A Subgame Perfect Nash Equilibria Analysis of Zero-Sum Utility Pairs Card Game

Allen Park Sayeed Tasnim Claus Zheng

December 5, 2014

1 Introduction

Pairs is an easy "press-your-luck" card game which uses a peculiar triangular deck where the number of cards of a particular rank occurs is equal to the rank of the card. Players take turns by choosing to "hit" or "fold" without violating some criteria resembling a structure similar to Black Jack. For example, in Black Jack, the basic aim is to not yield a sum of cards greater than 21. In Pairs, the objective is to not yield two cards of the same rank, also known as catching a pair. Players are penalized for catching pairs.

In this paper, we analyze the game of pairs under general cases. We consider generalizing the deck (an arbitrary number of ranks and an arbitrary number of cards) and the number of players. We use a particular zero-sum utility function reflecting the one loser policy of the game where the loser receives a negative utility proportional to the pair scored and all other players receive positive utility proportional to the pair scored.

We derive results and invariants about the game and the moves of particular strategies that will yield Subgame Perfect Nash Equilibria (SPNE) by considering best responses. With these results, we program computations to solve for the strategies under certain stages of the game. With this computation, we simulate the results playing against each other under different strategies, such as two SPNE strategies playing against each other and a SPNE player playing against a player who plays randomly.

2 Background

2.1 Description of Pairs

Deck The deck of the card game Pairs consists of a triangular deck. The deck contains 10 ranks of cards from 1 to 10. There exists one card with rank 1, two cards with rank 2, 3 cards with rank 3, and so on until there are ten cards of rank 10. There are no additional features other than the ranks of the cards. Hence, the deck consists of 55 cards.

Set-Up The deck is shuffled and five cards are *burned* and discarded, facedown, into the middle of the table. This begins the separate discard pile. Each time the deck is reshuffled from the discard pile, another set of five cards will be burned again to prevent players from counting cards.
At the beginning of a round, one card is dealt faceup to each player. The player with the lowest card will go first. To break ties for low, if two or more players are tied for low, additional cards are dealt to the tied players and those cards as used as tiebreakers. If the tiebreakers are tied, more cards are continued to be dealt until there is one unique player with the lowest card. If someone gets a pair during the tiebreaking process, the dealt paired card is discarded and another card is dealt as a replacement. A player is not penalized by a pair on the deal for the set-up, but players may wind up with several extra cards.
Turn On a player's turn, the player has two choices. The player may **hit** (take a card) or **fold**. If a player receives a pair of cards (two cards of the same rank), or folds, the round is over and the player scores points. Otherwise, play passes to the left.
Pairing Up When a player hits as his or her move, the player aims not to get a pair (any two cards of the same rank). If the player pairs up, then the player scores the number of points of the denomination of the pair. One card of the pair is kept to the side, faceup, to maintain score. For example, if a player catches a pair of 8's, the player scores 8 points, and sets aside one of the eights to keep score.
Folding A player can fold or surrender as his or her move. Instead of taking a card, the player takes the *lowest faceup card in play* on the table (cards set aside to count for score are not included). The taken faceup card is kept aside to maintain points. This card may be chosen from any player's stack, not just his or her own. Folding may sometimes be better than hitting, based on the probability of catching a pair.
Ending the Round As soon as one player catches a pair or folds, the round is over. All the cards in play are discarded, facedown into the middle of the table, and another round is dealt. The players keep the scoring cards aside, faceup to maintain points and will not return to the deck until the game is over.
Reshuffling When the deck runs out, discard pile is reshuffled and used as the deck. Five cards are burned again to make counting cards a bit more difficult. If there are fewer than five cards after the burned cards, discontinue play, terminate the game, and tally the points.
Losing the Game In the traditional game, there is no winner, just one loser. The game ends when one player reaches the target score. For example in a 4-player game, the loser is the first person to score 16 points. The target score is given by the formula $\lfloor \frac{60}{m} \rfloor + 1$ where m is the number of players in the game.

2.2 Definitions

Throughout the remainder of this paper, let n be the rank of the highest card in the deck we are considering, and let m be the number of players. This allows us to analyze more general decks other than the standard triangular Pairs deck in which $n = 10$.

Definition 1. A **deck** is an n-dimensional vector $d \in \mathbb{Z}^n$. We denote the jth element of a deck d as $d[j]$. For a deck d and an integer $q \in \mathbb{Z}$, we use the notation $d + q\langle j \rangle$ to the refer to the deck in which the jth element of d is incremented by q, i.e. $(d + q\langle i \rangle)[j] = d[j] + q\delta_{ij}$.

We use the abstract deck d to represent a physical deck with $d[i]$ cards of rank i. Then, $d - \langle i \rangle$ represents removing a card of rank i from the deck. For example, if a player hits from a deck d and receives a card of rank i, the new deck is $d - \langle i \rangle$.

Definition 2. The **normalized deck** of a deck $d \neq 0$ is the vector $d / \left(\sum_j d[j] \right)$. We denote the normalized deck of a deck $d \neq 0$ as $\tilde{d}$.

We can think of a normalized deck $\tilde{d}$ as a probability distribution $p(j) = \tilde{d}[j]$, where $p(j)$ represents the probability of drawing a card of rank j from a deck d uniformly at random. It is easily seen from the definition of a normalized deck that $p(j)$ corresponds to a valid probability distribution.

Definition 3. A **stack** is an n-dimensional vector $s \in \{0, 1\}^n$. For a stack s and an index j for which $s[j] = 0$, we use the notation $s + \langle j \rangle$ to refer to the stack in which the jth element of s is incremented by 1.

We use a stack s to represent the faceup cards of a player, where $s[j] = 1$ if and only if the player a faceup card of rank j. Because whenever a faceup pair occurs, either the dealt card is discarded (in the case of dealing to determine who goes first) or the player loses (in the case hitting), a stack s is sufficient to characterize any player's hand at any point in the game.

Definition 4. Let $D = \mathbb{Z}^n$ be the set of possible decks. Let $S = \{0, 1\}^n$ be the set of possible stacks. Then, a **configuration** of the game is a tuple $(d, \bar{d}, s_1, s_2, \ldots, s_m)$, where $d, \bar{d} \in D$ and $s_i \in S$ for $i \in \{1, 2, \ldots, m\}$.

