

Report: Project Ur Phase 1

Project Leader, Author, Programmer, Designer: Dun-Ming Brandon Huang

Table of Contents

- 0. Abstract
- 1. Introduction
 - 1.1: Royal Game of Ur
 - 1.2: Artificial Player and Its Place in This Project
 - 1.3: Objectives of The Project
- 2. Implementation of Game
 - 2.1: Aspects of Game Board
 - A. Cell Module
 - B. Pieces Module
 - C. Board Module
 - 2.2: Player and Artificial Player
 - 2.3: Game Module
- 3. Strategies of Artificial Players
 - 3.1: Strategy Masugu
 - 3.2: Strategy Ashikazu
- 4. Experiment on the Efficiency of Strategies
 - 4.1: Hypothesis
 - 4.2: Procedure
 - 4.3: Observations
 - 4.4: Analysis
- 5. Experiment on the Effectiveness of Strategies
 - 5.1: Hypothesis
 - 5.2: Procedure
 - 5.3: Observations
 - 5.4: Analysis
- 6. Conclusion
 - 6.1: Conclusion of Experiment
- 7. Developments for Upcoming Phases
 - 7.1: Developing Strategies of Artificial Players
 - 7.2: Developing an Interactive Interface
- 8. Work Cited

0. Abstract

Lorem Ipsum

1. Introduction

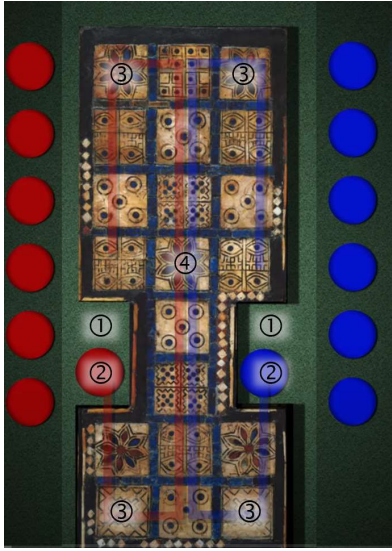
This project is about building a complete electronic version of “Royal Game of Ur”, an ancient board game, that shall involve artificial players as well as an interactable interface for players, and several other functions. This project started out as a summer project to raise my programming sense and skill. A draft of its contents was made in April, when my programming experience involved senior year volunteer programming works and the AP Computer Science A curriculum.

Phase 1 of this project started mid-late June when I flew back to Taiwan, during part of second week of quarantine, after having finished curriculum of CS61A until Midterm 2. In that time, the implementation of first artificial players’ strategies and means to simulate games has been completed, and in the following section introductions to basic concepts employed in the project is provided.

1.1: Royal Game of Ur

The Royal Game of Ur is a board game originating from ancient Mesopotamia, dating about 2400 BCE and named so for its discovery from royal cemeteries of Ur, another state of the Mesopotamian civilization. This board game was also used for gambling when it was popular in its origin. It is an indeterministic game.

The basic setting of the game follows:



On the left side is the board for the Royal Game of Ur.

The Royal Game of Ur is a race game, meaning the first player to have assembled a number of pieces at one destination will win the game.

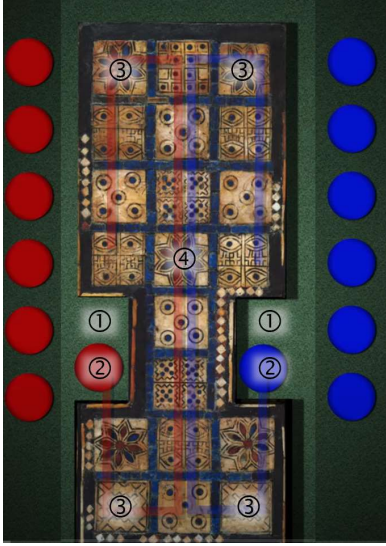
Each players have seven moveable pieces, and each players’ pieces have to start on the Start position (position 1) and reach the End position (position 2). The first player to assemble seven pieces on the End position of that player’s side wins the game.

The amount of movement by each piece are decided by four tetrahedral, fair die that will be introduced later in this section.

The above picture comes from the British Museum’s video introduction on YouTube of this board game.

Following are rules for pieces’ movements along the board for specifically Royal Games of Ur:

Rule 1: Cells on the board must not contain more than one piece.



Rule 2: The left player's pieces' movements follow the red path, while the right player's the blue.

Rule 3: When a piece attempts to travel to an occupied cell:

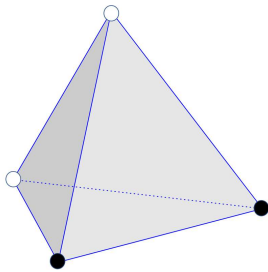
- If the occupying piece is of the same side as the traveling piece, the traveling piece cannot move to the occupied cell.
- If the occupying piece is not of the same side as the traveling piece, the traveling piece may move to the occupied cell, and if so, the occupying piece must return to the Start position of its side.

