Representing Incomplete Information in Game Description Language and Monte-Carlo Tree Search

Noah Morris

Hendrix College
Thesis in Computer Science

December 11, 2023

Contents

1 Abstract **4**

2 Introduction **4**

3 Background **7**

3.1 Game Theory 7

3.2 Monte-Carlo Tree Search 8

3.3 General Game Systems 10

4 Ludii's Syntax **13**

4.1 Containers and Components 14

4.2 Integers and Booleans 16

4.3 Defines, Start Rules, and Moves 20

5 Cards **22**

5.1 Stack, Deck, and Card Ludemes 22

5.2 Shuffling and Dealing 23

5.3 Agram Implementation 24

6 AI **27**

6.1 Imperfect Information AI 27

6.2 Information Set Determinization 28

6.3 AI Performance in Card Games 29

7 Conclusion **31**

8 Sources **32**

Dedicated to Dr. David Sutherland, Aunt Dominique, Grandpa Bob, and Mom.

Special thanks to my advisor and the creator of CardStock, Dr. Mark Goadrich, GGS Researcher Tyrone Mason, Dr. Ferrer, Dr. Yorgey, and the Digital Ludeme Project.

1 Abstract

A *deterministic* game has no hidden information or randomness. A *General Game System* is the union of a programming language through which parameters and goals may be specified, and a standardized, dynamic AI capable of engaging in intelligent play in any situation described to it. Most General Game AIs use Monte-Carlo Tree Search to determine moves, and struggle with non-deterministic games.

Ludii is a powerful open-source general game system that was released in 2020. It was written in Java, with a state-of-the-art, highly efficient general AI, and compact, readable syntax, which allows the user to specify information that should be hidden, and to generate random numbers. There's even Ludii syntax for cards, but it has never been utilized, in any of the over 1,500 games implemented for it.

The method presented in this paper acts on a Ludii game's context, essentially filling in all gaps in knowledge with possible outcomes before a move is chosen. This must consistently track which information is legitimate and which is assumption, keeping a base collection of data known as an "Information set". This strategy allows for intelligent, non-clairvoyant play in games of hidden information from MCTS.

I've implemented a medley of non-deterministic games in Ludii, such as Hearts, Golf, 98, and more that are available in the github repository linked below. Also available is an external AI capable of non-clairvoyant MCTS, and some Java files, for the internal AIs.

@ `https://github.com/brimstonetrader/ludiigames`

@ `https://github.com/brimstonetrader/ludiiworkshop`

2 Introduction

Let us consider a time of leisure, and suppose we find ourselves in the company of ≥ 2 other people, and that there is rain. The odds are slim that the Game chosen will be deterministic: likely, there will be randomly drawn cards, hidden from view of the other players. Over the past 25 years, the board game industry has grown massively, with thousands of new games released every year. As such the average modern board game involves many strategic variables, at least one of which is usually a specialized deck of cards. Consider modern classics such as Catan, Ticket to Ride, and Pandemic: three undeniably engaging and strategically dense games, all utilizing chance to some extent. The effect of the inclusion of unknowns into a game is variable: among humans, there exists a sentiment that games of chance or hiding are less difficult to play well than more

“pure” abstract strategy games: where in the latter one’s win is a sheer show of force, some unknowable portion of the credit in one’s win in the former is thought to be due only to fate.

Regardless of whether this sentiment is true, it is certain that games of incomplete information are more difficult for the AI algorithms in Ludii to comprehend. It is also certain that, in life, the vast majority of decisions that an individual must make are based on somewhat incomplete information. Even in billion-dollar Large Language Models, a major weakness is clear: many AI algorithms are essentially limited to the information that they have been given, and are incapable of making reasonable, logical assumptions from what they know, in service of solving open-ended problems. This paper will explore AI methods that combat this issue in games of chance, demonstrate the methods I have used to implement games of chance in a General Game System, and will hopefully encourage others to take this problem on in the many other contexts in which it manifests.

For a reasonable AI performance in a General Game System, it is sufficient to provide a ruleset. All Ludii AIs take in a tokenized version of the game description at outset in order to judge the value of any legal move[9]. Game descriptions can become quite complicated, but below is a sample implementation of Tic-Tac-Toe, which is not.

```
(game "Tic-Tac-Toe"
    (players 2)
    (equipment {
        (board (square 3))
        (piece "O" P1)
        (piece "X" P2)})
    (rules
        (play (move Add (to (sites Empty))))
        (end (if (is Line 3) (result Mover Win)))))
```

Ludii’s syntax, when it released in 2020[11], was a massive improvement on existing game description languages. It takes far less time to write a Ludii description, because the language is very robust, allowing complex statements to be expressed succinctly. This also makes it much easier to discern patterns in game descriptions, and for designers to playtest varying rulesets. The project has hundreds of games available on its website, an incredible, simultaneous feat of computation and anthropology. However, aside from the games and methods I will present in this paper, there are no card games, and no AI support for them.

The games that I have written for Ludii all represent their cards as objects called something like ”`Square1", "Square2", ...` The rank and suit get collapsed into one number, and one or the other can be extracted with integer

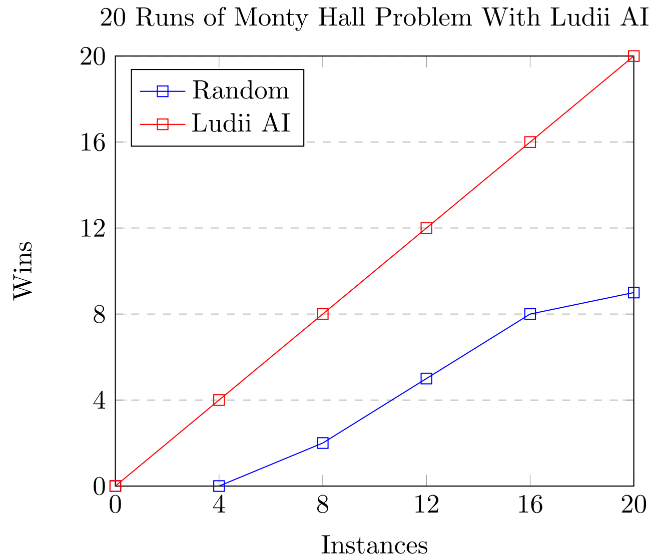
division or modulus. This direct correspondence of cards to integers is useful for my AI method, as it allows a set of cards to be represented as a HashSet of integers. This lets the AI method track what has been seen and not seen quickly, and make reasonable assumptions, regardless of the broader rules of the card game. The general strategy is essentially to enumerate all components in play, to create a discrete set of possibilities that can be mapped to some run of integers, and then do the same thing.

Information can't really be hidden in Ludii. Syntactically, there is a robust suite of aspects of an in-game situation that can be "set Hidden", but doing so does not impact what the AIs are able to see in any way. This is because, on setting something invisible to a particular player, one passes the "context", which is essentially the board position, through a function called "Information-Context" which removes the necessary information. This method, on occurance, only updates the GUI, meaning that the AIs are free to cheat. This method for AI play of games of hidden information, just pretending that there is no hidden information, will be henceforth referred to as "clairvoyant".

The Ludii AI's clairvoyance is demonstrated well by its performance in a very simple game of hidden information. The Monty Hall Problem is a classic, counterintuitive probability puzzle from the '70s, which is as follows. At outset, there exist three doors, behind two of which lie goats (worthless), and one, a car (good). The protagonist (person A) chooses a door, gaining some affinity for it in the process, and then the host (person B) opens one of the two remaining doors, revealing a goat. Thus, of the three doors, one is totally out of the running, one is the current choice, and one remains a possibility. Person B is about to ask if you want to switch.