For a configuration $(d, \bar{d}, s_1, s_2, \ldots, s_m)$, the deck d represents the true deck and the possible cards that may be dealt to players during the set-up phase of a round or during their turn. The deck $\bar{d}$ represents the discard deck, the set of cards that are occasionally discarded.

2.3 Formal Game Formulation

We formalize the general n-rank m-player game. To generalize in an extensive form considers a very large number of possibilities and situations so we describe the situations rather than giving a formal mathematical description of all of the possibilities.

First we discuss the effect of burning cards in our representation of our game. Suppose we burn k cards. In the traditional game, $k = 5$. We can model face-down burned cards as cards that remain in our deck by symmetry of the random selection of k cards to burn. Without loss of generality, we may let k cards be burned cards be the last k cards in our deck. Hence, we can reformulate our reshuffling condition. The deck is reshuffled when one of the following three cases occurs: a player catches a pair, a player folds, or the deck has k or fewer cards left to represent the burned cards so the deck needs to be repopulated with the discard pile $\bar{d}$. The first two cases to end rounds will be clear in our analysis. The last case is subtle and is expressed by $\sum_{j=1}^{n} d[j] \leq k$. Note that the case where fewer than k cards cannot occur because the cards are dealt one at the time and the case where the deck has k cards will occur and trigger either a reshuffling of the deck or a termination of the game before a case where the deck has more than k cards occurs. Hence, we simplify our condition to $\sum_{j=1}^{n} d[j] = k$.

2.3.1 Players

There will be m players in the game represented as the set M that will be given some order $\{1, \ldots, m\}$. This will represent the m players sitting around in a circle in a cyclic order where the next player is sitting to the person's left.

2.3.2 Actions

Each player will have the same possible set of actions of hit and fold for each node and stage of the game. Let A be the set of actions. Hence, A is defined as

$$A = \{\text{Hit}, \text{Fold}\}$$

2.3.3 Information Set

We need to describe the progression of the game under the course of actions by the players. We can divide the game into 'rounds'. Each round will begin with the set-up phase and will terminate under one of the following three cases: a player catches a pair, a player folds, or the deck cannot be reformed with the discard pile, i.e. $\sum_{j=1}^{n}(d + \bar{d})[j] \leq k$. Over the progression of the game, each node is described as the current configuration of the game $(d, \bar{d}, s_1, \ldots, s_m)$ and all past configurations of the game during each person's turn to characterize all of the information a player has. This includes all of the hidden information as well such as the order of the deck.
Set-Up The deck is shuffled at the beginning of the set-up phase. At the beginning of a round, the current node has a large number of children. Each child represents one of the potential permutation orders of the newly shuffled deck. Each permutation order is equally likely. Nature randomly selects one of the possible permutation orders to be the true order of the deck during the round. Each of these possible orders is assumed to be equally likely as the belief for each player to be consistent with nature.
Then the dealing to determine the first player for the initialization game begins. As cards are dealt, the deck is decremented and the corresponding stack is incremented appropriately. For each configuration, the order of the remaining

cards of the deck will each be equally likely and the players will also assume this as their beliefs while cards are revealed. They will, however, know the information of which cards are now part of stacks and in which order they were played. The progression of the nodes to which player is dealt another card follows exactly as in the normal rules for Pairs.
When additional cards are dealt to determine tie-breakers and pairs are formed, then the paired cards are sent to the discard pile. Note that this is the only situation cards will be sent to the discard pile (otherwise, cards move from the deck into stacks and from stacks into set aside cards to tally points). Again, the same symmetric probabilistic beliefs of the remaining cards of the deck are assumed for each player.
Turn During a player's turn, a player has the same set of actions A, to hit or to fold. Given the order of the deck, there is only one unique ordering of the deck after a player hits and does not catch a pair: the top card of the deck is removed and the deck is the tuple of all the remaining cards. For each information set with the same history and current configuration, each order of the remaining deck is equally likely and every player's belief will be consistent with these probabilities. If the player hits and catches a pair, folds, or the deck has k cards and the discard pile has a positive number cards (i.e. there are more than k cards in the deck and discard pile to reshuffle and continue play), then the deck is reshuffled and an intermediate node where nature shuffles the deck occurs. If the deck has k cards and the discard pile has no cards, then the game ends because it reaches the terminating condition. Note that this is the only terminating condition of the game. Intuitively, the players play until the deck effectively runs out of cards where there aren't k cards to burn and another card to continue play. The same consistent symmetric beliefs of the order of the cards of the deck hold as in the set-up phase.

2.3.4 Utility

We ignore the actual case of the game where there is a target score and players are trying to maintain a score less than the target score. Instead, the game runs to completion until the cards of the deck effectively run out. We use a casino-style variant of the game where the players pay every other player the number of points scored. Hence, the utility of a particular player i is the sum of utilities over all of the rounds of the game. For each round of the game, the player i yields a utility of $-(m-1)p$ where p is the score player i gets by either catching a pair or folding and a utility of p where p is the score another player $i' \neq i$ receives by catching a pair or folding. The last round where the game terminates due to insufficient cards in the deck yields 0 utility for all players.

3 Analysis of the Two-Player Case

We begin by considering a single round of Pairs with two players. This means that the game terminates when someone catches a pair, folds, or the deck runs out. This means that play does not continue after one of these events as in the complete game until the deck runs out. In this single round game, the score is the rank of either the paired card or the penalty card taken for folding, and we take the payoffs to be negative of the score for the losing player, the one that folds or forms a pair, and positive of the score for the winning, other player. We can think of these payoffs corresponding to the losing player paying the winning player the score that the losing player accumulated in the round, so over many rounds, the overall winner has gained the difference in score from the losing player.

3.1 Two-Player Single Round Analysis

For simplicity, we assume that whenever Fold and Hit at some configuration are expected to yield the same payoff, the players prefer Fold. This allows us to avoid bothersome edge cases in which a player faced with both choices yielding the same payoff might change his action depending on which history led him to the current configuration of the game. Under this assumption, we can prove the following theorem.

Theorem 1. In a subgame perfect equilibrium, a player's action at each decision node can depend only on the configuration of the game.