Rule 4: If a piece travels to a cell with a Rosetta (positions 3, 4), the owner of that piece immediately gains one extra turn.

Rule 5: Position 4 is a peaceful Rosetta. While it contains the effect of a Rosetta cell, any cell located on position 4 will not be deported to its start. This means it is an exception to Rule 3.

Rule 6: A piece will reach End position with no more than the exact needed number of needed step.

Royal Game of Ur is a turn-based game. Each tetrahedral dice has two painted, or white vertices, and two unpainted, or black vertices. At each player's turn or their encounter with Rosetta cells, they roll four tetrahedral dices, and depending on the amount of white vertices rolled, they move the piece for that amount of steps according to the movement rules and general settings of the game addressed above.



By the fairness of each tetrahedral dice, the probability of rolling a white vertex on each of the four tetrahedral dice is $\frac{1}{2}$. We may then deduce:

1. The minimum steps available is zero, while the maximum steps available is four.
2. The probabilities of rolling each number of steps may be calculated using binomial distribution.

Binomial Distribution states that for an event that will either success or fail, like flipping a coin which only results in two possible conclusions, the probability P of an event with probability p of success having exactly n successes out of N trials (meaning exactly $N - n$ failures) can be calculated as follows:

$$P = \binom{N}{n} p^n (1 - p)^{N-n}$$

Using Binomial Distribution, we may therefore conclude the probabilities for rolling each possible number of steps:

Number of Steps	Probability to Roll Number of Steps as Left
0	$P(0) = \binom{4}{0} \left(\frac{1}{2}\right)^0 \left(1 - \frac{1}{2}\right)^{4-0} = \frac{1}{16}$
1	$P(1) = \binom{4}{1} \left(\frac{1}{2}\right)^1 \left(1 - \frac{1}{2}\right)^{4-1} = \frac{1}{4}$
2	$P(2) = \binom{4}{2} \left(\frac{1}{2}\right)^2 \left(1 - \frac{1}{2}\right)^{4-2} = \frac{3}{8}$
3	$P(3) = \binom{4}{3} \left(\frac{1}{2}\right)^3 \left(1 - \frac{1}{2}\right)^{4-3} = \frac{1}{4}$
4	$P(4) = \binom{4}{4} \left(\frac{1}{2}\right)^4 \left(1 - \frac{1}{2}\right)^{4-4} = \frac{1}{16}$

Below is a summary for the procedure of gameplay:

1. Player 1 rolls four tetrahedral dice and move an available piece from its current position or start according to the number of steps rolled.
2. If piece reaches a Rosetta cell, repeat step 1.
3. Player 2 rolls four tetrahedral dice and move an available piece from its current position or start according to the number of steps rolled.
4. If piece reaches a Rosetta cell, repeat step 3.
5. Until a player wins the game, repeat step 1 through step 4.

For this project specifically, a turn is defined as the period of time a player is moving pieces on the board until this candidate alternates.

1.2: Artificial Player and Its Role in This Project

In this report, an Artificial Player is defined as a non-human player who follows a specific coded strategy that does not involve machine learning.

Artificial Players are also what we call “bots” in a majority of video games, in essence Artificial Players are computers that play against humans. Artificial Players are employed in this project instead of Artificial Intelligence for the following reasons:

1. The programmer’s incapability to employ machine learning algorithms given current scope of knowledge in computer science.
2. The difficulty for machine learning algorithms to find advanced strategies in an indeterministic game.

The implementation of Artificial Player is expected to be the most difficult aspect of this project, due to the lack of knowledge in decision-making algorithms from programmer. Even if so, Artificial Player proves to be a significant aspect of project for the experience in relevant knowledge it will provide. It will also be significant in offering a single player mode for the electronic version of Royal Game of Ur.

Further explanation of the composition for the Artificial Player will be found in Section 2.

1.3: Objectives of the Project

Here follows the objectives of Project Ur.

1. To enhance the programmer's ability and sense to implement different features of the project, such as game simulation, gameboard settings, artificial players, and interactable interface.
2. To implement decision-making algorithms and utilize mathematics and game-relevant knowledge to optimize strategies for the artificial player.
3. To investigate on the possibility of practicing a machine learning algorithm on playing Royal Game of Ur.
4. To investigate the quality of different decision-making algorithms with different util functions in the project by comparing them with other artificial strategies and in-game behaviors from human players.
5. To experience and learn the aspects of implementing an interactable interface, and file organization for constructing a graphic user interface.

2. Implementation of Game

This implementation by now is incomplete, because Phase I of the project has not reached the implementation of interface. Soon there should be modules containing interface-wise codes.

The implementation of this Game is accomplished via Object Oriented Programming. There are simpler ways to implement the game without using Object Oriented Programming to represent the entire gameboard. However, being a much familiar choice to the programmer and easier approach to comprehend and practice when imagining and implementing strategies, OOP becomes the adopted method of this project.

