GamesmanUni GUI Accessibility and Combinatorial Games

Avery Liou
Dan Garcia, Ed.
Joshua Hug, Ed.

Technical Report No. UCB/EECS-2022-148
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2022/EECS-2022-148.html

May 19, 2022

Acknowledgement

Professor Dan Garcia, for allowing me into his research group and supporting me along the 2 years I was in GamesCrafters (as well as in the two classes I took with him).
Professor Joshua Hug, for being my second reader and helping me improve this paper.
The GamesCrafters Web team, for going along with the improvements and new projects that I wanted to implement.
The GamesCrafters group, for helping with the constant issues with the website and API.
My friends and family, for supporting me along my college career.

GamesmanUni GUI Accessibility and Combinatorial Games

by Avery Liou

Research Project

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

Approval for the Report and Comprehensive Examination:

Committee:

Teaching Professor Dan Garcia
Research Advisor

(Date)

* * * * * * *

Teaching Professor Joshua Hug
Second Reader

(Date)

Table of Contents

Abstract

This report explores the creation of graphical interfaces for two-player, complete-information, abstract strategy games. My project can be divided into two smaller subgoals. First, I wanted to provide more intuitive graphical interfaces for the public to play games on the web that have been solved within the GamesCrafters research group, doing so through the web interface named GamesmanUni. The GamesCrafters Web team and my work on GamesmanUni GUI has been the creation of Graphical User Interfaces (GUIs) for pre-existing games that previously only had text interfaces, improving accessibility. Both the development of custom GUIs and the use of AutoGUI, an automatic GUI creation system, are encapsulated in this work.

The second goal was to bring the world of combinatorial games to the GamesmanUni ecosystem. Combinatorial games are mathematically quite rich, and are a specialization of abstract strategy games in which the goal is always to get the last move. Although the theory behind the games is regularly discussed in the group, they have not been brought into the main GAMESMAN software infrastructure such as the front-end web interface GamesmanUni. By adding several combinatorial games to GAMESMAN, we hope to expand the scope of games that the GamesCrafters research group will explore.

Acknowledgements

- Professor Dan Garcia, for allowing me into his research group and supporting me along the 2 years I was in GamesCrafters (as well as in the two classes I took with him).
- Professor Joshua Hug, for being my second reader and helping me improve this paper.
- The GamesCrafters Web team, for going along with the improvements and new projects that I wanted to implement.
- The GamesCrafters group, for helping with the constant issues with the website and API.
- My friends and family, for supporting me along my college career.

1. Introduction

GamesCrafters is a UC Berkeley research and development group created and headed by Teaching Professor Dan Garcia that has been running for more than two decades. The group originally focused on two-player, complete-information, abstract strategy games, such as Tic-Tac-Toe and Othello, and exhaustively solved the game tree for games that fit these three criteria. Since then, the group has expanded to include puzzles under its scope of examination.

In the last year, the group primarily focused its research and development efforts on three main projects:

1. **GamesmanClassic**: games that are encoded and fully solved in C. Several new games have been encoded and solved and added to this project.
2. **GamesmenUni**: the web graphical interface used to display playable versions of solved games to the general public. The most recent version of the website, Uni v5 [1], differs from the last major version Uni v4 [2], in that there are now computer players available who will always play the best move possible. However, Uni v4, contains some custom GUIs not ported to Uni v5 yet.
3. **GamesCraftersUWAPI (Universal Web API)**: the interface that *serves* the data including board states, move values, and remoteness values to GamesmenUni. The API compiles data from several sources, including GamesmanClassic, hard-coded games with their game tree data encoded in JSON files, and lastly, games written and solved in Python.

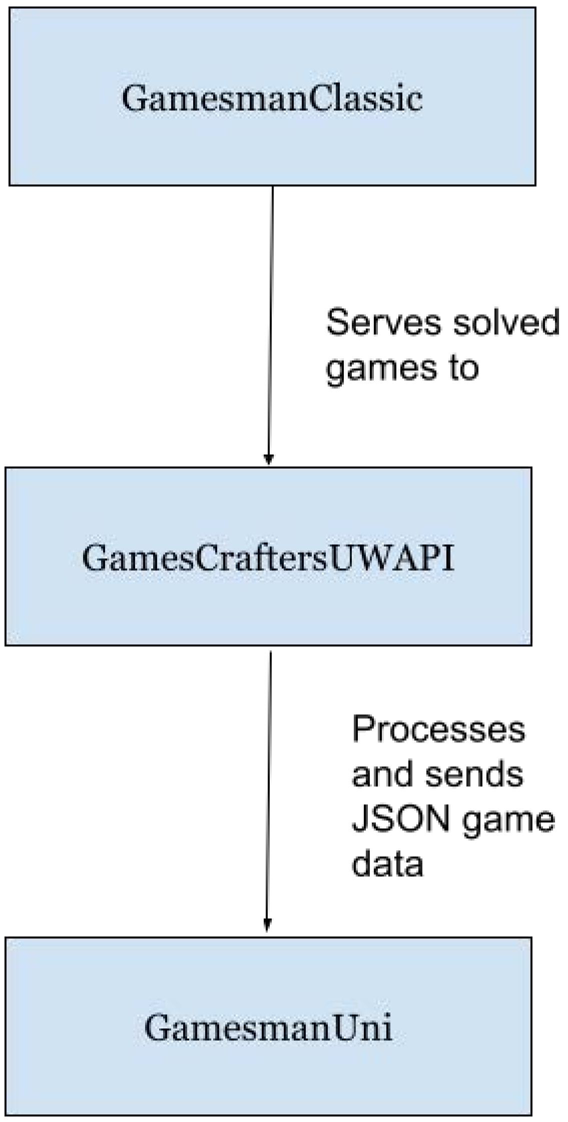


Figure 1: A diagram of the relationship between the main projects of GamesCrafters

The bulk of my project over the past two years has primarily been focused within GamesmanUni and GamesCraftersUWAPI. I also started working on a subset of perfect-information, two-player abstract strategy games known as *combinatorial* games. These games have the additional constraint that the goal is to get the last move (other games have goals to capture certain pieces or align N of one's own pieces in a row). For the front-end work on GamesmanUni I created, I worked on creating clear, identifiable game boards that were easy to play. I defined the following criteria that GUIs and other assorted interfaces had to fulfill:

1. **Complete information:** all possible pieces, moves, and values for a given game board state must be visible and clear.
2. **Ease of use**: the moves for a board must be easy to make.
3. **Click-only**: the moves on a board can only be executed through clicking
4. **Smoothness**: the game should run relatively smoothly on an average computer without significant lag or delay

For my work on GamesCraftersUWAPI, I focused on functionality – adding custom board string supports as well as combinatorial games. For anything that was added, I performed regression testing to ensure that nothing pre-existing in the API failed after committing changes.

2. Background

The following *Background* section will define important terms and projects commonly used in the GamesCrafters research group. It will also briefly summarize the definitions of combinatorial games and their winning and losing positions. Finally, this section will briefly define rules for games explored in this report, including how to play them, and what constitutes a win or a loss.

2.1. GamesCrafters Front-end and Back-end APIs

Before defining the front-end and back-end APIs that are developed and used in GamesCrafters, it is important to first define what "front-end", "back-end", "and "APIs" are. Front-end refers to the layer of a piece of software that presents information, usually the client or the main webpage [19]. In the case of GamesCrafters, this corresponds to GamesmanUni. Back-end refers to the layer of software that accesses data, usually from a database or some other store of information [19]. For GamesCrafters, this is GamesCraftersUWAPI. Finally, an API, standing for application programming interface, is a connection between different programs or pieces of software [20]. In our case, this generally refers to methods in one program that can be called by other programs to access specific information or data. GamesCraftersUWAPI is a prominent example of an API within GamesCrafters. [5]

The main interface for the general public to play games solved in GamesCrafters is through GamesmanUni [2]. Currently, GamesmanUni is in its 5th public version, referred to in this report as Uni v5. The previous version of GamesmanUni is referred to in this paper as

Uni v4. GamesmanUni is written primarily in *VueJS*, *HTML*, *SCSS*, and *TypeScript*. GamesmanUni currently supports interfaces of one-player games, referred to as ‘Puzzles’, and two-player games, referred to as ‘Games’. GamesmanUni also supports the conversion of board strings into front-end interfaces automatically through a system called AutoGUI, defined in the next section. Figure 2 shows the AutoGUI interface for a game called Dragons and Swans.

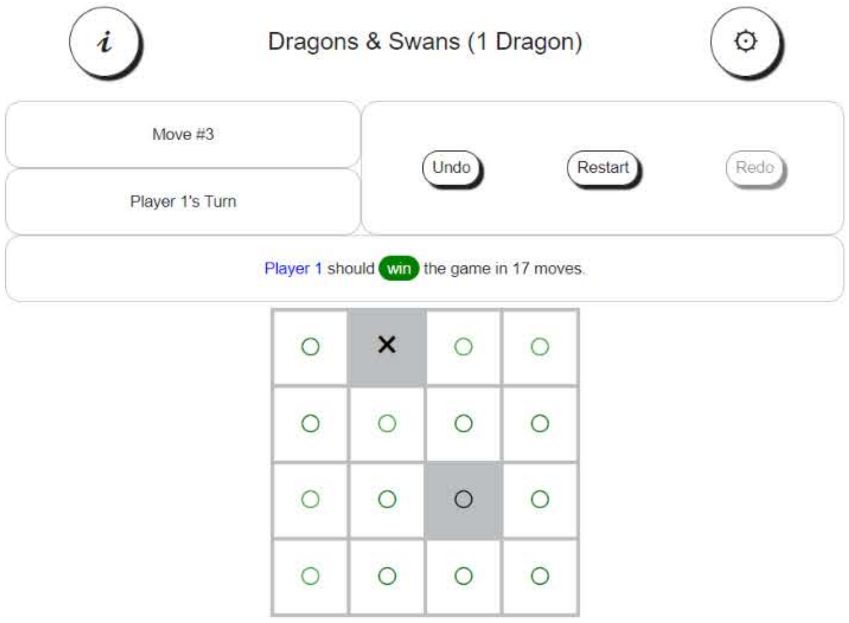
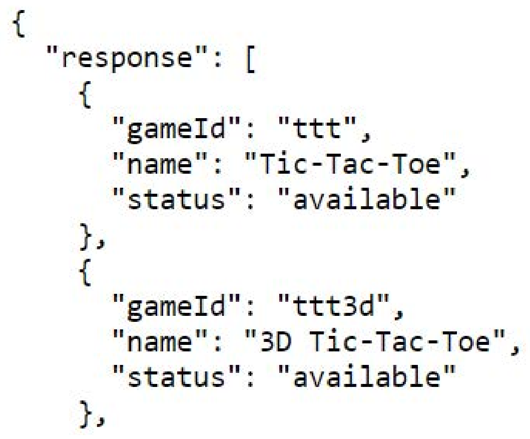

i
Dragons & Swans (1 Dragon)
Move #3
Player 1's Turn
Undo
Restart
Redo
Player 1 should win the game in 17 moves.

Figure 2: The AutoGUI interface for the Dragons and Swans game

The main data source that GamesmanUni draws information from is GamesCraftersUWAPI (GamesCrafters Universal Web Application Programming Interface). The API [4] draws data from several different sources including GamesmanClassic, the original C project used to solve games, and games defined in Python or JSON within the UWAPI repository itself. The architecture abstracts away where the database for each game’s tree is stored and hosted and presents a universal single place that any front end can draw from. GamesCraftersUWAPI provides the entire list of games that have been solved in entirety, solved variants for each game, and corresponding game boards, moves, and values for moves and positions. GamesCraftersUWAPI provides these responses in a JSON format shown in the figure below..



```
{
  "response": [
    {
      "gameId": "ttt",
      "name": "Tic-Tac-Toe",
      "status": "available"
    },
    {
      "gameId": "ttt3d",
      "name": "3D Tic-Tac-Toe",
      "status": "available"
    },
```

Figure 3: Part of the JSON response containing all of the games available on GamesCraftersUWAPI

2.2. AutoGUI

The AutoGUI system, or *auto*matic *graphical user interface system*, is an interface that was developed within GamesmanUni in 2020 to automatically create playable interfaces for grid-based games. The system can automatically generate these interfaces for two-player, complete-information, strategy games that fulfilled the following criteria:

1. Every board string can be displayed as a 2-dimensional rectangular grid.
2. Every move falls under one of the following three categories:

a. Add a piece to the board, replace an existing piece with a new one, or remove a piece.

b. Move an existing piece on the board.

c. Shift a row of the board in a direction by some amount.

