# A COMPUTATIONAL SOLUTION TO THE GAME OF CYCLES

JOCELYN ZONNEFELD, MIKE JANSSEN, AND ELIZA KAUTZ

Abstract. In *Mathematics for Human Flourishing*, Francis E. Su introduced *The Game of Cycles*, a game played on a finite simple planar graph. In game play, opponents alternate adding direction to the edges of the graph with the goal of creating a cycle or making the last legal move. Recent work has sought to determine winning strategies on certain classes of graphs. We introduce a tabular representation of a game state and provide a computer program that determines which player has a winning strategy on any legal game board. The program builds a directed graph of all possible game states, utilizing concepts of impartial game theory in the labeling of game states and determination of winning strategies.

## 1. Introduction

In his seminal volume [9], Francis E. Su introduces a game played on a finite simple graph called *The Game of Cycles*. As it is a finite, impartial, combinatorial game with perfect information and no draws, Zermelo's Theorem states that one of the two players must have a winning strategy [4]. Determining such a strategy for a given class of game boards is the focus of [3, 6, 2, 5], while [7] adapts the game to a class of spider graphs.

The Game of Cycles is played on any planar, connected graph. Two players take turns adding direction to edges by marking arrows on them without creating sinks or sources. A *sink* (alternatively, *source*) is a vertex with all adjacent edges directed towards (away) from itself. An edge is said to be *markable* if adding direction to the edge does not create a sink or a source, and and *unmarkable* otherwise. This can be seen in Figure 1, where the rightmost edge of the game board becomes adjacent to two *almost-sinks* and so becomes unmarkable.

The winner of the Game of Cycles is the player who marks the last available edge or who completes the first cycle (as in Figure 1). Cycles are completed when a player makes a *death move*, meaning that they marked the second-to-last edge of a cell where the next player can complete a cycle and win. For example, the fourth move in Figure 1 is a death move because it left one unmarked edge in a potential cycle cell. Note that a player does not win in the traditional Game of Cycles by creating a cycle around the *outer cell*, or the cell composed of the outer edges of the board.

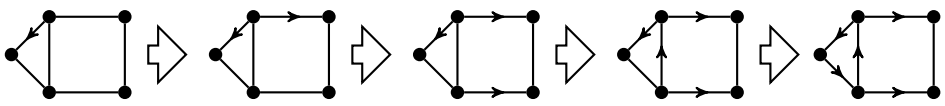

Figure 1. An example of game play in the Game of Cycles.

The primary purpose of this work is to introduce a computational solution to the Game of Cycles. In Section 2, we describe a representation of the game state that we leverage in the Python code posted on the first author's GitHub repository [1]. In Section 3, we describe the code, give examples of its use, and verify its agreement with existing theorems. Finally, in Section 4, we explore the code in more depth to describe how it determines the player with the winning strategy, before concluding with suggestions for future work in Section 5.

## 2. Isomorphisms of Game States

It seems natural to assume that, because game boards are essentially graphs, game boards whose underlying graphs are isomorphic are also isomorphic in the Game of Cycles. In reality, that is not the case. To show this, we will first define one key component that both graphs and game boards share.

**Definition 1.** We define a *face* of a planar graph to be a closed walk where all vertices and edges of the graph not in the walk lie in the exterior of the walk.

Figure 2 gives an example (left) and a non-example (right) of a face.

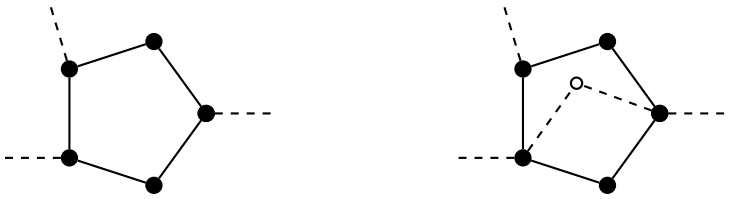

Figure 2. A face and a non-face.

**Definition 2.** Let $G = (V, E)$ be a planar graph. A *game board* (or just *board*) $B$ is $B = (G, F)$ for the set of faces $F$ where each $f \in F$ satisfies $f \subseteq V$.

We observe that when the faces (and thus cycles) of two game boards are different, the games played on the boards will also be different. As we will see, this is true even when the underlying graphs are themselves isomorphic, as they need not have faces preserved by the graph-theoretic isomorphism.

