Five Field Kono Solver: Simplicity Brought to Solving Complexity

Andrew Lee
Dan Garcia, Ed.

Electrical Engineering and Computer Sciences
University of California, Berkeley

Technical Report No. UCB/EECS-2023-195
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2023/EECS-2023-195.html

July 21, 2023

Acknowledgement

To Professor Dan Garcia, for allowing me to join his research group and fostering my, and this project's, development.
To GameCrafters lead, Cameron Cheung, for showing me the ropes of overcoming many challenges.
To GameCrafters veteran, Max Fierro, for helping with the backend and frontend implementation of the
game and the canonical position analysis.
To GamesCrafters veteran, Robert Shi, for walking me through the deep technical walkthrough of the backend codebase.
To My friends and family, for supporting me through my college career.

Five Field Kono Solver: Simplicity Brought to Solving Complexity

by Andrew Lee

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences,
University of California at Berkeley, as a Senior Thesis for the EECS Honors Program.

Teaching Professor Dan Garcia
Supervisor

Date

Abstract

This report explores the development of the programmatic solver for a traditional two-player, perfect information, finite-length strategy game from South Korea called Five Field Kono. The first goal of this research paper is to describe the application programming interfaces (API) available in the GAMESMAN software infrastructure used in the GameCrafters Computational Game Theory research group. Utilizing their APIs, I will identify some of the bottlenecks in the current infrastructure and suggest appropriate solutions to them.

As the title suggests, the second goal is to show how simple solutions can be brought to complex problems. I will demonstrate how one huge problem can be broken down into small individual parts using game-solving theories I learned in the research group, from canonical position calculation for symmetry discrimination to efficient hashing. In terms of reachable position count, Five Field Kono is the largest game that cannot be non-trivially separated into tiers that has ever been strongly solved in GamesCrafters. We hope that our work paves the way for additional optimizations that can be incorporated into the current GameCrafters infrastructure and support games with similar traits.

Acknowledgments

- Professor Dan Garcia, for allowing me to join his research group and fostering my, and this project's, development.
- GameCrafters lead, Cameron Cheung, for showing me the ropes of overcoming many challenges.
- GameCrafters veteran, Max Fierro, for helping with the backend and frontend implementation of the game and the canonical position analysis.
- GamesCrafters veteran, Robert Shi, for walking me through the deep technical walkthrough of the backend codebase.
- My friends and family, for supporting me through my college career.

Contents

Chapter 1

Introduction

GamesCrafters is an undergraduate research group founded by Professor Dan Garcia in 2001 to solve two-player, perfect information abstract strategy games. We made modifications to three major parts of the GamesCrafters GAMESMAN infrastructure:

- **GamesmanClassic**: A collection of games solved in C.
- **GamesCraftersUWAPI** (Universal Web API): The interface to serve board states, move values, and remoteness values from GamesmanClassic to GamesmanUni.
- **GamesmanUni**: An online Graphical User Interface to show games solved in GamesmanClassic (and solved by other solvers).

Figure 1.1: GAMESMAN Infrastructure

The Five Field Kono Solver mainly focuses on utilizing the lowest layer, GamesmanClassic. The aim is to complete the foundational layer for solving the game in GamesmanClassic and then utilize the already available API in the GamesCraftersUWAPI to visualize it on the GamesmanUni website.

This report first describes how we analyzed Five Field Kono by identifying the runtime and memory requirements for the solver. We then dive deeper into the architecture of the solver and the API we designed. Finally, we explore integration made to GamesmanUni for the final visualization of the game online.

Chapter 2

Background

In this section, we cover the rules of Five Field Kono, the main GamesmanClassic application programming interfaces that Five Field Kono interacts with – namely, the TextUI, Main Driver, Retrograde Loopy Solver, and Two-Bytes-Per-Position Database – and frontend work necessary to render the game on GamesmanUni.

2.1 Solving

When we strongly solve a game, we identify the game's outcome from each position of the game, assuming both Players play perfectly. When a Player plays perfectly, they win as fast as possible; and if they are unable to force a win, they aim to tie or draw; and if they cannot force a tie or draw, they aim to lose as slowly as possible. Our definition of perfect play identifies tying moves that end the game in a tie as fast as possible as better moves than tying moves that lead to a game that ends in more moves.

Tier-solving is a strategy for solving a game that involves partitioning the positions of the game into subspaces such that these subspaces form a directed acyclic graph after defining directed edges from each subspace A to each subspace B if there exists a position in A with a child position in B. We refer to these subspaces as *tiers*. The tiers are then solved in an order such that a tier is solved after all of its child tiers have been solved. In some cases, tier-solving can be useful for introducing checkpoints during a solve and reducing memory usage by eliminating the need to keep track of multiple positions across the game. Unfortunately, there does not exist a nontrivial tier definition for Five Field Kono, i.e., there is no way to partition positions into tiers that form a DAG so that nontrivial positions are separated into multiple tiers. As of the writing of this report, Five Field Kono is the largest game (in terms of position count) that does not have a nontrivial tier definition that has ever been solved in GamesCrafters.

2.2 Five Field Kono

Five Field Kono is a traditional Korean game [1] played on a five-by-five board. Player 1 owns the seven white pebbles, and Player 2 owns the seven black pebbles. None of the pebbles can be removed from the board at any point in the game. The game's rules are described as follows [2]...

1. Define the board as a list of x- and y-coordinates where the (0, 0) coordinate represents the top-right corner point of the board. Player 1 has white pebbles on $(0,3), (0,4), (1,4), (2,4), (3,4), (4,4)$, and $(4,3)$; while Player 2 has black pebbles on $(0,1), (0,0), (1,0), (2,0), (3,0), (4,0)$, and $(4,1)$.

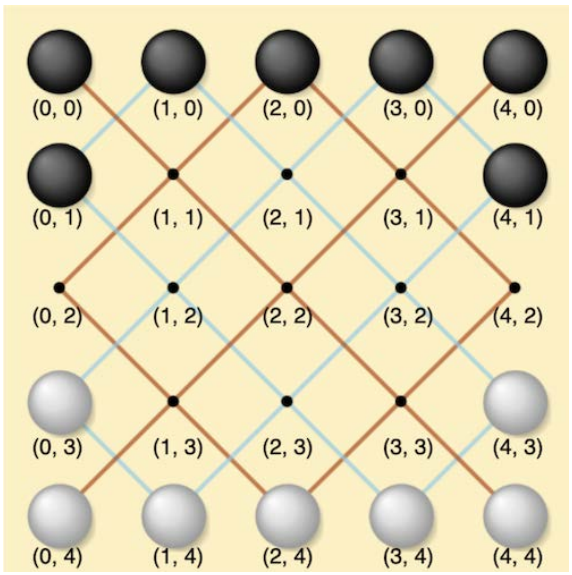

Figure 2.1: Five Field Kono board with labeled coordinates.

2. A Player can move one of their pieces diagonally (like a chess Bishop) to an adjacent point northeast, northwest, southwest, or southeast. This means a piece at an arbitrary point (x, y) can reach points $(x+1, y+1), (x+1, y-1), (x-1, y+1), (x-1, y-1)$ as long as the diagonal path is available (i.e., the destination point exists and is unoccupied).

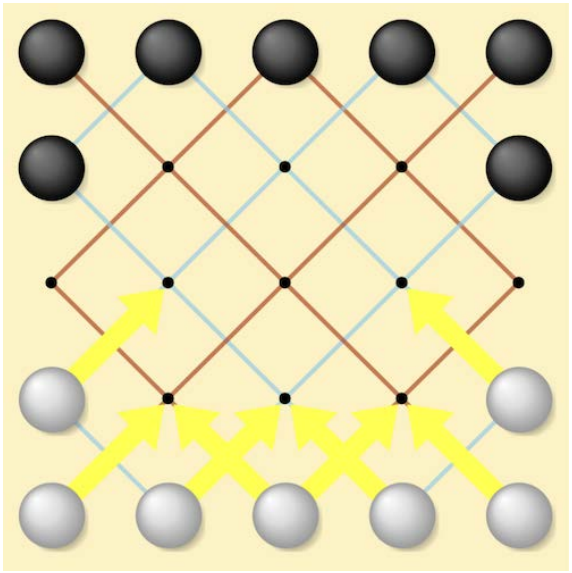

Figure 2.2: The yellow arrows indicate the available paths for Player 1's white pebbles.

3. A piece cannot move into or jump over a point occupied by another piece.

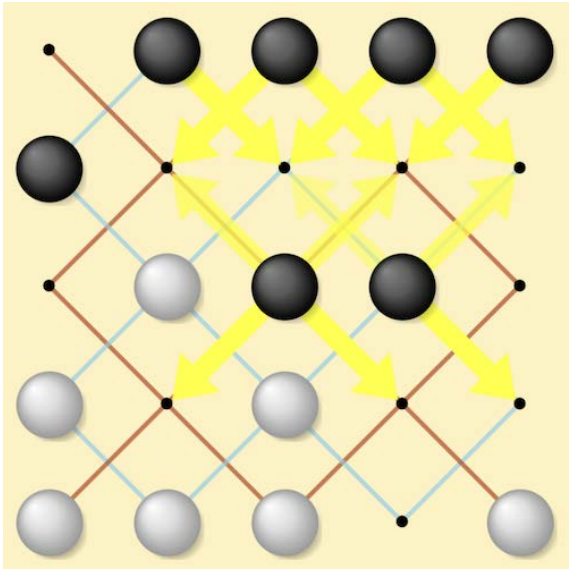

Figure 2.3: Black pebble at $(3, 2)$ is unable to move to the occupied point $(2, 3)$.

4. If none of the Player's pebbles can move, the game is a tie. For example, if Player 2's turn, but there are no available moves, the game is a tie (even if Player 1 still has moves available).

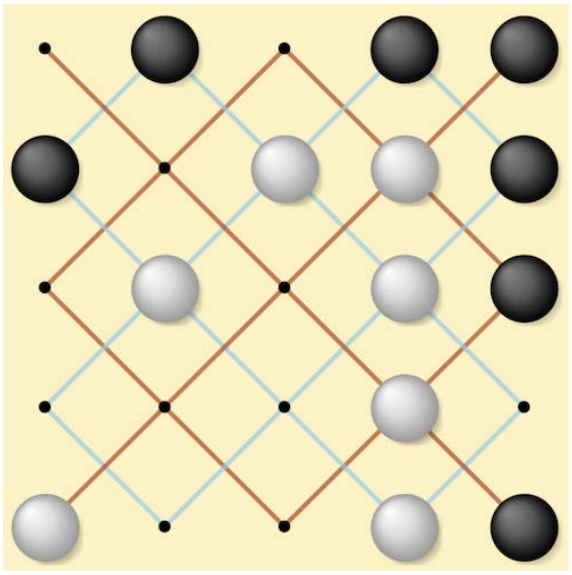

Figure 2.4: It is black's turn, and there are no legal moves, so this position is a tie.

5. If a Player manages to move all of their pebbles to the starting points of the opponents' pebbles, then the Player wins.

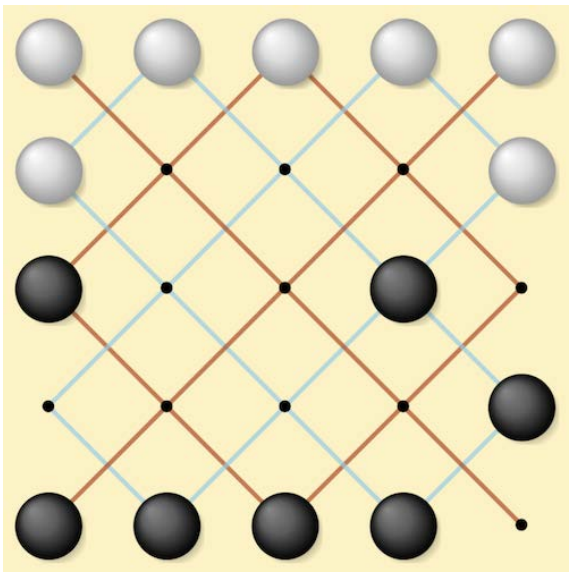

Figure 2.5: Player 1 win because the white pebbles now occupy Player 2's pieces' starting points.

The rules satisfy the three requirements for a game to be solvable in GamesmanClassic [3], which also makes the research objectives viable:

- **Two-Player**: White pebbles represent Player 1, and black pebbles represent Player 2. Either Player can go first; after that, they alternate, making moves.
- **Perfect Information**: All Players know all there is to know about the game simply by looking at the board (i.e., there is no hidden information).
- **Finite**: The number of possible board orientations in the game is finite since there are 25 slots with 7 white and 7 black pebbles to move to. With naive combinatorics, this gives a total of $\binom{25}{7}\binom{18}{7} = 15297796800 \approx 15.3$ billion positions (this includes unreachable board states).

2.3 Main Driver

The main driver [4] is the orchestrator for running the game on GamesmanClassic. It takes in the parameters set for the game and connects the 3 necessary components (database, game script, and solver) for users to interact and play. The components it connects for us in this game are the Five Field Kono game rules encoded as C routines, the Retrograde Loopy Solver, and the Two-Bytes-Per-Position database. The main driver also handles the text-based user interface that parses and handles the user commands.

2.4 Text User Interface (TextUI)

The TextUI handles all the user input commands typed on the terminal. We validate user-inputted move strings and convert them to an internal move representation. The internal move representation is then applied to the internal board representation, called position (covered in 3.3.3), leading to a new position and available moves. Finally, we implement a method that uses character art to draw the new position in the game in human-readable form on the terminal [3].

2.5 Two-Bytes-Per-Position Database

The Two-Bytes-Per-Position (TBPP) Database [5] is a customized database used by the Retrograde Loopy Solver in GamesmanClassic. Three definitions are essential to the TBPP Database:

- **Remoteness**: The number of moves left until the end of the game if both Players play perfectly.
- **Value**: Win, Lose, or Tie.
- **Canonical Position**: A position representing all other symmetric positions obtained from symmetry transformations applied to the current position p. This position p will have the smallest hash value out of the symmetric group, and all the other symmetric positions will be considered non-canonical.

Those three definitions are important so that the TBPP Database can use a "Write Once Read Many" (WORM) techniques for compression. As many games solved on GamesmanClassic range from 1 billion to 20 billion positions, storing only canonical positions is crucial to making normal games easily solved on a user's local laptop with a limited amount of RAM, to say nothing of the storage savings for the database.

The compression algorithm works as follows:

1. Through a canonical processing function in the game code, the TBPP Database receives the canonical position and stores $(remoteness, value)$ for that position.
2. All other positions that are considered non-canonical in this symmetric set will be filtered out by the canonical processing function and not received by the TBPP Database since the remoteness and value pair will be the same.
3. We initialize every position in our TBPP Database to 0. Since we will not be visiting non-canonical positions, our TBPP Database will have all non-canonical positions' bits set to 0, which results in many sequences of repeated 0s. The TBPP Database then uses gzip to compress the database into an index file and a data file to look up any hash positions immediately. The more consecutive 0s there are, the better the data compression will be, as the repeated bits of 0 can be unified.
4. For the canonical positions, it will use 8 bits for remoteness and 2 bits for the value. With the byte padding, it will use up 16 bits per position, where the remaining 6 bits are unused.

2.6 Retrograde Loopy Solver

The Retrograde Loopy Solver [6] handles loopy games like Five Field Kono. This solver was chosen since the game tree for Five Field Kono was cyclic (i.e., had loops).

The Retrograde Loopy Solver mechanism can be split into three phases:

Phase 1:

1. The Retrograde Loopy Solver first receives the total number of possible positions.
2. It then initializes a game state graph from the parent to all the descendants with the number of nodes equaling the total possible number of positions which is then stored in the heap memory.
3. Each position (node of the graph) is represented by its hashed value.
4. The Retrograde Loopy Solver now scans through the hash value in each node of the graph and checks for whether it's a primitive position by unhashing it into a board orientation and checking if `Primitive` returns anything but `Undecided`.
   (a) If it is a primitive position, it will be pushed to our frontier list and then marked as resolved. The remoteness and primitive value of this resolved position will be sent to the TBPP Database.
   (b) If not, it will keep track of its child count by seeing how many possible child positions it has.
5. As soon as all position nodes are scanned, our frontier list is complete with all the nodes having the number of child nodes they have.