Monty Hall Problem		
	A chose Car ($P(\frac{1}{3})$)	A Chose Goat ($P(\frac{2}{3})$)
Car	Choice	Switch
Goat 1	Out	Out
Goat 2	Switch	Choice

This means that there is a 2 in 3 probability that you're on the right side of the chart, where switching will get you the car. The Ludii implementation, as a one-player game, attempts to set what is behind the doors hidden: one would expect that an AI would approach a win rate between 1/2 and 2/3. However, I have found that all existing Ludii AIs win 100% of the time, demonstrating their ability to see things that have been "set Hidden". The primary goal of this project is to implement an honest AI that plays demonstrably better than random in arbitrary non-deterministic games. Some secondary goals are to implement card games for said AI to play, diagnose existing problems in representation of hidden information, and to explain the relevant syntax.



3 Background

3.1 Game Theory

Returning to Tic-Tac-Toe, assuming both players play perfectly, the game will always end in a draw. In Connect-4, the first player will win. In general, a *Nash Equilibrium* exists for any finite game (which can be represented as a grid of numbers, or matrix, that approaches a “steady state” of win, lose, or draw on sufficiently many iterations (or moves) like a Markov Chain). This means, basically, that any game that humans play either has an unbeatable strategy for the first player, second player, xor always draws with perfect play. In games with random elements, this equilibrium is called a *mixed strategy*. This is a probability distribution that chooses between pure strategies randomly. A mixed strategy will not assuredly win every time, but will perform better than any other with sufficiently many trials.

In the 1970s, a branch of mathematics called *Combinatoric Game Theory* arose, which analyzes Combinatoric Games as mathematical objects. A crucial algorithm used within this discipline is called *Minimax*. Essentially, this involves assessing each counter-move the opponent could make to each move the active player has available, to determine the best situation the active player could be in next turn, assuming the opponent plays perfectly. This decision rule leads to perfect play if one looks far enough into the future. The main drawback is that the resulting tree of moves is humongous, and impractical to assess in its entirety. This method has not been historically successful at games with large

branching factors, such as Chess and Go, and requires perfect information.

3.2 Monte-Carlo Tree Search

Taking random samples from an outcome space too large to assess traditionally is a classic and somewhat obvious method to draw a conclusion in a situation like this. If I wanted to know the average age of some website's users, I would not ask every user their age, but some representative sample, and get an approximation with a measurable degree of accuracy.

This concept is most commonly applied to Combinatoric Games with an algorithm first described in 2006, called Monte-Carlo Tree Search. We'll suppose there exists a data structure called "GameState", which stores the locations of all pieces and some other consequential information. We'll go more in detail on the general information such a data structure requires in the next section. Further, we'll suppose the existence of a method for GameStates called "getAllPossibleMoves()", which returns an array of possible moves from the position. Once a GameState has been fully played through, we can determine the scores for each player with the method "getScores()", which returns an array of doubles. In a two player game, this will be [1.0, 0.0] if the first player wins, or potentially [0.5, 0.5] if it's a draw. In a game with more than two players, this method will return an appropriate ranking for each player. Putting all of this together, we can determine a good move by checking random playouts for as long as we can.

```
def mcts(int time_allowed, GameState root, int player):

  #initialize data structures
  Move[]                 legalMoves = root.getAllPossibleMoves()
  dict(Move, double) scoresPerMove = new dict(Move, double)
  dict(Move, int)    visitsPerMove = new dict(Move, 1)
  int                numIterations = 0


  while (time_elapsed < time_allowed):

    #select which node we explore this time around.
    GameState leaf   = uct(root, legalMoves, scoresPerMove,
                           visitsPerMove, numIterations)


    #random playout until we reach end-state.
    GameState result = new GameState(leaf)
    while result is not terminal:
      result = result.apply(result.getRandomMove)
```

```
    #update data structures
    double myScore  = getScores(result)[player]
    scoresPerMove[leaf] += myScore
    visitsPerMove[leaf]++
    numIterations++

  #output most explored move.
  return visitsPerMove.keyWithMaxValue()
```

UCT stands for "Upper Confidence Trees", which is a method that chooses which node we will explore in a particular iteration, and ultimately decides the most explored, output move. The core equation assigns a value to each node, denoted S_i. This value comes from the sum of an *exploration* term and an *exploitation* term. The exploitation term for a node n_i is the number of winning playouts divided by the total number of playouts on n_i. This will always be between 0 and 1. The exploration term, for that same node n_i, is the total number of iterations divided by the number of iterations on n_i. This can get much larger than [0,1], so to make the two terms occupy similar ranges we take the logarithm of the total number of iterations, multiply it all by two, and take the square root. These may seem somewhat arbitrary, but they are just convenient mathematical tools to balance the two things we want our method to do.

$$S_i = \frac{w_i}{t_i} + \sqrt{\frac{2ln(N_i)}{t_i}}$$

$S_i \rightarrow$ Heuristic value of node n_i.

$w_i \rightarrow$ Number of wins for node n_i.

$t_i \rightarrow$ Number of playouts run for node n_i.

$N_i \rightarrow$ Total number of considered playouts.

We can define our UCT function, based on the available data for any game, like this:

```
def uct(GameState root,
        Move[] legalMoves,
        dict(Move, double) scoresPerMove,
        dict(Move, int) visitsPerMove,
        int numIterations):

  Move maxMove = null
```

```
double maxValue = -infinity
for (Move move : legalMoves):
  w_i = scoresPerMove[move]
  t_i = min(visitsPerMove[move], 1)
  s_i = (w_i / t_i) + (2 * ln(numIterations) / t_i) ^ 0.5
  if maxValue < s_i:
    maxMove = move
    maxValue = s_i
return root.apply(maxMove)
```

This method is adaptable to games with hidden information. All AIs in Ludii run a variation of this, with the default being named "UCT". This method was the core of AlphaGo, which DeepMind used to defeat Go grandmaster Lee Sedol in 2016, and the algorithm plays very well in many other games with larger branching factors than feasible for Minimax.

3.3 General Game Systems

The first General Game System, *Zillions of Games*, debuted in 1998. Its AI, according to the FAQ on its website, is "basically a classical, brute-force, tree-search engine", which "doesn't play as well in games with huge branching factors such as Shogi"[27]. It cannot handle cards, mancala, and many other common sorts of game.

The first academic exploration of the concept came in 2005, when Stanford released the "Game Description Language", and began a yearly series of competitions for AIs in varied strategy games. This language is still in use today, and an extension of it allows for hidden information and random elements. However, it is very verbose, taking 381[9] keywords to define Tic-Tac-Toe. It is good for representing games to computers, but not for humans. We consider similar broad concepts in many games, such as formation of lines, custodial capture, and the pie rule. A human-friendly GDL would need many predefined methods, to keep up with our capability to utilize common tropes in understanding situations.

CardStock is a General Game System that is particularly designed to render card games. Its syntax allows for representation of Trick-Taking, Adding, Draw-Discard, and many other styles of card game. There are some similarities to Ludii, but enough differences that it warrants examination. There are four base types: Strings (which are all-caps), Integers, Booleans, and Cards. The functions that return each are as follows.

```
String (all caps)
  (cardatt [String] [Card])
Integer (non-negative)
  (+, -, *, //, mod, ^)
  (score [Card] using [PointMap])
  (sum [Collection<Card>] using [PointMap])  (size [Collection<Card>])
  ([Owner] sto [String])  // variable assignment
  (random x y), (random x) //0 implied min
  (tri x)                 //1,3,6,10,15: factorialites
  (fib x)                 //1,1,2,3,5,8: fibonaccis

Card (created at startup, never directly called)
  (top      [List<Card>]) (bottom    [List<Card>])
  ([Integer] [List<Card>])
  (actual (min [List<Card>] using [PointMap]))
  (actual (max [List<Card>] using [PointMap]))
Boolean
  (and [Boolean] [Boolean]+) (or [Boolean] [Boolean]+) (not [Boolean])
  (> [Integer]  [Integer]) (<  [Integer] [Integer])
  (>= [Integer] [Integer]) (<= [Integer] [Integer])
  (== [Integer] [Integer]) (!= [Integer] [Integer])
  (== [String] [String])   (!= [String] [String])
  (== [Card] [Card])       (!= [Card] [Card])
  (== [Team] [Team])       (!= [Team] [Team])
  (== [Player] [Player])   (!= [Player] [Player])

  (all [Collection] [Variable] [Boolean]) (any [Collection] [Variable] [Boolean])
```

