aligned}
&h \;::\; \mathit{Num}\ a \rightarrow a \rightarrow a \rightarrow \mathit{Int}\\
&h\ x\ y\\
&\qquad \mid (x > y) \;= 1\\
&\qquad \mid (x < y) \;= 2\\
&\qquad \mid \mathit{otherwise} = 3
\end{aligned}
$$

The function h shown above uses guards to determine its behaviour. Note that if the arguments meet the second guard predicate, it means that they did not meet the first predicate.

A.2.4 Wildcards

Wildcards in are used to represent a variable whose value has no effect on the behaviour of the function. Wildcards in Haskell are written as '_'.

$$
\begin{aligned}
&g \;::\; \mathit{Maybe}\ a \rightarrow \mathit{Bool}\\
&g\ (\mathit{Just}\ _) = \mathit{True}\\
&g\ _ \qquad\quad\; = \mathit{False}
\end{aligned}
$$

The function g from section A.2.2 can be rewritten using wildcards as shown above. Since the value x is never used within the function, it can be replaced with a wildcard.

A.2.5 Lambdas

Lambdas are used to define functions which will only be used once and are typically used when defining a function to pass to an higher order function. The famous function *filter* is a higher order function which takes a predicate function and a list and returns a new list composed of all the elements of the input list which satisfy the predicate.

$$
\mathit{filter}\ (\lambda\, x \rightarrow \mathit{and}\ [x > 5,\ x < 10])\ [1, 2..20]
$$

The Haskell code above should return a list of all natural numbers which lie on the interval $(5, 10)$. The predicate function which is input to the *filter* function is defined using a lambda.

A.3 I/O with functional programming

The requirement that the return value of a function in functional programming be determined by the argument list alone makes the design of I/O functions difficult. However, as the code for the interactive user interface designed for this project is not included in this document, I/O in Haskell will not be discussed in detail. A detailed tutorial about I/O in Haskell can be found in [2].

Appendix B

Lists

B.1 The List data structure

One of the most important data structures in Haskell is the list. A list is a set of objects which can be accessed using an index, similar to an array in imperative languages. There is no restriction on what type of object may be stored in a list; however the list must be homogeneous, i.e. all objects stored within the list must be of the same type (integer, boolean, etc.). The notation $[a]$ is used when declaring an object to be a list of type a.

A list can be thought of as a recursive data structure with two possible values: an empty list (denoted in Haskell as $[\,]$), or a combination of a head element and a sub-list containing elements of the same type. The latter value is essentially a constructor which takes an element and a list and forms a new list, i.e. $l' = Cons(e, l)$ where e is the new element, and l is the input list. In Haskell this constructor is denoted as : and is used as an infix operator. Any non-empty list l can thus be expressed as $(x : xs)$, where x is the head element and xs is the rest of the list.

Lists in Haskell are typically written as $[e_1, e_2, e_3, ..., e_n]$ where e_i is the ith element in the list. A list containing the first three natural numbers in increasing order would therefore be written as $[0, 1, 2]$. However, this notation is just syntactic sugar for $(0 : (1 : (2 : [])))$, which shows the list as a recursive data structure ending with an empty list.

B.2 Common List functions

The following are common operations on lists which were used regularly in this project. Refer to Data.List from Hackage [10] for more information.

$head :: [a] \rightarrow a$
Returns the first element of a non-empty list.

$tail :: [a] \rightarrow [a]$
Returns all the elements after the head of a non-empty list.

$last :: [a] \rightarrow a$
Returns the last element of a non-empty list.

$init :: [a] \rightarrow [a]$
Returns all the elements of a non-empty list except the last element.

$(+\!\!+) :: [a] \rightarrow [a] \rightarrow [a]$
An infix operator. Concatenates two lists into a single list.

$(!!) :: [a] \rightarrow Int \rightarrow a$
An infix operator. Takes a list and an index and returns the element located at that index. Throws an error if the index is out of the range of the list.

$length :: [a] \rightarrow Int$
Returns the number of elements in a list.

$maximum :: Ord\, a \Rightarrow [a] \rightarrow a$
Returns the maximum value of a finite, non-empty list. The elements must be of an ordered type.

$sum :: Num\, a \Rightarrow [a] \rightarrow a$
Returns the sum of a finite list of numbers.

$product :: Num\, a \Rightarrow [a] \rightarrow a$
Returns the product of a finite list of numbers.

