Hogfish: Solving UC Berkeley's CS 61A "Hog" Game

Jeffrey Tong

October 8, 2023

Abstract

In my first semester (Fall 2022) as an undergraduate at UC Berkeley, I took the introductory computer science class CS 61A: The Structure and Interpretation of Computer Programs. The first project was to implement a command-line interface (CLI) for the board game *Hog*, invented by course staff and changed slightly every semester, including some trivial hard coded computer strategies. While not required for the class, I decided out of curiosity to program a more complete computer strategy that I called "Hogfish" (inspired by the Stockfish chess engine), an engine that can play Hog perfectly by quickly outputting the move with highest win probability $P(N)$ at any game state. In particular, under standard rules for the Fall 2022 class, the best first move for the first player is to roll three dice, yielding $P(N) = 51.736\%$. Hogfish can also display a human-readable analysis of every possible move at the game state, including the exact win probabilities. On a conventional laptop, Hogfish initializes by generating a tablebase in about two seconds, then computes as many individual moves as needed instantly, making it accessible to the general public. Planning and implementing the algorithms efficiently involved interesting results, heavily drawing on my background in competitive programming. With the aid of Hogfish, we can also find some surprising insights on how humans can play better.

Contents

1 Introduction

1.1 History

The *Hog* game seems to be unique to UC Berkeley's CS 61A class, with the earliest mention I found of a close ancestor being the *Pig* project from the Fall 2011 iteration of CS 61A. Similar games involving randomness, especially Blackjack, have appeared even earlier, although earlier iterations of the websites have less information and are thereby more difficult to investigate. At the time of this writing, the class was taught by Professor John DeNero and instructor Justin Yokota.

1.2 Rules

The rules at the time of this writing can be found on the *Hog* project website.

There are three special rules:

- Sow Sad
- Pig Tail
- Square Swine

2 Theory

2.1 Formalization

This description adapts ideas from game theory, in which N and P conventionally respectively denote the next and previous players to move, and we distinguish between the terms *move* and *turn*. For generalization, define constants D, S, and G as follows:

Constant	Default	Meaning
D	10	Maximum number of dice rolled each turn
S	6	Number of sides on each dice
G	100	Goal score

First, we algebraically formalize Hog. We will refer to the first and second players to move as *P1* and *P2*, respectively. In general, each game state can be encoded as an ordered tuple, here called a "node". A node can be implemented as an ordered triple (s_1, s_2, nxt), where s_1 and s_2 are the scores of P1 and P2, and nxt is either P1 or P2, depending on the next player to move. For the purposes of computing the next move, an engine only needs to consider states where neither player has reached the goal. Mathematically, $0 \leq s_1, s_2 \leq G$, and we define these states as "nonfinal" states, resulting in $2G^2$ nonfinal states.

Symmetry enables further reducing the number of nodes by only tracking scores from N's perspective. If s_{N} and s_{P} are the scores of N and P, the game state can be compressed into the ordered pair $(s_{\mathrm{N}}, s_{\mathrm{P}})$, where $0 \leq s_{\mathrm{N}}, s_{\mathrm{P}} < G$, reducing the number of nonfinal nodes to G^2 (default: 10000). This is small enough that a program can store all of them. However, care must be taken to stay in N's perspective: invert the order after each turn, so the node after t turns is $(s_{\mathrm{N}}[t], s_{\mathrm{P}}[t])$, and this is replaced with $(s_{\mathrm{P}}[t], s_{\mathrm{P}}[t+1])$ after $t+1$ turns. By the nature of this encoding, every node represents a unique game state, and every game ends with P winning and $0 \le s_{\mathrm{N}} < G \le s_{\mathrm{P}}$. Having eliminated the nxt variable from our encoding, whether N corresponds to P1 or P2 must be tracked separately to create a playable UI. When mentioning players, we will now use the terms *current player* and *other player.*

We can compute the probability of each possible dice outcome via a probability space. When a player rolls $d > 0$ dice, the sample space comprises the S^d equiprobable d-tuples of the die values. Here, we define the term *sad sum* denoted by m, the sum of all dice rolls after applying Sow Sad but no other special rules.

2.2 Observations

When rolling $d > 0$ dice, $m \ge 1$, with equality iff any dice land on 1.

- If no dice land on 1, an event with probability $\left(1 - \frac{1}{S}\right)^d$, the expected value of the sad sum is $\frac{S}{2}d$.
- If any dice land on 1, an event with probability $\frac{1}{S^d}$, the sad sum is simply 1.

Thus, the overall expected value is

$$E(d) = 1 + \left(1 - \frac{1}{S}\right)^d \frac{S}{2}d.$$

Taking a derivative,

$$\begin{aligned} E'(d) &= \ln\left(1 - \frac{1}{S}\right)\left(1 - \frac{1}{S}\right)^d \left(\frac{S}{2}d\right) + \left(1 - \frac{1}{S}\right)^d \frac{S}{2} \\ &= \left(1 - \frac{1}{S}\right)^d \frac{S}{2}\left(d \ln\left(1 - \frac{1}{S}\right) + 1\right) \\ &= \left(1 - \frac{1}{S}\right)^d \frac{S}{2}\left(1 - d \ln \frac{S}{S-1}\right). \end{aligned}$$

This gives the inequality

$$\begin{aligned} E'(d) > 0 &\iff d \ln \frac{S}{S-1} < 1 \\ &\iff d < \frac{1}{\ln \frac{S}{S-1}} = \log_{\frac{S}{S-1}} e, \end{aligned}$$

so by the first derivative test, $E(d)$ is a unimodal function with maximum at $d_{\max} = \log_{\frac{S}{S-1}} e$.