`Team` and `Player` are two subtypes of `Owner`. Every card component has an owner: if no player owns it (i.e. it hasn't been drawn yet) it has the third possible owner, `Game`. `PointMap`s are key-value sets that allow for each card to have an integer assigned to it, which can be useful for scoring and specialty decks. Variables can be declared globally, and also used within a single expression.

```
Variable := string w apostrophe first.
            ex. 'C, 'P, 'SUIT

   (declare [Type] [Variable]) // only at beginning of program; global
   (let [Type] [Variable] [Expression]) // anywhere; local

PointMap :=
  (set (game points [String]) Collection<(Card) (Integer)>)
  (using (game points POINTMAP) //to use already defined PointMap

Owners: Game
          (game)
        Player
          ([Integer] player) (current   player)
          (next      player) (previous  player)
          (owner [Card])
        Team
          ([Integer] team) (current   team)
```

```
        (next      team) (previous  team)
        (team  [Player])
```

Multiple of any of the base types can be combined to form a Collection.

```
Collection<String> := (String, String, ...)
  ex. (YELLOW, GREEN, BLUE, BROWN, RED, WHITE)

Collection<Integer> (range [Integer] .. [Integer])

Collection<Card>
  ([Owner] (vloc | iloc | hloc | mem) [String])
      vloc: visible to everyone
      iloc: visible to owner, invisible to others
      hloc: invisible to everyone, including owner
  (filter [CardCollection] [Variable] [Boolean])
  (union [CardCollection] [CardCollection]*)
  (top       [Collection<Collection<Card>>])
  (bottom    [Collection<Collection<Card>>])
  ([Integer] [Collection<Collection<Card>>])

Collection<Collection<Card>>:
  (tuples [Integer] [CardCollection] 'using' [PointMap])
  (all [Collection] [Variable] [CardCollection])

Collection<Player>: (player)       //all players
  (other player)
  (filter [Collection<Player>] [Variable] [Boolean])

Collection<Team> := (team)      //all teams
  (other team)
  (filter [Collection<Team>] [Variable] [Boolean])

Collection<A>:
  (intersection [Collection<A>] [Collection<A>])
  (disjunction  [Collection<A>] [Collection<A>])

Collection<Collection<A>>:
  (partition [Collection<A>])
```

Cardstock and Ludii have many major differences: in Cardstock there is no need for a notion of board position. Another major difference is that in Ludii, components are set hidden. In CardStock, locations are, with visibility for each being set the first time they are mentioned. Having worked with both, I believe CardStock's solution is better. It is more intuitive, when writing a

game description. Also, in Ludii's source code, locations are ultimately what is set hidden, and this incongruence between different levels of representation has led to many difficult-to-understand bugs, in my case.

A game gets set up by instantiating players and cards, and then repeating a sequence of stages, choice blocks, and do blocks until some end condition is met. Lastly, within turns, the gameplay is specified by `Actions`, which can be used to shuffle the deck, move cards, set a new active player, end a turn early, or change variables. The asterisk to the side indicates that the blocks can take any number of arguments, without using a list.

```
Block := (do     ([Action]*))
         (choice ([Action]*))

Setup := (create teams [Integer]*)
         (create players [Integer])
         (create deck [Collection<Card>] [Deck])

Stage := (stage player [Boolean] [Do | Choice | Stage]*)
         (stage team [Boolean] [Do | Choice | Stage]*)

Scoring := (scoring max [Integer])
           (scoring min [Integer])

Action :=
  (shuffle [CardCollection])
  (move [Card] [Card])
  ([Boolean] [Action])
  (turn pass)
  (repeat [Integer] [Action]) (repeat all [MoveAction])
  (inc [IntegerStorage] [Integer]) (dec [IntegerStorage] [Integer])
  (set [IntegerStorage] [Integer])
  (cycle current next) (cycle current previous)
```

Cardstock's syntax has been utilized to implement many card games. It simulates many games with random and MCTS players, gathering heuristics that are useful for game designers, such as 'Fairness' and 'Drama'. The MCTS model does not have Ludii's problem of ignoring hidden information: my new AI for Ludii uses a similar method to allow for this.

4 Ludii's Syntax

In 2020, Ludii was released, with its 24-word description of Tic-Tac-Toe, and much more than GDL's nine predefined concepts. The reference grammar has

over five hundred pages, and the many “Ludemes” allow for games to be represented succintly. The broad structure of a game description is below. Container, Component, Move, and StartMove are all data types which represent part of a game state. They are all fairly high level structures in Ludii, but ultimately, they are made of booleans, lists, enums, and integers, like anything else.

```
(game "Game_Name"
  (players <int>)
  (equipment {
    (<container(s)>)
    (<component(s)>)
  })
  (rules
    (start <startRule(s)>)
    (play
      (move <move(s)>)
      (end <result>)
    )
)
```

One goal of this project is to effectively represent cards in a General Game System. The reason this goal is worth pursuing is that it has not been done in the current industry leading product, Ludii, and the challenges of representing incomplete information to an AI algorithm have many broader applications. Another reason this goal is worth pursuing is that General Game AI, as a discipline, is in its infancy, and has not been properly applied to real-time video games, or modern board games like Catan, Ticket To Ride, or Pandemic. The issues we will work through here are present in those far-off future contexts; this is a stepping stone toward that. Our core method is Monte-Carlo Tree Search: at the end of the day, we will be assessing the tree of options to guess a reasonable move. This demonstrates the inherent connection between AI and linguistics. To search the tree efficiently, our syntax should be as direct as possible, so that we spend very little of our time parsing game descriptions. Before we begin representing cards, we need to thoroughly understand our foundation.

4.1 Containers and Components

Broadly, a container is the board, and a component is a piece. A component is defined like this:

```
(piece <string> <roleType> [<moves>])
    EX. (piece "Dog" P1 (step (to if:(is Empty (to)))))
        (piece "Pawn" Each)
        (piece "Square9" Shared)
```

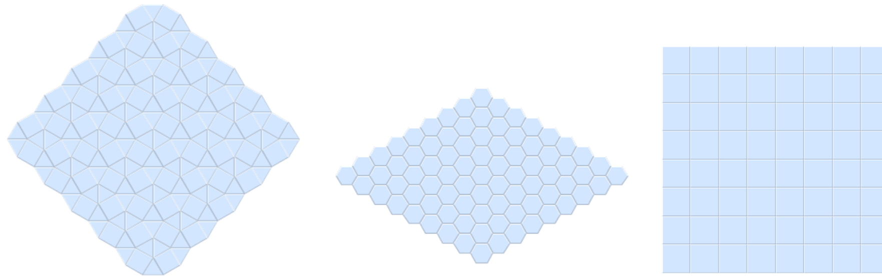


Figure 1: The example boards at the bottom of the page, as rendered by Ludii

Square brackets around an argument means that its presence is optional. In this context, one may define a piece as having certain moves automatically, or may define them later. A "roleType" is a player, or list of players. It can be one of these:

```
<roleType> ::= Ally  | Enemy    | Neutral | Player | Shared |
               Each  | NonMover | Mover   | Next   | Prev   |
               P1    | P2       | P3 ...
               Team1 | Team2    | Team3 ...
```

A "container" can be defined as any arbitrary graph, but the syntax to do so is complicated and beyond the scope of this paper. Consult sources [6] and [11] if that's what you want to do. For our purposes, the board container can be defined as a regular tiling of some size with a set basis. The optional "largeStack" boolean allows the user to determine whether or not stacks higher than 32 units high should be allowed. Do not activate this boolean: it crashes the engine in games of hidden information. It is activated by default when the "Card" Ludeme is instantiated, which is why that Ludeme does not work.

```
<basis> ::= (rectangle <int> <int>) | (square <int>) |
  (tri [<shapeType>] <int> [<int>]) |
  (hex [<shapeType>] <int> [<int>]) |
  (tiling <tilingType> <int> [<int>])
    <tilingType> ::= T31212 | T333333_33434 | T33336 | T33344 |
                     T33434 | T3464 | T3636 | T4612 | T488
    <shapeType>  ::= Diamond | Hexagon | Rectangle | Square |
                     Star | Triangle

<container.board> ::= (board <basis>  [largeStack:<boolean>])
    EX. (board (tiling T33434 5))
        (board (hex Diamond 9))
        (board (square 8))
```

4.2 Integers and Booleans

The integer is the second most basic type in Ludii: a player is an integer, a board location is an integer, and many other things rely upon them. Ludii supports +, *, max, and min for two integers as well as lists of integers. It also supports -, //, ^, and %, for two integers only. There is an abs function, which only takes one integer, and potential for if statements which always return integers, and thus have type integer. Ludii is statically typed, and its syntax resembles Lisp or Haskell, although some variables are mutable.