Phase 2:

1. The Retrograde Loopy Solver will now iterate through the frontier list it has created.
2. At each position p encountered in the frontier list, it decrements p's parent positions' child count to indicate one of its child counts (namely p) has been resolved.
3. Once a parent position's child count has been decremented to 0, all of its child nodes have been resolved, and now the Retrograde Loopy Solver learns the primitive value of the parent's position. The parent's position is marked as resolved and is added to the frontier list, and the remoteness and primitive value of this parent's position are sent to the TBPP Database.
4. Once the frontier list is empty from resolving through all reachable positions, it labels all unresolved and unreachable positions as a draw.

Phase 3:

1. Since the game has been solved, the TBPP Database is filled with all the positions' remoteness and primitive value and will produce compressed .gz files for the index and the data.

The major information that the game code will need to provide for the Retrograde Loopy Solver is the following:

- **Board Definition**: Phase 1 and Phase 2
- **The Upper Bound of Total Positions in Five Field Kono**: Phase 1
- **Hashing and Unhashing Scheme**: Phase 1

2.7 Automated Graphical User Interface (AutoGUI)

After the Backend (the game is solved in GamesmanClassic) work has been completed, the Frontend work can be done through AutoGUI [7]. The Frontend works by the user making a position request through the GamesmanUni. This position request then goes from the GamesmanUni to the GamesCrafters UWAPI Server (as in figure 1.1), where the server handles how to translate and respond to the user by invoking GamesmanClassic APIs (game state, Player, and opponent's pebble positions, and database). Aside from the previously completed Backend work, additional work for the AutoGUI APIs is required to ensure the user sees rendered images of the black and white pebbles moving in the desired direction.

2.8 Machine

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

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

Chapter 3

Analysis, Architecture, and Implementation

This section focuses on the analysis done on the game to make it suitable for solving by the Retrograde Loopy Solver, as well as a description of the overall implementation of the game movement rules and primitive value classification.

3.1 Board Definition

3.1.1 Even and Odd Components of the Board

The important thing to notice in the Five Field Kono board was that there exist *two separate boards* that are laid on top of one another. This means that the pieces on the two separate boards can never occupy each other's space, eliminating game states where the pieces of board 1 are on board 2.

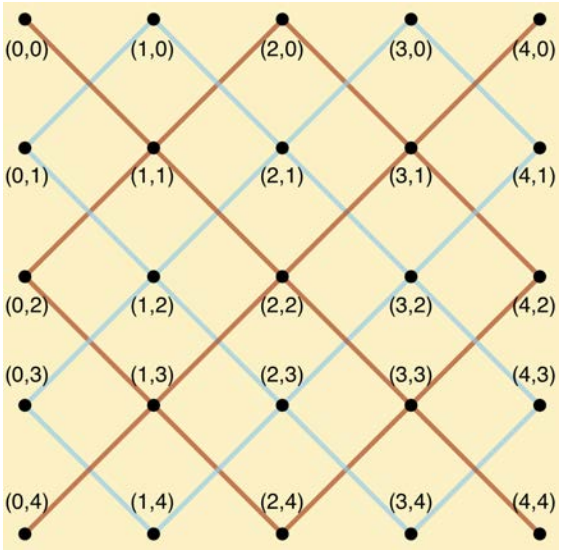

Figure 3.1: Five Field Kono board with labeled coordinates, without the pebbles.

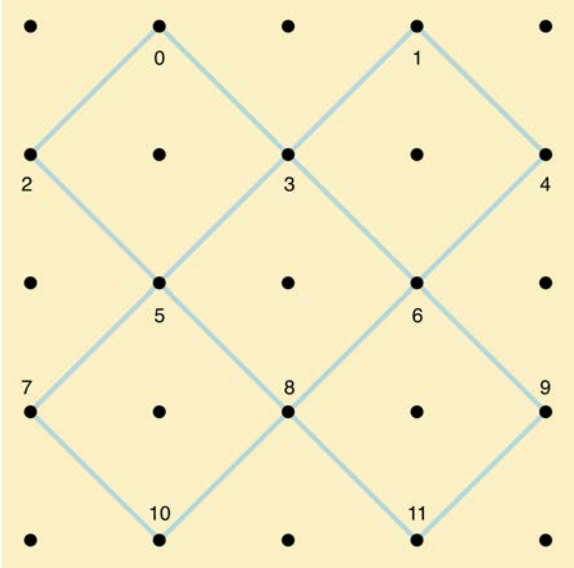

Figure 3.2: The even component of the Five Field Kono Board numbered from 0 to 11.

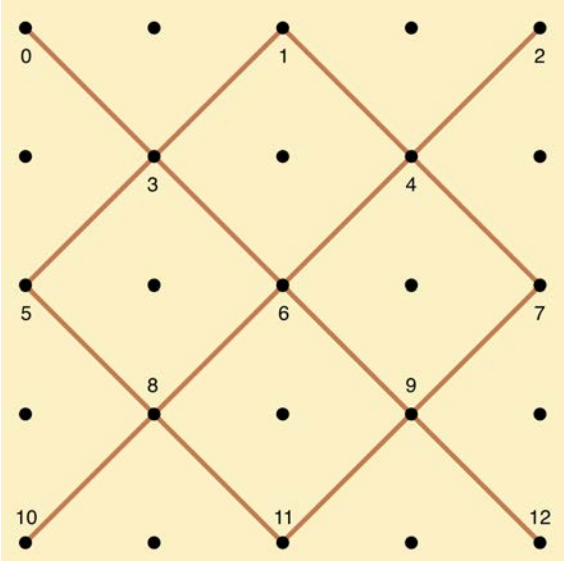

Figure 3.3: The odd component of the Five Field Kono Board numbered from 0 to 12.

From Figures 3.2 and 3.3, it is very clear that none of the even and odd components of the board clash with each other. In addition to separating the board into even and odd components, we defined each coordinate of the board into non-negative integers that start from the upper left corner of the board and end at the bottom right corner of the board (meaning that the number increments as you read the board in a zigzag fashion). Three advantages came with defining the even and odd components this way. First, regardless of which character array we have for the even or odd component, we can map the array back into the appropriate coordinates on the board and vice versa using a predefined bijection array. Secondly, this eliminated the complexity of developing a hashing algorithm for Player 1 and 2's pebble positions since we would have needed to specify how to combine the x and y coordinates if we used a coordinate system. Finally, this proved to be very useful for defining how moves for Players 1 and 2 are generated (covered in 3.4.1), making the hashing of the move information much easier.

The numbering system can be defined as the following:

- `even_num_sys` = $\{0:(1,0), 1:(3,0), 2:(0,1), 3:(2,1), 4:(4,1):, 5:(1,2):, 6:(3,2), 7:(0,3), 8:(2,3), 9:(4,3), 10:(1,4), 11:(3,4)\}$
- `odd_num_sys` = $\{0:(0,0), 1:(2,0), 2:(4,0), 3:(1,1), 4:(3,1), 5:(0,2), 6:(2,2), 7:(4,2), 8:(1,3), 9:(3,3), 10:(0,4), 11:(2,4), 12:(4,4)\}$

With this numbering system, the even component can be a 12-character long array with 4 empty spaces, 4 black pebbles, and 4 white pebbles placed in `even_num_sys[0]` to `even_num_sys[11]`, while the odd component can be a 13-character long array with 7 empty spaces, 3 black pebbles, and 3 white pebbles placed in `odd_num_sys[0]` to `odd_num_sys[12]`.

3.1.2 Even and Odd Component Move Generation

In section 3.1.1, we define a point numbering system for the even and odd components and discuss how this removed a lot of complexity. There is a universal method for generating moves from the pebble's current position. Five Field Kono allows a pebble to move from (x, y) to $(x+1, y+1), (x-1, y+1), (x-1, y-1), (x+1, y-1)$ and if we use one of the points in the even component as an example such as $(2, 3) = 8$, it will target $(3, 4) = 11, (3, 2) = 10, (1, 4) = 6, (1, 2) = 5$ as the destination points.

We witness a pattern and set up the following rule for finding a piece's destination points that apply to both `even_num_sys` and `odd_num_sys`. For a given point (x, y) containing a piece,

1. If there exists an upward left diagonal to $(x-1, y-1)$ and no piece exists at $(x-1, y-1)$, there's a new destination point at $p\prime = p - 3$.
2. If there exists an upward right diagonal path from (x, y) to $(x+1, y-1)$ and no piece exists in that position, there's a new position at $p\prime = p - 2$.
3. If there exists a downward left diagonal path from (x, y) to $(x-1, y+1)$ and no piece exists in that position, there's a new position at $p\prime = p + 2$.
4. If there exists a downward left diagonal path from (x, y) to $(x+1, y+1)$ and no piece exists in that position, there's a new position at $p\prime = p + 3$.

3.1.3 Primitive Position

This is when the game is determined to be "over" based on the game rules. Recalling the rules of Five Field Kono, such instances happen when one Player has no more moves left or if one of the Players' pieces has occupied the positions of the opponent's original setup positions. As a result, assuming it's White's turn, the primitive positions are defined as follows:

- `Lose` = {All the Black's pebbles occupy White's starting points: $(0, 3), (0, 4), (1, 4), (2, 4), (3, 4), (4, 4), (4, 3)$}
- `Tie` = {No moves available for White}

... and similarly, if it's Black's turn – it'll be a lose when White has just placed all their pebbles at the top, and a tie if Black has no moves available.

3.1.4 Canonical Positions

We calculate canonical positions so we can compress the space of the TBPP Database as much as possible through gzip. We store *only* canonical positions in our database. There are a total of 8 symmetries, so multiple possible symmetric positions can be found by applying symmetry transformations to a canonical position. Before introducing the symmetry transformations, we define the following:

- B_{even} is the even component of the board.
- B_{odd} is the odd component of the board.
- B_{turn} is the board turn information of the Players
- B_{color} is the board color information of the Players' pieces
- F_x is a transformation that takes a board component as input and flips it across the $x\text{-}axis$.
- F_y is a transformation that takes a board component as input and flips it across the $y\text{-}axis$.
- T is a transformation that takes a position as input and switches the turn.
- C is a transformation that switches Player 1 and Player 2's color

There is an interesting property in Five Field Kono's symmetry transformations. These transformations form an Abelian group. An Abelian group is defined by four axioms [8]:

1. **There exists an identity element in the group**: The board position giving the lowest hash (the canonical position) will be the identity element in the group.
2. **Multiplication between elements of the group is associative**: When multiple of the same symmetry operation is applied to the board position, the same result is obtained regardless of rearranging the parentheses to change the priority of group operations. For example, $(F_x + F_y) + T + C = F_x + (F_y + T) + C = F_x + F_y + (T + C)$ similar to $(1 + 2) + 3 + 4 = 1 + (2 + 3) + 4 = 1 + 2 + (3 + 4)$.
3. **Each element of the group has an inverse which also belongs to the group**: Each of the symmetry transformation's inverse is the operation itself as they affect different things. Flipping one of the board components twice on the same axis will not change anything about the board. Also, switching colors and turns twice will retain the original board.
4. **Multiplication between elements of the group is commutative**: It's important to recognize that swapping turn and swapping color on even and odd each affect different things, so these operations are commutative. And lastly, a flip by the $y\text{-}axis$ commutes with a flip by the $x\text{-}axis$.

The following are the possible transformations of the identity board:

1. $(Identity) : B_{\text{even}}B_{\text{odd}}B_{\text{turn}}B_{\text{color}} \to B_{\text{even}}B_{\text{odd}}B_{\text{turn}}B_{\text{color}}$
2. $(F_y \to B_{\text{odd}}) : B_{\text{even}}B_{\text{odd}}B_{\text{turn}}B_{\text{color}} \to B_{\text{even}}B^{y}_{\text{odd}}B_{\text{turn}}B_{\text{color}}$
3. $(F_y \to (B_{\text{even}})) : B_{\text{even}}B_{\text{odd}}B_{\text{turn}}B_{\text{color}} \to B^{y}_{\text{even}}B_{\text{odd}}B_{\text{turn}}B_{\text{color}}$
4. $(F_y \to (B_{\text{odd}} \wedge B_{\text{even}})) : B_{\text{even}}B_{\text{odd}}B_{\text{turn}}B_{\text{color}} \to B^{y}_{\text{even}}B^{y}_{\text{odd}}B_{\text{turn}}B_{\text{color}}$
5. $(T + C + (F_x \to (B_{\text{odd}} \wedge B_{\text{even}}))) : B_{\text{even}}B_{\text{odd}}B_{\text{turn}}B_{\text{color}} \to B^{x}_{\text{even}}B^{x}_{\text{odd}}B^{*}_{\text{turn}}B^{*}_{\text{color}}$
6. $(T + C + (F_x \to (B_{\text{odd}} \wedge B_{\text{even}})) + (F_y \to (B_{\text{odd}}))) : B_{\text{even}}B_{\text{odd}}B_{\text{turn}}B_{\text{color}} \to B^{x}_{\text{even}}B^{xy}_{\text{odd}}B^{*}_{\text{turn}}B^{*}_{\text{color}}$
7. $(T + C + (F_x \to (B_{\text{odd}} \wedge B_{\text{even}})) + (F_y \to (B_{\text{even}}))) : B_{\text{even}}B_{\text{odd}}B_{\text{turn}}B_{\text{color}} \to B^{xy}_{\text{even}}B^{x}_{\text{odd}}B^{*}_{\text{turn}}B^{*}_{\text{color}}$
8. $(T+C+(F_x \to (B_{\text{odd}} \wedge B_{\text{even}}))+(F_y \to (B_{\text{odd}} \wedge B_{\text{even}}))) : B_{\text{even}}B_{\text{odd}}B_{\text{turn}}B_{\text{color}} \to B^{xy}_{\text{even}}B^{xy}_{\text{odd}}B^{*}_{\text{turn}}B^{*}_{\text{color}}$

Symmetry transformation applied to the identity board forms the following graph where associative and commutative properties are shown (where the purple arrow represents the F_y transformation on one of the board components, and the green arrow represents the T, C, and F_x transformation on both board components):

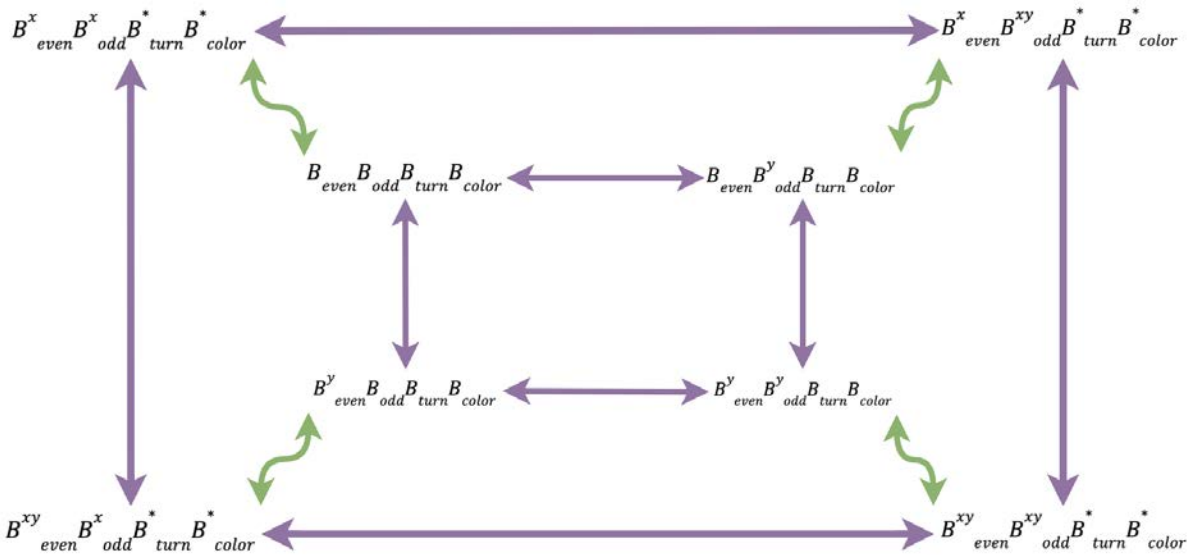

Figure 3.4: Symmetry transformation of the board with the purple arrow denoting *y-axis* transformation and the green arrow denoting *x-axis*, turn, and color transformation.

3.2 Upper Bound on the Number of Reachable Positions

We notice that the board can be split into two components – even and odd – because the pieces can only occupy certain points since they are limited to moving diagonally. A piece in one component can never move to a point in the other component.