Figure 3 shows two underlying graphs that are isomorphic, but whose respective game boards are not. The reason for this is that the face *abdia* in $B_1$ consists of five vertices, while no face in $B_2$ has five vertices.


Figure 3. Non-isomorphic game boards $B_1$ and $B_2$ with isomorphic underlying graphs.

Definitions 1 and 2 show that, in order for two boards to be isomorphic, we need more than just an isomorphism of the underlying graphs.

**Definition 3.** Let $B_1 = (G_1, F_1)$ and $B_2 = (G_2, F_2)$ be boards. We say $B_1$ and $B_2$ are isomorphic, and write $B_1 \cong B_2$, if there is an isomorphism of graphs $\varphi : G_1 \to G_2$ such that $f \in F_1$ if and only if $\varphi(f) \in F_2$, where, when $f = (v_{i_1}, v_{i_2}, \ldots, v_{i_k}, v_{i_1})$, we define $\varphi(f) = (\varphi(v_{i_1}), \varphi(v_{i_2}), \ldots, \varphi(v_{i_k}), \varphi(v_{i_1}))$.

It is clear that the game boards shown in Figure 4 are isomorphic, while the pair in Figure 3 are not.

FIGURE 4. Isomorphic game boards.

## 3. The computational solution

In this section, we introduce our computational solution to the Game of Cycles. We begin by describing a means of encoding a *game state* so that it can be stored in Python. We then briefly describe our Python code and present some examples generated using the code.

### 3.1. Game Tables and States.

A standard tool in graph theory is the *adjacency matrix*; given a graph $G = (V, E)$ with $n$ vertices, the adjacency matrix $M = (m_{ij})$ is an $n \times n$ matrix which encodes the edges of the graph: if vertices $i$ and $j$ are adjacent, then $m_{ij} = 1$; else, $m_{ij} = 0$. As described above, standard graph-theoretic techniques are insufficient for a full description of an unlabeled game board, so it should not be surprising that an additional tool is needed for describing the state of the game of cycles at any point.

**Definition 4.** Given a game board with $m$ vertices and $n$ faces, we define an $m \times (n+1)$ *initial game table* as follows:

(1) Assign a unique number $1, 2, \ldots, m$ to every vertex on the board.
(2) Assign an indexed variable $f_1, f_2, \ldots, f_n$ to every face on the board; assign $f_{n+1}$ to be the outer face.
(3) Construct an empty table with the faces labeling the columns and the vertices labeling the rows, in ascending order.
(4) For every entry in the table, if the vertex labeling the row is on the non-outer face labeling the column, fill the entry with the *number of the vertex immediately clockwise around the face to the vertex labeling the row.* If the vertex is on the face multiple times, as in Example 5, enter the vertices as a comma-separated list.

    The entries in the last column, corresponding to the outer face, will be filled in the counterclockwise direction. Though players do not win by creating a cycle around the outer face, those edges must be represented in the rows of their respective vertices because they enable our code to detect (and thus avoid creating) sinks and sources. For example, removing the

| | $f_1$ | $f_2$ | $f_3$ | $f_4$ |
|---|---|---|---|---|
| 1 | $+5$ | 0 | $+2$ | $-4$ |
| 2 | 0 | 0 | $+3$ | $-1$ |
| 3 | 0 | 0 | $+4$ | $-2$ |
| 4 | $+1$ | 0 | $-5$ | $-3$ |
| 5 | $+4$ | $-6$ | $-1,-7$ | 0 |
| 6 | 0 | 7 | $+5$ | 0 |
| 7 | 0 | $+5$ | 6 | 0 |

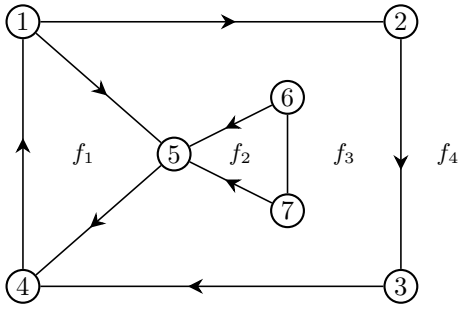


FIGURE 5. A game table describing the state of the game shown.

outer face (and thus the last column of the game table) in Figure 5 would incorrectly identify that vertices 1, 2, and 3 are sources.

(5) All other entries are set to zero.

As the game is played, the initial game table is updated to produce a *game state* representing the board with the arrows drawn at any particular time as follows.