$elem :: Eq\, a \Rightarrow a \rightarrow [a] \rightarrow Bool$
Tests if an element is in a list. For the function to return False, the list must be finite.

$zip :: [a] \rightarrow [b] \rightarrow [(a, b)]$
This function takes two lists and combines them into a list of associated pairs. The length of the resulting list will be equal to that of the shortest of the input lists. The excess elements of the longer list will be discarded.

$and :: [Bool] \rightarrow Bool$
This function returns the conjuction of a Boolean list. It returns False if any element of the list is False. For it to return True, the list must be finite and all of its elements

must be True.

$or :: [Bool] \to Bool$

This function returns the disjunction of a Boolean list. It returns True if any element of the list is True. For it to return False, the list must be finite and all if its elements must be False.

$map :: (a \to b) \to [a] \to [b]$

This function takes in a unary function f and a list xs and returns the list resulting from applying f to each element of xs.

$foldl :: (b \to a \to b) \to b \to [a] \to b$

This function takes in a binary function, an initial accumulator value, and a list and returns the accumulator after each element

$filter :: (a \to Bool) \to [a] \to [a]$

This function takes a predicate function p and a list xs and returns a list of all the elements in xs which satisfy p.

$partition :: (a \to Bool) \to [a] \to ([a], [a])$

This function takes a predicate function p and a list xs and returns a pair of lists (ys, zs) where ys contains all the elements in xs which satisfy p and zs contains all the elements in xs which do not satisfy p.

$any :: (a \to Bool) \to [a] \to Bool$

This function takes a predicate function p and a list xs and returns True if any element in xs satisfies p. In order for it to return False, it requires that the list be finite and that no element in the list satisfies p.

$all :: (a \to Bool) \to [a] \to Bool$

This function takes a predicate function p and a list xs and returns True if all the elements in xs satisfy p and that xs is finite. In order for it to return True, it requires that at least one element in the list does not satisfy p.

Appendix C

Maps

C.1 The Map data structure

Map data structures were used extensively in the implementation of the program for this project. Map structures are essentially association lists, i.e. lists of key-value pairs where each key uniquely identifies its corresponding value. The use of these structures requires the inclusion of the *Data.Map* module.

The elements in a map are stored in a way which makes traversal of the map quick and efficient. This implementation is based on *size balanced* binary trees, as is described in [11], [12].

C.2 Common Map functions

The following are common operations on maps which were used regularly in this project. Refer to Data.Map from Hackage [13] for more information.

$size :: Map\ k\ a \rightarrow\ Int$
Returns the number of elements contained in the map.

$member :: Ord\, k \Rightarrow\ k \rightarrow\ Map\ k\ a \rightarrow\ Bool$
$O(log\, n)$. Indicates whether or not a key is a member of a map.

$lookup :: Ord\, k \Rightarrow\ k \rightarrow\ Map\ k\ a \rightarrow\ Maybe\ a$
$O(log\, n)$. Retrieves an element from a map given its corresponding key value. If the key

is a member of the map, the function returns ($Just \quad \langle value \rangle$); otherwise it will return *Nothing*.

$insert :: Ord\, k \Rightarrow \; k \rightarrow \; a \rightarrow \; Map\ k\ a \rightarrow \; Map\ k\ a$
$O(log\, n)$. Inserts a key and its corresponding value into the map. If the key is already a member of the map, then its associated value will be replaced with the new value.

$update :: Ord\, k \Rightarrow \; (a \rightarrow \; Maybe\ a) \rightarrow \; k \rightarrow \; Map\ k\ a \rightarrow \; Map\ k\ a$
$O(log\, n)$. Updates the corresponding value x of a key value k, if k is a member of the map. A unary function f is input to the function and if $f\ x$ returns *Nothing* then the key-value pair will be deleted; if $f\ x$ returns ($Just\ y$) then the value x will be replaced with y.

$map :: (a \rightarrow \; b) \rightarrow \; Map\ k\ a \rightarrow \; Map\ k\ b$
$O(n)$. Takes a function f and a map m and returns the resulting map after f has been applied to each element in m.

$elems :: Map\ k\ a \rightarrow \; [a]$
$O(n)$. Returns a list of all the elements contained within a map in the ascending order of their key values.

$keys :: Map\ k\ a \rightarrow \; [k]$
$O(n)$. Returns a list of all the key values contained within a map in ascending order.

$toList :: Map\ k\ a \rightarrow \; [(k, a)]$
$O(n)$. Takes a map m and returns a list of all key-value pairs contained within m.