Proof. For any configuration c, let $k_c = \sum_i \sum_j s_i[j]$, where s_i are the stacks of c, denote the total number of cards among all of the players' faceup cards. Because $s_i[j] \in \{0, 1\}$ for each component, we have that $k_c \leq mn$, which is bounded, and it follows that there are finitely many possible values for k_c.

Suppose for contradiction that we have some subgame perfect equilibrium in which there exist two decision nodes x and y which have the same configuration c, but the player's action at the two nodes differ, and assume without loss of generality that x and y are chosen so that the associated k_c is as large as possible. Then, the possible actions for the player to move are Hit or Fold. In this case of Fold, the player takes the lowest-rank faceup card, which is encoded in the stacks of the configuration, and the resulting payoffs at either x or y are the same.

In the case of Hit, the probabilities of drawing each card are encoded in the deck of the configuration, so they are the same at either x or y. After choosing Hit, suppose that the player receives a card of rank j. Then, either the player has formed a pair and received a payoff of $-j$, or the player has avoided forming a pair, moving the game to a new decision node x' or y'. Note that both x' and y' will have the same configuration c' with $k_{c'} = k_c + 1 > k_c$, and because the number of faceup cards is nondecreasing throughout a round, we know that the players' strategies from here on only depend on the configuration of the game at each node, so the payoffs from the subtrees rooted at x' or y' depend only on c' and are thus equal. Thus, the payoffs from receiving any card from Hit are the same at both x and y, so the expected payoffs from Hit at x and y are also the same.

But if the expected payoffs at x and y are the same for each action, then the player would have chosen the action resulting in the higher payoff, and if both actions result in the same payoff, the player would have chosen to Fold, so it cannot be that the action at x and y, which have the same configuration, differ, which is a contradiction. $\square$

Corollary 2. Let C be the set of possible configurations. Then, any subgame perfect equilibrium can be described by a function $a : C \to \{\text{Hit}, \text{Fold}\}$. From any decision node, the expected payoff of the moving player from following the subgame perfect equilibrium given by a can be described by a function $u : C \to \mathbb{R}$.

Proof. Follows immediately from Theorem 1. $\square$

Using Corollary 2, we can find a simple recursive formula for finding a subgame perfect equilibrium.

Theorem 3. Under a subgame perfect equilibrium defined by Corollary 2, we have

$$u(d, \bar{d}, s_1, s_2) = \max\{u_{\text{Fold}}(d, \bar{d}, s_1, s_2), u_{\text{Hit}}(d, \bar{d}, s_1, s_2)\} \tag{1}$$

where the expected payoff from Fold is

$$u_{\text{Fold}}(d, \bar{d}, s_1, s_2) = -\min\{j \text{ such that } s_i[j] = 1 \text{ for some } i\} \tag{2}$$

and the expected payoff from Hit is

$$u_{\text{Hit}}(d, \bar{d}, s_1, s_2) = -\sum_{j=1}^{n} \left(j s_1[j]\tilde{d}[j] + u(\hat{d}_j, \hat{\bar{d}}_j, s_2, s_1 + \langle j \rangle)(1 - s_1[j])\tilde{d}[j] \right). \tag{3}$$

where $\hat{d}_j$ represents the deck after a card is dealt or after a reshuffling defined as

$$(\hat{d}_j, \hat{\bar{d}}_j) = \begin{cases} (d - \langle j \rangle, \bar{d}) & \text{if } \sum_{j=1}^{n} d[j] > k + 1 \\ (d + \bar{d} - \langle j \rangle, 0) & \text{if } \sum_{j=1}^{n} d[j] = k + 1 \text{ and } \sum_{j=1}^{n} \bar{d}[j] > 0 \end{cases} \tag{4}$$

In the new representation of the deck and the discard pile, the first case represents a card being dealt when there still remain cards to be dealt to the next player and the second case considers reshuffling the discard pile back into the deck.
In the first case, if there are more than $k+1$ cards in the deck, there are at least the last k cards to represent the burned cards and at least two additional cards, one deal to the current player and one more to deal to the next player on the next turn so a reshuffling is unnecessary. The discard pile remains the same.
For the second case, if we have $k + 1$ cards in the deck, one card will be dealt to the current player and the last k cards will represent the burned cards so the deck will need to be reshuffled for the next turn of the next player. The discard

pile is shuffled back into the deck so it is added to the current deck with one card of rank j removed. Note that the case where fewer than $k+1$ cards cannot occur because the cards are dealt one at a time and the case where $k+1$ cards will occur first and will trigger a shuffle. Also, if the deck has $k+1$ cards and the discard pile has no cards, then the game terminates after card j is dealt because there are no more than k cards to burn.

To prove this theorem, we will first need to prove two lemmas showing that the expected payoffs from Fold and Hit are indeed given by 2 and 3.

Lemma 4. Under a subgame perfect equilibrium defined by Corollary 2, the expected payoff from Fold is given by Equation 2.

Proof. After folding, the folding player chooses a faceup card of rank j from the table and receives a payoff $-j$. Clearly, he chooses the lowest-rank faceup card, which is given by the min expression in 2, and it follows that his (expected) payoff is given by Equation 2. □

Lemma 5. Under a subgame perfect equilibrium defined by Corollary 2, the expected payoff from Hit is given by Equation 3.

Proof. After hitting, the hitting player receives a card at random, and the probability of receiving a card of rank j is given by $\tilde{d}[j]$. Suppose that the hitting player receives a card of rank j. If he has a card of rank j among his faceup cards, i.e. $s_1[j] = 1$ and $1 - s_1[j] = 0$, then he has formed a pair and receives a payoff of $-j$. If instead he does not have a card of rank j among his faceup cards, i.e. $s_1[j] = 0$ and $1 - s_1[j] = 1$, then he does not form a pair, and the new configuration of the game is $c = (\hat{d}_j, \hat{\hat{d}}_j, s_2, s_1 + \langle j \rangle)$ as the deck has lost a card of rank j, the stack of the moving player has gained a card of rank j, the stack of the other player has not changed, and now it is the turn of the other player. Because this is a zero-sum game, the expected payoff of the player who just moved is then $-u(c)$. We see that the payoffs of both cases are encoded in the expression

$$-\Big(js_1[j] + u(\hat{d}_j, \hat{\hat{d}}_j, s_2, s_1 + \langle j \rangle)(1 - s_1[j])\Big)$$

as exactly one of $s_1[j]$ and $1 - s_1[j]$ is one, while the other is zero, so only the appropriate term contributes to the payoff. Finally, it follows that the expected payoff from Hit is the above expression weighted appropriately by the probability $\tilde{d}[j]$ of receiving a card of rank j, which is the expression given in 3. □

With these two lemmas, we are now in a position to prove Theorem 3.

