Optimally Stacking the Deck, and a Generalized Steepest Ascent Algorithm

Jeremy Kun

May 2, 2012

1 Overview

In this paper we investigate a specific combinatorial property of card games, namely, whether it's possible to stack a deck of cards so mischeviously so that no matter where the deck is cut, a specific player can always force a win. The main results are focused on whether these so-called "optimal stackings" exist for Texas Hold 'em, a popular poker variant. The main theorem is that optimal stackings exist and are relatively plentiful for two-player Texas Hold 'Em. We wrote a parallelized program to search for optimal stackings, and the program found them at a rate of roughly a hundred per minute. We also provide examples of card games for which no optimal stacking exists. We note that there may be a relationship between the complexity of a game and the existence of an optimal stacking.

To the best of our knowledge, the results presented in this paper are the first of their kind. While we doubt that such results are applicable in any discipline, they are curious and interesting to think about.

The paper is organized as follows. We first lay out a mathematical description of an optimal stacking, and provide examples of card games which have none. Then we proceed to a description of the generalized steepest ascent algorithm we designed in finding optimal stackings for Texas Hold 'Em. Next, we detail the rules of Texas Hold 'Em and how this fits into the problem of steepest ascent, and we describe our poker-hand evaluation engine. Finally, we describe two different methods of parallelizing the code for use on a shared-memory machine, and time it for various numbers of threads between 1 and 12.

2 Preliminary Definitions

In card games, especially gambling games, trust is rarely given freely. Instead, we incorporate rituals into standard play that assure fairness. One of the most common such rituals is cutting the deck. The player who shuffles the deck passes the deck to a second player, who cuts the deck in half, placing the bottom half above the top half. Assuming the two players are not colluding and there is no sleight of hand, this ritual absolves the dealer of any guilt. Even if the dealer had stacked the deck to deal himself favorable cards, the randomness of the cut will necessarily ruin his efforts. Moreover, the cutter is often chosen to be the player to the dealers right (the last one dealt, in most games) to make it all the more difficult for two players to collude.

An optimal stacking circumvents the tacit fairness assumed after a deck is cut. In particular, we search for an ordering of the cards in the deck which is so mischievous that no matter where it is cut, a specific player always wins.

This author doesnt condone cheating in card games, nor does he plan to use the discoveries in this article for anything more than a parlor trick. Instead, the existence or nonexistence of an optimal stacking is an interesting combinatorial property of card games, and it appears to be a measure of complexity.

As an example of this, simple games like Blind Mans Bluff (sometimes called Indian Poker) and Kuhn Poker do not have optimal stackings. For the latter, one can trivially enumerate all possible situations and check for a counterexample cut.

On the other hand, we constructed slightly more complicated games similar to Texas Hold 'Em (i.e., played with 9 cards and having only two different classes of hands) for which there are optimal stacking. And so we have the following conjecture, which is now a theorem:

Theorem 2.1. *There exist optimal stackings for two-player Texas Hold 'Em.*

In fact, we find much more than this, but we should formalize the notion a bit further and describe our method before constructing such a stacking.

Definition 2.1. *A cutting permutation σ of a list $(1, 2, \dots, n)$ is a permutation in S_n with the cycle form:*

$$(1\ 2\ \dots\ n)^k$$

and for a fixed $0 \leq k < n$, we call σ the cut at the k-th position.

Since any list imposes an ordering on its contents, we can apply cutting permutations to any list.

Definition 2.2. *A cut of a list $(c_1, \dots, c_n)$ is the list $(c_{\sigma(1)}, \dots, c_{\sigma(n)})$ for some cutting permutation σ. If the list is denoted l, we denote the cut $\sigma(l)$.*

Specifically to card games, we speak of cutting a deck as the process of replacing a list of cards with its cut. We can now define an optimal stacking.