The aspects of game to be implemented are classified into the following categories: Cell, Piece, Board, Player, Game. Each of these categories have a module named after it to contain relevant implementations. The Game module plays a special role in the implementation to integrate the four other modules in the list to realize in-game-operation mechanics, including player strategy.

Below we will introduce the purpose of these five modules and major aspects of their implementations.

2-1: Aspects of the Gameboard

The aspects of gameboard will involve Cell, Piece, and Board. These three modules are connected to each other as aspects of the gameboard, and inside these modules may contain multiple classes.

In the following sections, Cell and Piece are two easier and simpler modules, both containing only one class to represent the object they are assigned responsible to. Meanwhile, Board is a much complicated module containing thinking under the carryout of strategy and simulation, and methods of in-game operation will be much clearer after the introduction of this module.

2-1-A: Cell

A cell is any square space on the board that can contain pieces. In this implementation, the Start and End position of the board are also programmed as cells, with slightly different properties from the other given they may contain multiple pieces.

Each cell initiated will be designated to one location on a Board object, which is involved in a module that will be discussed in later section.

The Cell module contains a class Cell, representing a cell on the gameboard.

Upon initiation a Cell will acquire following properties:

Property	Purpose of Property
typeChart	A dictionary whose keys are the position of a cell and values are types of cells corresponding to that position. This is a class variable.
row	The row on board at which cell is located on. It is an integer between range from 0 to 7, inclusive. The coordinate system of Cell will be explained later in this section.
column	The column on board at which cell is located on. It is either “L”, “M”, or “R”.

position	The position of the cell on board, represented by an alphanumeric string with column first, row number second. For example, L7, R0, M4.
contain	A list involving the pieces this cell contains. It is an empty list by default.
isCopy	A Boolean value indicating whether this cell is copied. It is False by default.
property	The property, or typing, of a cell. For example, a peaceful Rosetta, a Start, an End... etc.

Following are actions one can operate on Cell objects and their purposes:

Method Header	Purpose of Method
getContainPiece(self)	Acquire the piece that this cell contains.
removePiece(self, tgP)	Remove piece tgP from this cell.
getCopy(self)	Acquire a copy of this cell. Cells that are copies will have their property isCopy be True .
__eq__(self, other)	Compare two Cell objects self , other by their row , column , and whether their property isCopy are equal. The necessity in the property isCopy is such that the program will not equate copies with actual cells, and its importance will be shown in an implementation in Board module.
__str__(self)	Acquire a String representation of a Cell object.
__repr__(self)	Acquire a representation of a Cell object.

2-1-B: Piece

A piece is a moveable object, each player possesses seven of them. It is significant to the victory condition of the game, can be contained by Cells, therefore are placed on Boards.

Upon initiation, each Pieces are contained by a cell with property Start, and as soon as the program detects seven non-copy Pieces on the non-copy End cell, the program will immediately dim the victory to the player.

The Piece module contains a class Piece, representing a piece on the gameboard.

Upon initiation a Piece will gain the following properties:

Property	Purpose of Property
canCapture	A Boolean value indicating whether a piece may be captured. It is True by default. This is a class variable.
pieceNumArr	A list indicating the current number of produced pieces on both sides. The number of produced pieces can surpass 7 since the program will produce copies of pieces. This is a class variable.
side	The side (or player) that this piece belongs to, can only be "L" or "R" .

position	The position that the piece is located at; it is None by default.
isCopy	A Boolean value indicating whether this piece is copied. It is False by default.
canMove	A Boolean value indicating whether this piece is moveable. It is True by default.
code	An alphanumeric string representing the code number of this piece. In this string, the player side letter comes first and the piece number second.
pieceNum	The piece number of the piece, in a range from 0 to 6, inclusive. This property of a piece is either designated in the initiator keyword parameter or dependent upon class variable by pieceNumArr default.

Following are actions one can operate on Piece objects and their purposes:

Method Header	Purpose of Method
sideNum(self, side)	Acquire the side number of variable side . If it is “ L ”, it is 0, else it is 1.
getCopy(Self)	Acquire a copy of this cell. Pieces that are copies will have their property isCopy be True .
__eq__(self, other)	Compare two Piece objects self , other by their side , pieceNum and whether their property isCopy are equal. The necessity in the property isCopy is such that the program will not equate copies with actual cells, and its importance will be shown in an implementation in Board module.
__str__(self)	Acquire a String representation of a Piece object.
__repr__(self)	Acquire a representation of a Piece object.

2-1-C: Board

A Board object represents the gameboard that is used in Royal Game of Ur. It is core to the simulation, gameplaying, as well as strategy construction of the project.