Proof of Theorem 3. With Lemmas 4 and 5, we see that the expected payoffs from Fold and Hit are indeed given by Equations 2 and 3, and because at every node in a subgame perfect equilibrium, the player chooses the action that results in the highest expected payoff, his resulting payoff is then given by the maximum of the two expected payoffs, as in Equation 1, completing the proof. □

Corollary 6. Under a subgame perfect equilibrium defined by Corollary 2, we have

$$a(d, \bar{d}, s_1, s_2) = \begin{array}{l} \text{Fold if } u(d, \bar{d}, s_1, s_2) = u_{\text{Fold}}(d, \bar{d}, s_1, s_2) \\ \text{Hit otherwise.} \end{array} \tag{5}$$

Proof. Follows from players choosing the payoff-maximizing action, preferring Fold when expected payoffs are equal. □

Corollary 7. The subgame perfect equilibrium given by Equation 5 is unique.

Proof. Consider the recurrence relation of Equation 1. Note that the only recursive term comes from Equation 3. Consider the expression $\sum_i \sum_j s_i[j]$, which is an integer between 0 and mn and increments by 1 with each step of the recurrence. Once $\sum_i \sum_j s_i[j] = mn$, the second term of Equation 3 is 0 as $1 - s_i[j] = 0 \;\forall\, i, j$, so the recursion has limited depth of mn. Furthermore, Equation 3 is still well-valued when this occurs, so the boundary conditions of the recurrence are "built in" with Equation 3. Thus, Equation 1 uniquely determines $u(d, \bar{d}, s_1, s_2)$.

Corollary 2 shows that every subgame perfect equilibrium must have the same indicated form $a(d, \bar{d}, s_1, s_2)$. Corollary 6 constructs a unique strategy of the indicated form $a(d, \bar{d}, s_1, s_2)$ from $u(d, \bar{d}, s_1, s_2)$. The claim immediately follows. □

According to Corollary 7, Theorem 3 and Corollary 6 are sufficient to calculate the unique subgame perfect equilibrium, which we will do in the next section with the aid of a computer. But first, we prove an additional result that will help to reduce the runtime of our computation.

Corollary 8. Under a subgame perfect equilibrium defined by Corollary 2, if

$$-\sum_{j=1}^{n} \Big(j s_1[j] \tilde{d}[j] \Big) \le \sum_{j=1}^{n} \Big(u_{\text{Fold}}(d, \bar{d}, s_2, s_1)(2 - s_1[j]) \tilde{d}[j] \Big)$$

then $u(d, \bar{d}, s_2, s_1) = u_{\text{Fold}}(d, \bar{d}, s_2, s_1)$ and $a(d, \bar{d}, s_2, s_1) = \text{Fold}$.

Proof. Note that

$$\begin{aligned} u(\hat{d}_j, \hat{\bar{d}}_j, s_2, s_1 + \langle j \rangle) &\ge u_{\text{Fold}}(\hat{d}_j, \hat{\bar{d}}_j, s_2, s_1 + \langle j \rangle) \\ &= -\min\{j' \text{ such that } s_2[j'] = 1 \text{ or } (s_1 + \langle j \rangle)[j'] = 1\} \\ &\ge -\min\{j' \text{ such that } s_2[j'] = 1 \text{ or } s_1[j'] = 1\} \\ &= u_{\text{Fold}}(d, \bar{d}, s_1, s_2). \end{aligned}$$

Then, we have

$$\begin{aligned}
u_{\text{Hit}}(d,\bar{d},s_1,s_2) &= -\sum_{j=1}^{n}\Big(js_1[j]\tilde{d}[j] + u(\hat{d}_j,\hat{\bar{d}}_j,s_2,s_1+\langle j\rangle)(1-s_1[j])\tilde{d}[j]\Big)\\
&\le -\sum_{j=1}^{n}\Big(js_1[j]\tilde{d}[j] + u_{\text{Fold}}(d,\bar{d},s_2,s_1)(1-s_1[j])\tilde{d}[j]\Big)\\
&\le \sum_{j=1}^{n}\Big(u_{\text{Fold}}(d,\bar{d},s_2,s_1)(2-s_1[j])\tilde{d}[j] - u_{\text{Fold}}(d,\bar{d},s_2,s_1)(1-s_1[j])\tilde{d}[j]\Big)\\
&= \sum_{j=1}^{n}\Big(u_{\text{Fold}}(d,\bar{d},s_2,s_1)\tilde{d}[j]\Big)\\
&= u_{\text{Fold}}(d,\bar{d},s_2,s_1).
\end{aligned}$$

And the claim follows. □

Intuitively, a player's expected payoff is at least the payoff he gets from Fold as he would otherwise not be playing a best response. Furthermore, the disutility of Fold is given by the smallest rank faceup card, which can only decrease as the round progresses, so we can bound the payoff a player can get from Hit and successfully force the other player to make a move. Corollary 8 states that if the expected loss of pairing up from choosing Hit is too large, i.e. outweighed by the possible return from getting the other player to make a move discounted by the certain outcome from Fold, then the player should choose Fold. Note that Corollary 8 allows us to avoid having to recurse in some cases, reducing computation time.

3.2 Two-Player Complete Game Analysis

Now we consider the concatenation of these rounds until the deck effectively runs out. The strategies for the players may vary slightly as they have to continue play until many of the cards are set aside to be counted for points. However, we show that this case results in the same strategy profile as the single round game.