Definition 2.3. *Definition: Fix a game G played with a set of k cards and n players. We say that G has an optimal stacking for player i if there exists an ordering of the cards $D = (c_1, \dots, c_k)$ such that for every cutting permutation σ, player i can force a win when $\sigma(D)$ is dealt.*

Moreover, a game is said to simply have an optimal stacking if it has one for any player. We do not yet know of games which have optimal stackings for some players but not others, and we conjecture that no natural games do (that would make the game unfair for some players, although this author has always found asymmetric games interesting). It may also benefit one to assume in general that all players are infinitely rational, a standard assumption in Game Theory.

Note that we do not want to assume the cheating player knows the order of the deck ahead of time. Instead, an optimal stacking should allow the player to force a win simply by logical play and the knowledge that the stacking is optimal. Specifically, for a game where players have individual hands, the cheating player should not need to know the contents of the other players' hands in order to win. In Texas Hold 'Em this becomes a null issue, since if a player knows he is being dealt an optimal stacking, his hand will always end up winning; there is no reason for him to fold, even if his chances seem slim to none by usual probabilistic analysis.

Before we get back to the main content of this paper, we take a moment to prove a nice fact.

Proposition 2.1. *The (slightly simplified) game of Hearts does not have an optimal stacking.*

Proof. First, we must clarify that a "win" in Hearts is to either collect 0 points in a round or shoot the moon, and such a score must be forced in order to have an optimal stacking. We allow no passing of cards. Moreover, we restrict our game of Hearts to a single round, so that the winner of the game is the one who takes the smallest number of points. Hence, if one cannot shoot the moon, one must try to minimize the number of points taken. In particular, there is no reason for one player to "take one for the team" by trying to stop another player from shooting the moon unless the first player can guarantee having the smallest number of points at the end.

The crux of the proof is that all cards in the deck are dealt before play begins. Suppose first that the goal is simply to collect 0 points. Indeed, suppose that some stacking D wins for player 1. Then a cut at position 1 shifts all of the hands right by one player, so then player 4 can guarantee collecting 0 points. If still player 1 can guarantee collecting zero points, then instead a cut at position 2 gives such guarantees to players 3 and 4. At this point, player 1 can still guarantee taking 0 points, then player 2 can shoot the moon, hence giving all other players 26 points.

If, on the other hand, player 1 can guarantee shooting the moon, the same argument shows that a cut at position 1 gives that guarantee to player 4. This is a contradiction since two players cannot shoot the moon simultaneously, nor can one shoot the moon while the other takes zero points. □

Even extending the definition of a win to "collecting the minimum number of points" does not change the existence of an optimal stacking. The same proof applies, along with the knowledge that 26 (the total number of points) is not divisible by 4 (the number of players), or more coarsely that the Queen of Spades already represents 13 points, and can only be taken by one player.

One aspect of this proof contradicts the claim that the existence of an optimal stacking is a measure of complexity. In particular, Hearts is considered a rather complex game (certainly more complex than Kicsi Poker), and yet it does not have an optimal stacking. Moreover, with a suitable definition of a "win," we can likely extend this proof to any game which deals out the entire deck to all players before starting (and where all players can see all their cards at all times).

While we do not have a formal rebuttal, it may instead be the case that the existence of an optimal stacking measures the ability for a game to diverge. In other words, in a game like Texas Hold 'Em, one can start with a poor hand and later improve it. On the other hand, it could simply be that Texas Hold 'Em has a complex deal: the five community cards, along with the dead cards burned in between each round, give the game enough "space" to allow for an optimal stacking.

3 The Steepest Ascent Algorithm

In this section we detail the steepest ascent algorithm used in our search for optimal stackings. The algorithm was implemented in C++, and parallelized for use on a desktop supercomputer.

3.1 Pseudocode

In general, given a discrete space of points X and a function $f : X \to \mathbb{R}$, one can not easily compute the gradient for the usual method of function maximization. Instead, we compute a discretized version of the gradient by comptuing a (heuristicaly chosen) candidate set of neighbors for a given point $x \in X$, and we say the gradient "points" in the direction of the neighbor with the largest image under f.