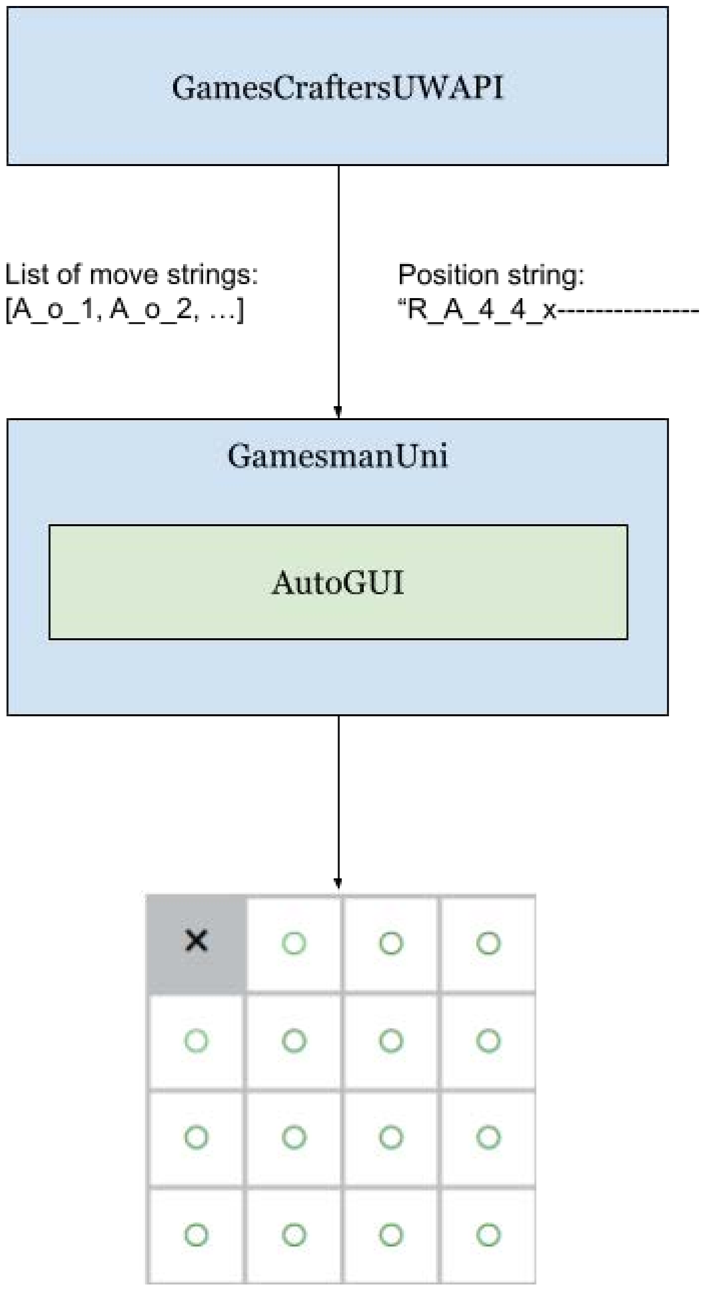
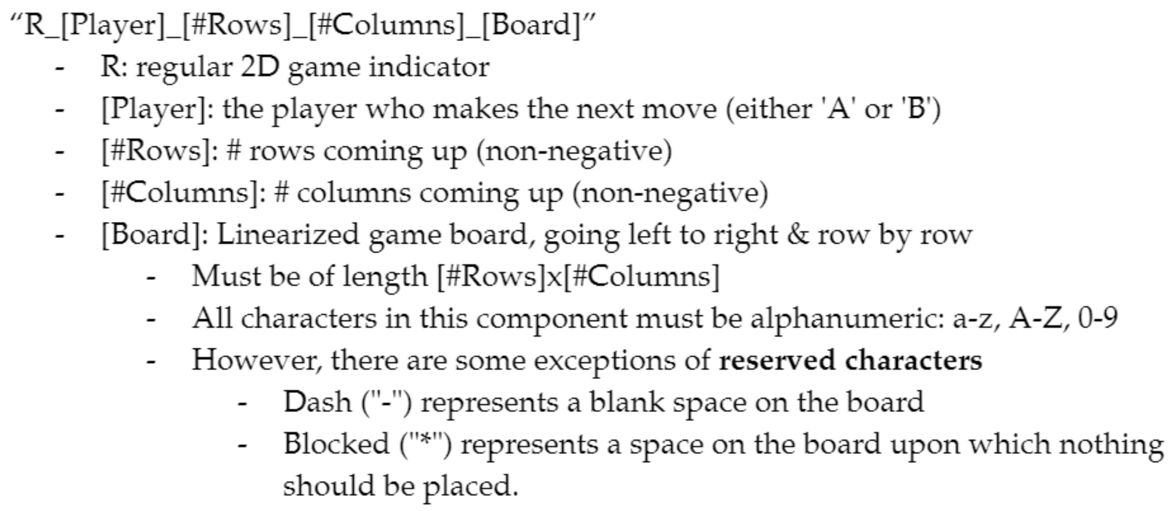


Figure 4: A diagram representing the use of AutoGUI for a game of Dragons and Swans

If the game fulfilled those criteria, then they could be provided to the AutoGUI system to generate a GUI. The board strings provided had to take the format shown in Figure 5. The move strings provided had to take the format shown in Figure 6. [21]


"R_[Player]_[#Rows]_[#Columns]_[Board]"

- R: regular 2D game indicator
- [Player]: the player who makes the next move (either 'A' or 'B')
- [#Rows]: # rows coming up (non-negative)
- [#Columns]: # columns coming up (non-negative)
- [Board]: Linearized game board, going left to right & row by row
    - Must be of length [#Rows]x[#Columns]
    - All characters in this component must be alphanumeric: a-z, A-Z, 0-9
    - However, there are some exceptions of **reserved characters**
        - Dash ("-") represents a blank space on the board
        - Blocked ("*") represents a space on the board upon which nothing should be placed.

Figure 5: The required string format for board strings provided to AutoGUI. [21]

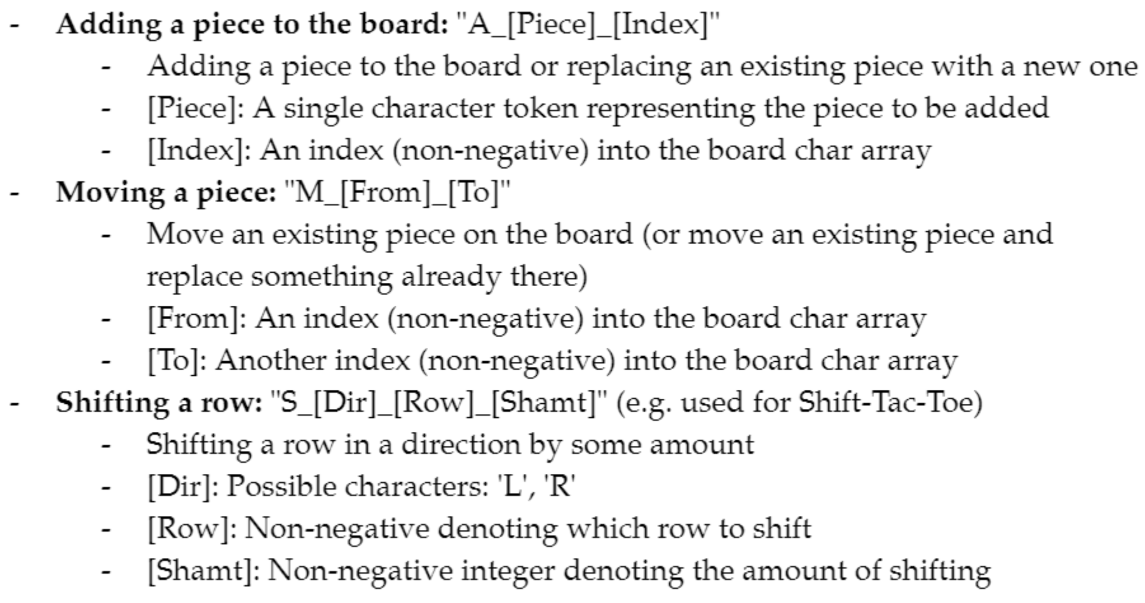

- **Adding a piece to the board:** "A_[Piece]_[Index]"
    - Adding a piece to the board or replacing an existing piece with a new one
    - [Piece]: A single character token representing the piece to be added
    - [Index]: An index (non-negative) into the board char array
- **Moving a piece:** "M_[From]_[To]"
    - Move an existing piece on the board (or move an existing piece and replace something already there)
    - [From]: An index (non-negative) into the board char array
    - [To]: Another index (non-negative) into the board char array
- **Shifting a row:** "S_[Dir]_[Row]_[Shamt]" (e.g. used for Shift-Tac-Toe)
    - Shifting a row in a direction by some amount
    - [Dir]: Possible characters: 'L', 'R'
    - [Row]: Non-negative denoting which row to shift
    - [Shamt]: Non-negative integer denoting the amount of shifting

Figure 6: The required string format for move strings provided to AutoGUI [21]

2.3. Combinatorial Game, Nimbers, and Mex values

Combinatorial games are a subset of 2-player, perfect information games where the goal is to make the last move. Some examples of combinatorial games include Nim and Kayles, which are explored more in depth later in this paper.

Combinatorial games are incredibly rich in depth of theory and mathematics. The foundation of these games are based in surreal numbers, a “totally ordered proper class containing the real numbers as well as infinite and infinitesimal numbers.” [22] There are many properties that are specific to combinatorial games that are not explored in this paper, such as Grundy values, nim-sequences, and P-positions [16]. We mainly focus on the position values of win/lose/tie/draw as well as the remoteness of a position instead. [1]

Any given combinatorial game position has an integer value, also called a nimber, representing whether or not a game is a win/loss for the player whose turn it is, given that players alternate turns playing only that game. Any game position whether the player is unable to make a move (and has thus, lost) has a nimber of 0. All other non-zero nimber values are winning positions. To evaluate non-terminal game positions, one must first calculate the nimbers of all *children* positions reachable in one move from the current position. The nimber of the current position is the **Mex** of the set of those nimbers. The mex value, standing for minimum excluded value, is the smallest non-negative integer from the whole set that does not belong to the set. For example, the mex of {0, 1, 3} is 2. [13]

A particular property of combinatorial games is the ability to combine them. One can combine multiple combinatorial games and find the nimber of the combined game by taking the Nim sum of all the individual games. The Nim sum is defined to be the bitwise XOR of the nimbers of each of the individual games. If a combinatorial game position with a nimber of 2 and a combinatorial game position with a nimber of 1 were combined, that combined game position would have a nimber of 2 XOR 1 = 3. [7]

2.4. Games

This section explores the rules of the games that have been worked on in this report. This includes combinatorial games added to the system as well as custom GUIs made for previous computational games.

2.4.1. Nim

The game of Nim is a two-player combinatorial game whose rules are quite simple– there are *n* number of piles, each with any number of sticks in each pile, which do not have to be equal. The play alternates between two players– on a player's turn, they may pick any number of sticks from a single pile. This can range from one stick to the entire pile. Like all combinatorial games, the player who is unable to make a move loses. All combinatorial games can be reduced to a Nim game in some capacity [6]. A Nim position is defined as the number of sticks in each pile, which can be represented as an array of numbers. For example, if the Nim game starts with piles of 1, 3, 5, and 7 sticks, we can represent the position as [1, 3, 5, 7].

The game of Nim is quite unique within the scope of combinatorial games in that it has a completely closed-form solution. The Mex value of any given Nim position is simply the bitwise XOR of all of the piles. In other words, for the aforementioned position of [1, 3, 5, 7], we find the position by calculating 1 XOR 3 XOR 5 XOR 7, thereby obtaining a MEX value of 0, or a losing position.

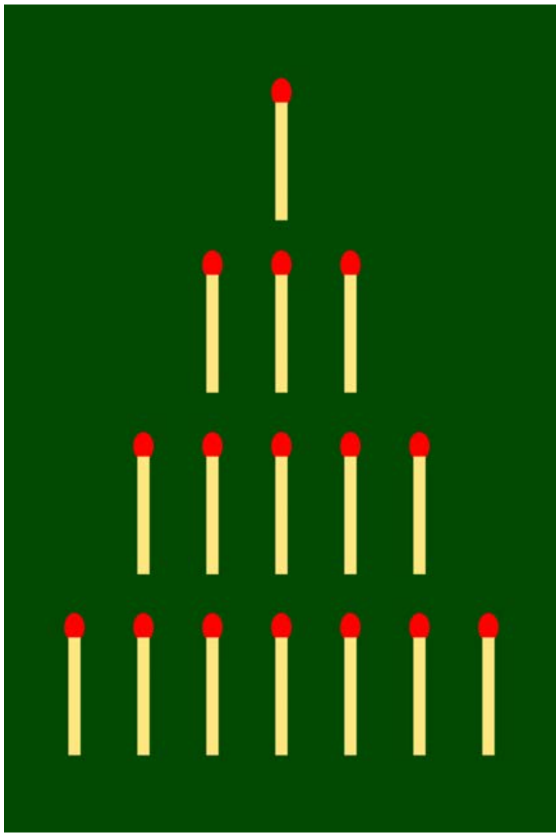


Figure 7: A Nim game with a game board of [1, 3, 5, 7]. [17]

2.4.2. Dawson's Chess

The game of Dawson's Chess is played on a 3 x n chessboard. On the first row is a row of pawns for one player, and on the last row is a row of pawns for the other player. Figure 8 depicts the starting position for a 3 x 12 Dawson's Chess board. Each player alternates turns– a player may only push a pawn forward if and only if there are no pawns that they can capture. If they can capture a pawn, their move must be a capture. Pawns that reach the other end of the board do not promote to another piece, as is in traditional chess rules, and cannot keep moving. The game ends when a player is unable to make a valid move. [9]

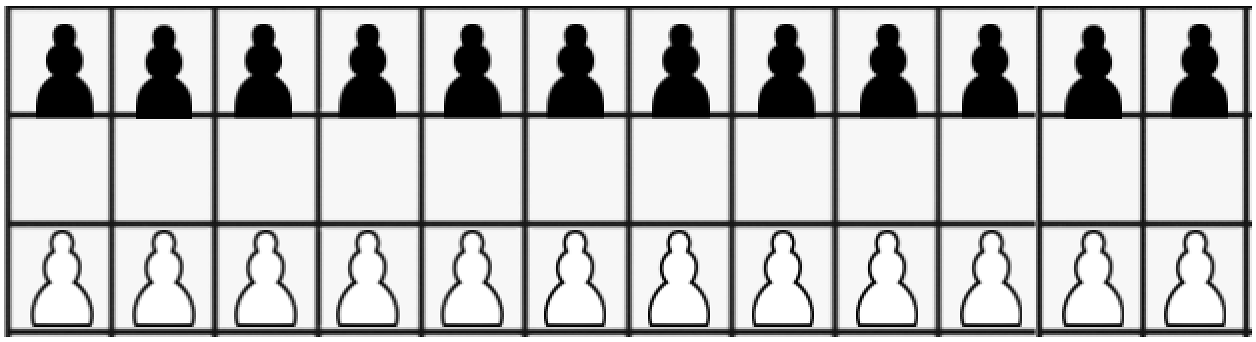


Figure 8: The 3 x 12 Dawson's Chess starting position

The game is reducible to a single row where the players take turns picking a spot on the board to place a piece. Placing a piece on the board also blocks out the adjacent spaces from pieces being placed. The position in Figure 10 depicts a position where one player has placed a piece in position 6, blocking position 5 and 7. Identically, if one player pushes their pawn in column 6 in Figure 8, because of the rule requiring captures to be made, this leads to a chain of captures where both players lose their own pawns in column 4 and column 6. Then, the pawns left in column 5 are directly adjacent and thus cannot be moved anymore. The sequence can be seen in the figure below.

