Techniques for Solving and Visualizing Large Games

Cameron Cheung
Dan Garcia, Ed.
Joshua Hug, Ed.

Electrical Engineering and Computer Sciences
University of California, Berkeley

Technical Report No. UCB/EECS-2023-186
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2023/EECS-2023-186.html

May 19, 2023

Copyright © 2023, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

Acknowledgement

Thank you to Professor Dan Garcia, for his guidance over the course of my projects and the writing of this report. Thank you to Professor Joshua Hug who kindly agreed to be my second reader. Some game interfaces presented in the Image AutoGUI chapter were developed in collaboration with Professor Garcia, Robert Shi, Arihant Choudhary, Harnoor Dhillon, Kaelyn Huang, and Jingfan Xia.

Techniques for Solving and Visualizing Large Games

by Cameron Cheung

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

Committee:

Teaching Professor Dan Garcia
Research Advisor

Date

* * * * * * *

Teaching Professor Josh Hug
Second Reader

Date

Abstract

We solved two large games, Nine Men's Morris and Quarto. These games involve multipart moves – moves that require more than one decision. We demonstrate the versatility of our techniques by using them to solve other games. We present position counts by value-remoteness for all games we solved. We discuss the optimizations used to speed up the solving process and to store the results in a way that is efficient for both solving and playing. Our report also covers the development of a new Image AutoGUI system. This tool supports multipart moves and streamlines the interface development process. We demonstrate the use of this tool by implementing multiple game interfaces. These projects were developed as part of the GAMESMAN software infrastructure, specifically the GamesmanClassic backend and the web application GamesmanUni.

Acknowledgments

Thanks to my advisor, Professor Dan Garcia, for his generosity and guidance over the past couple years, for the exciting game theory discussions, and for creating an environment in which I have improved my skills, met amazing people, and tackled interesting projects.

Thanks to Professor Josh Hug, who has kindly agreed to be my second reader.

Thanks also to both professors for teaching two of my most favorite classes I have taken at UC Berkeley, CS61B and CS61C.

Thanks to the GamesCrafters group for the software they have created that much of the work presented in this report is built upon. Thanks especially for the fantastic meetings we have had.

Credit goes to the following for some of the Image AutoGUIs presented in Chapter 5:

- To Arihant Choudhary, whom I worked with on the QuickCross interface.
- To Harnoor Dhillon and Kaelyn Huang, whom I worked with on the Achi interface.
- To Jingfan Xia, whom I worked with on the Mū Tōrere interface.
- To Professor Dan Garcia, whom I worked with on the Nine Men's Morris and Toot-and-Otto interfaces.

Credit goes to Andrew Lee who helped with Appendix A.

Credit goes to Robert Shi, Victoria Phelps, and Jatearoon "Keene" Boondicharern, whom over the past few semesters I have heavily discussed ideas with.

Thank you to to Matthew Yu, Felicity Wang, Alexis Tran-Luu, Lyle Lalunio, Rita Wang, and Elicia Ye.

Thanks to my family, most of all, for their support.

Contents

1 Introduction **7**

2 Background **8**

2.1 Definitions . . . 8

2.1.1 Tiers . . . 10

2.2 GamesCrafters Projects and Infrastructure . . . 12

2.3 GamesCrafters User Interfaces . . . 12

2.4 Large Games . . . 13

2.5 Machine . . . 13

3 Solving Quarto **14**

3.1 Rules . . . 14

3.2 Position Definition and String Representation . . . 15

3.3 Tier Definition . . . 17

3.3.1 Upper-Bounding the Number of Positions, Ignoring Symmetry . . . 18

3.4 Non-string Position Representations . . . 18

3.5 Hashing . . . 19

3.5.1 Hashing/Unhashing a Level 16 Position . . . 19

3.5.2 Optimizations . . . 20

3.5.3 Generalizing the Algorithm to a Position Outside of L_{16} . . . 21

3.6 Primitive/Unreachable Check . . . 21

3.7 Symmetries . . . 23

3.7.1 Board Slot Symmetries . . . 23

3.7.2 Piece Bit Symmetries . . . 25

3.7.3 Canonicalization and Tier Symmetries . . . 26

3.7.4 Symmetry Tables . . . 27

3.7.5 Upper-Bounding the Number of Positions, Utilizing Tier Symmetry . . . 27

3.8 Encoding Value-Remoteness . . . 28

3.8.1 Level Value-Remoteness Encoding Table . . . 28

3.8.2 Determining the Value-Remoteness of a Parent Position . . . 29

3.8.3 Encoding as a Bitstring . . . 30

3.9 Database and Live-Solving . . . 30

3.10 Solving . . . 32

3.10.1 Overview . . . 32

3.10.2 Optimizations . . . 35

3.11 Results . . . 35

3.11.1 Timing . . . 35

3.11.2 Value Counts . . . 35

3.12 Value-Moves Interface . . . 36

3.13 Reflection and Future Work . . . 37

4 Solving Tierable Loopy Games **39**
4.1 Solver . . . 39
4.2 General Strategies . . . 40
4.2.1 Database Compression . . . 41
4.3 Nine Men's Morris . . . 42
4.3.1 Rules . . . 42
4.3.2 Position Representation . . . 43
4.3.3 Tier Definition . . . 43
4.3.4 Hashing . . . 43
4.3.5 Symmetry . . . 44
4.3.6 Results . . . 44
4.3.7 Variants . . . 44
4.4 Bagh-Chal . . . 44
4.4.1 Rules . . . 44
4.4.2 Position Representation . . . 47
4.4.3 Tier Definition . . . 47
4.4.4 Results . . . 48
4.5 Tic-Tac-Two . . . 48
4.5.1 Rules . . . 48
4.5.2 Position Representation . . . 49
4.5.3 Tier Definition . . . 49
4.5.4 Results . . . 49
4.6 Topitop . . . 50
4.6.1 Rules . . . 50
4.6.2 Position Representation . . . 52
4.6.3 Tier Definition . . . 52
4.6.4 Results . . . 53
4.7 Future Work . . . 53

5 Image AutoGUI **54**
5.1 Overview . . . 54
5.2 Example: Achi AutoGUI . . . 54
5.2.1 Coordinates and SVGs . . . 54
5.2.2 Rendering Positions . . . 56
5.3 General Structure of Image AutoGUI Data . . . 59
5.4 Designing an Image AutoGUI for a Game . . . 60
5.5 New Interfaces Using Image AutoGUI . . . 61
5.5.1 Bagh-Chal . . . 62
5.5.2 Chess Endgame . . . 63
5.5.3 Chomp . . . 64
5.5.4 Connect 4 . . . 65
5.5.5 Dawson's Chess . . . 66
5.5.6 Dodgem . . . 67
5.5.7 Mū Tōrere . . . 68
5.5.8 QuickCross . . . 69
5.5.9 Shift-Tac-Toe . . . 70
5.5.10 Snake . . . 70
5.5.11 Tac Tix . . . 71
5.5.12 Toot-and-Otto . . . 72
5.6 Future Work . . . 73

6 Multipart-Move Interfaces **74**
6.1 Motivation . . . 74
6.2 Multipart-Move Interface Design and Implementation . . . 75
6.3 A Game in and of Itself . . . 75
6.4 New Image AutoGUI Interfaces Involving Multipart Moves . . . 76
6.4.1 L-game . . . 76
6.4.2 3-Spot . . . 77
6.4.3 Tic-Tac-Two . . . 77
6.4.4 Nine Men's Morris . . . 78
6.4.5 Topitop . . . 79
6.4.6 Chung-Toi . . . 80
6.5 Future Work . . . 81

7 Conclusion **82**

Bibliography **82**

A Explanation of Rules for Various Games **85**

B Value-Remoteness Counts **88**

C Code **94**
C.1 Quarto Tier Struct and Constructor . . . 94
C.2 Quarto Hash and Unhash . . . 94
C.3 Multipart Edge Struct . . . 95
C.4 Multipart Move Graph Generation for Topitop . . . 96

Chapter 1

Introduction

UC Berkeley GamesCrafters is a research and development group founded by Teaching Professor Dan Garcia in 2001. It has helped train hundreds of students in computational game theory through various projects including strongly solving games and implementing game interfaces.

This technical report consists of two major parts.

The first part describes our work in solving large games. In Chapter 3, we discuss Quarto, a tierable non-loopy game that we solved with custom code [1]. In Chapter 4, we discuss Nine Men's Morris, a tierable loopy game we solved using the GamesmanClassic [2] framework. We discuss solving strategies and optimizations that were used in the Nine Men's Morris solve. Later in Chapter 4 we also provide details on solving other games (Bagh-Chal, Tic-Tac-Two, and Topitop) to illustrate how this methodology is applied to tierable loopy games in general.

The second part of this report discusses the process of adding several new game interfaces to our web application, GamesmanUni. Many games that have been solved by the GamesCrafters group over the years remain inaccessible to the public because user interfaces for those games have not yet been implemented. The GamesmanUni interfaces not only provide the ability to play the game against another player or a computer, but they also provide details on legal moves' game-theoretic values which are important for gaining insight on strategy. In Chapter 5 we discuss the Image AutoGUI system, which allows developers to add more user-friendly interfaces to GamesmanUni with relatively little work. Additionally, some game interfaces involve multipart moves, which are moves that entail more than one decision. In Chapter 6, we discuss the considerations behind the creation of the multipart move system and how one can incorporate multipart moves into their game interface.

Chapter 2

Background

2.1 Definitions

An abstract strategy game [3] is a game in which players know the set of legal moves available to any player on their turn (perfect information), and the outcome of the game depends solely on the move choices that players make. No other factors, like randomness, influence the outcome of an abstract-strategy game. All games described in this report involve alternating turns between two players.

A game $\mathcal{G} = (G, V)$ consists of a graph G and a primitive value function V [4]. $G = (P, M)$ is a weakly connected directed graph made up of a position/vertex set P and a move/edge set M such that there is a single source vertex, the **initial position**. The games that we study require that for each vertex $p \in P$ there exists a directed path from p to a sink vertex. Sink vertices are referred to as **primitive** positions and represent the end of the game. V is a function that assigns a value **win**, **lose**, or **tie** to each primitive position. G is generated according to the game's movement rules and V is defined by the game's winning objective.

$\mathcal{G}$ is a **loopy game** if G is acyclic; otherwise it is **loop-free**.

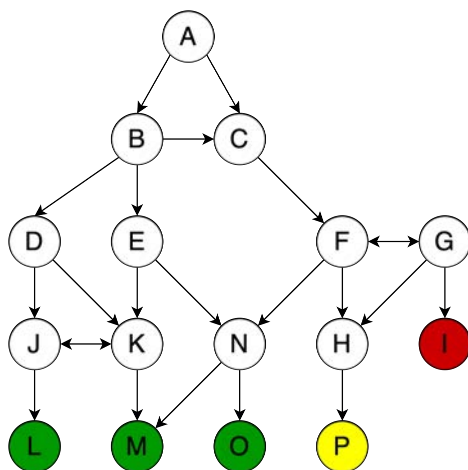

Figure 2.1: An example game $\mathcal{G}_x = (G_x, V_x)$

Refer to Figure 2.1. Position A is the initial position. Positions I, L, M, O, and P are primitive. $V_x(\text{I}) =$ LOSE, $V(\text{L}) = V(\text{M}) = V(\text{O}) =$ WIN, $V(\text{P}) =$ TIE. $\mathcal{G}_x$ is a loopy game (see positions F, G, J, and K).

If there exists a legal move from position p_1 to position p_2, then p_2 is a **child position** of p_1 and p_1 is a **parent position** of p_2.

We use the following definition of perfect play:

- If forcing a win is possible, then force a win in as few moves as possible.
- Otherwise, if forcing a tie is possible, then force a tie in as few moves as possible.
- Otherwise, force a lose in as many moves as possible.

The **value** of a position is the outcome of the game for the player whose turn it is assuming perfect play. If at a particular position p, the player whose turn it is at p can force a win, then p's value is WIN. If the player cannot force a win but can force a tie, then p's value is TIE. Otherwise, p's value is LOSE. The value of a position in a game $\mathcal{G} = (G, V)$ is recursively defined as follows...

- The value of a primitive position p_r is $V(p_r)$.
- The value of a position that has at least one LOSE child position is WIN.
- The value of a position that has no LOSE child position but at least one TIE child position is TIE.
- The value of a position that only has WIN child positions is LOSE.

In addition, a position's value is DRAW if neither player can force a win or a tie.

The value of a move is the perfect-play outcome of the game for the current player assuming they make that move. Recall that the value of a position is from the perspective of whose turn it is at a position. A **winning move** leads to a LOSE child position. A **tying/drawing move** leads to a TIE/DRAW child position. A **losing move** leads to a WIN child position.

A position's **remoteness** is the number of moves until the end of the game assuming perfect play.

- The remoteness of a primitive position is 0.
- The remoteness of a non-primitive win position is 1 + the remoteness of minimum-remoteness losing child position.
- The remoteness of a non-primitive lose position is 1 + the remoteness of the maximum-remoteness child position.
- The remoteness of a non-primitive tie position is 1 + the remoteness of the minimum-remoteness tying child position.
- The remoteness of a draw position is ∞.

We define the following ordering of value-remoteness pairs as worst to best: low-remoteness lose < high-remoteness lose < draw < high-remoteness tie < low-remoteness tie < high-remoteness win < low-remoteness win.

GamesCrafters uses the following stoplight-inspired color convention: green (good) represents WIN, yellow (moderate) represents TIE/DRAW, and red (bad) represents LOSE.

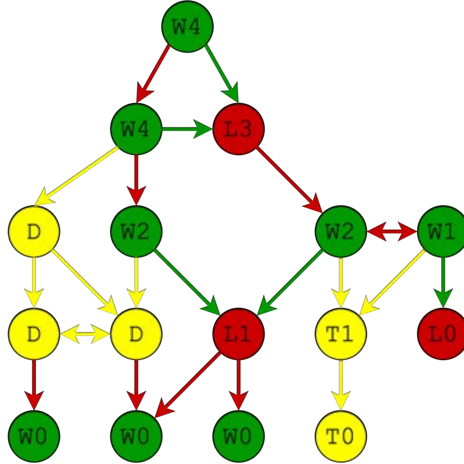

Figure 2.2: Example game G_x solved. Colors and letters represent value and numbers represent remoteness. This game is a win for the first player, who can guarantee that they win in four total moves (by both players). If the first player plays perfectly, the second player cannot change the outcome or extend the game. In general, if the first player does not play perfectly, the game may still be won but perhaps take longer; or if they make a drawing or tying move the game may end in a draw or tie; or if they make a losing move, "control" passes to the second player who can win if they themselves play perfectly.

Consider a game $\mathcal{G} = (G, V)$ where $G = (P, M)$. The **subgame** of $\mathcal{G}$ rooted at $p \in P$ is a game $\mathcal{G}^p = (G^p, V)$ where G^p is the induced subgraph of G from the position subset of P consisting of only the positions p' for which there exists a directed path from p to p'.

If the subgames rooted at p_1 and p_2 are equivalent, then p_1 and p_2 are **symmetric** positions. For example, rotating a tic-tac-toe board 90° yields a symmetric position.

2.1.1 Tiers

Suppose a game's graph is $G = (P, M)$.

1. Partition P into vertex subsets $T_1, T_2, ..., T_N$ according to a rule D which indicates which positions belong to which subsets. N depends on D. Specifically, D is a function that maps each position p to one of the N subsets.
2. On G, for $j = 0, 1, ..., N$, perform a vertex contraction [5] on T_j. The resulting graph is G'.

If G' is acyclic, then D is a **valid tier definition**. Then G' is a **tier graph** and $T_0, T_1, ..., T_N$ are **tiers** defined by D.

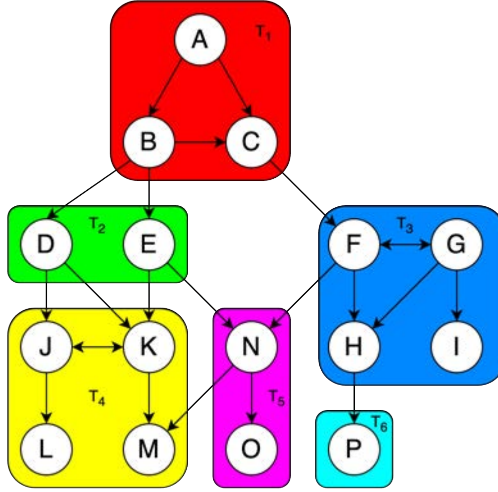

Figure 2.3: Tier Definition

Example: Again consider the example game $\mathcal{G}_x$. A rule D partitions the vertices of G_x into subsets T_1 = {A, B, C}, T_2 = {D, E}, T_3 = {F, G, H, I}, T_4 = {J, K, L, M}, T_5 = {N, O}, T_6 = {P} (Figure 2.3). If we perform a vertex contraction on each of $T_1, T_2, ..., T_6$, then we obtain a directed acyclic graph (see Figure 2.4), so D is a valid tier definition and $T_1, T_2, ..., T_6$ are tiers.

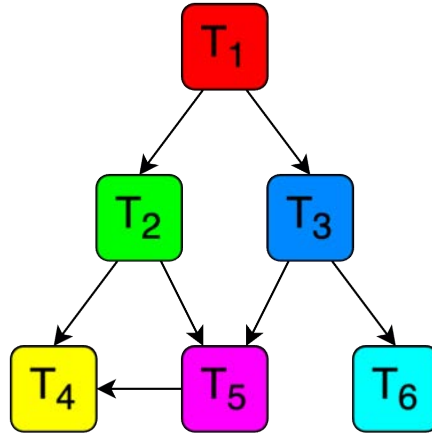

Figure 2.4: Tier Graph

A tier T_c is a **child tier** of another tier T_p if there exists a position in T_p with a child position in T_c.

A tier definition D is **trivial** if all non-primitive positions belong to the same tier according to D. A game is **tierable** if there exists a nontrivial tier definition.

In a **retrograde tier solve** [6] we only solve a tier once all of its child tiers have been solved (i.e., once all positions across all child tiers are solved) [7]. To perform a retrograde tier solve on 2.4, we can, for example, solve T_4 and T_6 first. T_5 is solved after T_4 is solved. T_3 is solved after T_5 and T_6 are solved. In Chapters 3 and 4 we discuss how to solve a tier given its solved child tiers. One benefit of tiersolving is that we can solve a game by only loading subsets of positions into memory at a time. Specifically, when solving a tier we need only load the positions in its child tiers.

2.2 GamesCrafters Projects and Infrastructure

Figure 2.5: GamesCrafters Servers

GamesmanUni [8] is a web application where users can play solved games with interfaces that show game-theoretic values of positions and moves. GamesmanUni makes API calls to the GamesCraftersUWAPI server to get data about game positions so that it knows what pieces and moves to render.

GamesCraftersUWAPI (Gamescrafters Universal Web API) [9] is a middleware that connects Gamesman frontend applications to different backend game servers. For games written in C served by GamesmanClassic, GamesCraftersUWAPI translates requests and responses between GamesmanUni and GamesmanClassic. There exist other backends such as GamesmanJava which serves games written in Java.

GamesmanClassic [10] is a project that serves as a framework for solving and playing abstract strategy games. It contains various solvers including a retrograde tier solver that we use to solve the games presented in Chapter 4.

2.3 GamesCrafters User Interfaces

GamesmanClassic and GamesmanUni come with solved-game interfaces that follow the color convention for position and move values. If a button to perform a move is red, then it corresponds to a losing move, for example.

For some of its games, GamesmanClassic has text and Tcl/Tk interfaces. To play with them, however, one needs to install and set up GamesmanClassic on their computer. This is part of the motivation for putting more games on GamesmanUni so that the games are more accessible.

GamesmanUni renders positions and buttons for performing moves according to a string representations of positions and moves. Position and move strings are relayed to GamesmanUni by GamesCraftersUWAPI and are referred to as UWAPI position and move strings.

- Custom GUI: Renders a position in a way specific to a game. Custom GUIs are typically created for games with interface features not supported by AutoGUI.

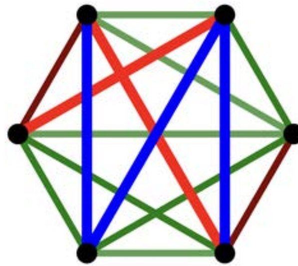

Figure 2.6: Sim Custom GUI

- Character AutoGUI: Renders a position in a grid-based format with characters representing pieces. A UWAPI position string contains information about the dimensions of the grid and the characters at each grid slot.

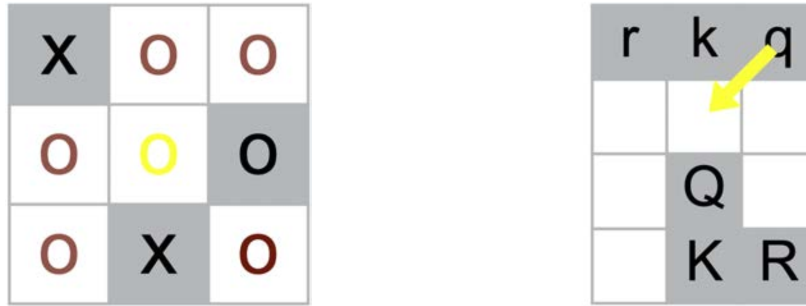

Figure 2.7: (Left) Tic-Tac-Toe Character AutoGUI and (Right) 4-by-3 Chess Character AutoGUI

Using the Character AutoGUI, a game interface creator does not need to handle features such as coloring move buttons according to move value and move button hover animations; those are handled automatically. On the other hand, a person creating a Custom GUI needs to handle those features. In Chapter 5, we discuss a new way to render a game, the Image AutoGUI.

2.4 Large Games

The game of Nine Men's Morris has been solved by Gasser [11], who found that it is a draw. It has since been strongly solved by other teams [12]. In Chapter 4, we explain how we solved it within the GamesmanClassic framework. It is the largest game (in terms of number of positions) with no known closed-form solution solved in GamesmanClassic to date.

The game of Quarto was solved by Goosens but seemingly his work can only be found in the "Internet Archive" [13]. We will demonstrate the techniques on how we solved Quarto in Chapter 3, including a novel way of defining tiers.

2.5 Machine

All solving presented in this paper was done on a dedicated machine assembled in 2021 with the following specs:

- CPU: AMD Ryzen 9 5900, 12-Core 3.8GHz
- 128 GB DDR4 RAM
- 1 TB SSD
- 10 TB Hard Drive

Chapter 3

Solving Quarto

3.1 Rules

There are 16 pieces. 8 of them are dark (`isLight`=FALSE) and 8 of them are light (`isLight`=TRUE). 8 of them are circular (`isSquare`=FALSE) and 8 of them are square (`isSquare`=TRUE). 8 of them are hollow (`isFilled`=FALSE) and 8 of them are filled (`isFilled`=TRUE). 8 of them are short (`isTall`=FALSE) and 8 of them are tall (`isTall`=TRUE). No two pieces are identical; for any Boolean variable assignment to `isLight`, `isSquare`, `isFilled`, and `isTall`, there is exactly one piece that is characterized by that assignment. In Figure 4.1 below, short and tall pieces are represented as small and large pieces.

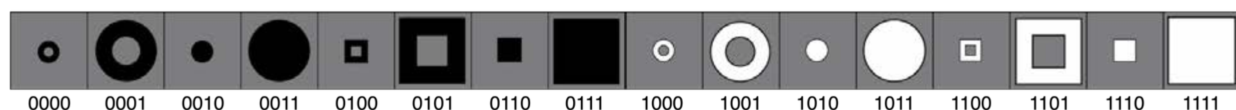

Figure 3.1: Quarto Pieces

Each piece can thus be represented as a nibble. Let the most significant bit represent `isLight`, the second most significant bit represent `isSquare`, the second least significant bit represent `isFilled`, and the least significant bit represent `isTall`. For example, piece `0b0101` = `0x5` is the dark, square, hollow, tall piece and `0b1100` = `0xC` is the light, square, hollow, short piece.

The game is played on a 4×4 grid. Let the slots of the grid be assigned $0, 1, 2, ..., 15$ in row-major order. The objective is to create a horizontal, vertical, or diagonal four-in-a-row of pieces (*quartet*) on the grid such that each piece in that four-in-a-row is similar in some way (e.g., all dark or all circular). In terms of the nibble representation, four pieces are similar in some way if and only if there is a matching bit across all four pieces, i.e., there exists $i \in \{0, 1, 2, 3\}$ such that the i-th bit is set among all four pieces or the i-th bit is unset among all four pieces. We refer to a four-in-a-row of pieces that are similar in some way as a *win quartet*. The player to place the fourth piece of a win quartet wins. Any piece quartet that is not a win quartet is a *tie quartet*.

At the beginning of the game, the first player chooses a piece for the second player to place. Then, the second player places that piece on an empty slot on the board and chooses one of the remaining pieces for the first player to place next. Then the first player places that piece on an empty slot on the board and chooses one of the remaining pieces for the second player to place, and so on. The game ends when either (1) a win quartet has been created or (2) when the board has been filled with no win quartets having been created, in which case the game is a tie.

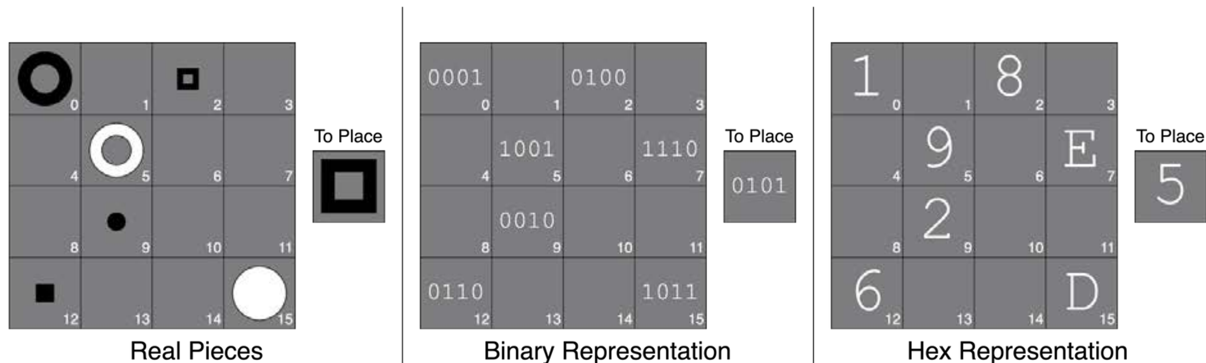

Figure 3.2: Example Position

From now on each piece will be referred to by its nibble representation (e.g., `0b1010` or `0xA`) rather than its real characteristics (e.g., light, circular, filled, short).

3.2 Position Definition and String Representation

Note: We take "position" and "board" to mean different things for this game. A position consists of a board, which has a particular arrangement of pieces, and a piece-to-place, which is either (1) a piece or (2) null, when there is no piece to place.

There are three types of positions: the initial position, primitive positions, and middlegame (neither initial nor primitive) positions.

A position can be represented by a 17-character string containing hex digits (`"0"`, `"1"`, ..., `"F"`) and dashes `"-"`. The first sixteen characters represent the pieces on the board, with dashes representing unoccupied slots. The last character represents the piece-to-place, in which case a dash means that there is no piece-to-place (i.e., either the initial position or a primitive position is represented).

The initial position is the state of the game before the first player has chosen a piece for the second player to place (the piece-to-place is null). The initial position's child positions are the states of the game after the first player has made that choice but before the second player has decided where to place the chosen piece. Once the values of the initial position's 16 child positions are known, it is simple to determine the value and remoteness of the initial position. Our discussion about solving mostly ignores the initial position henceforth. The non-string position representations do not apply to the initial position since it is abnormal in that it is the only position in the game in which the player only chooses a piece. The string representation of the initial position is `"-----------------"`.

A middlegame position is defined by the state of the board and the piece that the player whose turn it is must place.

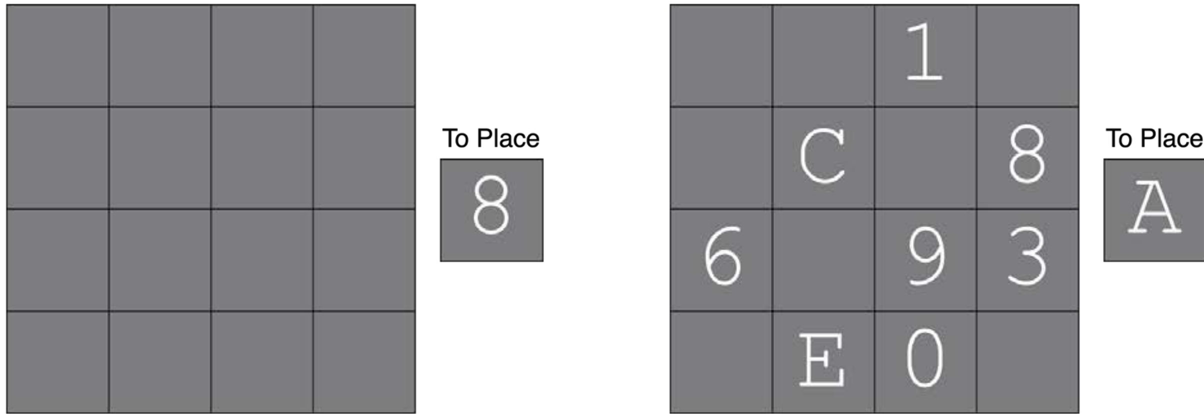

Figure 3.3: Middlegame Positions "----------------8" and "--1--C-86-93-E--A"

A primitive position is a position in which the game has ended (i.e., there exists a win quartet or the board is filled). In this position, the piece-to-place is null because no more placements occur.

We need not store a turn bit for each position because the number of pieces on the board tells us whose turn it is. It is the first player's turn when the number of pieces on the board is odd, and the second player's turn when the number of pieces on the board is even (except at the initial position).

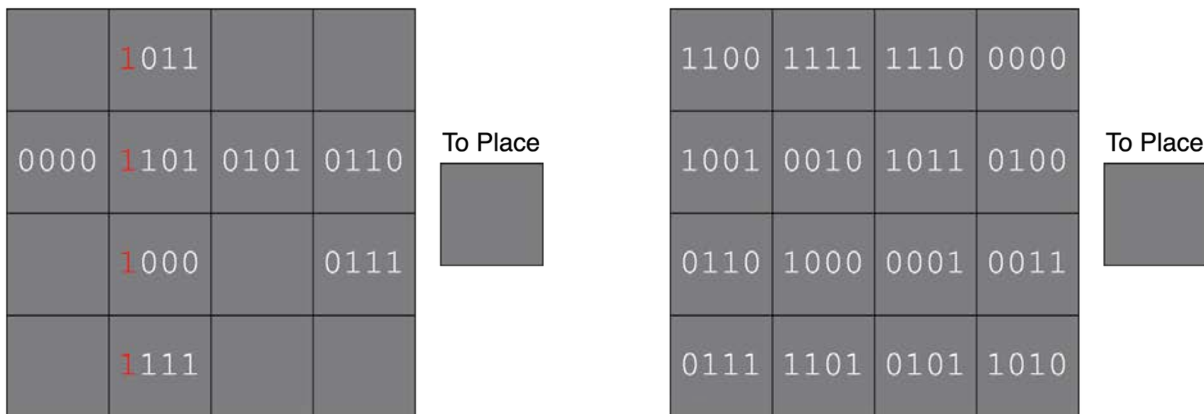

Figure 3.4: Primitive Lose Position "-B--0D56-8-7-F---" and Primitive Tie Position "CFE092B468137D5A-"

The other ways we represent positions, which will soon be introduced, represent every possible state that can result from the following construction:

1. Choose any subset of pieces that will be on the board (e.g., {0x0, 0x3, 0x6, 0x9, 0xC}).
2. For each piece in that subset, arrange them in some way on the board, i.e., injectively assign each piece in the subset a slot (e.g., 0x0 is on slot 8, 0x3 is on slot 5, 0x9 is on slot 0, etc.).
3. Of the pieces not in the subset, select one which will be the piece-to-place (e.g., 0x8).

The first consequence of these kinds of representation is the inclusion of unreachable positions, so we need to identify and skip them while solving and exclude them while counting positions. The second consequence is that primitive positions (besides those with 16 pieces on the board) now are given an inherent piece-to-place and there is a redundancy of primitive positions. For example, the primitive position "-B--0D56-8-7-F---" is represented by 8 equivalent states "-B--0D56-8-7-F--1", "-B--0D56-8-7-F--2", etc. where the last

character is chosen according to which pieces are not on the board $\{$0x1, 0x2, 0x3, 0x4, 0x9, 0xA, 0xC, 0xE$\}$. When counting, this position must be counted as a single position, rather than 8 different ones.

3.3 Tier Definition

We introduce two tier definitions.

The first tier definition D_L groups two positions in the same tier if in both positions, the same number of pieces are on the board. There are 17 different tiers, $L_0, L_1, L_2, ..., L_{16}$ that result from this tier definition. From now on we will call this type of tier (a tier according to D_L) a **level**. Level n or L_n is the set containing all positions with n pieces on the board. Remember that we are ignoring the initial position, so it does not belong to L_0. When a move is made from a position in L_n, the resulting position is in L_{n+1}. Note that it is the second player's turn at all positions in L_n for even n and the first player's turn at all positions in L_n for odd n.

A more granular tier definition D_T groups two positions in the same tier if in both positions, the same set of pieces exists on the board, the same slots are occupied, and they have the same piece-to-place. The only difference among positions in the same tier under this tier definition is that the pieces placed are permuted to the occupied slots of the board differently. For the remainder of this chapter, "tier" refers to a tier according to this tier definition D_T. Splitting the game into tiers according to D_T is useful because it lends itself to an efficient implementation of WORM (write once, read many) tier-symmetry removal, which is explained in 3.7.

Suppose we have a particular tier T.

- T's `OCCUPIED_SLOTS` (`OCCUPIED_SLOTS`$_T$) is a subset of $\{0, 1, ..., 15\}$ indicating which slots of the grid are occupied by a piece. `OCCUPIED_SLOTS`$_T$ can be represented as a 16-bit integer. If slot s contains a piece, then the s-th bit is set. For example, if `OCCUPIED_SLOTS`$_T = \{1, 2, 3, 5, 8, 10, 15\}$ then it can be represented as `0b1000010100101110` = `0x852E`.

- T's `PIECES_PLACED` (`PIECES_PLACED`$_T$) is a subset of $\{$`0x0`, `0x1`, ..., `0xF`$\}$ indicating which pieces are already on the board. Necessarily, $|$`PIECES_PLACED`$_T| = |$`OCCUPIED_SLOTS`$_T|$. `PIECES_PLACED`$_T$ can also be represented as a 16-bit integer. If piece u has been placed, then the u-th bit is set. For example, if `PIECES_PLACED`$_T$ = $\{$`0x0`, `0x4`, `0x6`, `0x7`, `0x9`, `0xC`, `0xD`$\}$ then it can be represented as `0b0011001011010001` = `0x32B1`.

- T's `PIECE_TO_PLACE` (`PIECE_TO_PLACE`$_T$) is a piece such that it is not in `PIECES_PLACED`$_T$. That means every position in T has a piece-to-place, including primitive positions.