In pseudocode the algorithm is:

```
1. Pick a random starting value $x$.
2. Until no neighbors $x_i$ of $x$ have $f(x_i) > f(x)$, do:
    (a) Replace $x$ with the $\text{argmax}_i(f(x_i))$.
```

We then repeat this process as many times as needed until we find a global maximum. Note that for the problem of finding optimal stackings, we will know ahead of time what the global maximum is. For other problems, one must simply approximate the global maximum if it is not known.

3.2 C++ Implementation, and Virtual Classes

To implement this algorithm we designed a pure virtual C++ class to represent a Position in the discrete space X, and wrote a function to operate on such Positions. Then a programmer can derive the Position class and implement the pure functions, and the same piece of code can be used. Specifically, the class has the following definition, taken from the file hillclimb.h:

```cpp
#include <vector>
#include <string>

class Position {
public:
    Position();
    virtual ~Position();
    virtual double value() = 0;
    virtual std::vector<Position *> *neighbors() = 0;
    virtual std::string show() = 0;
};
```

In particular, we require the neighbors function to allocate a new list of neighboring Positions, and then relinquish ownership of the pointers to those positions.

Then the actual function for hill climbing is as follows:

```
Position* hillclimb(Position* posn, const int numSteps) {
    double value = posn->value(), nextValue;
    int i;
    bool foundBigger;

    vector<Position *> *nbrs;
    vector<Position *>::iterator itr;

    for (i = 0; i < numSteps; i++) {
        nbrs = posn->neighbors(); // allocates neighboring Positions with new

        foundBigger = false;
        for (itr = nbrs->begin(); itr != nbrs->end(); itr++) {
            nextValue = (*itr)->value();

            if (nextValue > value) {
                foundBigger = true;
                value = nextValue;
                posn = *itr;
            }
        }

        // free the computed neighbors
        for (itr = nbrs->begin(); itr != nbrs->end(); itr++)
            if (*itr != posn)
                delete *itr;

        delete nbrs;
        if (!foundBigger)
            break;
    }

    return posn;
}
```

The code is a direct translation of the pseudocode provided above, with additonal lines added for memory management (i.e., the nonused Positions in the returned list of neighbors must be freed before the loop continues).

In the source code, we provide a pedantic example derivation of the Position class to optimize quartic functions of one variable, and show how to use the hillclimb function.

4 Texas Hold ’Em

In this section we briefly describe the rules of Texas Hold ’Em, and then describe how the problem of finding optimal stackings fits into the steepest ascent framwork, and then we describe the method we used for evaluating poker hands.

4.1 Rules

In Texas Hold ’Em each player is dealt two “pocket” cards face down, and then five “community” cards are placed face up in the center of the table. The winner of the game is the player with the best five card poker hand, combined in any way from his pocket and the community. There are betting rules as well, but for the sake of finding optimal stackings this is irrelevant. We only care about the order in which the players are dealt, and how one wins.

4.2 Steepest Ascent in Hold ’Em

For the sake of this paper we restrict the game to two players, and we search for optimal stackings for which the first player dealt wins.

In this case the discrete space X is the set of orderings of the 52 cards in a standard deck. The function f evaluating the fitness of a deck is defined as follows: if $D \in X$, then $f(D)$ is the number of cuts of the deck for which player 1 wins when dealt with that cut. Note that this will often be less than 52, and if $f(D)$ is ever 52, we can quit and output an optimal stacking.

Finally, the neighbors of an ordering D are computed by preforming all possible pairwise swaps from D to obtain new orderings.

The subclass of Position is defined and implemented in the files poker.h and poker.cpp, provided in the source code.