(1) Suppose the edge $\{i, j\}$ lies on the inner face $f_k$ so that $i$ appears before $j$ on a clockwise walk around the face. If we wish to draw an arrow on the edge to point at $j$, we update the sign on the game table entry in row $i$, column $f_k$ to read '$j$'. On the other hand, if we wished our arrow to point at $i$, we update the game table so that the entry in row $i$, column $f_k$ is '$-j$'.

(2) When we have marked an edge $\{i, j\}$ on a face $f_k$, we have also marked the edge on another face $f_\ell$, as each edge lies on two faces (including the outer face), though the direction around $f_\ell$ will be reversed. Thus, if we have marked the entry in row $i$, column $f_k$ as $\pm j$, we will have marked the entry in row $i$, column $f_\ell$ as $\mp j$.

We wish to emphasize that if $\{i, j\}$ is an edge on the face $f_k$, the entry in row $i$, column $f_k$ is always either $j$ or $\pm j$, where the presence of a sign indicate that an arrow has been drawn.

**Example 5.** The example in Figure 5 represents the use of a game table to numerically represent the properties of a board. The table contains information about the vertices, edges, and faces of the board: rows represent vertices, columns represent faces, and paired entries represent edges.

Note that the cycle on $f_1$ is seen in the game table in that the nonzero entries in column $f_1$ all have the same sign.

**Remark 6.** We make the following observations.

- Since the columns of the table represent faces, and the sign of the entries in a column a marked edge in the clockwise (positive sign) or counterclockwise (negative sign) directions, a cycle is formed when all of the nonzero entries in a column are assigned the same sign. (The exception is the last column, which represents the outer face and is not considered a cycle for the sake of winning the game.)
- A sink is formed at a vertex when all entries in the row corresponding to that vertex are given a negative label.
- A source is formed when all entries in the row corresponding to the vertex are given a positive label.

Furthermore, we note that the same board state may be represented with nonidentical tables due to the choice of labeling for vertices and faces. However, we will consider nonidentical tables representing the same game state *equivalent* tables; in this case, one table may be obtained from another via appropriate permutations of rows and columns.

It thus is straightforward to see that game tables are equivalent if and only if the game states they represent are.

3.2. **The Strategy Determination Program.** A major focus of this work is a computational means for determining which of the two players has the winning strategy for a given game board. We do so in the code found in [1] and describe the program below.

Our work is available for download at the URL [1] as a Jupyter notebook. Before describing it in detail, we recall a definition from game theory.

**Definition 7.** We define *impartial game positions* as the two types of game states in the course of an impartial game: a *P-position* places the *p*revious player (who just moved) in a winning position, and an *N-position* places the *n*ext player (who is about to move) in a winning position [8].

The obvious goal is thus to leave the board in a $P$-position following the conclusion of your turn.

By labeling convention, $P$-positions are only directed to $N$-positions, and $N$-positions are directed toward at least one $P$-position. The player with a winning strategy thus always has and must take an available move to a $P$-position, and the losing player only has the option of $N$-positions when given said $P$-position. These properties are ensured by labeling end states as $P$-positions and proceeding backwards through the gameplay.

Our program takes as input a game table, and builds a directed graph of all possible board states as follows:

(1) Determine sets of moves that create sinks/sources or cycles.
(2) Generate all possible boards and remove boards with sets of moves that create sinks/sources.
(3) For every board $B$ without a cycle, add a directed edge to $B$ to produce a board $B'$ (which is $B$ with one additional arrow); do so for all possible moves that create a $B'$ from $B$
(4) Starting at the end of the game, label given board states as $P$-positions or $N$-positions.

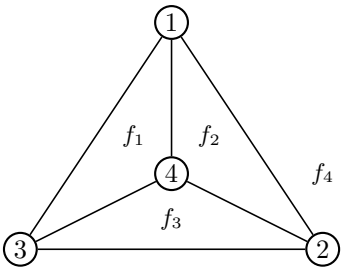


FIGURE 6. The graph $K_4$, with labeled vertices and faces.

The player with the winning strategy is the first player when the initial game state is labeled as an $N$-position; the second player has the winning strategy when the initial position is labeled as a $P$-position.

**Example 8.** We first demonstrate the program on $K_4$ and give a computational verification of [3, Theorem 1], in which the authors prove that Player 2 has a winning strategy. We label the board as seen in Figure 6.