The even component has 12 slots with 4 white pebbles and 4 black pebbles. The odd component has 13 slots with 3 white pebbles and 3 black pebbles. The number of ways to arrange white pebbles and black pebbles among a given number of slots is $\frac{(\text{total number of slots})!}{(\text{white pebbles})!\times(\text{black pebbles})!\times(\text{empty slots})!}$. Thus, there are $\frac{12!}{4!\times4!\times4!} = 34650$ ways to arrange the even component and $\frac{13!}{3!\times3!\times7!} = 34320$ ways to arrange the odd component. An upper bound for the number of positions is $34650 \times 34320 = 1189188000 \approx 1.2$ billion. This gives a far better upper bound than applying naive combinatorics to the entire board, which gave 15.3 billion positions, improving $\frac{15297796800}{1189188000} \approx 12.9$ times. The main improvements come from eliminating all arrangements of the board in which the incorrect number of pieces of each color occupy the even component and the odd component since – as mentioned earlier – pieces from the even component can never reach points on the odd component, and vice-versa. We decided to increase the total number of possible positions by $1189188000 \times 2 = 2378376000$ as the turn information needed to be included (discussed in the hashing and unhashing section).

One thing to note is that there are still positions unreachable after the compression by separating the board into even and odd components. For example, both Players' pieces cannot simultaneously be in the destination slots since the game is over once one Player has moved them into them. As a result, there will be many unvisited positions; these will be stored as a 0 in the TBPP Database since all positions are initially set to 0.

3.3 Hashing and Unhashing

3.3.1 Position Hashing

We first naively used a 25-length ternary number to represent the board since each slot is in one of three possible states (empty, occupied by a white pebble, or occupied by a black pebble). The maximum hash from this hashing algorithm is $3^{25} = 847288609443 \approx 8.5$ billion. Many of the states encoded within the

hash space (i.e., hash values 0 to 3^{25}) correspond to unreachable states, so most entries would not be utilized in the TBPP Database with this naive approach.

We instead use the Rearranger Hashing Algorithm [9] to hash the even and odd components separately and combine the hashes of each component into a final board hash. This was a more hash-efficient (i.e., the ratio of reachable states to total states encoded is higher) than the naive approach; the hash space contained 1189188000 states.

We now explain the rearranger hash by example. Suppose we wish to hash the even component. There are 4 O's and 4 X's in the even component. There are a total of $N = \frac{12!}{4!4!4!}$ total ways one can rearrange the 4 O's and 4 X's (and 4 blanks) among the 12 slots. We wish to bijectively assign each arrangement a number in $\{0, 1, 2, ..., N\}$. Specifically, the exact hash value for a given even component configuration is the place it appears in an "alphabetical" ordering of all possible even components. If we were to alphabetically order all possible even components (treating blank as alphabetically before O, which is alphabetically before X), then the order is as follows, with each even component arrangement numbered according to hash value:

```
    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----
```

To retrieve the bijective mapping from the even or odd component orientation to the rearranger hash, two threshold values need to be calculated using the formula $\frac{(\texttt{num\_slots})!}{(\texttt{num\_O})!\times(\texttt{num\_X})!\times(\texttt{num\_slots - num\_O - num\_X})!}$ where (`num_slots`) is the number of slots left to explore, (`num_O`) is the number of `O` left to explore, and (`num_X`) is the number of `X` left to explore in the even or odd component array. Given `num_slots` $= 10$, `num_O` $= 3$, and `num_X` $= 4$ in the array, `threshold_1` equals to $\frac{(\texttt{num\_slots}-1)!}{(\texttt{num\_O})!\times(\texttt{num\_X})!\times(\texttt{num\_slots - 1 - num\_O - num\_X})!} = \frac{9!}{4!\times4!\times1!} = 630$, while `threshold_2` equals to `threshold_1` $+ \frac{(\texttt{num\_slots}-1)!}{(\texttt{num\_O}-1)!\times(\texttt{num\_X})!\times(\texttt{num\_slots - 1 - (num\_O - 1) - num\_X})!} = 630 + \frac{9!}{3!\times4!\times2!} = 630 + 1260 = 1890$. If the $(\texttt{num\_slots} < 0) \vee (\texttt{num\_O} < 0) \vee (\texttt{num\_X} < 0) \vee (\texttt{num\_slots} < (\texttt{num\_O} + \texttt{num\_X}))$, the threshold formula will return a 0. This is demonstrated below, as two thresholds are calculated every time we progress each slot of the even or odd component array.

Algorithm 1 Rearranger Hashing Algorithm For Five Field Kono [A.1].

Take the `board_component`, the number of `num_O`, and the number of `num_X` as inputs.
Initialize `total_hash` to 0 and `num_slots` to `len(board_component)`
Let `REMAINING`, the remaining character array, consist only of `num_X` of '`X`', `num_O` of '`O`', and (`num_slots` − `num_X` − `num_O`) of '`-`'.
for `curr_idx` = 0 to `len(board_component) - 1` **do**
    Set `curr_piece` to `board_component[curr_idx]`.
    Set `threshold_1` to the index of the alphabetically first rearrangement of `REMAINING` from the index `curr_idx` to `len(board_component) - 1` that starts with the black pebble (`O`).
    Set `threshold_2` to the index of the alphabetically first rearrangement of `REMAINING` from the index `curr_idx` to `len(board_component) - 1` that starts with the white pebble (`X`).
    if the `curr_piece` is `O` **then**
        Add `threshold_1` to `total_hash`
        Decrement `num_O`.
    else if the `curr_piece` is `X` **then**
        Add `threshold_2` to `total_hash`
        Decrement `num_X`.
    end if
    Decrement `num_slots`.
    Remove one character matching `curr_piece` from `REMAINING`.
end for
Return `total_hash`.

This will give a hash from the range 0 to $\frac{(\texttt{total\_num\_slots})!}{(\texttt{total\_num\_O})!\times(\texttt{total\_num\_X})!\times(\texttt{total\_num\_slots - total\_num\_O - total\_num\_X})!}$ for each even and odd component. Effectively, we get $max\_even\_hash = \frac{12!}{4!\times4!\times4!} = 34650$ $(0 < even\_hash < 34650)$ for the even component and $max\_odd\_hash = \frac{13!}{3!\times3!\times7!} = 34320$ $(0 < odd\_hash < 34320)$ for the odd component, which we piece together as if it's a 2-dimensional coordinate system of $(even\_hash, odd\_hash)$. With such a coordinate system, it will attain $34650 \times 34320 = 1189188000$ possible coordinates ranging from 0 to 1189187999 using $even\_hash$ base representation of $odd\_hash \times even\_hash + even\_hash = final\_hash$.

One thing to note is that Five Field Kono is a game where Players must take turns. As a result, whenever we receive the hash for a position, we must obtain the turn information from the unhash. As a result, we use the tuple $(even\_hash, odd\_hash, turn)$ to represent a position, which works out as $(2 \times even\_hash)$ base representation: $base_{2\times \max\_even\_hash}\langle odd\_hash, (max\_even\_hash \times turn) + even\_hash\rangle = (odd\_hash \times 2 \times max\_even\_hash) + ((max\_even\_hash \times turn) + even\_hash) = final\_hash$. We set the total possible positions to $(max\_even\_hash \times max\_odd\_hash) \times 2 = 1189188000 \times 2 = 2378376000$.

3.3.2 Position Unhashing

Had we hashed the entire board's state and ignored storing whose turn it was, we would use a $(2 \times even\_hash)$ modulo to retrieve the $even\_hash$ and odd_hash of the board. The $odd\_hash = \lfloor (final\_hash \div (2 \times max\_even\_hash)) \rfloor$, while the $even\_hash = final\_hash \bmod (2 \times max\_even\_hash)$. To retrieve the turn information, we simply check whether the $even\_hash \geq max\_even\_hash$. If the turn equals 1, I can set the current board as Player 2's turn and set it as Player 1's turn if the turn equals 0. Afterward, we apply the Rearranger Unhashing Scheme to get the original state of the even and odd components. This is demonstrated below with the two thresholds calculation formula and utilization being the same as the position hashing from section 3.3.2.

Algorithm 2 Rearranger Unhashing Algorithm for Five Field Kono [A.1].

Take the `position_hash`, the component length `num_slots`, the number of black pebbles `num_O`, and the number of white pebbles `num_X` as inputs.
Initialize `component_arr`, a character array of length `num_slots`.
Initialize component index, `curr_idx`, as `0`.
while `num_slots` $>$ `0` **do**
  Set `threshold_1` to the index of the first rearrangement of the remaining character array from the index `curr_idx` to `len(component_arr) - 1` that starts with the black pebble (`O`).
  Set `threshold_2` to the index of the first rearrangement of the remaining character array from the index `curr_idx` to `len(component_arr) - 1` that starts with the white pebble (`X`).
  if the `position_hash` is smaller than `threshold_1` **then**
    Assign `-` to `component_arr[curr_idx]`
  else if the `position_hash` is bigger than or equal to `threshold_1` but smaller than `threshold_2` **then**
    Assign `O` to `component_arr[curr_idx]`.
    Subtract `threshold_1` from `position_hash`.
    Decrement `num_O`.
  else
    Assign `X` to `component_arr[curr_idx]`
    Subtract `threshold_2` from `position_hash`.
    Decrement `num_X`.
  end if
  Increment `curr_idx` by `1` and decrement `num_slots` by `1`.
end while
Return `component_arr`

3.3.3 Hashing and Unhashing Moves

Moves must be hashed so the Retrograde Loopy Solver can determine which child positions a parent node has and make appropriate game state transitions. Unlike the complexity involved in hashing the position of the board, the hashing scheme for the move was quite simple as we utilized a base 25 representation using the current position and the new position, which looks like $base_{25}\langle new\_pos, curr\_pos\rangle$. We would first need to define what each of the base 25 number maps to $\langle 24 :$ `odd_num_sys[12]`$, ..., 12 :$ `odd_num_sys[0]`$, 11 :$ `even_num_sys[11]`$, ..., 0 :$ `even_num_sys[0]`$\rangle$, position 24 of the base 25 hash represents the 12th index of the `odd_num_sys`, while position 0 of the base 25 hash represents the 0th index of `even_num_sys`. As a result, if we have a position change that occurs from `odd_num_sys[0]` $\rightarrow$ 12 to `odd_num_sys[3]` $\rightarrow$ 15, we will have a hash of $base_{25}\langle 15, 12\rangle = 15 \times 25 + 12 = 387$. Since the even and odd components of Five Field Kono are separate and never interact with each other, it would be easy to identify whether the move happened in the odd component or the even component by checking whether the current position is bigger than or equal to 12 (this will be discussed more in the next paragraph). Since the `MOVE` variable in the Retrograde Loopy Solver can take in a maximum hash of $2^{32} - 1 = 4294967295 \approx 4.3$ billion, our maximum hash representation is well suited for this requirement: $25^2 - 1 = 624 < 4294967295$.

To unhash, we apply the base 25 modulo to the `MOVE` hash to retrieve the *new_hash* and *curr_hash* from the $base_{25}\langle new\_pos, curr\_pos\rangle$. As a result, the $new\_pos = \lfloor \frac{move\_hash}{25} \rfloor$, while the $old\_pos = move\_hash \bmod 25$. Afterward, we check whether the $curr\_pos \geq 12$ to see if the `MOVE` occurred in the odd component or not (otherwise, the even component). If a `MOVE` occurs in the odd component, we subtract 12 from the *curr_pos* and *new_pos* to retrieve the correct `odd_num_sys` indexes. For the `even_num_sys`, we directly use both *curr_pos* and *new_pos* as they have not been incremented by 12 before hashing.

3.4 Architecture and Implementation

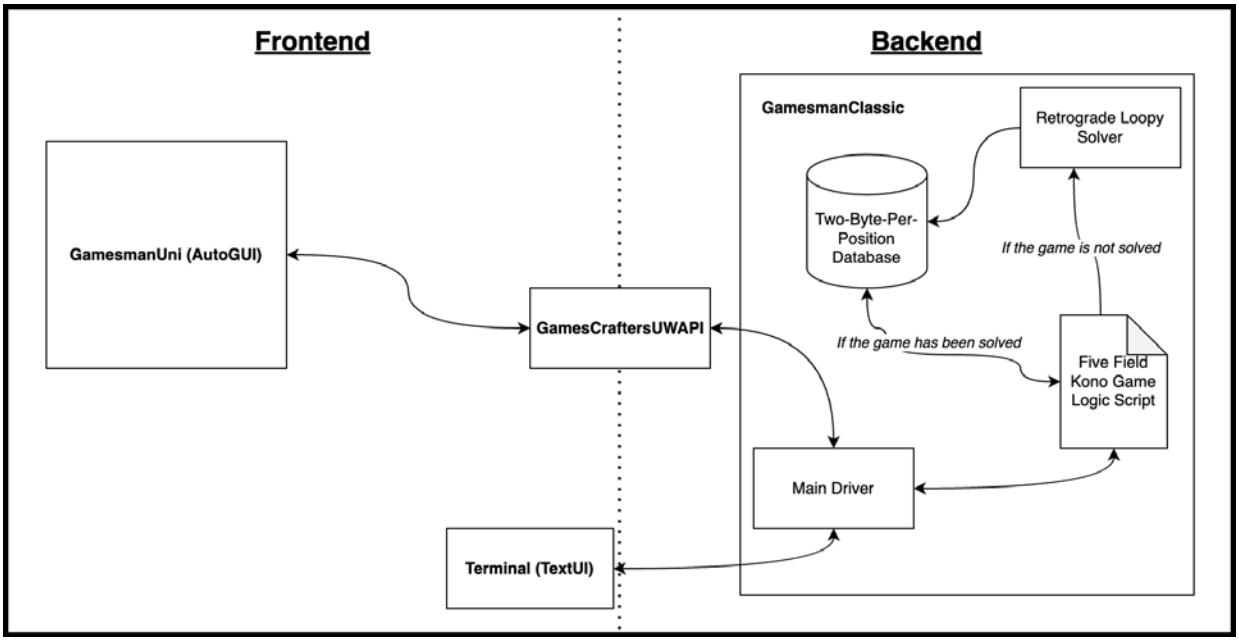

Figure 3.5: Architecture design behind Five Field Kono as a game service.

To bring the Five Field Kono Game into reality, we implemented APIs for the Five Field Kono Game Code [A.1], the Terminal TextUI [A.1][B], GamesCrafteresUWAPI [A.2], and GamesmanUni AutoGUI [A.1].

The architecture works the following way from the GamesmanUni:

1. When the user goes to the GamesmanUni Website to play Five Field Kono, it connects to the GamesCraftersUWAPI server, which then connects to the Main Driver of GamesmanClassic to initialize the necessary Five Field Kono Game Code, the Retrograde Loopy Solver, and the Two-Bytes-Per-Position Database.

2. A user decides to make a move in Five Field Kono, and this move request is sent to GamesCraftersUWAPI. The GamesCrafteresUWAPI processes this move request to be understandable by the Five Field Kono Game Script.

3. The Five Field Kono Game Code checks whether the game has been solved or not.

   (a) If it has been solved, it can directly connect to the Two-Bytes-Per-Position Database to retrieve the necessary information it needs.

   (b) If the game hasn't been solved, the Retrograde Loopy Solver will solve the game and send the fully solved game data to the Two-Bytes-Per-Position Database for the Five Field Kono Game Code to retrieve.

4. The Five Field Kono Game Code then sends the following data to the GamesCrafteresUWAPI server. The GamesCrafteresUWAPI then processes the data to make it understandable using graphics for the user.

The Terminal architecture is similar to the GamesmanUni Architecture except that it connects directly with the GamesmanClassic and the graphics rendering work being done there.

Overall, we spent about 2 months trying to complete the Five Field Kono Game Code and 1 week trying to complete TextUI, GamesCraftersUWAPI server, and AutoGUI. For the Five Field Kono Game Code, we had to convert all the analyses described above into code. For the TextUI, GamesCraftersUWAPI server, and AutoGUI, we had to make design choices by finding clean, appropriate, and easy-to-understand graphics for the user. As there were abundant examples of how to complete the Frontend and server work along with far fewer lines of code to write, the process took relatively shorter compared to the Backend work.

