Solving Two-Player Blackjack and Variations

Colby Ohlund, Andy Si, Gabriel Visotsky, Freya Zhai

December 14th, 2025

Mellon College of Science

Department of Mathematics

Carnegie Mellon University

Pittsburgh, Pennsylvania 15213

Table of Contents

1 Introduction 1

1.1 Objective and Intuition for Optimal Play 1

2 Problem Setup 2

2.1 General Description 2

2.2 The Specific Instance We Will Consider 3

3 Methods 4

3.1 Information Available to Each Player 4

3.2 Enumerate All Possible Strategies 5

3.3 Dynamic Programming for the Payoff Matrix 8

3.4 Solving the Game via Linear Programming 11

3.5 Solving the Game Via Strategy Elimination 12

4 Results 12

4.1 Private Stand Game 12

4.2 Public Stand Game 14

5 Discussion 15

6 Future Work 17

Appendix Supporting Python Code 18

Introduction

Blackjack is a game played by millions of people around the world because of its simplicity and ease of play. The most basic form of Blackjack poker consists of two players, one known as the "House" or the dealer, and the other simply as the "player". The game can be neatly divided into rounds, where each round goes as follows. The "player" is dealt two cards, with both face up to the dealer, and the dealer also gets two cards, but the player can only see one of the cards face up. Then, the player is up first to either take a card (a hit) or to stay (keep the cards they have). Then, after the player goes, the dealer takes their turn with the same options. To win each round, the goal is to have the sum of the cards equal 21 or as close as possible without going over. For the numerical cards in a deck of cards, it is self-explanatory what their values are, but for jacks, kings, and queens, they all equal ten, and aces can equal either one or eleven (up to the players' choice). By design, the dealer has an inherent edge to playing as they play after all the players have played. When players "hit" for more cards and bust by getting a sum higher than 21, they lose - regardless of whether the dealer would have or eventually busts or not. To impose disadvantages on the dealer to adjust for the benefits of playing second, the dealer must abide by a set of rules: they must hit the value of their hand less than 17 (or 17 with an Ace that counts as 11 in some variations). This means that even if the player has chosen to stand on a 14 and the dealer has a 16 and could theoretically win the hand by standing, they must hit and not bust in order to win. Now, naturally, there is an inherent benefit to playing as the dealer since not only do they have more information at the beginning of the game, but also they go after they already know the number that they need to get to beat the player.

1.1 Objective and Intuition for Optimal Play

Now, many other potential rulesets, configurations, and variations on Blackjack can be played, but for our purposes, we will be looking at this streamlined version. With these guidelines in place, the main question we are looking to answer is "What is the optimal strategy for the "player" when playing Blackjack with these rules?" To investigate this question, we will use a policy improvement algorithm and an analysis of a constructed payoff matrix so that an optimal strategy can be derived for the dealer to maximize their profit.

At this point in time, it's also worth spending a few moments to understand what would look like an ideal, or optimal strategy from the player's perspective. Since the dealer intends to stand on a 17 or higher, it's desirable to aim to stand on a 17 or higher as well. If the current value of the hand is high and the chance of busting on the next hit is high, the player could also elect to stand on a value lower - because the outcome space of the dealer only includes 17 to 21 and bust, the odds of say, a value of 3 and the value of 16 to win the hand are actually identical. A table of the odds of a player busting based on the current value of their hand is shown below

Problem Setup

2.1 General Description

We will consider problems of the following general form. We will have a set of players: p_1, p_2. Each player will follow a predetermined strategy that may be stochastic and each player's strategy is unknown to the other player. The goal of the game is for each player to maximize their expected payout. Any winnings given to a player are taken away from the other player. We will also have a deck of cards D, represented as a set of positive integers $d \in D$, and we can define a probability mass function over D, called P, where $P(d)$ represents the probability that a particular card appears on any draw from D, note that $P(d) \geq 0$ and $\sum_{d \in D} P(d) = 1$. Finally, we will have a positive number B that represents the bust threshold, and a function $r : \mathcal{I}_{\text{End of Game}} \to \mathbb{R}$ which acts as the reward function. We have $\mathcal{I}_{\text{End of Game}}$ represent all information that is available at the end of the game, such as the final sums of both players, the value of B, the number of cards drawn, if any player stood or not, etc. A generalized reward function can be a function of any of these metrics.

The game is then as follows:

1. Both p_1 and p_2 have counts, c_1 and c_2, that start at 0 and follow fixed strategies.
2. For each player who has not stood or busted yet, draw a card d from D with governing p.d.f F_k. Add d_1 to the sum for player p_1 and add d_2 to the sum for player p_2.
3. If $c_1 > B$ then p_1 busts, if $c_2 > B$ then p_2 busts.

4. For each player who has not busted, they are given the opportunity to stand. If a player stands, they will either announce this to the other player or stand privately, based on the game format.

5. If each player has either busted or stood, finish the game by giving p_1 reward $g(\mathcal{I}_{\text{End of Game}})$ and p_2 reward $-g(\mathcal{I}_{\text{End of Game}})$. Otherwise, go back to step 2.

In particular, note that this rules set means a player cannot hit again once they have stood.

2.2 The Specific Instance We Will Consider

Taking Blackjack as an inspiration for the game we wish to solve, we will consider two specific instances of the above generalized game that closely match with Blackjack. This means we will have the following parameters:

- $D = \{2, 3, 4, 5, 6, 7, 8, 9, 10, 11\}$
- $P(x) = \frac{1}{13}$ for $x \neq 10$ and $P(x) = \frac{4}{13}$ for $x = 10$. A 10 represents a face card that isn't an ace, of which there are 3 of and an ace represents an 11. Therefore, we are picking a card uniformly at random from the entire deck of 52 cards.
- A bust threshold of $B = 21$, which can be hit but not exceeded.
- Let c_1, c_2 be the final counts for the players, the reward function is as follows,

$$g(\mathcal{I}_{\text{End Of Game}}) = \begin{cases} 0 & \text{If } c_1 > B \text{ and } c_2 > B \\ 1 & \text{If } c_1 \leq B \text{ and } c_2 > B \\ -1 & \text{If } c_2 \leq B \text{ and } c_1 > B \\ \text{sign}(c_1 - c_2) & \text{If } c_1 \leq B \text{ and } c_2 \leq B \end{cases}$$

  This means we give a point to the player with the higher count when both haven't busted, if one player busted we give the point to the other player, and if both players busted it's a tie. The g function is taken to be in the perspective of p_1, meaning a positive reward is p_1 gaining points and a negative reward is p_1 losing points, with p_2 earning the negation of a negative reward, which is positive.