The same argument in Theorem 1 holds for the complete game just as it did for the single round game. During a decision, a player has the option to fold or to hit. The utilities for the current round are given in the analysis above. The utilities also need to add the utility of the future game where the new initial deck consists of the current deck, discard pile, and stacks without the card set aside for points. However, this new utility is 0 by the zero-sum property of the utilities and the symmetry of all the players. When some player i' scores some number of points j, a particular player i scored points with a probability of $\frac{1}{2}$ when $i' = i$ and does not score points with a probability of $\frac{1}{2}$ when $i' \neq i$ so the expected utility is $\frac{1}{2}(-j) + \frac{1}{2}(j) = 0$.

Hence, the players are indifferent to playing another game. A player thus maximizes the utility for the current round and this will result in maximizing the utility for the complete game.

4 Analysis of the m-Player Case

We now consider the more general case in which there are m players. As before, we first consider the single-round game, and we take the payoffs to be $-(m-1)$ times the score for the losing player and positive the score for the other players. Note that for $m = 2$, these payoffs reduce to the same payoffs we considered for the two-player case. For the general case, we can think of these payoffs corresponding to the losing player paying every other player the score that the losing player accumulated in the round, so over many rounds, the overall loser has paid each player the difference in their scores.

4.1 m-Player Single Round Analysis

As before, we assume for simplicity that whenever Fold and Hit at some configuration are expected to yield the same payoff, the players prefer Fold, and this allows us to restrict ourselves to strategies in which the action at each decision node depends only on the current configuration of the game. Note that Theorem 1 applies without modification, and the only change to the proof is that the payoff from pairing up from Hit is $-(m-1)j$ instead of j, but the same proof otherwise holds.

Although Corollary 2 is still true, we will modify it to facilitate the following analysis.

Corollary 9. Let C be the set of possible configurations. Then, any subgame perfect equilibrium can be described by a function $a : C \to \{\text{Hit}, \text{Fold}\}$. From any decision node, the expected payoffs from following the subgame perfect equilibrium given by a can be described by a function $u : C \to \mathbb{R}^m$, where $u(d, \bar{d}, s_1, s_2, \ldots, s_m) = (p_1, p_2, \ldots, p_m)$ indicates an expected payoff of p_i for the player with the stack s_i.

Proof. Follows immediately from Theorem 1. □

The only difference we made from Corollary 9 is that u is now an m-dimensional vector. In the two-player case, it was sufficient to know only the payoffs of one of the players because we could infer the payoff of the other player using the zero-sum property of the game. In the m-player case, however, we need to maintain each of the payoffs in order to get a full description of the payoffs.

With this new form of the payoffs, the analog to Theorem 3 is given by the following theorem.

Theorem 10. Under a subgame perfect equilibrium defined by Corollary 9, we have

$$\begin{aligned} u(d,\bar{d},s_1,s_{-1}) &= u_{\text{Fold}}(d,\bar{d},s_1,s_{-1}) \text{ if } u_{\text{Fold}}(d,\bar{d},s_1,s_{-1})[1] \geq u_{\text{Hit}}(d,\bar{d},s_1,s_{-1})[1] \\ &= u_{\text{Hit}}(d,\bar{d},s_1,s_{-1}) \text{ otherwise} \end{aligned} \tag{6}$$

where the expected payoff from Fold is

$$\begin{aligned} u_{\text{Fold}}(d,\bar{d},s_1,s_2) &= (-(m-1)r, r, r, \ldots, r) \\ r &= \min\{j \text{ such that } s_i[j] = 1 \text{ for some } i\} \end{aligned} \tag{7}$$

and the expected payoff from Hit is

$$\begin{aligned} u_{\text{Hit}}(d,\bar{d},s_1,s_2) = \sum_{j=1}^{n} &(-(n-1)j, j, j, \ldots, j) s_1[j]\tilde{d}[j] \\ &+ u_+(\hat{d}_j, \hat{\bar{d}}_j s_2, s_3, \ldots, s_m, s_1 + \langle j \rangle)(1 - s_1[j])\tilde{d}[j]. \end{aligned} \tag{8}$$

where u_+ refers to the vector cyclically shifted to the right by one element to rotate the utilities from the perspective of the next player, Player 2, to the perspective of the current player, Player 1. The same representation of the new deck and discard pile $(\hat{d}_j, \hat{\bar{d}}_j)$ after dealing a card of rank j and reshuffling is necessary is used as in Equation 4.

The theorem follows with the same reasoning and lemmas as those used in the 2-player case. The main difference, again, is to consider a tuple of the payoffs for the expected utilities. The actions also follow the same result because a player takes the particular action that yields the higher expected utility and defaults to Fold if they are the same. This also implies that the SPNE is unique for the m-player case.

4.2 m-Player Complete Game Analysis

We repeat a similar analysis to demonstrate that maximizing the expected utility for the current round yields maximizing the expected utility over the complete game. During a decision, a player has the option to fold or to hit with the associated single round utilities. The utilities also need to add the utility of the future game where the new initial deck consists of the current deck, discard pile, and stacks without the card set aside for points. However, this new utility is 0 by the zero-sum property of the utilities and the symmetry of all the players. When some player i' scores some number of points j, a particular player i scored points with a probability of $\frac{1}{m}$ when $i' = i$ and does not score points with a probability of $\frac{m-1}{m}$ when $i' \neq i$ by symmetry so the expected utility is $\frac{1}{m}(-(m-1)j) + \frac{m-1}{m}(j) = 0$.

Hence, the players are indifferent to playing another game. A player thus maximizes the utility for the current round and this will result in maximizing the utility for the complete game.

5 Computational Analysis

To test our analysis, we implemented the game of Pairs, a player that implements the SPNE analysis above, and several other players.

5.1 Implementation of Pairs

We implemented the game of Pairs as a Python program. The Pairs code is included in appendix 7.1. The program randomly generates a deck and takes player classes as parameters. A base player class that takes user input is included in appendix 7.2.

The program plays the standard version of the game described in section 2. The main difference from our analyzed game is that while the standard game ends when any player reaches a score of $\frac{60}{m} + 1$, our analyzed game ends when the deck runs out. Although they are not the same utilities for the game, these are the similar games because, as we argued in our previous analysis, each round is independent of all other rounds when their current point values are far from the target score. Therefore, the same moves we might take to avoid a score of $\frac{60}{m} + 1$, we would also take similar moves to minimize our score before the deck runs out.

