Strongly solving the Royal Game of Ur

Padraig Lamont, and Jeroen Olieslagers

March 2025

Abstract

We have strongly solved multiple conjectured rule sets for the Royal Game of Ur, an ancient stochastic two-player board game, including the Finkel, Blitz, Masters, and Aseb rule sets. Our system can be adapted to strongly solve other variants of these rule sets as well. For each rule set, we computed a map containing the probability that the first player would win when starting from each position, under optimal play. This is used to find the optimal move from every position by selecting the move that leads to the current players highest chance of winning. This report details our approach of using value iteration and the Bellman equations to generate the solution maps, provides metrics about the convergence of our algorithm, and presents practical optimization techniques that significantly reduce computational effort.

1 Introduction

This report is a work-in-progress. The method is complete, but we still plan to expand upon verification and results further.

The Royal Game of Ur is a two-player race game containing chance, that has been dated back over 4500 years. There have been many proposed rule sets for the Royal Game of Ur over time, designed to match historical evidence or to create a fun game to play. The most popular rule set today is the Finkel rule set, proposed by Irving Finkel. We have solved this rule set, along with other commonly played rule sets including the Blitz rule set, the Masters rule set, and a rule set for a similar game called Aseb that is played on a longer board. We strongly solved these rule sets by exhaustive computational analysis.

To "strongly solve" the game means to determine the optimal move from every possible game state, enabling perfect play from any starting point. This was accomplished by modeling the game as a Markov decision process and applying *value iteration*, a dynamic programming algorithm, to compute the probability that the first player (henceforth "light") would win starting from every possible game state, under optimal play.

Through our use of value iteration to solve the Bellman equations, the optimal win probabilities and corresponding optimal moves are obtained for all possible states in each rule set. This includes all of the 2.76×10^8 possible states under the Finkel rule set, the 8.25×10^7 states for Blitz, the 1.00×10^9 states for Masters, and the 8.25×10^7 states for Aseb.

This report describes the computational approach used to solve these rule sets, focusing on the formulation of the Bellman equations, the implementation of value iteration, and techniques to optimize convergence. We will also report metrics related to the convergence of the algorithm, metrics about our final solved map artifacts, and link to our open-source libraries that were used to solve the game.

2 Methods

2.1 Bellman Equations for Optimal Play

The Royal Game of Ur can be modeled as a stochastic game with perfect information and chance events (dice rolls). We define $V(s)$ as the value of state s, representing the probability that light eventually wins from s under optimal play. Terminal states (in which one player has already won) have boundary values: $V(s) = 1$ if s is a state where light has won, and $V(s) = 0$ if s is a state where dark (the opponent) has won.

For any non-terminal state, the Bellman optimality equations can be written to reflect the turn-based, adversarial, and stochastic nature of the game. If it is light's turn in state s, that player will choose an action (move) that maximizes their winning probability. If it is dark's turn, dark will choose a move that minimizes $V(s)$ (since dark's goal is to minimize light's chance of winning).

Moreover, before a move is chosen, a dice roll occurs, introducing a random chance over possible rolls. Let $\mathcal{R}$ be the set of possible dice rolls, and let $\mathcal{A}(s, r)$ be the set of legal moves in state s given a roll r. The probabilities

$P(r)$ for $r \in \mathcal{R}$ represent the chance of rolling r using the dice. We can then express the Bellman equations as follows:

$$V(s) = \begin{cases} \displaystyle\sum_{r \in \mathcal{R}} P(r) \max_{a \in \mathcal{A}(s,r)} V\big(s_{r,a}\big), & \text{if } s \text{ is light's turn,} \\ \displaystyle\sum_{r \in \mathcal{R}} P(r) \min_{a \in \mathcal{A}(s,r)} V\big(s_{r,a}\big), & \text{if } s \text{ is dark's turn.} \end{cases} \tag{1}$$

Here, $s_{r,a}$ denotes the successor state that results from state s after a dice roll r and a subsequent move a. These equations formalize how the value of a state is derived from the values of its immediate successor states: on light's turn one assumes optimal play by light (taking the maximum value outcome) and on dark's turn one assumes optimal play by dark (taking the minimum value outcome), and in all cases averaging over the probability distribution of dice outcomes. Equation (1) embodies Bellman's principle of optimality: at any state, the optimal play yields a value that is the best (or worst) achievable via one step followed by optimal play thereafter.

2.2 Value Iteration Algorithm

We employed value iteration to solve these Bellman equations. Value iteration is an iterative dynamic programming method that starts with an initial approximation of $V(s)$ for all states and repeatedly updates these values using the Bellman equations until convergence.

In our implementation, all states were stored in a map that represented V, where each state was mapped to the chance that light would win starting from that state (we call this map the "state map"). Terminal states were initialized to their true values (1 for light wins, 0 for dark wins), while all other states can be initialized arbitrarily (e.g., to 0.5). These initial guesses will later be corrected by iterations of value iteration.

In each iteration of value iteration, the algorithm visits every non-terminal state s and replaces the current stored value in the state map, $V(s)$, with a new value computed by the right-hand side of Equation (1). This update retrieves values of successor states, $V\big(s_{r,a}\big)$, from the state map as well.