4.3 Evaluating Poker Hands

Traditional methods for evaluating poker hands are quite slow in practice. On the other hand, methods which enumerate all possible poker hands for comparison use up a prohibitively large amount of memory. Our method to evaluate poker hands borrows the techniques from two poker enthusiasts Kevin Suffecool and Paul Senzee. [1] It is quick, in that it can evaluate on the order of fifteen million 5-card hands per second on a 1GHz processor. And it uses a small amount of memory, less than 1MB.

Their method relies on the observation that there are only 7,462 distinct 5-card poker hands up to which hand wins. For example, there are 1,020 different hands of the form AKQJ9 which are not a flush, and hence are all equivalent in value. This method utilizes perfect hashing to transform a set of five cards to the correct number between 1 and 7,462 representing its value among all possible hands. This is quite an improvement from the original 2.5 million distinct poker hands, and we also note that (being a hashing function) the calculaution is performed in constant time.

The files implementing Suffecool’s method are included in the source, included in poker.h, poker.cpp, and arrays.h (for the hashing lookup tables).

5 Parallelization

We parallelized the steepest ascent algorithm in two ways. First, at the coarsest granularity, every process works on a different hill-climbing instance and they combine their results at the end. Second, at the finest granularity, all processes cooperate on evaluating the fitness of a single ordering of the deck.

[1] see www.suffecool.net and www.paulsenzee.com, respectively.

We implemented the parallelization in both cases using OpenMP for use on a shared memory machine, but there is no reason one cannot rewrite the code for use on distributed memory machines using a library such as MPI.

5.1 Coarsest Granularity

The problem of executing steepest ascent is pleasingly parallel for the same reason that Monte Carlo methods are: each starting location is independent of the others. Indeed, in addition to being the fastest of our experiments, it was the easiest to code. The reader can see the source listing in the parallel/ directory of the included source files.

Here we include a table of running times for the pleasing parallelization:

Number of Trials	Trial Length	Threads	Runtime	Speedup	Efficiency
100	20	1	85.26	-	-
100	20	2	44.64	1.909	0.95
100	20	4	22.49	3.790	0.94
100	20	8	12.59	6.772	0.85
100	20	12	8.99	9.484	0.79

5.2 Finest Granularity

For the finest granularity we parallelized within the inner-most loop of the algorithm, so that every call to the neighbor-generating function and every call to the deck evluation function was parallelized. This method does not scale as well as the pleasing parallelization above, in part because it requires significantly more system time in its thread scheduling. For instance, in a run of 1000 trials of length bounded by 20, and 12 threads, we saw that the coarse parallelization exhibited a speedup of 1.17, and the fine parallelization required six seconds of system time, which accounted for about 93 percent of the discrepancy.

Nevertheless, we noted that the efficiency of the fine parallelization was decidedly better. We imagine that many of the trials end quickly for the coarse granularity, causing some threads to finish their work before others. This is because we implemented the parallelization with a static job distribution, but we are confident from our previous experiences with dynamic job queues that dynamic job allocation would be much slower for this problem, in part because the lenths of the jobs are not variable enough to overcome the additional overhead.

Here is a table of running times and speedups for the finest granularity parallelization:

Number of Trials	Trial Length	Threads	Runtime	Speedup	Efficiency
100	20	1	85.26	-	-
100	20	2	46.13	1.848	0.92
100	20	4	23.84	3.576	0.89
100	20	8	12.49	6.876	0.86
100	20	12	8.66	9.84	0.82

6 Results

The result of our computer search was affirmative: there exist optimal stackings for Texas Hold 'Em. Here is one such, including the table enumerating the hangs resulting from each of the 52 cuts of the deck:

```
Th5d7cAd3cQsJc4h6c8c9dTc6hQc8d9sJh5c7dAhAc9h2cKh5sJs
8s7h2d7s9c4dKd8hQd6d3sKs5h2hAs4s2sTs6sQhJd3h4cTd3dKc
```

```
 p1     p2      community       p1 hand     p2 hand
```