$fromList :: Ord\, k \Rightarrow \; [(k, a)] \rightarrow \; Map\ k\ a$
$O(n \cdot log\, n)$. Takes a list of key-value pairs and converts it into a map object. If the list has duplicate keys, then only the last value for the key is inserted.

$filter :: Ord\, k \Rightarrow \; (a \rightarrow \; Bool) \rightarrow \; Map\ k\ a \rightarrow \; Map\ k\ a$
$O(n)$. Removes all elements from a map which don't satisfy the input predicate function.

Appendix D

Code

D.1 Imported modules

While most of the standard functions and data types which were used were defined within the default *Prelude* module, some were defined in other modules which needed to be imported.

```
import qualified Data.Map as M
import qualified Data.List as L
import qualified Data.Maybe as Mb
import System.Random
```

Functions and data types from any of the qualified imports stated above will be preceded by the associated identification string. For example, a function imported from *Data.Map* will be preceded by '*M.*'.

D.2 General functions

- *removeElem*

  This function removes all occurrences of the element y from the list xs.

```
removeElem :: (Eq a) => [a] -> a -> [a]
removeElem xs y = foldr (\x acc -> if x == y then acc else x : acc) [] xs
```

- *getElem*

  This function retrieves the element with the key value k from a *Map* structure m. An error will be caused if there is no such element.

$getElem\ ::\ (Ord\ k) \Rightarrow k \rightarrow M.Map\ k\ a \rightarrow a$
$getElem\ k\ m\ =\ Mb.fromJust\ \$\ M.lookup\ k\ m$

- *randIndex*
  This function takes in a random number seed value *gen* and an integer i and returns an integer chosen randomly from the first i natural numbers. The function will not behave correctly if i is not a natural number.

  $randIndex\ ::\ StdGen \rightarrow Int \rightarrow Int$
  $randIndex\ gen\ i\ =\ fst\ \$\ randomR\ (0, i - 1)\ gen$

- *ceilDiv*
  This function takes in two integers n, d and returns $\lceil \frac{n}{d} \rceil$, where $\lceil \cdot \rceil$ is the ceiling operator.

  $ceilDiv\ ::\ Int \rightarrow Int \rightarrow Int$
  $ceilDiv\ n\ d\ =\ (div\ n\ d) + \mathbf{if}\ (mod\ n\ d) > 0\ \mathbf{then}\ 1\ \mathbf{else}\ 0$

D.3 Strings-and-Coins architecture

As was discussed in chapter 3, any game of Dots-and-Boxes has an associated Strings-and-Coins representation. This representation allows the board to be expressed as a mathematical graph.

D.3.1 Data types for nodes and edges

D.3.1.1 Edge and node identifiers

Each edge and node in the graph must have a unique identifier.

type $EdgeId = Int$
type $NodeId = Int$

D.3.1.2 Edge structures

Each edge contains the identifiers of the two nodes to which it connects as well as a boolean variable indicating whether or not the edge has been cut.

type $Edge = (NodeId,\ NodeId,\ Bool)$

D.3.1.3 Node structures

Each node is either a wall node or a coin. Coin nodes are divided into four categories depending on their valencies. Coins with attached edges store the identifiers of each attached edge.

```
data Node = Wall
          | EmptyNode                — Coin with no attached edges
          | Singleton EdgeId         — Coin with a single attached edge
          | BiNode (EdgeId, EdgeId)  — Coins with two attached edges
          | MultiNode [EdgeId]       — Coins with three or more attached edges
```

D.3.1.4 NodeTuple objects

It is often convenient to store both a node and its identifier in a single structure.

```
type NodeTuple = (NodeId, Node)
```

D.3.1.5 NodeMaps and EdgeMaps

All of the nodes and edges in the graph are stored in *Map* structures where each object can be accessed using its unique identifier.

```
type EdgeMap = M.Map EdgeId Edge
type NodeMap = M.Map NodeId Node
```

D.3.1.6 Database objects

All of the information in a graph is stored in a *Database* object. This object contains both the *NodeMap* and the *EdgeMap* associated with the graph.

```
type Database = (NodeMap, EdgeMap)
```

D.3.2 Node functions

- *createNode*
  This function creates a node given a set of *EdgeId*s. The cardinality of the set determines the node type.

```
createNode :: [EdgeId] → Node
createNode []     = EmptyNode
createNode [a]    = Singleton a
createNode [a, b] = BiNode (a, b)
createNode ns     = MultiNode ns
```