This labeling of $K_4$ produces the initial game table in Table 1.

| | $f_1$ | $f_2$ | $f_3$ | $f_4$ |
|---|---|---|---|---|
| 1 | 4 | 2 | 0 | 3 |
| 2 | 0 | 4 | 3 | 1 |
| 3 | 1 | 0 | 4 | 2 |
| 4 | 3 | 1 | 2 | 0 |

TABLE 1. The initial game table for the $K_4$ board from Figure 6.

This is entered into the second cell of the Jupyter notebook as follows (see Section 4 for more):

```
table = [[[4], [2], [0], [3]],
         [[0], [4], [3], [1]],
         [[1], [0], [4], [2]],
         [[3], [1], [2], [0]]]
```

The Jupyter notebook identifies Player 2 as the player with the winning strategy and produces the digraph seen in Figure 7.

We see the initial game state is labeled with a gold dot, i.e., it is a $P$-position. This means that the player previous to Player 1–that is, Player 2–has the winning position, as proved in [3].

**Example 9.** Another example of interest is that of an odd cycle, which we illustrate with $C_5$. In [3, Theorem 3], the authors note that when playing the game on $C_n$, there is no strategy–players can make any legal move, and the winner depends only on the parity of $n$. Player 1 wins when $n$ is odd, and Player 2 wins when $n$ is even.

Using the labeling of vertices and faces in Figure 8, we produce the game table in Table 2.

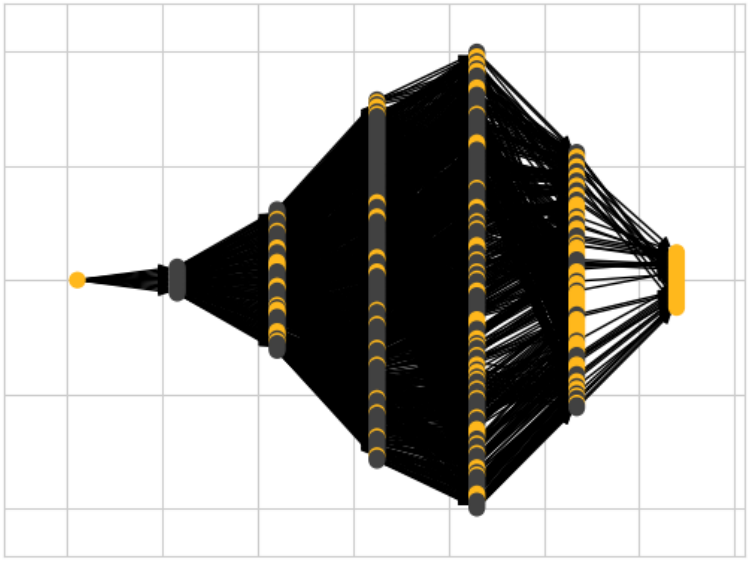

FIGURE 7. The digraph describing possible game states for the Game of Cycles on $K_4$.

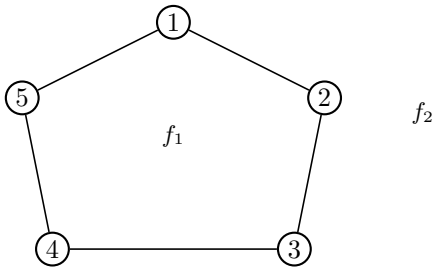


FIGURE 8. The graph $C_5$, with labeled vertices and faces.

| | $f_1$ | $f_2$ |
|---|---|---|
| 1 | 2 | 5 |
| 2 | 3 | 1 |
| 3 | 4 | 2 |
| 4 | 5 | 3 |
| 5 | 1 | 4 |

TABLE 2. The initial game table for $C_5$.

The digraph we produce is found in Figure 9, and shows the lack of strategy required when playing the game on $C_5$ in that each column of possible game states is monochromatic. Regardless of any given move a player makes, the question of

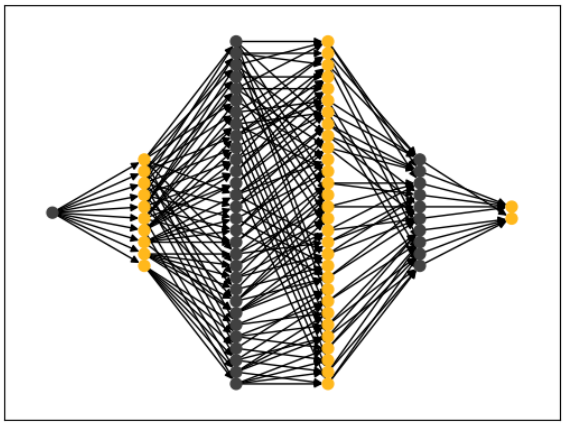

FIGURE 9. The digraph describing the game of cycles on $C_5$.

whether the game is in a $P$- or $N$-position depends only on the parity of $n$ and the number of moves that have been made.