Additionally, we will add the subtlety that 11s (Aces) can be treated as either 1s or 11s, whichever benefits the player more, as this is a rule of Blackjack.

We will solve two versions of the above game, one with a private stand, meaning when a player stands it is not announced, and one with a public stand, so that when a player stands, they must announce that to the opponent.

Methods

Here we describe how we compute equilibrium strategies for game described above. There are four main steps,

1. Consider what information each player had during the game
2. Specify a finite and sufficient strategy set for each player
3. Compute the expected payoff for every pair of strategies
4. Solve the payoff matrix via linear programming

Throughout, the reward function $g(\mathcal{I}_{\text{End of Game}})$, deck D, bust threshold B, and card probability mass function P are exactly as defined in the problem setup.

3.1 Information Available to Each Player

Both players always know their own current total t_i, their own stand status σ_i, the number of cards that they have drawn, the common deck distribution P and bust threshold B, the reward rule $g(\mathcal{I}_{\text{End of Game}})$.

The only difference between the two game variants is whether stand decisions are public or private:

- Public stand: each player also observes the opponent's stand status, so they know σ_{-i} exactly. They also know on what move the opponent has stood on, but don't have access to anything else about the opponents strategy.
- Private stand: each player does not observe whether the opponent has stood.

The players also have access to some other pieces of information, such as the number of cards that have been drawn so far, and the exact set of card values in a player's hand.

However, we will explain later why these pieces of information are irrelevant to creating a set of strategies that is exhaustive, meaning that adding more strategies or information into the strategy set will not yield a more optimal way to play.

3.2 Enumerate All Possible Strategies

In full generality, a deterministic strategy for a player is a rule that, at each point in time, maps everything they have ever seen (all past cards, actions, and whether they have stood) to either *Hit* or *Stand*. This "full history" strategy space is enormous (potentially on the order of 13^n, where n is the number of moves taken) and would be impossible to work with directly.

In solving the game, we instead restrict to a natural and much smaller class of strategies that satisfy two basic properties:

- Markovian in the totals: The action given by a strategy depends only on the current total and stand information, not on the full past history
- Monotone threshold rules: for a fixed information pattern (e.g. before the opponent stands vs. after they stand), the player hits on low totals and then eventually switches to standing and never switches back.

We now explain and justify these restrictions, and then describe what strategy set following these restriction leaves us with.

3.2.1 Why strategies can ignore the full game history

Because the deck distribution is fixed and does not change as we draw cards, and because the terminal reward g only depends on the final totals and who has busted, the expected value of a state is completely parameterized by the sums of both players c_1, c_2 and wether or not each player has stood, σ_1, σ_2, which we can represent as the tuple, $(c_1, c_2, \sigma_1, \sigma_2)$. If we require strategies to be fixed, then from the simulation's point of view, the process is Markov in $(c_1, c_2, \sigma_1, \sigma_2)$. In other words, conditional on this tuple, the distribution over all future states and the eventual terminal reward is independent of the specific sequence of cards and actions that led there, so our strategy only needs to be parameterized by $(c_1, c_2, \sigma_1, \sigma_2)$.

Therefore, whenever a strategy treats two histories differently that lead to the same $(c_1, c_2, \sigma_1, \sigma_2)$, we can construct an alternative strategy that simply uses the better of the two actions at that state and ignores the rest of the history. This alternative strategy performs at least as well and depends only on the current totals and stand statuses. For our problem, it is therefore without loss of generality to restrict attention to strategies that are functions of current totals (and, in the public game, the opponent's stand flag) only.

This also explains why we do not parametrize strategies by the number of cards drawn. In principle, if we know that exactly k cards have been dealt to the opponent, we can regard their hidden total as the sum of k i.i.d. draws from P, and hence there exists a family of probability distributions over their possible totals after k draws. To turn this into a meaningful posterior belief that reacts to observations, however, one must first specify a prior distribution over the opponent's strategy. Any such rule would be an additional modeling assumption rather than a consequence of the rules of the game. For this reason, we choose not to let strategies depend on the number of cards drawn and instead restrict them to depend only on a player's own current total (and, in the public-stand game, on whether the opponent has stood). This keeps the state space compact while still capturing all information that is relevant under our modeling assumptions.

3.2.2 Why threshold policies are sufficient

We assume both players are rational in the sense that, at any total c, they compare the expected value of standing versus hitting and choose the action with higher expected payoff. Let $\mathbb{E}_{\text{stand}}(c)$ denote the expected payoff from standing on total c, and $\mathbb{E}_{\text{hit}}(c)$ the expected payoff from taking one more card at total c. The latter can be generally decomposed as

$$\mathbb{E}_{\text{hit}}(c) = \mathbb{P}(\text{bust} \mid c, \text{hit}) \cdot \mathbb{E}\big[g(\mathcal{I}_{\text{EoG}}) \mid c, \text{hit, bust}\big] + \mathbb{P}(\text{no bust} \mid c, \text{hit}) \cdot \mathbb{E}\big[g(\mathcal{I}_{\text{EoG}}) \mid c, \text{hit, no bust}\big].$$

When we bust, the conditional payoff $\mathbb{E}[g(\mathcal{I}_{\text{EoG}}) \mid c, \text{hit, bust}]$ is always nonpositive: it is -1 if the opponent does not bust and 0 if both players bust. Thus the bust term makes hitting increasingly unattractive as $\mathbb{P}(\text{bust} \mid c, \text{hit})$ grows with c.