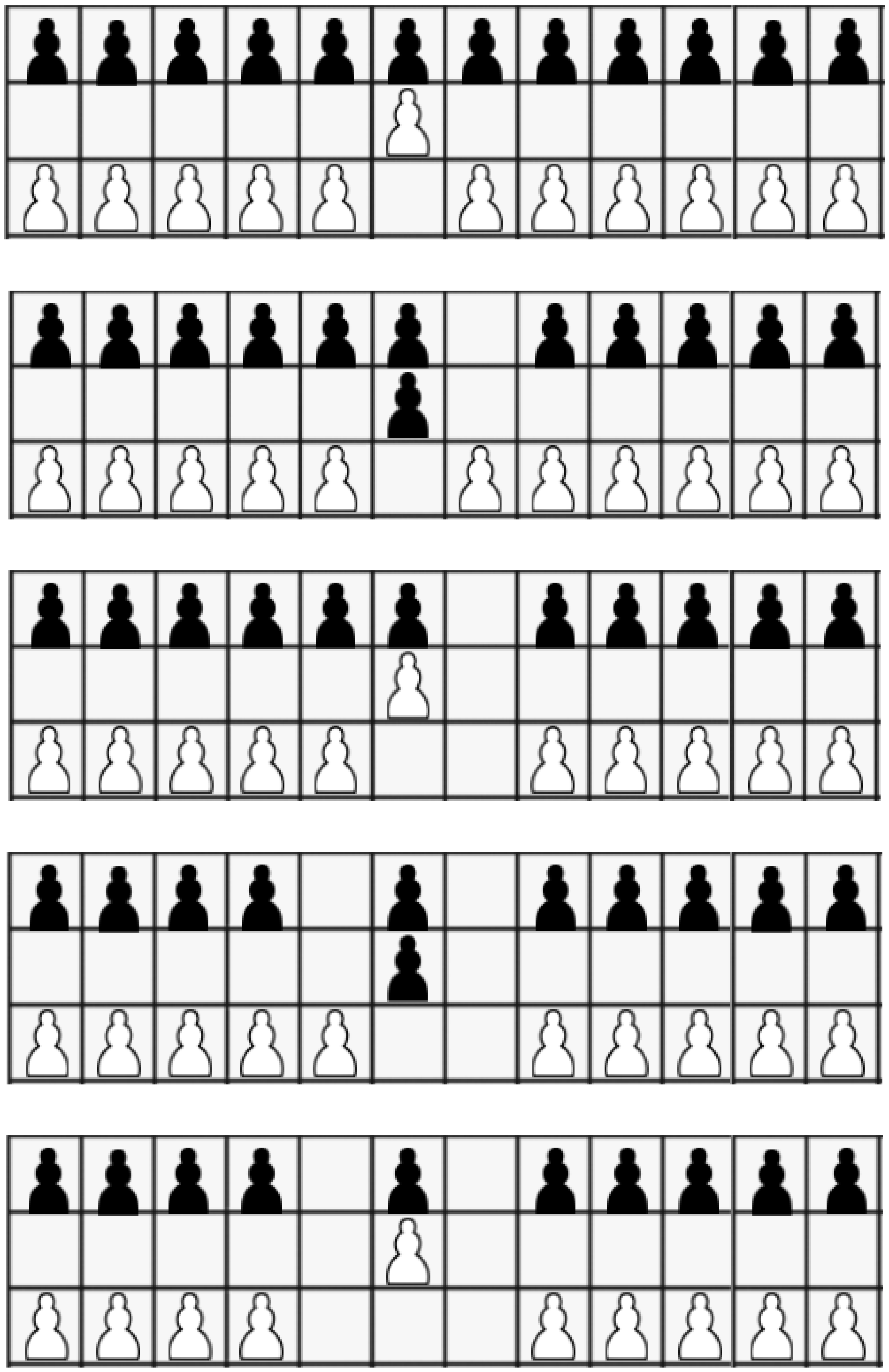


Figure 9: The sequence of moves that leads to the position equivalent to that of Figure 8. Neither white or black has pawns in position 5 or position 7, and the pawns in position 6 are blocking each other.

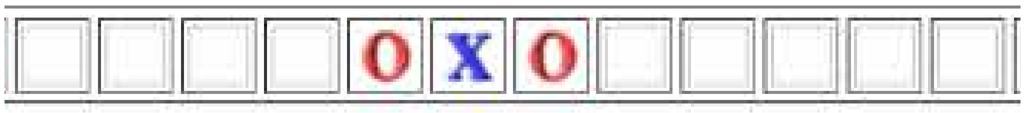
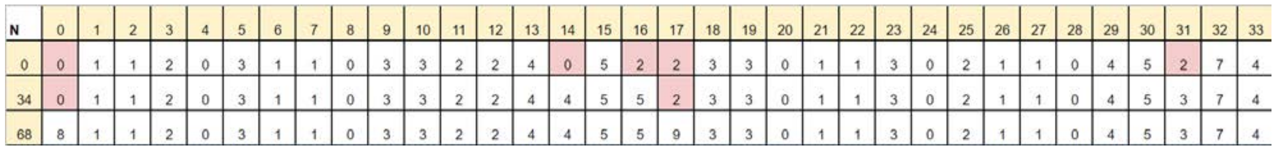


Figure 10: An equivalent n=12 Dawson’s Chess position where one player has placed a piece in position 6, represented by the x’s. Position 5 and position 7 are blocked off as a result, represented by the o’s. [9]

The numbers for any given starting Dawson’s Chess position can be looked up from the following table in Figure 11. The sequence given in Figure 11 has a period of 34 with exceptions at n=0, 14, 16, 17, 31, 34 and 51 (highlighted in red). Thus, we only have to record the periodic sequence (and included exceptions) to have the nimber for any given starting position.

Once a move has been made, the nimber of the board can be deduced by taking the Nim sum of all disjoint, continuous chains of empty spaces. In Figure 10, we see a space of size 4 on the left and a space of size 5 on the right. Thus, we can look up the nimbers of starting position 4 and starting position 5 and take their XOR. 0 XOR 3 is 3, thus the nimber value of the position in Figure 10 is 3, which is a winning position.



N	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33
0	0	1	1	2	0	3	1	1	0	3	3	2	2	4	0	5	2	2	3	3	0	1	1	3	0	2	1	1	0	4	5	2	7	4
34	0	1	1	2	0	3	1	1	0	3	3	2	2	4	4	5	5	2	3	3	0	1	1	3	0	2	1	1	0	4	5	3	7	4
68	8	1	1	2	0	3	1	1	0	3	3	2	2	4	4	5	5	9	3	3	0	1	1	3	0	2	1	1	0	4	5	3	7	4

Figure 11: The Nimber sequence for Dawson’s Chess with period 34. [10]

2.4.3. Kayles

Kayles is a game typically depicted as starting with a single row of N upright pins. Two players alternate throwing a ball to knock over either a single pin or two adjacent pins. The game ends when there are no pins left to knock over. [23]

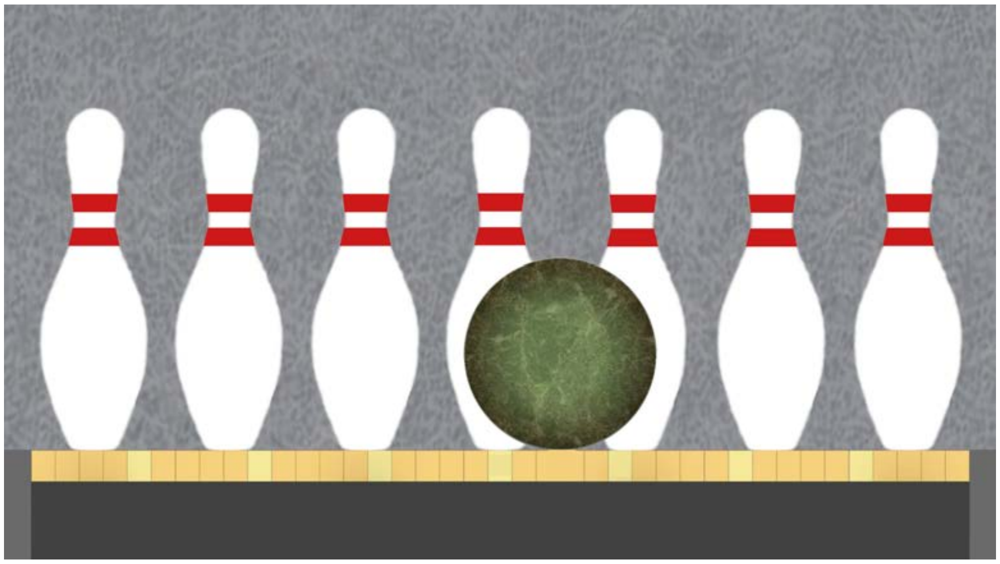


Figure 12: An n=7 game of Kayles. A player is about to knock over the pins in position 3 and position 4 [18]

The game can be equivalently represented as an row of n empty spaces, where players take turns placing a piece either in one space or two adjacent spaces.

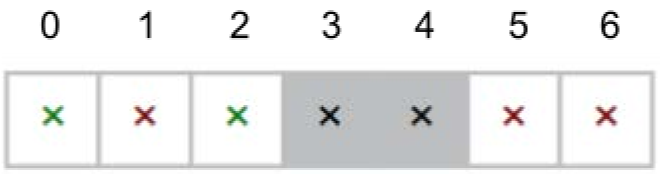
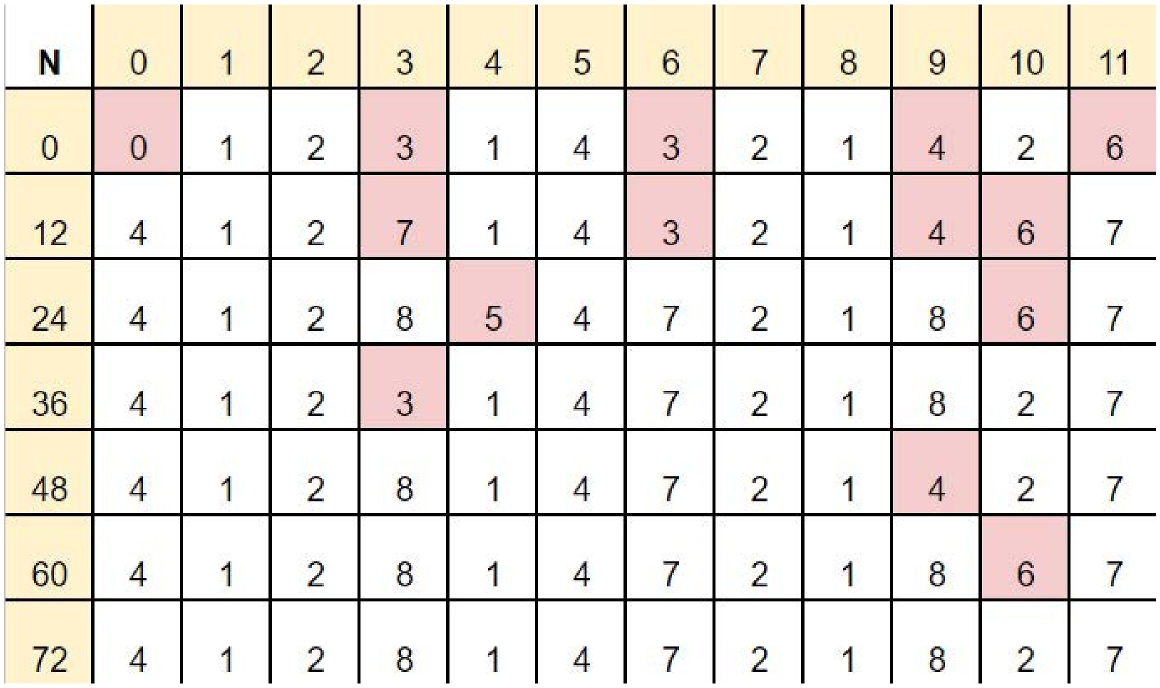


Figure 13: An n=7 game of Kayles, where the player placed pieces in adjacent positions 3 and 4.