Regarding notation: From now on we interchangeably refer to `OCCUPIED_SLOTS` and `PIECES_PLACED` by their set and their integer representations.

Figure 3.5: Different Boards of Positions in the Same Tier

Shown in Figure 3.5 are the boards of different positions belonging to the same tier $T \subset L_9$, where T is such that `OCCUPIED_SLOTS`$_T = \{0, 2, 5, 7, 8, 10, 11, 13, 14\}$ = `0b0110110110100101` and `PIECES_PLACED`$_T$ = $\{$`0x0`, `0x1`, `0x3`, `0x6`, `0x7`, `0x8`, `0x9`, `0xC`, `0xE`$\}$ = `0b0101001111001011`.

3.3.1 Upper-Bounding the Number of Positions, Ignoring Symmetry

For a given level n, we are interested in the number of possible values of OCCUPIED_SLOTS, PIECES_PLACED, and PIECE_TO_PLACE among all the tiers contained in L_n. There are $\binom{16}{n}$ possible values of PIECES_PLACED and $\binom{16}{n}$ possible values of OCCUPIED_SLOTS because there are $\binom{16}{n}$ 16-bit integers with n set bits. If $n < 16$, then there are $16 - n$ possible values of PIECE_TO_PLACE since there are $16 - n$ pieces that are not on the board, so L_n is a union of $\binom{16}{n}\binom{16}{n}(16-n)$ tiers. In L_{16} there is only one possible value of OCCUPIED_SLOTS (all slots are occupied), of PIECES_PLACED (all pieces are placed), and of PIECE_TO_PLACE (a null piece-to-place) so it consists of only one tier.

A tier in level $n < 16$ contains $n!$ positions because there are $n!$ ways to arrange the pieces placed among the occupied slots, so L_n contains $\binom{16}{n}\binom{16}{n}(16-n)n!$ positions. L_{16} contains 16! positions, i.e., ways to arrange the 16 pieces among all slots.

Level	# of Possible PIECES_PLACED	# of Possible PIECE_TO_PLACE	# of Possible OCCUPIED_SLOTS	# of Tiers	# of Positions
0	1	16	1	16	16
1	16	15	16	3,840	3,840
2	120	14	120	201,600	403,200
3	560	13	560	4,076,800	24,460,800
4	1,820	12	1,820	39,748,800	953,971,200
5	4,368	11	4,368	209,873,664	25,184,839,680
6	8,008	10	8,008	641,280,640	461,722,060,800
7	11,440	9	11,440	1,177,862,400	5,936,426,496,000
8	12,870	8	12,870	1,325,095,200	53,427,838,464,000
9	11,440	7	11,440	916,115,200	332,439,883,776,000
10	8,008	6	8,008	384,768,384	1,396,247,511,859,200
11	4,368	5	4,368	95,397,120	3,807,947,759,616,000
12	1,820	4	1,820	13,249,600	6,346,579,599,360,000
13	560	3	560	940,800	5,858,381,168,640,000
14	120	2	120	28,800	2,510,734,786,560,000
15	16	1	16	256	334,764,638,208,000
16	1	1	1	1	20,922,789,888,000
			Total	4,808,643,121	20,667,870,288,606,736

Table 3.1: Upper-Bounding the Number of Tiers and Positions in Each Level

In the Symmetries section below we ignore most positions that are equivalent under symmetry so that we need not explore 20.67 quadrillion positions, which would require 55 bits to address.

3.4 Non-string Position Representations

In addition to the string representation, we use two other position representations: the tier-and-bitboard representation and the tier-and-tierposition representation. The bitboard and tierposition are each used to identify a particular position in a given tier.

In the tier-and-bitboard representation, a position is defined by the tier it belongs to, and a 64-bit integer which we refer to as a **bitboard**. The bitboard is best explained as follows: if piece $b_3b_2b_1b_0$ is at slot s, then the s-th least significant nibble in the integer is set to $b_3b_2b_1b_0$. For example, if piece 0xA is at slot 7, then the 28th to 31st bits are 1010. If slot s is blank, then the s-th least significant nibble is 0. Note that the 0x0 piece and a blank have the same nibble encoding in the bitboard. The tier's OCCUPIED_SLOTS disambiguates these two slot states. The tier-and-bitboard representation is used to identify win and tie quartets for the primitive/unreachable check.

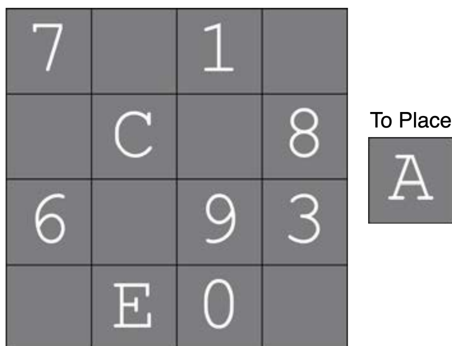

Figure 3.6: Example Position

Example: Consider the position shown in Figure 3.6. The bitboard is `0x00E0390680C00107`. Let T be the tier that this position belongs to. Given that $\texttt{OCCUPIED_SLOTS}_T = \{0, 2, 5, 7, 8, 10, 11, 13, 14\}$, we know that slot 14 is occupied, so the highlighted 0 in the bitboard represents the `0x0` piece and all other 0s in the bitboard represent blanks.

In the tier-and-tierposition representation, a position in level n is defined by the tier it belongs to and a **tierposition**, an integer $\rho \in \{0, 1, ..., n!\}$ such that ρ is a perfect hash value of a permutation of the `PIECES_PLACED` among the `OCCUPIED_SLOTS`. The tier-and-tierposition representation is used as a key for a position query of the hash-indexed database. In the next section, we discuss how to calculate the hash/tierposition of a position.

3.5 Hashing

Conversions between bitboards and tierpositions occur frequently during our solve. For every tier in each level n we aim to find a bijection between each possible bitboard in the tier and each tierposition $\rho \in \{0, 1, ..., n!\}$. We refer to the bitboard-to-tierposition conversion as *hashing* and the tierposition-to-bitboard conversion as *unhashing*.

We will first demonstrate the hashing algorithm by an example, then introduce formalization of the algorithm and optimizations, then introduce a second example.

3.5.1 Hashing/Unhashing a Level 16 Position

Let T_{16} be the single tier in level 16. We demonstrate the hashing algorithm first on a position $p_1 \in T_{16}$. Remember that $\texttt{OCCUPIED_SLOTS}_{T_{16}} = \texttt{PIECES_PLACED}_{T_{16}} = \texttt{0xFFFF}$.
Suppose this p_1's bitboard is `0xA5D731864B290EFC`. We look through each piece on the bitboard from right to left.

- The first piece is `0xC`. Of the pieces we have encountered so far, $\{\texttt{0xC}\}$, this is the 0th lowest-valued piece. $c_0 = 0$.
- Next is `0xF`. Of the pieces we have encountered so far, $\{\texttt{0xC}, \texttt{0xF}\}$, this is the 1st lowest valued-piece. $c_1 = 1$.
- Next is `0xE`. We encountered $\{\texttt{0xC}, \texttt{0xE}, \texttt{0xF}\}$ thus far and `0xE` is the 1st lowest-valued piece. $c_2 = 1$.
- Next is `0x0`. This is the 0th lowest-valued piece in $\{\texttt{0x0}, \texttt{0xC}, \texttt{0xE}, \texttt{0xF}\}$. $c_3 = 0$.
- We find the values of $c_3, c_4, ..., c_{13}$ are $1, 1, 3, 2, 3, 4, 1, 3, 6$, and 11, respectively.
- `0x5` is the 5th lowest-valued piece in $\{\texttt{0x0}, \texttt{0x1}, \texttt{0x2}, \texttt{0x3}, \texttt{0x4}, \texttt{0x5}, \texttt{0x6}, \texttt{0x7}, \texttt{0x8}, \texttt{0x9}, \texttt{0xB}, \texttt{0xC}, \texttt{0xD}, \texttt{0xE}, \texttt{0xF}\}$. $c_{14} = 5$.

- `0xA` is the 10th lowest-valued piece in $\{\texttt{0x0}, \texttt{0x1}, ..., \texttt{0xF}\}$. $c_{15} = 10$.

The tierposition is $\rho_1 = \sum_{i=0}^{n-1} c_i \cdot i! = 13584131338467$.

Now we demonstrate the unhashing algorithm. Notice that in the hashing algorithm, at iteration i, c_i is some value in $\{0, 1, ..., i\}$. Given a tierposition ρ_1 corresponding to some position in T_{16}, there is a unique assignment of values to $c_0, c_1, ..., c_{15}$ such that $\sum_{i=0}^{n-1} c_i \cdot i! = \rho_1$. From ρ_1, first we must determine $c_{15}, c_{14}, ..., c_0$ in that order which helps us identify the pieces. Suppose, in the context of the same tier T_{16}, we are given tierposition 13584131338467 and we need to convert it to a bitboard.

- Let set $R = \{\texttt{0x0}, \texttt{0x1}, ..., \texttt{0xF}\}$, i.e., R is equal to $\texttt{PIECES_PLACED}_{T_{16}}$.
- $c_{15} = 10$ because 10 is the highest value c such that $15!c \leq 13584131338467$. $13584131338467 - 15!c_{15} = 507387658467$. The 10th piece in R is `0xA`, so the bitboard is updated to `0xA000000000000000`. Remove `0xA` from R.
- $c_{14} = 5$ because 5 is the highest value c such that $14!c \leq 507387658467$. $507387658467 - 14!c_{14} = 71496202467$. The 5th piece in R is `0x5`, so the bitboard is updated to `0xA500000000000000`. Remove `0x5` from R.
- $c_{13} = 11$ because 11 is the highest value c such that $13!c \leq 71496202467$. $71496202467 - 13!c_{13} = 2998973667$. The 11th piece in R is `0xD`, so bitboard is updated to `0xA5D0000000000000`. Remove `0xD` from R.
- Essentially these are the steps in the hashing algorithm in reverse. We find that $c_{12}, c_{11}, ..., c_0$ are the same as before. Necessarily, $c_0 = 0$. Once all steps have been completed, we obtain the same bitboard, `0xA5D731864B290EFC`.

Thus ends the example for bitboard `0xA5D731864B290EFC`. Note that in the T_{16} context, using this algorithm, the hash of bitboard `0x0123456789ABCDEF` is 0 and the hash of bitboard `0xFEDCBA9876543210` is $16! - 1$.

3.5.2 Optimizations

We take note of repeated expensive operations during hashing/unhashing and how much space is required to memoize them.

- At each iteration during hashing, we are given a set of pieces Q and one piece from Q. That piece is the k-th lowest-valued piece in Q and we must determine k. This is equivalent to the problem of being given a 16-bit number B with each j-th bit set if $j \in Q$, and we must determine the index i of the k-th set bit. One way we can achieve this is to read each bit and count, but we would need to do this for each iteration of hashing. The solution is to precompute all such computations and store them in a table. The key is $(B,\ k)$ and the value is i. k ranges from 0 to 15 so it can be represented as a nibble, so the key $(B,\ k)$ is 20 bits. Thus there are 2^{20} different keys (not all are used). i can also be represented as a nibble but for ease of implementation we represent it as a byte. The size of this entire table is thus $2^{20}\text{B} = 1\text{MiB}$.
- At each iteration during unhashing, we are given a set of pieces Q and k, and we must determine which piece from Q is the k-th lowest value piece out of all pieces in Q. This is equivalent to the problem of being given a 16-bit number B with each j-th bit set if $j \in Q$, and we must determine which k-th set bit the i-th bit of B is. Again, we can use a table. The key is $(B,\ i)$ and the value is k. i ranges from 0 to 15 so it is a 4-bit number, so the key $(B,\ i)$ is 20 bits, so there are 2^{20} different keys (not all are used. For ease of implementation, the value k is a byte even if it can be represented as a nibble. The size of this entire table is also 1MiB.
- We can also precompute factorial-multiplied-by-constant calculations. The key is (c, m) and the value is $c \cdot m!$. Only values of c and m ranging from 0 to 15 are used during hashing/unhashing. Each value is 8 bytes, so the space required is $16 \cdot 16 \cdot 8\text{B} = 2048\text{B}$.

We can speed up our hashing and unhashing by initializing all of these tables beforehand, which have a total memory requirement of 2.002MiB.

3.5.3 Generalizing the Algorithm to a Position Outside of L_{16}

We introduced an example in which the board was full but we would like to generalize the algorithm to work with boards that have blank slots. For hashing and unhashing positions outside of L_{16}, we skip over any blanks in the bitboard and treat each piece value according to the piece's ordering value in `PIECES_PLACED`. For example, if `PIECES_PLACED` $= \{$`0x3`$,$ `0x5`$,$ `0x9`$,$ `0x15`$\}$, then these piece's values would be treated as $0, 1, 2,$ and 3, respectively.

Suppose we are to hash the bitboard of position $p_2 \in T \subset L_6$, where $\texttt{OCCUPIED_SLOTS}_T = \{0, 5, 8, 10, 13, 14\}$ and
$\texttt{PIECES_PLACED}_T = \{\texttt{0x0}, \texttt{0x3}, \texttt{0x7}, \texttt{0x8}, \texttt{0xB}, \texttt{0xE}\}$. Suppose p_2's bitboard is `0x80000E00700003B0`. The highlighted 0 nibble is the zero piece (as given by $\texttt{OCCUPIED_SLOTS}_T$) and all other 0s represent blank spaces. We look through each piece on the bitboard from right to left.

- Initialize `encounteredPieces` = `0b0000000000000000`.
- The piece occupying slot 0 is `0x8`. Set the 8th bit in `encounteredPieces` – `encounteredPieces` := `0b0000000100000000`. The 8th bit in `encounteredPieces` is the 0th set bit in `encounteredPieces`. $c_0 = 0$.
- Skip to the next occupied slot, which is slot 5. Slot 5 is is occupied by `0xE`. Set the 14th bit in `encounteredPieces` – `encounteredPieces` := `0b0100000100000000`. The 14th bit in `encounteredPieces` is the 1st set bit in `encounteredPieces`. $c_1 = 1$.
- The next piece is `0x7`. `encounteredPieces` := `0b0100000110000000`. The 7th bit in `encounteredPieces` is the 0th set in `encounteredPieces`. $c_2 = 0$.
- The next piece is `0x0`. `encounteredPieces` := `0b0100000110000001`. $c_3 = 0$.
- The next piece is `0x3`. `encounteredPieces` := `0b0100000110001001`. $c_4 = 1$.
- The final piece is `0xB`. `encounteredPieces` := `0b0100100110001001`. $c_5 = 4$.

The hash of the p_2's bitboard is $\rho_2 = \sum_0^5 c_i \cdot i!$. For more details on the exact implementation of hashing and unhashing, see Quarto Hash and Unhash in Appendix B.

3.6 Primitive/Unreachable Check

A position is unreachable if there exist two win quartets that cannot be simultaneously completed in a single move (i.e., the two win quartets cannot coexist). For example, a board containing at least two row win quartets, at least two column win quartets, or two diagonal win quartets is unreachable. A board state is also unreachable if it has exactly one row, one column, and one diagonal win quartet such that there does not exist a single slot covered by the three quartets.

Win quartets can coexist if all win quartets in question can be simultaneously completed in a single move. If all win quartets in a position can coexist, then the position is reachable. If a reachable position has a win quartet, then the position is a primitive lose (the player whose turn it is after their opponent has created a win quartet has lost); otherwise, if the board is filled, then the position is a primitive tie.

For the primitive/unreachable check, we wish to determine whether a given four-in-a-row of pieces forms a win quartet. The pieces form a win quartet if there is a matching bit among all four pieces.

In order to check whether there exists a matching bit, we first bitwise-AND all four pieces. If the result is nonzero, then there exists a matching set bit. We then flip the (four least significant) bits of each piece and bitwise-AND the the inverted pieces. If the result is nonzero, then there exists a matching unset bit.

We can modify this check to work with any four-in-a-row that might include blanks. We represent a blank as a `0` in both AND checks (we do not invert the blank piece in the common-unset-bit check). If either AND results in a nonzreo value then the four-in-a-row is a win quartet. Otherwise, it either is a tie quartet or an incomplete quartet.

The primitive check uses the bitboard representation of the position. For each tier it is helpful to keep a 64-bit integer which is each bit of `OCCUPIED_SLOTS` repeated four times (e.g., if `OCCUPIED_SLOTS` = `0b1011101110101001` then this integer is `0xF0FFF0FFF0F0F00F`). We refer to this number as the *occupied slots mask*. We can then XOR the bitboard with the occupied slots mask to obtain an inverted bitboard. This inverts actual piece nibbles and leaves nibbles representing blanks unchanged.

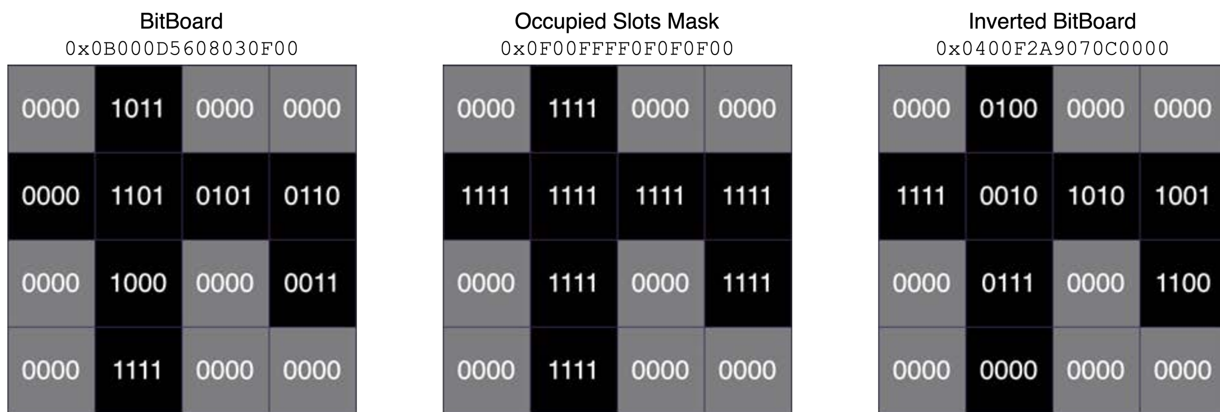

Figure 3.7: Inverting a BitBoard

Shown above is an example of a position (1) with bitboard `0x0B000D5608030F00` and (2) belonging to a tier such that the occupied slots mask is `0x0F00FFFF0F0F0F00`. For the slot in the second row, first column, notice that the corresponding nibble in the bitboard is `0x0` but the corresponding nibble in the occupied slots mask is `0xF` indicating that the slot contains the `0x0` piece and is not empty.

To check whether a position is primitive or unreachable, at most ten four-in-a-row checks take place (since there are a total of ten sets of four slots that create a four in a row). For a particular four-in-a-row check we select the appropriate nibbles from the bitboard and inverted bitboard.

We can perform all four column quartet checks in parallel by doing the following...

```
invBitBoard = bitBoard ^ OCCUPIED_SLOTS_MASK
bitBoardAND = bitBoard & (bitBoard >> 16) & (bitBoard >> 32) & (bitBoard >> 48)
invBitBoardAND = invBitBoard & (invBitBoard >> 16) & (invBitBoard >> 32)
                 & (invBitBoard >> 48)
```

Where "`^`" is a bitwise XOR, "`&`" is a bitwise AND, and "`>> x`" is a right shift by `x` bits. If the integer represented by the lower 16 bits of `bitBoardAND` is nonzero or the integer represented by the lower 16 bits of `invBitBoardAND` is nonzero, then there exists a column win quartet.

We can similarly perform all four row quartet checks in parallel by doing the following...

```
invBitBoard = bitBoard ^ OCCUPIED_SLOTS_MASK
bitBoardAND = bitBoard & (bitBoard >> 4) & (bitBoard >> 8) & (bitBoard >> 12)
invBitBoardAND = invBitBoard & (invBitBoard >> 4) & (invBitBoard >> 8)
                 & (invBitBoard >> 12)
```

If the any of the 0th, 4th, 8th, or 12th least-significant nibbles of `bitBoardAND` is nonzero or any of the 0th, 4th, 8th, or 12th least-significant nibbles of `invBitBoardAND` is nonzero, then there exists a row win quartet.

The main diagonal quartet check can be similarly done by shifting by 20, 40, and 60. The antidiagonal quartet check can be done by shifting by 12, 24, and 36.

It is sufficient to check whether AND results are nonzero if we can assume that the given position is reachable. If we are not given that information and we must determine ourselves whether the position is reachable, then we must not only determine whether each AND result is nonzero but also parse each AND result to determine which rows, columns, and diagonals are win quartets in order to check for incompatible quartets.

bitBoard			
0000	1011	0000	0000
0000	1101	0101	0110
0000	1000	0000	0011
0000	1111	0000	0000

bitBoard >> 16			
0000	1011	0101	0110
0000	1000	0000	0011
0000	1111	0000	0000
0000	0000	0000	0000

bitBoard >> 32			
0000	1000	0000	0011
0000	1111	0000	0000
0000	0000	0000	0000
0000	0000	0000	0000

bitBoard >> 48			
0000	1111	0000	0000
0000	0000	0000	0000
0000	0000	0000	0000
0000	0000	0000	0000

AND Result			
0000	1000	0000	0000
0000	0000	0000	0000
0000	0000	0000	0000
0000	0000	0000	0000

Figure 3.8: Example Column Quartet Checks. The `1000` in the second column of the AND result indicates that the second column contained pieces that all shared the highest bit (i.e., were all white).

3.7 Symmetries

Symmetric positions of a particular position p can be found through board slot and piece bit transformations.

In this section, we frequently discuss permutations. A permutation σ of a set is a bijection from that set to itself. If σ defines a permutation of a set of n elements, then σ is a member of the permutation group S_n. There are $n!$ different permutations in S_n.

Example: Let $n = 7$ and suppose $\sigma : M \to M$ (a member of S_7) is a permutation of $M = \{0, 1, 2, 3, 4, 5, 6\}$. Suppose $\sigma(0) = 1, \sigma(1) = 3, \sigma(3) = 0, \sigma(4) = 6, \sigma(6) = 4, \sigma(2) = 2$, and $\sigma(5) = 5$. Then σ can be expressed using the zero-indexed notation $\sigma = (0\ 1\ 3)(4\ 6)(2)(5)$ or, equivalently, $\sigma = (0\ 1\ 3)(4\ 6)$. If σ maps each element to itself then $\sigma = ()$.

3.7.1 Board Slot Symmetries

$G_{\text{BS}} = \langle r, f, i, o \rangle$ is the board slot symmetry transformation group. G_{BS} is a subgroup of the permutation group S_{16} and contains transformations defining how to permute the slots of a board to create a symmetric board. The generators r, f, i, and o are...

1. Rotate 90° Clockwise: $r = (0\ 3\ 15\ 12)(1\ 7\ 14\ 8)(2\ 11\ 13\ 4)(5\ 6\ 10\ 9)$.
2. Transpose: $f = (1\ 4)(2\ 8)(3\ 12)(6\ 9)(7\ 13)(11\ 14)$.
3. Inner Swap: $i = (1\ 2)(4\ 8)(5\ 10)(6\ 9)(7\ 11)(13\ 14)$.
4. Outer Swap: $o = (0\ 5)(1\ 4)(2\ 7)(3\ 6)(8\ 13)(9\ 12)(10\ 15)(11\ 14)$.

There are $|G_{\text{BS}}| = 32$ different permutations, i.e., there are 32 unique board slot symmetry transformations. When applying r α times, then applying f β times, then applying i γ times, then applying o δ times, different choices of $\alpha \in \{0, 1, 2, 3\}$ and $\beta, \gamma, \delta \in \{0, 1\}$ yield unique transformations. Following are explanations of how to apply transformation $\sigma \in G_{\text{BS}}$ to various objects:

- σ applied to a board b is the board b', formed by permuting the slots of board b according to σ. For each slot s of b, the piece (or blank) occupying s occupies slot $s' = \sigma(s)$ of board b'. Notation, $\sigma \cdot b = b'$.
- Let p be a position with board b. σ applied to p is the position p' which has the same piece-to-place as p but has $\sigma \cdot b$ as its board. Notation: $\sigma \cdot p = p'$.
- σ applied to a 16-bit integer N (e.g., an `OCCUPIED_SLOTS` value) results in the 16-bit integer N' formed by permuting the bits of N according to σ. For each $s \in \{0, 1, ..., 15\}$, if bit s is set (or unset) in N, then bit $s' = \sigma(s)$ is set (or unset) in N'. Notation: $\sigma \cdot N = N'$.
- σ applied to a tier T results in the tier T' such that $\texttt{PIECE_TO_PLACE}_{T'} = \texttt{PIECE_TO_PLACE}_{T}$ and $\texttt{PIECES_PLACED}_{T'} = \texttt{PIECES_PLACED}_{T}$, but $\texttt{OCCUPIED_SLOTS}_{T'} = \sigma \cdot \texttt{OCCUPIED_SLOTS}_{T}$. Notation: $\sigma \cdot T = T'$.

On the board, there are ten ways a particular quartet can be arranged. A quartet can exist across one of the ten four-in-a-rows of slots: the rows $\{0, 1, 2, 3\}$, $\{4, 5, 6, 7\}$, $\{8, 9, 10, 11\}$, and $\{12, 13, 14, 15\}$; the columns $\{0, 4, 8, 12\}$, $\{1, 5, 9, 13\}$, $\{2, 6, 10, 14\}$ and $\{3, 7, 11, 15\}$; and the diagonals $\{0, 5, 10, 15\}$ and $\{3, 6, 9, 12\}$. Each of the 32 board slot symmetry transformations σ when applied to a position results in a symmetric position because it **preserves** all ten four-in-a-rows – if s_0, s_1, s_2, and s_3 exist in a four-in-a-row on board b, then $\sigma(s_0)$, $\sigma(s_1)$, $\sigma(s_2)$, and $\sigma(s_3)$ also exist in a four-in-a-row (in no particular order).

Shown in Figure 3.9 are the transformations r, f, i, o each applied to a board b. All ten four-in-a-rows are preserved under these transformations and under any composition of them.

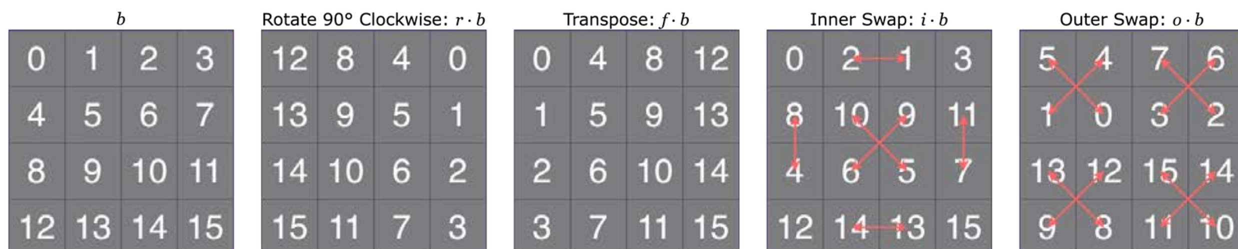

Figure 3.9: Board Symmetry Transformation Group Generators

3.7.2 Piece Bit Symmetries

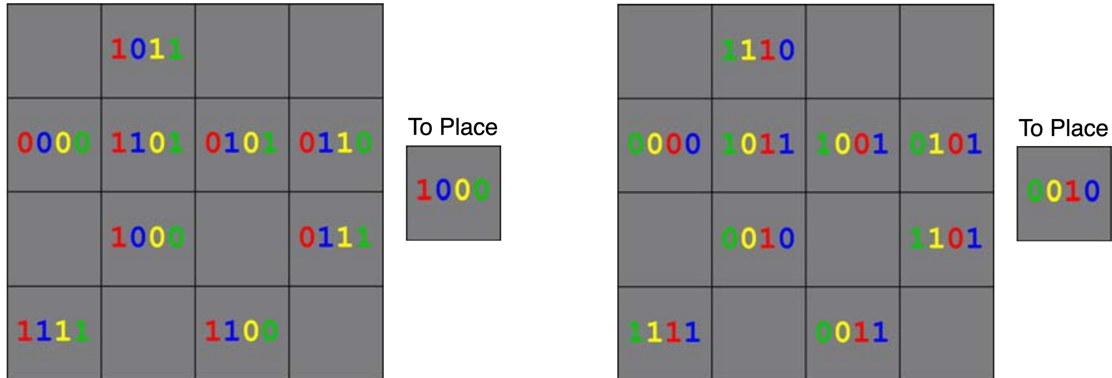

Figure 3.10: Piece Bit Permutation Symmetry. In this example, the bits are permuted according to (0 3 1 2), resulting in a symmetric position.

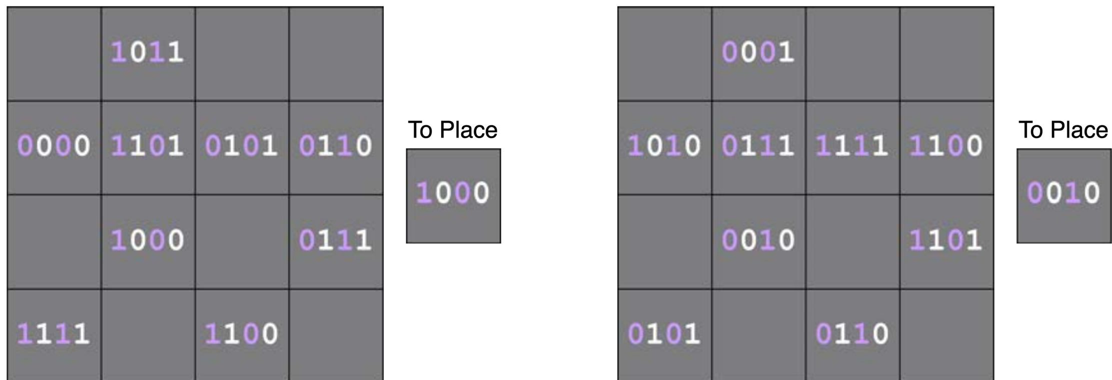

Figure 3.11: Piece Bit XOR Symmetry. In this example, we flip the 1st and 3rd bits of each piece, i.e., we bitwise XOR each piece with `0b1010` to obtain a symmetric position.

For any position p, we can permute the bits of each piece on p's board and of the piece-to-place to obtain a symmetric position. There are 24 ways to permute the bits. We can also choose a subset of the four bits and flip those bits on the board pieces and piece-to-place. In total there are 16 ways to choose a subset of bits to flip. Flipping some subset of the piece bits is equivalent to performing a bitwise XOR on each piece by a number from 0 to 15. When we bitwise XOR a piece and a number $k \in \{0, 1, ...15\}$, then the set bits of k indicate which piece bits to flip and the unset bits of k indicate which piece bits to leave unchanged.

Let the binary operator $\oplus$ denote bitwise XOR. Suppose we select any two pieces u_1 and u_2 that have z matching bits. Then $u_1' = (u_1 \oplus k)$ and $u_2' = (u_2 \oplus k)$ $(k \in \{0, 1, ..., 15\})$ have z matching bits as well because flipping the same bits in both pieces does not change the number of matching bits. Likewise, if we permute the bits of u_1 and u_2 then the number of matching bits between them also remains unchanged. These piece bit transformations yield symmetric positions because they preserve matching bits.

The XOR piece bit symmetry transformation group is $G_\oplus = (\{\oplus_0, \oplus_1, ..., \oplus_{15}\}, \circ)$, where $\circ$ is a binary operator with the following definition: $\oplus_a \circ \oplus_b = \oplus_{a \oplus b}$. One can easily verify that the group axioms are satisfied. The piece bit permutation transformation group is S_4.

$G_{\text{PB}} = S_4 \times G_\oplus$ is the piece bit symmetry transformation group. In total there are $|S_4| \cdot |G_\oplus| = 24 \cdot 16 = 384$ total piece bit symmetry transformations. Each member of this group is of the form $\tau = (\pi, \oplus_k)$ where $\pi \in S_4$ and $\oplus_k \in G_\oplus$. Following are explanations of how to apply transformation $\tau \in G_{\text{PB}}$ to various objects:

- The result of applying transformation $\tau = (\pi, \oplus_k)$ to a piece u (a 4-bit integer) is the piece u' that is the result of permuting the bits of $(u \oplus k)$ according to π. Notation: $\tau \cdot u = u'$. For example, $((1\ 2\ 4), \oplus_{11}) \cdot \texttt{0x9} = ((1\ 2\ 4), \oplus_{\texttt{0b1011}}) \cdot \texttt{0b1001} = (1\ 2\ 4) \cdot (\texttt{0b1011} \oplus \texttt{0b1001}) = (1\ 2\ 4) \cdot \texttt{0010} = \texttt{0b1010} = \texttt{0xA}$.

- Applying τ to position p gives the position p' formed by applying τ to each piece on p's board and to p's piece-to-place. Notation: $\tau \cdot p = p'$.

- τ applied to a 16-bit integer N (e.g., a `PIECES_PLACED` value) results in the 16-bit integer N' formed by permuting the bits of N according to τ. Specifically, treat each $u \in \{0, ..., 15\}$ as a piece or 4-bit integer – if bit u is set (or unset) in N, then bit $u' = \tau \cdot u$ is set (or unset) in N'. Notation: $\tau \cdot N = N'$.

- Applying τ to a tier T gives the tier T' such that $\texttt{OCCUPIED_SLOTS}_{T'} = \texttt{OCCUPIED_SLOTS}_T$ but $\texttt{PIECE_TO_PLACE}_{T'} = \tau \cdot \texttt{PIECE_TO_PLACE}_T$ and $\texttt{PIECES_PLACED}_{T'} = \tau \cdot \texttt{PIECES_PLACED}_T$. Notation: $\tau \cdot T = T'$.

Applying $\oplus_k \in G_\oplus$ alone to any of these objects is the same as applying $\tau = ((), \oplus_k)$. Applying $\pi \in S_4$ alone to any of these objects is the same as applying $\tau = (\pi, \oplus_0)$.

3.7.3 Canonicalization and Tier Symmetries

Suppose a group G acts on [14] a set of objects W. For a particular object $y \in Y$, the orbit of y (notated $G \cdot y$) [15] is the set of all elements in Y that result from any transformation $g \in G$. The **canonical** object of y is an object $y^{\text{C}} \in G \cdot y$ that is a "representative" of all elements in the orbit of y including y itself. When G is a symmetry transformation group, an orbit consists of objects that are "symmetrical" to each other. For any information requested about an object in an orbit, we refer to the canonical object in that orbit. For example, instead of storing information about all tiers in a set of tiers that are symmetric to each other, we only store information about the canonical tier in a file and any time we request any information about a non-canonical tier we read data from the canonical tier file.

The choice of which object in an orbit is canonical is completely arbitrary. Below we outline how we decide canonical `OCCUPIED_SLOTS` values, canonical (`PIECE_TO_PLACE`, `PIECES_PLACED`) tuples, and canonical tiers.