For very small totals c_{low}, we have $\mathbb{E}_{\text{hit}}(c_{\text{low}}) > \mathbb{E}_{\text{stand}}(c_{\text{low}})$, since standing on a very low total almost always loses while hitting offers a substantial chance to improve. For very large totals c_{high} close to the bust threshold B, the bust probability is high and the potential gain from a slightly higher total is limited, so we expect $\mathbb{E}_{\text{hit}}(c_{\text{high}}) < \mathbb{E}_{\text{stand}}(c_{\text{high}})$. We therefore have a monotone risk–reward tradeoff: as c increases, the expected payoff

from hitting, $\mathbb{E}_{\text{hit}}(c)$, decreases, while the expected payoff from standing, $\mathbb{E}_{\text{stand}}(c)$ increases. Under this single-crossing assumption, the curves $\mathbb{E}_{\text{hit}}(c)$ and $\mathbb{E}_{\text{stand}}(c)$ as functions of c intersect at most once. If a rational player ever prefers to stand at some total c (i.e., $\mathbb{E}_{\text{stand}}(c) \geq \mathbb{E}_{\text{hit}}(c)$), then for any higher total $c' > c$ we still have $\mathbb{E}_{\text{stand}}(c') \geq \mathbb{E}_{\text{hit}}(c')$. Consequently, the set of totals at which a player prefers to hit is an interval of the form $\{0, 1, \ldots, T-1\}$ for some threshold T, and the set of totals at which they prefer to stand is $\{T, T+1, \ldots, B\}$. In other words, optimal behavior within our model collapses to a threshold strategy: hit below a cutoff and stand at or above that cutoff for each relevant information pattern.

3.2.3 Strategy set for the private-stand game

In the private-stand game, a player does not know whether the opponent has already stood. The Markov/threshold argument above suggest that each player's strategy can be parameterized by a single integer threshold T.

Concretely, we consider thresholds

$$T \in \{6, 7, \ldots, 21\}.$$

Very small thresholds (e.g. standing on 2) are clearly dominated by standing on a higher value such as 6: they give up almost any chance to improve the total without gaining any advantage in terms of bust probability, so we exclude them for both conceptual and computational reasons.

A private-stand strategy is therefore fully specified by a threshold T: *Stand if the current sum is at least T*

3.2.4 Strategy set for the public-stand game

In the public-stand game, players know when the opponent has stood, so it is natural to allow them to use one threshold before the opponent stands and a (possibly different) threshold afterwards. For example, a player might be willing to take more risk (hit longer) if they know the opponent is already stood at a good total.

We therefore parametrize each strategy by a pair (X, Y) with:

$$X, Y \in \{6, 7, \ldots, 21\},$$

where:

- X is the hit/stand threshold while the opponent has *not* yet stood;
- Y is the hit/stand threshold once the opponent *has* stood.

The decision rule is: *If the opponent has not stood, stand if current sum is at least X. If the opponent has not stood, stand if the current sum is at least Y*

Within the class of strategies that (i) depend only on current totals and the opponent's stand flag and (ii) are monotone in the total in each regime, this family $\{\pi^{(X,Y)}\}$ is exhaustive: any such strategy is uniquely determined by specifying the smallest total at which it starts standing before the opponent stands, and the smallest total at which it starts standing after the opponent has stood.

3.3 Dynamic Programming for the Payoff Matrix

After fixing a finite strategy set for each player (either the thresholds T in the private game or the pairs (X, Y) in the public game), we need the expected payoff to player p_1 for every possible pair of strategies. We do this exactly (no Monte Carlo) by dynamic programming over the state space of totals.

3.3.1 State representation

The simulation maintains, at each step, an internal state $s = (c_1, a_1, c_2, a_2, \sigma_1, \sigma_2)$, where, c_i is the current total of player p_i, a_i is the number of Aces for p_i that are currently being counted as 11 (and hence can be "downgraded" to 1 if needed), and $\sigma_i \in \{0, 1\}$ indicates whether p_i has stood.

The use of a_i is purely to implement the Ace rule correctly. When a card is drawn for player p_i, the code:

1. Adds 11 to t_i and increments a_i if the card is an Ace;
2. Otherwise adds the card's value (2–10) to c_i;

3. While $c_i > B$ and $a_i > 0$, reduces c_i by 10 and decreases a_i by 1.

This is a greedy rule: always treat an Ace as 11 unless doing so would cause you to bust, in which case convert it to 1. This is optimal for our payoff structure because, conditional on a fixed set of cards, the best possible total is always the largest number $\leq B$ that you can get by counting some subset of Aces as 11 and the rest as 1. The algorithm above always produces exactly that maximum feasible total and never chooses a smaller total while staying below B.

3.3.2 Value function and base cases

For a fixed pair of strategies $(\pi_{\text{row}}, \pi_{\text{col}})$ (one for p_1 and one for p_2), we define a value function

$$V(s) = \mathbb{E}\big[g(\mathcal{I}_{\text{End of Game}}) \,\big|\, \text{current state } s,\ \pi_{\text{row}}, \pi_{\text{col}}\big],$$

where the expectation is over all future card draws, assuming both players follow their chosen strategies from state s onward.

The initial payoff entry for this pair of strategies is then $A_{ij} = V(0, 0, 0, 0, 0, 0)$, the expected reward starting from both totals equal to 0 and both players not having stood.

The terminal state occurs when each of the players has either stood or busted, at which point we have:

$$V(s) = g(\mathcal{I}_{\text{End of Game}}),$$

using the reward function already defined in the setup.