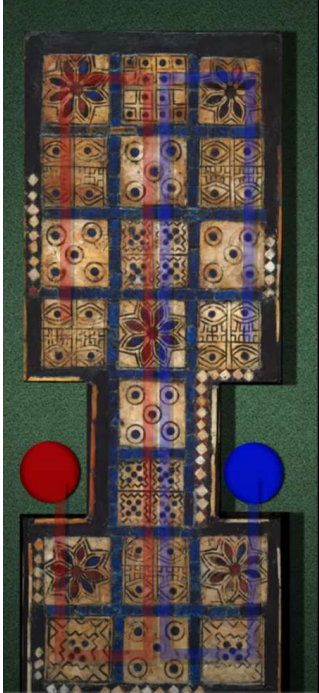
This module is built based on methods from the Cell module and Piece module and contains two classes. The first class is Board, representing a gameboard used in the game. The other class is iBoard, short for imaginary board, used solely in simulations for the strategy, which operations for artificial players are heavily based on. It is undesirable to directly manipulate the current gameboard for strategy simulation in a game, just as how someone shall not directly move pieces in a chess gameplay to simulate their strategies instead of in the player’s head. Using that analogy, iBoard is the imagined board in the artificial players’ head who are simulating future outcomes when they simulate moves.

The contribution of iBoard is in the depth of strategy thinking it enabled for artificial players. In the above analogy it created the brain of artificial player, although quite narrow-purposed for gameplay and fixed on one calculated, principled strategy. Other benefits and accompanying disbenefits will be analyzed when experiments on strategies are discussed.

Upon initiation a Board will gain the following properties:

Property	Purpose of Property
field	An array of Cells following the below coordinate system.
inGamePs	An array of available pieces (represented by Piece objects) in gameplay.

This is the coordinate system of the gameboard as well as for the Cell positions.



L0 Rosetta	M0	R0 Rosetta
L1	M1	R1
L2	M2	R2
L3	M3	R3
L4 Start	M4 Peace Rosetta	R4 Start
L5 End	M5	R5 End
L6 Rosetta	M6	R6 Rosetta
L7	M7	R7

Following are actions one can operate on Board objects and their purposes:

Method Header	Purpose of Method
getFieldInfo(self, limit = 7)	Acquire the information of all in-game pieces whose pieceNum are less than limit in a list.
getStart(self, side)	Acquire the cell of Start position at side .
getEnd(self, side)	Acquire the cell of End position at side .
getColNum(self, col)	Acquire the column number of column col . If it is the left column, number is 0; if middle, 1; if right, 2.
getCell(self, row, column)	Acquire the Cell object located at row and column .
nextCell(self, tgCell, side)	Acquire the cell immediately next to tgCell of the on-board path designated to player of side .
getPath(self, side)	Acquire the on-board path for player of side .
nStepNextCell(self, tgCell, steps, path)	Acquire the cell that is steps amount of step after tgCell on path . If that cell does not exist (for example, went some squares over the End position), return None instead because there is no such destination.

getDistance(self, cellA, cellB)	Acquire the distance between cells cellA and cellB on the board self .
movePiece(self, tgPiece, step)	Moves piece tgPiece for number of steps equal to step . However, if the destination of the piece after that number of steps does not exist (for example, went some squares over the End position), the piece will not be moved.
__str__(self)	Acquire a String representation of a Board object.
__repr__(self)	Acquire a representation of a Board object.

To represent an imaginary gameboard, it is convenient for the class `iBoard` to inherit from the class `Board`, and it does.

Before introducing the initiation of `iBoard`, it is necessary to recognize that it utilized a special implementation of the constructor of `Board` class such that it may freely define its own property **inGamePs**, unlimited to the superclass constructor.

Upon initiation an `iBoard` will gain the following properties:

Property	Purpose of Property
original	The Board by which this <code>iBoard</code> is based on.
leftPs	A list containing all pieces from the left player.
rightPs	A list containing all pieces from the right player.
piecesDict	A dictionary where sides are key corresponding to values being their list of pieces.
inGamePs	A concatenated list of leftPs and rightPs .
field	An array of copied Cells following the below coordinate system.

The information of the field is also processed in the `iBoard` constructor, while it does not later become an object property.

Following are actions one can operate on Board objects and their purposes:

Method Header	Purpose of Method
getPSet(self, side)	Acquire the list of pieces of player of side .
otherPSet(self, pSet)	Acquire the list of pieces of opponent of owner of pSet .
getPiece(self, side, pieceNum)	Acquire the piece that is of side and pieceNum on the board.

Moreover, because an `iBoard` is a imaginary board, it should be using copied Cells and Pieces instead of real ones that exist in the `Board` object used directly in the game. This point will be repeated when discussing implementation of strategies in Section III.

2-2: Player and Artificial Player

The Player module is rather concise within the progress of Phase 1, representing the elements of a player.

Upon initiation a Player will gain the following properties:

Property	Purpose of Property
playerNum	The ID of a player, starting from zero. For every new player initiated will this number increase by 1. This is a class variable
playerID	The ID of a player.
name	The name of a player. The current default name is 名もなき (nameless).
side	The side of a player.
pieces	A list of all pieces the player possesses.

Following are actions one can operate on Board objects and their purposes:

Method Header	Purpose of Method
<code>__str__(self)</code>	Acquire a String representation of a Player object.