Suppose G_{BS} acts on the set of all possible `OCCUPIED_SLOTS` values. Let N_{OS} be some `OCCUPIED_SLOTS` value. Then the canonical `OCCUPIED_SLOTS` value of N_{OS} is N_{OS}^{C}, the *smallest integer* in $G_{\text{BS}} \cdot N_{\text{OS}}$.

Suppose G_{PB} acts on the set of all possible (`PIECE_TO_PLACE`, `PIECES_PLACED`) tuples. Let (u, N_{PP}) be some (`PIECE_TO_PLACE`, `PIECES_PLACED`) tuple. Then the canonical tuple is $(\texttt{0x0}, N_{\text{PP}}^{\text{C}})$, where N_{PP}^{C} is the smallest integer in $S_4 \cdot (\oplus_u \cdot N_{\text{PP}})$. In other words, given a non-canonicalized (u, N_{PP}), first we XOR by u to arrive at a tuple $(\texttt{0x0}, N'_{\text{PP}})$. Then apply each piece bit permutation symmetry transformation to N'_{PP} and find the smallest integer N_{PP}^{C} among the results of those transformations.

Now we consider the combination of board slot and piece bit symmetries. The overall symmetry transformation group is $G_{\text{Q}} = G_{\text{PB}} \times G_{\text{BS}}$. The total number of symmetry transformations is $|G_{\text{Q}}| = |G_{\text{PB}}| \cdot |G_{\text{BS}}| = 12288$.

Suppose G_{Q} acts on the set of all tiers. Suppose a non-canonicalized tier T is such that $\texttt{OCCUPIED_SLOTS}_T = N_{\text{OS}}$, $\texttt{PIECES_PLACED}_T = N_{\text{PP}}$, and $\texttt{PIECE_TO_PLACED} = u$. Then the canonical tier of T is the tier T^{C} such that $\texttt{OCCUPIED_SLOTS}_{T^{\text{C}}} = N_{\text{OS}}^{\text{C}}$, $\texttt{PIECES_PLACED}_{T^{\text{C}}} = N_{\text{PP}}^{\text{C}}$, and $\texttt{PIECE_TO_PLACE}_{T^{\text{C}}} = \texttt{0x0}$. Indeed, every tier is symmetric to a tier with a `PIECE_TO_PLACE` of `0x0` because we can XOR each piece with the piece-to-place.

3.7.4 Symmetry Tables

For smaller games, typically one can apply every symmetry transformation to an object and then look through every object in its orbit to find the canonical one. For larger games like Quarto, it is helpful to use symmetry tables so that one can look up an object to determine its canonical object, without the need to "try" each symmetry transformation and see which transformation when applied to the object results in the canonical one. Symmetry tables allow us to identify the canonical tier T^{C} of a given tier T and give us information about the symmetry transformations that bijectively map each position in T to a position in T^{C}.

We use a table with which we can look up any non-canonicalized OCCUPIED_SLOTS value and obtain the canonical OCCUPIED_SLOTS value and transformation from the non-canonicalized value to the canonicalized value. Specifically, the key to the board slots symmetry table is $\texttt{OCCUPIED_SLOTS}_T$ and the value is $(\texttt{OCCUPIED_SLOTS}_{T^{\mathrm{C}}}, \sigma)$ where σ is board slot symmetry transformation from $\texttt{OCCUPIED_SLOTS}_T$ to $\texttt{OCCUPIED_SLOTS}_{T^{\mathrm{C}}}$. There are 2^{16} keys since there are 2^{16} possible values of $\texttt{OCCUPIED_SLOTS}_T$. There are 2^{16} possible values of $\texttt{OCCUPIED_SLOTS}_{T^{\mathrm{C}}}$ and $32 = 2^5$ possible transformations σ, so $(\texttt{OCCUPIED_SLOTS}_{T^{\mathrm{C}}}, \sigma)$ uses $16 + 5 = 21$ bits. For ease of implementation, we let $(\texttt{OCCUPIED_SLOTS}_{T^{\mathrm{C}}}, \sigma)$ take up 32 bits. Thus the total size of the board slots symmetry table is $2^{16} \cdot 32$ bits $= 256$ kiB.

We also use a table with which we can look up a PIECES_PLACED and PIECE_TO_PLACE value and obtain the canonical PIECES_PLACED value and canonicalization transformation. The key to the piece bit symmetry table is $(\texttt{PIECE_TO_PLACE}_T, \texttt{PIECES_PLACED}_T)$ and the value is $(\texttt{PIECES_PLACED}_{T^{\mathrm{C}}}, \pi_{PB})$, where π_{PB} is the piece bit permutation from $(\oplus_{\texttt{PIECE_TO_PLACE}_T} \cdot \texttt{PIECES_PLACED}_T)$ to $\texttt{PIECES_PLACED}_{T^{\mathrm{C}}}$. There are 16 possible values of $\texttt{PIECE_TO_PLACE}_T$ and 2^{16} possible values of $\texttt{PIECES_PLACED}_T$, so there are 2^{20} possible keys. There are 2^{16} possible values of $\texttt{PIECES_PLACED}_{T^{\mathrm{C}}}$ and 24 possible values of π_{PB}, so the value $(\texttt{PIECES_PLACED}_{T^{\mathrm{C}}}, \pi_{PB})$ takes up $16 + \lceil \log_2 24 \rceil = 21$ bits. For ease of implementation we let $(\texttt{PIECES_PLACED}_{T^{\mathrm{C}}}, \pi_{PB})$ take up 32 bits. The total size of the piece bit symmetry table is $2^{20} \cdot 32$ bits $= 4$ MiB.

We can speed up our symmetry calculations by initializing these symmetry tables beforehand, which have a total memory requirement of 4.25 MiB.

3.7.5 Upper-Bounding the Number of Positions, Utilizing Tier Symmetry

We can read through the symmetry tables to identify and count the unique canonical OCCUPIED_SLOTS and canonical PIECES_PLACED values. We can then construct canonical tiers – all the canonical tiers are constructed by using a tier that has every possible combination of a canonical OCCUPIED_SLOTS and a canonical PIECES_PLACED such that $|\texttt{OCCUPIED_SLOTS}| = |\texttt{PIECES_PLACED}|$. The number of canonical tiers in level n would be the number of canonical OCCUPIED_SLOTS at L_n multiplied by the number of canonical PIECES_PLACED at L_n. The canonical tier counts from the symmetry tables are presented in Table 3.2.

Level	# of Canonical PIECES_PLACED	# of Canonical OCCUPIED_SLOTS	# of Canonical Tiers	# of Positions Per Tier	# of Positions
0	1	1	1	1	1
1	4	2	8	1	8
2	13	10	130	2	260
3	39	28	1092	6	6552
4	97	84	8148	24	195552
5	187	168	31416	120	3769920
6	290	306	88740	720	63892800
7	365	410	149650	5040	754236000
8	365	476	173740	40320	7005196800
9	290	410	118900	362880	43146432000
10	187	306	57222	3628800	207647193600
11	97	168	16296	39916800	650484172800
12	39	84	3276	479001600	1569209241600
13	13	28	364	6227020800	2266635571200
14	4	10	40	87178291200	3487131648000
15	1	2	2	1307674368000	2615348736000
16	1	1	1	20922789888000	20922789888000
Total	1993	2494	649026		31770220181093

Table 3.2: Counting the Number of Tiers and Positions Unique Under Tier Symmetry

If we limit our solving to the canonical tiers only, then tier symmetries allow us to explore fewer than 32 trillion positions in order to solve the game, which is about 0.15% of the upper bound of 20.67 quadrillion positions we calculated earlier when symmetries were not accounted for.

3.8 Encoding Value-Remoteness

`LX`, `TX`, and `WX` mean Lose, Tie, and Win in X, respectively. V_p is the value-remoteness of position p. Primitive ties cannot occur unless the board is filled, so if $p \in L_n$ is a tie, then its remoteness is $16 - n$.

3.8.1 Level Value-Remoteness Encoding Table

Prior to our solve, we know nothing about the analysis of Quarto other than the following information:

- That it can be split into levels L_n as defined before and it has level sequence $L_0 \to L_1 \to ... \to L_{16}$.
- The only value-remoteness pairs possible for a position in L_{16} are `L0` and `T0`.

For $n \in \{0, 1, ..., 15\}$, we are interested in determining a small superset U_{L_n} of the value-remoteness pairs that will ever be seen during solving for positions in L_n, given only this information. In other words, we are *anticipating* what value-remoteness pairs will be encountered among all positions in L_n.

Initialize $U_{L_{16}} = \{L0, T0\}$. Perform the following in descending order $n = 15, 14, 13, ..., 0$ to construct each U_{L_n}:

1. Initialize $U_{L_n} = \{\texttt{L0}\}$. (Disregard the fact that primitive positions are not possible in L_0, L_1, L_2, and L_3. We are constructing U_{L_n} according to limited information about the analysis of Quarto. Indeed, including `L0` in U_{L_0}, U_{L_1}, U_{L_2}, and U_{L_3} leads to a space inefficiency but it is minor, as we discuss in 3.8.3.)
2. For each anticipated value-remoteness pair v in $U_{L_{n+1}}$, presume the existence of a position p in L_n whose child positions all have values that are v at best (for the player whose turn it is at L_{n+1}). Solve this hypothetical position V_p and add its value to U_{L_n}.

This procedure ensures by construction that during our solve of a position in L_n, the position's value will be contained in U_{L_n}, because we account for the possibility that a position in L_n will be a primitive lose (no positions outside of L_{16} can be primitive ties) or that it will have any set of value-remoteness pairs among its children.

For example, suppose we find that $U_{L_{13}} = \{\texttt{L0}, \texttt{W1}, \texttt{L2}, \texttt{W3}, \texttt{T3}\}$ and we must next construct $U_{L_{12}}$. `L0` is

in $U_{L_{12}}$ because L_{12} may have primitive positions. `W1`, `L2`, `W3`, `L4`, and `T4` are each contained in $U_{L_{12}}$ because some position in L_{12} may be such that its child positions are all at best `L0`, at best `W1`, at best `L2`, at best `W3`, or at best `T3`, respectively.

We list each value-remoteness pair in U_{L_n} for all n in the Table 3.8.1 from worst to best, i.e., in ascending order according to the following rule:
Low-Remoteness Lose < High-Remoteness Lose < Tie < High-Remoteness Win < Low-Remoteness Win

Encoding	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
L_0	L0	L2	L4	L6	L8	L10	L12	L14	L16	T16	W15	W13	W11	W9	W7	W5	W3	W1
L_1	L0	L2	L4	L6	L8	L10	L12	L14	T15	W15	W13	W11	W9	W7	W5	W3	W1	
L_2	L0	L2	L4	L6	L8	L10	L12	L14	T14	W13	W11	W9	W7	W5	W3	W1		
L_3	L0	L2	L4	L6	L8	L10	L12	T13	W13	W11	W9	W7	W5	W3	W1			
L_4	L0	L2	L4	L6	L8	L10	L12	T12	W11	W9	W7	W5	W3	W1				
L_5	L0	L2	L4	L6	L8	L10	T11	W11	W9	W7	W5	W3	W1					
L_6	L0	L2	L4	L6	L8	L10	T10	W9	W7	W5	W3	W1						
L_7	L0	L2	L4	L6	L8	T9	W9	W7	W5	W3	W1							
L_8	L0	L2	L4	L6	L8	T8	W7	W5	W3	W1								
L_9	L0	L2	L4	L6	T7	W7	W5	W3	W1									
L_{10}	L0	L2	L4	L6	T6	W5	W3	W1										
L_{11}	L0	L2	L4	T5	W5	W3	W1											
L_{12}	L0	L2	L4	T4	W3	W1												
L_{13}	L0	L2	T3	W3	W1													
L_{14}	L0	L2	T2	W1														
L_{15}	L0	T1	W1															
L_{16}	L0	T0																

Table 3.3: Anticipated Value-Remoteness Pairs at Each Level

$E(L_n, V)$ is the encoding of value-remoteness pair V, which is the number assigned to V given the level L_n context. Refer to the header of Table 3.8.1. For example, $E(L_6, \texttt{L2}) = 1$, $E(L_9, \texttt{W1}) = 8$, and $E(L_0, \texttt{L12}) = 6$.

3.8.2 Determining the Value-Remoteness of a Parent Position

Suppose position $p \in L_n$ has child positions $C \subseteq L_{n+1}$. In order to determine the p's value-remoteness encoding $E(L_n, V_p)$, we use the following formula.

$$E(L_n, V_p) = \begin{cases} 0 & \text{if } p \text{ is a primitive lose } (\texttt{L0}) \\ 1 & \text{if } p \text{ is a primitive tie } (\texttt{T0}) \\ 17 - n - \min\{E(L_{n+1}, V_c) \mid c \in C\} & \text{otherwise} \end{cases}$$

One can then determine the value-remoteness of p given the encoding and the level context L_n.

In other words, if p is a primitive lose, then its value-remoteness encoding is 0 because `L0` in any level context is encoded as 0. If p is a primitive tie `T0`, then it is in L_{16}, so its encoding is 1.

If $p \in L_n$ is nonprimitive, first determine the lowest-value encoding m among all of p's child positions (the child positions' encodings are to be interpreted in the L_{n+1} context). We subtract m from the highest-value encoding for L_n, which is always $17-n$, to obtain the encoding of p's value-remoteness in the L_n context.

Example:
$p \in L_{10}$ has children $C = \{c_0, c_1, c_2, c_3\}$ and $V_{c_0} = \texttt{W5}$, $V_{c_1} = \texttt{T5}$, $V_{c_2} = \texttt{L2}$, and $V_{c_3} = \texttt{W1}$.
p is non-primitive because it has child positions. Thus,

$E(L_{10}, V_p) = 17 - 10 - \min\{E(L_{11}, c) \mid c \in C\}$

$= 7 - \min\{E(L_{11}, \texttt{W5}), E(L_{11}, \texttt{T5}), E(L_{11}, \texttt{L2}), E(L_{11}, \texttt{W1})\}$
$= 7 - \min\{4, 3, 1, 6\}$
$= 6$

Finally, 6 in context L_{10} encodes `W3` so p's value-remoteness is "Win in 3".

The way the solver determines a position's value-remoteness is described by the pseudocode below. The function `DetermineValueRemotenessEncoding` takes n and a position $p \in L_n$ as input and returns the encoding for p's value-remoteness in the L_n context.

```
DetermineValueRemotenessEncoding(level, position):
    if position is primitive: return encoding 0 (PRIMITIVE_LOSE) or 1 (PRIMITIVE_TIE)

    minChildVREncoding = 17 - level
    for each child of position:
        childVREncoding = DetermineValueRemotenessEncoding(level + 1, child)
        if childVREncoding == 0: return 17 - level
        else: minChildVREncoding = min(minChildVREncoding, childVREncoding)

    return 17 - minChildVREncoding
```

A traditional way of encoding value

3.8.3 Encoding as a Bitstring

Suppose $p \in L_n$ and the encoding of its value-remoteness is x. There are $18-n$ anticipated value-remoteness pairs for L_n, so p's value-remoteness bit representation is the $\lceil \log_2(18-n) \rceil$ least significant bits of x.

Example 1: $p \in L_{16}$ and $E(L_{16}, V_p) = 1$: $\lceil \log_2(18-16) \rceil = 1$ bit needed, so the representation is `1`.
Example 2: $p \in L_7$ and $E(L_7, V_p) = 5$: $\lceil \log_2(18-7) \rceil = 4$ bits needed, so the representation is `0101`.
Example 3: $p \in L_{12}$ and $E(L_{12}, V_p) = 5$: $\lceil \log_2(18-12) \rceil = 3$ bits needed, so the representation is `101`.

There is a bit inefficiency one might observe after the solve is complete. For the lower levels (i.e., L_n for larger n), all of the value-remoteness pairs in U_{L_n} end up being used. But in the higher levels (i.e., L_n for smaller n), not all of the value-remoteness pairs in U_{L_n} are actually used. However, any attempt to compress the higher levels even further will not reduce the overall database size by much because the number of positions in the higher levels is extremely small compared to the number of positions in the lower levels.

3.9 Database and Live-Solving

We store the results of our solve in a `database` directory with the following structure.

- `database` contains 17 level directories `00`, `01`, ..., and `16` corresponding to $L_0, L_1, ..., L_{16}$ respectively.
- Each level directory contains the results files for all tiers in that level. Example: There are 173740 canonical tiers in level 8. Thus, `08` contains 173740 tier results files: one file per tier to store value-remoteness data for each position in the tier.
- Suppose tier T is canonical. Its results are stored in the file with a name created by concatenating the last four hex digits of $\texttt{PIECES_PLACED}_T$ with the last four hex digits of $\texttt{OCCUPIED_SLOTS}_T$. For example if $\texttt{OCCUPIED_SLOTS}_T = \texttt{0x01FE}$ and $\texttt{PIECES_PLACED}_T = \texttt{0x017F}$, then $T \subset L_8$ so the results of T are stored in `database/08/01FE017F`. `PIECE_TO_PLACE` is not indicated in the tier results filename because all canonical tiers have a `PIECE_TO_PLACE` of `0x0`.

- Suppose a canonical tier results file corresponds to a tier $T \in L_n$. The file is tierposition-indexed and contains $n!$ $\lceil \log_2(18-n) \rceil$-bit entries. Example: Suppose a canonical tier results file corresponds to a tier in L_8 and one wants to read the value-remoteness of a position in that tier whose tierposition is 25312. Each value-remoteness in level 8 is encoded using $\lceil \log_2(18-8) \rceil = 4$ bits. Read 4 bits, starting at 25312×4 bits from the start of the file, in order to get the encoded value-remoteness of this position.

A position is queried with its string representation as the key. First, convert to a tier-and-bitboard representation. Next, determine the canonical tier and transformed bitboard as described in the Symmetries section. Then hash the transformed bitboard and read from the appropriate canonical tier results file.

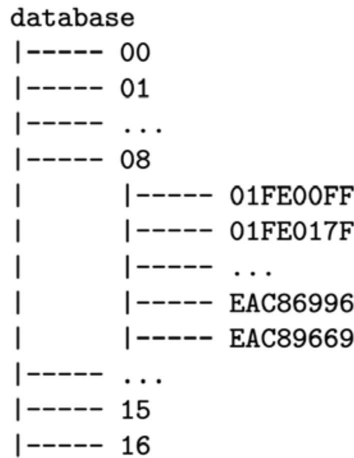

Figure 3.12: Database Directory Tree

Table 3.9 shows the raw size of the quarto database. Example: There are 173740 canonical tiers in level 8 each containing 8! positions which require 4 bits each to encode value-remoteness, so the total raw size of level 8 is $173740 \times 8! \times \frac{4}{8}$ B = 3.263 GiB.

Level	Number of Canonical Tiers	Number of Positions Per Tier	Number of Value-Remoteness Bits Per Position	Size of Tier Results File (GiB)	Total Size of Canonical Tier Results Files in Level Directory (GiB)	Cumulative Database Size (GiB)
0	1	1	5	0.001	0.001	0.001
1	8	1	5	0.001	0.001	0.001
2	130	2	4	0.001	0.001	0.001
3	1092	6	4	0.001	0.001	0.001
4	8148	24	4	0.001	0.001	0.001
5	31416	120	4	0.001	0.002	0.002
6	88740	720	4	0.001	0.030	0.032
7	149650	5040	4	0.001	0.352	0.383
8	173740	40320	4	0.001	3.263	3.645
9	118900	362880	4	0.001	20.092	23.737
10	57222	3628800	3	0.002	72.520	96.257
11	16296	39916800	3	0.014	227.179	323.436
12	3276	479001600	3	0.168	548.040	871.476
13	364	6227020800	3	2.175	791.614	1663.089
14	40	87178291200	2	20.298	811.912	2475.000
15	2	1307674368000	2	304.467	608.934	3083.934
16	1	20922789888000	1	2435.734	2435.734	5519.667

Table 3.4: Raw Database Size

At level n, the last column of Table 3.9 (cumulative database size) indicates the size of the database if only levels 0 through n inclusive are stored. If all levels are stored, then the total raw size of the database is 5519.687 GiB = 5.39 TiB, which is undesirably large.

Instead of storing every level, choose a *boundary level* n such that we only store all levels at least as high as the boundary level (i.e., all levels L_w such that $w \leq n$). Now suppose a query is received for a

position belonging to a level lower than the boundary level (i.e., $p \in L_w$ and $w > n$). Instead of performing canonicalization and reading from a database file, we solve the Quarto subgame with initial position p to determine p's value. We refer to this as **live-solving** a position and it is an effective way to compress the extremely large lower levels. In order to determine the value of p via live-solving, it is necessary for the solving to be extremely efficient (solving will be discussed in more detail in the Solving section).

The subgames rooted at positions in the lowest levels are not large since they are shallow (the end of the game is guaranteed to occur in few moves) and few legal moves are available (low fanout). At higher levels, live-solving takes longer. Level 12 was chosen as the boundary level for the solve, so the raw database size is 871.4 GiB (again, refer to the Cumulative Database Size column). Other boundary levels may be chosen depending on what the user desires with regard to how fast a position query should be and how much space they want the database to take up. One can simply delete the appropriate level directories if they decide to use a higher boundary level for position queries.

Timing tests can help determine what the boundary level should be. A timing experiment was run to determine how long it takes on average to live-solve a position, at each of the lower tiers. The following was done for each lower level n:

- Generate N random position strings corresponding to positions at level n. Start the timer.
- For each position string, convert it to its tier-and-bitboard representation and live-solve using that representation. Then end the timer.
- The average time to live-solve a position in L_n is the total time divided by N. For the lowest levels $16, 15, ..., 11$, we used $N = 1000000000$. For levels $10, 9, 8$ we used $N = 1000$, and for level 7 we used $N = 100$.

Level n	16	15	14	13	12	11	10	9	8	7
Average Time To Live-Solve 1 Million Positions in L_n (s)	0.053	0.065	0.081	0.12	0.36	3.5	67	2,200	99,000	6,300,000

Table 3.5: Results of Live-Solve Timing Experiment

If, for example, one does not mind as long as a single position query takes less than a second, live-solving can be done for positions in levels 8 or lower, and 7 can be chosen as the boundary level so that the total raw database size is 0.383 GiB.

3.10 Solving

Our goal is to solve the game and count the number of positions per value-remoteness pair, i.e., count how many positions are `L0`, `T0`, `W1`, etc. We achieve this by solving and counting each position (filtering out unreachable positions) in each canonical tier. In the Results section we discuss how to postprocess the counts from each canonical tier. Algorithm 1 shows at a high level how the entire game is solved. In the actual implementation, some loops and conditionals are reordered.

3.10.1 Overview

Terminology: A level k is "above" level n if $k < n$ and "below" level n if $k > n$.

We only explore positions in canonical tiers (Line 3). For each canonical tier, we perform value-remoteness counting (Lines 4, 14, 16). For every tier at or above the boundary level, the value-remoteness of each position in the tier is written to the tier results file (Lines 5, 14, 17).

For every tier at or below the boundary level, we live-solve each position to determine the position's value

Algorithm 1 Main Driver for Solving Quarto

```
1: Initialize any tables useful for symmetries and hashing. Determine list of all canonical tiers.
2: for all levels n = 16, 15, 14, ..., 0 do
3:   for all canonical tiers T in L_n do
4:     Initialize counts, a data structure for storing T's value-remoteness counts.
5:     If n ≤ BOUNDARY, initialize results, a n!-length array for storing the value-remoteness of each
       position in T.
6:     If n < BOUNDARY, perform initialization required for solving T.
7:     for all tierpositions ρ in {0, 1, 2, ..., n!} do
8:       bitBoard = Unhash(T, ρ)
9:       if n ≥ BOUNDARY then
10:        vr = SolveLive(T, bitBoard)
11:      else
12:        vr = SolveFromChildTiers(T, bitBoard)
13:      end if
14:      Given vr (which is one of UNREACHABLE, L0, T0, W1, etc.), update counts and, if n ≤ BOUNDARY,
         update results.
15:    end for
16:    Write counts to T's value-remoteness counts file.
17:    If n ≤ BOUNDARY, then write vr to T's position results file.
18:  end for
19: end for
```

(Line 10). They make use of the function `SolveLive` which uses tree recursion to determine a position's value-remoteness.

For every tier above the boundary level, we determine the value of each position by reading the value-remoteness of its child positions from the child tier files (Line 6, 12).

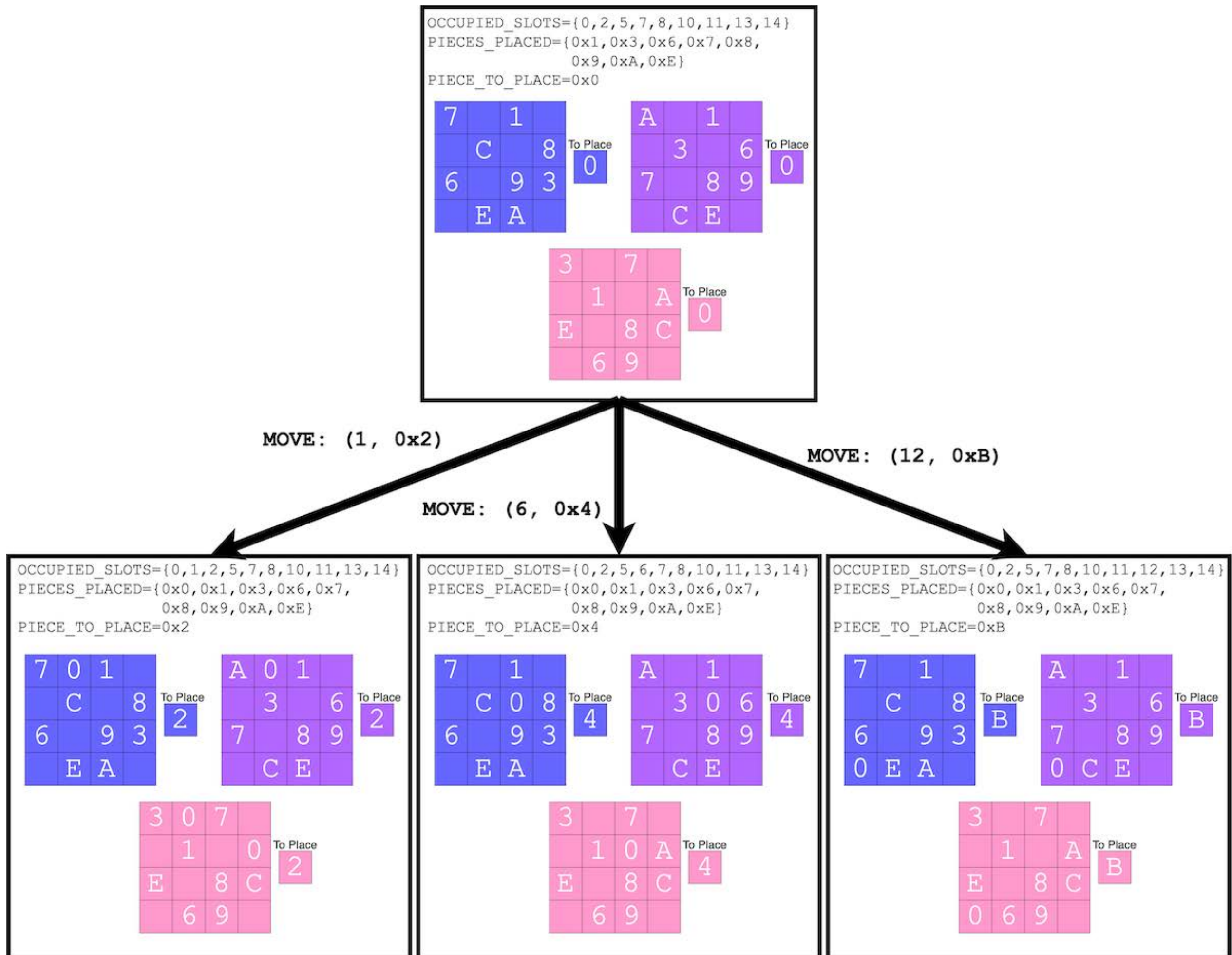

Figure 3.13: Moves and Child Tiers

A *move* is a 2-tuple (e.g., $(9, \texttt{0x3})$) indicating the slot at which one decides to place the piece-to-place and the piece they choose as the next piece-to-place. We defined tiers in such a way that all non-primitive positions in the same tier have the same set of legal moves because they all have the same set of empty slots and set of pieces that have not yet been chosen as a piece-to-place. In Figure 3.13, we show three non-canonicalized child tiers of a parent tier that result from three different legal moves available to every non-primitive position in the parent tier. We also show three different positions in the parent tier and what their child positions are after each of the three moves are made. After canonicalization, each of these child tiers will have a piece-to-place of `0x0` and likely a different `OCCUPIED_SLOTS` and `PIECES_PLACED`.

At the beginning of solving a tier T, we perform initialization for solving a tier from child tiers (Line 6). First we determine what the legal moves are and what non-canonicalized child tiers result from those moves. We then determine what the canonical versions of each of these child tiers are and what transformations are made to canonicalize the child tiers. These transformations can then be applied to each child position to obtain a position symmetric to the child position that belongs to a canonical tier so that we can read its value. Applying these transformations to positions is done in `SolveFromChildTiers` (Line 12).

As discussed in subsection 3.7.3 Canonicalization and Tier Symmetries, every canonical tier has a `PIECE_TO_PLACE` of `0x0`, so for any position p in a canonical tier, every child position c has exactly the same bitboard as p because blanks and the zero piece are encoded the same way. This is a nice property of our tier definition and canonicalization that we use in both `SolveLive` and `SolveFromChildTiers`.

3.10.2 Optimizations

When solving a tier from child tiers, we load all child tier files into memory. For the larger levels that are not live-solved, the entire child level does not fit entirely into memory, so only a few child tiers from the lower level can be loaded. As a result, child tiers are reloaded on multiple occasions when solving an entire level. We make use of an LRU cache to reduce the frequency of reloading child tiers. The size of the LRU cache is adjustable, but a cache size of 100GB is used for this solve.

We use OpenMP [16] to parallelize position solving within a tier. Specifically, Line 7 of Algorithm 1 is parallelized. The solver is configured to use all available cores on the machine.

The LRU cache reduces the I/O footprint. While the solver was running, the CPU utilization was close to 100%, indicating that the solver was not blocked by I/O.

3.11 Results

3.11.1 Timing

Level	Number of Positions Among Canonical Tiers	Duration (HH:MM:SS)	Seconds	Positions / Second
16	20922789888000	38:33:44	138824	150714501
15	2615348736000	5:37:16	20236	129242376.8
14	3487131648000	6:38:19	23899	145911194.9
13	2266635571200	5:27:13	19633	115450291.4
12	1569209241600	9:51:57	35517	44181919.7
11	650484172800	10:08:47	36527	17808310.9
10	207647193600	5:31:15	19875	10447657.5
9	43146432000	1:51:50	6710	6430168.7
8	7005196800	0:27:03	1623	4316202.6
7	754236000	0:04:31	271	2783158.7
6	63892800	0:00:38	38	1681389.5
5	3769920	0:00:07	7	538560
4	195552	0:00:01	1	195552
3	6552	0:00:01	1	6552
2	260	0:00:00	0.03	8666.7
1	8	0:00:00	0.01	800
0	1	0:00:00	0.01	100

Table 3.6: Level Solve Durations

In total it took 3.51 days to solve and position-count on a 12-core machine (1010.54 core-hours). If we were not interested in position-counting the non-stored levels (16, 15, 14, 13), then it would have taken 1.16 days to solve and position-count the stored levels. If we were only interested in solving and not position-counting, then there would be no need to perform unreachable position checks in the primitive function, and the time to solve would have been less than 1.16 days.

3.11.2 Value Counts

While solving, we keep track of the number of positions per value-remoteness pair in that tier. For each canonical tier, we multiply its counts by the number of tiers it is symmetric to (i.e., the length of the tier's orbit). To get value-remoteness counts for a particular level, we add up all the counts for all tiers in the level. However, we must adjust the counts due to the consequence of our position representation that each primitive positions is represented and counted multiple times. (See section 3.2 Position Definition and String Representation.) A primitive position at level $n < 16$ is represented $16 - n$ times since the primitive board state can be paired with any piece-to-place out of the $16 - n$ remaining pieces. All primitive positions at level $n < 16$ are "Lose in 0", so we divide the count for "Lose in 0" by $16 - n$ to get the real count after filtering out primitive position duplicates under our position representation.

The final counts for value-remoteness for each value-remoteness pair and level are presented in the table below.

LEVEL	Tie in (16 - LEVEL)	Lose in 0	Win in 1	Lose in 2	Win in 3	Lose in 4	Win in 5	Lose in 6	Win in 7	Lose in 8	Win in 9	Lose in 10	Win in 11	Level Total
0	1.60E+01													1.60E+01
1	3.84E+03													3.84E+03
2	4.03E+05													4.03E+05
3	2.39E+07		5.15E+05											2.45E+07
4	8.54E+08	1.29E+05	7.41E+07										2.47E+07	9.53E+08
5	2.05E+10	1.85E+07	4.33E+09		4.40E+06		1.66E+06		5.23E+06		8.51E+06	1.43E+07	6.60E+07	2.50E+10
6	2.35E+11	1.12E+09	1.34E+11	8.85E+05	1.88E+09	5.01E+05	1.79E+09	2.40E+06	9.00E+09	3.60E+06	6.84E+10	2.82E+07		4.52E+11
7	2.35E+12	3.72E+10	2.44E+12	5.37E+08	1.11E+11	1.30E+09	9.96E+10	3.40E+09	1.72E+11	4.33E+10	3.81E+11			5.64E+12
8	1.04E+13	7.45E+11	2.69E+13	3.71E+10	2.16E+12	7.23E+10	2.88E+12	9.48E+10	4.75E+12	1.85E+11				4.82E+13
9	3.53E+13	9.28E+12	1.81E+14	9.23E+11	1.54E+13	1.86E+12	1.38E+13	3.36E+12	1.53E+13					2.77E+14
10	8.84E+13	7.23E+13	7.35E+14	7.66E+12	6.16E+13	9.34E+12	5.14E+13	9.32E+12						1.03E+15
11	1.28E+14	3.45E+14	1.71E+15	3.09E+13	1.15E+14	3.21E+13	6.91E+13							2.43E+15
12	1.55E+14	8.76E+14	2.12E+15	5.46E+13	1.24E+14	3.95E+13								3.37E+15
13	9.29E+13	1.20E+15	1.23E+15	4.75E+13	7.09E+13									2.64E+15
14	4.07E+13	7.26E+14	2.67E+14	1.98E+13										1.05E+15
15	6.63E+12	1.49E+14	1.24E+13											1.68E+14
16	4.14E+11	5.58E+12												5.99E+12
Value Total	5.61E+14	3.38E+15	6.28E+15	1.61E+14	3.89E+14	8.28E+13	1.37E+14	1.28E+13	2.03E+13	2.28E+11	4.49E+11	4.25E+07	9.07E+07	1.10E+16

Table 3.7: Number of Positions Per Value-Remoteness and Level. See Appendix B for a table of exact counts.

Notice that for levels 0, 1, and 2, the game is a tie in 16, 15, and 14, respectively. We need not reference the database if asked the value-remoteness of a position in these levels; we can just return “Tie”.