The numbers for any given starting Kayles position can be looked up from the following table in Figure 14. The sequence given in Figure 14 has a period of 12 with exceptions at n=0, 3, 6, 9, 11, 15, 18, 21, 22, 28, 34, 39, 57 and 70, highlighted in red.

Once a move has been made, the nimber of the board can be deduced by taking the Nim sum of all disjoint, continuous chains of empty spaces, similar to Dawson's Chess. In Figure 13, we see a space of size 3 on the left and a space of size 2 on the right. Thus, we can look up the nimbers of starting position 2 and starting position 3 and take their XOR. 3 XOR 2 is 1, thus the nimber value of the position in Figure 13 is 1, which is a winning position.



N	0	1	2	3	4	5	6	7	8	9	10	11
0	0	1	2	3	1	4	3	2	1	4	2	6
12	4	1	2	7	1	4	3	2	1	4	6	7
24	4	1	2	8	5	4	7	2	1	8	6	7
36	4	1	2	3	1	4	7	2	1	8	2	7
48	4	1	2	8	1	4	7	2	1	4	2	7
60	4	1	2	8	1	4	7	2	1	8	6	7
72	4	1	2	8	1	4	7	2	1	8	2	7

Figure 14: The Nimbers for Kayles with period 12. [11]

2.4.4. Mancala

Mancala is a family of two-player games involving small stones placed in pits, or pockets, where the objective is to capture the opponent's pieces. The version of Mancala that is solved and presented in GamesmanUni has three pockets on opposing sides of a board, each with three pieces. On the left and right sides of the board, there are large pockets, called stores, each belonging to one player. Players make moves by "scooping up their stones from one of their pockets and dropping one stone into the pocket to the right, continuing up the board counterclockwise placing one stone into each pocket." The game ends when there are no stones left in the non-store pockets– whichever player has more stones in their corresponding store wins. This game has no closed-form solution and was exhaustively solved for a 6-pocket version in GamesmanClassic. [24]

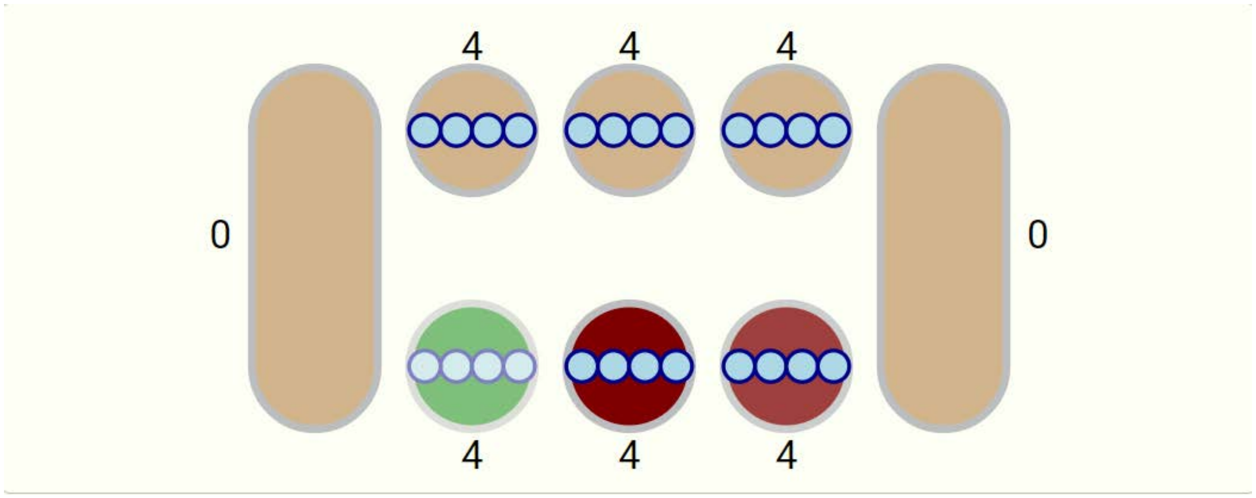


Figure 15: The Mancala available in GamesmanUni

2.4.5. Dao

Dao is a game played on a 4x4 square board. One player has pieces on each square of one diagonal, and the other player has pieces on each square of the other diagonal, shown in the figure below. [25]

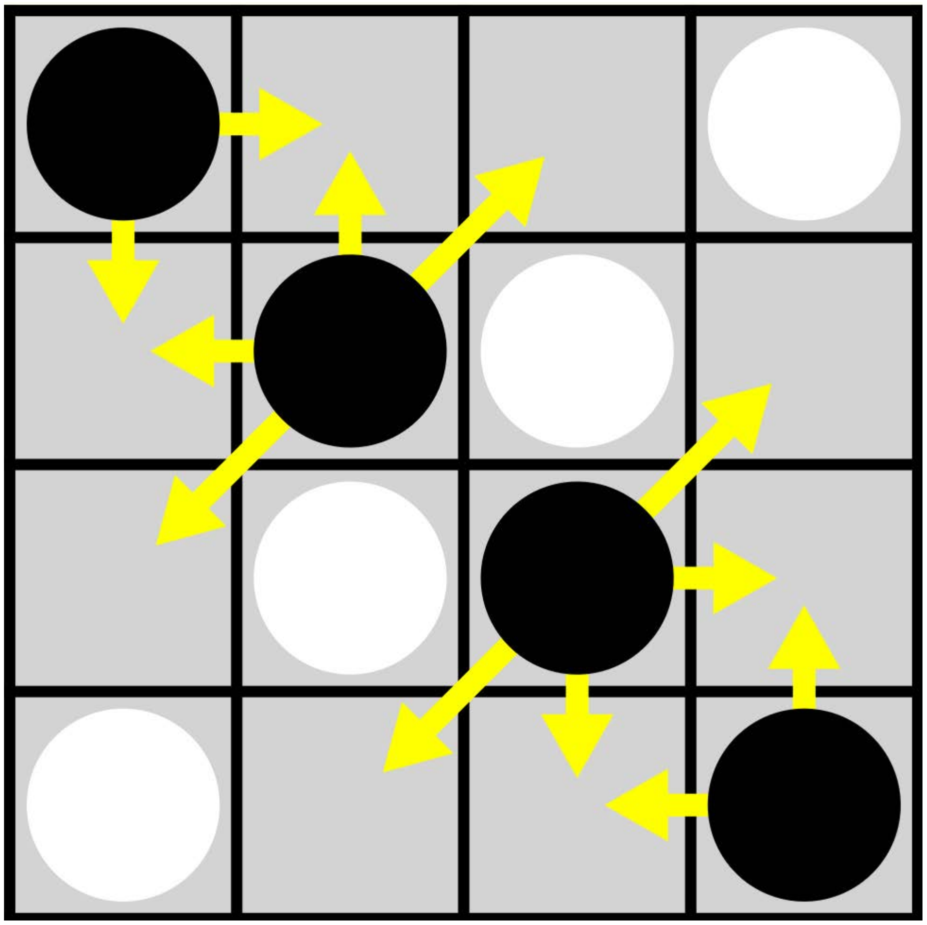


Figure 16: The starting board for a Dao game.

Players take turns moving a single one of their pieces horizontally, vertically, or diagonally. Upon moving a piece in a direction, the piece slides until it hits another piece or the edge of the board, where it stops. For example, moving the top left piece in the figure above to the right results in the board shown below.

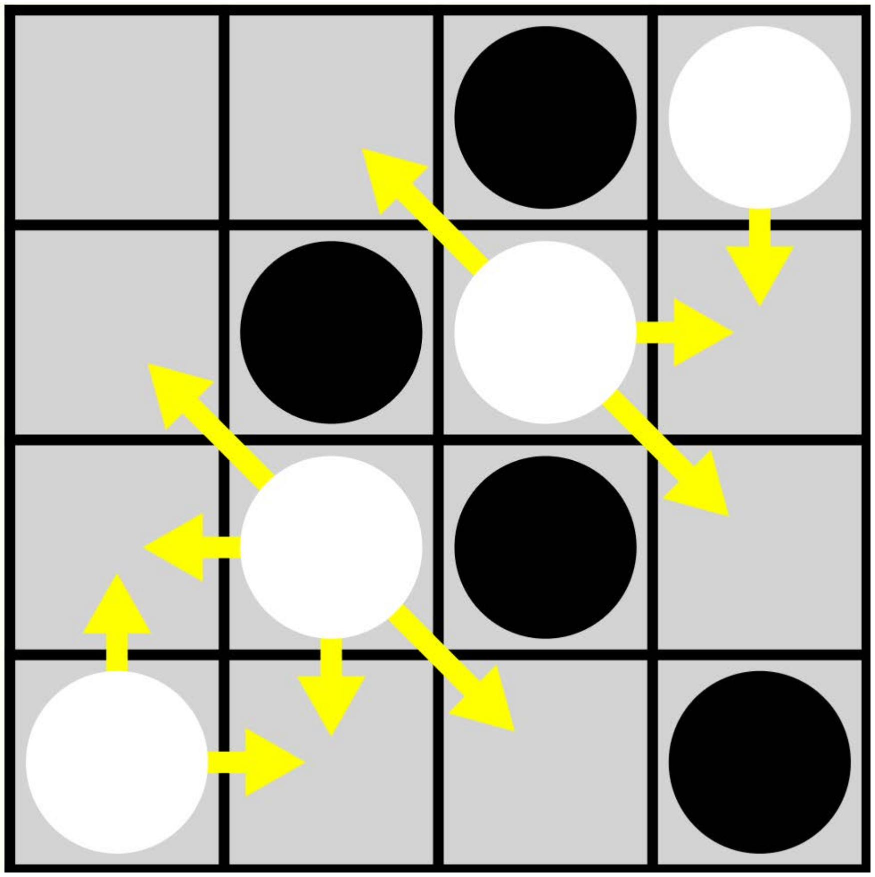


Figure 17: The board after moving the top left piece to the right.

A player wins when they have all four of their pieces fulfilling one of the following criteria:

1. In a straight line (vertically, horizontally, or diagonally).
2. In each of the four corners of the board
3. In a 2x2 square

3. Results

The *Results* section details GUIs created on Uni v4 and Uni v5, as well as the addition of games with custom starts and combinatorial games in Uni v5.

3.1. Uni v4

The Uni v4 section focuses on work done on the previous version on the website. Result on Uni v4 was focused on the custom GUIs of Mancala and Dao.

3.1.1. Mancala Custom GUI

The Mancala custom GUI is currently located on Uni v4 [3]. The GUI was created for a 6-pocket (3 on each player's side) version of the game, with each player having 4 stones in each pocket on their side of the board, totaling 12 pieces for each player. The custom GUI was created entirely using SVGs, and thus is automatically scalable. Features of the game beyond basic GamesmanUni functionalities include:

1. Numeric counters or the number of pieces in each pocket and store
2. Opacity lightening on mouseover of each pocket
3. Coloring on pockets that indicate winning/losing moves, including different opacities to show varying remotenesses of the moves

As seen in Figure 18, the pockets are represented by circles while the stores are represented by oblong ovals. The blue circles represent the 'stones'. The numbers adjacent to each shape display the number of pieces in that pocket/store. A move is triggered when a user clicks anywhere within their pocket.