Besides above properties, the composition of a player should also involve a strategy. Unfortunately, a Strategy module has not been created yet, and is currently involved in the Game module. In later phases, the Strategy module should be created such that all artificial player strategies are located in this file, and there shall be more properties added to the Player module or some other relevant location when implementing an interactable interface.

2-3: Game Module

A Game object will simulate a gameplay, the main engine for in-game operations as it imports from the Cell, Board, Piece, Player module to initiate games.

Upon initiation a Game will gain the following properties:

Property	Purpose of Property
pArray	A list involving the probabilities of rolling each possible number of steps from dices.
player1	A Player object representing the left-side player.
player2	A Player object representing the right-side player.
playerDict	A dictionary where the keys are sides and values are players of corresponding side.
board	A Board object representing the gameboard.

From the initiator the Game object has assembled all elements needed for a game to be held. A notated board, two players, the Sumerian 1d4, and 14 pieces are prepared. However, since the game is held in an electronic setting, there must be other operations to initiate and continue the game:

Method Header	Purpose of Method
<code>roll_dice(self)</code>	Simulate a Sumerian 1d4 based on the description in Section 1.
<code>getPieces(self, side)</code>	Acquire the list of pieces held by player of side .
<code>getPiece(self, side, pieceNum)</code>	Acquire the specific piece of side and pieceNum .
<code>oppSide(self, side)</code>	Acquire the opposing side of player of side .
<code>otherPlayer(self, side)</code>	Acquire the opponent of player of side .
<code>piece_filter(self, side, currStep, blackboard)</code>	Acquire a filtered list of pieces of side that will not violate movement rules when moved for currStep amount of steps using an imaginary board (iBoard) blackboard .
<code>select_move(self, utilArr, side)</code>	Select the move for player of side that grants the best util, with util calculation results listed in utilArr .
<code>victoryCheck(self, side)</code>	Checks whether player of side has won the game.
<code>Play(self, strategyLeft, strategyRight, depthLeft = 2, depthRight = 2)</code>	Simulate a gameplay with left-side player's strategy strategyLeft being with depth depthLeft and player's strategy strategyRight being with depth depthRight .

Note that this is not the entirety of Game module, as the strategies of artificial players have not been introduced.

From the above introduction, basic gameplay mechanics, rules, and its implementations have been introduced. Later sections will build on the knowledge provided here and discuss the two currently implemented strategies of artificial players.

3. Strategies of Artificial Players

The strategies of Artificial Players are essentially algorithms. It is either an algorithm forged by the programmer himself, or an algorithm modified from an already existing one. In the two implemented strategies for artificial players in Phase 1, one of them is an original algorithm that has a recursion depth of 0, while the other one is an expectimax-based algorithm of customizable depth (virtually, as Python itself has a maximum recursion depth length).

In the implementation of strategies, each strategy has one main strategy function with name `<strategy_name>` and util methods with name format `<util_name_condition>`. The strategy function will let the util methods evaluate the amount of util in moving a specific piece via a customized scale of utils. Finally, strategy functions return a list of tuples later passed to the `select_move` function of Game module to determine a move.

In each calculation in util methods and call to strategy functions will an iBoard be initiated, such that the “in-head” simulations of artificial players, which are required for above operations, do not influence the main board. Therefore, whenever an iBoard is produced, all cells and pieces in the iBoard are copies of themselves, such that when there are movements on the imaginary board, it does not interfere with the main board. Such is the significance in the object property `is_copy`.

In the following introductions for each of the implemented strategies, there will also be graphs of predicted movements from the artificial player using that strategy, so to provide examples and patterns of its movements beside the text explanation offered.

3.1: Strategy Massugu

Strategy Massugu is an original, nonrecursive strategy that emphasizes “greed”. It attempts to move the piece in the way that expands one specific piece’s advantage the most. In this case, the advantage is its distance from its Start position. This is done for two reasons: to approach the End position faster, and to be able to escape opponent piece chasing to deport it.

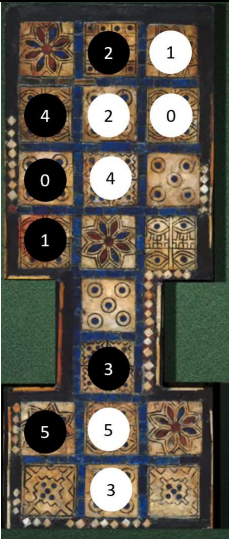
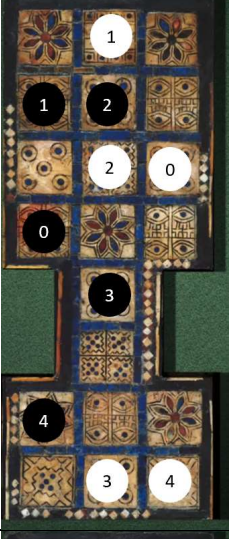
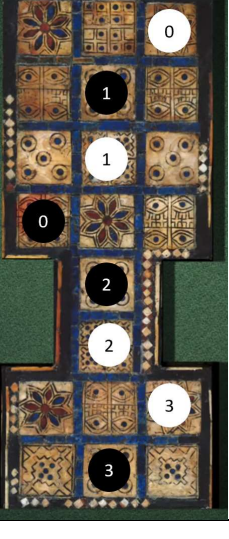
The naming of this strategy comes from the Japanese word 真つ直ぐ, meaning straightforward, reflecting the straightforwardness of this strategy to just proceed as far as possible with available cells.