- *getNode*

  This function retrieves a node from the *Database* given its ID. An error will be thrown if the node cannot be found.

```
getNode :: Database → NodeId → Node
getNode (nm, _) n
                | M.member n nm = getElem n nm
                | otherwise = error "getNode : Node not in database"
```

- *isSingleton*

  This function indicates whether or not a node is a singleton.

```
isSingleton :: Node → Bool
isSingleton (Singleton _) = True
isSingleton _             = False
```

- *isBiNode*

  This function indicates whether or not a node is a binode.

```
isBiNode :: Node → Bool
isBiNode (BiNode _) = True
isBiNode _          = False
```

- *isMultiNode*

  This function indicates whether or not a node is a multinode.

```
isMultiNode :: Node → Bool
isMultiNode (MultiNode _) = True
isMultiNode _             = False
```

- *isTriNode*

  This function indicates whether or not a node is a multinode of valency three.

```
isTriNode :: Node → Bool
isTriNode (MultiNode ns) = (length ns == 3)
isTriNode _              = False
```

- *isEmptyNode*

  This function indicates whether or not a node is an emptynode.

```
isEmptyNode :: Node -> Bool
isEmptyNode EmptyNode = True
isEmptyNode _         = False
```

- *order*

  This function takes a pair of *NodeTuple* objects and arranges them in a useful order for pattern matching in other functions.

```
order :: (NodeTuple, NodeTuple) -> (NodeTuple, NodeTuple)
order x@((_, Wall), _)           = x
order (a, b@(Wall, _))           = (b, a)
order x@((_, Singleton _), _)    = x
order (a, b@(Singleton _, _))    = (b, a)
order x@((_, BiNode _), _)       = x
order (a, b@(BiNode _, _))       = (b, a)
order x                          = x
```

D.3.3 Edge functions

- *getEdge*

  This function retrieves an edge from the *Database* given its ID. An error will be thrown if the edge cannot be found.

```
getEdge :: Database -> EdgeId -> Edge
getEdge (_, em) e
                  | M.member e em = getElem e em
                  | otherwise = error "getEdge : Edge not in database"
```

- *isOpen*

  This function indicates whether or not an edge has been cut. A value of *False* indicates that the edge has been cut.

```
isOpen :: Edge -> Bool
isOpen (_, _, b) = b
```

- *nodes*

  This function takes an *Edge* object and returns the two nodes to which it is attached as well as their IDs.

```
nodes :: Edge → Database → (NodeTuple, NodeTuple)
nodes (a, b, _) db = order ((a, f a), (b, f b))
    where
      f x = getNode db x
```

D.3.4 Functions for graph traversal

- *hop*

  This function takes in an *Edge* and the *NodeId* of one the nodes to which it is connected and returns the *NodeId* of the other node. An error will be thrown if the input *NodeId* is not associated with the *Edge*.

```
hop :: Edge → NodeId → NodeId
hop (n1, n2, _) n
    | not $ any (n ==) [n1, n2] = error "hop : edge does not contain node n"
    | otherwise = head $ filter (n ≠) [n1, n2]
```

- *nextNode*

  This function takes a *NodeId* as well as the ID of an edge associated with the node and returns the next *NodeTuple* which is reached by following the edge.

```
nextNode :: NodeId → EdgeId → Database → NodeTuple
nextNode n e db = (n′, getNode db n′)
    where
      n′ = flip hop n $ getEdge db e
```

- *nextNode_*

  This function is almost identical to *nextNode* except that it returns only the *NodeId* of the next node.

```
nextNode_ :: NodeId → EdgeId → Database → Node
nextNode_ n e db = snd $ nextNode n e db
```

- *nextEdge*

  This function takes a *BiNode* and the ID of one of its associated edges and returns the ID of the other associated edge.

```
nextEdge_ :: Node → EdgeId → EdgeId
nextEdge (BiNode (a, b)) e = if a == e then b else a
nextEdge _ _              = error "nextEdge : invalid node type"
```

D.4 Chains

D.4.1 The chain data type

As was discussed in chapter 7, a chain data structure must store the set of nodes contained within the chain, the set of multinodes which bound the chain, and a label indicating if the chain is open, closed, or a loop chain.

```
type Chain = ([NodeId], [NodeTuple], Label)
data Label = Closed | Open | Loop
```

The above code defines a chain data structure to be a 3-tuple composed of a list of *NodeIds* (the IDs of the coins contained within the chain), a list of *NodeTuples* (the set of bounding multinodes), and a *Label* data type which can have the value *Closed*, *Open*, or *Loop*.