The initial position is not included in this table, so we add 1 to our total count of reachable positions. The initial position is “Tie in 17” because its 16 child positions are all “Tie in 16”.

In total, there are 11,029,662,094,763,537 reachable Quarto positions.

3.12 Value-Moves Interface

GamesmanClassic handles Quarto position requests by reading the database or live-solving to return value-remoteness data, so GamesmanUni can now make Quarto API calls. I created a Quarto Custom GUI that is now available on GamesmanUni. As with other games on GamesmanUni, the move buttons are colored according to their corresponding move value.

- At the initial position, the user clicks one of the 16 moves buttons to indicate which piece the first player gives to the second player.
- Each move thereafter is a placement of a piece and a choice of the next piece-to-place via a single click. When a move button is clicked, the piece-to-place is placed on the slot where the move button is located and the selected next piece-to-place is the piece displayed on the move button. Especially at the beginning of the game, there are many such buttons displayed simply because there are so many legal moves available. I settled on this design for showing the value for every single move.
- A different kind of move button is used when a move immediately leads to a primitive position. Clicking it performs a piece placement only and there is no selection of a next piece-to-place since the game has ended.

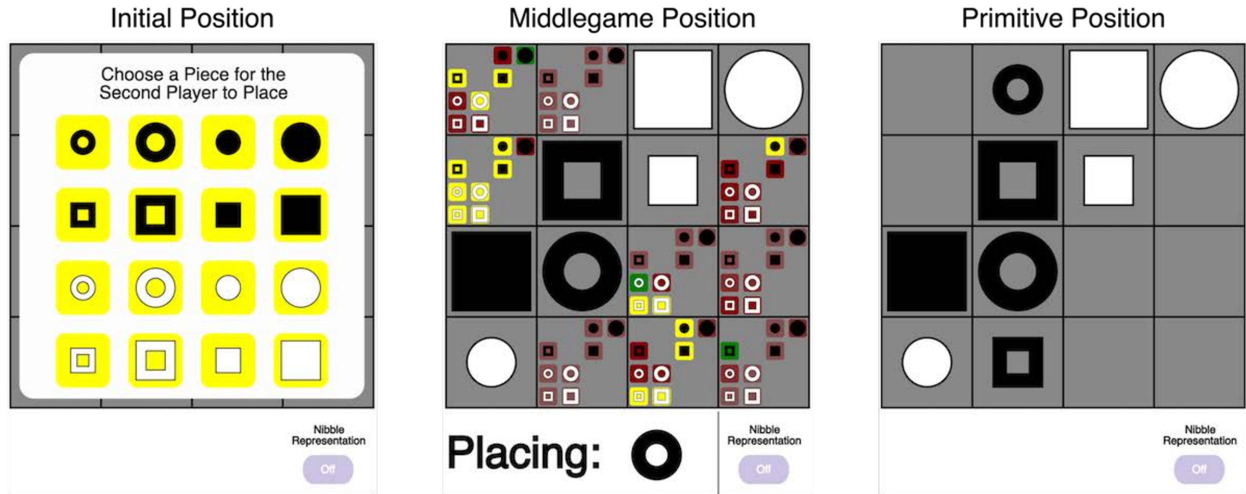

Figure 3.14: Quarto Interface

The bottom-right button toggles the nibble representation of the pieces. The piece bits are arranged in a square.

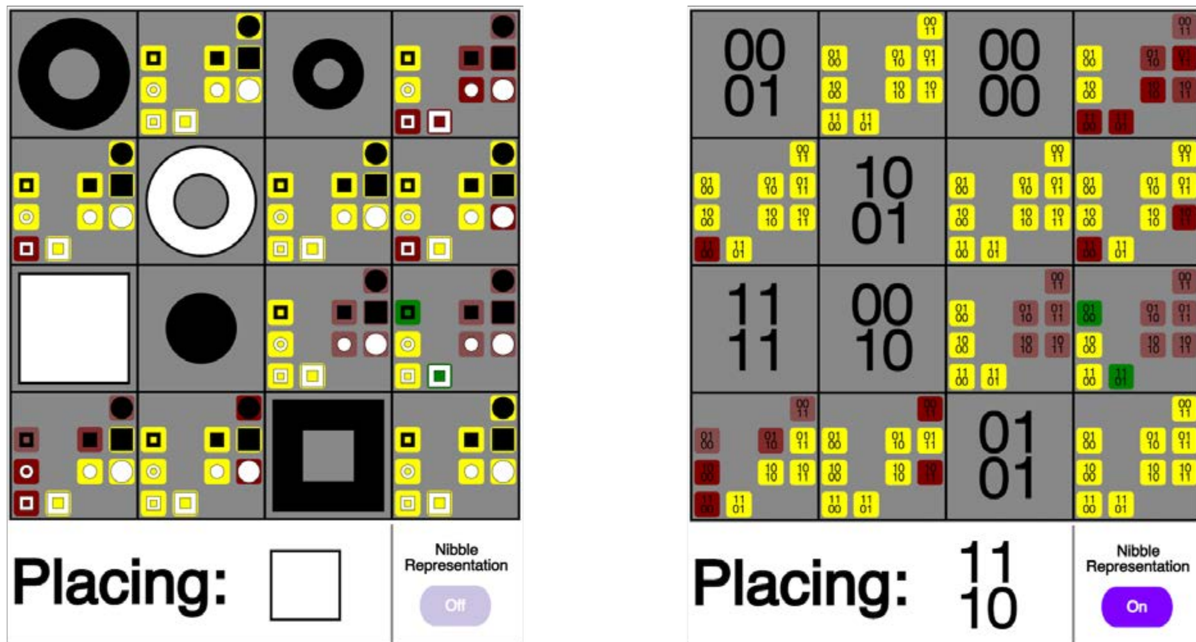

Figure 3.15: Toggling Nibble Representation

3.13 Reflection and Future Work

Seemingly, the value-remoteness encoding idea that we used for Quarto is most applicable to dartboard games in which a tie only occurs when the board is filled. For example, Tic-Tac-Toe and Connect4 can each be split into 10 levels and 43 levels, respectively, and the anticipated value-remoteness pair table for each can be constructed according to the same process by which the table for Quarto was constructed.

Live-solving as a form of compression was particularly helpful for Quarto because it is a shallow game and lev-

els with the highest numbers of positions were located near the bottom of the level tree. Consider a different theoretical game that has the following level sequence $L_0 \rightarrow L_1 \rightarrow ... \rightarrow L_a \rightarrow L_b \rightarrow L_c \rightarrow ... \rightarrow L_{N-1} \rightarrow L_N$. Suppose live-solving in this game is reasonably fast. Then, as with Quarto, we can choose to live-solve the bottommost levels L_{N-j} to L_N for small j rather than storing them. In addition, if $|L_b|$, for example, is significantly greater than either $|L_a|$ or $|L_c|$, and if data for a position in L_b is requested, it can be live-solved by loading its child positions from L_c, which would be more space-efficient than storing each position in L_b.

Future work includes but is not limited to...

- Incorporating some of the ideas from the Quarto solve into solvers and databases in GamesmanClassic. For example, one can create a new database format in GamesmanClassic that utilizes the value-remoteness encoding idea or further develop TierGamesman so that it supports tier symmetries.
- Compressing the Quarto database even further using machine learning, which would involve finding the smallest model possible that, after training on each instance in the Quarto database we produced, can classify each Quarto position by its value-remoteness pair either (1) with 100% accuracy or (2) with extremely high accuracy but for which we would need to keep track of a separate database of incorrectly classified positions. In the second case, the value-remoteness of a position is obtained by first checking the separate database. If the position exists in the database, its value-remoteness is returned based on the database. Otherwise, if the position is outside the database, the model's classification is returned since the model correctly classifies every such position.
- Creating a Quarto value-remoteness interface that incorporates multipart moves (explained in Chapter 6) in which each move is specified by two clicks – the first of which specifies where to place the piece-to-place and the second of which specifies which piece to select. The current interface displays many small move buttons and a multipart-move interface would be less cluttered, although it may sacrifice the feature from the current interface that all of a position's legal moves and their values are presented all at once. It is ideal to give users the option to switch between the current interface and one that uses multipart moves.

Chapter 4

Solving Tierable Loopy Games

We now discuss how we solved four games – Nine Men's Morris, Bagh-Chal, Tic-Tac-Two, and Topitop. Unlike Quarto, these games are loopy and in all of them there exist draw positions. The solves described in this chapter were done within the GamesmanClassic framework.

We describe the solver and general strategies used to solve and analyze each of the games. Then for each game we present its rules, our tier definition, the time to solve, and value-remoteness counts.

4.1 Solver

Algorithm 2 shown below for solving loopy games uses the *frontier* F, a priority queue of positions in which a position p_a has a higher priority than another position p_b if and only if p_a's value-remoteness is worse than that of p_b.

At any point in the execution of this algorithm, any positions p that have ever been enqueued to F are positions whose values are *finalized*, i.e., p's actual value is known.

Algorithm 2 An Algorithm for Handling Loopy Games

```
1:  For each position p, initialize valueremoteness[p] ← UNDECIDED.
2:  for all primitive positions p_r do
3:      Set valueremoteness[p_r] to p_r's primitive value (with remoteness 0).
4:      Enqueue p_r to F.
5:  end for
6:  while F is nonempty do
7:      Dequeue a position c from F.
8:      for all parent positions p of c do
9:          if p has never been in F then
10:             Update valueremoteness[p] according to valueremoteness[c].
11:             if all of p's child positions have been in F or valueremoteness[c] is LOSE then
12:                 Enqueue p to F.
13:             end if
14:         end if
15:     end for
16: end while
17: for all positions p_u that have never been in F do
18:     if valueremoteness[p_u] ∈ {LOSE, UNDECIDED} then
19:         valueremoteness[p_u] ← DRAW
20:     end if
21: end for
```

Line 9 ensures that no position is put into frontier twice. This is a sufficient condition to ensure that the algorithm – particularly the while loop – terminates.

At any point during the solve, if `valueremoteness`[p] is not UNDECIDED, then it contains the value-remoteness-at-best of position p based on the value-remotenesses of p's finalized children. p's child positions are finalized one-at-a-time, so we keep track of the value of p in Line 10.

There are several ways one can assemble the necessary data structures for this algorithm. One can perform a DFS from the initial position to visit all reachable positions and assemble a graph which keeps track of each position's parent positions. For large games this requires too much memory so we have a method known as `UndoMove` which can derive the parent positions of a given child position. The `UndoMove` is algorithmic and does not require additional memory, but it comes with the tradeoff that the solve is slightly slower.

Algorithm 3 Solving Positions in a Given Tier T

```
 1: For each position p ∈ T, initialize valueremoteness[p] ← UNDECIDED.
 2: Enqueue every child position from every child tier of T to F.
 3: for all primitive positions p_r do
 4:   Set valueremoteness[p_r] to p_r's primitive value (with remoteness 0).
 5:   Enqueue p_r to F.
 6: end for
 7: while F is nonempty do
 8:   Dequeue a position c from F.
 9:   for all parent positions p of c such that p ∈ T do
10:     if p has never been in F then
11:       Update valueremoteness[p] according to valueremoteness[c].
12:       if all of p's child positions have been in F or valueremoteness[c] is LOSE then
13:         Enqueue p to F.
14:       end if
15:     end if
16:   end for
17: end while
18: for all positions p_u ∈ T that have never been in F do
19:   if valueremoteness[p_u] ∈ {LOSE, UNDECIDED} then
20:     valueremoteness[p_u] ← DRAW
21:   end if
22: end for
```

Algorithm 3 discusses how to solve a particular tier of a game in a retrograde tier-solve. The differences from Algorithm 2 are highlighted in red. The solves described in this chapter use Algorithm 3.

4.2 General Strategies

Each of the four games were split into tiers. Each tier in each of the following games involves some form of rearrangement of pieces among the slots of the board. Each tier contains a number of positions that are represented by a hash value obtained by a perfect hash function that hashes into a value between 0 and $|T| - 1$, where $|T|$ is the size of the tier.

We first demonstrate the idea behind rearrangement hashing by example. Suppose we wish to hash an arrangement of 4 "O" pieces and 4 "X" pieces among 12 slots. There are a total of $N = \frac{12!}{4!4!4!} = 34650$ total ways one can arrange the 4 O's and 4 X's (and 4 blanks) among the 12 slots. This is equivalent to the problem of hashing all possible anagrams of the string "`----OOOOXXXX`". The rearrangement hash bijectively assigns a hash value in $\{0, 1, 2, ..., N - 1\}$ to each possible arrangement.

The exact hash value for a given even component configuration is the place it appears in an lexicographical ordering of all anagrams. Suppose we define the following symbol ordering: '-' < 'O' < 'X'. Presented below is a lexicographical ordering of the anagrams with their associated hash values.

```
    0: ----OOOOXXXX
    1: ----OOOXOXXX
    2: ----OOOXXOXX
    3: ----OOOXXXOX
    4: ----OOOXXXXO
    5: ----OOXOOXXX
    ...
17570: OOOXX-O-XX--
17571: OOOXX-OX---X
17572: OOOXX-OX--X-
17573: OOOXX-OX-X--
17574: OOOXX-OXX---
17575: OOOXX-X---OX
    ...
34644: XXXXOO-OO---
34645: XXXXOOO----O
34646: XXXXOOO---O-
34647: XXXXOOO--O--
34648: XXXXOOO-O---
34649: XXXXOOOO----
```

In GamesmanClassic, the *generic hash* library [17] is used for hashing rearrangements; however, we implemented an optimized version of the hash for each game. The specific optimized version of the Nine Men's Morris hash is presented in the next section.

Initial profiling of the Nine Men's Morris solve revealed that most time is spent unhashing. The unhash function took up 63% of the time of the solve. We thus implemented an unhash cache to store the results of recent unhashing.

Another optimization is to reduce the amount of computation by utilizing symmetries. We define a canonical position as the position with the smallest hash value in its orbit.

4.2.1 Database Compression

Playing the game after it has been solved often involves random access of positions in tier files. Some tiers are so large that random access of a position in a compressed tier file is extremely slow. To address this issue, instead of writing a tier results array directly to a file and gzipping the file, we use a checkpoint technique that allows random access without needing to uncompress a database up to the position.

The compression of each tier .gz file now works as follows:

1. Split the raw data into 1MB chunks.
2. gzip each chunk individually.
3. Take note of the sizes of each chunk and record their sizes in order. We refer to this file as the gzip checkpoint table for this tier.
4. Concatenate the gzipped chunks into a new .gz file.

This compression is backward compatible [18] with the existing GamesmanClassic database reader because it can be gunzipped in its entirety to obtain the original data. During gameplay, if a gzip checkpoint table

is detected, it can be used to uncompress only the 1MB chunk of data containing the position.

To obtain a position's data from a file directly:

1. Determine the tier and tierposition of the given position.
2. Calculate the index in the uncompressed tier file where the value would be located (based on the tierposition).
3. Calculate the chunk the position data is in and the offset within the uncompressed chunk.
4. From the checkpoint table, lseek to the start of the correct chunk, then gzseek to the offset within the uncompressed chunk. Then gzread to get the position's data.

Compared to gzipping the raw data directly, compressing the Nine Men's Morris database (using 1MB chunks) this way results in a 1% increase in the overall compressed database size. We thus felt justified in using this form of compression for other games to allow for fast random access. The fast random access also reduces the load on the server hosting the games.

4.3 Nine Men's Morris

4.3.1 Rules

Nine Men's Morris [19] is played on a board shown in Figure 4.3.1. Pieces are placed on line intersections and can move to an adjacent empty intersections.

The first player controls nine white pieces. The second player controls nine black pieces. A vertical or horizontal three-in-a-row of same-colored pieces is called a *mill*. At any point in the game, if a player creates a mill, they can remove one of their opponent's pieces unless it is in a mill, in which case it is removable only if all of their opponent's pieces are also in a mill.

The game consists of two phases. In the first phase of the game, each player takes turns placing one of their pieces on an empty intersection point on the board. Once all pieces have been placed, the second phase begins, in which each player takes turns moving one of their pieces to an adjacent empty intersection point. If a player has been reduced to three pieces, the pieces can *fly*, i.e., the pieces can be moved to any empty intersection point on the board.

A player wins when they have reduced their opponent to two pieces or they have trapped their opponent's pieces such that their opponent has no legal moves.

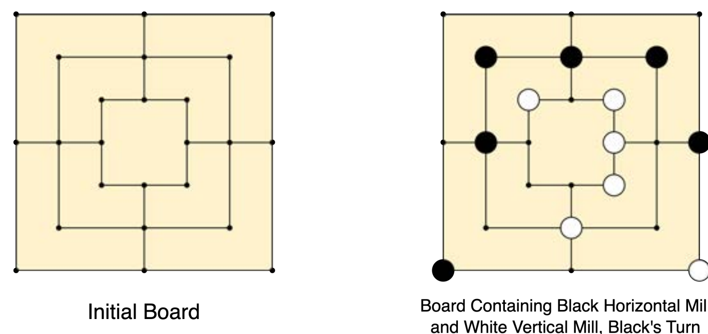

Figure 4.1: Example Nine Men's Morris Positions

4.3.2 Position Representation

A position is defined by a 3-tuple $(R, B, \texttt{TURN})$, where R is the total number of pieces left to place, B is the position's board state, and TURN is a bit indicating whose turn it is at the position. $R > 0$ indicates that this tier belongs to Phase 1. Nonzero even R indicates that it is white's turn to place a piece and odd R indicates that it is black's turn to place a piece. A board B contains the state of each slot of the board, whether it is empty, occupied by a black piece, or occupied by a white piece. A board can be represented as a 24-character string, with each character corresponding to a slot on the board in row major order. For example, the boards in 4.3.1 are represented as "`------------------------`" and "`---BBBW-W-B-W-B--W-W---W`".

4.3.3 Tier Definition

A tier is defined by a 3-tuple (R, N_W, N_B), where R indicates the number of remaining pieces to place, N_W indicates the number of white pieces on the board, and N_B indicates the number of black pieces on the board. A position $p = (R_p, B_p, \texttt{TURN}_p)$ belongs to tier $T_1 = (R_{T_1}, N_{T_1,W}, N_{T_1,B})$ if there are $N_{T_1,W}$ white pieces and $N_{T_1,B}$ black pieces on B_p.

Suppose we are at a Phase 1 position $p_1 \in T_1$. If, for example, it is white's turn, then either white can place a piece without opponent piece removal or, if white has enough pieces to form a mill, then white can place with opponent piece removal, which means that T has two child tiers. Now suppose we are at some Phase 2 position p_2 in a different tier $T_2 = (R_{T_2} = 0, N_{T_2,W}, N_{T_2,B})$. It can be either black's turn or white's turn at p_2, and a piece may be removed. Thus, the child tiers of T_2 be $(0, N_{T_2,W} - 1, N_{T_2,B})$ and $(0, N_{T_2,W}, N_{T_2,B} - 1)$.

Under this tier definition and rule for generating child tiers of a given tier, there are 270 Phase 1 and 53 Phase 2 tiers, for a total of 333 tiers considered during the solve.

Each tier $T = (R_T, N_{T,W}, N_{T,B})$ contains all states formed via every combination of (1) all possible turn bits and (2) all board states – the possible ways to arrange $N_{T,W}$ white pieces and $N_{T,B}$ black pieces among the $S = 24$ intersection points. The number of board states is given by:

$$\frac{S!}{N_{T,W}!N_{T,B}!(S - N_{T,W} - N_{T,B})!}$$

If T is in Phase 1, then the parity of R_T indicates whose turn it is. If T is in Phase 2, then a selected position in T may be such that it is the first player's turn or the second player's turn, so we multiply the number of board states in Phase 2 by 2 to obtain the number of positions in the tier.

$N_{T,W} = N_{T,B} = 8$ are the values of $N_{T,W}$ and $N_{T,B}$ such that the number of possible board arrangements is maximized (9,465,511,770). The tier with the most positions is (0, 8, 8), with 18,931,023,540 positions.

4.3.4 Hashing

We define a hash function that takes a 24-character board string as input and calculates a value between 0 and $M - 1$ where M is the number of ways the pieces in the board string can be arranged.

Define $Q(s, w, b)$ as the number of ways one can arrange w white pieces and b black pieces among s slots.

$$Q(s, w, b) = \frac{s!}{w!b!(s - w - b)!}$$

To speed up the hashing, all possible values of $Q(s, w, b)$ are pre-computed so only a lookup is necessary when hashing.

Algorithm 4 Hash Value of a Board State with Board String B

```
1: hashvalue ← 0
2: s ← 23
3: while s > 0 do
4:   if B[s] = 'W' then
5:     w ← w − 1
6:   else if B[s] = 'B' then
7:     if w > 0 then
8:       hashvalue ← hashvalue + Q(s, w − 1, b)
9:     end if
10:    b = b − 1
11:  else
12:    if w > 0 then
13:      hashvalue ← hashvalue + Q(s, w − 1, b)
14:    end if
15:    if b > 0 then
16:      hashvalue ← hashvalue + Q(s, w, b − 1)
17:    end if
18:  end if
19:  s ← s − 1
20: end while
```

4.3.5 Symmetry

Nine Men's Morris has 16 board symmetry transformations – rotating, reflecting, and swapping the inner and outer rings each yield symmetric positions. To take advantage of symmetries, we define a canonical position as the position with the smallest hash value out of a set of symmetric positions.

To find the canonical position of any position, we can perform 16 symmetry transformations, hash the resulting board strings, and find the one with the smallest hash value. However, a more efficient method is to compare the board string of the transformations character by character, starting with the last character, and only hash the transformation that yields the smallest hash value. This method is correct because based on the definition of the hash value, the last character in the string representations is most significant, followed by the second from last, etc.

4.3.6 Results

The game is a draw. The solve took 9.4 days. Although the $(0, 8, 8)$ tier contains the most positions, the $(0, 9, 8)$ and $(0, 8, 9)$ tiers each required the most memory to solve – they each involved keeping track of the values of 16,827,576,480 current tier positions and 35,758,600,020 child tier positions. The size of the gzipped database directory is 22.05 GiB.

4.3.7 Variants

We have also solved variants of Nine Men's Morris. These variants may allow the removal of an opponent's piece even if it is in a mill, or disallow flying when a player has been reduced to 3 pieces. These variants are also draw games. The approach to solving these variants are the same and the solve time is similar.

4.4 Bagh-Chal

4.4.1 Rules

Bagh-chal [20] is played on a 5x5 (25-space) board with edges between spaces indicating adjacencies (see Figure 4.4.1). The first player plays with 20 goats and the second player plays with 4 tigers. At the initial

Pieces Remaining	# of White Pieces	# of Black Pieces	# of Positions	Solve Time (HH:MM:SS)
2	8	8	9,465,511,770	03:16:10
1	8	8	9,465,511,770	03:13:55
2	8	7	8,413,788,240	02:54:11
1	7	8	8,413,788,240	02:51:23
1	9	7	8,413,788,240	02:49:58
2	7	8	8,413,788,240	02:45:51
1	8	7	8,413,788,240	02:45:09
1	9	8	8,413,788,240	02:41:11
3	8	7	8,413,788,240	02:38:57
1	7	7	6,731,030,592	02:19:36
		: 260 tiers hidden :		
			285,325,357,201	93:43:20

Table 4.1: Timing for Solve of 9 Men's Morris Phase 1

# of White Pieces	# of Black Pieces	# of Positions	Solve Time (HH:MM:SS)
8	8	18,931,023,540	09:35:06
9	8	16,827,576,480	09:12:43
8	9	16,827,576,480	09:04:10
9	7	16,827,576,480	08:27:38
7	9	16,827,576,480	08:24:47
8	7	16,827,576,480	07:55:58
7	8	16,827,576,480	07:48:30
7	7	13,462,061,184	05:47:52
9	9	13,088,115,040	07:45:17
6	9	13,088,115,040	05:57:23
	: 53 tiers hidden :		
		286,071,923,192	131:00:36

Table 4.2: Timing for Solve of 9 Men's Morris Phase 2

position, none of the goats are on the board and the tigers occupy the corners, as shown in 4.2 (left).

In the first phase of the game, the first player places one goat on their turn on any empty space on the board. Once 20 goats have been placed, the second phase of the game begins, and the first player, on each turn, moves one goat to an adjacent empty space.

Throughout the entire game, on the second player's turn, they either (1) move one of the tigers to an adjacent empty space or (2) capture a goat, i.e., move one of their tigers two spaces in a straight line if doing so crosses over a goat to an empty space, after which the goat is removed from the board.

The first player's objective is to leave the second player with no legal moves and the second player's objective is to capture 5 goats.

	Non-Canonical	Canonical
L0	255,079,008	15,980,258
W1	1,903,621,882	119,060,379
L2	615,396,939	38,503,549
W3	1,415,377,337	88,512,437
L4	1,113,734,328	69,655,726
W5	4,572,801,412	285,917,866
L6	3,139,672,428	196,318,051
W7	9,478,600,192	592,596,006
L8	6,072,111,812	379,659,724
W9	14,134,103,120	883,627,685
L10	8,419,632,190	526,407,582
W11	16,851,953,344	1,053,507,912
L12	9,392,209,316	587,195,266
W13	16,935,674,846	1,058,732,607
L14	9,144,963,894	571,731,049
W15	15,121,560,678	945,323,806
L16	8,309,785,868	519,512,760
W17	13,074,027,742	817,323,476
L18	7,252,751,092	453,424,982
W19	11,188,665,960	699,456,273
L20	6,150,063,609	384,485,973
W21	9,425,910,213	589,256,698
L22	5,059,997,114	316,337,275
W23	7,716,064,746	482,365,244
L24	4,058,257,060	253,710,464
W25	6,030,166,088	376,972,440
L26	3,160,365,980	197,576,508
W27	4,525,621,070	282,916,953
L28	2,395,053,138	149,730,885
W29	3,326,453,534	207,951,774
L30	1,771,767,878	110,765,681
W31	2,412,964,822	150,845,678
L32	1,278,070,730	79,901,055
W33	1,723,171,702	107,723,630
L34	900,387,630	56,290,570
W35	1,210,338,322	75,664,928
L36	624,986,986	39,073,302
W37	838,926,402	52,446,297
L38	432,998,562	27,070,830
W39	575,950,828	36,006,775
L40	304,663,154	19,047,931
W41	401,795,648	25,119,403
L42	221,524,288	13,850,087
W43	293,141,416	18,326,858
L44	168,335,156	10,525,056
W45	226,788,696	14,178,843
L46	132,878,090	8,308,178
W47	184,733,934	11,549,714
L48	108,468,976	6,782,374
W49	157,794,672	9,865,508
L50	91,592,544	5,727,062
W51	140,247,252	8,768,524
L52	79,840,476	4,992,257
W53	125,976,660	7,876,285
L54	70,987,716	4,438,769
W55	114,332,508	7,148,462
L56	64,128,812	4,009,879
W57	104,409,816	6,527,875
L58	59,243,228	3,704,240
W59	96,874,196	6,056,666
L60	55,324,696	3,459,337
W61	90,156,204	5,636,706
L62	51,746,432	3,235,414
W63	83,457,100	5,217,841
L64	47,875,696	2,993,551
W65	76,658,804	4,792,797
L66	43,727,400	2,734,066
W67	69,154,772	4,323,709
L68	39,570,052	2,474,321
W69	61,507,952	3,845,605
L70	35,760,992	2,235,951
W71	53,333,888	3,334,546
L72	32,256,824	2,017,028
W73	45,846,656	2,866,574
L74	28,485,086	1,781,001
W75	38,113,332	2,383,009
L76	25,146,748	1,572,389
W77	32,422,600	2,027,207
L78	22,548,352	1,409,801
W79	28,495,236	1,781,714
L80	20,397,908	1,275,442
W81	25,864,800	1,617,259
L82	18,476,742	1,155,299
W83	23,509,540	1,469,959
L84	16,804,164	1,050,752
W85	21,435,632	1,340,310
L86	14,971,864	936,175
W87	18,956,504	1,185,305
L88	12,655,856	791,369
W89	15,889,040	993,440
L90	10,493,440	656,197
W91	13,693,792	856,252
L92	8,974,860	561,248
W93	12,353,480	772,402
L94	8,075,556	505,040
W95	11,703,640	731,829
L96	7,402,468	462,892
W97	11,200,600	700,367
L98	6,865,884	429,344
W99	10,687,952	668,336
L100	6,702,512	419,120
W101	10,971,048	686,012
L102	6,841,600	427,826
W103	11,985,962	749,417
L104	7,137,816	446,306
W105	13,437,004	840,111
L106	7,307,320	456,938
W107	14,261,132	891,644
L108	7,367,708	460,692
W109	14,747,768	921,969
L110	7,086,256	443,077
W111	13,643,296	852,964
L112	6,614,344	413,564
W113	12,004,192	750,459
L114	5,962,240	372,799
W115	9,935,144	621,131
L116	5,310,192	332,024
W117	8,150,272	509,514
L118	4,633,832	289,710
W119	6,587,024	411,834
L120	3,980,136	248,892
W121	5,322,152	332,773
L122	3,419,408	213,795
W123	4,359,920	272,580
L124	2,891,440	180,807
W125	3,681,088	230,151
L126	2,593,368	162,152
W127	3,224,976	201,621
L128	2,505,544	156,687
W129	3,072,584	192,105
L130	2,469,072	154,374
W131	2,879,856	180,043
L132	2,297,752	143,668
W133	2,505,736	156,675
L134	2,068,216	129,307
W135	2,153,248	134,623
L136	1,808,904	113,095
W137	1,832,280	114,552
L138	1,547,240	96,732
W139	1,584,184	99,054
L140	1,325,832	82,889
W141	1,351,124	84,478
L142	1,133,392	70,871
W143	1,170,620	73,220
L144	996,392	62,295
W145	1,038,896	64,948
L146	840,328	52,543
W147	879,264	54,981
L148	689,936	43,137
W149	736,268	46,037
L150	541,848	33,875
W151	598,856	37,439
L152	438,732	27,429
W153	532,576	33,299
L154	405,800	25,368
W155	526,872	32,936
L156	384,552	24,040
W157	540,248	33,772
L158	373,112	23,327
W159	453,424	28,350
L160	399,800	25,000
W161	427,888	26,755
L162	458,664	28,676
W163	477,216	29,834
L164	491,016	30,702
W165	503,272	31,470
L166	477,080	29,838
W167	468,592	29,300
L168	429,912	26,887
W169	381,300	23,841
L170	348,872	21,813
W171	288,504	18,036
L172	284,944	17,814
W173	226,240	14,143
L174	216,552	13,535
W175	160,512	10,037
L176	156,680	9,796
W177	110,340	6,898
L178	113,640	7,103
W179	78,968	4,938
L180	93,864	5,867
W181	54,616	3,417
L182	81,728	5,110
W183	43,304	2,708
L184	63,384	3,962
W185	25,944	1,622
L186	34,872	2,180
W187	15,536	971
L188	20,144	1,259
W189	9,184	574
L190	9,616	601
W191	6,720	420
L192	8,352	522
W193	5,296	331
L194	8,416	526
W195	4,400	275
L196	7,440	465
W197	3,440	215
L198	4,272	267
W199	1,312	82
L200	2,560	160
W201	544	34
L202	704	44
W203	96	6
L204	160	10
Draw	70,184,308,733	4,388,120,765
Total	296,852,156,051	18,558,930,949

Table 4.3: Nine Men's Morris Value-Remoteness Counts

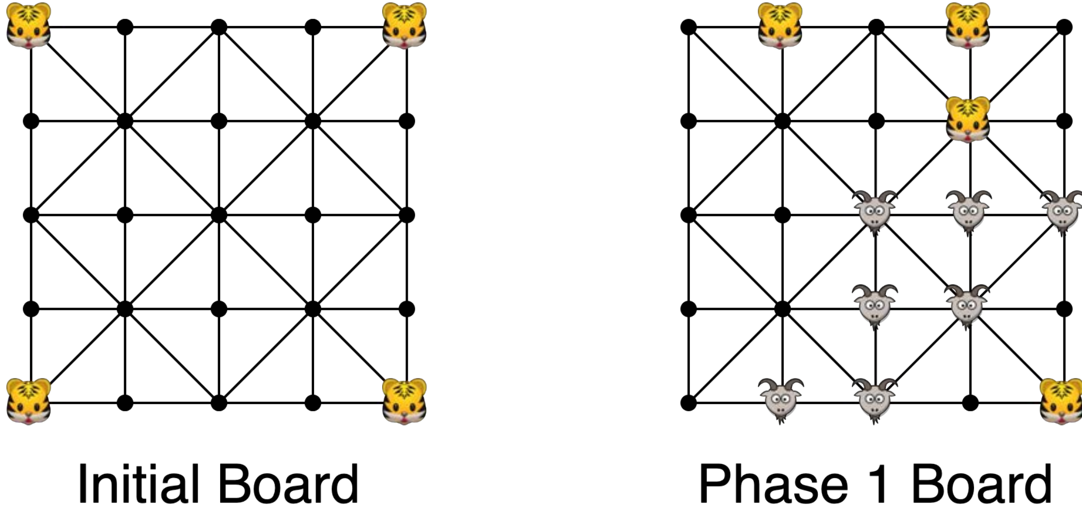

Figure 4.2: Example Bagh-Chal Boards

4.4.2 Position Representation

A state of the board B can be represented as a 25-length ternary string. The i-th trit represents the state of the i-th slot on the board (0=Unoccupied, 1=Occupied by Goat, 2=Occupied by Tiger). A position is defined by a 4-tuple $(R, C, B, \texttt{TURN})$, where R is the number of goats that have yet to be placed, C is the number of goats that have been captured, B is the board state, and `TURN` is a bit indicating whose turn it is at the position.

4.4.3 Tier Definition

We use a 3-tuple $T = (R_T, C_T, \texttt{TURN}_T)$ to represent a tier, where $\texttt{TURN}_T$ is either `GOATS` or `TIGERS` in Phase 1; and is `NULL` in Phase 2. T contains all positions $p = (R_p, C_p, B_p, \texttt{TURN}_p)$ such that $R = R_p$, $C = C_p$, and if R is nonzero, then $\texttt{TURN}_T = \texttt{TURN}_p$.

For the tier definition, the turn bit only matters when the tier belongs to the first phase of the game. Any move made from a tier in phase 1 immediately leads to another tier. The initial tier is $T_I = (20, 0, \texttt{GOATS})$ (which contains the initial position). After a move has been made, the child tier is $T_J = (19, 0, \texttt{TIGERS})$. There are two possible child tiers of T_J because with the tiger's next move, a goat may or may not be captured. The child tiers of T_J are $(19, 0, \texttt{GOAT})$ (if no capture occurs) and $(19, 1, \texttt{GOAT})$ (if a capture occurs). Eventually, we reach a tier in phase 2 such as $T_K = (0, 2, \texttt{NULL})$ (0 goats left to place, 2 goats captured). Any move made from a position in T_K may lead to a position either in T_K or in $(0, 3, \texttt{NULL})$, its single child tier (as a result of a goat being captured). Any tier $(R_X, C_X, \texttt{TURN}_X)$ such that $C_X = 5$ has no child tiers.