3.4.1 Board Drawing for Users

To make our even and odd components of the board visually understandable for the users, we decided to combine the odd and even component as follows:

Algorithm 3 Algorithm to combine the odd and even components to represent the Five Field Kono Board [A.1].

1: Create a 26-character array called `char_board`, which is a 1-dimensional representation of the Five Field Kono board using a similar numbering system to that of the even and odd components.
2: **for** `curr_idx` = `0` to `24` **do**
3: **if** `curr_idx` mod `2` equals to `0` **then**
4: Set `char_board[curr_idx]` to `odd_num_sys[`$\frac{\texttt{curr\_idx}}{2}$`]`.
5: **else**
6: Set `char_board[curr_idx]` to `even_num_sys[`$\frac{\texttt{curr\_idx - 1}}{2}$`]`.
7: **end if**
8: **end for**
9: Set `char_board[25]` to `NULL`.
10: Return `char_board`.

The visualization of the numbering system on the `char_board` to the `odd_num_sys` and `even_num_sys` looks like (where pink numbers are the full Five Field Kono Board's numbering system, brownish-orange numbers are the odd component's numbering system, and the blue numbers are the even component's numbering system):

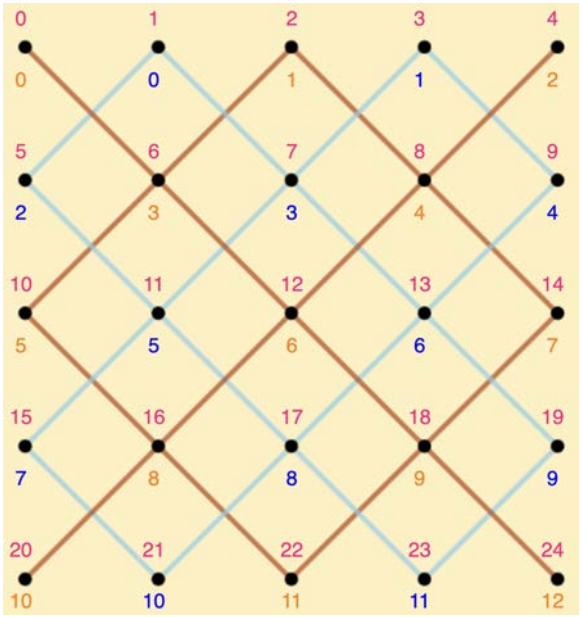

Figure 3.6: Five Field Kono Board numbered from 0 to 24 in pink; even component numbered from 0 to 11 in blue; odd component numbered from 0 to 12 in orange.

To retrieve the even and odd components' information from this 1-dimensional Five Field Kono Board, we applied the following transformation:

Algorithm 4 Algorithm to retrieve the odd and even components from the Five Field Kono Board [A.1].

1: Take in the 26-character array `char_board` as an input.
2: Create the `board` structure, which contains the `even_num_sys` and `odd_num_sys` arrays as attributes (also a `turn` attribute which is handled during the `MOVE` hashing and unhashing).
3: **for** `curr_idx` = `0` to `24` **do**
4: **if** `curr_idx` mod `2` equals to `0` **then**
5: Set `odd_num_sys[`$\frac{\texttt{curr\_idx}}{2}$`]` to `char_board[curr_idx]`.
6: **else**
7: Set `even_num_sys[`$\frac{\texttt{curr\_idx - 1}}{2}$`]` to `char_board[curr_idx]`.
8: **end if**
9: **end for**
10: Set `even_num_sys[12]` to `NULL`.
11: Set `odd_num_sys[13]` to `NULL`.
12: Return `board`.

3.4.2 Move Definition for Users

As previously mentioned, one of the game's goals was to make moves easy to understand and choose. For the GamesmanUni AutoGUI, this requirement was already satisfied as the user can directly interact with the board and move his or her pebbles to places using the click of the mouse. However, this was not the case for the Terminal's TextUI, where you had to type in the move to see the piece transition to your desired state. We figured that most users would have dealt with the popular board game Chess when they were young; hence we decided to use the same position system. This made the transition in the game states very easy to understand, along with figuring out which pieces are in what positions for someone not used to the (x, y) coordinate system.

As the final product, we configured the Terminal TextUI board to accept moves in the format of (Current Position)(Next Position) from a list of available moves.

For the board UI, we wanted to have the alphabet and numbers visible on the side of the board to follow a structure similar to the image below (for ease of identifying points, we also attached each position's $(alphabet)(number)$). We made sure to make the alphabet increment rightward on the $x\text{-}axis$, while the number increment upward on the $y\text{-}axis$ as the default option of the game makes the user start as Player 1, which is the bottom of the board. In addition, we used the 1-indexing system (starting from 1 instead of 0) for ease of reading the positions.

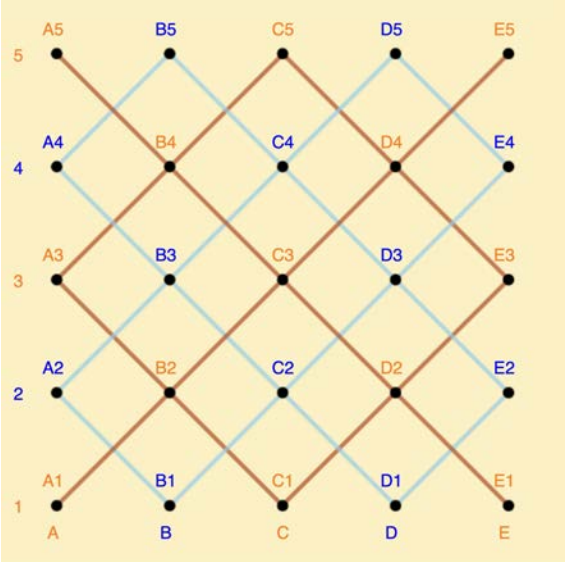

Figure 3.7: Five Field Kono Board with Chess' alphabet to number coordinate system.

Altogether, the user would insert (`current alphabet`)(`current number`)(`next alphabet`)(`next number`) in the Terminal's TextUI to make a move. For example, if the user inputs `a2b3` (or `A2B3`) into the terminal command line, it would make the user's pebble move from point A2 to B3 in Figure 3.7.

Chapter 4

Results

The results section will detail the Retrograde Loopy Solver's metrics to fully solve Five Field Kono, the graphical user interface on the Terminal, and the user interface on GamesmanUni.

4.1 Value-Remoteness Counts

We run a BFS from the initial position to determine which states are reachable. For each reachable position, we identify how many positions there are per value-remoteness pair. We also count the number of positions unique under symmetry.

Remoteness	Win	Lose	Tie	Total
0		7,830	8,088	15,918
1	28,786		74,974	103,760
2		33,356	98,292	131,648
3	90,078		836,175	926,253
4		73,187	1,762,485	1,835,672
5	196,417		8,213,820	8,410,237
6		101,578	15,835,795	15,937,373
7	263,244		48,127,638	48,390,882
8		83,979	83,630,471	83,714,450
9	226,373		166,270,067	166,496,440
10		36,641	223,515,021	223,551,662
11	107,690		254,371,720	254,479,410
12		8,286	191,565,143	191,573,429
13	31,588		104,416,494	104,448,082
14		1,523	38,165,498	38,167,021
15	9,334		7,817,877	7,827,211
16		784	981,962	982,746
17	5,172		65,422	70,594
18		412	2,564	2,976
19	2,368		36	2,404
20		188		188
21	762			762
22		68		68
23	136			136
24		6		6
Total	961,948	347,838	1,145,759,542	1,147,069,328

Table 4.1: Standard Five Field Kono Counts (Symmetries Not Removed)

The total number of reachable positions was interesting as it turned out to be $1147069328 \approx 1.15$ billion, which is about half ($1147069328 \div 2378376000 = 0.4822909952 \approx 0.5$) the upper bound we calculated for the number of possible positions. The number of board positions that were classified as a *tie* was 1145759542, which was about $1145759542 \div 1147069328 = 0.9988581457 \approx 99.886\%$ of the reachable positions, while the number of winning positions were $961948 \div 1147069328 = 0.0008386136535 \approx 0.084\%$ and losing positions were $347838 \div 1147069328 = 0.0003032406076 \approx 0.030\%$. The extremely large ratio of tie positions to win/lose positions shows how difficult it is to win or lose in the game. Other noteworthy observations would

be how the number of win positions available peaks at a remoteness value of 7 at 263244 while the lose positions peak at a remoteness value of 6 at 101578.

Remoteness	Win	Lose	Tie	Total
0		1,185	1,218	2,403
1	5,358		10,292	15,650
2		4,902	13,522	18,424
3	15,641		110,650	126,291
4		10,313	235,529	245,842
5	29,985		1,064,158	1,094,143
6		14,188	2,069,997	2,084,185
7	38,778		6,199,867	6,238,645
8		11,416	10,888,303	10,899,719
9	32,504		21,452,445	21,484,949
10		5,224	29,115,903	29,121,127
11	15,778		33,046,936	33,062,714
12		1,286	25,164,149	25,165,435
13	4,808		13,713,791	13,718,599
14		274	5,077,412	5,077,686
15	1,258		1,061,935	1,063,193
16		150	146,073	146,223
17	630		11,428	12,058
18		87	525	612
19	366		14	380
20		41		41
21	132			132
22		20		20
23	45			45
24		2		2
Total	145,283	49,088	149,384,147	149,578,518

Table 4.2: Standard Five Field Kono Counts (Symmetries Removed)

When the symmetries were removed from the reachable positions, the total number of positions shrank to 149578518, which reduced the original number of positions to $149578518 \div 1147069328 = 0.1304005907 \approx 13.040\%$. That's a comparable metric to if every canonical position had 8 symmetries, which would have given a $1/8 = 0.125 = 12.5\%$ reduction to the total reachable positions (when symmetries are removed). This entails how most reachable canonical positions have 8 symmetrical positions as the symmetry removal compresses the reachable positions to near 12.5%.

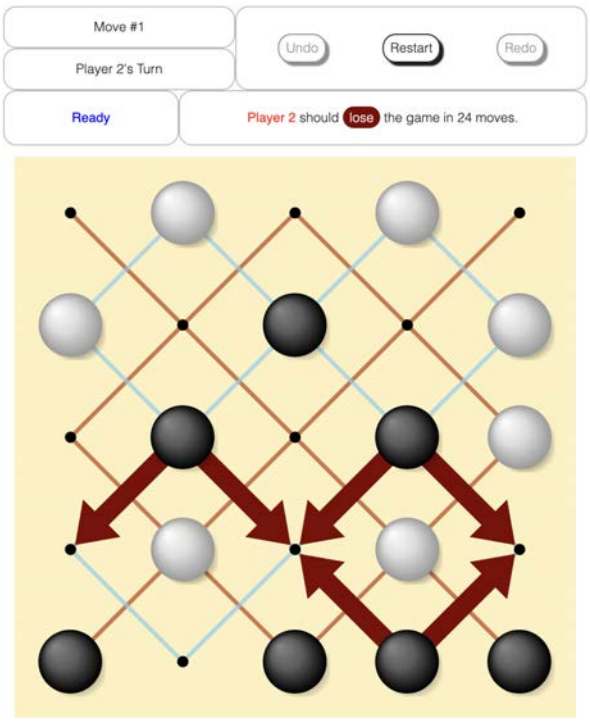

Figure 4.1: Five Field Kono Board Position with the Highest Remoteness.

The figure 4.1 shows the Five Five Kono Board we found using BFS to have the highest remoteness value, which corresponds to a losing move when the remoteness value is equal to 24. As there are 2 unique positions for a remoteness value of 24 with symmetries removed, we can find one more canonical position with the same remoteness value.

4.2 Terminal TextUI

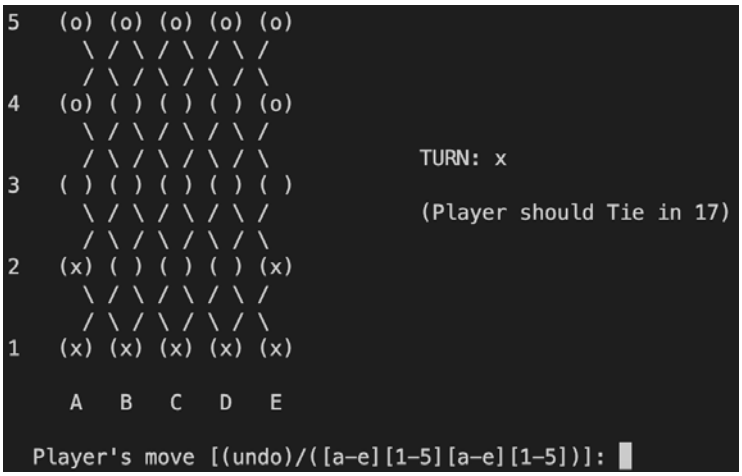

Figure 4.2: Terminal TextUI of the Five Field Kono Board [B].

The final TextUI had all the intricate details we envisioned before implementation (recall section 3.4.1). There is alphabetical labeling for the rows (ranks) and numerical labeling for the columns (files) for users

to figure out each bracket's position. We represented Player 1's white pebbles as "`x`", such that it's easily distinguishable from Player 2's black pebbles represented as "`o`". Along with those features, we've added the "`TURN`" at the board's right to indicate whose turn it is and made use of a library call to include the "prediction" of a position (shown as "`Player should Tie in 17`").

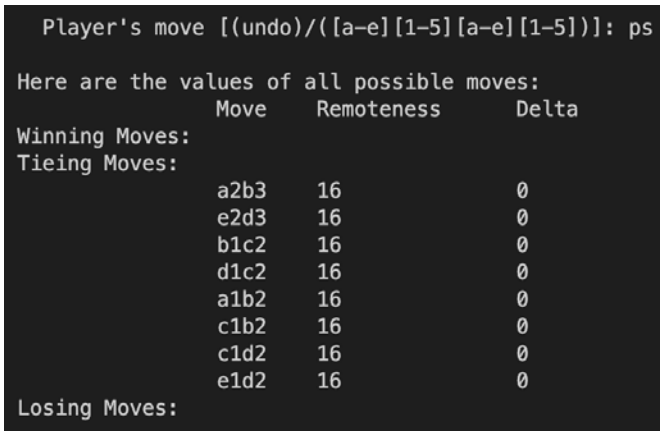

Figure 4.3: Terminal TextUI of the moves available in the Five Field Kono Board [B].

We added support for the user to put in `ps` when unsure what moves they can make. By typing in this command, they will be able to see the list of `Winning Moves`, `Tying Moves`, and `Losing Moves`. The `Remoteness` and `Delta` columns were also attached to the moves available such that users can know how many moves there are left until the game ends and also know which one is the better move (indicated by, the lower `Delta` value).

`Delta Remoteness` (shown as the `Delta` column) is a way to assign a number to all available moves such that the lower the number, the better it is. Low `Remoteness` wins have a lower `Delta Remoteness` than high `Remoteness` wins. High `Remoteness` wins have a lower `Delta Remoteness` than ties or draws. Ties or draws have lower `Delta Remoteness` than high `Remoteness` loses. High `Remoteness` loses have a lower than `Delta Remoteness` than low `Remoteness` loses. Since the `Remoteness` value is the same for all the moves, the `Delta` is the same for figure 4.3.

4.3 GamesmanUni AutoGUI

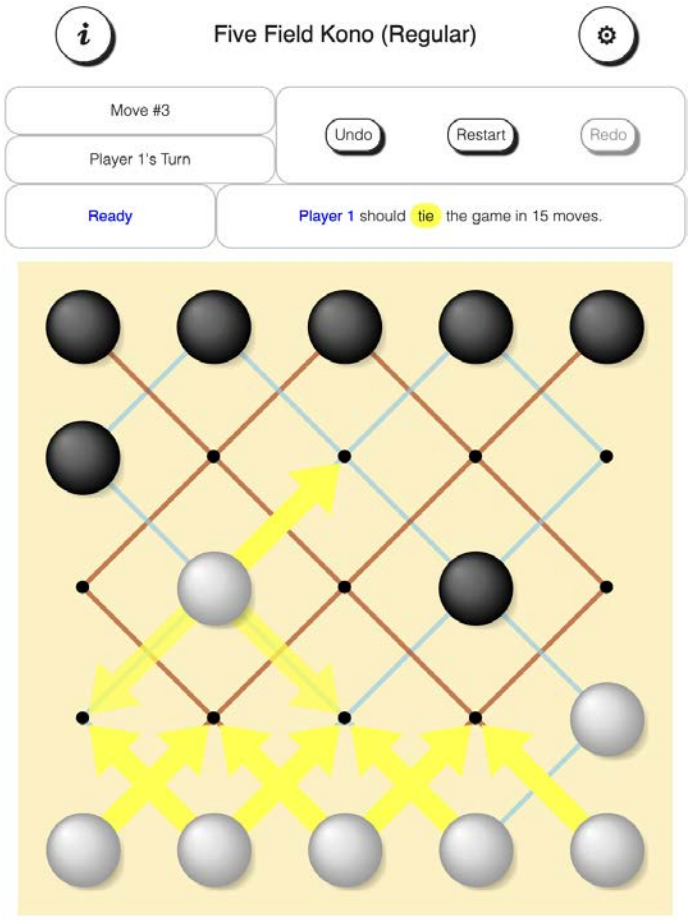

Figure 4.4: GamesmanUni AutoGUI of the Five Field Kono Board [10]

The final board design on GamesmanUni used a beige-colored background, sky blue to denote the even component, and rust to denote the odd component. A Player could interact with the board by clicking the yellow arrow to select where to move a piece. Along with these features, the Player could undo, restart, or automate Player 1 or 2 by the computer.