3.3.3 Recursive step

If the state is not terminal, each player chooses an action according to their strategy. In the code this is implemented by a helper function that, given:

- The player's policy (threshold T or pair (X, Y)),
- The player's current total
- If the player has already stood

- In the public game, whether the opponent has already stood,

returns either *Hit* or *Stand*. If a player has already stood ($\sigma_i = 1$), their action is forced to be a stand. Given the actions (a_1, a_2), we update the stand flags:

$$\sigma_i' = \begin{cases} 1, & \sigma_i = 1 \text{ or } a_i = \textit{Stand}, \\ 0, & \text{otherwise}, \end{cases}$$

and then transition as follows,

- If both players stand after this update, the next state is terminal and we simply evaluate g at the current totals.
- If exactly one player hits, only that player draws a card. For example, if p_1 hits and p_2 stands, then for each possible card $d \in D$ we compute the updated (t_1', a_1') using the Ace-handling rule, form the next state s', and take the expectation

$$V(s) = \sum_{d \in D} P(d)\, V(s').$$

- If both players hit, then both draw independently from D. We consider all pairs (d_1, d_2), update (t_1', a_1') and (t_2', a_2'), and then compute

$$V(s) = \sum_{d_1 \in D} \sum_{d_2 \in D} P(d_1)\, P(d_2)\, V(s').$$

Since the state space is finite and the game always eventually terminates (a player who never stands will eventually bust), this recursion is well-defined and evaluates $V(s)$ exactly for every state s that can be reached under the given strategies.

3.3.4 Constructing the payoff matrix

For each ordered pair of strategies (π_i, π_j) in our finite strategy set, we call the dynamic program starting from the initial state and record the resulting expected payoff:

$$A_{ij} = V^{(\pi_i, \pi_j)}(0, 0, 0, 0, 0, 0).$$

This produces a payoff matrix $A \in \mathbb{R}^{m \times m}$ where m is the number of strategies we enumerated (e.g. $m = 16$ in the private-stand game and $m = 256$ in the public-stand

game). We can then interpret A_{ij} as the expected reward to player p_1 when p_1 uses π_i and p_2 uses π_j.

3.4 Solving the Game via Linear Programming

Once the payoff matrix $A = (a_{ij})$ is computed, we view the problem as a finite two–player zero–sum game. A mixed strategy for the row player (p_1) is a vector $p = (p_1, \ldots, p_m)$ with

$$p_i \geq 0, \quad i = 1, \ldots, m, \qquad p_1 + \cdots + p_m = 1,$$

and similarly the column player (p_2) uses a mixed strategy $q = (q_1, \ldots, q_n)$ with nonnegative entries summing to one. If the row player plays p and the column player plays q, the expected payoff to p_1 is

$$\mathrm{PAY}(p, q) = p^\top A q.$$

The row player chooses p to maximize her guaranteed expected payoff

$$\max_{p} \; \min_{1 \leq j \leq n} \sum_{i=1}^{m} a_{ij} p_i.$$

Introducing a scalar variable ξ for this guaranteed value, based on the derivation in the notes, we obtain the standard linear program

$$\begin{aligned} \max \quad & \xi \\ \text{subject to} \quad & \xi \;\leq\; \sum_{i=1}^{m} a_{ij} p_i, \qquad j = 1, \ldots, n, \\ & p_1 + \cdots + p_m = 1, \\ & p_i \geq 0, \qquad i = 1, \ldots, m. \end{aligned}$$

In our implementation, we solve this LP (after converting it to the equivalent minimization form required by the numerical solver) to obtain the optimal row strategy $p^\star$ and the game value $\xi^\star$. A symmetric LP, obtained by interchanging the roles of rows and columns, yields the optimal mixed strategy $q^\star$ for the column player.

3.5 Solving the Game Via Strategy Elimination

In addition to the linear programming approach, we also analyze the game by iterated elimination of strictly dominated strategies. Starting from the full payoff matrix A, we repeatedly remove any row or column strategy that is strictly dominated by a mixture of the remaining strategies. This procedure is applied in the same way for both the private-stand (single-threshold T) and public-stand (threshold pairs (X, Y)) games, always working directly with the corresponding payoff matrix.

Iterated elimination never removes a strategy that could appear in a Nash equilibrium of a finite zero-sum game. Therefore, if this process happens to converge to a single remaining strategy (a single threshold $T^\star$ or a single pair $(X^\star, Y^\star)$), that surviving strategy must be the pure equilibrium within our restricted strategy class and will agree with the solution obtained from the linear program.

Results

4.1 Private Stand Game

Below is the payoff matrix for the private stand game generated via dynamic programming. A strategy called x indicates a strategy of *Stand if my current total is at least* x. We then conduct iterative policy elimination on this payoff matrix.

After solving the payoff matrix via both strategy elimination and by solving the LP formulation of the game (we get the same results both ways), we are left with a threshold of 18 as being optimal for both players. This means that **in the private stand game, the optimal strategy for both players is to stand if the player's current sum is at least** 18**. The optimal value of the game is** 0 **for both players.**

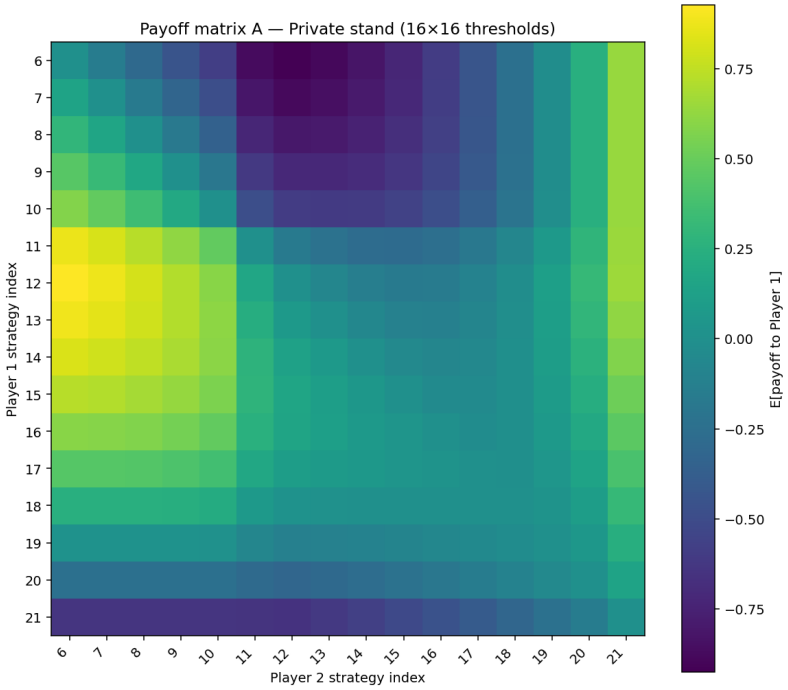


Figure 4.1. Payoff matrix for the private-stand game.