Under this tier definition and rule for generating child tiers of a given tier, there are 196 tiers. Each tier $T = (R_T, C_T, \texttt{TURN}_T)$ contains every board state that is achieved by rearranging $20 - R_T - C_T$ goats and 4 tigers among the 25 slots of the board. In addition, phase 2 tiers need to include every (turn, rearrangement) combination since they do not have a defined turn; whereas phase 1 tiers do not. An upper bound on the number of positions calculated by adding up all the possible rearrangements per tier is 291,915,283,550 positions. However, some positions are equivalent under symmetry. There are 8 board symmetry transformations – rotating or reflecting the board yields a symmetric position.

4.4.4 Results

The game is a draw. The solve took 3.41 days. The $(8, 1, \texttt{FALSE})$ tier required the most memory to solve – it involved keeping track of the values of 4,461,857,400 current tier positions and loading in 8,923,714,800 child tier positions. The size of gzipped database directory is 14.34 GiB. The Bagh-Chal interface is discussed in Chapter 5.

Turn	Remaining	Captured	# of Positions	Solve Time (HH:MM:SS)
TIGER	8	2	4,461,857,400	01:32:03
TIGER	10	0	4,461,857,400	01:31:41
TIGER	9	1	4,461,857,400	01:29:47
TIGER	7	3	4,461,857,400	01:29:41
TIGER	9	0	4,461,857,400	01:29:24
TIGER	6	4	4,461,857,400	01:29:18
TIGER	7	2	4,461,857,400	01:28:17
TIGER	8	1	4,461,857,400	01:28:02
TIGER	6	3	4,461,857,400	01:27:58
TIGER	5	4	4,461,857,400	01:26:40
GOAT	10	0	4,461,857,400	01:14:58
GOAT	8	2	4,461,857,400	01:14:31
GOAT	9	1	4,461,857,400	01:14:20
GOAT	6	4	4,461,857,400	01:12:35
GOAT	7	3	4,461,857,400	01:12:16
GOAT	8	1	4,461,857,400	01:09:39
GOAT	9	0	4,461,857,400	01:08:46
GOAT	7	2	4,461,857,400	01:08:27
GOAT	5	4	4,461,857,400	01:08:14
GOAT	6	3	4,461,857,400	01:08:10
GOAT	4	5	4,461,857,400	00:26:10
GOAT	5	5	4,461,857,400	00:26:02
TIGER	9	2	3,718,214,500	01:19:18
TIGER	11	0	3,718,214,500	01:17:48
TIGER	7	4	3,718,214,500	01:17:09
			:	
			165 tiers hidden	
			:	
			289,836,660,850	81:20:16

Table 4.4: Timing for Solve of Bagh-chal Phase 1

Remaining	Captured	# of Positions	Solve Time (HH:MM:SS)
0	5	1,372,879,200	00:08:09
0	4	514,829,700	00:19:34
0	3	151,420,500	00:05:47
0	2	33,649,000	00:01:22
0	1	5,313,000	00:00:16
0	0	531,300	00:00:02
		2,078,622,700	00:35:11

Table 4.5: Timing for Solve of Bagh-chal Phase 2

4.5 Tic-Tac-Two

4.5.1 Rules

The game is played on a 5×5 (25-slot) board with a 3x3 tic-tac-toe grid frame as shown below. The first player controls 4 X marks and the second player controls 4 O marks. At the start of the game, each player takes turns placing one of their marks on any empty slot contained within the tic-tac-toe grid. Once each player has placed at least two of their marks on the board, they may do one of three things on their turn: (1) place one of their remaining marks on an empty slot within the tic-tac-toe grid, (2) move the tic-tac-toe grid such that it is centered at a slot one space horizontally, vertically, or diagonally away from its original center, or (3) move one of their marks that is already on the board (regardless of whether or not it is within the tic-tac-toe grid) to an empty slot within the grid. The first player to create a three-in-a-row of their own

marks within the tic-tac-toe grid wins. If in a single move the grid has been moved such that it contains both a 3-in-a-row of X and a 3-in-a-row of O, then the game is a tie.

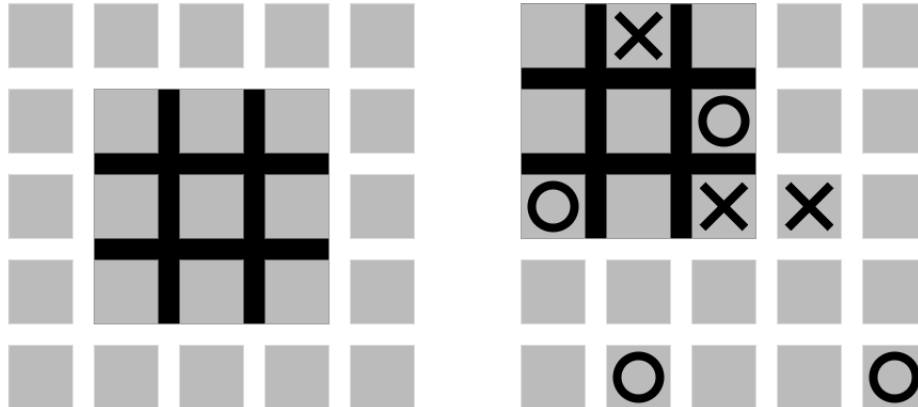

Figure 4.3: Example Tic-Tac-Two Boards

4.5.2 Position Representation

A position is defined by a 2-tuple $(B, \texttt{TURN})$ where B is the state of the board (where the tic-tac-toe grid is centered and where each X and O is) and `TURN` is a bit indicating whose turn it is.

4.5.3 Tier Definition

Each tier is defined as a 2-tuple (N_X, N_O). A position $p = (B_p, \texttt{TURN}_p)$ belongs to a tier $T = (N_{T,X}, N_{T,O})$ if B_p contains $N_{T,X}$ X marks and $N_{T,O}$ O marks. Because placing is mandatory up until both players have 2 marks each on the board, the first four moves result in the tier sequence $(0,0) \to (1,0) \to (1,1) \to (2,1) \to (2,2)$. From this point forward, each tier has two child tiers which result from either player choosing to place one of their pieces on their turn. For example, the child tiers of $(2,3)$ are $(3,3)$ (X places a mark) and $(3,4)$ (O places a mark).

Under this tier definition and rule for generating child tiers of a given tier, there are 13 tiers. For each tier $T_\alpha = (N_{T,X}, N_{T,O}) \in \{(0,0),(1,0),(0,1),(2,1)\}$, the grid is centered at the center of the board and each position in T has the same turn bit, so T_α contains all possible ways of arranging $N_{T,X}$ X marks and $N_{T,O}$ O marks among the 9 slots within the grid. For every other tier $T_\beta = (N_{T,X}, N_{T,O})$, T_β contains every combination of (1) possible grid centers (of which there are 9), (2) possible arrangements of $N_{T,X}$ X marks and $N_{T,O}$ O marks among the 25 board slots, and (3) possible states of the turn bit. T thus contains at most $9 \times \dfrac{25!}{N_{T,X}!N_{T,O}!(25 - N_{T,X} - N_{T,O})!} \times 2$ reachable positions. An upper bound on the number of positions calculated by adding up all the possible states per tier is 2,148,349,834 positions. However, some positions are equivalent under symmetry. There are 8 board symmetry transformations that were utilized in our solve – rotating or reflecting the board yields a symmetric position.

4.5.4 Results

The first player can tie in 14 moves. The solve took 5.58 hours. The (3,4) and (4,3) tiers each required the most memory to solve – it involved keeping track of the values of the values of 302,841,000 current tier positions and 1,362,784,500 child tier positions. The size of the gzipped database directory is 0.57 GiB. The Tic-Tac-Two interface is discussed in Chapter 6.

Number of X's	Number of O's	# of Positions	Solve Time (HH:MM:SS)
4	4	1,362,784,500	03:24:31
3	4	302,841,000	00:49:58
4	3	302,841,000	00:49:44
3	3	63,756,000	00:11:56
2	4	47,817,000	00:07:46
4	2	47,817,000	00:07:37
2	3	9,563,400	00:01:45
3	2	9,563,400	00:01:45
2	2	1,366,200	00:00:15
2	1	252	00:00:00
1	1	72	00:00:00
1	0	9	00:00:00
0	0	1	00:00:00
		2,148,349,834	05:35:17

Table 4.6: Timing for Solve of Tic-Tac-Two

Remoteness	Win	Lose	Tie	Total
0	24,747,048	25,065,288	79,224	49,891,560
1	582,627,412		46,944	582,674,356
2		40,664,176	556,848	41,221,024
3	261,355,240		1,298,280	262,653,520
4		66,505,040	3,274,056	69,779,096
5	203,050,792		15,635,664	218,686,456
6		50,659,728	62,270,012	112,929,740
7	88,505,734		150,960,064	239,465,798
8		26,287,608	213,837,300	240,124,908
9	32,870,732		186,093,200	218,963,932
10		7,656,904	63,490,258	71,147,162
11	7,901,376		15,601,536	23,502,912
12		1,711,376	3,370,482	5,081,858
13	2,081,320		720,000	2,801,320
14		426,264	149,376	575,640
15	544,168		35,112	579,280
16		114,696	9,904	124,600
17	174,144		2,384	176,528
18		40,864	160	41,024
19	83,664		16	83,680
20		23,304		23,304
21	70,024			70,024
22		18,848		18,848
23	48,560			48,560
24		25,064		25,064
25	27,592			27,592
26		13,464		13,464
27	13,168			13,168
28		4,016		4,016
29	5,888			5,888
30		2,064		2,064
31	1,328			1,328
32		480		480
33	432			432
34		704		704
35	176			176
36		336		336
∞			7,258,968	7,258,968
Total	1,204,108,798	219,220,224	724,689,788	2,148,018,810

Table 4.7: Tic-Tac-Two Value-Remoteness Counts (Without Symmetry)

4.6 Topitop

4.6.1 Rules

There are 4 different types of building components. In total, there are ten components: 2 B, 2 R, 4 S, and 4 L. The first player may only place components B, S, or L on the board. The second player may only place components R, S, or L on the board.

Remoteness	Win	Lose	Tie	Total
0	1,548,630	1,568,553	5,010	3,122,193
1	36,427,062		2,934	36,429,996
2		2,542,743	34,803	2,577,546
3	16,338,501		81,165	16,419,666
4		4,157,899	204,657	4,362,556
5	12,693,142		977,329	13,670,471
6		3,167,430	3,892,452	7,059,882
7	5,533,432		9,436,799	14,970,231
8		1,643,944	13,368,113	15,012,057
9	2,055,480		11,635,742	13,691,222
10		478,872	3,970,881	4,449,753
11	494,063		975,832	1,469,895
12		107,028	210,769	317,797
13	130,116		45,021	175,137
14		26,649	9,339	35,988
15	34,019		2,195	36,214
16		7,172	619	7,791
17	10,889		149	11,038
18		2,556	10	2,566
19	5,231		1	5,232
20		1,460		1,460
21	4,382			4,382
22		1,182		1,182
23	3,037			3,037
24		1,567		1,567
25	1,727			1,727
26		843		843
27	824			824
28		251		251
29	368			368
30		129		129
31	83			83
32		30		30
33	27			27
34		44		44
35	11			11
36		21		21
∞			453,972	453,972
Total	75,281,024	13,708,373	45,307,792	134,297,189

Table 4.8: Tic-Tac-Two Value-Remoteness Counts (Unique Under Symmetry)

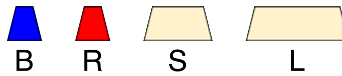

Figure 4.4: Building Components

Over the course of the game, there are 9 possible buildings that can exist on the 3×3 board.

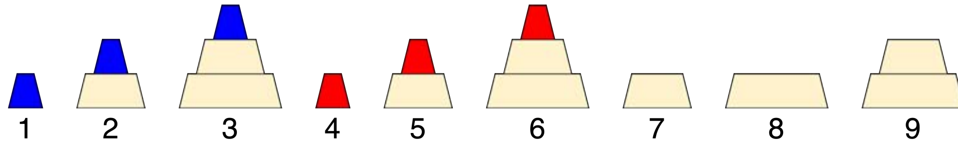

Figure 4.5: Buildings

The first player (blue) may move any building on the board except buildings 4,5, and 6. The second player (red) may move any building on the board except buildings 1,2, and 3. Buildings 7,8, and 9 are referred to as *neutral* buildings. The buildings are created via the following stacking rules.

Valid Stackings: $\texttt{stack}(1,7) \rightarrow 2$, $\texttt{stack}(2,8) \rightarrow 3$, $\texttt{stack}(1,9) \rightarrow 3$, $\texttt{stack}(4,7) \rightarrow 5$, $\texttt{stack}(5,8) \rightarrow 6$, $\texttt{stack}(4,9) \rightarrow 6$, $\texttt{stack}(7,8) \rightarrow 9$. For every other building pair $\beta_1, \beta_2 \in \{1, ..., 9\}$, $\texttt{stack}(\beta_1,\beta_2)$ is undefined. The order of arguments matters, e.g., $\texttt{stack}(1,7) \rightarrow 2$ but $\texttt{stack}(7,1)$ is undefined.

The board starts empty. On a player's turn they may either (1) place one of the remaining building components they own on an empty slot on the board (placing B, R, S, or L on a particular empty slot establishes building 1, 4, 7, or 8 on that slot, respectively) or (2) move one of the buildings β_1 horizontally, vertically, or diagonally one space into either an empty slot or to a slot containing a building β_2 such that $\texttt{stack}(\beta_1, \beta_2)$ is defined, after which the destination slot will hold the building that results from $\texttt{stack}(\beta_1, \beta_2)$. If a player moves a neutral building to an empty slot, then the opponent on their next turn may not move that same neutral structure back to its original slot (it may be moved back on a following turn; just not on the turn immediately after). If a player is unable to make a legal move, they pass their turn. A building may never be taken apart. Player 1 wins once two of building 3 has been created, and the second player wins once two of building 6 has been created.

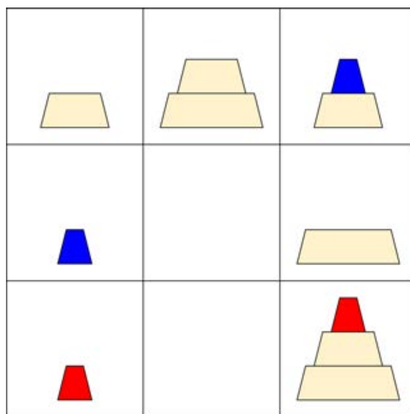

Figure 4.6: Example Topitop Board

4.6.2 Position Representation

A position is defined by a 3-tuple $p = (B, \texttt{TURN}, \texttt{illegalUndo})$ is sufficiently defined by the board B, whose turn it is $\texttt{TURN}$, and the slide of a neutral piece that may not be immediately undone $\texttt{illegalUndo}$. B contains information about the buildings currently on the board and how they are arranged. From the existing structures one can deduce how many of each component remains to be placed.

4.6.3 Tier Definition

Each tier is defined as a 9-tuple $(c_1, c_2, c_3, c_4, c_5, c_6, c_7, c_8, c_9)$. A position $p = (B, \texttt{TURN}, \texttt{illegalUndo})$ belongs to $T = (c_1, c_2, c_3, c_4, c_5, c_6, c_7, c_8, c_9)$ if on B there exist exactly c_j of building j for all $j \in \{1, ..., 9\}$. In other words, two positions belong to the same tier if they have the same set of buildings, the same count of each building, and the same count of each remaining component. In T, there may exist positions in which one can place a component on an empty slot, which means that some of T's child tiers may be tiers with incremented values of c_1, c_4, c_7, and c_8. The other child tiers are results of the stacking rule which are best demonstrated by example: if $c_1 > 0$ and $c_7 > 0$, then one child of the child tiers would be a tier with decremented c_1 and c_7 and incremented c_2.

Under this tier definition and the rule for generating child tiers of a given tier, there are 2231 tiers. Each tier $T = (c_1, c_2, c_3, c_4, c_5, c_6, c_7, c_8, c_9)$ contains all states formed via every combination of (1) all possible $\texttt{illegalUndo}$ values, of which there are 41 (40 ways to move from one slot to an adjacent slot and also a null $\texttt{illegalUndo}$ if a prior move did not involve sliding a neutral building), (2) all possible turn bits, and (3) all possible arrangements of c_1 of building 1, c_2 of building 2,..., c_9 of building 9 among the 9 board slots. T thus contains at most $41 \times 2 \times \dfrac{9!}{\left(\prod_{j=1}^{9} c_j!\right)\left(9 - \sum_{j=1}^{9} c_j\right)!}$ reachable positions.

An upper bound on the number of positions calculated by adding up all possible states across all tiers is $2,972,507,380$ positions. However, some positions are equivalent under symmetry. There are 8 board symmetry transformations that were utilized in our solve – rotating or reflecting the board yields a symmetric position.

4.6.4 Results

The game is a win in 31 moves. The solve took 3.04 hours. The $(1,1,1,2,1,1,1,0,0)$ tier required the most memory to solve (not a unique maximum) – it involved keeping track of the values of $14,878,080$ current tier positions and $64,471,680$ child tier positions. The size of the gzipped database directory is 0.28 GiB. The Topitop interface is discussed in Chapter 6.

c1	c2	c3	c4	c5	c6	c7	c8	c9	# positions	Solve Time (HH:MM:SS)
1	1	1	1	0	0	1	1	1	14,878,080	00:00:52
1	1	1	0	0	1	1	1	1	14,878,080	00:00:50
1	1	0	1	1	0	1	1	1	14,878,080	00:00:50
1	1	0	0	1	1	1	1	1	14,878,080	00:00:42
1	1	1	1	0	0	1	2	1	14,878,080	00:00:35
1	1	0	1	1	0	1	2	1	14,878,080	00:00:31
1	1	1	0	0	1	1	2	1	14,878,080	00:00:31
1	1	0	1	0	0	1	2	1	7,439,040	00:00:35
1	1	1	0	0	0	1	2	1	7,439,040	00:00:34
1	1	0	1	0	0	2	1	1	7,439,040	00:00:31
:										
2221 tiers hidden										
:										
									2,972,507,380	03:02:20

Table 4.9: Timing for Solve of Topitop

4.7 Future Work

In Nine Men's Morris, Tic-Tac-Two, and Topitop, there are other symmetries that were not utilized in our solve partly due to the fact that GamesmanClassic does not support tier-symmetries. With the tier definitions that I used, color-and-turn symmetries could not be implemented in the current GamesmanClassic framework because these are relations according to which positions from different tiers may be symmetric to each other. An example of a color-and-turn symmetry is in Nine Men's Morris Phase 2: if the piece colors and turn bit are flipped, then the resulting position is symmetric. Future work involves updating GamesmanClassic so that it supports tier-symmetries.

Chapter 5

Image AutoGUI

5.1 Overview

The Image AutoGUI system is a GUI creation system that only requires a developer to, on top of generating the appropriate UWAPI position and move strings for each position query, specify a list of relevant coordinates, SVG information, and a few settings to create a game interface that comes with many of the automatically handled features of the Character AutoGUI (e.g., move button coloring based on move values, hover animations, API calls) but is more user-friendly. While Character AutoGUIs can only render grid-based games and use a limited set of characters to represent pieces, Image AutoGUIs can be used to represent various non-grid-based games and use SVGs for pieces and background images. The idea was first discussed in GamesCrafters meetings with the goal of addressing the limitations of the Character AutoGUI. I later implemented the Image AutoGUI system and it has been used ever since to create several game interfaces on GamesmanUni.

5.2 Example: Achi AutoGUI

We first explain how GamesmanUni renders a position by using the Achi Image AutoGUI as an example. See Explanation of Rules for Various Games for an explanation of the rules of Achi.

5.2.1 Coordinates and SVGs

The Achi interface uses the following data when rendering each position. Some attributes are hidden or refactored for this explanation.

```
{
    "defaultTheme": "basic",
    "themes": {
        "basic": {
            "backgroundGeometry": [100, 100],
            "backgroundImage": "achi/achiboard.svg",
            "centers": [
                [10, 10],
                [50, 10],
                [90, 10],
                [10, 50],
                [50, 50],
                [90, 50],
                [10, 90],
                [50, 90],
```

```
                [90, 90]
            ],
            "entities": {
                "x": { "image": "achi/X.svg", "scale": 15 },
                "o": { "image": "achi/O.svg", "scale": 15 }
            }
        }
    }
}
```

We loosely refer to this as the Image AutoGUI JSON data. `centers` is the coordinate list and `entities` is the entity mapping. An *entity* loosely refers to an element of the position display that moves or changes (e.g., slides, rotates, appears, disappears). In Achi and many of the other interfaces we present, the only entities are pieces.

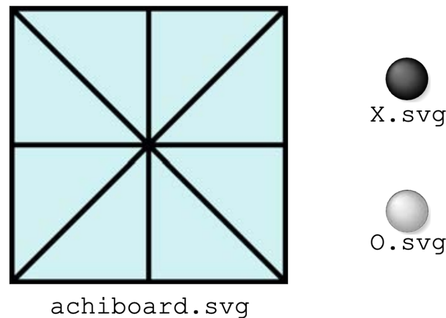

Figure 5.1: There are three SVGs that are used: the background image `achi/achiboard.svg` and two piece images `achi/X.svg` and `achi/O.svg`.

The `backgroundGeometry` defines the coordinate space for the board display, as shown in 5.2.1. In this example, the `backgroundGeometry` is `[100, 100]`, so the top left corner of the board view is treated as coordinate $(0, 0)$ and the bottom right corner of the board view is treated as coordinate $(100, 100)$. There are 9 coordinates in the coordinate list that are used as entity centers.

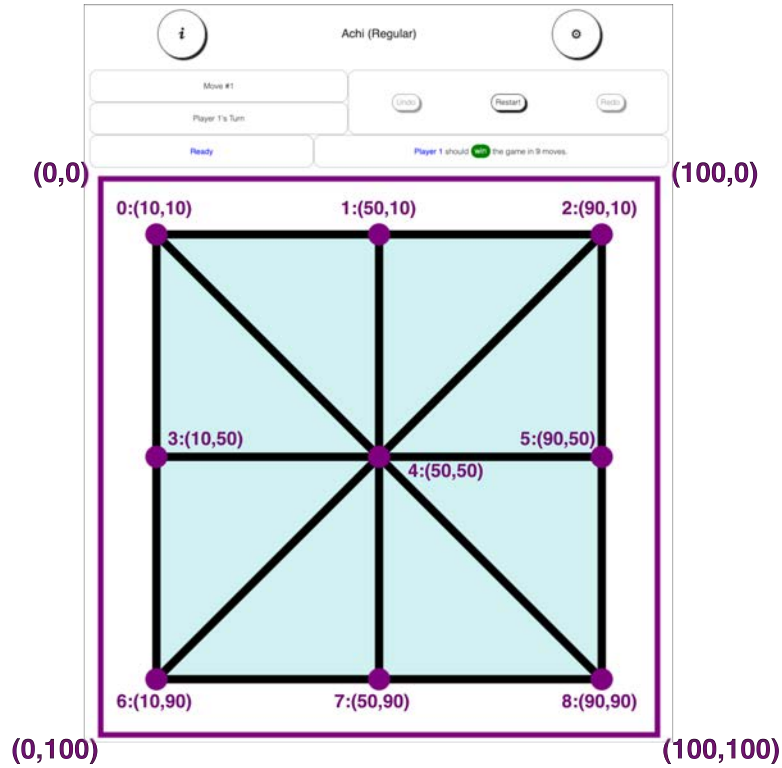

Figure 5.2: Achi Coordinates

5.2.2 Rendering Positions

GamesmanUni first renders the background image (if one is specified), then the entities, then the foreground image (if one is specified), then the move buttons.

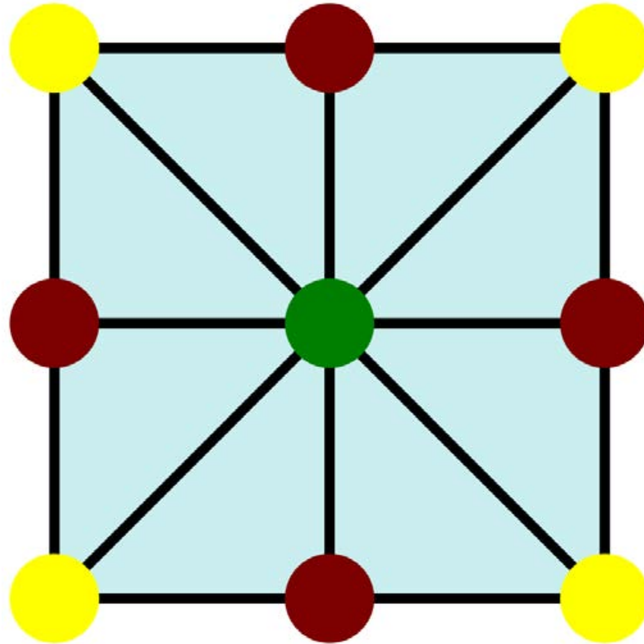

Figure 5.3: Achi Initial Position

Our first example (Figure 5.2.2) is the initial position. In the initial position, the first player chooses one of 9 intersection points to place a piece. The user clicks one of the circle-shaped move buttons to specify the piece placement to perform. As in the Character AutoGUI, each move button is colored according to the move's game-theoretic value (Green=Win, Yellow=Tie/Draw, Red=Lose).

GamesmanUni sends a request for the Achi initial position to GamesCraftersUWAPI and receives the following response.

```
{
    "position": "R_A_3_3_---------",
    "moves": [
        {
          "move": "A_-_4",
          "moveValue": "win",
          "position": "R_B_3_3_----o----"
        },
        {
          "move": "A_-_8",
          "moveValue": "draw",
          "position": "R_B_3_3_--------o"
        },
        {
          "move": "A_-_7",
          "moveValue": "lose",
          "position": "R_B_3_3_-------o-"
        }
    ]
}
```

Some move objects in the `moves` list and move attributes are omitted for brevity. Three of the nine move objects are shown.

1. The background image is drawn.
2. Rendering Entities: The current position is given by the position string `R_A_3_3_---------`. `R_A_3_3_` is the position string header and `---------` is the entity substring, which tells us which entities to render. In the entity substring, `'-'` indicates that an entity is nonexistent. The first character (index=0) in the entity substring is '-', so at coordinate 0, which is (10, 10), there is no entity drawn. The second (index=1) character in the entity substring is also `'-'`, so nothing is drawn at coordinate 1, which is (50, 10). We do the same for the remaining seven characters in the entity substring.
3. There is no foreground image. Continue to the next step.
4. Rendering Moves: `moves` is a list of move objects. In total there are nine but three are shown in this example. In each move object, the `move` attribute specifies the shape and location of the move button, the `moveValue` attribute specifies the value of the move (for the sake of coloring the move button accordingly), and the `position` attribute indicates which position to send a request for and render next, assuming this move button corresponding to this move is clicked.

   In the first move object, `move` is `A_-_4`. "`A`" and "`-`" indicate that this move uses the default move button shape, which is a circle. We explain custom move button shapes further in the next subsection. `4` indicates that this move button is to be centered at coordinate 4, which is (50, 50). `moveValue` is `win`, so this move button is to be colored green. If this move button is clicked, then we load the position whose position string is `R_B_3_3_----o----` next. In the Achi board, this move button is the green button in the center of the board.

   We do the same for the remaining move objects. The second and third move objects shown correspond to move buttons that will be centered at coordinate 8 and coordinate 7 respectively and colored yellow and red respectively.

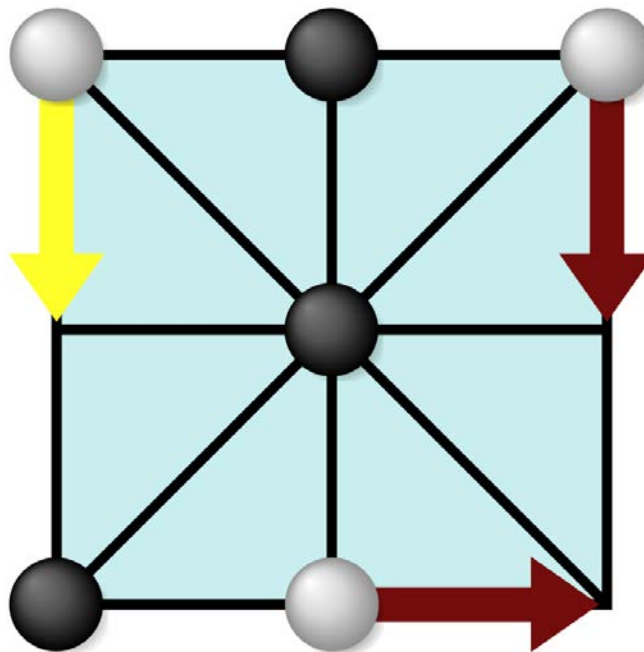

Figure 5.4: Achi Middlegame Position

Our second example (Figure 5.2.2) is a middlegame position, in which all pieces have been placed and now they are being moved around the board. The user clicks on of the arrow move buttons to specify

the move to perform. In the previous position, a move button was clicked to load the position with position string `R_A_3_3_oxo-x-xo-`. GamesmanUni receives the following response for the current position, `R_A_3_3_oxo-x-xo-`.


```
{
    "position": "R_A_3_3_oxo-x-xo-",
    "moves": [
        {
          "move": "M_0_3",
          "moveValue": "draw",
          "position": "R_B_3_3_-xoox-xo-"
        },
        {
          "move": "M_7_8",
          "moveValue": "lose",
          "position": "R_B_3_3_oxo-x-x-o"
        },
        {
          "move": "M_2_5",
          "moveValue": "lose",
          "position": "R_B_3_3_ox--xoxo-"
        }
    ]
}
```


We follow the same steps, again referencing the coordinate list and entity mapping.

- Rendering Entities: The entity substring is `oxo-x-xo-`. At index 0, the character is `'o'`, so the SVG corresponding to `'o'`, which is `achi/O.svg`, should be drawn, centered coordinate 0. At index 1, the character is `'x'`, so the SVG corresponding to `'x'`, which is `achi/X.svg`, is drawn, centered at coordinate 1. Continuing on, `achi/O.svg` is drawn at coordinates 2 and 7, and `achi/X.svg` is drawn at coordinates 4 and 6. The `scale` for these pieces is `15` and these are square SVGs, so each entity image will take up 15/100 (100 is specified by the `backgroundGeometry`) of the width and height.
- Rendering Moves: There are three available moves at this position. In the first move object, `move` is `M_0_3`. `M` indicates that the move button is arrow-shaped. This arrow is to be drawn from coordinate 0 to coordinate 3. The `moveValue` is `draw` so this arrow is yellow. In 5.2.2, this move button is the arrow pointing at the center left from the top left. We follow the same procedure for displaying the other two moves.

5.3 General Structure of Image AutoGUI Data

```
{
    "defaultTheme": <name of default theme>,
    "themes": {
        <name of theme1>: {
            "backgroundGeometry": [<width>, <height>],
            "backgroundImage": <path to background image>,
            "foregroundImage": <path to foreground image>,
            "piecesOverArrows": <true/false>,
            "defaultMoveTokenRadius": <radius, relative to backgroundGeometry>,
            "arrowThickness": <thickness, relative to backgroundGeometry>,
            "lineThickness": <thickness, relative to backgroundGeometry>,
            "centers": [ [<x0>,<y0>], [<x1>, <y1>], [<x2>, <y2>], [<x3>, <y3>], ... ]
```

```
            "entities": {
                <char1>: {
                    "image": <path to entity image>, "scale": <image scale>
                },
                <char2>: {
                    ...
                }
                ...
            }
        },
        <name of theme2>: {
            ...
        },
        ...
    }
}
```

The general structure of the Image AutoGUI data is shown above. Each variant of a game has its own Image AutoGUI data. This data consists of multiple *themes*. Different themes can be included to give users the option to switch between different game board appearances (e.g., different board coloring or piece designs in chess). Each theme consists of a coordinate list, an entity mapping, SVG information, and various settings. `centers` lists all the coordinates that are used in the interface (whether they are entity centers or coordinates used for positioning move buttons). `entities` defines the mapping of alphanuemric characters that appear in the entity substring to SVGs. `backgroundImage` and `foregroundImage` define the background image and foreground image path, respectively. Miscellaneous settings are:

- `piecesOverArrows`: whether arrow buttons should be drawn on top of or underneath entity SVGs.
- `defaultMoveTokenRadius`: the radius (relative to `backgroundGeometry`) of any circle move buttons.
- `arrowThickness`: the width (relative to `backgroundGeometry`) of the arrow stem of any arrow move buttons.
- `lineThickeness`: the width (relative to `backgroundGeometry`) of any line move buttons.

There are three types of move buttons. *A-type* move buttons (movestring: `A_<shape>_<center>`) are centered at a given `center` coordinate. An A-type move button has either a default circle shape (`shape='-'`) or a custom shape as specified by an SVG (`shape` is a character corresponding to the desired SVG in the `entities` mapping). *M-type* move buttons (movestring: `M_<from>_<to>`) are arrow-shaped move buttons pointing from the `from` coordinate to the `to` coordinate. *L-type* move buttons (movestring: `L_<p1>_<p2>`) are line-shaped move buttons with endpoints `p1` and `p2`. All move buttons have a pulsing hover animation.

In section 5.5 New Interfaces Using Image AutoGUI we show different interfaces that have been created in the Image AutoGUI system using various AutoGUI settings and move buttons.

5.4 Designing an Image AutoGUI for a Game

When the Image AutoGUI was first developed, one of the first goals was to modify existing Character AutoGUIs to use the Image AutoGUI system. It may be helpful for one to create a Character AutoGUI for their game before incorporating images; however, with more experience, one can skip the Character AutoGUI step entirely.

Typically, one follows the design process below to create an Image AutoGUI.

- In the interface, what are all the entities? What objects change or move in the interface?

- What are all the possible move buttons that may appear on the interface?
- What SVGs are needed for the background, foreground, and entities?
- What coordinates will be used? Coordinates are needed to specify the centers of entities and move tokens. They are also needed for the endpoints of arrow move buttons and line move buttons.
- Think about other settings (e.g., whether pieces should be drawn above or below arrow move buttons, how thick arrow move buttons should be, etc.).

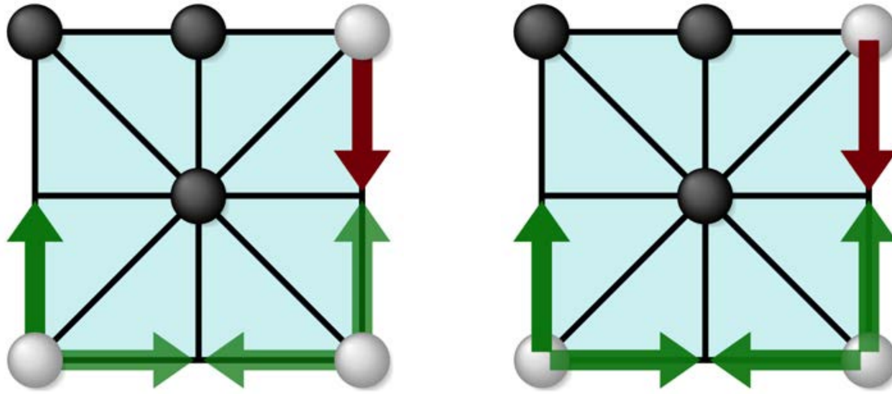

Figure 5.5: Example: Toggling the `piecesOverArrows` setting.