There are a few keywords which analyze the GameState and return an integer based on it. `(mover)` returns the index of the player that is currently moving, and `(prev), (next)` return the index of the previous or next player to go. Each component has four integer variables, called "who", "state", "what" and "value." "Who" is the player index assigned to a particular component (or 0 if it's Shared or Neutral). For, example, if P1 has a piece at index 17, `(who at:17)` returns 1. The other three variables can be assigned and changed in the rules, and are important when a component has many possible states, as with cards. The user specifies the index of the location where the component is located, and potentially the level if stacks exist. "What" is immutable, set when the component is named at the beginning of the game description. For any component name, "what" of "Name1" is 1, "what" of "Name2" is 2, and so on. State and Value may be set at any point throughout the game. Another variable, "score", can be set for any player, and can be consequential to win-states.

```
(state at:<int> [level:<int>])
(value Piece at:<int> [level:<int>])
(what at:<int> [level:<int>])
(who at:<int> [level:<int>])
(score <roleType>)
```

The user can set standalone integer variables throughout the game, calling them with `(var <string>)`. For accessing an element of an array of integers, `(arrayValue <{int}> index:<int>)` takes the list and index. To get a random integer selected from some range, use `(value Random (range <int> <int>))`. The top level of the stack at some index on the board is `(topLevel at:<int>)` (although, again, be very careful with stacks). Player's hands may be accessed directly, by index, or by the command `(handSite (<roleType>) [<int>])`

A group of integer functions begin with "size", because they measure the size of some configuration. The simplest is `(size Array <{int}>)`, which returns the length of an array. There are also commands to measure the size of a contiguous group, stack, and territory as it's defined in Go. Two very common integer functions that return indexes of locations are `(last To)` and `(last From)`, which return the location that the last player to move went to or

from. Distinct from the list, a useful, immutable data structure definable in the equipment section is `pair`, which allows the user to define a map of key-value pairs at the beginning of the game. It can associate roleTypes, ints, and strings, allowing for particular conversions. I have found it useful in scoring, or creating arbitrary decks.

Next is the group of integer functions prefixed "count". Truthfully, there is not much of a theoretical distinction between "size" and "count", but this is the way this system works. Many of these rely on the `<directions>` enum. Relative to the active player, the user can specify Leftward, Rightward, Forward, and Backward, as well as diagonal combinations, and the three closest directions by suffixing an s. The most common three for game descriptions are Orthogonal, Diagonal, and Adjacent. For example, a chess king can move anywhere Adjacent, and a checker can move to (intersection Diagonal Forwards).

```
              O              O O              OOO
ORTHOGONAL:  OXO  DIAGONAL:   X    ADJACENT:  OXO
              O              O O              OOO

                                               O
LEFTWARD:    OX   BACKWARDS:  X   RIGHTWARDS:  XO
                             OOO               O
```

Arrays of integers may be operated on with traditional set operations, like difference, intersection, and union. They are able to branch with if statements, and regions may be converted to lists of integers with the `(regions ... )` function. An array can also be created from a beginning and endpoint of a range of integers, like `{1..8}`.

For any given set of directions, there are functions to count the number of contiguous groups and liberties (another Go construct), as well as the amount of available locations and neighbors. Some simple, but useful counts are `(count Players)`, `(count Turns)`, and `(count legalMoves)`.

Lastly, in Ludii there are many functions prefixed "sites", which return a particular subset of the board-space. All, ultimately, can be accomplished through `(sites {int})`, specifying board indices directly, but many are useful in practice. One that appeared in an earlier example is `(sites Empty)`, which is often the only place a player is allowed to move. The opposite is `(sites Occupied)`, which is included in full below.

```
(sites {int})
(sites Empty)
(sites Between from:<int> [fromIncluded:<boolean>] to:<int>
                          [toIncluded:<boolean>] [cond:<boolean>])
(sites Hidden [<hiddenData>] (to:<moves.player> | to:<roleType>))
```

```
(sites Random [<sites>] [num:<int>])
(sites Group (at:<int> | from:<sites>) [<direction>] [if:<boolean>])
(sites Occupied (by:<moves.player> | by:<roleType>))
(sites LineOfSight [at:<int>] [<direction>])
(sites [<player> | <roleType>] [<string>])
(sites Hand [<player> | <roleType>])
(difference <sites> <sites>)
(if <boolean> <sites> [<sites>])
(union ({<sites>} | <sites> <sites>))
(intersection ({<sites>} | <sites> <sites>))
```

The set functions, if, union, intersection, and difference, can all be used on regions, for instance, to find a spot in one's hand that is not empty. `sites Between` gets all sites between two on the board. A 'LineOfSight' is a line emanating from some location in some direction through exclusively empty spaces, as defined by Mike Zapawa's "Tumbleweed." A 'Group' is a set of pieces each of which are adjacent, orthogonal, or next to each other via whichever direction is specified, as in Christian Freeling's "Symple."

The most primitive type, the boolean, is rampant in Ludii. The basic, very common comparison operators for two integers are available: ≠, <, ≤, >, ≥, and =. Equals may be used between an int and a roleType, which is useful in contexts where you're using both. The standard functions of two booleans are also available: and, or, and xor. The unary not, and ternary if, are also available. Both 'and' and 'or' may be applied to a list of booleans.

To gauge whether the last player or all players have passed, there exist booleans `(was Pass)` and `(all Passed)`. A boolean also represents whether a given player has any pieces of a given type, as "no Pieces" below. Whether a board location has ever been visited is simply `(is Visited <int>)`. Screening for repeated board positions is done by `(is Repeat Positional)`. `is Hidden` checks whether one of our four integer variables has been set hidden to some player at a particular board location. `Loop` judges if there exists a fully surrounded board territory, and `Blocked` and `Connected` determine whether or not two sites are connected.

```
(no Pieces [<roleType> | of:<int>] [<string>] [in:<sites>])
(is Hidden [<hiddenData>] at:<int> [level:<int>] to:<roleType>)
  <hiddenData> ::= State | Value | What | Who
(is <isPlayerType> (<int> | <roleType>)) |
  <isPlayerType> ::= Active | Enemy | Friend | Mover | Next | Prev
(is Loop [{<roleType>}] [<direction>] [<int>] [<int> | <sites>])
(is Connected [<int>] [at:<int>] [<direction>] {<sites>})
(is Blocked   [<int>] [at:<int>] [<direction>] {<sites>})
(is Line <int> [<absoluteDirection>] [through:<int> |
  throughAny:<sites>] [<roleType>] [exact:<boolean>]
  [contiguous:<boolean>])
```

```
(is Pattern {<stepType>})
  <stepType> ::= F | L | R
```