Th7c	5dAd	Qs Jc 4h 8c Tc	One Pair	High Card
5dAd	7c3c	Jc 4h 6c 9d 6h	One Pair	One Pair
7c3c	AdQs	4h 6c 8c Tc Qc	Flush	One Pair
AdQs	3cJc	6c 8c 9d 6h 8d	Two Pair	Two Pair
3cJc	Qs4h	8c 9d Tc Qc 9s	Flush	Two Pair
Qs4h	Jc6c	9d Tc 6h 8d Jh	Straight	Two Pair
Jc6c	4h8c	Tc 6h Qc 9s 5c	Flush	High Card
4h8c	6c9d	6h Qc 8d Jh 7d	One Pair	One Pair
6c9d	8cTc	Qc 8d 9s 5c Ah	One Pair	One Pair
8cTc	9d6h	8d 9s Jh 7d Ac	Straight	One Pair
9d6h	TcQc	9s Jh 5c Ah 9h	Trips	One Pair
TcQc	6h8d	Jh 5c 7d Ac 2c	Flush	High Card
6h8d	Qc9s	5c 7d Ah 9h Kh	Straight	One Pair
Qc9s	8dJh	7d Ah Ac 2c 5s	One Pair	One Pair
8dJh	9s5c	Ah Ac 9h Kh Js	Two Pair	Two Pair
9s5c	Jh7d	Ac 9h 2c 5s 8s	Two Pair	High Card
Jh7d	5cAh	9h 2c Kh Js 7h	Two Pair	High Card
5cAh	7dAc	2c Kh 5s 8s 2d	Two Pair	One Pair
7dAc	Ah9h	Kh 5s Js 7h 7s	Trips	One Pair
Ah9h	Ac2c	5s Js 8s 2d 9c	One Pair	One Pair
Ac2c	9hKh	Js 8s 7h 7s 4d	One Pair	One Pair
9hKh	2c5s	8s 7h 2d 9c Kd	Two Pair	One Pair
2c5s	KhJs	7h 2d 7s 4d 8h	Two Pair	One Pair
KhJs	5s8s	2d 7s 9c Kd Qd	One Pair	High Card
5s8s	Js7h	7s 9c 4d 8h 6d	Straight	One Pair
Js7h	8s2d	9c 4d Kd Qd 3s	High Card	High Card
8s2d	7h7s	4d Kd 8h 6d Ks	Two Pair	Two Pair
7h7s	2d9c	Kd 8h Qd 3s 5h	One Pair	High Card
2d9c	7s4d	8h Qd 6d Ks 2h	One Pair	High Card
7s4d	9cKd	Qd 6d 3s 5h As	Straight	High Card
9cKd	4d8h	6d 3s Ks 2h 4s	One Pair	One Pair
4d8h	KdQd	3s Ks 5h As 2s	Straight	One Pair
KdQd	8h6d	Ks 5h 2h 4s Ts	One Pair	High Card
8h6d	Qd3s	5h 2h As 2s 6s	Two Pair	One Pair
Qd3s	6dKs	2h As 4s Ts Qh	One Pair	High Card
6dKs	3s5h	As 4s 2s 6s Jd	Flush	Flush
3s5h	Ks2h	4s 2s Ts Qh 3h	One Pair	One Pair
Ks2h	5hAs	2s Ts 6s Jd 4c	One Pair	High Card
5hAs	2h4s	Ts 6s Qh 3h Td	One Pair	One Pair
2h4s	As2s	6s Qh Jd 4c 3d	One Pair	High Card
As2s	4sTs	Qh Jd 3h Td Kc	Straight	One Pair
4sTs	2s6s	Jd 3h 4c 3d Th	Two Pair	One Pair
2s6s	TsQh	3h 4c Td Kc 5d	Straight	One Pair
TsQh	6sJd	4c Td 3d Th 7c	Trips	One Pair
6sJd	Qh3h	Td 3d Kc 5d Ad	Flush	One Pair
Qh3h	Jd4c	3d Kc Th 7c 3c	Trips	One Pair
Jd4c	3hTd	Kc Th 5d Ad Qs	Straight	One Pair
3hTd	4c3d	Th 5d 7c 3c Jc	Two Pair	One Pair
4c3d	TdKc	5d 7c Ad Qs 4h	One Pair	High Card
TdKc	3dTh	7c Ad 3c Jc 6c	Flush	One Pair
3dTh	Kc5d	Ad 3c Qs 4h 8c	One Pair	High Card

```
Kc5d    Th7c    3c Qs Jc 6c 9d    High Card    High Card
```