Assign each entity and custom-shape move button a unique alphanumeric character and assign each coordinate an ID. Then the Image AutoGUI JSON data is created accordingly and the GamesCraftersUWAPI server is updated to return this data when GamesmanUni requests information on how to render the game.

5.5 New Interfaces Using Image AutoGUI

Following are Image AutoGUIs that I worked on in addition to the Achi interface. Unless otherwise specified, the rules of any of the following games that are lesser known are explained in Explanation of Rules for Various Games. For each of the following games, we first show examples of how its interface looks, then briefly discuss each part of its interface (e.g., what kinds of move buttons are used). We also individually show the coordinates and entities that are used so that readers can gain insight into how one might design their own Image AutoGUI.

5.5.1 Bagh-Chal

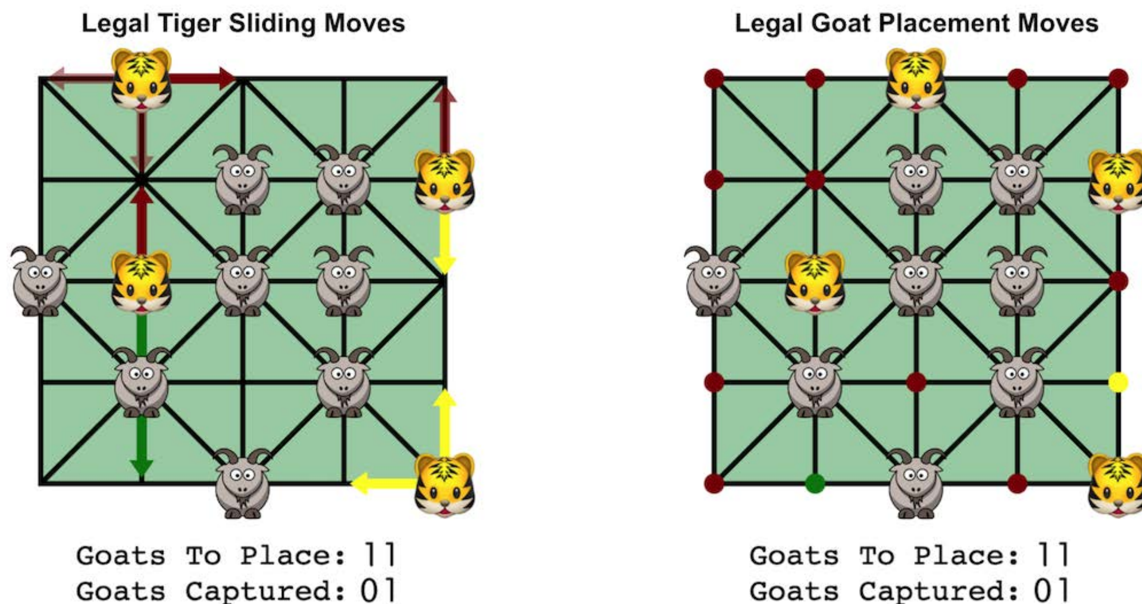

Figure 5.6: Bagh-Chal Interface

For information on the rules and solving, see Chapter 4. The entities are not only the tiger and goat pieces but also the digits of the remaining-goats and captured-goats counters. Arrow move buttons are used for moving tigers and goats. Circle (default) move buttons are used for placing goats.

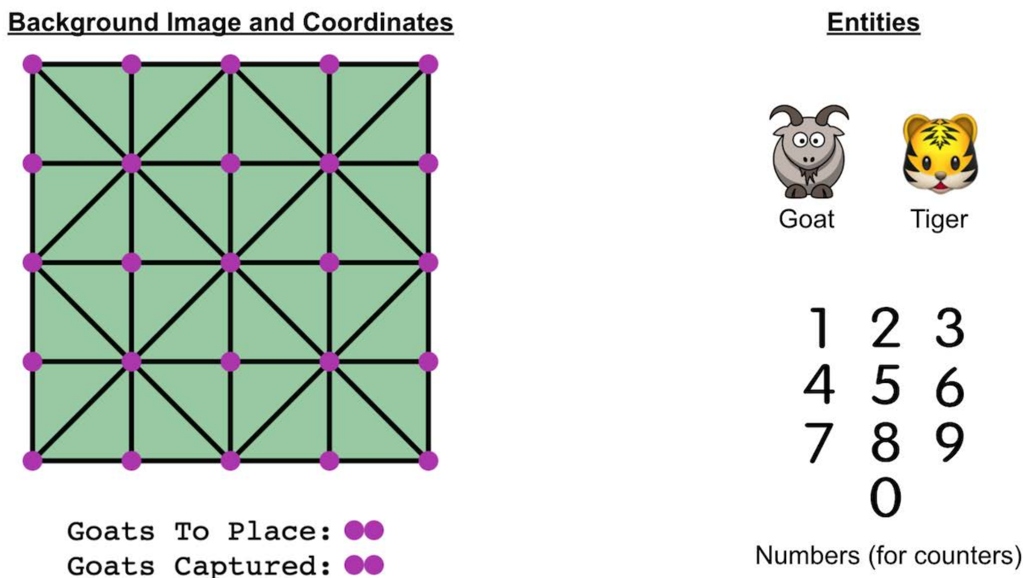

Figure 5.7: Bagh-Chal AutoGUI Design

5.5.2 Chess Endgame

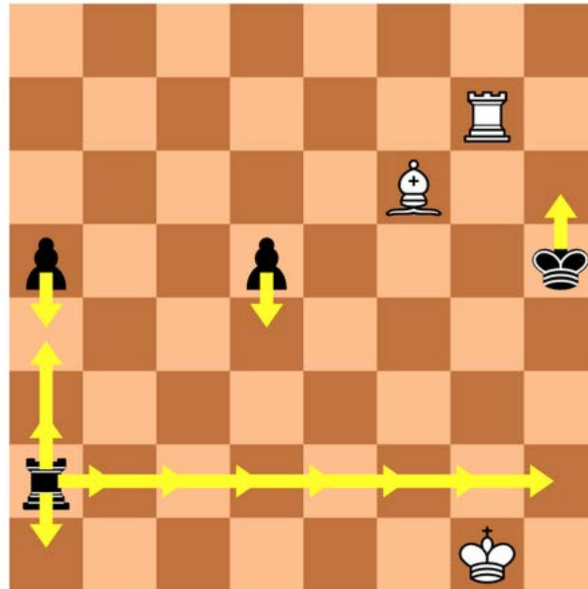

Figure 5.8: Chess Interface

Here, there are 64 center coordinates. The only entities are the chess pieces. Piece movements are executed by clicking arrow move buttons.

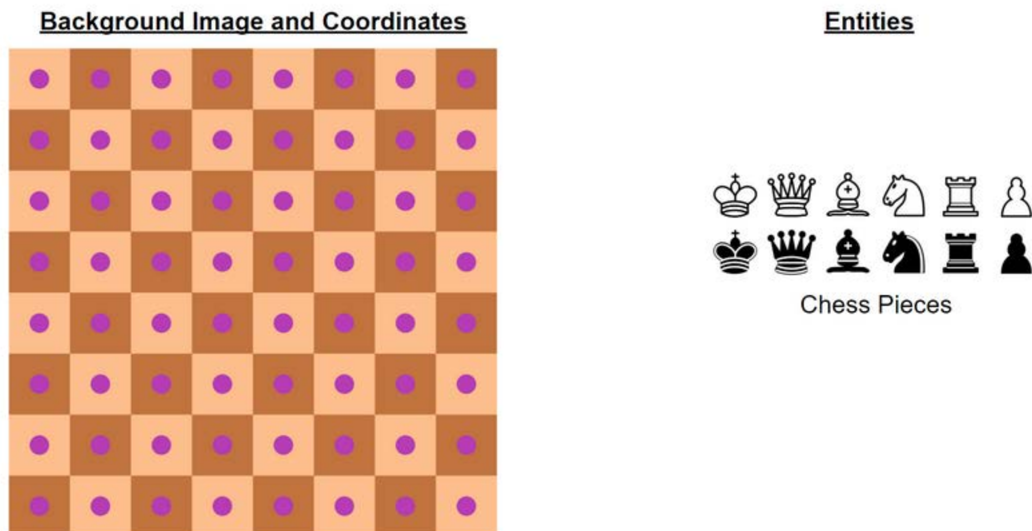

Figure 5.9: Chess AutoGUI Design

With the same coordinates but different SVGs for the background and pieces, we can add another theme to chess, as shown in 5.10. The user can choose which theme to use in the gameplay options menu.

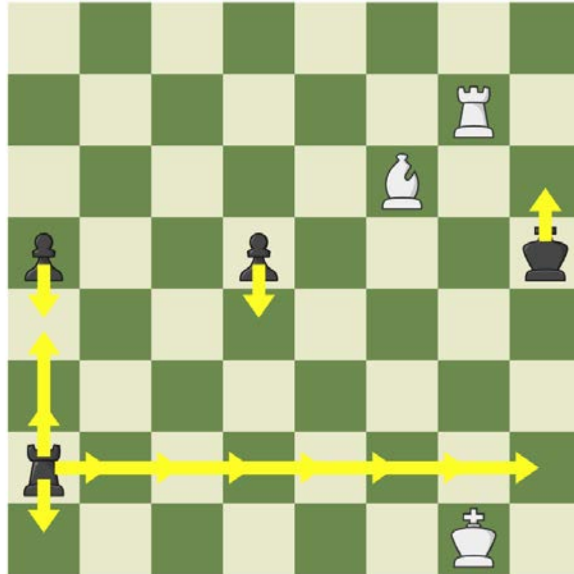

Figure 5.10: Another Theme for Chess

5.5.3 Chomp

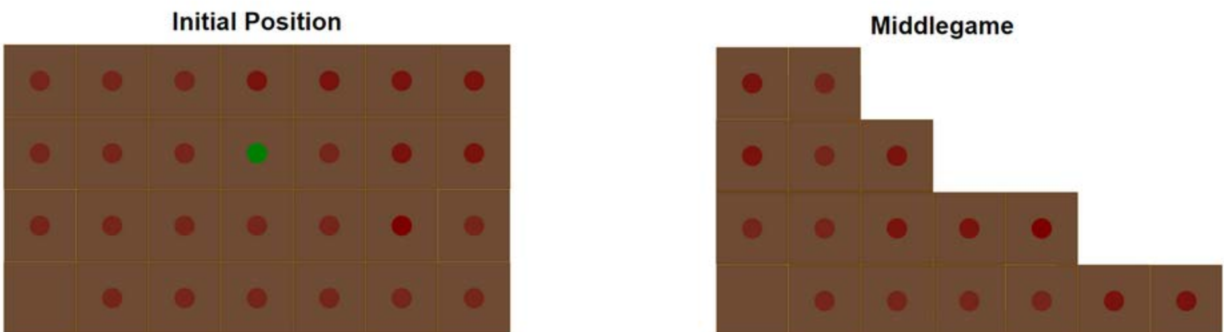

Figure 5.11: Chomp Interface

The entities are the squares of the chocolate bar. A circle move button is clicked to indicate the bottom-left corner of a rectangular removal.

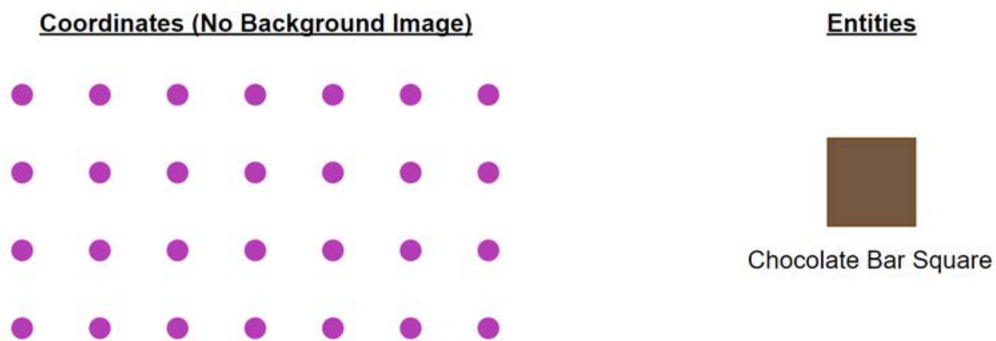

Figure 5.12: Chomp AutoGUI Design

5.5.4 Connect 4

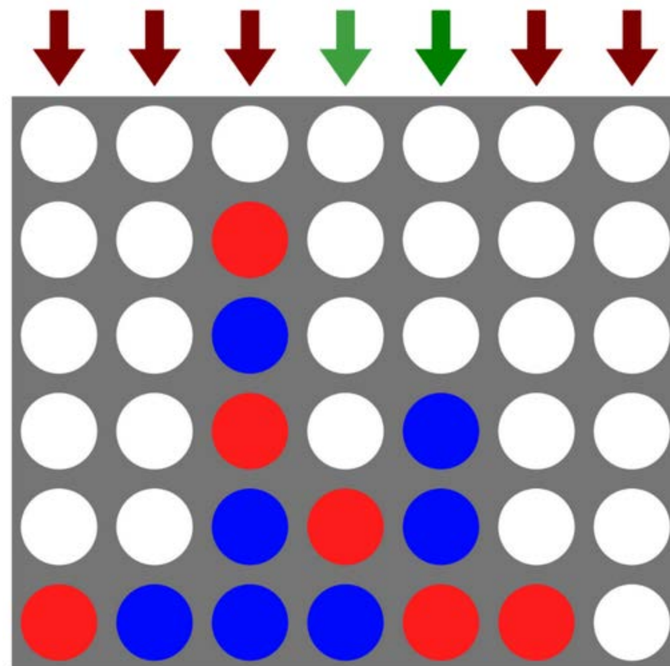

Figure 5.13: Connect 4 Interface

This is an example that uses a foreground image – the Connect 4 frame. Technically, without animation, the frame could instead be a background image with the pieces' scales adjusted to be the same size as the frame holes, but with animation, the pieces should be temporarily and partially obscured by the frame as they are dropped from the top, so a foreground image is necessary to represent the frame. Two endpoint coordinates above the frame are needed to draw each downward-facing arrow button, as shown in Figure 5.14. To place a piece in a particular column, one clicks the appropriate arrow button. The only entities are the pieces themselves.

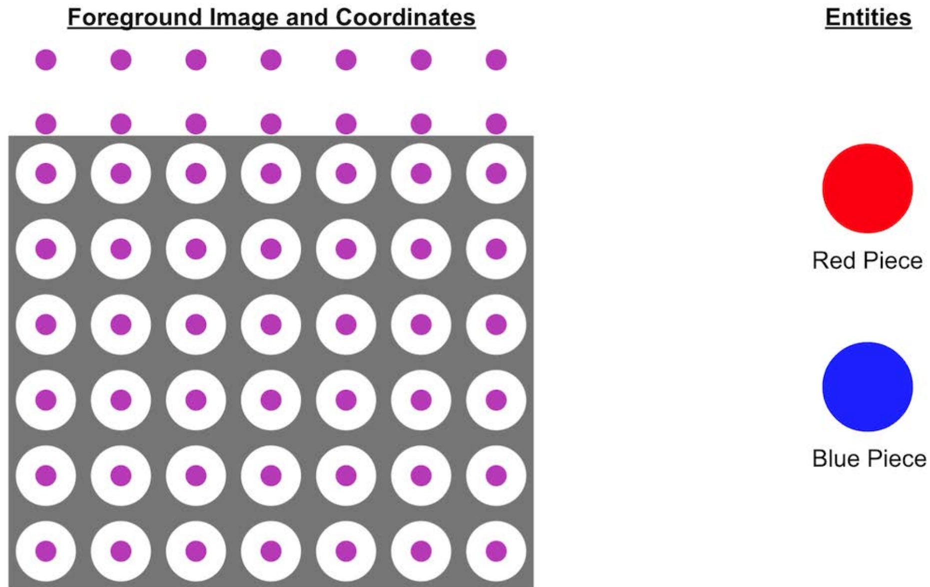

Figure 5.14: Connect 4 AutoGUI Design

5.5.5 Dawson's Chess

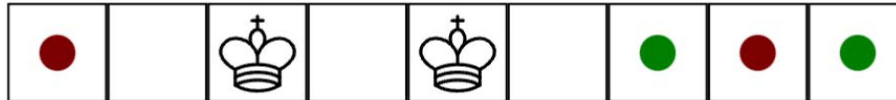

Figure 5.15: Dawson's Chess Interface

This is an example of how one can generalize the coordinate lists to different variants of a game. Different board lengths make for different variants of Dawson's chess [21]. To avoid creating various grid background images for each variant, we instead choose to have no background and have the entities be an occupied cell or an empty cell. When a move button is clicked, the empty cell is replaced with an occupied cell. We can calculate the appropriate center coordinates for a given board dimension.

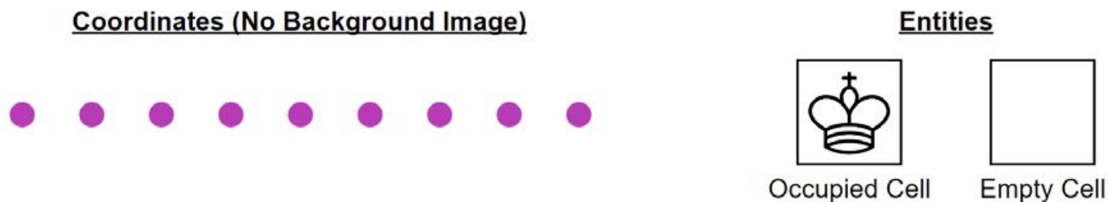

Figure 5.16: Dawson's Chess AutoGUI Design

5.5.6 Dodgem

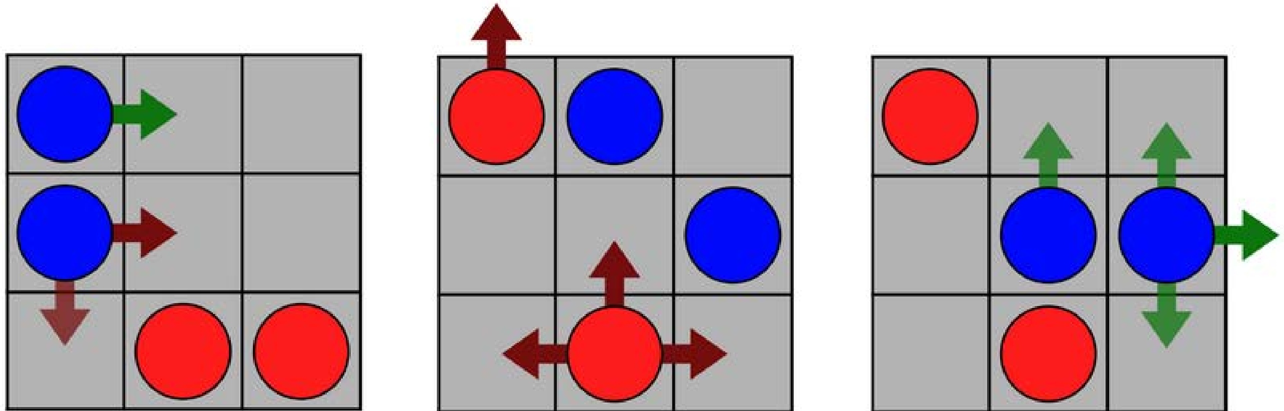

Figure 5.17: Dodgem Interface

Arrow move buttons indicate where to slide a piece. The coordinates that exist outside the board are used for the endpoints of arrows that point away from the board.

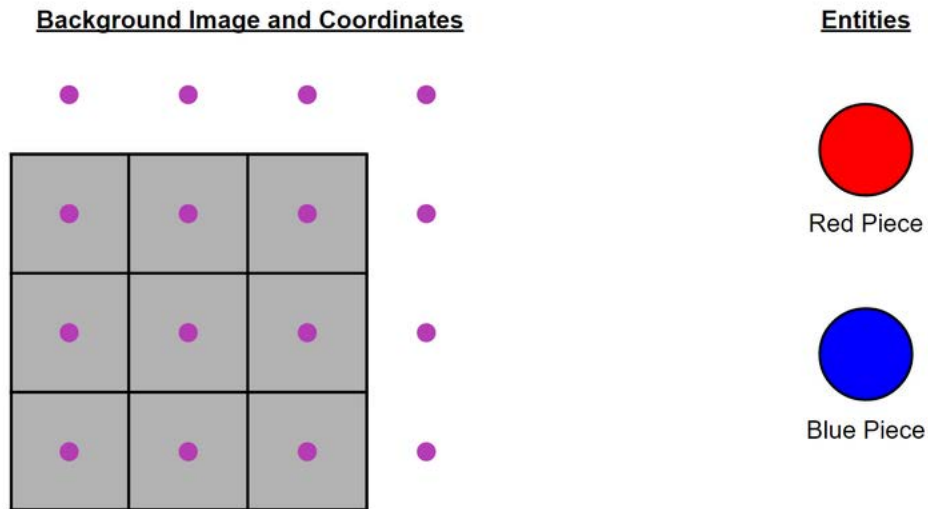

Figure 5.18: Dodgem AutoGUI Design

5.5.7 Mū Tōrere

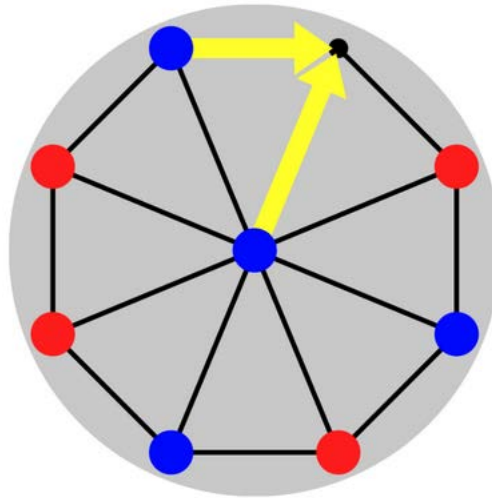

Figure 5.19: Mū Tōrere Interface

The only entities are the eight pieces. Arrow move buttons are used to slide pieces.

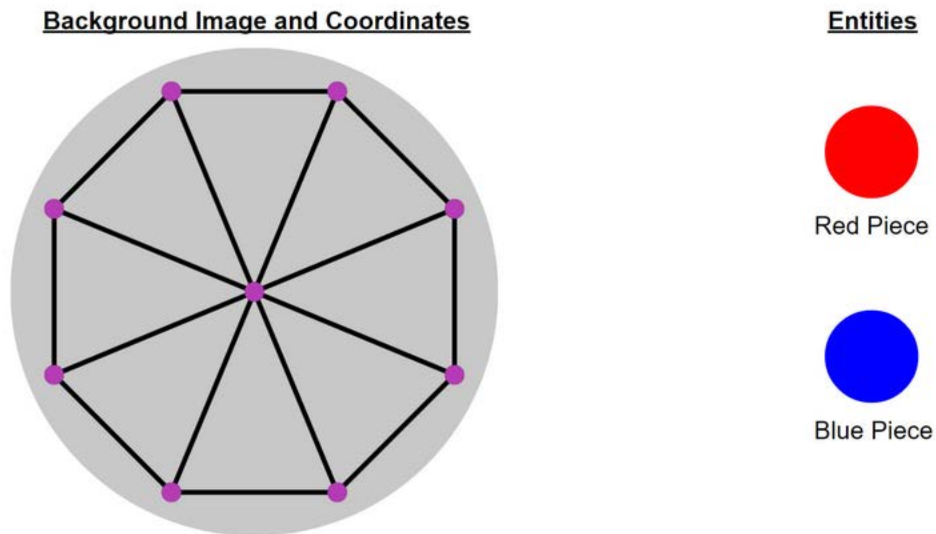

Figure 5.20: Mū Tōrere AutoGUI Design

5.5.8 QuickCross

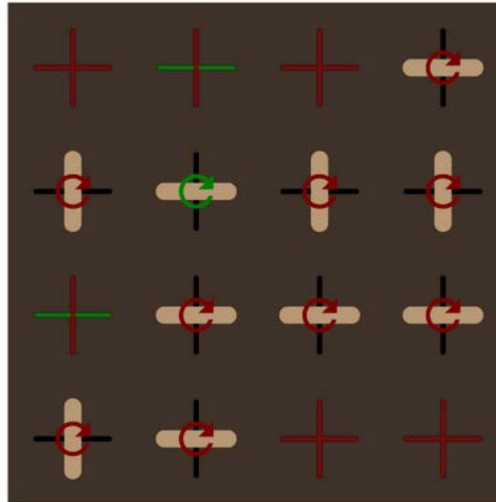

Figure 5.21: QuickCross Interface

The QuickCross AutoGUI makes use of custom move button shapes and line moves. The only entities are the pieces. When a particular cross is empty, a vertical line move button and horizontal line move button are displayed on the cross. If each is clicked, a piece is placed horizontally or vertically, respectively. If a particular cross contains a piece, then a custom-shape move button (circular arrow) appears on top of it which, when clicked, changes the orientation of the piece. There are five points for each cross: the point at the cross center is used as the center for an entity or circular arrow move button, and the other four points are used as endpoints of line move buttons.

We provide an SVG of an all-black circular arrow for the custom move button. AutoGUI applies a filter to the SVG to color it green, yellow, or red depending on the value of the move. Custom move buttons are technically square because any transparency in the SVG can still be clicked, thus clicking the transparency in the center of a circular arrow move button still performs the move.

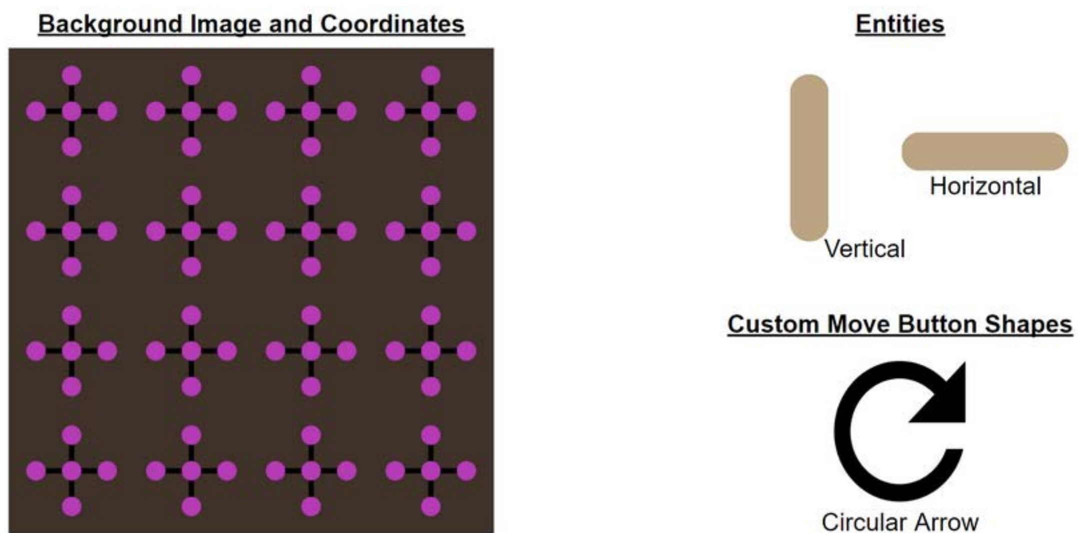

Figure 5.22: QuickCross AutoGUI Design

5.5.9 Shift-Tac-Toe

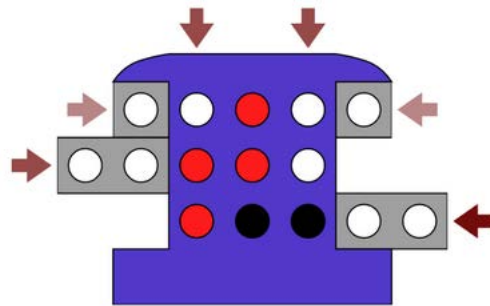

Figure 5.23: Shift-Tac-Toe Interface

The shift-tac-toe interface is one of the more complex examples. The entities are not only the pieces but also the three sliders. Like in Connect 4, a foreground image is used. The left and right-facing arrow buttons are used to move the sliders. The downward-facing move buttons are used to specify which column to drop a piece in. The nine coordinates

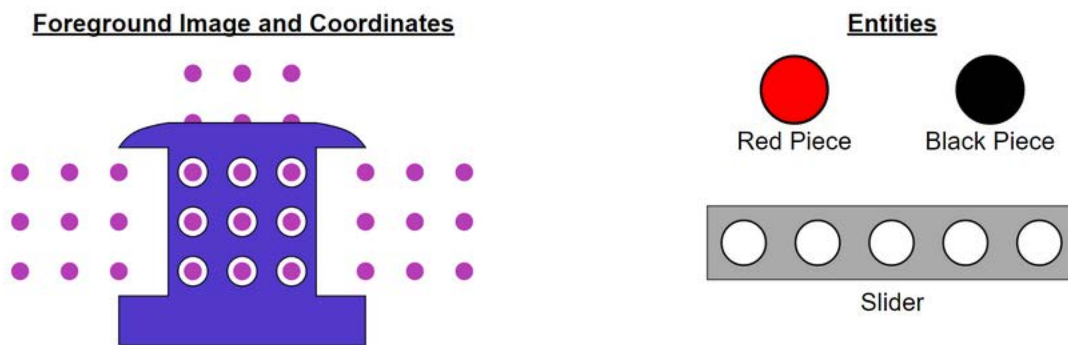

Figure 5.24: Shift-Tac-Toe AutoGUI Design

5.5.10 Snake

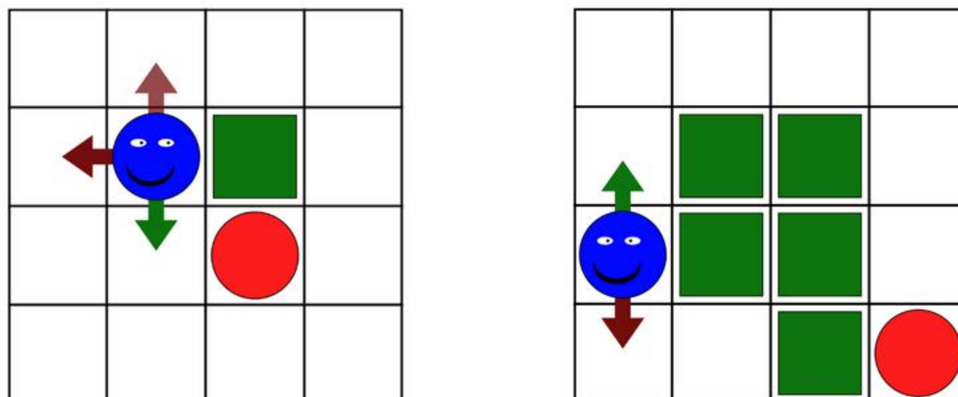

Figure 5.25: Snake Interface

The entities are the head, body, and tail components of the snake. An animated SVG is used for the head.

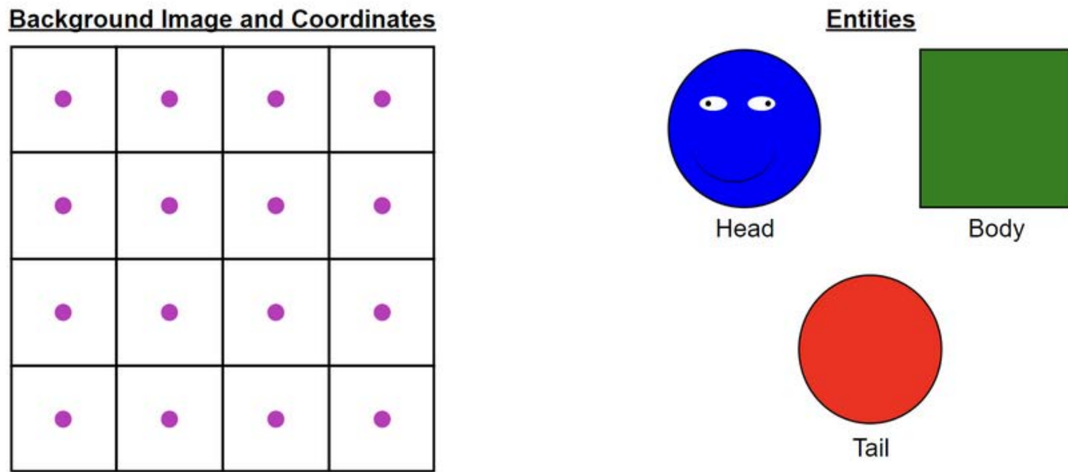

Figure 5.26: Snake AutoGUI Design

5.5.11 Tac Tix

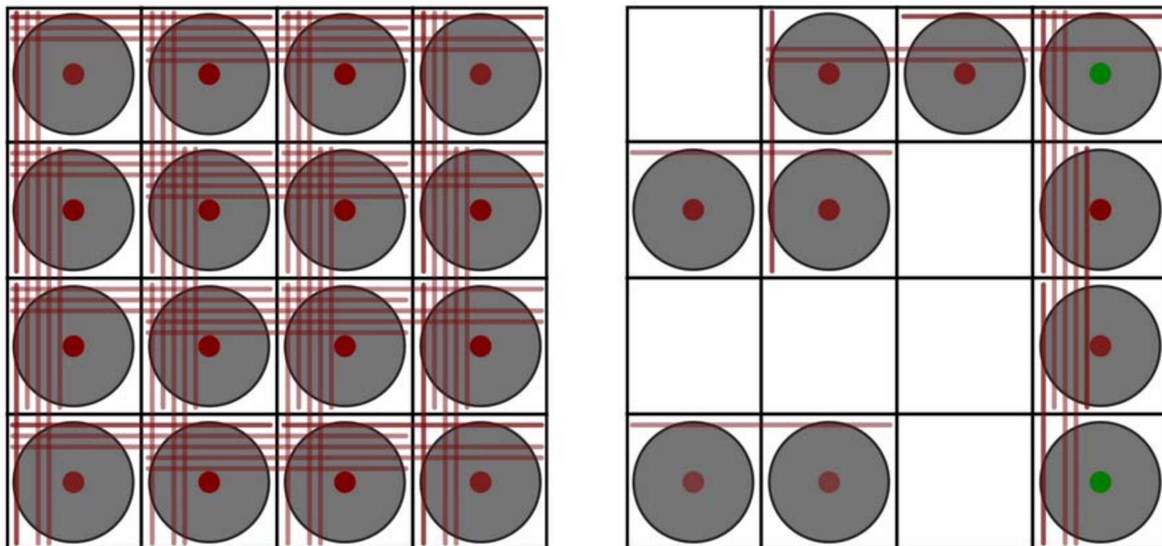

Figure 5.27: Tac Tix Interface

This interface was adapted from the Tcl/Tk version of Tac Tix in GamesmanClassic and makes use of line move buttons. The entities are the coins. When a line move button is clicked, the coins that it crosses over are removed.