`Line` takes an integer length (three in the case of Tic-Tac-Toe), and carries a variety of optional, self-explanatory parameters. The `is Pattern` boolean is a bit more general, allowing for recognition of any polyomino as a series of steps. These steps are F (forward), L (left), and R (right). An empty list of steps corresponds to the monomino, one F is a two unit domino. The sequence tracks every unit crossed by following the path on a grid (which generalizes to whatever tiling is specified as the board). If one goes forward, turns left, goes forward again, turns left again, and goes forward one last time on a square grid, they will have traced out a square, regardless what direction they began facing. Because of that last property, patterns work for any rotation of the given polyomino. Below are three pentominoes, represented in this format.

```
              XX
              X                  XX                   XXX
{F F F L F} = X    {F L F L L = XX   {F L F F L F} = X X
              X     F L F R F}   X
```

All end conditions are based on booleans. Most of the time, it's just an if-statement ending in `(result Mover Win)`. The slightly more complex abstract syntax is below.

```
<end>                   ::= (end (<endRule> | {<endRule>}))
  <endRule>             ::=
    <end.forEach>       ::= (forEach Player if:<boolean> <result>)
    <end.if>            ::= (if <boolean> <result>)
      <result>          ::= (result <roleType> <resultType>) | <byScore>
        <resultType>    ::= Draw | Loss | Win
        <byScore>       ::= (byScore [{<end.score>}] [misere:<boolean>])
          <end.score>   ::= (score <roleType> <int>)

EX. (end (if (no Moves Next)
             (byScore {(score P1 (count Pieces P1))
                       (score P2 (count Pieces P2))})))
    (end {(if (no Pieces Mover)        (result Mover Win))
          (if (= (count LegalMoves) 0) (result Mover Loss))})
    (end {(if ("HandEmpty" Mover) (result Mover Win))
          (if (all Passed)        (result Mover Draw))})
```

"byScore" is usually set to the "Score" variable that exists for each player, although it can be any integer. The "misere" boolean decides whether the object is to minimize or maximize.

4.3 Defines, Start Rules, and Moves

Before the game starts, the user may define functions, allowing for refactoring and less repetition. Arguments are dynamically typed, referred to as "#1, #2," and so on. For example, below is a function which finds the amount of bombs Adjacent to the given position, which is used in Minesweeper.

```
(define "Nbors"
    (count Sites
        in:(intersection
            (sites Around #1)
            (sites Occupied by:P1 component:"Bomb"))))
```

At the beginning of the game, there are two groups of commands that can be used to enact the "StartRules": those that begin with "set" and those that begin with "place". The former relate to variables that are not on the board. A player's score can be set, a region can be deemed to belong to a particular player, and any of the four component variables may be set hidden. The final one is very vital to this project, and will be examined further later on. With "place", components may be placed at any specified location, in a stack, or within a region at random. StartRules can be expressed with forEach loops on player roleTypes, lists of integers, particular board locations, and ranges. With that, we can express the full suite of possible startRules.

```
(forEach <ints>                      <startRule>)
(forEach Value min:<int> max:<int>   <startRule>)
(forEach Player                      <startRule>)
(forEach Site <sites> [if:<boolean>] <startRule>)
(place Random <sites> {<string>})
(place Stack items:{<string>} <int>)
(place <string> {<int>} [state:<int>] [value:<int>])
  // fill region with one item
(place <string> <int> [state:<int>] [value:<int>])
  // place one item in one location
(set Score <roleType> <int>)
(set <roleType> [{<int>}] [<sites>] [{<string>}])
(set Hidden <hiddenData> at:<int> [level:<int>] to:<roleType>)
```

After setup is done, a game can be either a sequence of phases with defined transitions, or one large "play" block. In either case, the rest of the game assumes there will be one active player, who begins their turn with one or more "decision moves", and enacts some set of "non-decision moves" afterward in a "then" block. Multiple decision-moves can be expressed with many different operators. The "and" operator means all moves in the list must be done in

a turn. “Or” lets the player choose one. “Priority” means the player must take the first move if it’s possible, and otherwise may take the second, like for compulsory capture in Checkers. There are many possible “ForEach” loops that return moves, as well as regular if and while blocks.

I’ve found 12 decision moves to be all needed for writing and understanding most game descriptions. There are much more available, but the following list is sufficient for most purposes. “Step” allows a piece to move once in a specified direction. “Slide” allows moving any amount of times, like a Chess Queen or Rook. “Shoot” places a new component that emanates from some existing component in a specified direction, as in my favorite game, “Amazons.” “Select” lets the user select a location, which can be operated upon later with the “last From” integer function. “Swap” lets the player choose two locations, which swap components. The empty, default “move” command moves a component from any location to any location. “Promote” lets the player replace a component with another. “Pass” does nothing. “Hop” jumps over some location and potentially captures an opponent piece, like in Checkers, while “Leap” does not capture, like a Chess Knight. “Add” lets the user add some component to a selected board location, and “Remove” does the opposite.

```
<from> ::= (from [<sites> | <int>] [level:<int>] [if:<boolean>])
<to>   ::= (to   [<sites> | <int>] [level:<int>] [if:<boolean>])
<then> ::= (then <nonDecision>)

(move <from> <to> [copy:<boolean>] [stack:<boolean>] [<then>])
(move Add <piece> <to> [stack:<boolean>] [<then>])
(move Remove <int> [level:<int>] [at:<whenType>] [<then>])
  <whenType> ::= EndOfTurn | StartOfTurn
(move Select <from> [<then>]) |
(move Shoot <piece> [<absoluteDirection>] [<to>] [<then>])
(move Slide [<from>] [<direction>] [<to>] [stack:<boolean>] [<then>])
(move Step [<from>] [<direction>] <to> [stack:<boolean>] [<then>])
(move Swap Pieces [<int>] [<int>] [<then>])
(move Promote [<int>] <piece> [<moves.player> | <roleType>] [<then>])
(move Pass [<then>])
(move Hop  [<from>] [<direction>] <moves.to> [stack:<boolean>] [<then>])
(move Leap [<from>] {<stepType>} [rotations:<boolean>] <to> [<then>])
```

Non-decision moves can be chained with use of multiple then statements. There are a couple of commands that can be used as both decisions and non-decisions, based on whether they are preceded by the keyword “move”. Two very common ones are “add” and “remove”. To move a piece from one location to the other as a nondecision, the keyword is instead “fromTo”. A few nondecision moves that cannot be stated as decisions are “addScore”, “moveAgain”, and “custodial”, which captures surrounded pieces as in Pente. Sowing, like in Mancala, is robust, available, and beyond our scope. Lastly, at this stage in a turn, many of the “set” prefixed StartRule ludemes may be applied.

```
(add [<piece>] <to> [stack:<boolean>] [<then>])
(fromTo <from> <to> [copy:<boolean>] [stack:<boolean>] [<then>])
(remove (<int> | <sites>) [level:<int>] [at:<whenType>] [<then>])
(addScore (<player> | <roleType>) <int> [<then>])
(custodial [<from>] [<absoluteDirection>] [<to>] [<then>])
(sow [<int>] [count:<int>] [numPerHole:<int>] ...)
(moveAgain [<then>])
(set Hidden <hiddenData> (at:<int> | <sites>) [level:<int>]
                         (to:<player> | to:<roleType>) [<then>])
(set NextPlayer (<player> | <ints>) [<then>])
(set <setSiteType> at:<int> [level:<int>] <int> [<then>])
     <setSiteType> ::= Count | State | Value
(set Var [<string>] [<int>] [<then>])
```

This is enough Ludii grammar to attempt to implement the first card game in the system.

5 Cards

5.1 Stack, Deck, and Card Ludemes

There exist Ludemes representing Cards, which are classified as Components, and Decks, which are classified as Containers. Alongside them, there exist a whole parallel corpus of specific card-based Ludemes, which are essential in using them. Below is a sample card game, using some Ludemes I did not cover in the previous section.

```
(game "DECK"
  (players 3)
  (equipment {
    (board (square 10))
    (deck {(card Seven rank:0 value:0 trumpRank:0 trumpValue:0) ...
           (card Ace rank:7 value:11 trumpRank:5 trumpValue:11)})
    (hand Each size:2)
  })
  (rules
    (start {(deal Cards 2)})
    (play (move PlayCard))
    (end (if "HandEmpty" (result Mover Win)))
  )
)
```

This code is syntactically correct: Ludii's parsers do not throw any errors

on it. Once it compiles, though, the engine has a glitch attack. If one replaces `PlayCard` with `Pass`, it compiles successfully, but the cards cannot be seen, even when set manually in the graphics metadata. Ludii's developers are aware of this, and plan to rebuild this part of the syntax in the future, but I live in the present.