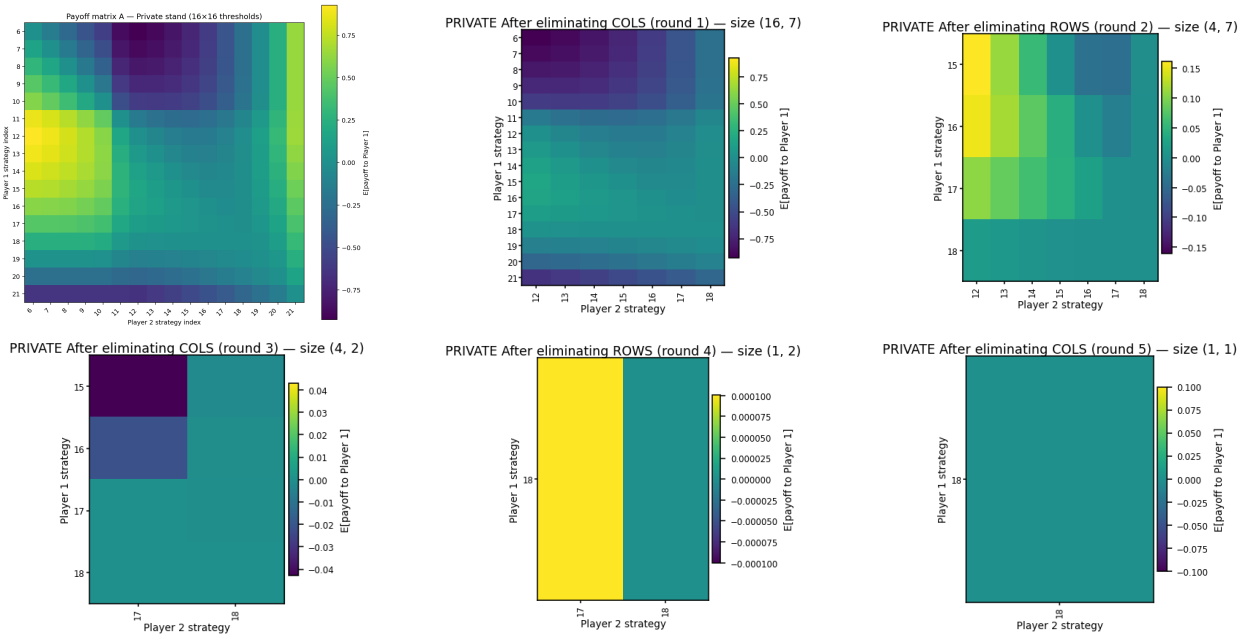


Figure 4.2. Iterative strategy elimination in the private-stand game. Each panel shows the payoff matrix after one round of eliminating strictly dominated strategies, starting from the top left and proceeding left-to-right across the top row, then across the bottom row. The process converges to a single pair of strategies.

4.2 Public Stand Game

Below is the payoff matrix for the public stand game generated via dynamic programming. A strategy called x indicates a strategy of *Stand if my current total is at least* x. We then conduct iterative policy elimination on this payoff matrix.

After solving the payoff matrix via both strategy elimination and by solving the LP formulation of the game (we get the same results both ways), we are left with a threshold of $(17, 18)$ as being optimal for both players. This means that **in the public-stand game, the optimal strategy for both players is to stand at 17 or above while the opponent has not yet stood, and stand at 18 or above once the opponent has stood. The optimal value of the game is** 0 **for both players.**

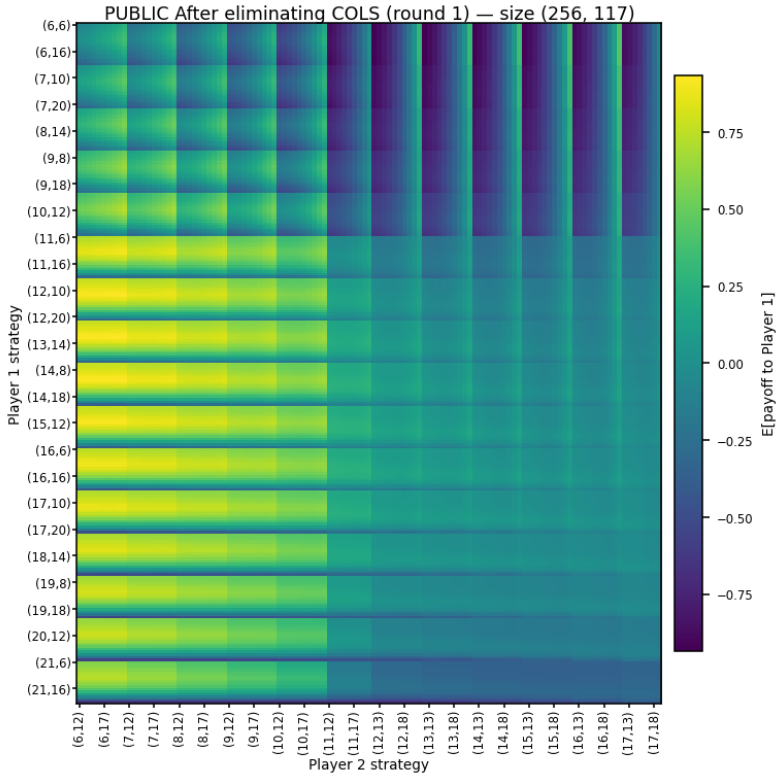


Figure 4.3. Payoff matrix for the public-stand game.