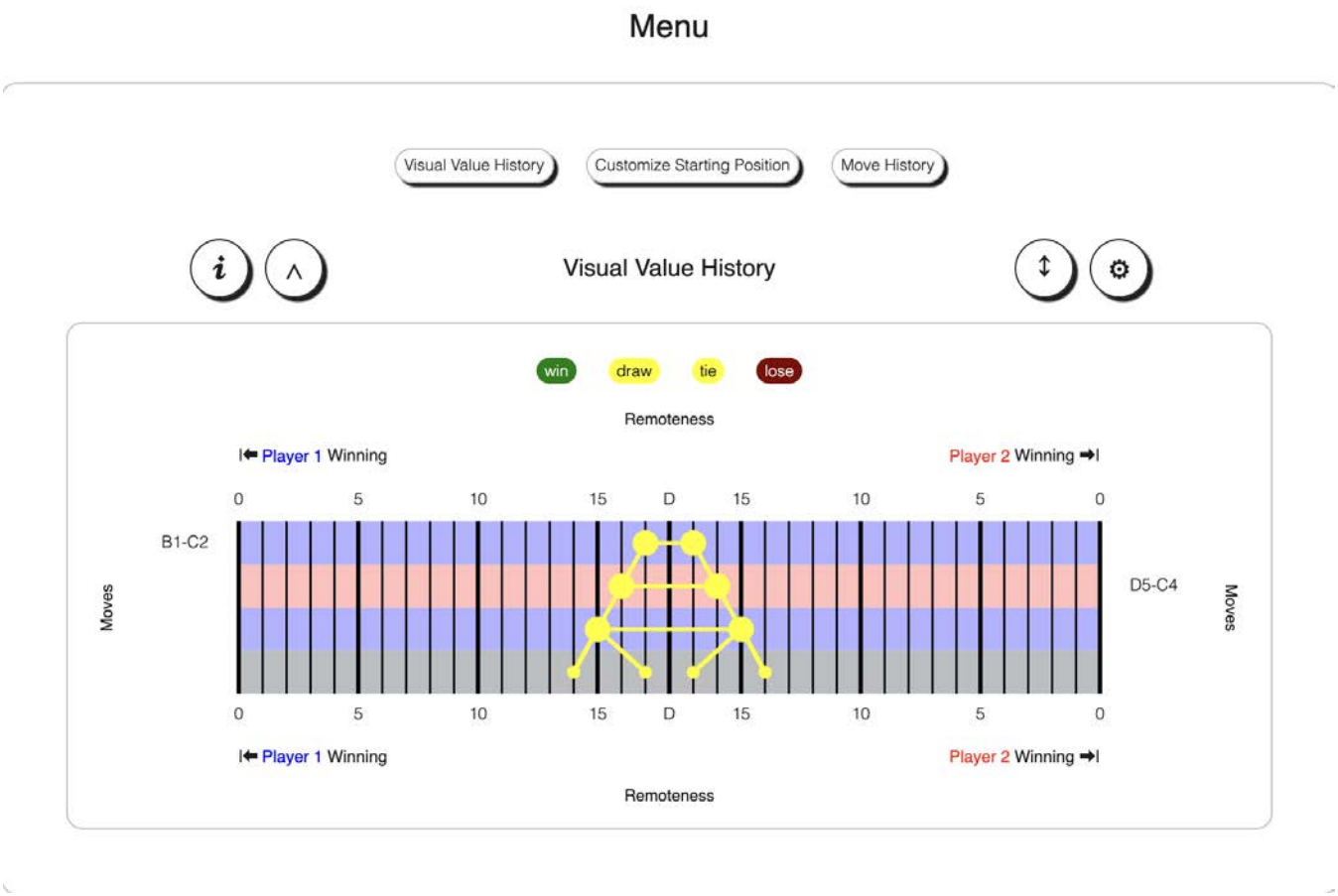

Figure 4.5: GamesmanUni Visual Value History of Five Field Kono [12].

On the right-hand side of the game, there is a Visual Value History diagram to record all of the Players' moves and show the remoteness until a primitive win, loss, or tie occurs. The diagram above demonstrates whether the game will increase or decrease remoteness depending on Player 1 or 2's next moves. As the game leads to a tie in the optimal scenario, it's noticeable how the remoteness graph is entirely colored in yellow.

Chapter 5

Extension

As the Five Field Kono game (with the standard rule that not being able to move is a primitive tie) turns out to be a Tie in 17 (and not very interesting to play), we explored two new variations on the "can't move" rule:

- Variant 1: A Player loses if they cannot make a legal move.
- Variant 2: A Player wins if they cannot make a legal move.

The original goal of getting all pieces to the other side does not change.

Figure 5.1: Five Field Kono Variant Selection in GamesmanUni [11]

5.1 Variant 1 and 2 Results

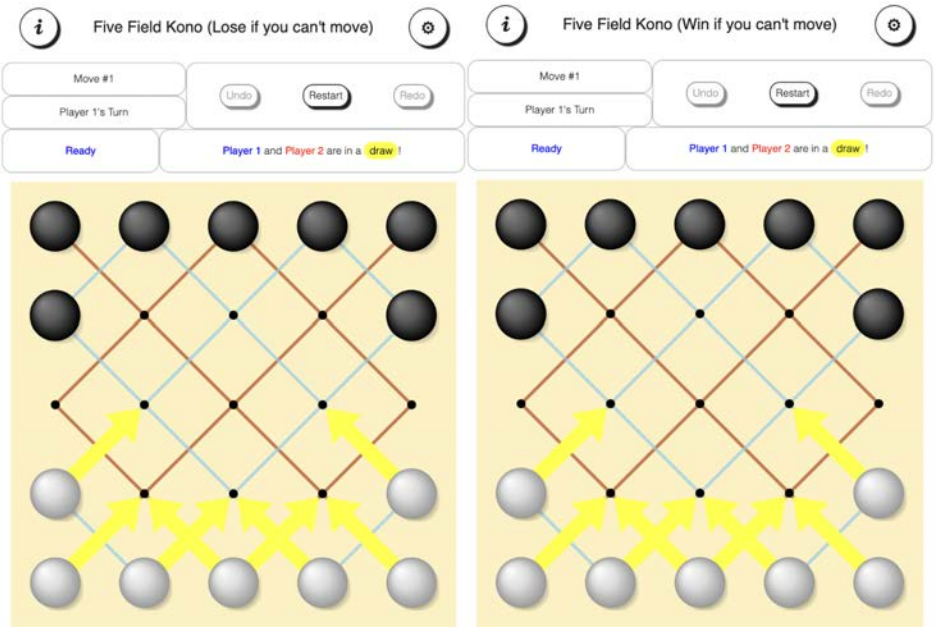

Figure 5.2: Optimal gameplay showing up as a draw for Variants 1 and 2 of Five Field Kono [12][13]

The most interesting result was that the game's value was a draw, evident from the abundance of draw positions in the two value-remoteness count tables below for the two variants and how the game starts as a draw from 5.2. Unless a Player makes a fatal error (we call a "fumble"), the game will go on forever in perfect play. After playing the game for a while, it is evident why this is – either Player (defensively, if they thought they were going to lose) can just leave a single piece in their home row, preventing the other Player from winning. They can also avoid being trapped because there just aren't enough pieces of one Player to trap all of the "defensive" (for the variant where one option to win is to trap the opponent's pieces). We see this game after game – it typically takes being up by many pieces to trap a Player whose pieces can move around forward and backward. It takes four geese to trap one clever Fox in *Fox and Geese* [14]. In *Hare and Hounds* [15], it takes three hounds to trap one hare. In *Bagh-Chal* [16], it takes 20 goats to trap 4 tigers (although granted, the fox can eat the sheep by jumping over them).

Another thing to note was that since primitive ties (ties occurring from a Player not having valid moves) did not exist in Variant 1 and 2 of the game, the Retrograde Loopy Solver did not need to calculate the remoteness value for the ties that existed in the original game; rather, it set most would-be tie positions to draws. As a result, the solve time was much less: from 24406 seconds for the original game to 13536 seconds for Variant 1 and 13428 seconds for Variant 2 (given the machine configurations of section 2.7).

One difference would be the number of winning positions of Variant 1 versus Variant 2. It's observable that the winning and losing counts are about $624865 \div 263244 \approx 2.37$ times bigger in Variant 1 than in Variant 2 (just by comparing the maximum winning position for remoteness). However, the maximum losing counts aren't too different as Variant 1 provides Player 1 with 147550 positions, while Variant 2 provides Player 1 with 101578 positions at the peak. In addition, there are $1145751434 - 1143637726 = 2113708 \approx 2.1$ million more draw positions in Variant 2 than Variant 1, and this implies the fewer number of winning positions in Variant 2 is filled by the draw positions.

Value-Remoteness	Count	Canonical Count
Lose in 0	15,918	2,403
Win in 1	103,760	15,650
Lose in 2	43,800	6,336
Win in 3	185,868	28,385
Lose in 4	97,607	13,798
Win in 5	403,326	56,870
Lose in 6	147,550	20,366
Win in 7	624,865	85,540
Lose in 8	136,555	18,464
Win in 9	585,388	78,641
Lose in 10	90,518	12,437
Win in 11	462,974	60,882
Lose in 12	45,766	6,058
Win in 13	270,024	35,197
Lose in 14	23,885	3,366
Win in 15	127,212	16,970
Lose in 16	9,508	1,329
Win in 17	44,054	5,981
Lose in 18	1,908	297
Win in 19	9,648	1,306
Lose in 20	278	53
Win in 21	954	152
Lose in 22	68	20
Win in 23	162	41
Lose in 24	6	2
Draw	1,143,637,726	149,107,974
Total	1,147,069,328	149,578,518

Table 5.1: Position Counts Per Value-Remoteness for Variant 1

Value-Remoteness	Count	CanonicalCount
Win in 0	8,088	1,218
Lose in 0	7,830	1,185
Win in 1	28,786	5,358
Lose in 2	33,356	4,902
Win in 3	90,078	15,641
Lose in 4	73,197	10,315
Win in 5	196,427	29,987
Lose in 6	101,578	14,188
Win in 7	263,244	38,778
Lose in 8	83,979	11,416
Win in 9	226,373	32,504
Lose in 10	36,641	5,224
Win in 11	107,690	15,778
Lose in 12	8,286	1,286
Win in 13	31,588	4,808
Lose in 14	1,523	274
Win in 15	9,334	1,258
Lose in 16	784	150
Win in 17	5,172	630
Lose in 18	412	87
Win in 19	2,368	366
Lose in 20	188	41
Win in 21	762	132
Lose in 22	68	20
Win in 23	136	45
Lose in 24	6	2
Draw	1,145,751,434	149,382,925
Total	1,147,069,328	149,578,518

Table 5.2: Position Counts Per Value-Remoteness for Variant 2

Chapter 6

Future Work

This section covers some improvements that can be made to the current GamesCrafters infrastructure to make it possible for others in the future to gain additional insights about Five Field Kono and more easily research other loopy games.

6.1 Pure Draw Analysis

Pure Draw Analysis [17] allows us to compare draw positions. Suppose Player 1 makes a mistake by making a losing move on a Draw-Lose position (see Figure 6.1). By making the Draw-Lose move, Player 2 will only receive a Draw-Win position, giving Player 1 a Draw-Lose position if Player 2 makes an optimal move. This pattern repeats until Player 1 is left with only losing moves. Since Variant 1 and 2 of the game result in a constant draw when played optimally, it would be a good idea to run a Pure Draw Solver that gives additional insights into the game. If the game turns out to be a Pure Draw, a Player can now make moves that give the other Player Draw-Lose positions hoping he or she messes up.

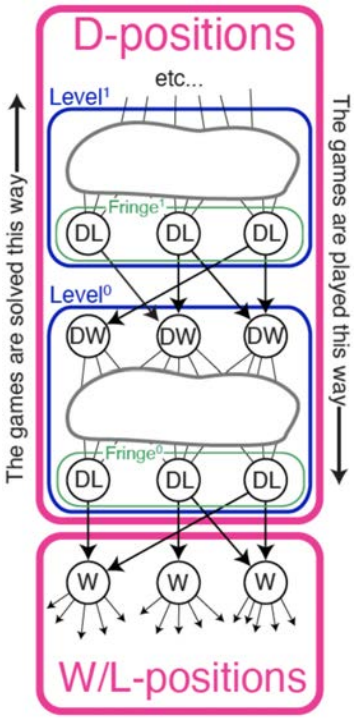

Figure 6.1: Pure Draw diagram of Draw-Lose and Draw-Win moves to lead to a win-or-lose position [10].

Currently, this Pure Draw Solver already exists in the GamesmanClassic system but is programmed memory inefficient with few feature support for loopy games. Future work can focus on utilizing efficient memory structures for this solver and new feature support for loopy games that allow you to undo moves.

6.2 New Variants

Across all new variants, added perfect play leads to a tie or draw for the 3 variants. In the future, new variations on the size of the board and the number of pieces can be implemented. This way, there may be a loophole to winning the game when played optimally, giving users a clear goal and a better experience.

Some variants we have in mind are:

1. **Pebble pieces can be eliminated**: This way, a tie condition would be harder to appear since a blocked path in the game's original version can be unblocked by taking over the opponent's piece when conditions are met.
2. **Bigger board and the same or fewer number of pebbles per Player**: More available paths for each Player to take within the board such that a tying condition is harder to appear.
3. **Once a Player gets to the other side, they win**: This can be a game where Players try their best to reach the other side of the board faster than the other. This can be played using a bigger board, so players strategize which pieces to move first and reduce the chance of a tie.
4. **Players can only move forward** This would require a bigger board with fewer pebbles such that both players frequently have moves available instead of being stuck with no forward paths. The game would most likely favor the Player who starts first to win as they would intuitively reach the other side sooner.
5. **If a single piece is trapped in their back row by opponents' piece(s), it's immediately a lose**: Since the original rules of Five Field Kono make the opponent's or the Player's piece trapped very easily (leading to a tie), it would be a good idea to use this property to make a losing condition, which would make the path to winning or losing easier.

6.3 Additional Documentation

GamesCraftersUWAPI and GamesmanUni didn't have much documentation when I tried to integrate the Backend code into the Frontend. As a result, it would be nice to provide additional documentation to improve productivity instead of scheduling meetings to get a step-by-step walkthrough of the integration process - especially for newcomers.

This can be done effectively by creating a centralized document repository using available open-source technologies or an in-house documentation tool.

Some available technologies are:

1. **Confluence**: It's the most commonly used enterprise documentation platform with numerous integrations for Git workflows and out-of-the-box features. The only downside would be the $7.75 monthly subscription fee.
2. **Notion**: The most popular documentation and scheduler platform for students. With web hosting options and a very customizable interface, it would be easy to use for everyone. The only downside would be how documentation can't be transferred to a PDF format for printing, etc.
3. **GitBook**: A web page-like documentation platform on GitHub. Things can be made like a book with different chapters, and the service is hosted for free on GitHub. The only downside would be how members need to learn how to properly format `README` files along with someone who knows how to maintain the GitBook server.