This iterative process can be applied either synchronously (read from an old map of V, and write into a new map to use in the next iteration) or asynchronously (update one copy of V in-place). In either case, the values converge to the true optimal values as the updates are repeated. In our case, we updated the state map in-place.

The iteration is stopped when the maximum change, $\max_s |V_{\text{new}}(s) - V_{\text{old}}(s)|$, falls below a small threshold. This indicates that $V(s)$ has stabilized within the desired precision. For our largest and most accurate maps, we stopped value iteration when there was not a single bit that changed in the entire map during an iteration when using 64-bit floating point numbers to store the win probabilities.

2.3 Optimization using Irreversible Moves

The first major optimization for solving the Royal Game of Ur arises due to irreversible moves. In the Royal Game of Ur, scoring a piece is irreversible. Once you score a piece, it is not possible to reach a state before you scored the piece again. We can exploit this to optimize our use of value iteration.

The reason that value iteration is required to solve the Markov decision process representing the game is due to the cycles within the game tree. These cycles stop us from simply evaluating the game tree in a single pass. However, the irreversible move of scoring a piece breaks all cycles in the game tree, providing directed boundaries. These boundaries break the game tree up into subsets, where each subset only depends upon its successor subsets (the next subsets of states that can be reached by scoring a piece). We can, therefore, solve each of these subsets one-by-one, in an order that goes backwards from the end of the game.

This still guarantees optimality, and it reduces the computational effort required to solve the game significantly. Instead of iterating through the entire state space every iteration, we can just iterate through a subset of the state space until it converges, before moving on to the next subset. This reduces time spent on wasted calculations for states whose successor states have poor estimates of their value.

2.4 Optimization using Game Symmetry

The second major optimization for solving the Royal Game of Ur arises due to the symmetry between the two players in the game. If you swapped the two players, the chance of winning starting from the swapped position would not change. This allows us to only compute and store half of the states in each rule set. In our case, we stored only states where it is light's turn.

When we want to look up a state where it is dark's turn, we swap the players in the position, look up the chance that light would win if it were light's turn, and then invert the resulting probability by computing $1-p$. This will give us the probability that light would win from a state where it is dark's turn.

2.5 State Map Implementation

A challenge in solving the Royal Game of Ur is the size of its state space. For example, the Finkel rule set contains 2.76×10^8 reachable states in total, and the Masters rule set contains over a billion states. Storing this many states requires careful memory management. The main concern is that storing the game states in the map can take up a lot of memory.

We addressed this by encoding each state into a compact bit representation for use as keys in our state map. Each state in the Finkel rule set is encoded into 32-bits, and each state under the Blitz, Masters, and Aseb rule sets are encoded into 34-bits. These encodings include all essential information (position of pieces and scores), using bit-packing techniques to remove redundancies and compress the contents of the board.

Due to the symmetry of the game, the turn indicator bit can be omitted by always treating a state from the perspective of the light player. The number of pieces that each player has yet to play can also be omitted, as this can be calculated from the state of the game board and the players' scores.

The position of all pieces on the game board is stored in two separate sections of the binary key. The first section stores the tiles in each players' safe zones, and the second section stores the tiles in the war zone. In the safe zones, only one bit is needed to store the state of each tile, as each tile is only accessible by one player. We just need to store whether each tile is occupied or not. The war zone tiles have 3 states they could take. Tiles could be empty, be occupied by light, or be occupied by dark. To store these states, we encode each war-zone tile using 2-bits and then compress the results using a simple dictionary encoding.

3 Verification

We have generated maps down to a stopping tolerance of 10^{-8}. At this tolerance, and all tolerances at or below 0.0001% (10^{-4}), there are no ties in win percentage between available moves from each state. This means that we know the single best move to make from each state at a precision down to approximately 0.00000001%. We are confident that this tolerance on the win percentages that we calculated leads to the selection of optimal moves.

However, at the 10^{-3} tolerance that the smaller HuggingFace models are saved to, there are ties between moves in some states. This occurs because there are some states where the difference in win percentage between moves is less than 10^{-3}, which our 16-bit values cannot represent.

4 Results

The final artifacts that we produced for each rule set are maps containing the win probability for light from every state where it is light's turn. We have generated two versions of these maps for each rule set. Currently, the HuggingFace only contains maps with 16-bit values. We plan to upload new 32-bit versions of these models that are more accurate soon, for applications where absolute optimal play is desired.

Rule Set	States	Map Entries	Size (16-bit)
Finkel	2.76×10^8	1.38×10^8	827.4 MB
Blitz	8.25×10^7	4.12×10^7	247.5 MB
Masters	1.00×10^9	5.01×10^8	3.01 GB
Aseb	8.25×10^7	4.12×10^7	247.5 MB

Table 1: Size of maps for each rule set

We provide downloads for the solved maps on HuggingFace, and libraries for generating and reading them on GitHub:

- Download the maps on HuggingFace:
  `https://huggingface.co/sothatsit/RoyalUrModels`
- Read and generate the maps using RoyalUr-Java:
  `https://github.com/RoyalUr/RoyalUr-Java`
- Read the maps using RoyalUr-Python:
  `https://github.com/RoyalUr/RoyalUr-Python`
- Generate the maps using Jeroen's Julia implementation:
  `https://github.com/JeroenOlieslagers/game_of_ur`