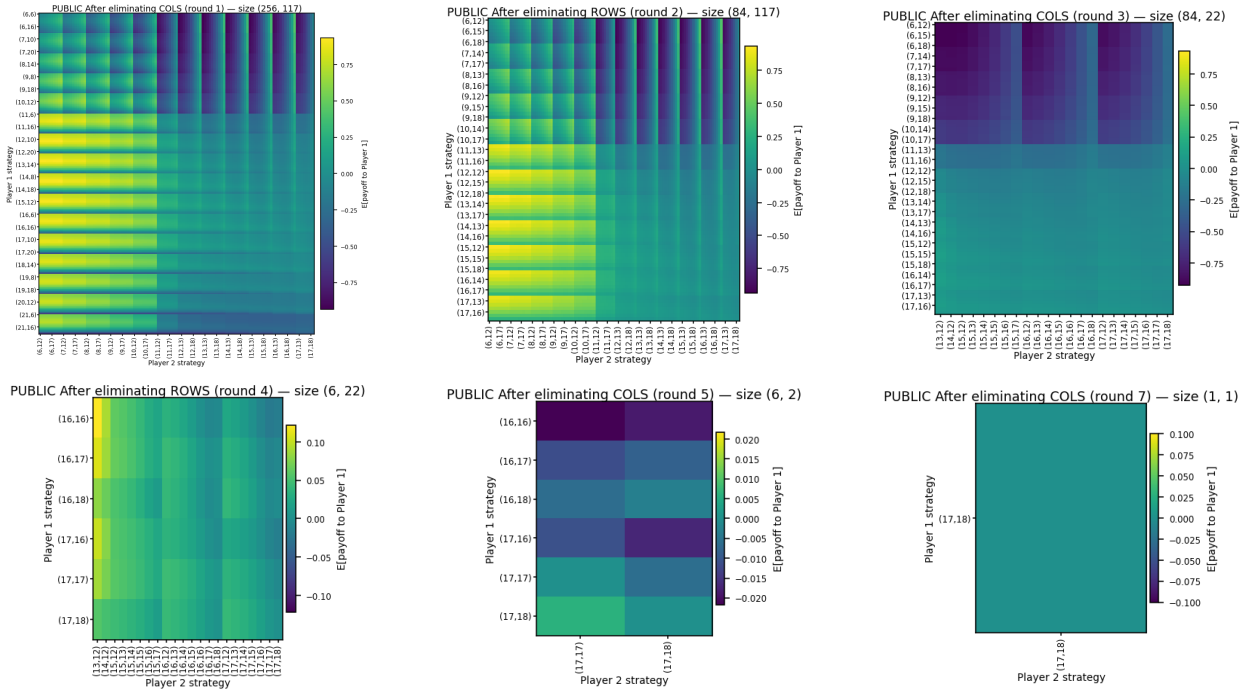


Figure 4.4. Iterative strategy elimination in the public-stand game, with panels formatted in same way as above iteration algorithm.

Discussion

The payoff matrices computed above are *antisymmetric*, in the sense that they satisfy $A^\top = -A$, i.e. the diagonal entries are zero and $A_{ij} = -A_{ji}$ for all i, j. This is exactly what we expect from a symmetric zero-sum game. If both players choose the same strategy, the game is perfectly symmetric, so neither should have an advantage and the expected payoff must be zero, yielding a zero diagonal. More generally, if player p_1 receives payoff A_{ij} when using strategy i against an opponent using strategy j, then by symmetry the situation in which p_1 uses j against i is equivalent to the original game with the players' roles swapped, so player p_1 now receives $-A_{ij}$. This implies $A_{ji} = -A_{ij}$, confirming antisymmetry, which is clearly reflected in the payoff matrices reported in the results section.

During our process of determining an optimal strategy for the game of Blackjack, we utilized techniques from Dynamic Programming and two-player zero-sum games to find an optimal solution for playing the game. Our results also cover more than just one deterministic game where both players see each other's cards, but also the "private"

variation where the other player's cards are entirely unknown to the other. One interesting phenomenon to note with the results of each of these variants is that an ideal payout method uses extremely similar strategies. Since in each game, both players have the same information as each other, the resulting pay-off matrix should always have a result of zero if the players are both playing optimally, which was the assumption our code functions under. When looking at the final results, standing on 18 (or higher) every time for the "private" code was the optimal strategy, and since there is no information outside of what you have, this would not change depending on what another player has. This is quite similar to the "public" variation, where standing at 17 or 18 is optimal, yet it is dependent on whether the other player has stood or not and what value they have. Examining the heatmap of the games when no information is provided to the algorithm and row and column reductions aren't used, reveals a significant amount of data to process and potential choices to make, allowing for the development of an optimal strategy. However, recognizing the fact that some options are strictly worse than others because of column or row dominance allowed us to reduce the payoff matrix over and over until it converged to a concise strategy for each player, with the result providing an expected payoff of 0 for players A and B over a long enough timeline.

Although there are slightly different rules to the game of Blackjack that we were working with, there is some merit in the results we derived. While no, it is not the same as playing against a dealer in a casino, the general rule of thumb for a player going against the house is that they should hit until they get to 17 and typically stand there depending on additional information about the dealer's hand. Our data reaffirms this commonly held piece of knowledge by Blackjack players, despite the circumstances being slightly altered for our purposes.