Chapter 7

Conclusion

This report describes the work to solve Five Field Kono and its variants. Integrations with the Backend and Frontend APIs and how the Main Driver orchestrated the Game Script, Retrograde Loopy Solver, and Two-Byte-Per-Position Database were discussed. Towards the end, Five Field Kono paved the way for how similar cyclic games can be solved. Completing those implementations, we discovered Five Field Kono is not enjoyable when played optimally, as it always results in a tie. The number of tie positions was overabundant as it occupied 99.886% of the total reachable positions, while win and lose positions totaled 0.114%. The 2 variants of the game yielded similar results to the original game, with the only major difference being the solving time. From solving Five Field Kono, some of the bottlenecks existing in the Pure Draw Solver were discovered, new variants for the game were suggested, and new documentation platforms for productivity in GamesCrafters were discussed. With new GameCrafters members joining every semester, I hope to see those changes made to scale the possibilities of games GamesCrafters can solve and a better experience for newcomers and the average user.

Bibliography

[1] Culin S. Korean Games: With Notes on the Corresponding Games of China and Japan. HathiTrust; 1895.

[2] Erica. Davis F, editor. Playing five-field-kono makes your kids smarter. What Do We Do All Day; 2020. Available from: https://www.whatdowedoallday.com/five-field-kono/.

[3] Garcia D. GAMESMAN: A Finite, Two-Person, Perfect-Information Game Generator. GamesCrafters. 1990. Available from: https://people.eecs.berkeley.edu/~ddgarcia/software/gamesman/GAMESMAN.pdf.

[4] Zentner K. Cheung C, editor. Main Driver. GamesCrafters; 2009. Available from: https://github.com/GamesCrafters/GamesmanClassic/blob/master/src/core/main.c.

[5] Zentner K. Shi R, editor. Two-Byte-Per-Position Database. GamesCrafters; 2006. Available from: https://github.com/GamesCrafters/GamesmanClassic/blob/master/src/core/bpdb.c.

[6] Zentner K. Shi R, editor. Retrograde Loopy Solver. GamesCrafters; 2009. Available from: https://github.com/GamesCrafters/GamesmanClassic/blob/master/src/core/solveloopy.c.

[7] Liou A. Garcia D, editor. Web implementation of GamesmanClassic. GamesCrafters; 2000. Available from: https://github.com/GamesCrafters/GamesmanUni.

[8] Katz A. Corn P, editor. Abelian group. Brilliant; 2012. Available from: https://brilliant.org/wiki/abelian-group.

[9] Delgadillo M. Lu S, editor. Generic Hash API. GamesCrafters; 2007. Available from: https://github.com/GamesCrafters/GamesmanClassic/blob/master/doc/files/generic_hash_api.pdf.

[10] Lee A. Cheung C, editor. Five Field Kono Regular. GamesCrafters; 2023. Available from: https://nyc.cs.berkeley.edu/uni/games/fivefieldkono/variants/regular.

[11] Lee A. Cheung C, editor. Five Field Kono Variants Page. GamesCrafters; 2023. Available from: https://nyc.cs.berkeley.edu/uni/games/fivefieldkono/variants.

[12] Lee A. Cheung C, editor. Five Field Kono Variant 1. GamesCrafters; 2023. Available from: https://nyc.cs.berkeley.edu/uni/games/fivefieldkono/variants/delta.

[13] Lee A. Cheung C, editor. Five Field Kono Variant 2. GamesCrafters; 2023. Available from: https://nyc.cs.berkeley.edu/uni/games/fivefieldkono/variants/omega.

[14] Provenzo AB, Eudge F Provenzo J. Favorite Board Games You Can Make and Play. New York: Dover; 1981.

[15] Gardner M. About two new and two old mathematical board games. Scientific American; 1963.

[16] Parlett D. The Oxford History of Board Games. Oxford University Press Inc; 1999.

[17] Garcia D. Garcia D, editor. Draw positions are not all equal. University of California, Berkeley; 2000. Available from: https://people.eecs.berkeley.edu/~ddgarcia/papers/OpenPositionsWLD.html.

Appendix A

Code

We include the code for all Five Field Kono implementation elements mentioned in Section 3 (Analysis, Architecture, and Implementation).