D.4.2 Chain functions

D.4.2.1 General chain functions

- *isClosedChain*
  This function indicates whether or not a chain is closed (and non-circular).

```
isClosedChain :: Chain -> Bool
isClosedChain (_, _, x) = (x == Closed)
```

- *isOpenChain*
  This function indicates whether or not a chain is open.

```
isOpenChain :: Chain -> Bool
isOpenChain (_, _, x) = (x == Open)
```

- *isLoop*
  This function indicates whether or not a chain is a loop.

```
isLoop :: Chain -> Bool
isLoop (_, _, x) = (x == Loop)
```

- *isDipperLoop*

  This function indicates whether or not a chain is a dipper loop. It returns *True* if the chain is bounded at both ends by the same trinode; otherwise it returns *False*.

```
isDipperLoop :: Chain -> Bool
isDipperLoop (_, [(ai, n), (bi, _)], _) = and [ai == bi, isTriNode n]
isDipperLoop _                          = False
```

- *fromChain*

  This function takes a chain and returns the list of nodes contained within the chain.

```
fromChain :: Chain -> [NodeId]
fromChain (ns, _, _) = ns
```

- *chainLen*

  This function returns the length of the input chain.

```
chainLen :: Chain -> Int
chainLen c = length $ fromChain c
```

- *getBoundingNodes*

  This function returns the list of bounding multinodes from an input chain. If the chain is not complex, then this function will return an empty list.

```
getBoundingNodes :: Chain -> [NodeTuple]
getBoundingNodes (_, ys, _) = ys
```

- *boundedBy*

  This function takes a chain and a *NodeId* and indicates whether or not the node is contained within the set bounding multinodes of the chain.

```
boundedBy :: NodeId -> Chain -> Bool
boundedBy ni (_, ys, _) = any ((ni ==) . fst) ys
```

- *handout*

  This function returns the minimum required handout to perform a double-dealing move on an independent, closed chain (simple or circular).

```
handout :: Chain -> Int
```

```
handout (_, [], x)
                    | x == Loop = 4
                    | x == Closed = 2
handout             _ = error "handout : invalid input"
```

- *effLen*

  This function returns the effective length of a chain. If the chain is not a dipper loop, then this function simply returns its length.

```
effLen :: Chain → [Chain] → Database → Int
effLen c@(_, (n : _), _) cs db
    | isDipperLoop c = sum [1, chainLen c, effLen' n c cs db]
    | otherwise      = chainLen c
effLen c _ _ = chainLen c
```

- *effLen′*

  This function is used as part of the *effLen* function stated above. It returns the length of the non-circular region of a dipper structure. It does this by examining the first node of the non-circular region. If this node is a singleton node, then it is an open 1-chain and thus will return 1, if this node is a binode, then the function returns the length of the chain which contains that node; otherwise, the function will return 0. Note that open 1-chains will not be detected by the chain detection algorithms and thus must be treated specially.

```
effLen' :: NodeTuple → Chain → [Chain] → Database → Int
effLen' (ni, MultiNode es) c cs db =
    let ns  = map (λ e → nextNode ni e db) es
        ns' = filter (λ (id, _) → not $ elem id $ fromChain c) ns
    in case ns' of
        [(_, Singleton _)] → 1
        [(id, BiNode _)]   → chainLen $ head $ filter (elem id . fromChain) cs
        [_]                → 0
        xs                 → error "effLen' : Invalid case"
```

D.4.2.2 Chain detection functions

- *getChains*

  This function returns the set of chains contained within the board by examining the associated *Database* object. It does this by calculating a list of all the binodes contained within the board and calls the chain detection function, *toChain*, on each binode which is not already contained within one of the detected chains.

```
getChains :: Database → [Chain]
getChains db@(nm, _) = foldl (fn) [] $ M.toList $ M.filter (isBiNode) nm
    where
      fn acc a@(ni, _)
                | any (elem ni) $ map (fromChain) acc = acc
                | otherwise = (toChain a db) : acc
```

- *toChain*

  This function is the implementation of algorithm 7.2. It takes a binode and calculates the sub-chains in either direction by following its two associated edges and calling the function *toChain′*. It then combines these sub-chains with the input binode to form the entire chain.

```
toChain :: NodeTuple → Database → Chain
toChain (ni, BiNode (a, b)) db = foldl (λ acc e → toChain′ (fn e) acc e db)
                                       ([ni], [], Closed)
                                       [a, b]
                                       where fn x = nextNode ni x db
```