5.2 Implementation of Players

In order to test our analysis, we implemented several players to test against our SPNE player. Overall, we implemented a two-player SPNE player, a m-player SPNE player, a random player, a user player, and a player that uses a heuristic recommended on `http://cheapass.com/node/142`. The code for these are included in Appendices 7.3, 7.4, 7.5, 7.2, and 7.6 respectively.

We implemented a two-player SPNE player separately from the m-player SPNE player since Corollary 8 gives us an early fold condition in the two-player case that allows a drastic speed-up.

The heuristic is applied on every turn. Follow these rules in order:

1. If folding would put you over the limit, hit.
2. If gaining the average of your cards would put you over the limit, fold.
3. If your score is greater than $18x$ the fold value, fold.
4. Otherwise, hit.

5.3 Implementation of the SPNE Analysis

In order to test our analysis in real play, we implemented two programs to generate the SPNEs specified by the analysis. The first program implements a two-player SPNE analysis as in Theorem 3 and the second program implements a m-player SPNE analysis as in Theorem 10. The code for the two-player

1\2	Two-player	m-player	Random	Heuristic
Two-player	4997\5003	5352\4648	495\9505	4791\5209
m-player	-	5000\5000	454\9546	4684\5316
Random	-	-	5028\4972	9534\466
Heuristic	-	-	-	5027\4973

Table 1: Loss rates of one vs. one match-ups over 10,000 games.

m/R/R	m/H/R	m/H/H
320/5983/3697	555/991/8454	3255/3327/3418

Table 2: Loss rates of three player match-ups over 10,000 games.

SPNE analysis is included in Appendix 7.7 and the code for the m-player SPNE analysis is included in Appendix 7.8.

When we calculated a utility for a configuration, we had to recursively calculate the utility for future configurations. Because of the recursive nature of our program, we implemented the SPNE analysis with dynamic programming. We saved a utility for each configuration, a deck and stacks, for which we calculated a final utility. Because we often reached future configurations for which we had already calculated a final utility, the dynamic programming allowed for a much faster calculation of decisions.

5.4 Results of 2-Player Tests

In Table 1, we show the results of our playing from one vs. one match-ups over 10,000 games. The table contains the number of losses for each player. Because of the nature of the game, we recorded losses rather than victories.

The table shows that every match-up between the same players was mostly even, as expected. The random player consistently did very poorly against the two-player SPNE player, the m-player SPNE player, and the heuristic player. Otherwise, the two-player SPNE player and the m-player SPNE player seemed to perform similarly, as expected.

We showed that the SPNE players did win consistently over the other players. The heuristic player did surprisingly well, but it still lost overall against the SPNE players.

5.5 Results of m-Players Tests

In Table 2, we show the results of a three player match-up over 10,000 games. The first column has an m-player SPNE player vs. a random player vs. a random player. The second column has a m-player SPNE player vs. a heuristic player vs. a random player. The third column has a m-player SPNE player vs. a heuristic player vs. a heuristic player.

In all games played by the m-player SPNE player, the SPNE player won. In all games involving a random player, the SPNE player won by a large margin.

Against two heuristic players, the SPNE player had a harder time, but still was able to have a better loss ratio than the heuristic players.

6 Conclusion

In our paper, we were able to thoroughly analyze the game of Pairs. We found a recursive equation in both the two-player (Theorem 3) and the m-player case (Theorem 10) for Pairs, as well as a good condition for folding in the two-player case (Corollary 8). With these recursive equations, we solved for the unique SPNE solution to Pairs.

In addition, we implemented Pairs in a Python program and tested our SPNE analysis in a player against other players. In both the two-player case and the m-player case, the SPNE player was able to win over the random player and the heuristic player, thus showing that our SPNE analysis did work well. Although the heuristic player did well against the SPNE player, our SPNE player did still win against the other players over 10,000 games.

7 Appendix

7.1 Pairs Code

File pairs.py

```
import random

class Pairs:
    def __init__(self, debug=True):
        self.debug = debug

    def dbg(self, msg):
        if self.debug:
            print msg

    def init_game(self, clses):
        # initialising the deck
        self.deck = []
        self.deckCount = []
        for i in xrange(1, 11):
            self.deck += [i]*i
            self.deckCount.append(i)
        random.shuffle(self.deck)
        self.discards = []
        # initialising players
        self.players = []
        for i in xrange(len(clses)):
```

```
        self.players.append(clses[i](i))
    return self.init_round()

def init_round(self):
    self.dbg("")
    # discarding all cards in play
    for i in xrange(len(self.players)):
        p = self.players[i]
        self.discards += p.hand
        self.players[i].hand = []
        self.dbg("Player " + str(i) + " score: " + str(p.score) + " and scored: " + str
    self.dbg("")
    # dealing initial hand
    lowest = []
    lowest_card = 11
    for i in xrange(len(self.players)):
        hand = self.pop_from_deck(1)
        self.players[i].insert_into_hand(hand)
        if hand[0] < lowest_card:
            lowest = [i]
            lowest_card = hand[0]
        elif hand[0] == lowest_card:
            lowest.append(i)
        self.dbg("Player " + str(i) + " is dealt " + str(hand[0]))
    # determining first player
    self.dbg("Lowest is players " + str(lowest) + " with card " + str(lowest_card))
    while len(lowest) > 1:
        lowest_again = []
        lowest_card_again = 11
        for i in lowest:
            another = self.pop_from_deck(1)
            self.dbg("Player " + str(i) + " is dealt " + str(another[0]))
            while another[0] in self.players[i].hand:
                self.dbg("This is a duplicate! Discarded.")
                self.discards.append(another[0]);
                another = self.pop_from_deck(1)
                self.dbg("Player " + str(i) + " is dealt " + str(another[0]))
            self.players[i].insert_into_hand(another)
            if another[0] < lowest_card_again:
                lowest_again = [i]
                lowest_card_again = another[0]
            elif another[0] == lowest_card_again:
                lowest_again.append(i)
        lowest = lowest_again
        self.dbg("Lowest is players " + str(lowest) + " with card " + str(lowest_card_a
    self.dbg("Player " + str(lowest[0]) + " goes first.")
```

```
        return lowest[0]

    def pop_from_deck(self, n):
        r = []
        for i in xrange(n):
            while len(self.deck) <= 5:
                if len(self.discards) == 0:
                    print "ERROR: discards is empty while reshuffling"
                self.dbg("Reshuffling!")
                self.deck += self.discards
                self.discards = []
                random.shuffle(self.deck)
                self.deckCount = [0]*10
                for c in self.deck:
                    self.deckCount[c-1] += 1
            r.append(self.deck.pop())
            self.deckCount[r[-1]-1] -= 1
        return r

    def play(self, clses, dbg=True):
        self.debug = dbg
        pid = self.init_game(clses)
        limit = (60 / len(clses)) + 1
        while True:
            p = self.players[pid]
            mv = p.decide(self.players, self.deckCount)
            self.dbg("")
            if mv:
                # hit
                deal = self.pop_from_deck(1)[0]
                if deal in p.hand:
                    # you got a pair
                    p.score += deal
                    p.scored.append(deal)
                    self.dbg("Player " + str(pid) + " has hit for " + str(deal) + " and got
                    if p.score >= limit:
                        break
                    pid = self.init_round()
                    continue
                else:
                    p.insert_into_hand([deal])
                    self.dbg("Player " + str(pid) + " has hit for " + str(deal) + " and now
            else:
                # fold
                all_cards = [y for x in self.players for y in x.hand]
                lowest = min(all_cards)
                steal_from = p.handle_fold(lowest, self.players)
                if lowest in self.players[steal_from].hand:
                    self.players[steal_from].hand.remove(lowest)
                    p.score += lowest
                    p.scored.append(lowest)
                    self.dbg("Player " + str(pid) + " has folded and taken a " + str(lowest)
                    if p.score >= limit:
                        break
                    pid = self.init_round()
                    continue
                else:
                    print "----------------------------------------------------------------
                    print "ERROR: Lowest " + str(lowest) + " not in player " + str(steal_fro
                    print "----------------------------------------------------------------

            pid = (pid + 1) % len(self.players)
        self.dbg("")
        self.dbg("Player " + str(pid) + " has lost!")
        self.dbg("")
        self.dbg("Final scores:")
        for i in xrange(len(self.players)):
            p = self.players[i]
            self.dbg("Player " + str(i) + " score: " + str(p.score) + " and scored: " + str(
        return pid
```