Since the initial position is labeled with a grey dot, it is an $N$-position, meaning that Player 1 has the winning position, as proved in [3].

We encourage the reader to explore their own examples, though an additional Jupyter notebook of examples is available as `Game Table Examples.ipynb` [1].

## 4. Exploring the Program

We now describe our algorithm in additional depth. We present a brief explanation of each cell of code in the file `Cycles Strategy.ipynb` [1].

The first cell imports necessary libraries for code function. The `networkx` library includes a digraph data structure that contains nodes with attributes and directed edges. In our code, the nodes represent game states with an attribute for $P$-position or $N$-position, and the edges represent legal moves from $B$ to $B'$, as described above. The `itertools` library contains a product function that can build an iterable cartesian product of a given length, which will be used to generate all possible game states.

```
import networkx as nx
import itertools as it
```

We next input a game table built using Definition 4. As a table, we naturally require at two dimensions for input into Python. Because game tables such as Figure 5 require comma-separated lists of vertices in a two-dimensional table entry, we introduce a third dimension to the table that assigns each entered vertex a unique index. These unique indices are critical for the program when accessing an individual vertex. Each value in the game table is then decremented given the zero-based indexing in Python, and thus non-adjacent vertices are represented by $-1$ in the program.

```
table = [[[4], [2], [5], [0], [0], [0], [7]],
                [[0], [5], [0], [3], [0], [8], [1]],
```

```
                [[0], [0], [0], [6], [4], [2], [8]],
                [[7], [0], [1], [0], [6], [0], [3]],
                [[0], [1], [6], [2], [0], [0], [0]],
                [[0], [0], [4], [5], [3], [0], [0]],
                [[1], [0], [0], [0], [0], [0], [4]],
                [[0], [0], [0], [0], [0], [3], [2]]]

#Convert table to zero-based numbering
for i in range(len(table)):
    for j in range(len(table[i])):
        for k in range(len(table[i][j])):
            table[i][j][k] -= 1
```

We now compress the game table into one dimension by building a list of the game edges. The mapping represents the pairs of vertices involved in each edge, with the reverse mapping simply switching the key and value in the Python dictionary. In doing so, we represent the game table as a list filled with 0, 1, or $-1$ where 0 represents an unmarked edge, 1 represents an edge from the mapping dictionary key to its value, and $-1$ represents an edge from the reverse mapping key to its value. For each edge in the compressed game table, we also append an empty list to `boards` so that we can sort game states based on quantity of marked edges in the future.

```
mapping = {}
reverseMapping = {}
boards = [[]]

#Get moves from the table by iterating through values in the table
for i in range(len(table)):
    for j in range(len(table[i])):
        for k in range(len(table[i][j])):
            value = table[i][j][k]
            startPoint = (i,j,k)

            #Consider value if it represents an edge
            if value != -1:

                #Find the pair of the initial value
                for u in range(len(table[value])):
                    for v in range(len(table[value][u])):
                        if table[value][u][v] == i:
                            endPoint = (value, u, v)

                #Check if the endPoint already has its own mapping
                if endPoint not in mapping.keys():
                    mapping[startPoint] = endPoint
                    reverseMapping[endPoint] = startPoint
                    boards.append([])
```

Having compressed the game table, we now build lists of all compressed game states that represent illegal game states and cycles. We first create a compressed game state that represents the initial game state using entirely zeroes. For each vertex (row), we build two compressed game states with only the moves that build