- *toChain′*

  This function is the implementation of the function $f(\cdot)$ from algorithm 7.2. See chapter 7 for more information.

```
toChain′ :: NodeTuple → Chain → EdgeId → Database → Chain
toChain′ (ni, node) c@(ns, ys, x) e db
        | elem ni ns          = (ns, ys, Loop)
        | isBiNode node       = toChain′ t′ (ni : ns, ys, x) e′ db
        | isSingleton node    = (ni : ns, ys, Open)
        | isMultiNode node    = (ns, t : ys, x)
        | otherwise           = c
  where
    e′ = nextEdge node e
    t′ = nextNode ni e′ db
```

D.5 Action policy functions

D.5.1 Double-dealing decision functions

- *doubleDeal*

  This function determines whether or not a double-dealing move should be performed by examining the input list of chains. It makes its decision based on the theory outlined in chapter 6. Note that the function will return *False* if any of the following conditions are met:

  – There number of open chains is not equal to one.

  – There are multinodes in the database.

  – There are no long chains.

```
doubleDeal :: [Chain] → Database → Bool
doubleDeal cs (nm, _)
      | (0 ≠) $ M.size $ M.filter (isMultiNode) nm = False
      | length opens ≠ 1 = False
      | null longs = False
      | and [even ns, even n1] =  (n ≤) $ (2∗) $ sum
                                  [ceilDiv n1 2, (2∗) $ ceilDiv n2 2, rl]
      | and [even ns, odd n1] =   (n ≤) $ (2∗) $ sum
                                  [ceilDiv n1 2, (2∗) $ div n2 2, rl]
      | and [odd ns, odd n1] =    (n >) $ (2∗) $ sum
                                  [div n1 2, (2∗) $ ceilDiv n2 2, lo, rl]
      | and [odd ns, even n1] =   (n >) $ (2∗) $ sum
                                  [div n1 2, (2∗) $ div n2 2, lo, rl]
       where
         (opens, cs′) = L.partition (isOpenChain) cs
         (longs, shorts) = L.partition (λ c → (chainLen c) > 2) cs′
         (ones, twos) = L.partition (λ c → (chainLen c) == 1) shorts
         (end, longs′) = getEndChain longs
         [n1, n2, ns] = map (length) [ones, twos, shorts]
         n = sum $ map (chainLen) cs
         lo = chainLen $ head opens
         rl =foldl (λ acc c → acc − (handout c) + (chainLen c))
                (chainLen end) longs′
```

- *getEndChain*

  This function is called as part of the *doubleDeal* function. It takes a list of chains and returns the last chain expected to be opened as well as the list of chains with this chain removed. The last chain expected to be opened is the longest chain in the set. Priority is given to loop chains over non-circular chains of the same length.

```
getEndChain :: [Chain] → (Chain, [Chain])
getEndChain cs =
    let   l x = chainLen x
          f (m, s) c | (l c) > (l m) = (c, m : s)
                     | (l c) < (l m) = (m, c : s)
                     | isLoop c      = (m, c : s)
                     | otherwise     = (c, m : s)
    in    foldl (f) (head cs, []) $ tail cs
```

- *doubleDealComplex*

  This function determines whether or not to perform a double-dealing move when there are complex chains on the board. It makes the decision using the theory outlined in chapter 7. The function will immediately return *False* if any of the following conditions are met:

  – The number of open complex chains on the board is not equal to one.

  – There is one open complex chain on the board but the chain is not bounded by a trinode

  If none of the above conditions are met, then there is exactly one open complex chain on the board. The function converts the board into an equivalent scenario where all the chains are independent and then makes the decision using the *doubleDeal* function. See chapter 7 for a more detailed explanation.

```
doubleDealComplex :: [Chain] → EdgeId → Database → Bool
doubleDealComplex cs e (nm, em)
     | (1 ≠) $ length opens = False
     | f c = let n = fst $ head $ getBoundingNodes c
                 cs′ = filter (not . boundedBy n) cs
                 a = (fromChain c, [], Open)
                 nm′ = M.map (λ n → removeEdge n e) nm
                 b = toChain (n, getElem n nm′) (nm′, em)
             in  doubleDeal (a : b : cs′) (nm′, em)
     | otherwise = False
```

```
where
  opens = filter (λ x → and [isOpenChain x, isComplex x]) cs
  c = head opens
  f (_, ys, _) = any (isTriNode . snd) ys
```