A.1 Five Field Kono Source Code on GamesmanClassic

```
/************************************************************************
**
** NAME:        mfivefieldkono.c
**
** DESCRIPTION: Five-Field Kono
**
** AUTHOR:      Andrew Lee
**
** DATE:        2023-02-24
**
************************************************************************/

#include <stdio.h>
#include <math.h>
#include "gamesman.h"

/* GLOBAL VARIABLES */

/* The person who implemented this game */
STRING kAuthorName = "Andrew Lee";

/* Full name of the game */
STRING kGameName = "Five-Field Kono";

/* Name of the game for databases */
STRING kDBName = "fivefieldkono";

/* How big our POSITION hash can get -- note that this isn't the same
as the upper bound on the number of positions of the game for inefficient
(< 100% efficient) hashing methods */
// 2 * 1189188000 = 2378376000
POSITION gNumberOfPositions = 2378376000;
```

```
/* The hash value of the initial position of the board */
POSITION gInitialPosition = 0;

/* The hash value of any invalid position */
POSITION kBadPosition = -1;

/* There can be different moves available to each player */
BOOLEAN kPartizan = TRUE;

/* It is possible to tie or draw */
BOOLEAN kTieIsPossible = TRUE;

/* The game is loopy */
BOOLEAN kLoopy = TRUE;

/* GetCanonicalPosition will be implemented */
BOOLEAN kSupportsSymmetries = TRUE;

/* TODO: No clue what this does */
BOOLEAN kDebugDetermineValue = FALSE;

/* For initializing the game in Tcl non-generically */
void *gGameSpecificTclInit = NULL;

/* Useful when there are variants available */
BOOLEAN kGameSpecificMenu = TRUE;

/* Enables debug menu for... debugging */
BOOLEAN kDebugMenu = FALSE;

/* Help strings for human players */
STRING kHelpGraphicInterface = "";
STRING kHelpTextInterface = "";
STRING kHelpOnYourTurn = "";
STRING kHelpStandardObjective = "";
STRING kHelpReverseObjective = "";
STRING kHelpTieOccursWhen = "";
STRING kHelpExample = "";

/* 12!/(4!4!4!) = 34650 */
POSITION max_even_hash = 34650;

/* 13!/(3!3!7!) = 34320 */
POSITION max_odd_hash = 34320;

/* optimized factorial lookup: 0 to 25 */
long fact_array[14];

/* BOARD DEFINITION */

/* A Five-Field Kono board consists of 25 spots, but has two connected
components (such that pieces that start out in one cannot possibly move
```

```
into the other). Since clever solving might be possible due to this, we
store them separately. */
// TRUE = your turn (piece x), FALSE = opponent turn (piece o)
typedef struct
{
  char even_component[12];
  char odd_component[13];
  BOOLEAN oppTurn;
} FFK_Board;

/* FUNCTIONAL DECLARATION */

/* Solving functions. */
void InitializeGame();
POSITION GetInitialPosition();
MOVELIST *GenerateMoves(POSITION hash);
POSITION GetCanonicalPosition(POSITION position);
POSITION DoMove(POSITION hash, MOVE move);
VALUE Primitive(POSITION position);

/* Solving helper functions. */
void evaluateEven(int currPos, int newPos, MOVELIST **moves, char *even_component,
  BOOLEAN oppTurn);
void evaluateOdd(int currPos, int newPos, MOVELIST **moves, char *odd_component,
  BOOLEAN oppTurn);
int convertCharToInt(char char_component);

/* Transformation functions. */
void permute(char *target, int *map, int size);
void flipComponent(FFK_Board *board, BOOLEAN evenComponent);
void switchBoard(FFK_Board *board);
char oppositePiece(char board_char);

/* Tier Functions for TierGamesman Support */
TIERLIST *getTierChildren(TIER tier);
TIERPOSITION numberOfTierPositions(TIER tier);
UNDOMOVELIST *GenerateUndoMovesToTier(POSITION position, TIER tier);
POSITION UndoMove(POSITION position, UNDOMOVE undoMove);
POSITION swapTurn(POSITION position);

/* Board hashing functions. */
POSITION Hash(FFK_Board *board);
POSITION compute_hash(char board_component[], int slots, int num_x, int num_o);
FFK_Board *Unhash(POSITION in);
void compute_unhash(char board_component[], POSITION in, int slots, int num_x,
  int num_o);
POSITION rearrangements(int slots, int x, int o);
void precompute_fact(long fact_array[], int limit);
POSITION swapTurn(POSITION hash);
char swapPiece(char board_char);

/* Board value functions. */
BOOLEAN isWin(FFK_Board *board);
```

```
BOOLEAN isLose(FFK_Board *board);
BOOLEAN isTie(FFK_Board *board);

/* Move hashing functions. */
MOVE hashMove(int oldPos, int newPos);
void unhashMove(MOVE mv, int *oldPos, int *newPos);

/* TextUI functions */
void PrintPosition(POSITION position, STRING playerName, BOOLEAN usersTurn);
void PrintComputersMove(MOVE computersMove, STRING computersName);
USERINPUT GetAndPrintPlayersMove(POSITION position, MOVE *move, STRING playerName);
void availableMoves(POSITION position);
BOOLEAN ValidTextInput(STRING input);
MOVE ConvertTextInputToMove(STRING input);
STRING MoveToString(MOVE move);
void PrintMove(MOVE move);

/* Variant Functions */
int NumberOfOptions();
int getOption();
void setOption(int option);

/* INTERACT FUNCTIONS */
POSITION InteractStringToPosition(STRING board);
STRING InteractPositionToString(POSITION position);
STRING InteractPositionToEndData(POSITION position);
STRING InteractMoveToString(POSITION position, MOVE move);

/* UNUSED INTERFACE IMPLEMENTATIONS */
void DebugMenu();
void SetTclCGameSpecificOptions(int theOptions[]);
void GameSpecificMenu();

/* INITIAL BOARD DESCRIPTION */

/* Amount of 'o' pieces initially in each component. */
int initial_even_num_o = 4;
int initial_odd_num_o = 3;

/* Amount of 'x' pieces initially in each component. */
int initial_even_num_x = 4;
int initial_odd_num_x = 3;

/* Amount of pieces in the even component of the baord. */
int even_comp_size = 12;

/* Amount of pieces in the odd component of the baord. */
int odd_comp_size = 13;

/* Describes the initial piece positions in the even part of the board. */
char initial_even_component[12] =
    {'o', 'o', 'o', '-', 'o', '-', '-', 'x', '-', 'x', 'x', 'x'};
```

```
/* Describes the initial piece positions in the even part of the board. */
char initial_odd_component[13] =
    {'o', 'o', 'o', '-', '-', '-', '-', '-', '-', '-', 'x', 'x', 'x'};

char posToAlpha[25] = {'b', 'd', 'a', 'c', 'e', 'b', 'd', 'a', 'c', 'e', 'b', 'd', 'a',
                       'c', 'e', 'b', 'd', 'a', 'c', 'e', 'b', 'd', 'a', 'c', 'e'};
int posToIdx[25] = {5, 5, 4, 4, 4, 3, 3, 2, 2, 2, 1, 1, 5,
                    5, 5, 4, 4, 3, 3, 3, 2, 2, 1, 1, 1};

/* BOARD TRANSFORMATION ARRAYS */

/* Describes a board transformation of a vertical flip (a reflection
across the y-axis, so to say). The piece at original[i] should end up
at destination[array[i]], where 'array' is one of the arrays below. */
int even_xflip_pos[12] = {10, 11, 7, 8, 9, 5, 6, 2, 3, 4, 0, 1};
int odd_xflip_pos[13] = {10, 11, 12, 8, 9, 5, 6, 7, 3, 4, 0, 1, 2};

/* Describes a board transformation of a horizontal flip (a reflection
across the y-axis, so to say). The piece at original[i] should end up
at destination[array[i]], where 'array' is one of the arrays below. */
int even_yflip_pos[12] = {1, 0, 4, 3, 2, 6, 5, 9, 8, 7, 11, 10};
int odd_yflip_pos[13] = {2, 1, 0, 4, 3, 7, 6, 5, 9, 8, 12, 11, 10};

/* BOARD VARIANT */
int variant_type = 0;
VALUE variant_condition[3] = {tie, lose, win};

/* SOLVING FUNCIONS */

/* Initialize any global variables or data structures needed. */
void InitializeGame()
{
  precompute_fact(fact_array, 13);
  gMoveToStringFunPtr = &MoveToString;
  gInitialPosition = GetInitialPosition();
  gCanonicalPosition = GetCanonicalPosition;
  gSymmetries = TRUE;

  /* FOR THE PURPOSES OF INTERACT. FEEL FREE TO CHANGE IF SOLVING. */
  if (gIsInteract)
  {
    gLoadTierdbArray = FALSE; // SET TO TRUE IF SOLVING
  }
  /*******************************/

  /* Tier-Related Initialization */
  gTierChildrenFunPtr = &getTierChildren;
  gNumberOfTierPositionsFunPtr = &numberOfTierPositions;
  gInitialTierPosition = gInitialPosition;
  kSupportsTierGamesman = TRUE;
  kExclusivelyTierGamesman = TRUE;
  gInitialTier = 0; // There will only be one tier and its ID will be 0
  gUnDoMoveFunPtr = &UndoMove;
```

```
  gGenerateUndoMovesToTierFunPtr = &GenerateUndoMovesToTier;
}

/* Return the hash value of the initial position. */
POSITION GetInitialPosition()
{
  FFK_Board *initial_board = malloc(sizeof(FFK_Board));
  for (int i = 0; i < even_comp_size; i++)
    initial_board->even_component[i] = initial_even_component[i];
  for (int i = 0; i < odd_comp_size; i++)
    initial_board->odd_component[i] = initial_odd_component[i];
  initial_board->oppTurn = FALSE;
  POSITION result = Hash(initial_board);
  free(initial_board);
  return result;
}

/* Return a linked list of possible moves. */
MOVELIST *GenerateMoves(POSITION hash)
{
  FFK_Board *newboard = Unhash(hash);
  MOVELIST *moves = NULL;
  for (int i = 0; i < even_comp_size; i++)
  {
    if (i != 4 && i != 9)
    {
      evaluateEven(i, i - 2, &moves, newboard->even_component, newboard->oppTurn);
      evaluateEven(i, i + 3, &moves, newboard->even_component, newboard->oppTurn);
    }
    if (i != 2 && i != 7)
    {
      evaluateEven(i, i + 2, &moves, newboard->even_component, newboard->oppTurn);
      evaluateEven(i, i - 3, &moves, newboard->even_component, newboard->oppTurn);
    }
  }
  for (int j = 0; j < odd_comp_size; j++)
  {
    if (j != 0 && j != 5 && j != 10)
    {
      evaluateOdd(j, j + 2, &moves, newboard->odd_component, newboard->oppTurn);
      evaluateOdd(j, j - 3, &moves, newboard->odd_component, newboard->oppTurn);
    }
    if (j != 2 && j != 7 && j != 12)
    {
      evaluateOdd(j, j - 2, &moves, newboard->odd_component, newboard->oppTurn);
      evaluateOdd(j, j + 3, &moves, newboard->odd_component, newboard->oppTurn);
    }
  }
  free(newboard);
  return moves;
}

/* Return the resulting position from making 'move' on 'position'. */
```

```
POSITION DoMove(POSITION hash, MOVE move)
{
  // Get current board
  FFK_Board *board = Unhash(hash);

  // Get move information (from, to = indices in board[25])
  int from, to;
  unhashMove(move, &from, &to);

  // Change the oppTurn --> !oppTurn to reflect change in turn
  board->oppTurn = !(board->oppTurn);

  if (from < 12)
  {
    // Piece to be moved is in board->even_component[12] which means
    // that 'to' is as well; mutate board accordingly
    char piece = board->even_component[from];
    board->even_component[from] = '-';
    board->even_component[to] = piece;
  }
  else
  {
    // Piece to be moved is in board->odd_component[13] which means
    // that 'to' is as well
    from = from - 12;
    to = to - 12;

    // Mutate board accordingly
    char piece = board->odd_component[from];
    board->odd_component[from] = '-';
    board->odd_component[to] = piece;
  }

  // Compute hash for post-move
  POSITION result = Hash(board);
  free(board);
  return result;
}

/* Symmetry Handling: Return the canonical position. For reference:
"A" - refers to the even component of the board.
"B" - refers to the odd component of the board.
"X_e" - refers to the X component of the board with no
        transformations applied.
"X_f" - refers to the X component of the board with a
        y-axis flip applied (defined by flipComponent()).
"(STATE)_s" - refers to the STATE of the board with a
              switch (defined by switchComponent()) applied.
So for example, the board in its base state is "A_e B_e," and doing
switchComponent() on this is "(A_e B_e)_s." Need I say more? */
POSITION GetCanonicalPosition(POSITION position)
{
  FFK_Board *board = Unhash(position);
```

```
  POSITION symmetries[8];
  POSITION canonical = gNumberOfPositions;

  // 'Flip only' symmetries
  symmetries[0] = position; // A_e B_e
  flipComponent(board, TRUE);
  symmetries[1] = Hash(board); // A_f B_e
  flipComponent(board, FALSE);
  symmetries[2] = Hash(board); // A_f B_f
  flipComponent(board, TRUE);
  symmetries[3] = Hash(board); // A_e B_f

  // 'Board switched' symmetries
  switchBoard(board);
  symmetries[4] = Hash(board); // (A_e B_f)_s
  flipComponent(board, TRUE);
  symmetries[5] = Hash(board); // (A_f B_f)_s
  flipComponent(board, FALSE);
  symmetries[6] = Hash(board); // (A_f B_e)_s
  flipComponent(board, TRUE);
  symmetries[7] = Hash(board); // (A_e B_e)_s

  // Choose the smallest hash as canonical
  for (int i = 0; i < 8; i++)
  {
    if (symmetries[i] < canonical)
    {
      canonical = symmetries[i];
    }
  }

  free(board);
  return canonical;
}

/* Return lose, win, tie, or undecided. See src/core/types.h
for the value enum definition. */
VALUE Primitive(POSITION position)
{
  FFK_Board *board = Unhash(position);
  BOOLEAN is_win = isWin(board);
  BOOLEAN is_lose = isLose(board);
  BOOLEAN is_tie = isTie(board);
  free(board);
  if (is_win || is_lose)
  {
    return lose;
  }
  if (is_tie)
  {
    return variant_condition[variant_type];
  }
  return undecided;
```

```
}

/* The tier graph is just a single tier with id=0. */
TIERLIST *getTierChildren(TIER tier)
{
  return CreateTierlistNode(0, NULL);
}

/* We use a single tier for this entire game. This
is returns the upper bound */
TIERPOSITION numberOfTierPositions(TIER tier)
{
  return gNumberOfPositions;
}

/* Return a linked list of all possible moves that could have been made in
order to arrive at the input position. The movement rules of FFK are
nice in that we can reuse GenerateMoves on the position with the turn
swapped. The result of GenerateMoes can be our undoMoves, except that we
need to filter out moves that come from primitive positions.
There is only one tier, so all undoMoves will lead to previous positions
that are in the same tier, so tier is ignored. */
UNDOMOVELIST *GenerateUndoMovesToTier(POSITION position, TIER tier)
{
  MOVELIST *moves = GenerateMoves(swapTurn(position));
  MOVELIST *head = moves;
  UNDOMOVELIST *undoMoves = NULL;
  while (moves != NULL)
  {
    if (Primitive(UndoMove(position, moves->move)) == undecided)
    {
      undoMoves = CreateUndoMovelistNode(moves->move, undoMoves);
    }
    moves = moves->next;
  }
  FreeMoveList(head);
  return undoMoves;
}

/* Return the parent position given the undoMove. */
POSITION UndoMove(POSITION position, UNDOMOVE undoMove)
{
  return swapTurn(DoMove(swapTurn(position), undoMove));
}

/* SOLVING HELPER FUNCTIONS */

void evaluateEven(int currPos, int newPos, MOVELIST **moves, char *even_component,
  BOOLEAN oppTurn)
{
  if (newPos < 0 || newPos >= even_comp_size)
  {
    return;
```

```
  }
  int currElem = convertCharToInt(even_component[currPos]);
  int newElem = convertCharToInt(even_component[newPos]);
  int match = oppTurn ? 1 : 2;
  if (currElem == match && newElem == 0)
  {
    *moves = CreateMovelistNode(hashMove(currPos, newPos), *moves);
  }
}

void evaluateOdd(int currPos, int newPos, MOVELIST **moves, char *odd_component,
  BOOLEAN oppTurn)
{
  if (newPos < 0 || newPos >= odd_comp_size)
  {
    return;
  }
  int currElem = convertCharToInt(odd_component[currPos]);
  int newElem = convertCharToInt(odd_component[newPos]);
  int match = oppTurn ? 1 : 2;
  if (currElem == match && newElem == 0)
  {
    *moves = CreateMovelistNode(hashMove(12 + currPos, 12 + newPos), *moves);
  }
}

/* Converts a character to its ternary integer equivalent
for hash calculations. */
int convertCharToInt(char char_component)
{
  if (char_component == '-')
  {
    return 0;
  }
  else if (char_component == 'o')
  {
    return 1;
  }
  else if (char_component == 'x')
  {
    return 2;
  }
  return -1;
}

/* TRANSFORMATION FUNCTIONS */

/* Performs an in-place rearrangement of the contents of TARGET as outlined
by a transformation array MAP, assuming they are both the same SIZE. Does not
allocate memory and is linear in the size of the array being permuted. */
void permute(char *target, int *map, int size)
{
  char temp[size];
```

```
  for (int i = 0; i < size; i++)
  {
    temp[i] = target[map[i]];
  }
  memcpy(target, temp, size * sizeof(char));
}

/* Transforms the board by flipping it horizontally (a reflection
about the y-axis). */
void flipComponent(FFK_Board *board, BOOLEAN evenComponent)
{
  if (evenComponent)
  {
    permute(board->even_component, even_yflip_pos, even_comp_size);
  }
  else
  {
    permute(board->odd_component, odd_yflip_pos, odd_comp_size);
  }
}

/* Switches whose turn it is, replaces each piece with one of the opposite
color, and flips the board across the x-axis. */
void switchBoard(FFK_Board *board)
{
  // Flip across the x-axis
  permute(board->even_component, even_xflip_pos, even_comp_size);
  permute(board->odd_component, odd_xflip_pos, odd_comp_size);

  // Replace pieces with opposites
  for (int i = 0; i < even_comp_size; i++)
    board->even_component[i] = oppositePiece(board->even_component[i]);
  for (int j = 0; j < odd_comp_size; j++)
    board->odd_component[j] = oppositePiece(board->odd_component[j]);

  // Switch whose turn it is
  board->oppTurn = !(board->oppTurn);
}

/* Replaces a board piece to the opponent's piece type relative to whose
piece it is. */
char oppositePiece(char board_char)
{
  if (board_char == 'o')
    return 'x';
  else if (board_char == 'x')
    return 'o';
  else
    return '-';
}

/* BOARD HASHING FUNCTIONS */
```

```
/* Returns the index that IN would have in the alphabetical ordering of all
possible strings composed of the same kinds and amounts of characters. */
POSITION Hash(FFK_Board *board)
{
  // Collect the number of slots <-- 12 + 13 = 25 slots
  POSITION even_hash = compute_hash(board->even_component, even_comp_size,
    initial_even_num_x, initial_even_num_o);
  POSITION odd_hash = compute_hash(board->odd_component, odd_comp_size,
    initial_odd_num_x, initial_odd_num_o);
  int turn = board->oppTurn ? 1 : 0;

  // odd_hash * (2*max_even_hash) + (2**turn)*even_hash <= 2.4 billion positions
  return odd_hash * (2 * max_even_hash) + ((turn * max_even_hash) + even_hash);
}

POSITION compute_hash(char board_component[], int slots, int num_x, int num_o)
{
  POSITION total = 0;
  int arr_len = slots;
  for (int i = 0; i < arr_len; i++)
  {
    int t1 = rearrangements(slots - 1, num_x, num_o);
    int t2 = t1 + rearrangements(slots - 1, num_x, num_o - 1);
    if (board_component[i] == 'o')
    {
      total += t1;
      num_o -= 1;
    }
    else if (board_component[i] == 'x')
    {
      total += t2;
      num_x -= 1;
    }
    slots -= 1;
  }
  return total;
}

/* The inverse process of hash. */
FFK_Board *Unhash(POSITION in)
{
  FFK_Board *newBoard = (FFK_Board *)malloc(sizeof(FFK_Board));
  POSITION even_hash = in % (2 * max_even_hash);
  POSITION odd_hash = in / (2 * max_even_hash);
  if (even_hash >= max_even_hash)
    newBoard->oppTurn = TRUE;
  else
    newBoard->oppTurn = FALSE;
  if (even_hash >= max_even_hash)
    even_hash -= max_even_hash;
  compute_unhash(newBoard->even_component, even_hash, even_comp_size,
    initial_even_num_x, initial_even_num_o);
  compute_unhash(newBoard->odd_component, odd_hash, odd_comp_size,
```

```
    initial_odd_num_x, initial_odd_num_o);

  return newBoard;
}

void compute_unhash(char board_component[], POSITION in, int slots, int num_x,
  int num_o)
{
  int idx = 0;
  while (slots > 0)
  {
    int t1 = rearrangements(slots - 1, num_x, num_o);
    int t2 = t1 + rearrangements(slots - 1, num_x, num_o - 1);
    if (in < t1)
    {
      board_component[idx] = '-';
    }
    else if (in < t2)
    {
      board_component[idx] = 'o';
      in -= t1;
      num_o -= 1;
    }
    else
    {
      board_component[idx] = 'x';
      in -= t2;
      num_x -= 1;
    }
    slots -= 1;
    idx += 1;
  }
}

/* Returns the amount of ways to put X x's and O o's into SLOTS slots. */
POSITION rearrangements(int slots, int x, int o)
{
  if ((slots < 0) || (x < 0) || (o < 0) || (slots < (o + x)))
    return 0;
  // Essentially returns the number of ways to place the -'s, x's, and o's in the slots
  return fact_array[slots] / (fact_array[x] * fact_array[o] * fact_array[slots-x-o]);
}

void precompute_fact(long fact_array[], int limit)
{
  fact_array[0] = 1;
  for (long i = 1; i <= limit; i++)
  {
    fact_array[i] = i * fact_array[i - 1];
  }
}

/* Swap the turn and return the new hash */
```

```
POSITION swapTurn(POSITION hash)
{
  BOOLEAN turn;
  POSITION even_hash = hash % (2 * max_even_hash);
  POSITION odd_hash = hash / (2 * max_even_hash);
  // if even_hash >= max_even_hash it was the opponent's turn
  // hence, set the turn to FALSE to swap it to your turn
  // and vice versa for
  if (even_hash >= max_even_hash)
  {
    turn = FALSE;
    even_hash -= max_even_hash;
  }
  else
    turn = TRUE;
  return odd_hash * (2 * max_even_hash) + ((turn * max_even_hash) + even_hash);
}

/* BOARD VALUE FUNCTIONS */

/* X wins when all of the spots originally populated by O's are
filled with X's. */
BOOLEAN isWin(FFK_Board *board)
{
  return board->even_component[0] == 'x' && board->even_component[1] == 'x'
    && board->even_component[2] == 'x' && board->even_component[4] == 'x'
    && board->odd_component[0] == 'x' && board->odd_component[1] == 'x'
    && board->odd_component[2] == 'x';
}

/* O wins when all of the spots originally populated by X's are
filled with O's. */
BOOLEAN isLose(FFK_Board *board)
{
  return board->even_component[11] == 'o' && board->even_component[10] == 'o'
    && board->even_component[9] == 'o' && board->even_component[7] == 'o'
    && board->odd_component[12] == 'o' && board->odd_component[11] == 'o'
    && board->odd_component[10] == 'o';
}

/* If there are no moves left to be made, then the game is a tie. */
BOOLEAN isTie(FFK_Board *board)
{
  MOVELIST *possible_moves = GenerateMoves(Hash(board));
  if (possible_moves == NULL)
  {
    FreeMoveList(possible_moves);
    return TRUE;
  }
  FreeMoveList(possible_moves);
  return FALSE;
}
```

```c
/* MOVE HASHING FUNCTIONS */

/* Uses the start and destination index of a piece in the connected
component of the board it belongs to and whose turn it is to generate
a unique hash for a game state graph edge. */
MOVE hashMove(int oldPos, int newPos)
{
  // Ob<base 25 newPos, base 25 oldPos>
  return (25 * newPos) + oldPos;
}

/* Obtains the start and destination index of a piece in the connected
component of the board it belongs to and whose turn it is based on
a unique hash for a game state graph edge. */
void unhashMove(MOVE mv, int *oldPos, int *newPos)
{
  // Ob<base 25 newPos, base 25 oldPos>
  *oldPos = mv % 25;
  *newPos = mv / 25;
}

/* TEXTUI FUNCTIONS */

/* This will print the board in the following format:

  5   (o) (o) (o) (o) (o)
        \ / \ / \ / \ /
        / \ / \ / \ / \
  4   (o) ( ) ( ) ( ) (o)
        \ / \ / \ / \ /
        / \ / \ / \ / \
  3   ( ) ( ) ( ) ( ) ( )
        \ / \ / \ / \ /
        / \ / \ / \ / \
  2   (x) ( ) ( ) ( ) (x)
        \ / \ / \ / \ /
        / \ / \ / \ / \
  1   (x) (x) (x) (x) (x)

       A   B   C   D   E

*/
void PrintPosition(POSITION position, STRING playerName, BOOLEAN usersTurn) {
  FFK_Board* board = Unhash(position);
  char* fb = malloc(sizeof(char)*25);
  printf("\n\n");
  for (int i = 0; i < 25; i++) {
    if (i % 2 == 0) {
      char curr = board->odd_component[i/2];
      fb[i] = (curr == '-') ? ' ' : curr;
    } else {
      char curr = board->even_component[(i-1)/2];
      fb[i] = (curr == '-') ? ' ' : curr;
```

```
    }
  }
  for (int j = 0; j < 4; j++) {
    if (j == 0) printf("5");
    if (j == 1) printf("4");
    if (j == 2) printf("3");
    if (j == 3) printf("2");
    printf("   (%c) (%c) (%c) (%c) (%c) \n", fb[5*j], fb[(5*j)+1], fb[(5*j)+2],
↪   fb[(5*j)+3], fb[(5*j)+4]);
    printf("      \\ / \\ / \\ / \\ /   ");
    if (j == 2) {
      printf("         %s\n", GetPrediction(position, playerName, usersTurn));
    } else {
      printf("\n");
    }
    printf("      / \\ / \\ / \\ / \\   ");
    if (j == 1) {
      printf("         TURN: %c\n", (board->oppTurn) ? 'o' : 'x');
    } else {
      printf("\n");
    }
  }
  printf("1   (%c) (%c) (%c) (%c) (%c) \n\n", fb[20], fb[21], fb[22], fb[23], fb[24]);
  printf("     A   B   C   D   E \n\n");
  free(fb);
  free(board);
}

void PrintComputersMove(MOVE computersMove, STRING computersName)
{
  printf("%8s's move                                  : ", computersName);
  PrintMove(computersMove);
  printf("\n");
}

USERINPUT GetAndPrintPlayersMove(POSITION position, MOVE *move, STRING playerName)
{
  USERINPUT ret;

  do
  {
    /* List of available moves */
    // availableMoves(position);
    printf("%8s's move [(undo)/([a-e][1-5][a-e][1-5])]:  ", playerName);

    ret = HandleDefaultTextInput(position, move, playerName);
    if (ret != Continue)
      return (ret);

  } while (TRUE);
  return (Continue); /* this is never reached, but lint is now happy */
}
```

```
void availableMoves(POSITION position)
{
  MOVELIST *available_moves = GenerateMoves(position);
  MOVELIST *ptr = available_moves;
  printf("  %8s: \n", "Available Hash Moves {(Current Position)-(Next Position)}");
  while (available_moves != NULL)
  {
    MOVE move_val = available_moves->move;
    int from, to;
    unhashMove(move_val, &from, &to);
    char fromToAlpha = posToAlpha[from];
    char toToAlpha = posToAlpha[to];
    int fromToIdx = posToIdx[from];
    int toToIdx = posToIdx[to];
    printf("  {%c%d-%c%d} ", fromToAlpha, fromToIdx, toToAlpha, toToIdx);
    available_moves = available_moves->next;
  }
  FreeMoveList(ptr);
  printf("\n");
}

/* Return whether the input text signifies a valid move. Rows are letters, and
columns are numbers:
- A piece in rows {a, c, e} can only go to one of {b, d} and vice versa.
- A piece in columns {1, 3, 5} can only go to one of {2, 4} and vice versa.
- Both the rows and columns must differ in 'distance' by exactly 1.
Example valid moves: {"a1b2", "b2c3", "e4d5"}. */
BOOLEAN ValidTextInput(STRING input)
{
  // Check for obvious malformations
  if (strlen(input) == 4)
  {
    // Extract characters from string
    char c1 = (char)tolower(input[0]);
    int r1 = atoi(&input[1]);
    char c2 = (char)tolower(input[2]);
    int r2 = atoi(&input[3]);

    // Determine if both slots are on the board using ASCII ranges
    if (c1 < 'a' || c1 > 'e' || c2 < 'a' || c2 > 'e')
      return FALSE;
    if (r1 < 1 || r1 > 5 || r2 < 1 || r2 > 5)
      return FALSE;

    // Use ASCII values to determine 'distance', which guarantees that
    // the piece moves along a valid edge and that it moves a distance
    // of exactly 1
    if (abs(r1 - r2) != 1 || abs(c1 - c2) != 1)
      return FALSE;

    return TRUE;
  }
  else if (strlen(input) <= 3)
```

```
  {
    int inputToHash = atoi(input);
    if (inputToHash >= 650 || inputToHash <= 0)
      return FALSE;
    else
    {
      int from, to;
      unhashMove(inputToHash, &from, &to);
      if (abs(from - to) == 3 || abs(from - to) == 2)
        return TRUE;
      else
        return FALSE;
    }
  }
  else
  {
    return FALSE;
  }
}

/* Assume the text input signifies a valid move. Return
the move hash corresponding to the move. */
MOVE ConvertTextInputToMove(STRING input)
{
  if (strlen(input) <= 3)
  {
    return atoi(input);
  }
  else
  {
    int from = -1;
    int to = -1;
    char c1 = (char)tolower(input[0]);
    int r1 = atoi(&input[1]);
    char c2 = (char)tolower(input[2]);
    int r2 = atoi(&input[3]);
    for (int i = 0; i < 25; i++)
    {
      char currAlpha = (char)tolower(posToAlpha[i]);
      int currIdx = (int)posToIdx[i];
      if (currAlpha == c1 && currIdx == r1)
        from = i;
      if (currAlpha == c2 && currIdx == r2)
        to = i;
    }
    return hashMove(from, to);
  }
}

/* Return the string representation of the move.
Ideally this matches with what the user is supposed to
type in. */
STRING MoveToString(MOVE move)
```

```
{
  int from, to;
  unhashMove(move, &from, &to);
  char fromToAlpha = posToAlpha[from];
  char toToAlpha = posToAlpha[to];
  char fromToIdx = (char)('0' + posToIdx[from]);
  char toToIdx = (char)('0' + posToIdx[to]);
  STRING output = (STRING)malloc(6 * sizeof(char));
  output[0] = fromToAlpha;
  output[1] = fromToIdx;
  output[2] = toToAlpha;
  output[3] = toToIdx;
  output[4] = '\0';
  return output;
}

/* Basically just print the move. */
void PrintMove(MOVE move)
{
  int from, to;
  unhashMove(move, &from, &to);
  char fromToAlpha = posToAlpha[from];
  char toToAlpha = posToAlpha[to];
  int fromToIdx = posToIdx[from];
  int toToIdx = posToIdx[to];
  printf("%c%d%c%d", fromToAlpha, fromToIdx, toToAlpha, toToIdx);
}

/* VARIANT FUNCTIONS */

/* Amount of variants supported. */
int NumberOfOptions()
{
  return 3;
}

/* Return the current variant ID (0 in this case). */
int getOption()
{
  return variant_type;
}

/* The input is a variant id. This function sets any global variables
or data structures according to the variant specified by the variant id.
But for now you have one variant so don't worry about this. */
void setOption(int option)
{
  if (option >= 0 && option <= 2)
    variant_type = option;
  else
    printf(" Sorry I don't know that option\n");
}
```

```
/* INTERACT FUNCTIONS */

POSITION InteractStringToPosition(STRING board)
{
  enum UWAPI_Turn turn;
  unsigned int num_rows, num_columns;
  STRING charBoard;
  if (!UWAPI_Board_Regular2D_ParsePositionString(board, &turn, &num_rows, &num_columns,
↪  &charBoard))
  {
    // Failed to parse string
    return INVALID_POSITION;
  }
  FFK_Board real_board;
  for (int i = 0; i < 25; i++)
  {
    if (i % 2 == 0)
      real_board.odd_component[i / 2] = charBoard[i];
    else
      real_board.even_component[(i - 1) / 2] = charBoard[i];
  }
  real_board.oppTurn = (turn == UWAPI_TURN_A) ? 0 : 1;
  free(charBoard);
  return Hash(&real_board);
}

STRING InteractPositionToString(POSITION position)
{
  // UWAPI_Board_Regular2D_MakeBoardString(turn, 25, charBoard);
  // enum UWAPI_Turn turn = (whosTurn == V) ? UWAPI_TURN_A : UWAPI_TURN_B;
  FFK_Board *board = Unhash(position);
  enum UWAPI_Turn turn = (board->oppTurn == 0) ? UWAPI_TURN_A : UWAPI_TURN_B;
  char charBoard[26];
  for (int i = 0; i < 25; i++)
  {
    if (i % 2 == 0)
      charBoard[i] = board->odd_component[i / 2];
    else
      charBoard[i] = board->even_component[(i - 1) / 2];
  }
  charBoard[25] = '\0';
  free(board);
  return UWAPI_Board_Regular2D_MakeBoardString(turn, 25, charBoard);
  ;
}

STRING InteractPositionToEndData(POSITION position) { return NULL; }

STRING InteractMoveToString(POSITION position, MOVE move)
{
  // return UWAPI_Board_Regular2D_MakeMoveString(fromSlot, toSlot);
  int from, to;
  unhashMove(move, &from, &to);
```

```
  if (from >= 12)
  {
    // Odd Component
    from = 2 * (from - 12);
    to = 2 * (to - 12);
  }
  else
  {
    // Even Component
    from = (2 * from) + 1;
    to = (2 * to) + 1;
  }
  return UWAPI_Board_Regular2D_MakeMoveString(from, to);
}

/* UNUSED INTERFACE IMPLEMENTATIONS */

/* Debug menu for... debugging */
void DebugMenu() {}

/* Should be configured when Tcl is initialized non-generically */
void SetTclCGameSpecificOptions(int theOptions[]) {}

/* For implementing more than one variant */
void GameSpecificMenu()
{
  char inp;
  while (TRUE)
  {
    printf("\n\n\n");
    printf("        ----- Game-specific options for Quick Cross -----\n\n");
    printf("        Select a game board:\n\n");
    printf("        1)      Default 5 x 5 Board with Stalemate Being a Tie \n");
    printf("        2)      Default 5 x 5 Board with Stalemate Being a Win \n");
    printf("        3)      Default 5 x 5 Board with Stalemate Being a Loss \n");
    printf("        4)      (B)ack = Return to previous activity.\n\n\n");
    printf("\nSelect an option: ");
    inp = GetMyChar();
    if (inp == '1')
    {
      setOption(0);
      return;
    }
    else if (inp == '2')
    {
      setOption(1);
      return;
    }
    else if (inp == '3')
    {
      setOption(2);
      return;
    }
```

```
    else if (inp == '4' || inp == 'b' || inp == 'B')
      return;
    else
    {
      printf("Invalid input.\n");
    }
  }
}
```