The util function of Massugu calculates the util in moving a piece X whose destination has piece Y as:

$$\boxed{distance(start, X) + distance(start, Y)}$$

Again, like its naming, it is very straightforward. Massugu always looks to move pieces that are far from its start and can deport opponent’s pieces back to their start. In summary, it is a simple greedy strategy, attempting to make as much progress as possible on one single piece. Because of such property, the util for Strategy Masuugu will always be a nonnegative value.

Below are examples of decisions it makes for a given board and the average time it took for the left player (black pieces) to make the decisions.

Board	Step Count	Average Time	Decision and Outcome
	1	3.2 milliseconds	Moved Piece L3 Outcome: 1 step forward, deport Piece R5
	2		Moved Piece L3 Outcome: 2 steps forward, deport Piece R3
	3		Moved Piece L3 Outcome: 3 steps forward, 2 steps until End position
	4		Moved Piece L4 Outcome: 4 steps forward, deport Piece R4
	1	3.8 milliseconds	Moved Piece L4 Outcome: Reached End position
	2		Moved Piece L3 Outcome: 2 steps forward
	3		Moved Piece L3 Outcome: 3 steps forward, deport Piece R3
	4		Moved Piece L1 Outcome: 4 steps forward, deport Piece R2
	1	7.2 miliseconds	Moved Piece L2 Outcomes: 1 step forward, deport Piece R1
	2		Moved Piece L3 Outcomes: 2 steps forward, 1 step until End, Rosetta
	3		Moved Piece L3 Outcomes: Reached End position
	4		Moved Piece L1 Outcomes: 4 steps forward, deport Piece R2

3.2: Strategy Ashikazu

Strategy Ashikazu is a recursive algorithm that emphasizes its utility calculation on the difference in step number both sides can generate in a number of turns.

The decision-making of this algorithm is more complex than that of Strategy Massugu. Following is the procedure in this algorithm:

1. Acquire the amount of utils produced in moving a piece for a number of steps.
2. For the designated recursion depth N of this algorithm, it executes the following procedure for $N-1$ times:
 1. Calculate the maximum util the player of next turn is able to make.
 2. If the next player is the player that the strategy is calculating utils for, add that amount of utils to the current util of each resembling piece; else, subtract.
 3. Alternate the player calculated utils for in step 1 (opponent into self, self into opponent).

Therefore, the calculation for util is approximately as follows:

$$\boxed{\text{distance made by self} + \text{deport distance by self} - \text{distance made by opponent} - \text{deport distance by opponent}}$$

The time complexity of this algorithm should be $O(4^n)$, where n is the designated recursive depth of the algorithm, since each simulation of next turn's on-board dynamics will have four branches, each resembling an emulation of the possible steps the opponent can take. In summary, this is a quite complex algorithm to operate with, despite its current rapid execution time in experiments that will be observed later in this section.

For the purpose of completing both step 1 and step 2-2, this algorithm contains two util functions, each representing different modes of calculating utils. The first util function calculates the utils for a movement with a concrete step number, which is used for util estimation in step 1. The second util function provides an optimistic estimation for the amount of utils a piece can get given there is an unknown available step number in the interval from 1 to 4, inclusive.

The advantage in this algorithm is that it is expected to produce moves that are more calculated, presumably those that would lead it to more victories. However, the experiment explained in later sections shall provide a better, empirical insight on this prediction.

The implementation-wise disadvantage in this algorithm is more conspicuous and prominent. First of all is the complexity of this algorithm with exponential complexity. Albeit complexity and efficiency are not completely synonymous, big order of growth of complexity posts worries for efficiency of the algorithm. Second of all is the OOP feature, iBoard, that posts worry for the space complexity for this algorithm. One iBoard object is generated for each simulation to facilitate it and segregate piece movements in simulation from that of the actual gameboard (again, it is the artificial player's "brain").