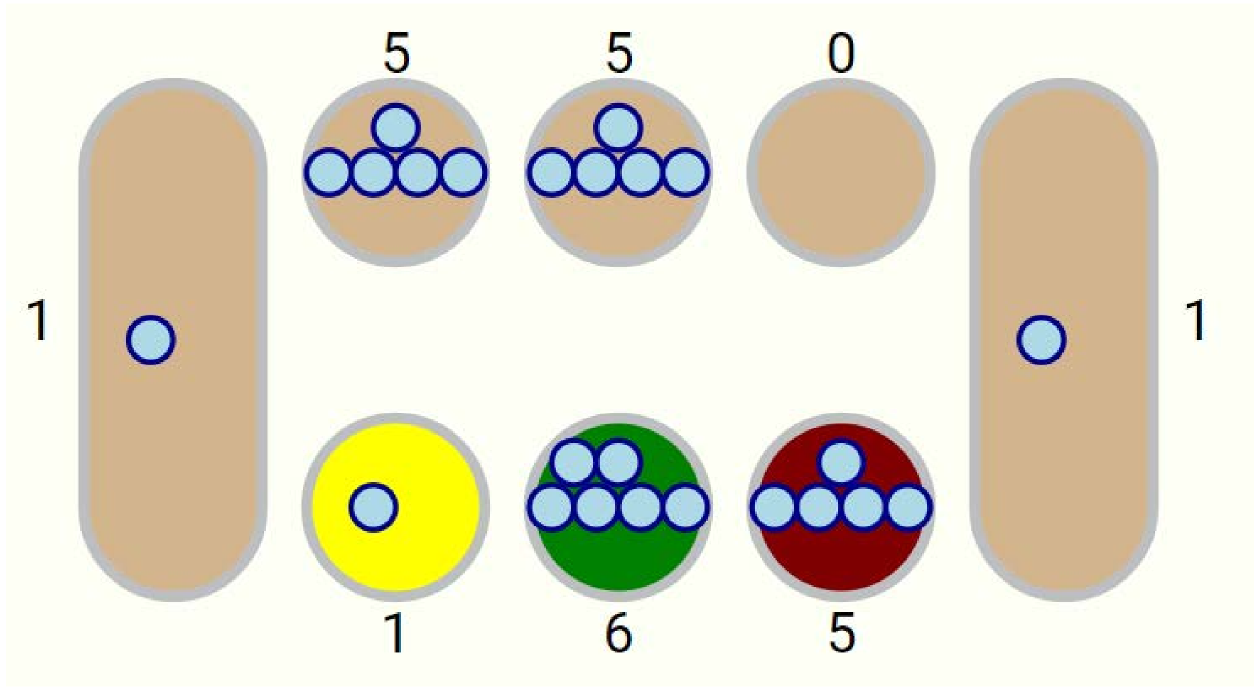


Figure 18: The Mancala custom GUI. [14]

3.1.2. Dao Custom GUI

The Dao custom GUI is currently located on Uni v4. The GUI was created for a 4x4 version of the game, with each player having pieces across a diagonal of the board. The custom GUI was created entirely using SVGs, and thus is automatically scalable. The animations for the arrows and the sliding pieces were done with built-in CSS properties. The GamesmanUni Dao is one of the first online playable versions of the game and also the first version of the game with animations. Features of this game beyond basic GamesmanUni functionalities include:

1. Pulsing animations for mouseover on each arrow
2. Coloring on arrows that indicate winning/losing moves, including different opacities to show delta remoteness
3. Custom animations for sliding of pieces on move, including undo, redo, and restart

As seen in Figure 19, the board is a 4x4 grid with black pieces representing one player, and white pieces representing the other. For the player whose turn it is, arrows are displayed on each of their pieces, showing where the piece can be slid. In this figure, the third black piece on the diagonal has a larger arrow in the upper right direction because the mouse is hovering over it. The arrows are sized so that even when crossing each other, they remain distinct, and are consistent in styling with the arrows in our Tcl/Tk GUI [1].

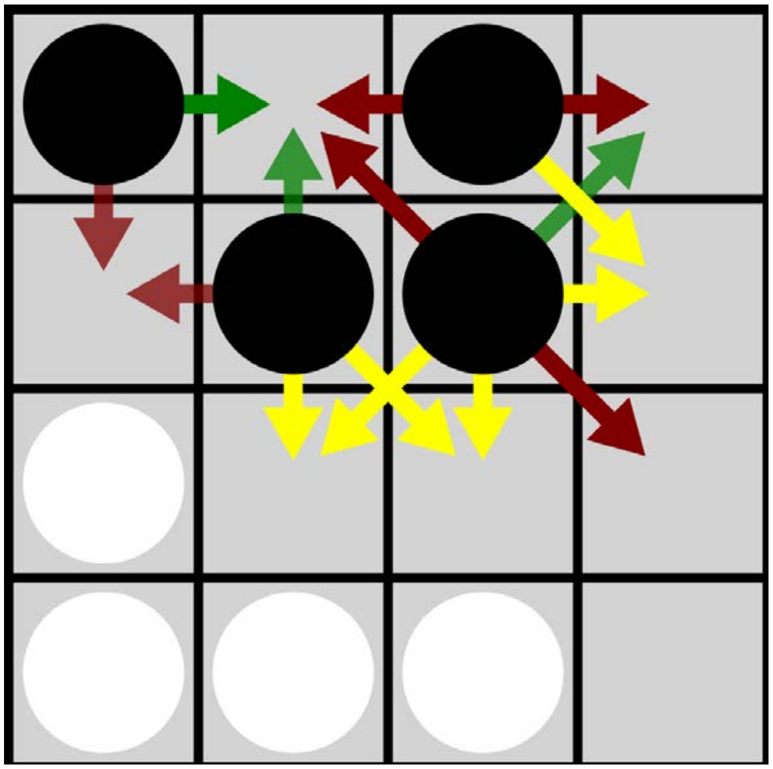


Figure 19: The Dao custom GUI. [15]

3.2. Uni v5 AutoGUI Port

As part of the transition from Uni v4 to Uni v5, the AutoGUI had to be ported over into the new codebase. In the case of this port, my goal was to port AutoGUI in a form identical to that of Uni v4. As seen in Figure 20 and Figure 21, Othello, which is a game utilizing the AutoGUI system, looks nearly identical in its game board between Uni v4 and Uni v5, albeit with some small differences in circle sizes. The readability and ease of use that is present in the AutoGUI in Uni v4 is mirrored in the AutoGUI in Uni v5.

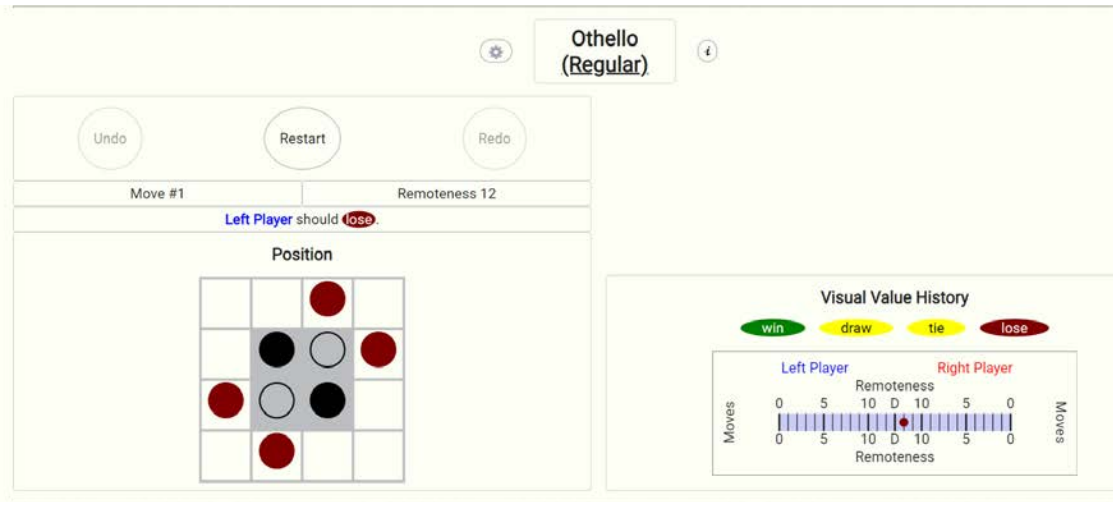


Figure 20: Othello in Uni v4

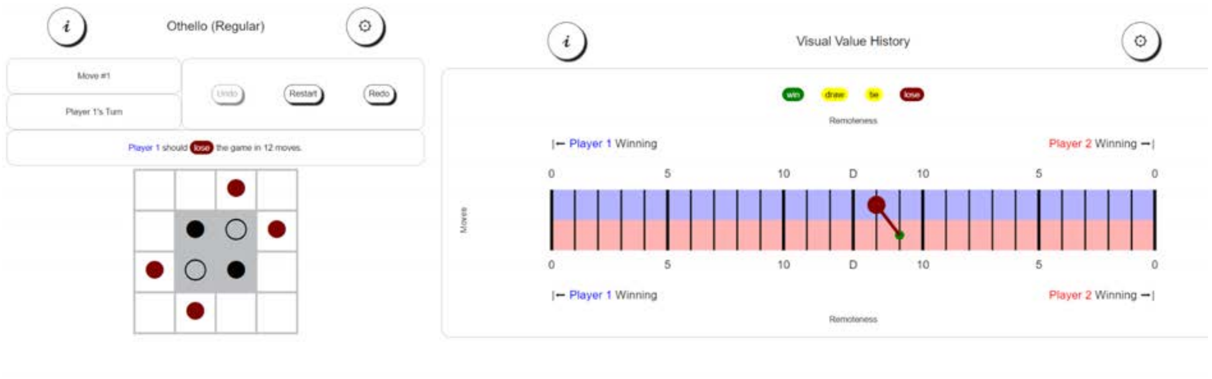


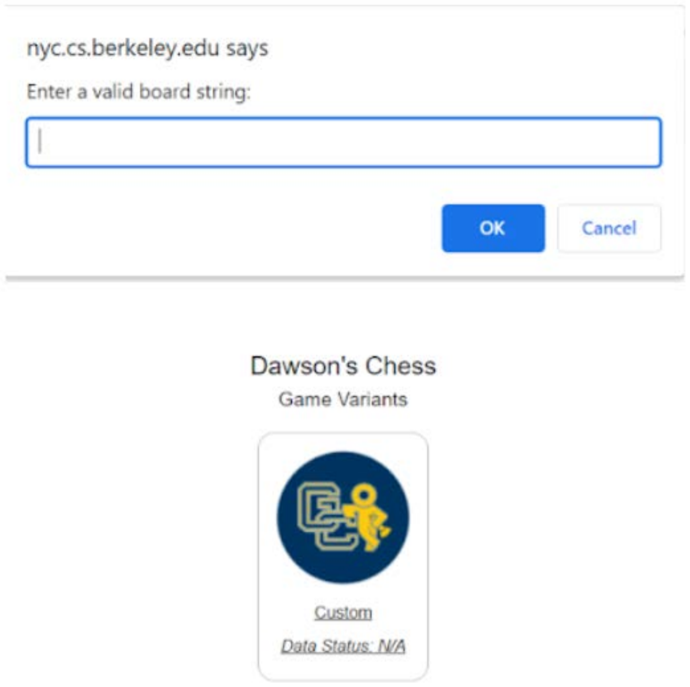
Figure 21: Othello in Uni v5

3.3. Custom Board String Input

For games with closed-form solutions, there is no need to fully solve the entire game tree if we can deduce the solution from a given board string. This condition is particularly common in combinatorial games such as Nim and Kayles, where we can calculate whether or not a position is winning or losing from the board string from their periodic sequence (or in the case of Nim, a simple calculation of bitwise XOR). Since we can deduce the solution from any board string, we are not constrained to only board sizes that we have solved – we can play any board size the user requests.

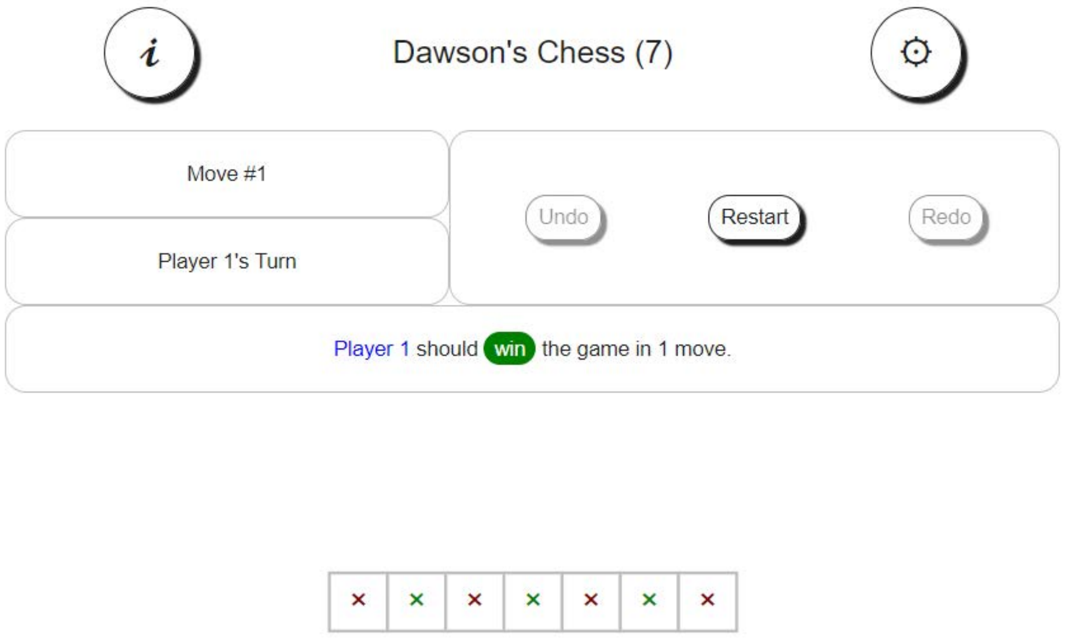
The interface for games that have this property allows the user to type in and enter a starting board string or size in a specified format. As seen in Figures 22 and 23, after beginning a Dawson's Chess game, the user has input a starting board size of 7. Then, a Dawson's Chess board of N = 7 has been created. Although the computational limit for these custom boards have not been found, in the case of Dawson's Chess and Kayles, the readability of the board is dramatically impaired for N > 10 on a 15-inch screen.

Although each board string that is sent by Uni to UWAPI must have its solution calculated in the backend instead of through a database lookup, these custom boards run quite fast in practice, with no noticeable delay or lag in gameplay. The "calculation" itself is a simple lookup and XOR of MEX numbers, resulting in a constant runtime. In practice, the performance of these boards is more limited by the size impacting readability rather than the calculation runtime.


nyc.cs.berkeley.edu says

Enter a valid board string:

Figure 22: The user inputting a starting board size of 7



Player 1 should win the game in 1 move.

Figure 23: A starting board of n=7 has been created

3.4. Combinatorial Games

The *Combinatorial Games* section details the combinatorial games added to GamesCraftersUWAPI and GamesmanUni. Each game also has a custom starting board input available, as detailed in the *Custom Board String Input* section. Because each of these games had a closed form solution where we did not exhaustively solve the entire game tree, we also did not have access to the remoteness information of any given move. We used a remoteness of 1 as a placeholder for all moves of these combinatorial games.