The stacktrace when one tries to recreate the aforementioned glitch begins with line 2728 in Game.java, where the cards are created. They are generated in a Stack, in some random order. Regardless of whether the 'largeStack' boolean is set, line 579 in ContainerStateStacksLarge.java becomes problematic whenever this stack is placed, taking particular umbrage with the cards being hidden. I have not been successful in getting hidden information to compile in Ludii with the largeStack boolean set at all. Decks, frequently, have more than 31 cards. Dr. Eric Piette, who created Ludii, stated on the website's forum in 2022 that "I do not think we have games with stack + hidden info (since I disabled the card games in Ludii because that was not enough efficient to describe these games)". Stacks, at present, do not work, but we can represent collections in other ways. Let's try a different approach out.

5.2 Shuffling and Dealing

Representing cards in Ludii is somewhat arduous. Each card can behave differently with the integer variables discussed in the previous section. It is tempting and somewhat natural to use two of the available options, say, "state" and "value", to represent the rank and suit of a given card, but in this paper I will only be using the immutable one. I've found that graphics are easier to render when using two, but dealing is infeasible, unless one wants to select with replacement, which I do not.

As an arbitrary choice, the component I will use as my base for all card representations is the "Square". In the equipment section of the game description, one must instantiate each card, as below. This is very similar to what the "Deck" Ludeme attempts to do.

```
(piece "Square1"   Shared), (piece "Square2"   Shared), ...,
(piece "Square51"  Shared), (piece "Square52"  Shared)
```

I've found most card games to be reasonably played on a rectangle with length equal to the number of players, and height of one or two. Each player needs a place to lay down a card, and occasionally a place to store tricks they've won. To assign a specific board location to each player, we can use the `map` ludeme, to associate each player to a place on the board (represented as an integer). Each player needs a hand, and there still should be enough space for each card to be located at a particular index. We want to be able to grab the

Square$x$	$\lfloor x/4 \rfloor = 0$	$\lfloor x/4 \rfloor = 1$	$\lfloor x/4 \rfloor = 2$	...	$\lfloor x/4 \rfloor = 12$
$x \% 4 = 1$	A♥	2♥	3♥		K♥
$x \% 4 = 2$	A♦	2♦	3♦		K♦
$x \% 4 = 3$	A♣	2♣	3♣		K♣
$x \% 4 = 0$	A♠	2♠	3♠		K♠

Table 1: Integer Division and Modulus ⇀ Card

card at a board index without knowing what it is: this will be how we represent drawing a card. This means we'll need to add to the length, to have some places cards can go if they might get drawn later.

To deal, we can use the `place Random` startRule to place cards into each player's hand and off to the cropped-out, invisible-to-the-user-and-AIs part of the board. Before play begins, we can set all of those hidden, except that we must allow each player to view their hand. Because Ludii allows the user to define functions for future use, like many programming languages, I like to put all of this into "SetHiddenEach", which I can copy/paste into whatever game description needs it.

Now, each player has a randomly dealt collection of five integers. We can associate each one to a pair of integers, where each card Square$x$ has rank $\lfloor \frac{x}{4} \rfloor$ and suit $x \% 4$. This is very easy to check for in code, as Ludii supports both operations.

5.3 Agram Implementation

The first card game that I implemented for Ludii is Agram, a four-player trick-taking game that uses a 35 card deck. There are four suits and ranks A, 3-10, with no Ace of Spades. Before play begins, each player is dealt 6 cards. The winner of a hand is the smallest number played, with following suit required if possible. The winner is the player who takes the last trick.

```
(game "Agram"
  (players 4)
  (equipment {
    (board (rectangle 1 15))
    (piece "Square1"  Shared) (piece "Square2"   Shared) ...
    (piece "Square34" Shared) (piece "Square35" Shared)
    (hand Each size:6)
    (map "Table" {(pair P1 1) (pair P2 2)
                  (pair P3 3) (pair P4 4)})})
  (rules
    (start {
```

```
  (place Random (sites (array {5..39}))
    {"Square1"  "Square2"  ... "Square35"})
  (forEach Player
    (set Hidden What (sites (array {5..15})) to:Player))
  "SetHiddenEach"
})
```

Each player has six spaces in their hand, and there are four players, and thirty-five cards total. With a seventeen space board, all 35 cards may be placed randomly either within one of the 24 spaces in a player's hand, or the 11 spaces that come after the playing area. We set all 11 of those hidden to everyone, then set hands hidden appropriately with a function that hides each player's hand from each other.

```
phases:{
  (phase "Lead"
    (play
      (move
        (from (sites Hand Mover))
        (to (mapEntry "Table" (mover)))
        (then (and {
          (set Var "LedSuit"
            (% (what at:(to) level:(topLevel at:(to))) 4))
          (set Score Mover
            (/ (what at:(to) level:(topLevel at:(to))) 4))})))))
        (nextPhase "Follow"))
```

Arithmetic, like all functions in Ludii, uses Polish Notation, like $-\ 9\ 4 = 5$, instead of the more familiar infix notation, $9 - 4 = 5$. When leading, a player may move any card from their hand to their place on the table. They set the suit that must be followed if possible, which is set to a variable. We allow a stack to form of played cards, because they are no longer hidden, and the engine no longer has trouble rendering them. We could remove them from the game altogether, which would look cleaner, but allowing them to linger lets the AIs maintain their memory of played cards easily. The player's score is just the rank of the played card. This phase goes into the next once it has happened once.

```
  (phase "Follow"
    (play
      (priority {
        (move
          (from (sites Hand Mover))
          (to (mapEntry "Loc" (mover))
          if:(= (% (what at:(from)) 4) (var "LedSuit")))
```

```
            (then (set Score Mover
              (/ (what at:(mapEntry "Loc" Mover)
                  level:(topLevel at:(mapEntry "Loc" Mover)))
                4))))
          (move
            (from (sites Hand Mover))
            (to (mapEntry "Loc" (mover)))
            (then (set Score Mover (10))))}))
    (nextPhase (= (% (count Moves) 5) 4) "Trick"))
```

We define following suit in Ludii with the `priority` operator, which takes two moves, and allows the player to make a move of the first kind, unless there are no legal moves of the first kind, in which case the second kind is allowed. In this case, we require the player to play a card of the same suit as our variable defined in "Lead", and set their score as we did before. If this is impossible, the player may play any card, but their score is 10, meaning there is no way for them to win the trick (because the minimum wins). Switching players and assigning the winner of a trick is counted as its own turn, so each cycle of play has five turns total, and we will switch players when the count of turns % 5 = 4.

```
    (phase "Trick"
      (play (move Pass (then "Switch")))
      (end {(if ("HandEmpty" Next) (byScore misere:True))})
      (nextPhase "Lead"))}))

(metadata
  (graphics {
    (piece Foreground "Square1"  image:"A"  fillColour:(colour Hidden)
      edgeColour:(colour Red)    scale:0.7)
    (piece Foreground "Square2"  image:"A"  fillColour:(colour Hidden)
      edgeColour:(colour Yellow) scale:0.7)
    ...
    (piece Foreground "Square35" image:"8"  fillColour:(colour Hidden)
      edgeColour:(colour Blue)   scale:0.7)
    (board Placement scale:5)
    (stackType None)
  })
)
```

Rank and Suit are illustrated by number and color, in this game. Those two variables are easy to set to whatever the user desires. The winner of the trick becomes the new leader, which is defined off-screen in "Switch". If the game is done, the player with the minimum score wins overall. Otherwise, the cycle begins again, with the new leader leading. To get the cards to look appropriate, a metadata section must specify how each looks. I generally just choose one of