Below are examples of decisions this algorithm makes for a given board and depth with relevant data:

Board	Depth	Average Time	Step	Decision
	2	24.2 ms	1	Moved L3
			2	Moved L3
			3	Moved L1
			4	Moved L0
	3	21.6 ms	1	Moved L3
			2	Moved L3
			3	Moved L3
			4	Moved L4
	4	25.8 ms	1	Moved L2
			2	Moved L3
			3	Moved L1
			4	Moved L4
	2	28 ms	1	Moved L2
			2	Moved L1
			3	Moved L3
			4	Moved L0
	3	25.2 ms	1	Moved L2
			2	Moved L1
			3	Moved L3
			4	Moved L2
	4	38.2 ms	1	Moved L2
			2	Moved L1
			3	Moved L0
			4	Moved L2
	2	15 ms	1	Moved L2
			2	Moved L5 (at start)
			3	Moved L5 (at start)
			4	Moved L1
	3	23 ms	1	Moved L1
			2	Moved L0
			3	Moved L0
			4	Moved L1
	4	39.8 ms	1	Moved L2
			2	Moved L6 (at start)
			3	Moved L6 (at start)
			4	Moved L1

4. Experiment on the Efficiency of Strategies

This section will discuss an experiment on the efficiency of strategies; in particular, how much its results of calculation will defer from each of the strategies' conclusion for specific boards as well as the amount of time needed to complete decisions.

In this experiment, the programmer will provide three specific boards to each of the aforementioned implemented strategies (and for strategy Ashikazu it will be tested upon different recursive depths). The strategies will present their calculated conclusion of optimal move to the three boards provided, and these results will involve the amount of time used in each decision, the set of optimal moves generated by the strategies, and analysis will be written upon relevant observations.

4.1: Hypothesis

By now, our current knowledge is that the recursive algorithm has an exponential order of growth in time complexity. Thus, the time it takes to make those calculations should grow exponentially. Meanwhile, although the experiment does not collect expected value of utils for each possible moves on the board, it should be expected that the margin of difference from the shallower calculations to deepest level of calculation dwindle, and that consecutive depths have very small changes given the emulations of consecutive depths are one move away from each other.

Therefore, we can make the following hypothesis:

If the recursion depth of algorithm Strategy Ashikazu increases, the time it takes to calculate an optimal move will increase and its difference from the deepest calculation will decrease, because the algorithm has a time complexity of $O(4^n)$ and its calculation grows closer to the deepest as results of emulation repeat more.

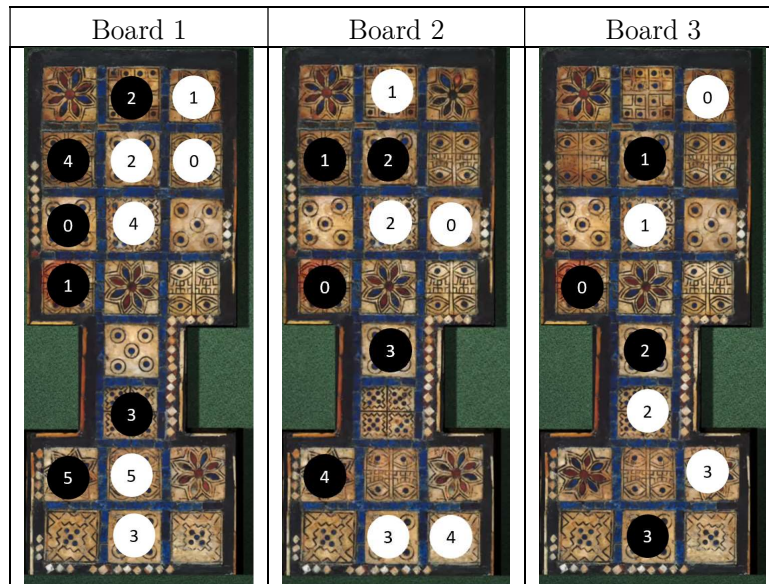
4.2: Procedure of Experiment

The experiment is carried first by producing three boards. Then, using the strategy functions and other contents of the Game module, it will grant the optimal move for each board calculated by the strategy functions.

To further simplify the operation, the Game module involves strategy functions modified into a lambda function such that testing the aforementioned properties on strategy Ashikazu of different recursion depth becomes more convenient and concise on code:

```
STRATEGY_ASHIKAZU = lambda deep: lambda currPl, game, stepNum: game.select_move(
    game.strategy_ashikazu(
        currPl,
        iBoard(
            game.board,
            game.getPieces("L"),
            game.getPieces("R")
        ),
        currStep = stepNum, depth = deep
    ),
    currPl.side
)
```

Then, the programmer creates three virtual boards to work with:



Strategy Massugu and Strategy Ashikazu of depths between 1 and 15, inclusive, are then tested upon these three boards for aforementioned data.