It's worth noting that the results of the game are in line with what was expected. As the game was a zero-sum symmetrical game between two players, the result is that the strategies are mirrored between the two players. Additionally, as a player's score only goes up as they hit, all odds are known; it's not possible to impact the other player's score, and it's not possible to know what hand the other player stood, even in the public iteration of the game. The game does not have a mixed strategy; instead, we have a symmetrical single

strategy outcome.

Future Work

Several different expansions or improvements could be made with this code or the structure of the game, and the code/methods we used are general enough to apply to a large class of blackjack type problems. There could be multiple players added to the game, there could be additional rules or ways to win, such as a "five card payoff" where hitting until you get five cards while still being under 21 nets you a bonus reward, and countless other alternatives to the rules while still keeping the core tenets of the game the same. In the future, it would be fruitful to examine a list of alterations someone could make to the game and how each of them would ultimately impact the strategy for optimal play or yield the most profitable outcome. Unfortunately, due to a lack of time or the increasing complexity of certain edge cases, it isn't necessarily feasible to continue our research into more and more rule sets.

However, the results of our research have implications outside of the outcome space of the game itself. First of all, this game itself is fairer than the existing blackjack. Given that blackjack games are extremely common even in college settings - anecdotally, after friendly poker games, blackjack would get played among a smaller group of two to four - this adapted version would remove the issues of a "dealer edge" scenario among friends, between the fact that the optimal play is symmetric and the practical, one strong application of the results of this project is that many games involving skill and variance can often be broken down into the same parts. First, there is a zero-sum simple strategy part, which is the blackjack demonstrated in the project. Second, there are tweaks to add variance, both for enjoyment and the benefit of the casino. Many games, not just blackjack, could be adapted in a similar fashion such that they offer a fairer and more competitive environment.

Appendix A

Supporting Python Code

Below is the Python code we used to compute the payoff matricies and solve the games. All code, except functions used to format heatmap plots, that is used to generate results is shown below.