7.2 Base Player Code

File pairsPlayer.py

```
class PairsPlayer:
    def __init__(self, pid):
        self.type = "Default Player"
        self.pid = pid
        self.hand = []
        self.score = 0
        self.scored = []

    def insert_into_hand(self, cards):
        new_cards = [x for x in cards]
        new_hand = []
        while len(self.hand) > 0 or len(new_cards) > 0:
            if len(self.hand) == 0:
                new_hand += new_cards
                break
            if len(new_cards) == 0:
                new_hand += self.hand
                break
            if self.hand[0] <= new_cards[0]:
                new_hand.append(self.hand.pop(0))
            else:
                new_hand.append(new_cards.pop(0))
        self.hand = new_hand

    # Returns the pid of the player you want to steal from
    # The player must have lowest in their hand
    def handle_fold(self, lowest, players):
        # OVERRIDE THIS IF YOU DEFINE A NEW PLAYER
        print
        for pid in xrange(len(players)):
            print "Player " + str(pid) + " has hand " + str(players[pid].hand)
        print "You are player " + str(self.pid) + " taking a " + str(lowest)
        x = int(raw_input("Which player will you take from? "))
        while lowest not in players[x].hand:
            print "Player " + str(x) + " does not have " + str(lowest)
            x = int(raw_input("Which player will you take from? "))
        return x

    # Returns true if hit and false if fold
    def decide(self, players, deck):
        # OVERRIDE THIS IF YOU DEFINE A NEW PLAYER
        print
        print "Player " + str(self.pid) + "'s hand is " + str(self.hand)
        ans = raw_input("Will you hit or fold? (h/f) ")
        while len(ans) == 0:
            print "That's not an answer."
            ans = raw_input("Will you hit or fold? (h/f) ")
        return ans[0] == 'h'
```

7.3 Two-Player SPNE Player

File twoPlayerPlayer.py

```
from randomPlayer import RandomPlayer
from twoPlayerStrategy import twoAction

class TwoPlayerPlayer(RandomPlayer):
    def decide(self, players, deck):
        if len(players) != 2:
            print "NUMBER OF PLAYERS FOR TWO PLAYER PLAYER IS NOT 2"
            return
        action = twoAction(players[self.pid].hand, players[1 - self.pid].hand, deck)
        return action == "Hit"
```

7.4 m-Player SPNE Player

File nPlayerPlayer.py

```
from randomPlayer import RandomPlayer
from nPlayerStrategyHeuristic import Configuration
from nPlayerStrategyHeuristic import nAction

class NPlayerPlayer(RandomPlayer):
    def decide(self, players, deck):
        Configuration.m = len(players)
        action = nAction([p.hand for p in players], deck)
        return action == "Hit"
```

7.5 Random Player

File randomPlayer.py

```
import random
from pairsPlayer import PairsPlayer

class RandomPlayer(PairsPlayer):
    def decide(self, players, deck):
        return random.random() < 0.5

    def handle_fold(self, lowest, players):
        eligible = [i for i in xrange(len(players)) if lowest in players[i].hand]
        return random.choice(eligible)
```

7.6 Heuristic Player

File heuristicPlayer.py

```
from randomPlayer import RandomPlayer

class HeuristicPlayer(RandomPlayer):
    def decide(self, players, deck):
        limit = (60 / len(players)) + 1
        all_cards = [y for x in players for y in x.hand]
        lowest = min(all_cards)
        if self.score + lowest >= limit:
            return True
        a = sum(self.hand) / len(self.hand)
        if self.score + a >= limit:
            return False
        if self.score > 18 * lowest:
            return False
        return True
```