a sink or source, appending the game states to the list of illegal states. For each finite face, we build two compressed game states with only the moves that build a clockwise and counterclockwise cycle around that face. The `illegal` and `cycles` lists therefore contain all compressed game states with illegal game states and cycles.

```
#Create lists for illegal boards and cycles, and create an empty board
illegal = []
cycles = []
emptyBoard = []
for i in range(len(mapping)):
    emptyBoard.append(0)

#Get sinks/sources from the table by examining all rows
for i in range(len(table)):
    source = emptyBoard.copy()
    sink = emptyBoard.copy()

    #Set all moves to be the same direction with respect to the vertex
    for j in range(len(table[i])):
        for k in range(len(table[i][j])):
            if table[i][j][k] > -1:

                #Determine parity based on edge orientation in mapping
                if (i,j,k) in mapping.keys():
                    source[list(mapping.keys()).index((i,j,k))] = 1
                    sink[list(mapping.keys()).index((i,j,k))] = -1
                else:
                    source[list(reverseMapping.keys()).index((i,j,k))]
     = -1
                    sink[list(reverseMapping.keys()).index((i,j,k))] =
     1

    #Add to list of sinks and sources
    illegal.append(source)
    illegal.append(sink)

#Get cycles from the table by examining all finite face columns
for j in range(len(table[0])-1):
    clockwise = emptyBoard.copy()
    counterClockwise = emptyBoard.copy()

    #Set all moves to be the same direction with respect to the cell
    for i in range(len(table)):
        for k in range(len(table[i][j])):
            if table[i][j][k] > -1:

                #Determine parity based on edge orientation in mapping
                if (i,j,k) in mapping.keys():
                    clockwise[list(mapping.keys()).index((i,j,k))] = 1
                    counterClockwise[list(mapping.keys()).index((i,j,k
    ))] = -1
                else:
```

```
                    clockwise[list(reverseMapping.keys()).index((i,j,k
    ))] = -1
                    counterClockwise[list(reverseMapping.keys()).index
    ((i,j,k))] = 1

    #Add clockwise and counterclockwise cycles to the list of cycles
    cycles.append(clockwise)
    cycles.append(counterClockwise)
```

After defining a `networkx` digraph, we consider every possible compressed game state by building an iterable Cartesian product of 0, 1, and $-1$ with a length corresponding to the number of edges in the mapping. For each compressed game state, we check if the game state is legal using the list of illegal game boards and check if the game state contains a cycle using the list of cycles. If the board is legal, we add it as a node to the digraph and append it to the list in `boards` corresponding to the number of marked edges. If the board contains a cycle and represents an end state, we label it as a $P$-position.

```
#Consider all possible boards and add legal boards to the digraph
digraph = nx.DiGraph()
for combination in it.product([0,1,-1],repeat = len(mapping)):

    #Check for sinks/sources using list of illegal game states
    legal = True
    for sink in illegal:
        equivalent = True
        for i in range(len(combination)):
            if (sink[i] != 0) and (sink[i] != combination[i]):
                equivalent = False
        if equivalent == True:
            legal = False

    #Check for cycles using list of cycles
    unfinished = True
    for cycle in cycles:
        equivalent = True
        for i in range(len(combination)):
            if (cycle[i] != 0) and (cycle[i] != combination[i]):
                equivalent = False
        if equivalent == True:
            unfinished = False

    #Add legal combinations to boards and digraph
    if legal:
        moves = 0

        #Add to boards list corresponding to number of marked edges
        for move in combination:
            if move != 0:
                moves += 1
        boards[moves].append(combination)

        #Label as an end state if a cycle exists
```

```
        if unfinished:
            digraph.add_node(combination,position = "",layer = moves)
        else:
            digraph.add_node(combination,position = "p",layer = moves)
```

Using the list `boards`, we add edges from non-end states by sequentially marking every unmarked edge in positive and negative directions. If this marking results in a digraph node and therefore a legal board, then an edge is added from the previous to the new game state.

```
#Add edges to digraph by checking every node
for i in range(len(mapping)):
    for board in boards[i]:

        #Only add edges to boards without cycles
        if digraph.nodes[board]["position"] == "":

            #Fill each 0 with 1 or -1 and add edge if new board exists
            for j in range(len(board)):
                if board[j] == 0:
                    positive = list(board)
                    negative = list(board)
                    positive[j] = 1
                    negative[j] = -1
                    if tuple(positive) in digraph.nodes:
                        digraph.add_edge(board,tuple(positive))
                    if tuple(negative) in digraph.nodes:
                        digraph.add_edge(board,tuple(negative))
```