Listing A.1: Code for Python simulation

```
import numpy as np
from scipy.optimize import linprog
import matplotlib.pyplot as plt
import math


CARD_PROBS = {
    1: 1 / 13,  # Ace (handled as 1 or 11)
    2: 1 / 13,
    3: 1 / 13,
    4: 1 / 13,
    5: 1 / 13,
    6: 1 / 13,
    7: 1 / 13,
    8: 1 / 13,
    9: 1 / 13,
    10: 4 / 13,  # 10/J/Q/K
}
CARD_VALUES = list(CARD_PROBS.keys())


def add_card(total, usable_aces, card):
    """
    Update (total, usable_aces) after drawing 'card' where:
      - card==1 means Ace (count as 11 if possible else 1)
      - otherwise card is its pip value (10 covers 10/J/Q/K)
    """
    if card == 1:
        if total + 11 <= 21:
```

```
            total += 11
            usable_aces += 1
        else:
            total += 1
    else:
        total += card

    while total > 21 and usable_aces > 0:
        total -= 10
        usable_aces -= 1

    return total, usable_aces


def act(policy, own_total, opponent_stood_public, observable_stand):
    t_base, t_after = policy
    thresh = t_after if (observable_stand and opponent_stood_public)
       else t_base
    return "S" if own_total >= thresh else "H"


def make_private_policies(thresholds):
    return [(t, t) for t in thresholds]


def make_public_policies(thresholds):
    """
    Use x before opponent stands; use y after opponent stands.
    """
    return [(x, y) for x in thresholds for y in thresholds]


def expected_payoff(policy1, policy2, observable_stand, s1_start=0,
   s2_start=0):
    """
    Returns E[payoff to Player 1] in {-1,0,+1}, starting from totals (
       s1_start, s2_start).
    """

    def terminal(t1, t2, stood1, stood2):
        bust1 = (t1 > 21)
        bust2 = (t2 > 21)

        if bust1 or bust2:
            if bust1 and bust2:
                return 0.0
            return -1.0 if bust1 else 1.0
```

```
    if stood1 == 1 and stood2 == 1:
        if t1 > t2:
            return 1.0
        if t1 < t2:
            return -1.0
        return 0.0

    return None

def V(t1, a1, t2, a2, stood1, stood2):
    tt = terminal(t1, t2, stood1, stood2)
    if tt is not None:
        return tt

    # Public info about "opponent has stood"?
    opp_stood_to_1 = (stood2 == 1) if observable_stand else False
    opp_stood_to_2 = (stood1 == 1) if observable_stand else False

    # If already stood, forced to stand
    a1_move = "S" if stood1 == 1 else act(policy1, t1,
       opp_stood_to_1, observable_stand)
    a2_move = "S" if stood2 == 1 else act(policy2, t2,
       opp_stood_to_2, observable_stand)

    stood1n = 1 if (stood1 == 1 or a1_move == "S") else 0
    stood2n = 1 if (stood2 == 1 or a2_move == "S") else 0

    # both stand now
    if stood1n == 1 and stood2n == 1:
        return terminal(t1, t2, stood1n, stood2n)

    # one hits
    if a1_move == "H" and a2_move == "S":
        out = 0.0
        for c, pc in CARD_PROBS.items():
            nt1, na1 = add_card(t1, a1, c)
            out += pc * V(nt1, na1, t2, a2, stood1n, stood2n)
        return out

    if a1_move == "S" and a2_move == "H":
        out = 0.0
        for c, pc in CARD_PROBS.items():
            nt2, na2 = add_card(t2, a2, c)
            out += pc * V(t1, a1, nt2, na2, stood1n, stood2n)
        return out
```

```
        # both hit
        out = 0.0
        for c1, p1 in CARD_PROBS.items():
            nt1, na1 = add_card(t1, a1, c1)
            for c2, p2 in CARD_PROBS.items():
                nt2, na2 = add_card(t2, a2, c2)
                out += p1 * p2 * V(nt1, na1, nt2, na2, stood1n, stood2n
                    )
        return out

    return V(s1_start, 0, s2_start, 0, 0, 0)


def build_payoff_matrix(policies, observable_stand):
    m = len(policies)
    A = np.zeros((m, m), dtype=float)

    for i in range(m):
        if i % 10 == 0:
          print(f"{i}/{256}")
        for j in range(m):
            A[i, j] = expected_payoff(policies[i], policies[j],
                observable_stand=observable_stand)

    return A


def solve_row_player_lp(A):
    A = np.asarray(A, float)
    m, n = A.shape

    c = np.zeros(m + 1)
    c[-1] = -1.0  # minimize -v

    A_ub = np.zeros((n, m + 1))
    b_ub = np.zeros(n)
    for j in range(n):
        A_ub[j, :m] = -A[:, j]
        A_ub[j, m] = 1.0

    A_eq = np.zeros((1, m + 1))
    A_eq[0, :m] = 1.0
    b_eq = np.array([1.0])

    bounds = [(0.0, None)] * m + [(None, None)]
    res = linprog(c, A_ub=A_ub, b_ub=b_ub, A_eq=A_eq, b_eq=b_eq, bounds
        =bounds, method="highs")
```

```
    return res.x[:m], float(res.x[m])


def solve_col_player_lp(A):
    A = np.asarray(A, float)
    m, n = A.shape

    c = np.zeros(n + 1)
    c[-1] = 1.0

    A_ub = np.zeros((m, n + 1))
    b_ub = np.zeros(m)
    for i in range(m):
        A_ub[i, :n] = A[i, :]
        A_ub[i, n] = -1.0

    A_eq = np.zeros((1, n + 1))
    A_eq[0, :n] = 1.0
    b_eq = np.array([1.0])

    bounds = [(0.0, None)] * n + [(None, None)]
    res = linprog(c, A_ub=A_ub, b_ub=b_ub, A_eq=A_eq, b_eq=b_eq, bounds
       =bounds, method="highs")

    return res.x[:n], float(res.x[n])


# ------------------------------
# Runners
# ------------------------------
def run_private(show_heatmap=True):
    thresholds = list(range(6, 22))  # 16
    policies = make_private_policies(thresholds)

    A = build_payoff_matrix(policies, observable_stand=False,
       skew_symmetrize=True)
    p_star, v = solve_row_player_lp(A)
    q_star, w = solve_col_player_lp(A)

    print("\n" + "=" * 70)
    print("PRIVATE STAND (opponent does NOT know you stood)")
    print("Strategies: stand iff total >= t, for t=6..21 (ace can be 1
       or 11 internally)")
    print(f"Game value: v     {v:.6g} (dual w     {w:.6g})")
    print("Row mix (prob > 1e-3):")
    for t, prob in zip(thresholds, p_star):
```

```
        if prob > 1e-3:
            print(f"  t={t:2d}  prob={prob:.4f}")

    if show_heatmap:
        plot_heatmap(
            A,
            title="Payoff matrix A      Private stand (16 16  thresholds)",
            tick_step=1,
            tick_labels=[str(t) for t in thresholds],
        )

    return A, policies, p_star, q_star, v


def run_public(show_heatmap=True):
    thresholds = list(range(6, 22))   # 16
    policies = make_public_policies(thresholds)  # 256 (x,y)

    A = build_payoff_matrix(policies, observable_stand=True,
        skew_symmetrize=True)
    print("Done building payoff")
    p_star, v = solve_row_player_lp(A)
    q_star, w = solve_col_player_lp(A)

    print("\n" + "=" * 70)
    print("PUBLIC STAND (opponent DOES know you stood)")
    print("Strategies: (x,y) where")
    print("  before opponent stands: stand iff total >= x")
    print("  after opponent stands: stand iff total >= y")
    print("  with x,y in {6..21} (256 strategies), ace=1/11 internally")
    print(f"Game value: v     {v:.6g} (dual w     {w:.6g})")
    print("Row mix (prob > 1e-3):")
    for (x, y), prob in zip(policies, p_star):
        if prob > 1e-3:
            print(f"  (x={x:2d}, y={y:2d})  prob={prob:.4f}")

    if show_heatmap:
        labels = [f"({x},{y})" for (x, y) in policies]
        plot_heatmap(
            A,
            title="Payoff matrix A      Public stand (256 256  (x,y) policies)",
            tick_step=16,
            tick_labels=labels,
        )
```

```
    return A, policies, p_star, q_star, v

def find_dominated_cols(A, tol=1e-12):
    """Dominated columns for the column player (who minimizes A)."""
    A = np.asarray(A, float)
    _, n = A.shape
    dominated = set()
    for j in range(n):
        if j in dominated:
            continue
        for k in range(n):
            if j == k:
                continue
            # col j dominated by col k if A[:,j] >= A[:,k]
               componentwise (and strict somewhere)
            if np.all(A[:, j] >= A[:, k] - tol) and np.any(A[:, j] > A
               [:, k] + tol):
                dominated.add(j)
                break
            # remove exact duplicates too (safe)
            if np.all(np.abs(A[:, j] - A[:, k]) <= tol) and j > k:
                dominated.add(j)
                break
    return dominated


def find_dominated_rows(A, tol=1e-12):
    """Dominated rows for the row player (who maximizes A)."""
    A = np.asarray(A, float)
    m, _ = A.shape
    dominated = set()
    for i in range(m):
        if i in dominated:
            continue
        for k in range(m):
            if i == k:
                continue
            # row i dominated by row k if A[i,:] <= A[k,:]
               componentwise (and strict somewhere)
            if np.all(A[i, :] <= A[k, :] + tol) and np.any(A[i, :] < A[
               k, :] - tol):
                dominated.add(i)
                break
            # remove exact duplicates too (safe)
            if np.all(np.abs(A[i, :] - A[k, :]) <= tol) and i > k:
                dominated.add(i)
```

```
                break
    return dominated


A_private, policies_private, p_private, q_private, v_private =
    run_private(show_heatmap=True)

A_public, policies_public, p_public, q_public, v_public = run_public(
    show_heatmap=True)
```