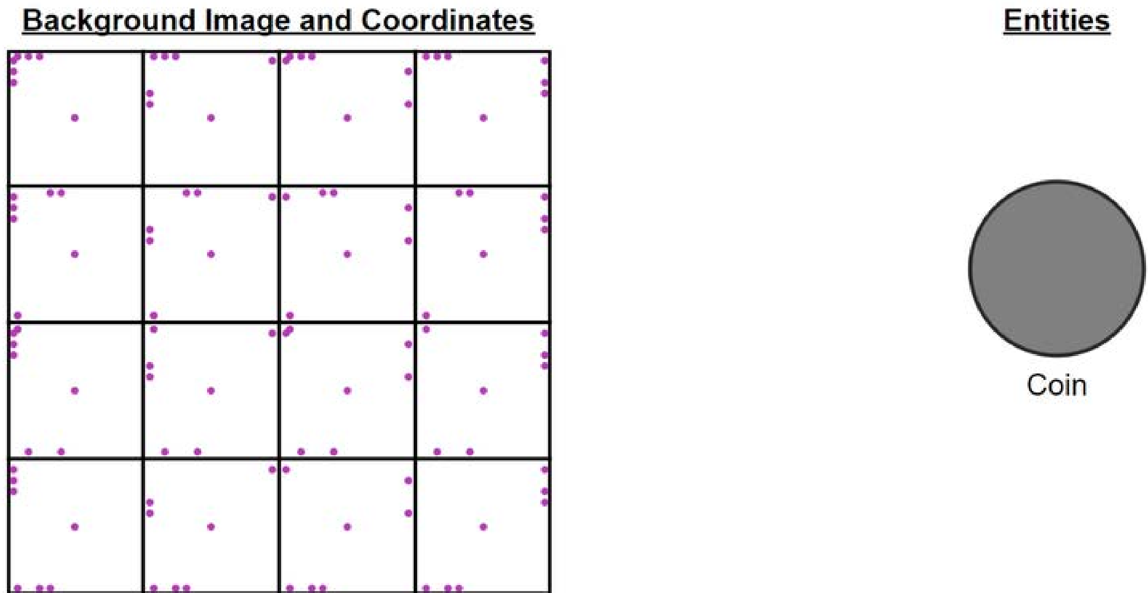

Figure 5.28: Tac Tix AutoGUI Design

5.5.12 Toot-and-Otto

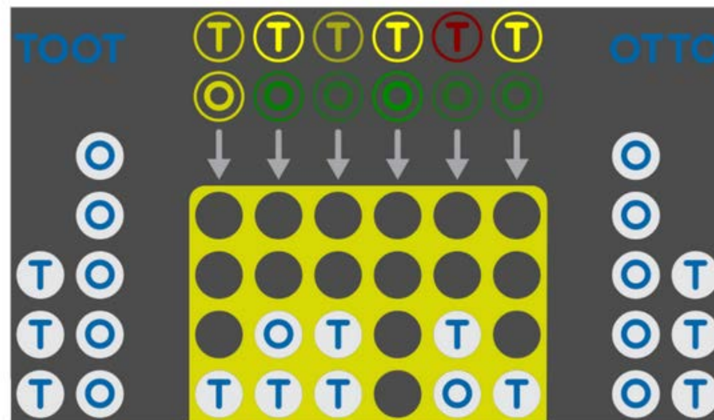

Figure 5.29: Toot-and-Otto Interface

The entities are the "T" and "O" pieces, whether they have been placed or whether they are in reserve. Custom-shape move buttons (T and O) are used which, when clicked, specify which piece to use and which column to drop it in.

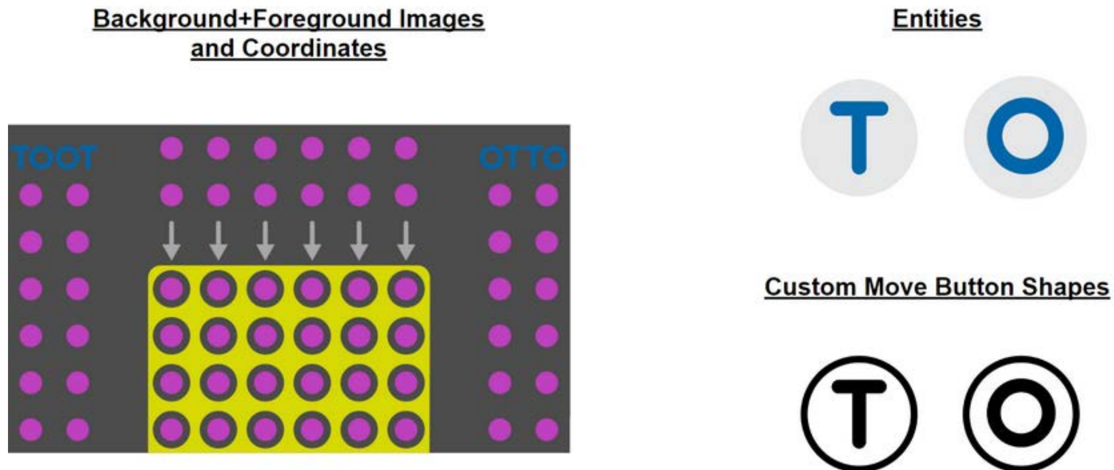

Figure 5.30: Toot-and-Otto AutoGUI Design

5.6 Future Work

Anyone working on GamesmanUni interfaces is encouraged to think about what an "ideal" value-moves interface for a specific game they are working on would look like and compare that vision to what the best interface for that game would look like if one were to be created from only the features that the current AutoGUI supports. Any feature in the "ideal" interface that is not currently supported by AutoGUI should be considered a future feature of AutoGUI as long as it is general enough for a large class of games.

For example, there are many games for which it is appropriate to use curved arrow buttons (perhaps if it indicates a curved trajectory that a piece moves along). Currently AutoGUI only supports straight arrow buttons, so having support for curved arrows would be an improvement. One can indeed "hardcode" curved arrows via custom-shape move buttons as done in QuickCross, but it would be undesirable to create an SVG for every type of arrow for a hypothetical game interface involving multiple curved arrow buttons varying in length and curvature.

Another example: One common pattern that we notice in some games is the existence of counters. For example, in Bagh-Chal, there are counters for goats left to place and goats captured. In Toot-and-Otto, there are counters for how many of each "T" piece and "O" piece each player has in their reserve. Counters are used for games that involve scoring as well. The current way of displaying counters is to display number SVGs at particular coordinates on the game board. It would be easier for anyone working on an AutoGUI if there exists an "automatic" system of creating a standardized dashboard to display counts so that they need not decide how to position number SVGs or any icons or text associated with counts.

In addition, anyone who has gone through the current AutoGUI workflow should note which parts of the process of implementing an AutoGUI can be improved to make creation and testing easier.

The most important next step is support for animation. Ideally an AutoGUI-with-animation system (1) makes it as easy to incorporate animation into an interface with as little work as possible in addition to writing the Image AutoGUI JSON data and (2) supports a large class of animations ranging from simple piece sliding animations to general linear transformations of entities, with support for multi-phase animations and ways to customize how to time animations of different entities.

Chapter 6

Multipart-Move Interfaces

6.1 Motivation

Consider a hypothetical game played on a 6-by-2 grid involving a circle and diamond piece. On a player's turn, they must move *both* pieces to orthogonally adjacent slots.

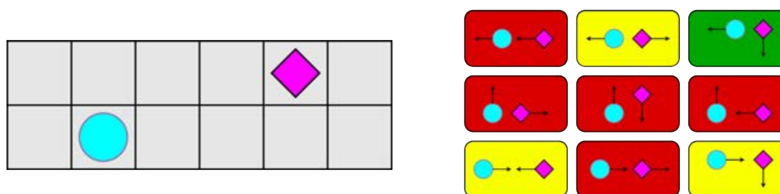

Figure 6.1: A Full-Move-Buttons Interface

One way to create a value-moves interface for this game is to display the state of the board and a list of buttons to the right, with each button corresponding to a unique way to slide the two pieces, as shown in 6.1. The downside of this type of interface is that the user's attention is drawn away from the board when they look at the move buttons [22]. Additionally, if this type of interface were used for a variant of game played on a larger board (e.g., 100-by-100), in which each piece can slide to any location on the board, then there would be a large number of move buttons to choose from.

We can instead create a multipart-move interface to address these two issues. Each move will be specified by two part-moves. For the first part-move, the user clicks one of the arrow buttons displayed by the circle piece to specify where it should move. Doing so loads an **intermediate state** that shows the circle piece at its new location and arrow buttons to specify the second part-move. For the second part-move, the user clicks one of the arrows displayed by the diamond piece to specify where it should move. All part-move arrows should be colored according to the perfect-play outcome of the game for the current player assuming they click that part-move.

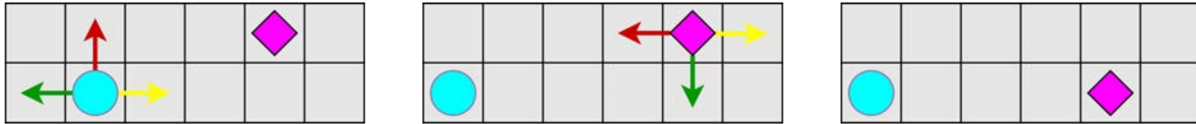

Figure 6.2: Consider the same position p as shown in the left image. Suppose the player whose turn it is at p wishes to move the circle piece left and the diamond piece down. The center image shows the intermediate state after selecting the circle-piece-left part-move. The right image shows the child position that results from then selecting the diamond-piece-down part-move.

We use the term **real position** to refer to actual positions and distinguish them from intermediate states. A full-move is a transition between two real positions. In a multipart-move interface, a full-move is specified by a sequence of **part-moves**.

We developed the **multipart move system** to create interfaces for games, like the one shown in the example, that involve moves that are not "naturally" specifiable by a single click. The goal of the multipart-move system is to allow people, with as little work as possible on their part, to create a multipart-move interface for their solved game. In order to color the part-move buttons, we would need to determine the values of intermediate states; however we wish not to solve and store the values of intermediate states because they exist only for the purpose of the interface and there may be a large number of intermediate states that exist compared to the number of real positions. The multipart-move system automatically handles live-solving for part-moves and intermediate states.

6.2 Multipart-Move Interface Design and Implementation

To take advantage of the multipart move system, a developer is required to write a function that can generate a multipart-move graph for any position. They need to consider what intermediate states and part-moves the interface should display. This graph must be finite and acyclic and will consist of (1) real positions and intermediate states as nodes and (2) part-moves as edges. The root node is the current real position p and the sink nodes are the real child positions of p. The graph is fed into the multipart move handler in GamesCraftersUWAPI which assigns values to intermediate states and part-moves. The assumption behind the multipart moves supported in this interface are that the multipart move graphs are relatively simple. With all of the example games presented, the multipart move graphs are shallow.

6.3 A Game in and of Itself

Again consider the same position from the example game. We discuss how the multipart move handler solves a multipart move graph.

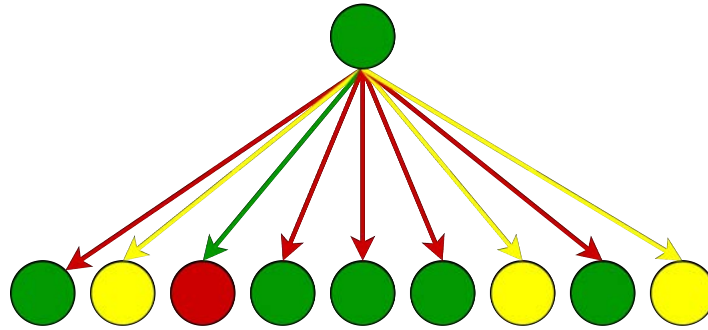

Figure 6.3: The real parent position and its 9 real child positions. There are 5 losing full-moves, 3 tying full-moves, and 1 winning full-move.

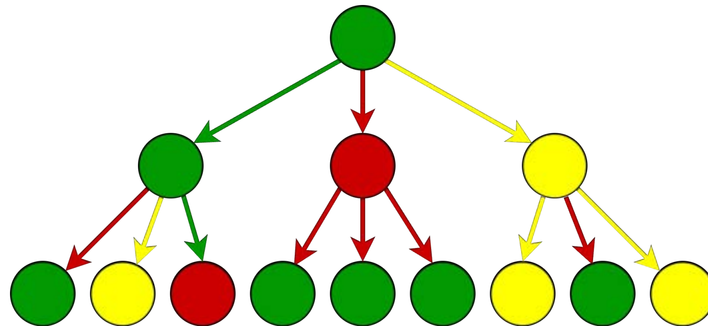

Figure 6.4: The choice of where to move the circle piece introduces three intermediate states. The player whose turn it is at the real parent position also moves at any intermediate state. At the real child positions, the opponent is to move. Any part-move leading to an intermediate state has the same value as that intermediate state. If a part-move leads from an intermediate state or real position to a real child position, then the part-move has value WIN if the real child position has value LOSE, TIE/DRAW if the real child position has value TIE/DRAW, and LOSE if the real child position has value WIN.

The value of an intermediate state is defined as the outcome of the game for the player whose turn it is at that intermediate state, assuming perfect play. The remoteness of an intermediate state is the number of full-moves (including the one that the current sequence of part-moves is specifying) until the end of the game assuming perfect play.

6.4 New Image AutoGUI Interfaces Involving Multipart Moves

What follows are game interfaces built using the multipart-move Image AutoGUI system. Unless otherwise specified, the rules of any of the following games that are lesser known are explained in Explanation of Rules for Various Games. For each of the following games, we first show examples of how its interface looks, then briefly explain the multipart nature of the games' moves and how to specify moves while using its interface.

6.4.1 L-game

The interface uses 3-part moves. In the first part-move, the players move the L-piece. This is achieved by clicking an L-shaped move button. When a particular L-shaped move button is clicked, then the piece will be oriented as the shape of the move button shows and its corner will be at the slot where the move button is. In the second part-move, they select which neutral piece will be moved. In the third part-move, they

select the new location of the selected neutral piece. The player indicates that they wish not to move either of the neutral pieces by selecting one of the neutral pieces and placing it in its original location.

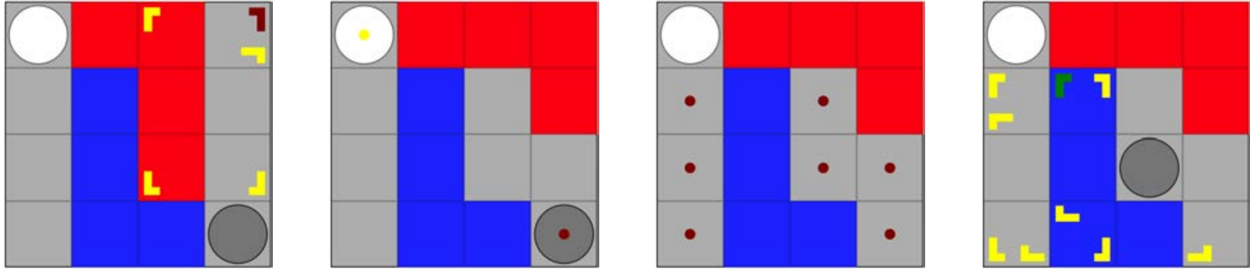

Figure 6.5: The first image shows the initial position, where L-shaped move buttons are used to specify where to move the red L-piece. The second image shows the intermediate state after clicking the tying L-shaped move button in the top-right corner. At this state, there are two move buttons available for the player to specify which neutral piece to move. After selecting the gray neutral piece, a second intermediate state is displayed (third image) in which available slots where the gray neutral piece can be moved to are indicated. The fourth image shows the child position that results from making the third part-move.

6.4.2 3-Spot

2-part moves are used. For the first part-move, the player specifies where to move the piece belonging to them. For the second part-move, the player specifies where to move the neutral piece. Elliptical move buttons crossing over two board slots indicate how a piece should be placed.

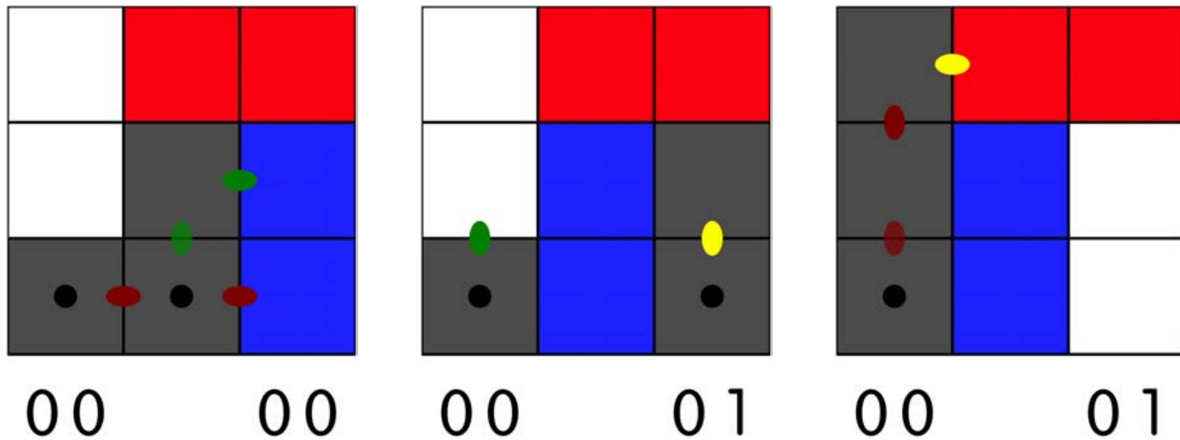

Figure 6.6: The left image shows a real position with blue to move. Suppose for the first part-move, the vertical elliptical move button is clicked. An intermediate state (center image) is loaded that shows the new location of the blue piece and an updated score. The updated score can either be shown after or before the white piece is moved, but we decide to update it before. Suppose for the second part-move the right elliptical move button is clicked. The right image shows the white piece at its new location.

6.4.3 Tic-Tac-Two

See Chapter 4 for an explanation of the rules of Tic-Tac-Two. Making moves on the Tic-Tac-Two interface works as follows:

- Placing a mark within the tic-tac-toe grid is a single-part move. The move is performed by clicking a circle move button on the desired empty slot within the grid.

- Moving the grid is a 2-part move. The first part-move is indicating that player wishes to move the grid by clicking the circle move button at the bottom of the board. An intermediate state is then loaded with circle move buttons the player can click to specify the new board (second part-move).
- Moving a mark that has already been placed on the board is a 2-part move. The first part-move is indicating which mark the player wants to move. An intermediate state is then loaded with arrows the player can click to specify where to move the selected mark. We could have used a single-part move to perform this kind of move, which would involve displaying arrows from every movable mark to all their possible destinations; however, we decided against it because there could be many arrows displayed at a time, cluttering the board.

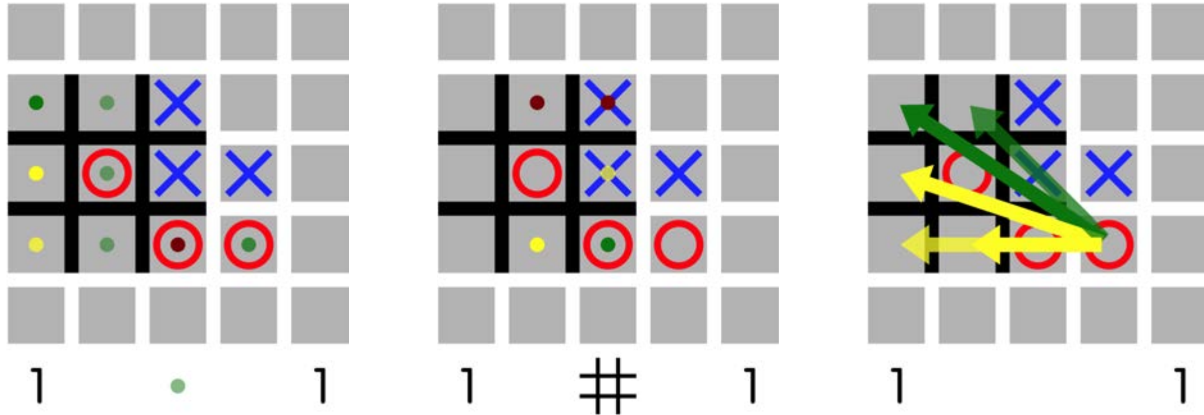

Figure 6.7: The left image shows a real position with O to move. The circle button below the board is clicked if the player wishes to move the grid, and any circle move button on top of any "O" mark is clicked if the player wishes to move the corresponding "O". The center image shows the intermediate state loaded after the "move grid" button is clicked. The right image shows the intermediate state loaded if the player indicates that they instead wish to move the rightmost "O".

6.4.4 Nine Men's Morris

See Chapter 4 for an explanation of the rules of Nine Men's Morris. To place a piece, the user clicks the circle move button over the intersection point at which they wish to place. To slide a piece, the user clicks the arrow pointing from the piece they wish to move to the intersection point they wish to move the piece to. If a mill is created as a result of the placement or movement, then the user must make a second part-move – specifying which of their opponent's pieces they wish to remove from the board. A button to perform the second part-move is displayed on each of the opponent's removable pieces.

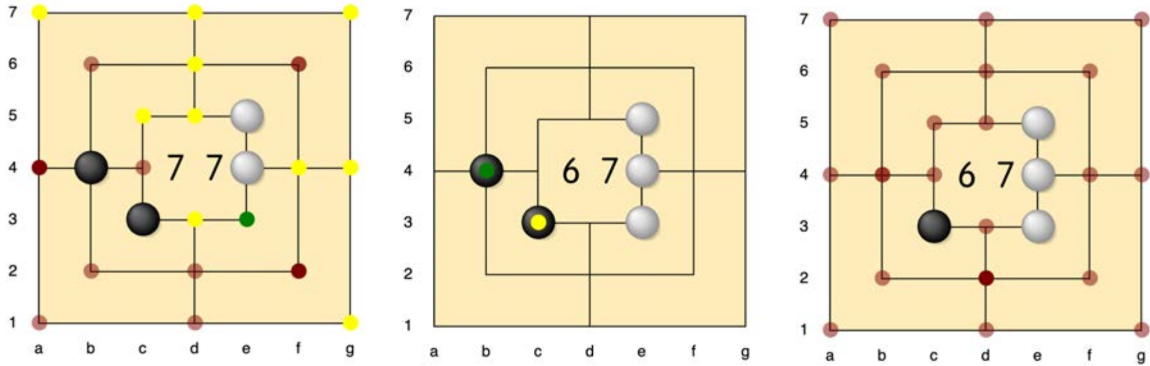

Figure 6.8: The left image shows a real position with white to move. Suppose a white piece is placed on e3, creating a mill. An intermediate state is loaded (center image) where circle move buttons are displayed which, when clicked, remove the corresponding piece from the board. The right image shows the real child position resulting from removing the piece on b4.

6.4.5 Topitop

See Chapter 4 for an explanation of the rules of Topitop. Making moves on the Topitop interface works as follows:

- Placing a building component on the board is a 2-part move. For the first part-move, the player chooses which building component to place. For the second part-move, the player chooses where to place the building component. Circle move buttons for the first part-move are displayed by the remaining building components. When one of these buttons is clicked, an intermediate state is displayed at which (1) the selected building is highlighted and (2) circle move buttons for the second part-move are displayed at the empty slots of the board.
- Sliding an existing structure is a single-part move. The move is performed by clicking an arrow button from the source slot to the destination slot.

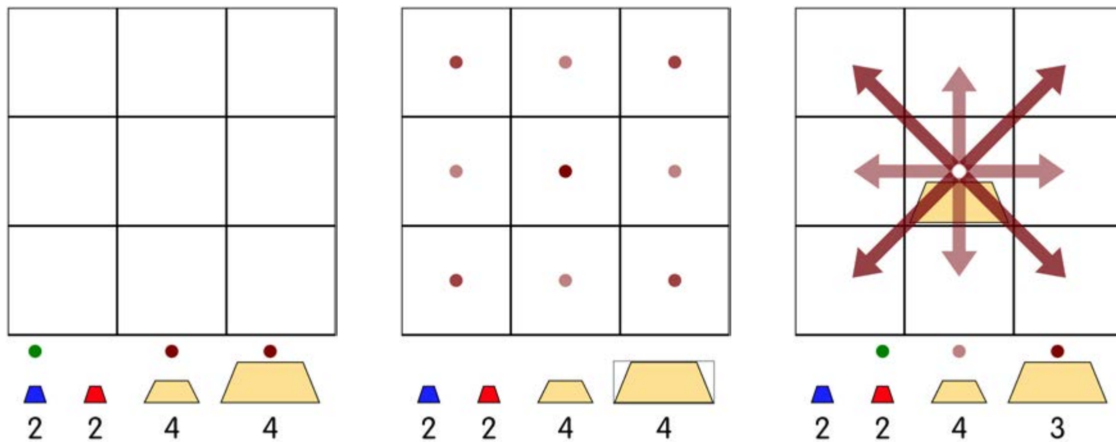

Figure 6.9: The initial position is shown on the left. The center shows the intermediate state after the large building component is selected. The right shows the child position that results from then placing the building component in the center. At this child position, the second player can either slide the large piece already on the board or place another building component.

6.4.6 Chung-Toi

Every move is 2-part. To place a piece, the first part-move is choosing a slot in which to place and the second part-move is choosing the piece's orientation. To slide a piece, the first part-move is clicking an arrow indicating which piece to slide and where it should go, and the second part-move is choosing the piece's new orientation. In the second phase, if any of the circular move buttons over the player's pieces are clicked (first part-move), then the second part-move is choosing either the same orientation of the piece as before (passing a turn) or the other orientation (rotating that piece). After any first part-move, an intermediate state is reached in which (1) a piece on the board is shown with an undefined orientation and (2) a control panel, which the player will use to choose the piece's orientation, appears below the board.

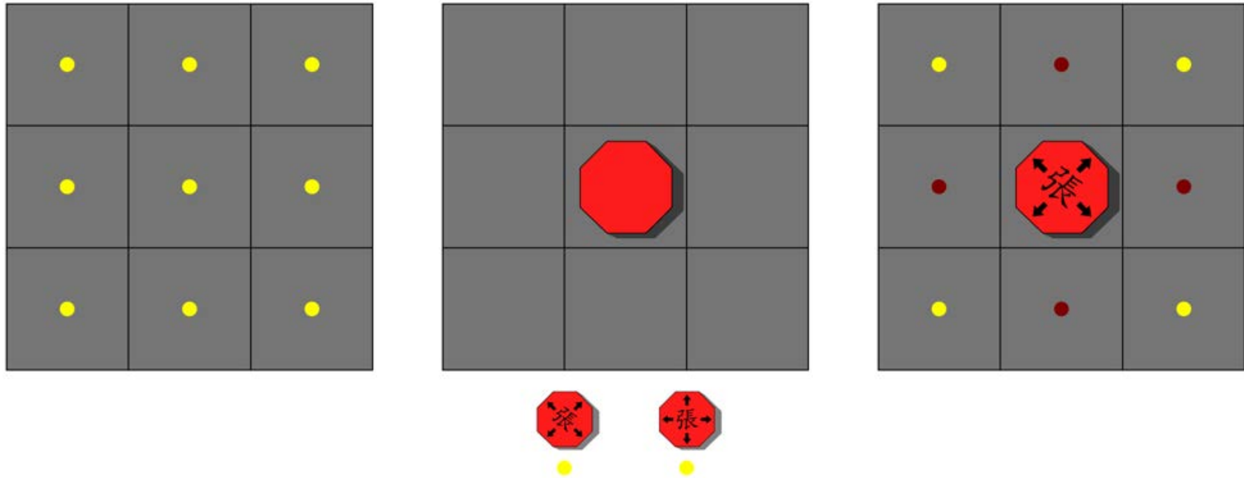

Figure 6.10: The left image shows the initial position. Suppose the center move button is clicked, i.e., the first part-move is choosing to place at the center of the board. An intermediate state is loaded (center image) with a control panel for the player to choose the piece's orientation. The right image shows the real position resulting from choosing to set the piece to the "×" orientation as the second part-move.

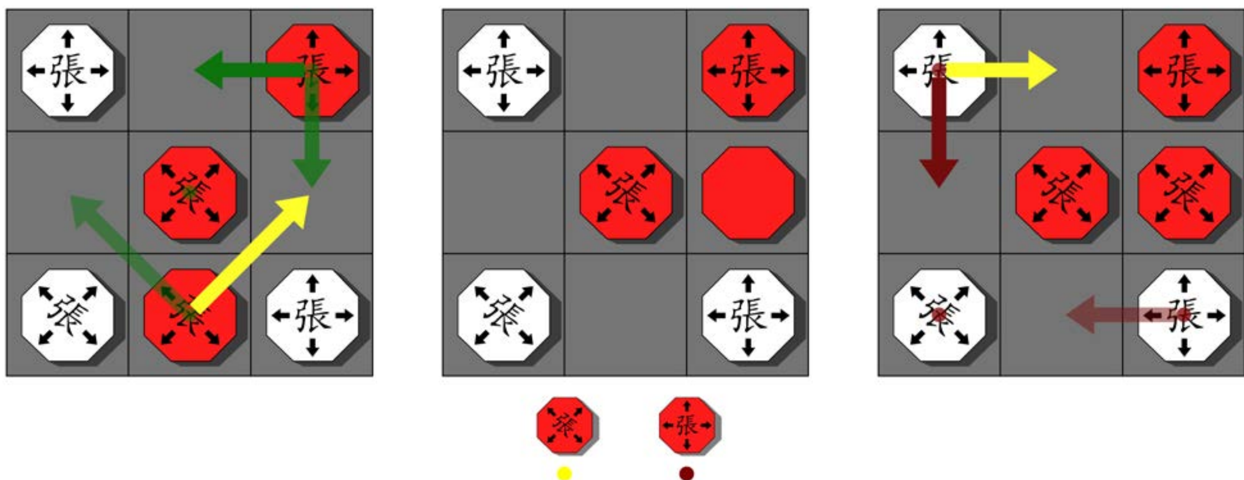

Figure 6.11: The left image shows a real position with red to move. If the arrow button pointing northeast is clicked (first part-move), then an intermediate state is loaded (center image) with the piece moved to the center right and a control panel for selecting orientation. The right image shows the real child position resulting from choosing to set the piece to the "×" orientation as the second part-move.

6.5 Future Work

One downside of a multipart-move interface is that information about the entire distribution of full-move values is not shown. At any time, the only information displayed is the best possible outcome of the game for each current legal part-move. It does not, for example, show the worst sequence of part-moves one can make. Suppose there are two legal first part-moves the current player can make: one tying part-move and one winning part-move. The worst sequence of part-moves the current player can make after clicking the tying part-move may lead to a child position that is better for the current player than the worst sequence of part-moves the current player can make after clicking the winning part-move. In other words, one does not immediately know whether the worst full-move is specified by first clicking the winning part-move.

In the future, a way to visualize the entire full-move-value distribution should be added to Gamesman-Uni for multipart-move interfaces.

Chapter 7

Conclusion

We presented the general methodology we used to solve and count positions by value-remoteness for both loopy and loop-free games. For Quarto, we used a custom retrograde solver with a novel tier definition. For Nine men's Morris, Bagh-Chal, Tic Tac Two and Topitop, we used the retrograde solver within the Gamesman framework. Quarto is a tie in 17, Bagh-Chal is a draw, Tic-Tac-Two is a tie in 12, and Topitop is a win in 31. We also described the Image AutoGUI developed to support user interfaces for a number of games, including games with multipart moves.

Additional information about the work, including data files and errata, can be found at
http://www.cameroncheung.com/ucbtechreport.html.

Bibliography

[1] Cheung C. Quarto Solver; 2023. [Accessed 29-April-2023]. https://github.com/GamesCrafters/QuartoFall2022.

[2] Garcia DD. GAMESMAN A finite, two-person perfect-information game generator; 1990. [Accessed 29-April-2023]. https://people.eecs.berkeley.edu/~ddgarcia/software/gamesman/GAMESMAN.pdf.

[3] Abstract Strategy Game; 2023. [Accessed 29-April-2023]. https://en.wikipedia.org/wiki/Abstract_strategy_game/.

[4] GamesCrafters Glossary; 2023. [Accessed 29-April-2023]. https://nyc.cs.berkeley.edu/wiki/Glossary.

[5] Weisstein EW. Vertex Contraction; 2023. [Accessed 29-April-2023]. From MathWorld–A Wolfram Web Resource. https://mathworld.wolfram.com/VertexContraction.html.

[6] Delgadillo M, Chen Y. Using Tier Gamesman; 2006. [Accessed 29-April-2023]. https://github.com/GamesCrafters/GamesmanClassic/blob/master/doc/files/UsingTierGamesman.pdf.

[7] Kahn AB. Topological sorting of large networks; 1962. [Accessed 29-April-2023]. https://dl.acm.org/doi/10.1145/368996.369025.

[8] GamesmanUni;. [Accessed 29-April-2023]. https://github.com/GamesCrafters/GamesmanUni/.

[9] GamesCraftersUWAPI;. [Accessed 29-April-2023]. https://github.com/GamesCrafters/GamesCraftersUWAPI.

[10] GamesmanClassic;. [Accessed 29-April-2023]. https://github.com/GamesCrafters/GamesmanClassic.

[11] Gasser R. Solving Nine Men's Morris; 1996. Computational Intelligence, Volume 12, Number 1.

[12] Gévay GE. Calculating Ultra-Strong and Extended Solutions for Nine Men's Morris, Morababa, and Lasker Morris; 2014. [Accessed 29-April-2023]. Article in IEEE Trasactions on Computational Intelligence and AI in Games.

[13] Goossens L. Quarto; 2001. [Accessed 29-April-2023]. https://web.archive.org/web/20010222153235/http://ssel.vub.ac.be/Members/LucGoossens/quarto/quartotext.htm.

[14] Group Action; 2023. [Accessed 29-April-2023]. https://en.wikipedia.org/wiki/Group_action.

[15] Orbit (Group Theory); 2020. [Accessed 29-April-2023]. https://proofwiki.org/wiki/Definition:Orbit_(Group_Theory).

[16] Mattson T. A "Hands-On" Introduction to OpenMP;. [Accessed 29-April-2023]. https://www.openmp.org/wp-content/uploads/Intro_To_OpenMP_Mattson.pdf.

[17] Delgadillo M. GENERIC HASH API; 2007. [Accessed 29-April-2023]. https://github.com/GamesCrafters/GamesmanClassic/blob/master/doc/files/generic_hash_api.pdf.

[18] GNU Gzip Advanced Usage;. [Accessed 29-April-2023]. https://www.gnu.org/software/gzip/manual/html_node/Advanced-usage.html.

[19] Make a Nine Men's Morris Board;. [Accessed 29-April-2023]. https://gunstonhall.org/learn/learning-from-home/games-and-amusements/colonial-games-make-a-nine-mens-morris-board/.

[20] Tandukar S. Bagh Chal: A native board game on the brink of extinction; 2021. [Accessed 29-April-2023]. https://kathmandupost.com/art-culture/2021/09/09/bagh-chal-a-native-board-game-on-the-brink-of-extinction.