D.5.2 Edge priority functions

- *getPriority*

  This function assigns a priority value to an edge by examining the two nodes to which it connects using pattern matching. These two nodes are input as a pair of *NodeTuple* objects which are assumed to have been ordered using the *order* function. Open 1-chains and dipoles are detected here and further analysis is performed on edges which are connected to at least one binode.

```
getPriority :: (NodeTuple, NodeTuple) → EdgeId → [Chain] →
                    Database → Int
getPriority ((_, Wall), (_, Singleton _)) _ _ _                   = 3
getPriority ((_, Wall), (ni, node@(BiNode _))) e cs db            = †
getPriority ((_, Wall), (_, MultiNode _)) _ _ _                   = 0
getPriority ((_, Singleton _), (_, Singleton _)) _ _ _            = 3
getPriority ((_, Singleton _), (_, BiNode _)) _ _ _               = 1
getPriority ((_, Singleton _), (_, MultiNode _)) _ _ _            = 3
getPriority ((na, a@(BiNode _)), (nb, b@(BiNode _))) e cs db      = ††
getPriority ((ni, node@(BiNode _)), (_, MultiNode _)) e cs db     = †††
getPriority ((_, MultiNode _), (_, MultiNode _)) _ _ _            = 0
getPriority _ _ _ _ = error "getPriority : Invalid Pattern"


†   getPriorityB n′ ni cs db
  where
    n′ = nextNode_ ni (nextEdge node e) db
††  getPriorityG (f na a, f nb b) na cs db
  where
    f n x = nextNode_ n (nextEdge x e) db
††† getPriorityH n′ ni e cs db
  where
    n′ = nextNode_ ni (nextEdge node e) db
```

- *getPriorityB*

  This function is called as part of *getPriority* when the edge connects a wall node to a binode. It examines the other node which is attached to the binode by pattern matching and assigns a priority value depending on what type of node this is.

  If the node is a singleton then the structure is an open 2-chain and cutting the edge would perform a double-dealing move. The function*doubleDeal* is called upon to see if the double-dealing move should be performed. If so, then the edge is assigned a priority value of 2 (the second highest possible priority value); if not, the edge is assigned a priority value of -4 (-2 times the length of the chain).

  If the node is not a singleton then the function returns a priority value of -2 times the effective length of the chain containing the binode.

```
getPriorityB :: Node → NodeId → [Chain] → Database → Int
getPriorityB (Singleton _) _ cs db = if doubleDeal cs db then 2 else − 4
getPriorityB _ ni cs db            = − 2 ∗ le
     where
       le = effLen c cs db
       c  = head $ filter (elem ni . fromChain) cs
```

- *getPriorityH*

  This function is similar to *getPriorityB* only that it is called upon if the edge connects a binode to a multinode. Again, the other node connected to the binode is examined and the priority is assigned depending on its type.

  If the node is a singleton, then a double-dealing opportunity has presented itself. Since the open 2-chain is complex, then the function *doubleDealComplex* is called upon to decide whether or not to perform the double-dealing move.

  If the node is not a singleton then the function returns a priority value of -2 times the effective length of the chain containing the binode.

```
getPriorityH :: Node → NodeId → EdgeId → [Chain] → Database → Int
getPriorityH (Singleton _) _ e cs db = if doubleDealComplex cs e db
                                       then 2 else − 4

getPriorityH _ ni _ cs db            = − 2 ∗ le
     where
       le = effLen c cs db
       c  = head $ filter (elem ni . fromChain) cs
```

- *getPriorityG*

  This function is called upon as part of *getPriority* when the edge connects two binodes together. The function uses pattern matching on the other node connected to each binode in order to assign an priority value.

  If both of these nodes are singletons, then a double-dealing opportunity has been detected. The function *doubleDeal* is called to decide whether or not to perform the double-dealing move. If not, the edge is assigned a priority of -8 (-2 times the length of the chain).

  If the nodes are each one of a wall node or a multinode, then cutting the edge would perform a hard-hearted handout. This move is assigned a priority value of -3, slightly higher priority than a typical action which opens a closed 2-chain.