3.4.1. Nim

The game of Nim was added to GamesCraftersUWAPI and GamesmanUni along with custom board input mentioned in the *Custom Board String Input* section. The GUI for Nim in GamesmanUni was created through the AutoGUI system. For the custom starting board string input, Nim accepts underscore delimited integers representing the size of each pile. Figure 24 shows the game board created for the custom board string input of the game of 1, 3, and 5 sticks, represented in each column of the board. Each column represents one pile of sticks and the player takes all the sticks from the piece they select to the top of the pile. For example, in Figure 24, if the player took the stick highlighted in green, they would take away that stick as well as the two directly above it.

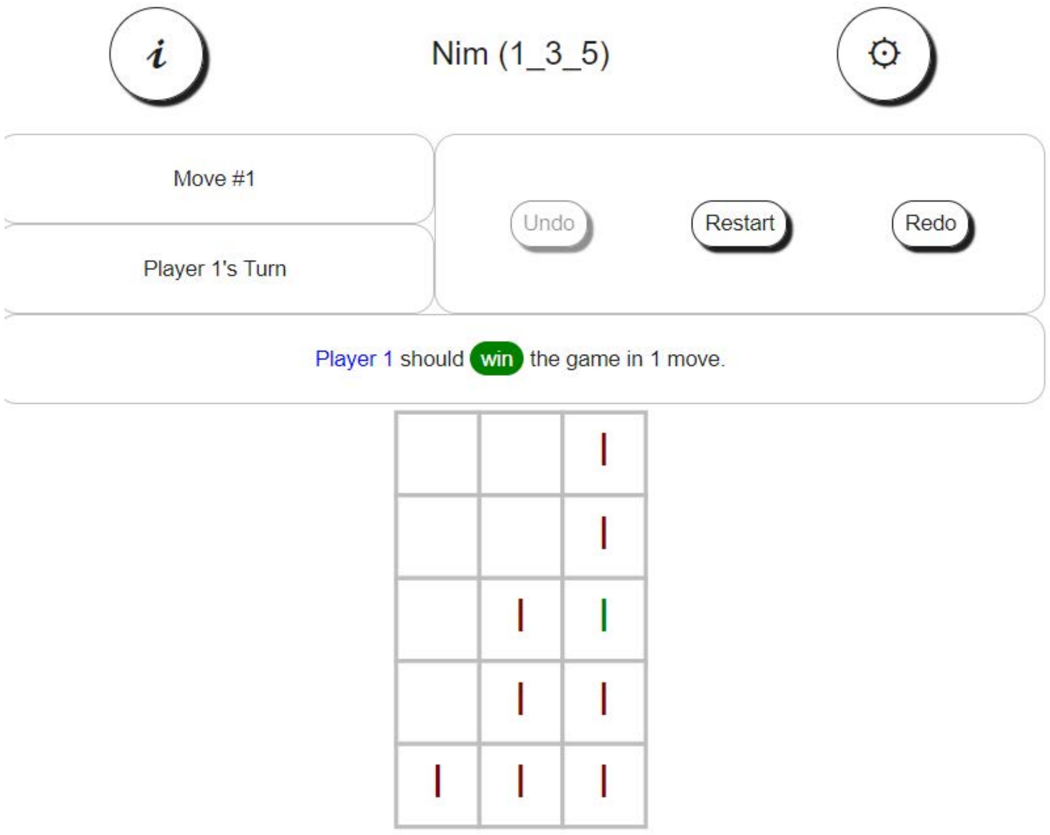


Figure 24: Nim board for 1_3_5 in Uni v5

3.4.2. Dawson's Chess

The game of Dawson's Chess was added to GamesCraftersUWAPI and GamesmanUni along with custom board input. The GUI for Dawson's Chess in GamesmanUni was created through the AutoGUI system. The Dawson's Chess graphical representation follows the form of the game described in Figure 10. For the custom starting board input, Dawson's Chess accepts single integer values representing the length of the board, limited only by readability issues for N > 10 mentioned in the *Custom Board String Input* section. Figure 25 shows the game board created for the custom input of 7. Each space in the row represents a place where a piece can be placed.

After the player places a piece, the two adjacent spaces next to it are automatically filled as well to denote that are unavailable for piece placement.

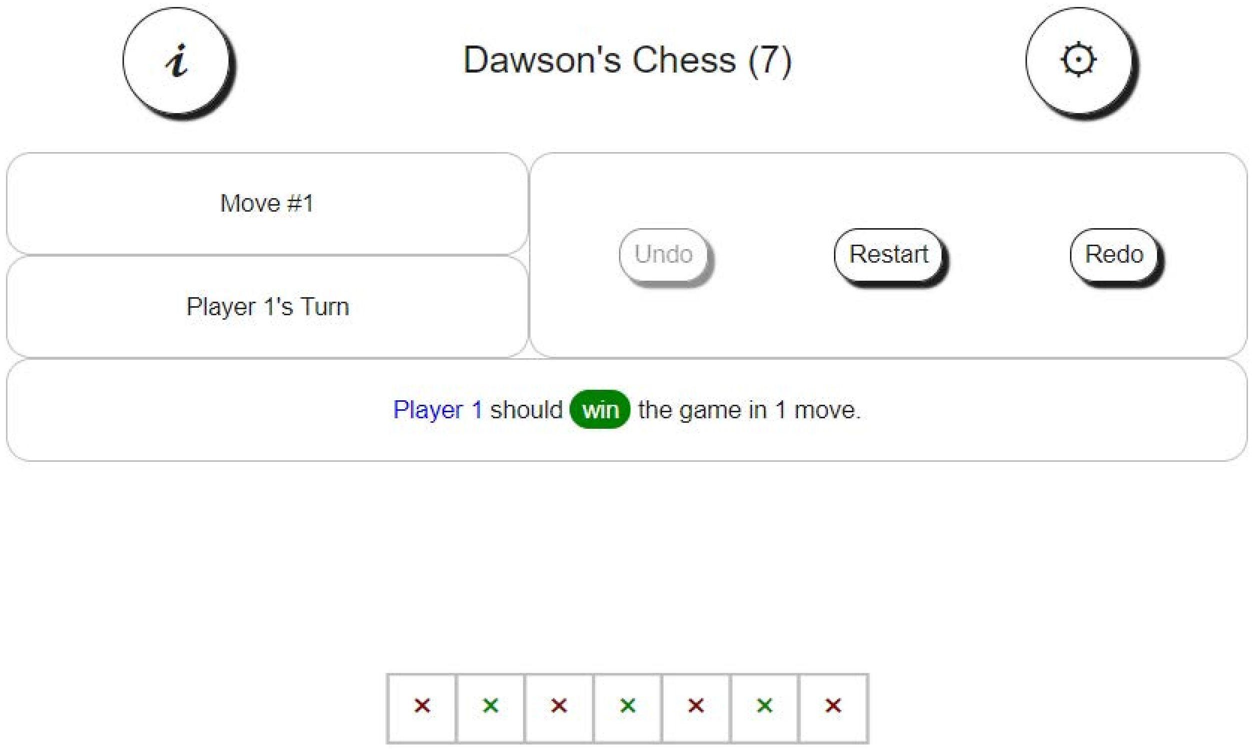
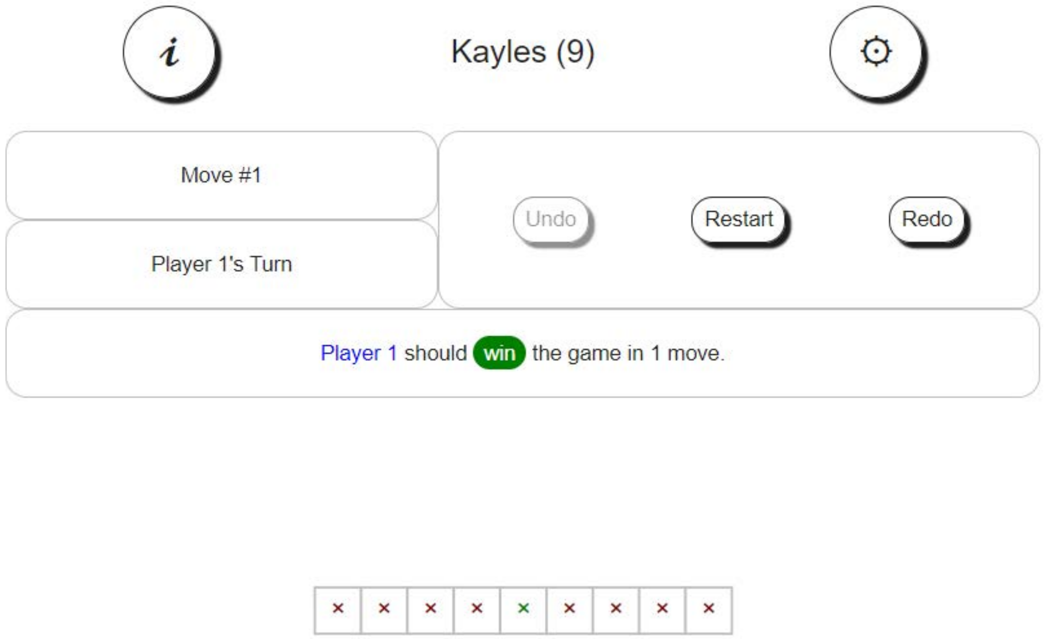


Figure 25: Dawson's Chess board for input 7 in Uni v5

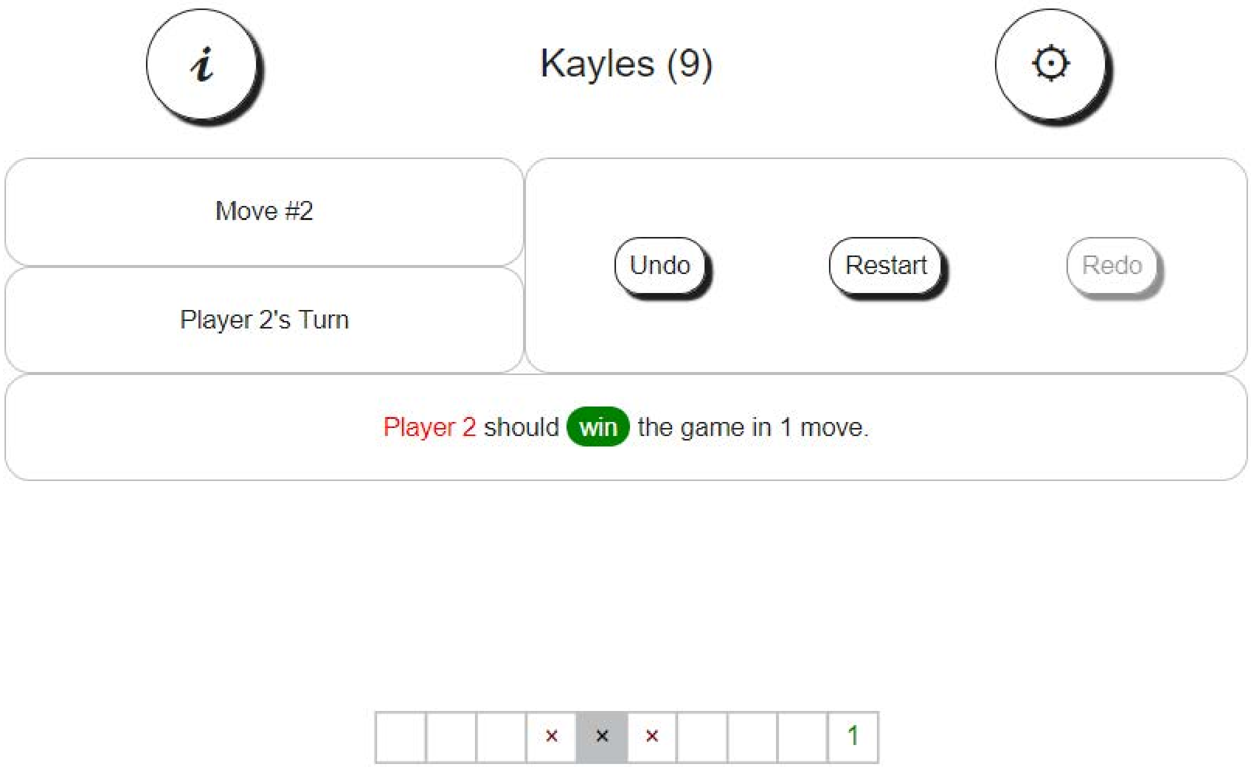
3.4.3. Kayles

The game of Kayles was added to GamesCraftersUWAPI and GamesmanUni along with custom board input. The GUI for Kayles in GamesmanUni was created through the AutoGUI system. The Kayles graphical representation follows the form of the game described in Figure 13. For the custom starting board input, Kayles accepts single integer values representing the length of the board. Figure 26 shows the game board created for the custom input of 9. Each space in the row represents a place where a piece can be placed. After the player places a piece, they have the option of selecting one of the adjacent spaces to place two pieces down in one turn or selecting the ‘1’ at the very end of the board to end their turn with a singular piece placement. This additional ‘1’ space allows for these two-phase moves, where the same player clicks twice to complete any given move. Figure 27 depicts the middle of the player’s turn - after placing the piece down, the player can end their turn by clicking ‘1’ or place one of the adjacent pieces.