A.2 GamesCraftersUWAPI Integration

We include any code written for GamesCraftersUWAPI to connect to the GamesmanClassic backend and relay AutoGUI data for Five Field Kono to any frontend applications.

A.2.1 Five Field Kono Variants on GamesCraftersUWAPI

```
games = {
    'fivefieldkono': Game(
        name='Five Field Kono',
        desc="Largest game in GamesmanUni that can't be separated into tiers.",
        variants={
            'regular': GameVariant(
                name="Tie if you can't move",
                desc='Regular',
                data_provider=GamesmanClassicDataProvider,
                data_provider_game_id='fivefieldkono',
                data_provider_variant_id=0,
                status='available',
                gui_status='v2'),
            'delta': GameVariant(
                name="Lose if you can't move",
                desc='Delta',
                data_provider=GamesmanClassicDataProvider,
                data_provider_game_id='fivefieldkono',
                data_provider_variant_id=1,
                status='available',
                gui_status='v2'),
            'omega': GameVariant(
                name="Win if you can't move",
                desc='Omega',
                data_provider=GamesmanClassicDataProvider,
                data_provider_game_id='fivefieldkono',
                data_provider_variant_id=2,
                status='available',
                gui_status='v2')
            },
        gui_status='v2'),
}
```

A.2.2 Piece SVG and Coordinate AutoGUI JSON Data

```
def get_fivefieldkono(variant_id):
    if variant_id not in ["regular", "delta", "omega"]:
        return None
    return {
        variant_id: {
            "defaultTheme": variant_id,
            "themes": {
                variant_id: {
                    "backgroundGeometry": [
                        200, 200
                    ],
                    "backgroundImage": "fivefieldkono/board.svg",
                    "piecesOverArrows": True,
                    "arrowThickness": 5,
                    "centers": [[20 + 40 * i, 20 + 40 * j]
                         for j in range(0,5) for i in range(0,5)],
                    "pieces": {
                        "x": {
                            # White pieces for X
                            "image": "369mm/O.svg", "scale": 25.0
                        }, "o": {
                            # Black pieces for O
                            "image": "369mm/X.svg", "scale": 25.0
                        }
                    }
                }
            }
        }
    }.get(variant_id, None)

autoGUIv2DataFuncs = {
    "fivefieldkono": get_fivefieldkono
}

def get_autoguiV2Data(game_id, variant_id):
    if game_id in autoGUIv2DataFuncs:
        return autoGUIv2DataFuncs[game_id](variant_id)
    return None
```

Appendix B

Example TextUI Gameplay

We show an example of TextUI gameplay with the "Cannot Move = Lose" variant. The user specifies a move by entering a source and destination (e.g., `b1c2`). The user can also enter `ps` for a complete list of legal moves from the current position and their values.

```
5   (o) (o) (o) (o) (o)
      \ / \ / \ / \ /
      / \ / \ / \ / \
4   (o) ( ) ( ) ( ) (o)
      \ / \ / \ / \ /
      / \ / \ / \ / \           TURN: x
3   ( ) ( ) ( ) ( ) ( )
      \ / \ / \ / \ /           (Player should draw)
      / \ / \ / \ / \
2   (x) ( ) ( ) ( ) (x)
      \ / \ / \ / \ /
      / \ / \ / \ / \
1   (x) (x) (x) (x) (x)

     A   B   C   D   E

  Player's move [(undo)/([a-e][1-5][a-e][1-5])]: b1c2


5   (o) (o) (o) (o) (o)
      \ / \ / \ / \ /
      / \ / \ / \ / \
4   (o) ( ) ( ) ( ) (o)
      \ / \ / \ / \ /
      / \ / \ / \ / \           TURN: o
3   ( ) ( ) ( ) ( ) ( )
      \ / \ / \ / \ /           (Data should draw)
      / \ / \ / \ / \
2   (x) ( ) (x) ( ) (x)
      \ / \ / \ / \ /
      / \ / \ / \ / \
1   (x) ( ) (x) (x) (x)

     A   B   C   D   E
```

```
    Data's move                                 : a4b3


5   (o) (o) (o) (o) (o)
      \ / \ / \ / \ /
      / \ / \ / \ / \
4   ( ) ( ) ( ) ( ) (o)
      \ / \ / \ / \ /
      / \ / \ / \ / \           TURN: x
3   ( ) (o) ( ) ( ) ( )
      \ / \ / \ / \ /           (Player should draw)
      / \ / \ / \ / \
2   (x) ( ) (x) ( ) (x)
      \ / \ / \ / \ /
      / \ / \ / \ / \
1   (x) ( ) (x) (x) (x)

     A   B   C   D   E

  Player's move [(undo)/([a-e][1-5][a-e][1-5])]: e2d3


5   (o) (o) (o) (o) (o)
      \ / \ / \ / \ /
      / \ / \ / \ / \
4   ( ) ( ) ( ) ( ) (o)
      \ / \ / \ / \ /
      / \ / \ / \ / \           TURN: o
3   ( ) (o) ( ) (x) ( )
      \ / \ / \ / \ /           (Data should draw)
      / \ / \ / \ / \
2   (x) ( ) (x) ( ) ( )
      \ / \ / \ / \ /
      / \ / \ / \ / \
1   (x) ( ) (x) (x) (x)

     A   B   C   D   E

    Data's move                                 : b3a4


5   (o) (o) (o) (o) (o)
      \ / \ / \ / \ /
      / \ / \ / \ / \
4   (o) ( ) ( ) ( ) (o)
      \ / \ / \ / \ /
      / \ / \ / \ / \           TURN: x
3   ( ) ( ) ( ) (x) ( )
      \ / \ / \ / \ /           (Player should draw)
      / \ / \ / \ / \
2   (x) ( ) (x) ( ) ( )
      \ / \ / \ / \ /
      / \ / \ / \ / \
```

```
1   (x) ( ) (x) (x) (x)

     A   B   C   D   E

  Player's move [(undo)/([a-e][1-5][a-e][1-5])]: a1b2


5   (o) (o) (o) (o) (o)
      \ / \ / \ / \ /
      / \ / \ / \ / \
4   (o) ( ) ( ) ( ) (o)
      \ / \ / \ / \ /
      / \ / \ / \ / \           TURN: o
3   ( ) ( ) ( ) (x) ( )
      \ / \ / \ / \ /           (Data should draw)
      / \ / \ / \ / \
2   (x) (x) (x) ( ) ( )
      \ / \ / \ / \ /
      / \ / \ / \ / \
1   ( ) ( ) (x) (x) (x)

     A   B   C   D   E

    Data's move                                 : b5c4


5   (o) ( ) (o) (o) (o)
      \ / \ / \ / \ /
      / \ / \ / \ / \
4   (o) ( ) (o) ( ) (o)
      \ / \ / \ / \ /
      / \ / \ / \ / \           TURN: x
3   ( ) ( ) ( ) (x) ( )
      \ / \ / \ / \ /           (Player should draw)
      / \ / \ / \ / \
2   (x) (x) (x) ( ) ( )
      \ / \ / \ / \ /
      / \ / \ / \ / \
1   ( ) ( ) (x) (x) (x)

     A   B   C   D   E

  Player's move [(undo)/([a-e][1-5][a-e][1-5])]: ps

Here are the values of all possible moves:
                Move         Remoteness         Delta
Winning Moves:
Tieing Moves:
                d3e2         Draw                 0
                a2b3         Draw                 0
                a2b1         Draw                 0
                c2b1         Draw                 0
                c2b3         Draw                 0
```

```
                d1e2         Draw                  0
                b2a1         Draw                  0
                b2a3         Draw                  0
                b2c3         Draw                  0
                c1d2         Draw                  0
                e1d2         Draw                  0
Losing Moves:



5   (o) ( ) (o) (o) (o)
      \ / \ / \ / \ /
      / \ / \ / \ / \
4   (o) ( ) (o) ( ) (o)
      \ / \ / \ / \ /
      / \ / \ / \ / \           TURN: x
3   ( ) ( ) ( ) (x) ( )
      \ / \ / \ / \ /           (Player should draw)
      / \ / \ / \ / \
2   (x) (x) (x) ( ) ( )
      \ / \ / \ / \ /
      / \ / \ / \ / \
1   ( ) ( ) (x) (x) (x)

     A   B   C   D   E

  Player's move [(undo)/([a-e][1-5][a-e][1-5])]: d1e2


5   (o) ( ) (o) (o) (o)
      \ / \ / \ / \ /
      / \ / \ / \ / \
4   (o) ( ) (o) ( ) (o)
      \ / \ / \ / \ /
      / \ / \ / \ / \           TURN: o
3   ( ) ( ) ( ) (x) ( )
      \ / \ / \ / \ /           (Data should draw)
      / \ / \ / \ / \
2   (x) (x) (x) ( ) (x)
      \ / \ / \ / \ /
      / \ / \ / \ / \
1   ( ) ( ) (x) ( ) (x)

     A   B   C   D   E
```