  If none of the above cases are met, then function returns a priority of -2 times the effective length of the associated chain. If the associated chain is a loop chain, then the assigned priority value is increased by 1, so that opening a loop chain would have a higher priority than opening a non-circular chain of the same length.

```
getPriorityG :: (Node, Node) -> NodeId -> [Chain] -> Database -> Int
getPriorityG (Singleton _, Singleton _) _ cs db = if doubleDeal cs db
                                                     then 2 else -8

getPriorityG (Wall, Wall) _ _ _                  = -3
getPriorityG (Wall, MultiNode _) _ _ _           = -3
getPriorityG (MultiNode _, Wall) _ _ _           = -3
getPriorityG (MultiNode _, MultiNode _) _ _ _    = -3
getPriorityG _ ni cs db = (-2 * le) + (if isLoop c then 1 else 0)
  where
    le = effLen c cs db
    c = head $ filter (elem ni . fromChain) cs
```

D.5.3 The move selection function

This function analyses all of the open edges on the board and calculates their priority values using the *getPriority* function. It then returns an edge selected at random from the set of edges with the highest priority values.

```
makeMove :: Database -> StdGen -> EdgeId
makeMove db@(_, es) gen = M.keys ps' !! (randIndex gen $ M.size ps')
  where
    cs   = getChains db
    f x  = getPriority (nodes (getEdge db x) db) x cs db
    ps   = foldl (\acc e -> M.insert e (f e) acc) (M.fromList [])
                 $ M.keys $ M.filter (isOpen) es

    max  = maximum $ M.elems ps
    ps'  = M.filter (== max) ps
```

D.6 Update functions

- *update*

  This function takes a *Database* object and an *EdgeId* and returns an updated *Database* object with the indicated edge having been cut. The *Edge* object is updated by setting its boolean value to *False* and the edge is removed from the *Node* objects by use of the *removeEdge* function.

```
update :: Database -> EdgeId -> Database
update (nm, em) e = (nm', em')
  where
    nm' = M.map (\n -> removeEdge n e) nm
    em' = M.update (\(a, b, _) -> Just (a, b, False)) e em
```

- *removeEdge*

  This function takes a *Node* object and an *EdgeId* and removes the edge from the node, if it was connected to the node. It achieves this by compiling all of the edges associated with the node into a list and then removes all instances of the input edge from the list. This new list is then used to generate a new *Node* object using the *createNode* function.

```
removeEdge :: Node → EdgeId → Node
removeEdge (Singleton a) e     = createNode $ removeElem [a] e
removeEdge (BiNode (a, b)) e   = createNode $ removeElem [a, b] e
removeEdge (MultiNode ns) e    = createNode $ removeElem ns e
removeEdge EmptyNode _         = EmptyNode
removeEdge Wall _              = Wall
```

D.7 Other functions and data types

Although many other functions and data types were written for the program such as initialisation functions, functions and data types for the interactive user interface etc., these were omitted from this document for brevity. The source code for the project can be found in the CD submitted with this document and all functions and data types can be found there.

Bibliography

[1] ShareLaTeX. URL www.sharelatex.com.

[2] Miran Lipovača. *Learn you a Haskell for great good!: a beginner's guide*. 2011.

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

[4] Aviezri S Fraenkel. Combinatorial games: selected bibliography with a succinct gourmet introduction. *Electron. J. Combin*, 1, 1994.

[5] Elwyn R Berlekamp, John H Conway, and Richard K Guy. Winning ways for your mathematical plays, volume 4. *AMC*, 10:12, 2003.

[6] Charles L Bouton. Nim, a game with a complete mathematical theory. *The Annals of Mathematics*, 3(1/4):35–39, 1901.

[7] R. Sprague. Über mathematische kampfspiele. *Tohoku Mathematical Journal, First Series*, 41.

[8] Patrick M Grundy. Mathematics and games. *Eureka*, 2(5):6–8, 1939.

[9] Richard Bird. Functional pearl: A program to solve sudoku. *Journal of Functional Programming*, 16(06):671–679, 2006.

[10] hackage.haskell.org. Data.list, . URL https://hackage.haskell.org/package/base-4.7.0.2/docs/Data-List.html.

[11] Stephen Adams. Efficient sets: a balancing act. *Journal of functional programming*, 3(04):553–561, 1993.

[12] Jürg Nievergelt and Edward M Reingold. Binary search trees of bounded balance. *SIAM journal on Computing*, 2(1):33–43, 1973.

[13] hackage.haskell.org. Data.map, . URL https://hackage.haskell.org/package/containers-0.4.0.0/docs/Data-Map.html.