Player 1 should win the game in 1 move.

Figure 26: Kayles board for input 9 in Uni v5



Move #2

Player 2's Turn

Player 2 should win the game in 1 move.

Figure 27: A piece has been placed down as indicated by the grey filled square – the player has the option of selecting one of the adjacent spaces to place down a second piece, or ending their turn with just one piece placed by clicking the number "1" at the end.

4. Future Work

The below *Future Work* section details changes and improvements that are in the roadmap for Uni v4 and Uni v5. This includes features in Uni v4 that were not added to Uni v5 like the Dao and Mancala custom GUIs. It also explores possible improvements to combinatorial games within GamesCraftersUWAPI and GamesmanUni and the added custom board string input interface.

4.1. Port Custom GUIs from Uni v4

Many of the custom GUIs from Uni v4 such as the Mancala and Dao Custom GUI mentioned earlier in the paper have not been ported to Uni v5. As a result, these games are present in Uni v5 as either AutoGUI interfaces, which tend to be less clear and accessible than their custom counterparts, or completely text interfaces, which are difficult to interpret and visualize. Porting the GUIs over is crucial to completely finishing the migration from Uni v4 to Uni v5 while maintaining GamesmanUni's mission of accessibility and ease of use.

4.2. New Combinatorial Games

There are many combinatorial games that have not been added to the database. Many combinatorial games have closed form solutions and can be added rather easily, as long as their board strings and moves fit in the AutoGUI system. Other games like *Mock Turtles* and *Dots and Boxes* have not been added because they do not have an easy closed-form solution that fits in the design of GamesmanUni. However, they can still be exhaustively solved for a fixed size rather than for custom board strings and added in this limited manner. Since combinatorial games are newly explored within GamesmanUni and GamesCraftersUWAPI, there are still many games to add.

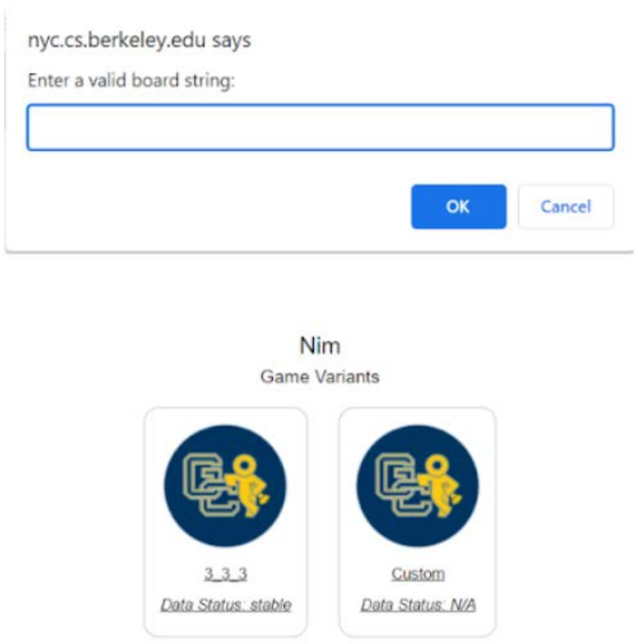
4.3. Combinatorial Games API

With the increased addition of combinatorial games, there are methods that are common to every closed-form combinatorial game that can be implemented. Every closed form combinatorial game must implement a custom start function and a way to get the nimber for a given board string. Adding an API to describe these methods would make it clearer for future contributors of closed-form combinatorial games to ensure they have the proper methods to make their games work with GamesCraftersUWAPI.

4.4. Custom Board String Improvements

4.4.1. Descriptive Custom Input Interface

Currently, the interface for accepting a user input starting board string for customized-start games does not detail the valid format for such strings in each game. As seen below in Figure 28 and Figure 29, the input interfaces for Nim and Dawson's Chess are identical. However, valid input strings for Nim are underscore delimited integers (e.g., 3_3_4) and valid input strings for Dawson's Chess are integers.

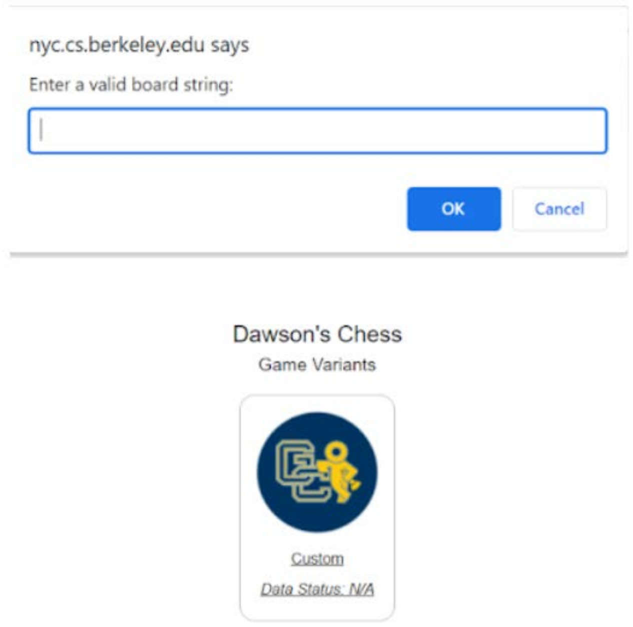

nyc.cs.berkeley.edu says

Enter a valid board string:

Nim

Game Variants

Figure 28: Current interface for Nim custom board string


nyc.cs.berkeley.edu says

Enter a valid board string:

Dawson's Chess

Game Variants

Figure 29: Current interface for Dawson's Chess custom board string

A more detailed description of what entails a valid board string for each game is necessary for ease of use. Future developments could also show a preview of the resulting board from the user's input, or even have a GUI for the selection to make custom games even more accessible.

4.4.2. Custom Board String Validation

Currently, if the custom board string is invalid for the given game, GamesmanUni has no way to validate and reject the input. Instead, it redirects to a completely empty board, as seen in Figure 30. Adding validation for these custom board strings for each game would ensure graceful failures and prevent a confusing empty board.

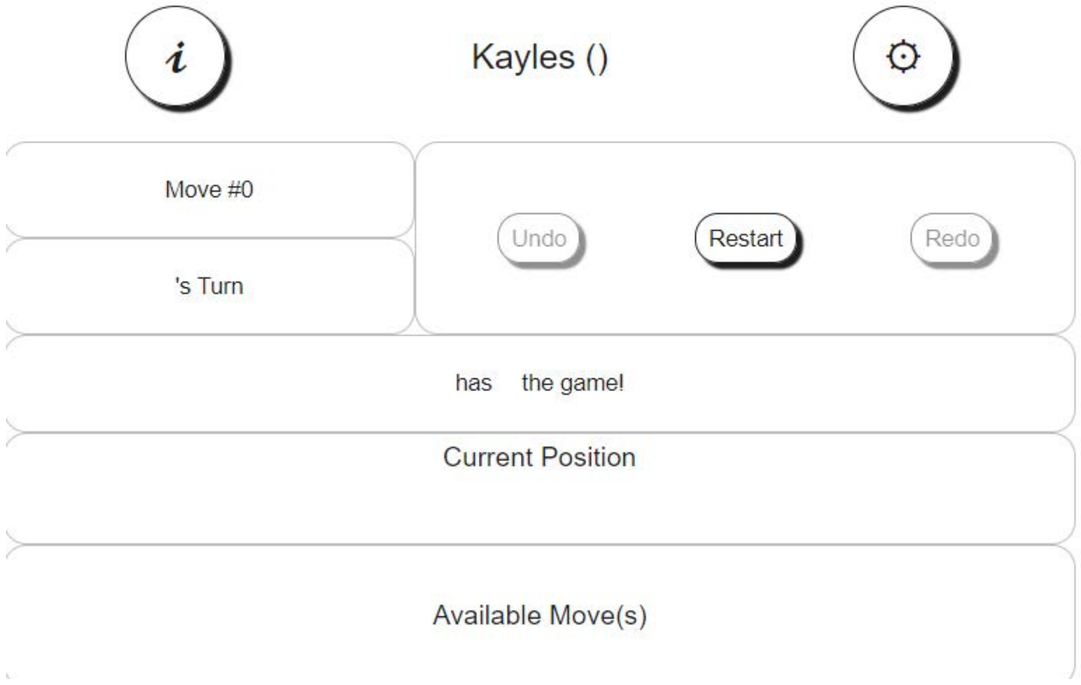


Figure 30: The Kayles board created for the input 'a', which is invalid.

4.5. Documentation

Both GamesmanUni v5 and GamesCraftersUWAPI are not extensively documented – as a result, newcomers to the GamesCrafters research group often find it difficult to contribute without the direct guidance of someone who is already familiar with the codebase. Adding centralized documentation detailing available functions and classes that are already widely used in the current code would facilitate development greatly. This is particularly pertinent for GamesmanUni v5, which has a relatively large codebase compared to many other GamesCrafters projects and is significantly modularized – documentation of the location of each individual component file would help reduce onboarding time.

5. Conclusion

This report describes work done to improve accessibility and ease of use to the current GamesmanUni project, which will in turn improve the general user experience of the website. We improved the GUIs for individual games as well as increased the flexibility of the AutoGUI system. In addition, it details our addition of combinatorial games, which have not previously deeply explored within the GamesCrafters research group or been added to projects like GamesmanUni. We hope that future GamesCrafters members will continue to add more games with both custom GUIs and AutoGUIs, as well as explore the scope of combinatorial games and add more games with closed-forms solutions.

6. References

1. Garcia, D. D. (1990). GAMESMAN [A finite, two-person, perfect-information game generator]. GamesCrafters. https://people.eecs.berkeley.edu/~ddgarcia/software/gamesman/GAMESMAN.pdf
2. GamesmanUni. (n.d.). Retrieved May 12, 2022, from https://nyc.cs.berkeley.edu/uni
3. *GamesmanUni*. Gamesmanuni. (n.d.). Retrieved May 12, 2022, from https://dev01.nyc.berkeley.mintkit.net/uni
4. GamesCraftersUWAPI. (n.d.). Retrieved May 12, 2022, from https://nyc.cs.berkeley.edu/universal/v1/
5. GamesCrafters Wiki. (n.d.). Retrieved May 12, 2022, from https://nyc.cs.berkeley.edu/wiki
6. Wikimedia Foundation. (2021, December 25). *Sprague–Grundy theorem*. Wikipedia. Retrieved May 12, 2022, from https://en.wikipedia.org/wiki/Sprague%E2%80%93Grundy_theorem
7. Ferguson, T. S. (n.d.). *Game Theory*. Retrieved May 12, 2022, from http://www.inf.ufsc.br/~joao.dovicchi/pos-ed/pos/games/comb.pdf
8. Ferguson, T. S. (n.d.). *A note on Dawson's chess - UCLA mathematics*. Retrieved May 12, 2022, from https://www.math.ucla.edu/~tom/papers/unpublished/DawsonChess.pdf
9. *Dawson's Chess*. Dawson's chess. (n.d.). Retrieved May 12, 2022, from https://www.math.ucla.edu/~tom/Games/dawson.html

10. *Sprague-Grundy values for Dawson's Chess*. OEIS. (n.d.). Retrieved May 12, 2022, from https://oeis.org/A002187

11. *Sprague-Grundy values for Kayles*. OEIS. (n.d.). Retrieved May 12, 2022, from https://oeis.org/A002186

12. *Theory of impartial games - MIT*. (n.d.). Retrieved May 12, 2022, from https://web.mit.edu/sp.268/www/nim.pdf

13. Wikimedia Foundation. (2021, April 18). *Mex (mathematics)*. Wikipedia. Retrieved May 12, 2022, from https://en.wikipedia.org/wiki/Mex_(mathematics)

14. *Mancala (Regular)*. GamesmanUni. (n.d.). Retrieved May 12, 2022, from https://dev01.nyc.berkeley.mintkit.net/uni/games/mancala/variants/regular

15. *Dao (Regular)*. GamesmanUni. (n.d.). Retrieved May 12, 2022, from https://dev01.nyc.berkeley.mintkit.net/uni/games/dao/variants/regular

16. Berlekamp, E. R., Conway, J. H., & Guy, R. K. (2001). *Winning ways: For your mathematical plays*. A.K. Peters.

17. Wikimedia Foundation. (2022, April 29). *Nim*. Wikipedia. Retrieved May 12, 2022, from https://en.wikipedia.org/wiki/Nim#/media/File:NimGame.svg

18. Wikimedia Foundation. (2022, May 10). *Kayles*. Wikipedia. Retrieved May 12, 2022, from https://en.wikipedia.org/wiki/Kayles#/media/File:Bowling_ball_and_pins_for_strike_-_front_view.jpg

19. Wikimedia Foundation. (2022, January 16). *Frontend and backend*. Wikipedia. Retrieved May 16, 2022, from https://en.wikipedia.org/wiki/Frontend_and_backend

20. Wikimedia Foundation. (2022, May 7). *API*. Wikipedia. Retrieved May 16, 2022, from https://en.wikipedia.org/wiki/API

21. GamesCrafters. (2020, May 1). *GamesCrafters Universal Web API Regular 2D Board Encoding Format*. GamesCrafters.

22. Wikimedia Foundation. (2022, March 30). *Surreal number*. Wikipedia. Retrieved May 16, 2022, from https://en.wikipedia.org/wiki/Surreal_number