4.3: Observations

Below is the resulting data for this experiment:

Diagram 1: Table of Average Runtime, Decision, and Margin of Difference across Different Depths

Depth	Average Runtime (ms)	Board Decision	Margin of Difference from Deepest Calculation
0	2.867	333443312331	10
1	3.667	334421312561	9
2	25	331021302551	9
3	20.733	333421321001	8
4	26.333	231421022661	8
5	32.6	221421012661	8
6	39.467	221426011441	6
7	43.333	221426000441	5
8	51.067	221421020661	6

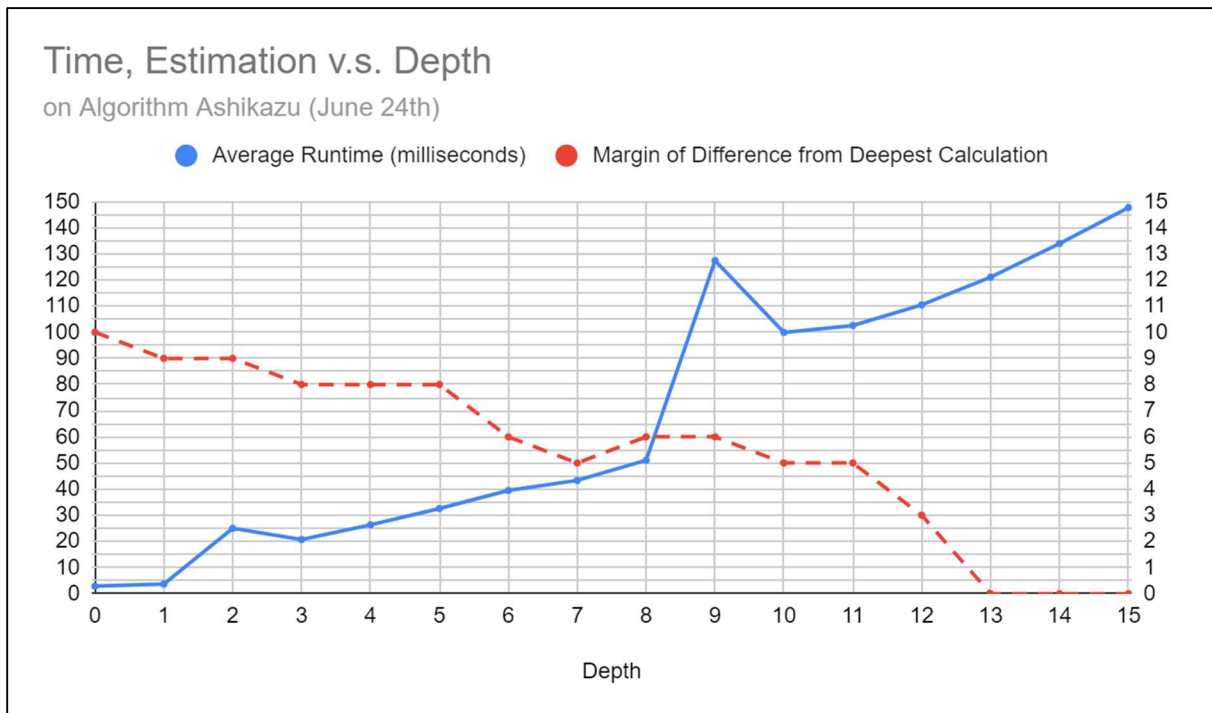
9	127.4	221421020001	6
10	99.933	221421320001	5
11	102.567	221421320661	5
12	110.467	421425320661	3
13	121.067	421425320444	0
14	133.933	421425320444	0
15	147.667	421425320444	0

Additional Notes:

1. Depth 0 stands for Strategy Massugu
2. The string of numbers in column Board Decision is represented as the piece number moved by each strategy as its optimal move.
3. The margin of difference of measurement is calculated as the number of different decisions made by one depth of algorithm compared to that made by the depth 15 calculation.

And here is a graph summarizing the above information:

Diagram 2: Graph of Average Runtime, Decision, and Margin of Difference across Different Depths



4.4: Analysis

Diagram 2 demonstrates the two following trends on average runtime and margin of difference from deepest calculation:

1. The average runtime of the implemented strategies across increasing recursion depths grows in an exponential pattern, as observed from the graph. This is coherent to the presented hypothesis.
2. The margin of difference in strategies' conclusions across increasing recursion depths dwindles in small degrees for most consecutive depths. This is coherent to the presented hypothesis.

Using this information above, the project may then be lead into a second experiment regarding the effectiveness of these implemented strategies against each other, based on the difference in decision they carry out. Strategies may then be classified into the following leagues:

League	League Definition	Qualified and Representative Strategy
A	Strategy is non-recursive.	Qualified: SM Representative: SM (so to have one end of spectrum on the experiment data for comparison).
B	MDDC is in [8, 9].	Qualified: SA-1, SA-2, SA-3, SA-4, SA-5 Representative: SA-1
C	MDDC is above [6, 7].	Qualified: SA-6, SA-8, SA-9 Representative: SA-6
D	MDDC is [4, 5].	Qualified: SA-7, SA-10, SA-11 Representative: SA-10 (SA-7 might resemble behaviors too close to SA-6).
E	MDDC is [1, 3].	Qualified: SA-12 Representative: SA-12
F	MDDC is 0.	Qualified: SA-13, SA-14, SA-15 Representative: SA-15 (so to have one end of spectrum on the experiment data for comparison).
Additional Notes: <ol style="list-style-type: none">1. MDDC stands for Margin of Difference in from deepest calculation.2. SM stands for Strategy Massugu, SA-D stands for Strategy Ashikazu of recursive depth D.		