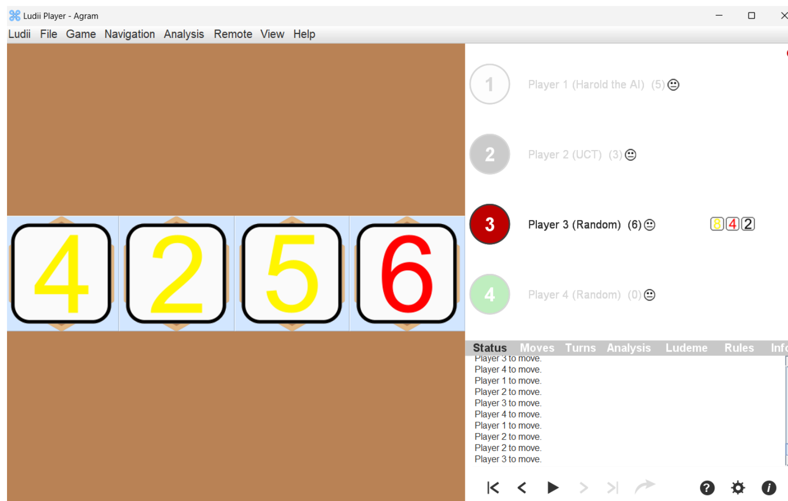


Figure 2: Agram In Progress

‘"A2345678910JQK" to represent the suits, and use red, fuchsia, light blue, and black to represent hearts, diamonds, clubs, and spades respectively.

Along with Agram, I’ve implemented five other games of hidden information in Ludii, and three deterministic ones. Multiple use a standard 52-Card Deck: I would recommend copy-pasting the metadata section and tweaking it to one’s aesthetic preferences, if one wanted to implement another card game in Ludii.

6 AI

6.1 Imperfect Information AI

The tree produced in a game of perfect information branches at decisions, like that for a deterministic game, as well as at *chance nodes*, where information is revealed. In such a tree, a given node cannot be fully evaluated, but we can average across all possible chance nodes to get an expected value. An exhaustive search of all possibilities would return a Nash Equilibrium, and a good approximation can be found from a random partial search, like with deterministic games.

In *Artificial Intelligence: A Modern Approach*, authors Russell and Norvig show that a brute force Tree Search can fully solve a game of imperfect information in $O(b^m n^m)$ time, where b is the amount of legal moves, m is the maximum amount of moves in a game, and n is the amount of possible outcomes from the random event, in this case, the number of cards. This additional exponential

factor makes games of chance particularly difficult for classical game-playing algorithms, such as Alpha-Beta Search. Monte-Carlo Tree Search, on the other hand, does not try to fully explore the tree, and can search a random permutation of cards for each iteration (a process called *determinization*). The first practical implementation of this style of algorithm came in 2001, with Matthew L. Ginsberg's "GIB Player", which played bridge with "roughly equivalent strength to human experts"[21].

There are two possible methods for determination. The "lazy" method is to assign each chance event as it occurs, and ignore all unknown information. This can be effective in single-player puzzles[21], but generally falls short in multiplayer card games. The standard method is to establish every possible unit of information randomly at outset, and to proceed as if it is a game of perfect information. This strategy, in addition to Bridge, has been utilized in Cardstock, and effectively utilized in many stochastic games, including Phantom Go and Magic: The Gathering.

6.2 Information Set Determinization

We need to make a few tweaks to our MCTS pseudocode to accommodate hidden information in the above way. To start with, we need a function that assembles a set of all components that are currently hidden, then replaces each with a randomly selected piece.

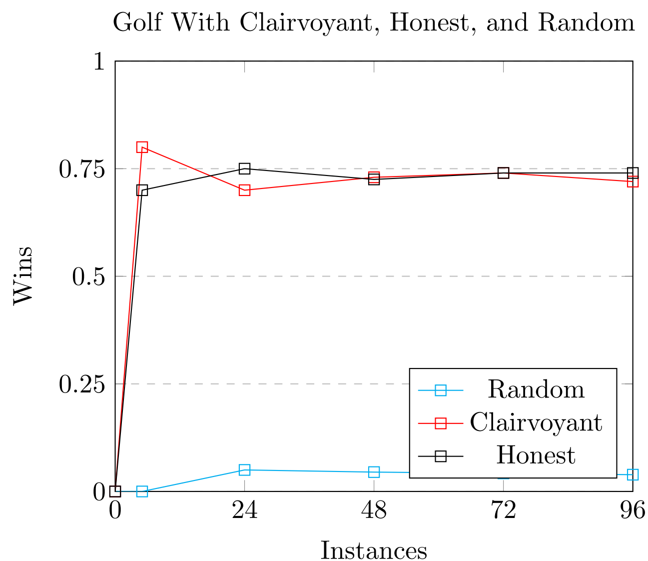
```
def determinize(GameState game):
  is = HashSet(game.pieces())
  for (location l in game.boardLocations()):
    if (!game.isSupposedToBeHidden(l) or game.isEmpty(l)):
      is.remove(l)
      if (!game.isEmpty(l)):
        ps[pieceAt(l)] = null
  for (location l in game.boardLocations()):
    if (game.isSupposedToBeHidden(l)):
      p = is.popRandom()
      game.removePieceAt(l)
      game.placePieceAt(p, l)
```

This method must only be called on the initial move, after the UCT search makes its decision, but prior to the playout. There are many undefined functions above, to the ill-defined "GameState". These functions are all possible (although often by different names) operations on the overarching Game data structure in Ludii, which we explored in the previous chapter, although they follow a much different and more complex hierarchy of instructions. This loops through locations, not pieces, has no effect on components that are not supposed to be

hidden, and guarantees a possible outcome. There are some particular cases where this approach is suboptimal, such as the Monty Hall Problem. It doesn't factor in how likely each option is, assuming a uniform distribution, when in that case one option has twice the probability of the other. With only two options, and a very weighted die, not having that knowledge of dependent probabilities can be difficult. However, most card games have a far larger branching factor than the Monty Hall Problem in both chance and choice nodes, and this much larger outcome space allows this uniform-probabilities assumption to work well in practice. The full method, written in Java, is available at `MCTSStoHi.java` in the attached Github repository.

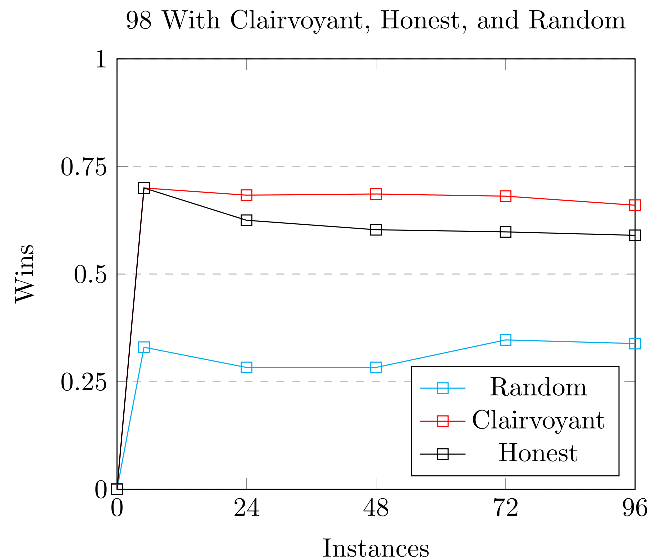
6.3 AI Performance in Card Games

For each of my games, I ran 96 simulations. 96 is divisible by 4!, so this balances for n'th player advantage in four player games in a way that 100 games couldn't. In each, "Clairvoyant" is the standard Ludii UCT AI, and "Honest" is the agent linked in the github repository. "CardStock" is the default MCTS agent used in CardStock. All AIs were given one second of thinking time per round. The values presented are the average value assigned to each player on average in a game. All have three or more players, so these values don't sum to one, but their relative sizes demonstrate how well the various AIs did over time.