7.7 Two-Player SPNE Analysis Code

File twoPlayerStrategy.py

```
def twoAction(stack1, stack2, deck):
    player1 = 0
    player2 = 0
    low = 11
    deck = [k+1 for k in xrange(10)]
    for c in stack1:
        player1 += 2**(c-1)
        if c < low:
            low = c
        deck[c-1] -= 1
    for c in stack2:
        player2 += 2**(c-1)
        if c < low:
            low = c
        deck[c-1] -= 1
    return Configuration(deck, player1, player2, low).action()

class Configuration:
    udict = dict()
    c = 0
    #####!!!!!IMPORTANT TO CHANGE m and n!!!!!#####
    n = 10 #number of ranks in deck
    m = 2 #number of players, should always be 2
    num_stages = 5
    def __init__(self,deck,stack1,stack2,low,recursionDepth=0):
        self.deck = deck
        self.stack1 = stack1
        self.stack2 = stack2
        self.low = low
        self.rd = recursionDepth

    def __repr__(self):
        return '(' + str(self.deck) + ',' + bin(self.stack1)[2:].zfill(Configuration.n)[::-

    def hit(self,j):
        newDeck = self.deck[:]
        newDeck[j-1] -= 1
        return Configuration(newDeck, self.stack2, self.stack1 | 2**(j-1), min(self.low,j),

    def ufold(self):
        return - self.low

    def uhit(self):
        Configuration.c += 1
        if not Configuration.c % 10000: print Configuration.c, '2Processing:', self
        s = bin(self.stack1)[2:].zfill(Configuration.n)
        p = 0 #the payoff to return
        z = 0 #the limiting payoff
        h = [] #the values requiring recursion
        for j in xrange(1,Configuration.n+1):
            if s[-j] == '1':
                p += j*self.deck[j-1]
                z += 1*self.deck[j-1]
            else:
                if self.deck[j-1]: h.append(j)
                z += 2*self.deck[j-1]
        if -p <= -z*self.low: return -float("inf")
        for j in h:
            p += self.hit(j).umax()*self.deck[j-1]
        return - float(p) / sum(self.deck)

    def umax(self):
        s = str(self.deck) + str(self.stack1).zfill(Configuration.n/2) + str(self.stack2).zf
        if s in Configuration.udict: return Configuration.udict[s]
        elif self.rd >= Configuration.num_stages: return self.use_heuristic()
        elif sum(self.deck) <= 5: return self.use_heuristic()
        else:
            u = max(self.uhit(),self.ufold())
            if self.rd < Configuration.num_stages - 0:
                Configuration.udict[s] = u
            return u

    def use_heuristic(self):
        '''
        limit = 31
        hand = bin(self.stack1)[2:].zfill(Configuration.n)[::-1]
        wa = sum([(i+1)*self.deck[i]*int(hand[i]) for i in xrange(len(hand))]) / sum(self.de
        if self.score + self.low >= limit:
            return wa
        a = sum(self.hand) / len(self.hand)
        if self.score + a >= limit:
            return self.low
        if self.score > 18 * self.low:
            return self.low
        return wa
        '''
        hand = bin(self.stack1)[2:].zfill(Configuration.n)[::-1]
        ws = sum([(i+1)*self.deck[i]*int(hand[i]) for i in xrange(len(hand))])
        wa = ws / sum(self.deck)
        return max(wa, self.ufold())

    def action(self):
        if self.umax() == self.ufold(): return 'Fold'
        else: return 'Hit'

if __name__ == "__main__":
    for (i,j) in [(i,j) for i in xrange(Configuration.n) for j in xrange(Configuration.n) if
        a = [k+1 for k in xrange(Configuration.n)]
        a[i] -= 1
        a[j] -= 1
        #print (a,2**i,2**j,i+1)
        Configuration(a,2**i,2**j,i+1).umax()
    print 'Done'
```

7.8 m-Player SPNE Analysis Code

File nPlayerStrategy.py

```
def nAction(stacks):
    players = []
    low = 11
    deck = [k+1 for k in xrange(10)]
    for stack in stacks:
        players.append(0)
        for c in stack:
            players[-1] += 2**(c-1)
            deck[c-1] -= 1
            if c < low:
                low = c
    return Configuration(deck, players, low).action()

class Configuration:
    udict = dict()
    c = 0
    #####!!!!!IMPORTANT TO CHANGE m and n!!!!!#####
    n = 10 #number of ranks in deck
    m = 3 #number of players

    def __init__(self,deck,stacks,low):
        self.deck = deck
        self.stacks = stacks
        self.low = low

    def __repr__(self):
        return '(' + str(self.deck) + ',' + ','.join(map(lambda x: bin(x)[2:].zfill(Configur
    def hit(self,j):
        newDeck = self.deck[:]
        newDeck[j-1] -= 1
        newStacks = self.stacks[1:] + [self.stacks[0] | 2**(j-1)]
        return Configuration(newDeck, newStacks, min(self.low,j))

    def ufold(self):
        p = [self.low] * Configuration.m
        p[0] += -Configuration.m*self.low
        return p

    def uhit(self):
        Configuration.c += 1
        if not Configuration.c % 10000: print Configuration.c, 'nProcessing:', self
        s = bin(self.stacks[0])[2:].zfill(Configuration.n)
        p = [0]*Configuration.m #the payoff to return
        z = float(sum(self.deck)) #normalization
        for j in xrange(1,Configuration.n+1):
            if self.deck[j-1] == 0: continue
            if s[-j] == '1':
                p[0] += -Configuration.m*j*self.deck[j-1]/z
                for i in xrange(Configuration.m): p[i] += j*self.deck[j-1]/z
            else:
                q = self.hit(j).umax()
                for i in xrange(Configuration.m): p[i] += q[i-1]*self.deck[j-1]/z
        return p

    def umax(self):
        s = str(self.deck) + ''.join(map(lambda x: str(x).zfill(Configuration.n/2),self.stac
        if s in Configuration.udict: return Configuration.udict[s]
        else:
            u = max(self.uhit(),self.ufold())
            Configuration.udict[s] = u
            return u

    def action(self):
        if self.umax() == self.ufold(): return 'Fold'
        else: return 'Hit'

if __name__ == "__main__":
    for (i,j,k) in [(i,j,k) for i in xrange(Configuration.n) for j in xrange(Configuration.n
        a = [z+1 for z in xrange(Configuration.n)]
        a[i] -= 1
        a[j] -= 1
        a[k] -= 1
        #print (a,[2**i,2**j,2**k],i+1)
        Configuration(a,[2**i,2**j,2**k],i+1).umax()
    print 'Done'
```