[21] Liou A. GamesmanUni GUI Accessibility and Combinatorial Games; 2022. [Accessed 29-April-2023]. https://www2.eecs.berkeley.edu/Pubs/TechRpts/2022/EECS-2022-148.pdf.

[22] Black N. Why the Legend is Doing You More Harm than Good; 2014. [Accessed 29-April-2023]. https://www.onepager.com/community/blog/why-the-legend-is-doing-you-more-harm-than-good/.

[23] Weisstein EW. Wheel Graph; 2023. [Accessed 29-April-2023]. From MathWorld–A Wolfram Web Resource. https://mathworld.wolfram.com/WheelGraph.html.

Appendix A

Explanation of Rules for Various Games

Each game listed here is one that I worked on an Image AutoGUI interface for but was not involved in solving. See the preceding chapters for game rule explanations for all other games discussed in this report.

3-Spot: The game is played on a 3×3 board, where two players take turns moving their respective 2-by-1 pieces. The first player controls the blue piece, while the second player controls the red piece. Both players also control a neutral 2-by-1 white piece.

On a player's turn, they must first move their own piece to a 2-by-1 area on the board that contains at least one new square. If their piece covers two new squares in the bottom row of the board, then the player gains two points. If their piece covers only one new square in the bottom row, the player gains one point. If their piece does not cover any new squares in the bottom row, the player does not score any points. The player then moves the neutral piece to a 2-by-1 area that contains at least one new square.

At any time during the game, no two pieces can cover the same squares on the board. A player wins if they have at least 12 points while their opponent has at least 6 points. Alternatively, a player wins if they have fewer than 6 points while their opponent has at least 12 points.

Achi: One player controls 3 white pieces and the other controls 3 black pieces. The board starts empty. At the beginning of the game, each player takes turns placing one of their pieces on the board. Once all pieces have been placed, players take turns sliding their pieces to adjacent slots. The first player to form a straight line of their pieces wins.

Chomp: There is a m-row, n-column chocolate bar. On a player's turn, any square aside from the bottom left square that is available may be selected. When a square is selected, all squares located on or to the right of its column and on or above its row disappear, i.e., those squares become unavailable. A player loses when the bottom left square remains, i.e., no available squares remain.

Chung-Toi: The game is played on a tic-tac-toe board. The first player has three red pieces and the second player has three white pieces. A player wins when if their pieces form a 3-in-a-row vertically, horizontally or diagonally.

- First Phase: Each player takes turns placing one of their pieces on the board. Each piece is placed either in a "+" or "×" orientation.
- Second Phase: Begins once all pieces have been placed. Players take turns either (1) sliding pieces of their own color, (2) rotating a piece in place, or (3) passing their turn. Pieces in a "+" orientation can only slide horizontally and vertically and pieces in a "×" orientation can only slide diagonally. When a piece slides, it must slide into an empty slot but it can jump over any piece that blocks its path

between its departure and destination slot. After sliding a piece, the player chooses to change or keep the piece's orientation.

Connect 4: There is a 6-row, 7-column grid. The first player has 21 red pieces and the second player has 21 yellow pieces. Players take turns placing one of their pieces by selecting a column in which the piece will occupy the first (counting from the bottom upward) unoccupied slot in that column. A player wins when they create a diagonal, horizontal, or vertical four-in-a-row of their color.

Dawson's Chess: There is a $1 \times n$ chessboard. Players take turns placing a king on the board such that it is not attacked by any already-placed kings. A player loses when they are unable to legally place a king on their turn.

Dodgem: The game is played on a 3×3 board. The top-left and center-left slots are each occupied by a blue piece. The bottom-left and bottom-center slots are each occupied by a red piece. One player controls the blue pieces and may only slide a piece one square to the right, up, or down but not left; and the other controls the red pieces and may only slide a piece up, left, or right but not down. Players take turns sliding one of their pieces. The first player to slide both of their pieces off the board wins.

L-game: The game is played on a 4×4 board. There are four pieces: 2 neutral pieces, 1 red L-piece, and 1 blue L-piece. At any time, none of these pieces can occupy the same slots. The first player controls the red L-piece and the second player controls the blue L-piece. A player makes a move by (1) moving their L-piece such that it covers a different set of 4 slots, and (2) choosing either to move one of the neutral pieces to any slot on the board or to leave the neutral pieces in the same slots. An L-piece must always cover four slots, i.e., it must be contained in the 4x4 grid. If a player cannot find a valid way to move their L-piece then they lose.

Mū Tōrere: The game is played on a wheel graph W_9 [23]. Four black pieces and four white pieces are placed on the degree 3 nodes such that there are only two edges connecting a node occupied by a black piece and a node occupied by a white piece. The first player controls the black pieces; the other controls the white pieces. Players take turns moving one of their pieces to an unoccupied adjacent vertex. A player loses if they are unable to legally move one of their pieces on their turn.

QuickCross: The game is played on a 4×4 board where players take turns either (1) placing a chip vertically or horizontally on an empty cross on the board or (2) altering one of the existing crosses by rotating it 90°. The first player to create a full column, row, or diagonal with chips facing in the same direction wins.

Shift-Tac-Toe: This game incorporates features from Tic-Tac-Toe and Connect 4. It uses a 3×5 board, where the center 3×3 board is filled in a manner exactly like Connect 4. On a player's turn, they can either drop a piece into one of the three columns or they can shift one of the rows left or right. When shifting a row, any pieces that are at the side of the 3×3 center and are shifted off disappear. The first player to have a three-in-a-row of their own color wins. It is possible to create a three-in-a-row on behalf of the opponent. It is possible to create three-in-a-rows of both colors simultaneously, resulting in a tie.

Snake: The game is played on a board of arbitrary shape. The first player controls the head of the snake. The second player controls the tail. Each player takes turns moving their piece (head or tail) to an orthogonally adjacent empty slot. After a piece is moved, the cell it originally occupied is replaced with a "body" piece indicating that the cell is no longer empty. A player loses when they have no legal moves.

Tac Tix: Coins are arranged in a $n \times n$ square. Players take turns removing a set of connected coins that occupy the same row or column. The first player unable to do so on their turn loses.

Toot-and-Otto: The game is played on a 4-row, 6-column board. On their turn, a player places either a "T" piece or an "O" piece on the board and the placement rule is the same as that of Connect 4. Throughout the game a player can only have placed at most 6 "T" pieces and at most 6 "O" pieces total.

The first player aims to spell "TOOT" with four pieces vertically, horizontally, or diagonally. The second player aims to spell "OTTO" with four pieces vertically, horizontally, or diagonally. A person wins if their word appears on the board at any point in the game, unless both words are created simultaneously in which case the game is a tie. The game is also a tie if the board is filled with neither word being spelled on the board.

Appendix B

Value-Remoteness Counts

LEVEL	Tie in (16 - LEVEL)	Lose in 0	Win in 1	Lose in 2	Win in 3	Lose in 4	Win in 5	Lose in 6	Win in 7	Lose in 8	Win in 9	Lose in 10	Win in 11	Level Total
0	16													16
1	3,840													3,840
2	403,200													403,200
3	23,946,240		514,560											24,460,800
4	853,641,216	128,640	74,096,640										24,689,664	952,556,160
5	20,547,176,448	18,524,160	4,333,768,704		4,402,176		1,661,952		5,225,472		8,506,368	14,324,736	66,008,064	24,999,598,080
6	234,892,710,912	1,120,711,680	134,471,780,352	884,736	1,879,314,432	500,736	1,794,416,640	2,399,232	9,003,205,632	3,597,312	68,437,982,208	28,151,808		451,635,655,680
7	2,350,181,369,856	37,219,221,504	2,439,792,291,840	537,194,496	110,652,515,328	1,304,961,024	99,579,386,880	3,399,840,768	171,763,553,280	43,333,972,992	380,908,416,000			5,638,672,723,968
8	10,382,470,537,728	744,639,054,336	26,906,093,405,184	37,127,675,904	2,158,718,386,176	72,306,997,248	2,880,857,478,144	94,770,441,216	4,753,371,087,360	185,010,020,352				48,215,365,083,648
9	35,322,282,445,824	9,283,942,821,888	181,478,129,854,464	922,529,832,960	15,367,022,404,608	1,856,545,969,152	13,808,640,798,720	3,360,404,127,744	15,336,728,589,312					276,736,226,844,672
10	88,373,016,870,912	72,313,487,511,552	734,658,708,814,848	7,662,268,793,856	61,574,001,524,736	9,336,322,228,224	51,443,574,635,520	9,318,693,921,792						1,034,680,074,301,440
11	128,493,614,512,128	344,778,103,799,808	1,708,338,837,209,088	30,870,189,699,072	115,214,294,513,664	32,078,262,226,944	69,062,042,456,064							2,428,835,344,416,768
12	155,142,013,851,648	875,793,844,162,560	2,116,998,194,107,392	54,610,257,792,000	123,985,358,942,208	39,493,492,660,224								3,366,023,161,516,032
13	92,870,290,851,840	1,200,273,645,182,976	1,230,113,923,903,488	47,504,382,554,112	70,887,608,727,552									2,641,649,851,219,968
14	40,654,511,314,944	725,940,028,575,744	267,313,322,661,888	19,812,293,603,328										1,053,720,156,155,904
15	6,628,770,256,896	148,698,615,810,048	12,365,767,360,512											167,693,153,427,456
16	414,298,141,056	5,578,178,254,848												5,992,476,395,904
Value Total	560,887,768,034,704	3,383,442,843,759,744	6,280,751,649,768,960	161,419,588,030,464	389,299,540,730,880	82,838,235,543,552	137,296,490,833,920	12,777,270,730,752	20,270,871,661,056	228,347,590,656	449,354,904,576	42,476,544	90,697,728	11,029,662,094,763,536

Table B.1: Quarto Value-Remoteness Counts

Pieces Left	# White	# Black	Position Upper Bound	Reachable Positions	Reachable Canonical Positions	Highest Win Remoteness	Highest Lose Remoteness	Percentage
18	0	0	1	1	1			100.00%
17	1	0	24	24	4			100.00%
16	1	1	552	552	46			100.00%
15	2	1	6,072	6,072	428	125		100.00%
14	2	2	63,756	63,756	4,200	113	124	100.00%
13	3	2	425,040	421,680	26,855	179	112	99.21%
13	3	1	42,504	336	29		74	0.79%
12	3	3	2,691,920	2,649,552	166,821	187	178	98.43%
12	2	3	425,040	3,360	243	53	108	0.79%
12	3	2	425,040	3,360	243	113		0.79%
11	4	3	12,113,640	11,638,296	730,214	183	186	96.08%
11	4	2	2,018,940	63,840	4,111	95	126	3.16%
11	3	3	2,691,920	21,088	1,392	171	52	0.78%
11	3	2	425,040	576	49	73		0.14%
10	4	4	51,482,970	48,285,882	3,024,492	185	182	93.79%
10	3	4	12,113,640	379,584	23,997	149	172	3.13%
10	4	3	12,113,640	379,584	23,997	173	94	3.13%
10	3	3	2,691,920	9,600	646	59	74	0.36%
10	2	3	425,040	576	49		40	0.14%
9	5	4	164,745,504	147,072,000	9,202,533	185	192	89.27%
9	5	3	41,186,376	3,203,448	201,196	161	176	7.78%
9	4	4	51,482,970	1,569,168	98,672	177	148	3.05%
9	4	3	12,113,640	172,800	10,971	123	58	1.43%
9	5	2	7,268,184	4,104	289		62	0.06%
9	3	3	2,691,920	9,408	625	117		0.35%
9	3	2	425,040	576	49	57		0.14%
8	5	5	494,236,512	420,122,292	26,274,839	201	184	85.00%
8	4	5	164,745,504	12,464,160	781,129	175	180	7.57%
8	5	4	164,745,504	12,464,160	781,129	197	160	7.57%
8	4	4	51,482,970	1,180,920	74,346	133	186	2.29%
8	3	5	41,186,376	23,016	1,508		80	0.06%
8	3	4	12,113,640	172,800	10,971	89	136	1.43%
8	5	3	41,186,376	23,016	1,508	119		0.06%
8	4	3	12,113,640	1,200	90	39		0.01%
8	3	3	2,691,920	9,408	625		76	0.35%
8	2	3	425,040	576	49			0.14%
7	6	5	1,153,218,528	900,940,620	56,334,808	195	200	78.12%
7	6	4	411,863,760	60,865,920	3,808,437	177	196	14.78%
7	5	5	494,236,512	35,323,212	2,210,897	191	174	7.15%
7	5	4	164,745,504	9,336,576	585,290	185	146	5.67%
7	6	3	109,830,336	445,776	28,248	77	142	0.41%
7	5	3	41,186,376	40,872	2,669	95	120	0.10%
7	4	5	164,745,504	89,184	5,731	123		0.05%
7	4	4	51,482,970	1,180,920	74,346	157	92	2.29%
7	4	3	12,113,640	172,800	10,971	143	36	1.43%
7	5	2	7,268,184	720	52		36	0.01%
7	3	3	2,691,920	9,408	625	117		0.35%
7	3	2	425,040	576	49	73		0.14%
6	6	6	2,498,640,144	1,796,173,170	112,300,784	201	194	71.89%
6	5	6	1,153,218,528	161,027,904	10,070,698	199	190	13.96%
6	6	5	1,153,218,528	161,027,904	10,070,698	195	176	13.96%
6	5	5	494,236,512	35,165,016	2,201,534	173	184	7.12%
6	4	6	411,863,760	1,619,280	102,075	133	144	0.39%
6	4	5	164,745,504	9,351,840	586,248	169	178	5.68%
6	6	4	411,863,760	1,619,280	102,075	149	76	0.39%
6	5	4	164,745,504	426,000	26,977	121	94	0.26%
6	4	4	51,482,970	1,180,920	74,346	133	172	2.29%
6	3	5	41,186,376	23,256	1,528		88	0.06%
6	3	4	12,113,640	172,800	10,971	109	116	1.43%
6	5	3	41,186,376	10,416	667	79		0.03%
6	4	3	12,113,640	1,200	90	47		0.01%
6	3	3	2,691,920	9,408	625		104	0.35%
6	2	3	425,040	576	49			0.14%
5	7	6	4,283,383,104	2,703,410,700	169,007,212	201	200	63.11%
5	7	5	2,141,691,552	497,817,924	31,125,938	177	194	23.24%

Table B.2: Nine Men's Morris Phase 1, Part 1

Pieces Left	# White	# Black	Position Upper Bound	Reachable Positions	Reachable Canonical Positions	Highest Win Remoteness	Highest Lose Remoteness	Percentage
5	6	6	2,498,640,144	316,248,792	19,774,769	189	198	12.66%
5	6	5	1,153,218,528	158,838,672	9,934,794	199	172	13.77%
5	7	4	823,727,520	13,088,208	820,602	121	148	1.59%
5	6	4	411,863,760	3,920,736	246,469	169	174	0.95%
5	5	6	1,153,218,528	4,245,324	266,547	149	132	0.37%
5	5	5	494,236,512	35,232,780	2,205,788	189	168	7.13%
5	5	4	164,745,504	9,351,840	586,248	171	132	5.68%
5	7	3	235,350,720	32,244	2,086	55	68	0.01%
5	6	3	109,830,336	172,104	10,845	65	88	0.16%
5	5	3	41,186,376	40,872	2,669	69	84	0.10%
5	4	5	164,745,504	93,024	5,980	99		0.06%
5	4	4	51,482,970	1,180,920	74,346	143	108	2.29%
5	4	3	12,113,640	172,800	10,971	113	60	1.43%
5	5	2	7,268,184	720	52			0.01%
5	3	3	2,691,920	9,408	625	117		0.35%
5	3	2	425,040	576	49			0.14%
4	7	7	6,731,030,592	3,741,325,068	233,882,415	199	200	55.58%
4	6	7	4,283,383,104	903,241,452	56,469,514	197	192	21.09%
4	7	6	4,283,383,104	903,241,452	56,469,514	197	176	21.09%
4	6	6	2,498,640,144	379,616,460	23,737,869	185	198	15.19%
4	5	7	2,141,691,552	31,866,576	1,995,365	167	172	1.49%
4	5	6	1,153,218,528	160,369,476	10,030,878	177	190	13.91%
4	7	5	2,141,691,552	31,866,576	1,995,365	149	120	1.49%
4	6	5	1,153,218,528	20,270,352	1,269,979	173	168	1.76%
4	5	5	494,236,512	35,350,860	2,213,252	183	170	7.15%
4	4	7	823,727,520	108,696	6,937	49	86	0.01%
4	4	6	411,863,760	1,701,000	107,282	63	132	0.41%
4	4	5	164,745,504	9,351,840	586,248	137	148	5.68%
4	7	4	823,727,520	108,696	6,937	83	54	0.01%
4	6	4	411,863,760	1,040,040	65,333	137	64	0.25%
4	5	4	164,745,504	374,880	23,712	157	68	0.23%
4	4	4	51,482,970	1,180,920	74,346	121	116	2.29%
4	3	5	41,186,376	23,256	1,528		58	0.06%
4	3	4	12,113,640	172,800	10,971		116	1.43%
4	5	3	41,186,376	10,416	667	83		0.03%
4	4	3	12,113,640	1,200	90	67		0.01%
4	3	3	2,691,920	9,408	625			0.35%
4	2	3	425,040	576	49			0.14%
3	8	7	8,413,788,240	3,878,561,460	242,460,039	199	198	46.10%
3	8	6	5,889,651,768	1,821,702,372	113,878,890	187	198	30.93%
3	7	7	6,731,030,592	1,217,519,856	76,112,540	191	196	18.09%
3	7	6	4,283,383,104	1,070,115,708	66,902,564	197	184	24.98%
3	8	5	3,212,537,328	140,648,256	8,799,160	171	166	4.38%
3	7	5	2,141,691,552	89,478,588	5,600,199	169	176	4.18%
3	6	7	4,283,383,104	56,843,688	3,557,965	171	166	1.33%
3	6	6	2,498,640,144	383,715,144	23,994,858	195	176	15.36%
3	6	5	1,153,218,528	160,761,984	10,055,584	173	182	13.94%
3	8	4	1,338,557,220	1,813,500	113,920	73	144	0.14%
3	7	4	823,727,520	7,388,736	462,923	117	136	0.90%
3	6	4	411,863,760	3,602,712	226,547	147	178	0.87%
3	5	7	2,141,691,552	261,384	16,523	85	48	0.01%
3	5	6	1,153,218,528	4,809,072	302,122	141	62	0.42%
3	5	5	494,236,512	35,234,364	2,205,887	175	136	7.13%
3	5	4	164,745,504	9,351,840	586,248	155	120	5.68%
3	8	3	411,863,760	1,656	123		32	0.00%
3	7	3	235,350,720	32,736	2,127		62	0.01%
3	6	3	109,830,336	171,636	10,805		82	0.16%
3	5	3	41,186,376	40,872	2,669		84	0.10%
3	4	5	164,745,504	93,024	5,980	77		0.06%
3	4	4	51,482,970	1,180,920	74,346	159		2.29%
3	4	3	12,113,640	172,800	10,971	51		1.43%
3	5	2	7,268,184	720	52			0.01%
3	3	3	2,691,920	9,408	625			0.35%
3	3	2	425,040	576	49			0.14%
2	8	8	9,465,511,770	3,644,358,480	227,824,019	197	188	38.50%

Table B.3: Nine Men's Morris Phase 1, Part 2

Pieces Left	# White	# Black	Position Upper Bound	Reachable Positions	Reachable Canonical Positions	Highest Win Remoteness	Highest Lose Remoteness	Percentage
2	7	8	8,413,788,240	2,237,971,464	139,895,782	197	198	26.60%
2	8	7	8,413,788,240	2,237,971,464	139,895,782	197	178	26.60%
2	7	7	6,731,030,592	1,759,472,964	109,993,521	197	196	26.14%
2	6	8	5,889,651,768	230,153,064	14,396,669	169	172	3.91%
2	6	7	4,283,383,104	1,105,001,124	69,085,320	175	194	25.80%
2	8	6	5,889,651,768	230,153,064	14,396,669	161	170	3.91%
2	7	6	4,283,383,104	277,578,048	17,360,805	181	168	6.48%
2	6	6	2,498,640,144	389,930,838	24,384,456	181	172	15.61%
2	5	8	3,212,537,328	4,027,152	252,388	69	144	0.13%
2	5	7	2,141,691,552	36,613,260	2,293,000	119	164	1.71%
2	5	6	1,153,218,528	160,419,348	10,034,027	165	176	13.91%
2	8	5	3,212,537,328	4,027,152	252,388	143	72	0.13%
2	7	5	2,141,691,552	24,790,512	1,551,799	171	116	1.16%
2	6	5	1,153,218,528	18,630,408	1,167,157	185	126	1.62%
2	5	5	494,236,512	35,350,860	2,213,252	165	154	7.15%
2	4	8	1,338,557,220	5,148	374		28	0.00%
2	4	7	823,727,520	115,680	7,404		72	0.01%
2	4	6	411,863,760	1,701,000	107,282		134	0.41%
2	4	5	164,745,504	9,351,840	586,248	45	158	5.68%
2	8	4	1,338,557,220	5,148	374	31		0.00%
2	7	4	823,727,520	115,584	7,395	61		0.01%
2	6	4	411,863,760	1,033,020	64,866	131		0.25%
2	5	4	164,745,504	374,880	23,712	131		0.23%
2	4	4	51,482,970	1,180,920	74,346	45	50	2.29%
2	3	5	41,186,376	23,256	1,528		58	0.06%
2	3	4	12,113,640	172,800	10,971			1.43%
2	5	3	41,186,376	10,416	667	103		0.03%
2	4	3	12,113,640	1,200	90			0.01%
2	3	3	2,691,920	9,408	625			0.35%
2	2	3	425,040	576	49			0.14%
1	9	8	8,413,788,240	2,510,097,156	156,918,254	189	188	29.83%
1	9	7	8,413,788,240	2,934,340,260	183,422,929	189	196	34.88%
1	8	8	9,465,511,770	2,011,860,900	125,762,159	197	176	21.25%
1	8	7	8,413,788,240	3,173,480,688	198,375,028	195	196	37.72%
1	9	6	6,544,057,520	604,776,168	37,817,360	169	182	9.24%
1	8	6	5,889,651,768	706,167,708	44,158,734	183	190	11.99%
1	7	8	8,413,788,240	274,516,452	17,168,684	175	164	3.26%
1	7	7	6,731,030,592	1,817,022,240	113,594,446	193	170	26.99%
1	7	6	4,283,383,104	1,116,773,928	69,823,090	181	180	26.07%
1	9	5	3,926,434,512	26,988,564	1,689,074	121	144	0.69%
1	8	5	3,212,537,328	101,061,816	6,321,699	137	170	3.15%
1	7	5	2,141,691,552	86,835,276	5,434,251	165	184	4.05%
1	6	8	5,889,651,768	6,441,060	403,539	143	68	0.11%
1	6	7	4,283,383,104	75,188,808	4,706,065	141	118	1.76%
1	6	6	2,498,640,144	384,456,990	24,041,421	177	164	15.39%
1	6	5	1,153,218,528	160,761,984	10,055,584	167	164	13.94%
1	9	4	1,784,742,960	190,644	12,134	3	42	0.01%
1	8	4	1,338,557,220	1,947,480	122,400	7	74	0.15%
1	7	4	823,727,520	7,714,512	483,302	35	130	0.94%
1	6	4	411,863,760	3,602,712	226,547	49	130	0.87%
1	5	8	3,212,537,328	11,256	753	27		0.00%
1	5	7	2,141,691,552	306,516	19,444	59		0.01%
1	5	6	1,153,218,528	4,809,072	302,122	133		0.42%
1	5	5	494,236,512	35,234,364	2,205,887	161		7.13%
1	5	4	164,745,504	9,351,840	586,248	55	44	5.68%
1	8	3	411,863,760	1,680	126	1	32	0.00%
1	7	3	235,350,720	32,736	2,127	1	24	0.01%
1	6	3	109,830,336	171,636	10,805		102	0.16%
1	5	3	41,186,376	40,872	2,669	1		0.10%
1	4	5	164,745,504	93,024	5,980	125		0.06%
1	4	4	51,482,970	1,180,920	74,346	29		2.29%
1	4	3	12,113,640	172,800	10,971	7		1.43%
1	5	2	7,268,184	720	52			0.01%
1	3	3	2,691,920	9,408	625			0.35%
1	3	2	425,040	576	49	1		0.14%
Phase 1 Total			275,610,620,697	51,153,431,825	3,198,110,285	201	200	18.56%

Table B.4: Nine Men's Morris Phase 1, Part 3

Pieces Left	# White	# Black	Position Upper Bound	Reachable Positions	Reachable Canonical Positions	Highest Win Remoteness	Highest Lose Remoteness	Percentage
0	9	9	13,088,115,040	3,070,141,476	191,938,998	189	190	23.46%
0	8	9	16,827,576,480	9,739,439,748	608,835,130	201	196	57.88%
0	9	8	16,827,576,480	9,737,310,308	608,701,478	201	196	57.87%
0	8	8	18,931,023,540	18,194,155,270	1,137,348,200	195	196	96.11%
0	7	9	16,827,576,480	12,450,353,992	778,303,625	201	202	73.99%
0	7	8	16,827,576,480	16,739,514,968	1,046,444,018	203	204	99.48%
0	9	7	16,827,576,480	12,449,657,588	778,259,849	201	202	73.98%
0	8	7	16,827,576,480	16,739,598,720	1,046,449,271	203	204	99.48%
0	7	7	13,462,061,184	13,461,977,164	841,580,739	181	180	100.00%
0	6	9	13,088,115,040	11,071,988,996	692,158,588	173	172	84.60%
0	6	8	11,779,303,536	11,767,387,948	735,661,291	185	186	99.90%
0	6	7	8,566,766,208	8,566,713,156	535,583,135	185	184	100.00%
0	9	6	13,088,115,040	11,071,986,440	692,158,417	173	172	84.60%
0	8	6	11,779,303,536	11,767,520,228	735,669,576	185	186	99.90%
0	7	6	8,566,766,208	8,566,712,784	535,583,107	185	184	100.00%
0	6	6	4,997,280,288	4,997,268,876	312,457,941	167	166	100.00%
0	5	9	7,852,869,024	7,235,826,192	452,366,452	113	114	92.14%
0	5	8	6,425,074,656	6,422,015,572	401,513,237	153	160	99.95%
0	5	7	4,283,383,104	4,283,320,220	267,817,755	159	160	100.00%
0	5	6	2,306,437,056	2,306,428,104	144,231,543	163	162	100.00%
0	9	5	7,852,869,024	7,235,826,104	452,366,445	113	114	92.14%
0	8	5	6,425,074,656	6,422,020,072	401,513,526	153	160	99.95%
0	7	5	4,283,383,104	4,283,319,876	267,817,730	159	160	100.00%
0	6	5	2,306,437,056	2,306,428,040	144,231,539	163	162	100.00%
0	5	5	988,473,024	988,471,560	61,828,743	57	56	100.00%
0	4	9	3,569,485,920	3,455,647,176	216,070,554	103	110	96.81%
0	4	8	2,677,114,440	2,676,725,952	167,386,939	111	112	99.99%
0	4	7	1,647,455,040	1,647,370,720	103,027,648	111	112	99.99%
0	4	6	823,727,520	823,714,656	51,530,732	157	156	100.00%
0	4	5	329,491,008	329,489,728	20,620,902	29	28	100.00%
0	9	4	3,569,485,920	3,455,647,176	216,070,554	103	110	96.81%
0	8	4	2,677,114,440	2,676,725,950	167,386,938	111	112	99.99%
0	7	4	1,647,455,040	1,647,370,712	103,027,647	111	112	99.99%
0	6	4	823,727,520	823,714,636	51,530,729	157	156	100.00%
0	5	4	329,491,008	329,489,728	20,620,902	29	28	100.00%
0	4	4	102,965,940	102,965,822	6,451,182	9	8	100.00%
0	3	9	1,189,828,640	1,180,422,880	73,827,320	25	30	99.21%
0	3	8	823,727,520	823,727,520	51,527,672	33	34	100.00%
0	3	7	470,701,440	470,701,440	29,451,376	31	30	100.00%
0	3	6	219,660,672	219,660,672	13,750,640	7	6	100.00%
0	3	5	82,372,752	82,372,752	5,160,780	31	2	100.00%
0	3	4	24,227,280	24,227,280	1,520,796	33	32	100.00%
0	9	3	1,189,828,640	1,180,422,880	73,827,320	25	30	99.21%
0	8	3	823,727,520	823,727,520	51,527,672	33	34	100.00%
0	7	3	470,701,440	470,701,440	29,451,376	31	30	100.00%
0	6	3	219,660,672	219,660,672	13,750,640	7	6	100.00%
0	5	3	82,372,752	82,372,752	5,160,780	31	2	100.00%
0	4	3	24,227,280	24,227,280	1,520,796	33	32	100.00%
0	3	3	5,383,840	5,383,840	339,252	25	26	100.00%
0	2	9	274,575,840	73,202,084	4,581,812			26.66%
0	2	8	176,513,040	34,280,956	2,146,761			19.42%
0	2	7	94,140,288	12,155,068	762,053			12.91%
0	2	6	41,186,376	3,159,940	198,562			7.67%
0	2	5	14,536,368	569,596	36,052			3.92%
0	2	4	4,037,880	63,816	4,109			1.58%
0	2	3	850,080	3,360	243			0.40%
0	9	2	274,575,840	73,202,084	4,581,812			26.66%
0	8	2	176,513,040	34,280,956	2,146,761			19.42%
0	7	2	94,140,288	12,155,068	762,053			12.91%
0	6	2	41,186,376	3,159,940	198,562			7.67%
0	5	2	14,536,368	569,596	36,052			3.92%
0	4	2	4,037,880	63,816	4,109			1.58%
0	3	2	850,080	3,360	243			0.40%
Phase 2 Total			286,071,923,192	245,698,724,226	15,360,820,664	203	204	85.89%

Table B.5: Nine Men's Morris Phase 2

Appendix C

Code

We present the most interesting pieces of relevant code here. All other code is publicly viewable in the GamesCrafters organization GitHub repositories QuartoFall2022, GamesmanClassic, GamesCraftersUWAPI, and GamesmanUni.

C.1 Quarto Tier Struct and Constructor

```
typedef struct tier {
    uint8_t level; // Number of pieces on board
    uint8_t pieceToPlace; // 0-15; undefined if level=16
    uint16_t piecesPlaced; // 16-bit map; nth LSB is 1 if piece n is on board
    uint16_t occupiedSlots; // 16-bit map; nth LSB is 1 if nth slot is nonempty
    uint8_t occupiedSlotsListX4[16]; // list of occupied slots, each multiplied by 4
    uint64_t occupiedSlotsMask;
} TIER;

void buildTier(TIER *tier, uint8_t pieceToPlace, uint16_t piecesPlaced,
               uint16_t occupiedSlots) {
    tier->pieceToPlace = pieceToPlace;
    tier->piecesPlaced = piecesPlaced;
    tier->occupiedSlots = occupiedSlots;
    tier->occupiedSlotsMask = 0;

    uint8_t counter = 0;
    for (int i = 0; i < 16; i++) {
        if (occupiedSlots & (1 << i)) {
            tier->occupiedSlotsListX4[counter++] = i << 2;
            tier->occupiedSlotsMask |= UINT64_C(0xF) << (i << 2);
        }
    }
    tier->level = counter;
}
```

C.2 Quarto Hash and Unhash

Here we make use of tables that we described in subsection 3.5.2 Optimizations. Namely, `fact[i]` gives us $i!$, `factmul[i << 4 | pieces[i]]` gives us `pieces[i]` $\cdot\, i!$, `whichSetBit[remaining << 4 | idx]` gives us

which set bit the `idx`-th set bit in the `remaining` bitstring is, and `nthSetBit[remaining << 4 | pieces[i]]` gives us k such that the `pieces[i]`-th bit is the k-th set bit in the `remaining` bitstring.

```
// TierPosition to BitBoard
uint64_t unhash(TIER *tier, uint64_t tierPosition) {
    int8_t i;
    uint8_t idx;
    uint8_t pieces[16] = {0};
    uint64_t bitBoard = 0;
    uint32_t remaining = tier->piecesPlaced;

    for (i = tier->level - 1; i > 0; i--) {
        pieces[i] = tierPosition / fact[i];
        tierPosition %= fact[i];
    }

    for (i = tier->level - 1; i >= 0; i--) {
        idx = nthSetBit[remaining << 4 | pieces[i]];
        remaining ^= 1 << idx; // Flip piece to indicate taken out of "remaining" set
        bitBoard |= ((uint64_t) idx) << tier->occupiedSlotsListX4[i];
    }

    return bitBoard;
}

// BitBoard to TierPosition
uint64_t hash(TIER *tier, uint64_t bitBoard) {
    uint8_t i, idx = 0;
    uint8_t pieces[16];
    uint64_t tierPosition = 0;
    uint32_t remaining = 0;

    for (i = 0; i < tier->level; i++) {
        idx = (bitBoard >> tier->occupiedSlotsListX4[i]) & 0xF;
        remaining ^= 1 << idx;
        pieces[i] = whichSetBit[remaining << 4 | idx];
    }

    for (i = 1; i < tier->level; i++) {
        tierPosition += factmul[i << 4 | pieces[i]];
    }

    return tierPosition;
}
```

C.3 Multipart Edge Struct

```
typedef struct multipartedgelist_item
{
        POSITION from;
        POSITION to;
        MOVE partMove;
```

```
        MOVE fullMove;
        BOOLEAN isTerminal;
        struct multipartedgelist_item *next;
}
MULTIPARTEDGELIST;
```

C.4 Multipart Move Graph Generation for Topitop

```
MULTIPARTEDGELIST* GenerateMultipartMoveEdges(
    POSITION position, MOVELIST *moveList, POSITIONLIST *positionList
) {
        MULTIPARTEDGELIST *mpel = NULL;
        BOOLEAN edgeToAdded[4] = {FALSE, FALSE, FALSE, FALSE};
        while (moveList != NULL) {
                MOVE move = moveList->move;
                if (move == NULLMOVE) {
                        break;
                }
                int to = move & 0b1111;
                int from = (move >> 4) & 0b1111;
                MOVE piece = (move >> 8) & 0b11;
                POSITION intermediateMarker = ((POSITION) (4L | piece)) << 61;
                if (from == to) {
                        // Select piece to place
                        if (!edgeToAdded[piece]) {
                                mpel = CreateMultipartEdgeListNode(
                                        position,
                                        position | intermediateMarker,
                                        move | 0b10000000000,
                                        0,
                                        FALSE,
                                        mpel );
                                edgeToAdded[piece] = TRUE;
                        }

                        // Place selected piece
                        mpel = CreateMultipartEdgeListNode(
                                position | intermediateMarker,
                                positionList->position,
                                move | 0b100000000000,
                                move,
                                TRUE,
                                mpel );
                }

                // Ignore sliding moves, they are single-part
                moveList = moveList->next;
                positionList = positionList->next;
        }
        return mpel;
}
```