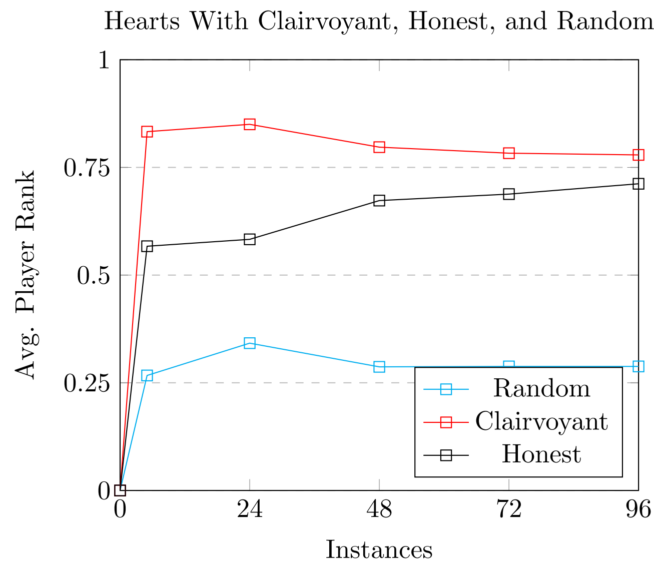


Golf is the simplest game tested. You draw and then discard a card each turn, and whoever has the lowest score wins. The two Monte Carlo Tree Searches are basically equivalent here, both discarding the highest possible card on each turn, but this is a good barometer. Ultimately, the Honest method won by a

hair, but the margin is slim enough that we can safely conclude the two play similarly in very simple games of chance.



98 poses an interesting problem. Each turn, you add the value of the played card to a grand total, with kings setting the value to 98, 10s having value -10, and Queens/Jacks having value zero. The first player to bring the score above 98 loses: there is no direct winner. This means our heuristic has much less to go off of: nevertheless, the deterministic model played well, competitively with the clairvoyant Ludii AI.



	Honest	Clairvoyant	Random
Agram	0.49	0.69	0.41
Golf	0.74	0.72	0.04
98	0.59	0.66	0.41
Hearts	0.71	0.78	0.29
Bottle Imp	0.72	0.80	0.25

Table 2: MCTS Competition

Hearts is a complex and popular trick-taking game with a standard 52-card deck. Each of the 13 Hearts is worth -1 point, and the Queen of Spades is worth -13 points. The winner is the player with the highest score, unless someone has amassed all 26 points, in which case they win. This game's graphics section has my representation of a deck of cards in Ludii, which I recommend to anyone who needs one, as writing it by hand was arduous. This is the most complicated game that I implemented, the only one that I've played frequently with non-computer scientists. I was expecting the added complexity to hurt the honest MCTS's chances, but the method absolutely played competitively.

In general, it appears that a deeper and more complex strategy improves the Honest MCTS's chances, likely because that is such a disservice to a random player. While this model doesn't beat the Clairvoyant model dramatically in any context, it played well and honestly, which means that such a thing in a GGS is possible.

7 Conclusion

Games with incomplete information can exist in a General Game System, and the Monte-Carlo Tree Search algorithm which allows for General Game AIs can make strategic moves without clairvoyance, in a way that is competitive with clairvoyant players. I have done this for "Ludii", a General Game System released in 2020, and demonstrated that this happens in "CardStock", a General Card Game System. Over the course of this paper, we have learned a great deal about how games are defined, how AIs use trees, and context-free grammar.

AIs must grapple with incomplete information any time they take on a remotely realistic problem, and usually the option to cheat will not exist. This style of model is much more capable in more situations, and Information-Set Determinization is a useful tool for any Stochastic AI that sets out to solve arbitrary problems.

8 Sources

1. http://ludeme.eu/index.html

2. "Go master Lee says he quits unable to win over AI Go players", Yoo Cheong-mo, November 27 2019, https://en.yna.co.kr/view/AEN20191127004800315

3. "Ludii Games", Noah Morris, July 21 2023, https://github.com/brimstonetrader/LudiiGames

4. "Winning Ways for your Mathematical Plays", John Conway & Elwyn Berlekamp & Richard Guy, 1982, https://archive.org/details/winningwaysforyo02berl

5. "Non-Cooperative Games", John Nash, 1951, https://web.archive.org/web/20150420144847/http://www.princeton.edu/mudd/news/faq/topics/Non-Cooperative_Games_Nash.pdf

6. "Ludii Game Logic Guide", Eric Piette & Cameron Browne & Dennis Soemers, https://ludii.games/downloads/LudiiGameLogicGuide.pdf

7. "RECYCLEd CardStock", Mark Goadrich, 2018, https://cardstock.readthedocs.io/en/latest/index.html

8. "A Practical Introduction to the Ludii General Game System", Eric Piette & Cameron Browne & Dennis Soemers & Matthew Stephenson, 2019

9. "Ludii - The Ludemic General Game System", Eric Piette & Cameron Browne & Dennis Soemers & Matthew Stephenson & Chiara F. Sironi & Mark H. M. Winands, 2020

10. "General Board Game Concepts", Eric Piette & Cameron Browne & Dennis Soemers & Matthew Stephenson, 2021

11. "Ludii Language Reference", Eric Piette & Cameron Browne & Dennis Soemers, May 16 2023, https://ludii.games/downloads/LudiiLanguageReference.pdf

12. "Ludii and XCSP: Playing and Solving Logic Puzzles", Eric Piette & Cameron Browne & Dennis Soemers & Matthew Stephenson, August 23 2019, https://cris.maastrichtuniversity.nl/ws/portalfiles/portal/123479428/Soemers-2019-Ludii-and-XCSP-Playing-and.pdf

13. "CardStock", mgoadrich, https://github.com/mgoadric/cardstock

14. "GGP in Ludii", brayo303, https://github.com/brayo303/GGPinLudii

15. "Ludii AI Dev", z5164964, https://github.com/z5164964/LudiiAIDev

16. "Ludii Stuff", mgrider, https://github.com/mgrider/ludii-stuff/ tree/-main

17. “Final Year Project”, schererl, https://github.com/schererl/FinalYearProject/tree/main/src

18. “Ludii Example AI”, Ludeme, https://github.com/Ludeme/LudiiExampleAI

19. “Improving Solvability for Procedurally Generated Challenges”, Mark Goadrich & James Droscha, May 26 2019, https://arxiv.org/pdf/1810.01926.pdf

20. “Understanding the Success of Perfect Information Monte Carlo Sampling in Game Tree Search”, Jeffrey Long & Nathan R. Sturtevant & Michael Buro & Timothy Furtak, https://webdocs.cs.ualberta.ca/ nathanst/papers/pimc.pdf

21. “Determinization and information set Monte Carlo Tree Search for the card game Dou Di Zhu”, Daniel Whitehouse & Peter I. Cowling & Edward J. Powley, https://www.researchgate.net/publication/224259865

22. “On Numbers and Games”, John Conway, https://senseis.xmp.net/?OnNumbersAndGames

23. “Imperfect-Information Game AI Agent Based on Reinforcement Learning Using Tree Search and a Deep Neural Network”, Xin Ouyang & Ting Zhou, https://www.mdpi.com/2079-9292/12/11/2453

24. “Hidden Information General Game Playing with Deep Learning and Search”, Zachary Partridge & Michael Thielscher, https://cgi.cse.unsw.edu.au/ mit/-Papers/PRICAI22.pdf

25. “General Game Playing with Imperfect Information”, Michael Schofield & Michael Thielscher, https://www.jair.org/index.php/jair/article/download/11844/26543/22599

26. “Monte Carlo Tree Search for games with Hidden Information and Uncertainty”, Daniel Whitehouse, https://core.ac.uk/download/pdf/30267707.pdf

27. https://www.zillions-of-games.com/supportedFAQ.html

28. https://web.archive.org/web/20220323054404/http://logic.stanford.edu/classes/cs227/2013/readings/