Having built a directed graph of all legal game states, we now label each game state as a $P$-position or $N$-position. Starting with the full game board, we label end states as $P$-positions and step backwards through the gameplay. Game states only directed toward $N$-positions are labeled as $P$-positions, and game states directed toward at least one $P$-position are labeled as $N$-positions.

```
#Starting at the game end, label digraph nodes as N or P positions
for i in range(len(boards)):
    for board in boards[len(boards) - i - 1]:

        #Remove nodes with an in-degree of zero because they cannot be
     reached without the game already ending
        if digraph.in_degree(board) == 0 and i != len(boards) - 1:
            turns = 0
            for entry in board:
                if entry != 0:
                    turns += 1
            digraph.remove_node(board)

        #P-position if an end state
        elif len(digraph.edges(board)) == 0:
            digraph.nodes[board]["position"] = "p"

        else:
            #N-position if directed to a p-position
            for edge in digraph.edges(board):
```

```
                if digraph.nodes[edge[1]]["position"] == "p":
                    digraph.nodes[board]["position"] = "n"

            #P-position if only directed to N-positions
            if digraph.nodes[board]["position"] == "":
                digraph.nodes[board]["position"] = "p"
```

Using the labeled digraph, we now conclude the player with a winning strategy by determine the label of the initial game state. As described previously, the first player has a winning strategy when the initial game state is an $N$-position and the second player has a winning strategy when the initial game state is a $P$-position.

```
if digraph.nodes[tuple(emptyBoard)]["position"] == "n":
    print("Player 1 has a winning strategy")
else:
    print("Player 2 has a winning strategy")
```

The player with a winning strategy can thus use the digraph to achieve a guaranteed victory. The current compressed game state can be inputted into the following cell in place of `emptyBoard`, and the winning player must select an outputted move that results in a $P$-position.

```
for edge in digraph.edges(tuple(emptyBoard)):
    print(edge, "\t", digraph.nodes[edge[1]]["position"])
```

## 5. Future Work

There are three main avenues for future work.

Our first question is in regards to the game table itself (Definition 4). Our initial hope was to follow standard graph theory techniques and utilize the adjacency matrix to describe the game state, as it is well-known to encode several features of the graph, e.g., the way the entries in powers of the adjacency matrix describe the number of walks around the graph. However, as described above, the adjacency matrix is insufficient for describing the full game, as it stores no information about the graph's faces. We wonder if an alternative to the adjacency matrix can be developed with which meaningful linear algebra can be done.

A second direction is to use our code to develop and prove strategy conjectures for additional classes of game boards, such as the cacti in [2].

A final direction is to improve the efficiency of our code. In developing it for use on small graphs, we made few attempts to optimize the code. Possible directions include parallelizing the code, utilizing alternative data structures, or finding a more efficient algorithm.

## References


[1] https://github.com/jocezonn/Strategy-Determination-Program, April 2023.

[2] Samuel Adefiyiju, Heather Baranek, Abigail Daly, Xadia M Goncalves, Mary Leah Karker, Alison LaBarre, and Shanise Walker. Playing games with cacti. *arXiv preprint arXiv:2302.08593*, 2023.

[3] R. Alvarado, M. Averett, B. Gaines, C. Jackson, M. L. Karker, M. A. Marciniak, F. Su, and S. Walker. The game of cycles. *The American Mathematical Monthly*, 128(10):868–887, 2021.

[4] Rabah Amir and Igor Evstigneev. On zermelo's theorem. *Journal of Dynamics  Games*, 4(3), April 2017.

[5] Christopher Barua, Eric Burkholder, Gabriel Fragoso, and Zsuzsanna Szaniszlo. Analyzing a graph theory game. *arXiv preprint arXiv:2208.05147*, 2022.

[6] Robbert Fokkink and Jonathan Zandee. Some remarks on the game of cycles. *arXiv preprint arXiv:2209.14771*, 2022.
[7] Bryant G Mathews. The game of arrows on 3-legged spider graphs. *arXiv preprint arXiv:2110.08738*, 2021.
[8] n.a. Theory of impartial games. http://web.mit.edu/sp.268/www/nim.pdf, 2009.
[9] F. Su and C. Jackson. *Mathematics for Human Flourishing*. Yale University Press, 2020.

Mathematics/Statistics Department, Dordt University, 700 7th St NE, Sioux Center IA, USA 51250

Mathematics/Statistics Department, Dordt University, 700 7th St NE, Sioux Center IA, USA 51250
*Email address*: mike.janssen@dordt.edu

Mathematics/Statistics Department, Dordt University, 700 7th St NE, Sioux Center IA, USA 51250