For the default value of $S = 6$, $d \approx 5.48$, and coincidentally $E(5) = E(6) \approx 7.03$. **It follows that for a strategy consisting solely of playing a fixed number of dice d, both $d = 5$ and $d = 6$ are optimal.**

Player scores always increase on their turn, and scores remain constant during the other player's turn, so every game's node sequence can be represented as a directed acyclic graph (DAG). Win probabilities after every move at every node assuming perfect play can be computed exactly from future nodes assuming minimax if the probability of each possible dice outcome can be found. Hogfish thus effectively generates a full tablebase. We define win probability (WP) and lose probability (LP) from N's persepective; Hog has no draws, so WP $= 1 - $ LP. Note that the WP values at $(s_{\mathrm{N}}, s_{\mathrm{P}})$ and $(s_{\mathrm{P}}, s_{\mathrm{N}})$ do not necessarily sum to 1 due to turn ordering. Note also that just maximizing the expected value of the next score is insufficient because Pig Tail penalizes gaining too many points in some cases, and reducing risk in the endgame may require selecting a lower EV.

It is also interesting to compute the maximum length of a game. To achieve this, both players could use the greedy strategy of rolling 1 die on each turn and

1. gain 1 point if their score is not 1 less than a perfect square or,
2. gain 2 points if their score is 1 less than a perfect square.

This would take P1 $T = G - \lfloor\sqrt{G-1}\rfloor$ (default: 91) turns to reach a score of 100. The game then lasts a total of $2T - 1$ (default: 181) plies. This strategy is optimal for prolonging the game since triggering Square Swine would increase the score by at least 3.

3 Implementation

3.1 Dice Table

We now describe an algorithm for computing the probability of each possible dice outcome efficiently. Sad sums do not apply for $d = 0$ as that case obeys a separate mechanism. For each sad sum $0 \leq m \leq SD+1$, we create an event with size n_m, the number of such d-tuples summing to m.

If any 1 rolls, $m = 1$, so $n_1 = S^d - (S-1)^d$. If no 1 rolls, $2d \leq m \leq Sd$, and the frequencies $n_{2d}, n_{2d+1}, ..., n_{Sd}$ are the coefficients of the ordinary generating function (OGF) $f(x) = (x^2 + ... + x^S)^d = x^{2d}(1 + ... + x^{S-2})^d$. f is a polynomial with $(S-2)d+1$ terms and we can expand with the standard algorithm for multiplication for small D. We store all $Sd+1$ n_s values in a list; repeat this for $1 \leq d \leq D$, and for $d = 0$, the list is empty. In each row, dividing the coefficients by S^d produces a list equivalent to a probability function in the variables d and m. Call this final table the dice table; its dimensions are $(D+1) \times (SD+1)$.

3.2 Move Table

Using the dice table, after each possible move from each node, we compute the resultant WP (for the current player) via dynamic programming (DP) as a weighted average. The base cases are the finished game nodes ($s_{\mathrm{P}} \geq G$), which we won't store, and LP $= 1$ for these. Our DP transitions consist of computing the next value of sp after considering all sad sum values, Pig Tail, and Square Swine, and combining them to find WP. If the next s_{P} is below G, WP $=$ next LP for this turn. Otherwise, we've reached a base case. The transition order cannot follow the standard row-by-row pattern because of the constant perspective inversions. Instead, we observe that the score total $s_{\mathrm{T}} = s_{\mathrm{N}} + s_{\mathrm{P}}$ is strictly increasing, so we can compute values for one new value of st at a time (tracing a diagonal-by-diagonal pattern). We store the final WPs for each possible move, as well as the maximum WP (equivalently, WP at that node under perfect play) in the tablebase.

4 Resource Consumption

These two tables take significant computational resources to generate and store, especially the tablebase. To accommodate this, Hogfish has two phases: an initialization phase and a playing phase. Hogfish begins with the initialization phase, which only occurs once, during which it generates the tables. During the playing phase, Hogfish finds the best moves at the given states.

To find the best move, Hogfish only needs to iterate over the $D+1$ possible moves and check which one(s) have resultant WP equal to the node's WP. If there are multiple moves with maximum WP, Hogfish selects one uniformly at random for variety.

Finally, we compute big-O time and space complexities as a function of D, S, and G.

Operation	Time	Memory	Def. Time	Def. Memory
Generate Dice Table	SD^3	SD^2	6000	600
Generate Tablebase	SD^2G^2	DG^2	6×10^6	10^5
Analyze Move	$D \log D$	D	33	10

Under standard rules, the tablebase took a maximum of 1.71 s to generate over five trials on my laptop (2.9 GHz Dual-Core Intel Core i5).