23. Wikimedia Foundation. (2022, May 10). *Kayles*. Wikipedia. Retrieved May 17, 2022, from https://en.wikipedia.org/wiki/Kayles

24. Ho, K. (n.d.). *Mancala*. GamesCrafters. Retrieved May 17, 2022, from http://gamescrafters.berkeley.edu/games.php?game=mancala

25. Garcia, D. (n.d.). *Dao*. GamesCrafters. Retrieved May 17, 2022, from http://gamescrafters.berkeley.edu/games.php?game=dao

7. Appendix

Nim in GamesCraftersUWAPI

```
`import json

from .models import AbstractGameVariant

def nim_custom_start(variant_id):
    try:
        piles = variant_id.split('_')
        for i in range(len(piles)):
            piles[i] = int(piles[i])
    except Exception as err:
        return None
    return NimGameVariant(piles)

class NimGameVariant(AbstractGameVariant):

    piece_char = 'l'

    def __init__(self, start_piles):
        name = "custom"
        desc = "custom"
        status = "stable"
        self.start_piles = start_piles
        self.board_rows = max(self.start_piles)
        self.board_cols = len(self.start_piles)
        super(NimGameVariant, self).__init__(name, desc, status)

    def start_position(self):
        return NimGameVariant.getUWAPIPos(self.board_rows, self.board_cols,
self.start_piles, "A")

    def stat(self, position):
        try:
            position_arr = NimGameVariant.get_position_arr(position,
self.board_rows, self.board_cols)
            position_value = NimGameVariant.position_value(position_arr)
            remoteness = 1
        except Exception as err:
            print(f'Other error occurred: {err}')
        else:
            response = {
                "position": position,
                "positionValue": position_value,
                "remoteness": remoteness,
            }
            return response

    def next_stats(self, position):
        rows = self.board_rows
        cols = self.board_cols
        try:
            position_arr = NimGameVariant.get_position_arr(position, rows, cols)
            player = NimGameVariant.get_player(position)
            moves = NimGameVariant.get_moves(position_arr, rows, cols, player)
        except Exception as err:
            print(f'Other error occurred: {err}')
        else:
            response = [{
                "move": move,
                **self.stat(position)
            } for move, position in moves.items()]
            return response

    def getUWAPIPos(rows, cols, position_arr, player):
        elements = ['R', player, rows, cols]
        board_str = ""
        for i in range(cols):
            num_pieces = position_arr[i]
            next_col = NimGameVariant.piece_char * num_pieces + "-" * (rows -
num_pieces)
            board_str += next_col
        board_str = NimGameVariant.rotateBoardStr(board_str, rows, cols)
        elements.append(board_str)
        return "_".join(map(str, elements))

    def rotateBoardStr(board_str, rows, cols):
        new_board_str = ""
        for i in range(rows):
            for j in range(cols):
                start_index = j * rows
                offset = rows - i - 1
                new_board_str += board_str[start_index + offset]
        return new_board_str

    def get_player(position_str):
        return position_str.split('_')[1]

    def get_board_str(position_str):
        return position_str.split('_')[4]

    def position_value(position_arr):
        value = 0
        for pile in position_arr:
            value = value ^ int(pile)
        if value == 0:
            return "lose"
        return "win"

    def get_position_arr(position_str, rows, cols):
        board_str = NimGameVariant.get_board_str(position_str)
        assert(rows * cols == len(board_str))

        piles = []
        for i in range(cols):
            pile_sum = 0
            for j in range(rows):
                index = i + j * cols
                if board_str[index] == NimGameVariant.piece_char:
                    pile_sum += 1
            piles.append(pile_sum)
        return piles

    def get_moves(position_arr, rows, cols, player):
        move_arr = ["A", NimGameVariant.piece_char, 0]
        moves = {}
        for i in range(len(position_arr)):
            pile_amount = position_arr[i]
            for j in range(pile_amount):
                row_coord = rows - j - 1
                placement = row_coord * cols + i
                move_arr[2] = str(placement)
                move = '_'.join(move_arr)

                next_position = position_arr[:]
                next_position[i] = j
                next_position_str = NimGameVariant.getUWAPIPos(rows, cols,
next_position, NimGameVariant.next_player(player))

                moves[move] = next_position_str
        return moves

    def next_player(player):
        return 'B' if player == 'A' else 'A'
```

Dawson’s Chess in GamesCraftersUWAPI

```python
import json

from .models import AbstractGameVariant

def dawsonschess_custom_start(variant_id):
    try:
        board_len = int(variant_id)
    except Exception as err:
        return None
    return DawsonsChessGameVariant(board_len)

class DawsonsChessGameVariant(AbstractGameVariant):

    def __init__(self, board_len):
        name = "custom"
        desc = "custom"
        status = "stable"
        self.board_str = ''.join(['-' for i in range(board_len)])
        super(DawsonsChessGameVariant, self).__init__(name, desc, status)

    def start_position(self):
        return DawsonsChessGameVariant.getUWAPIPos(1, len(self.board_str), 
self.board_str, "A")

    def stat(self, position):
        try:
            position_str = DawsonsChessGameVariant.get_position_str(position)
            position_value = DawsonsChessGameVariant.position_value(position_str)
            remoteness = 1
        except Exception as err:
            print(f'Other error occurred: {err}')
        else:
            response = {
                "position": position,
                "positionValue": position_value,
                "remoteness": remoteness,
            }
            return response

    def next_stats(self, position):
        try:
            position_str = DawsonsChessGameVariant.get_position_str(position)
            player = DawsonsChessGameVariant.get_player(position)
            moves = DawsonsChessGameVariant.get_moves(position_str, player)
        except Exception as err:
            print(f'Other error occurred: {err}')
        else:
            response = [{
                "move": move,
                **self.stat(position)
            } for move, position in moves.items()]
            return response

    def getUWAPIPos(rows, cols, board_str, player):
        elements = ['R', player, rows, cols, board_str]
        return "_".join(map(str, elements))

    def get_player(position):
        return position.split('_')[1]

    def get_position_str(position):
        return position.split('_')[4]

    def position_value(position):
        value = 0
        pile_lengths = DawsonsChessGameVariant.get_pile_lengths(position)
        for pile_len in pile_lengths:
            pile_mex = DawsonsChessGameVariant.get_mex(pile_len)
            value = value ^ pile_mex
        if value == 0:
            return "lose"
        return "win"

    def get_pile_lengths(position):
        pile_lengths = []
```

```
        curr_pile = 0
        for i in range(len(position)):
            if position[i] == '-':
                curr_pile += 1
            elif position[i] != '-' and curr_pile != 0:
                pile_lengths.append(curr_pile)
                curr_pile = 0
        if curr_pile != 0:
            pile_lengths.append(curr_pile)
        return pile_lengths


    def get_moves(position, player):
        move_arr = ["A", 'x', 0]
        moves = {}
        for i in range(len(position)):
            if position[i] == '-':
                move_arr[2] = str(i)
                move = '_'.join(move_arr)

                next_position = list(position)
                next_position[i] = 'x'
                if i > 0:
                    next_position[i-1] = 'x'
                if i < len(position) - 1:
                    next_position[i+1] = 'x'
                next_position = ''.join(next_position)

                next_position_uwapi = DawsonsChessGameVariant.getUWAPIPos(1,
len(position), next_position, DawsonsChessGameVariant.next_player(player))

                moves[move] = next_position_uwapi
        return moves

    def next_player(player):
        return 'B' if player == 'A' else 'A'

    def get_mex(board_len):
        periodic = [8, 1, 1, 2, 0, 3, 1, 1, 0, 3, 3, 2, 2, 4, 4, 5, 5, 9, 3, 3,
0, 1, 1, 3, 0, 2, 1, 1, 0, 4, 5, 3, 7, 4]
        zero_exceptions = [0, 14, 34]
        two_exceptions = [16, 17, 31, 51]

        if board_len in zero_exceptions:
            return 0
        elif board_len in two_exceptions:
            return 2
        else:
            return periodic[board_len % 34]
```

Kayles in GamesCraftersUWAPI

```
import json

from .models import AbstractGameVariant

def kayles_custom_start(variant_id):
    try:
        board_len = int(variant_id)
    except Exception as err:
        return None
    return KaylesGameVariant(board_len)

class KaylesGameVariant(AbstractGameVariant):

    piece_char = 'l'

    def __init__(self, board_len):
        name = "custom"
        desc = "custom"
        status = "stable"
        self.board_str = ''.join(['-' for i in range(board_len)])
        super(KaylesGameVariant, self).__init__(name, desc, status)

    def start_position(self):
        return KaylesGameVariant.createUWAPIPos(1, len(self.board_str),
self.board_str, "A")

    def stat(self, position):
        try:
            position_str = KaylesGameVariant.get_position_str(position)
            prev_move = KaylesGameVariant.get_prev_move(position)
            player = KaylesGameVariant.get_player(position)
            move_value = None
            if prev_move != None:
                moves = KaylesGameVariant.get_moves(position_str, player,
prev_move)
```

```
                position_value =
KaylesGameVariant.get_position_value_from_moves(moves)
                move_value = position_value
            else:
                position_value = KaylesGameVariant.position_value(position_str)
            remoteness = 1
        except Exception as err:
            print(f'Other error occurred: {err}')
        else:
            response = {
                "position": position,
                "positionValue": position_value,
                "remoteness": remoteness,
                "moveValue": move_value,
            }
            return response

    def next_stats(self, position):
        try:
            position_str = KaylesGameVariant.get_position_str(position)
            prev_move = KaylesGameVariant.get_prev_move(position)
            player = KaylesGameVariant.get_player(position)
            moves = KaylesGameVariant.get_moves(position_str, player, prev_move)
        except Exception as err:
            print(f'Other error occurred: {err}')
        else:
            response = [{
                "move": move,
                **self.stat(position)
            } for move, position in moves.items()]
            return response

    def createUWAPIPos(rows, cols, board_str, player):
        elements = ['R', player, rows, cols, board_str]
        return "_".join(map(str, elements))

    def get_player(position):
        return position.split('_')[1]
```

```
def get_position_str(position):
    return position.split('_')[4]

def get_prev_move(position):
    position_arr = position.split('_')
    if len(position_arr) >= 6:
        prev_move_str = position_arr[5]
        return int(prev_move_str.split('=')[1])
    else:
        return None

def position_value(position):
    value = 0
    pile_lengths = KaylesGameVariant.get_pile_lengths(position)
    for pile_len in pile_lengths:
        pile_mex = KaylesGameVariant.get_mex(pile_len)
        value = value ^ pile_mex
    if value == 0:
        return "lose"
    return "win"

def get_position_value_from_moves(moves):
    for _, position in moves.items():
        pos_value = KaylesGameVariant.position_value(position)
        if pos_value == "lose":
            return "win"
    return "lose"

def get_pile_lengths(position):
    pile_lengths = []
    curr_pile = 0
    for i in range(len(position)):
        if position[i] == '-':
            curr_pile += 1
        elif position[i] != '-' and curr_pile != 0:
            pile_lengths.append(curr_pile)
            curr_pile = 0
    if curr_pile != 0:
        pile_lengths.append(curr_pile)
        return pile_lengths


    def get_moves(position, player, prev_move):
        move_arr = ["A", 'x', 0]
        moves = {}
        # If this is the second part of the multi-part move, you can only pick
the piece adjacent to the first part of the multi-part move
        if prev_move != None:
            next_moves = [prev_move + 1, prev_move - 1]
            for move_idx in next_moves:
                if move_idx >= 0 and move_idx < len(position):
                    if position[move_idx] == '-':
                        move_arr[2] = str(move_idx)
                        move = '_'.join(move_arr)

                        next_position = list(position)
                        next_position[move_idx] = 'x'
                        next_position = ''.join(next_position)[:-1] # Exclude
last character because it isn't used for first part of multi-part
                        next_position_uwapi = KaylesGameVariant.createUWAPIPos(1,
len(next_position), next_position, KaylesGameVariant.next_player(player))

                        moves[move] = next_position_uwapi

            one_pin_only = ['A', '1', str(len(position) - 1)]
            one_pin_move = '_'.join(one_pin_only)
            next_position = position[:-1] # Exclude last character because it
isn't used for first part of multi-part
            next_position_uwapi = KaylesGameVariant.createUWAPIPos(1,
len(next_position), next_position, KaylesGameVariant.next_player(player))
            moves[one_pin_move] = next_position_uwapi

        else:
            for i in range(len(position)):
                if position[i] == '-':
                    move_arr[2] = str(i)
                    move = '_'.join(move_arr)
                    next_position = list(position)
                    next_position[i] = 'x'
                    next_position.append('-')
                    next_position_len = len(next_position)
                    next_position = ''.join(next_position) + '_prevmove=' +
str(i)

                    next_position_uwapi = KaylesGameVariant.createUWAPIPos(1,
next_position_len, next_position, player)

                    moves[move] = next_position_uwapi

        return moves

    def next_player(player):
        return 'B' if player == 'A' else 'A'

    def get_mex(board_len):
        periodic = [4, 1, 2, 8, 1, 4, 7, 2, 1, 8, 2, 7]
        exceptions = {
            0: 0,
            3: 3,
            6: 3,
            9: 4,
            11: 6,
            15: 7,
            18: 3,
            21: 4,
            22: 6,
            28: 5,
            34: 6,
            39: 3,
            57: 4,
            70: 6,
        }
        if board_len in exceptions:
            return exceptions[board_len]
        else:
            return periodic[board_len % len(periodic)]
```