We also note a few properties we discovered about optimal stackings:

- Optimal stackings are plentiful. About one in six randomly chosen decks can be improved to optimal by pairwise card swaps.
- The number of steps required in the hill climbing process is small. In all of our experiments, we never witnessed a deck requiring more than 15 steps to reach a local maximum. We do not have a proof that this upper bound is sharp. Moreover, some decks were optimized to win for all 52 cuts in as few as 5 steps.
- There are optimal stackings for both players, which we found by a slight manipulation of the program to find optimally *losing* decks for player 1.

We have posted online [2] a file containing the enumeration tables for a list of approximately 100,000 such optimal stackings for two-player Hold 'Em. We did not check explicitly that none are duplicated up to cuts, but the probability is infinitesimal that any are. Here is a small list of such stackings, so that the casual reader need not download the entire file:

```
AcQhJd8h9c9h6d3cTc3dQdKhJs6h3hThQc7d3sJc4h2h6c8d7c5c
7s8cAd4dTd9sKs5h8s2c4c2d2s5sAhKd9dKcQs4s5dAs7hJhTs6s

8h4h3cAsQd7dTh3s5h9dAhKh9sTs6s4d7s2hKdJc9h6h3h4c4sQc
JsTd5d2c9c3dQhJd8c5c2dKs8s6cJh7hAcAdKc8d2sTc6dQs7c5s

6d2cAc7sQc9sThJs2dQd7h6c9dAsKs2sKc3d4c9hJc2hAdKh3cQs
4h5c3hTdAh4s8c8hQh7dTc8s8d5s6hJd4d3s5h6sTsKd9c7cJh5d

Th2s9s6hJh3s8c5h5cAh6s3cQhJcKc4c9dAcTcJd3hQs7h8sAsTs
2d8d4s3d5s7sKsAd9hKd5d2hTd4h7d7cJs9cQdQc2c8h6cKh4d6d

Js7hQhTh4h8s9hAh7sQc6d3sTc9d3d5hTs8cJhAcQd5s5cTd4dKs
KcJd8h2hKh4c6s7d2sQs7c6h3cKdAs2d9cJc6cAd8d3h9s4s5d2c

Jd2cAc9s5c8h9cTh4c7d2dTcAdKcKsQc8c4h7s4s9d5dJs5h7hKd
Qh7c4d9h8d2h6dQdJc3cAs3sKh8sTd6s3hAh5sJhQsTs6h6c2s3d

6dJs5c4s8d8h9s6s4dTh8s2c6h3hQd9d4hAc2d8cKcJh3d5sJdQh
7dAd3cTsAs7c5h7hQs2sTcKs4cAhKdTd7s6cKhQcJc9c3s9h2h5d

9dTdAd7hKd9h3cJcTsKh3h9c8s3d4hKsJs3s5c8h6c6s5d4dKcAs
7dQc6d5h2c4s4c7c5s2dQh9s7s8cAc2sJhQsThQd6hJd8dAhTc2h

3cJd4s2s9hAd6sKcTd3h5dTs3s9d8d4cAcQsKsKdJc3d2h5h4dJs
8cAh9c7sKh6d5s4hTc7cQhQc8hJh6c9s5c2d7dQd2c8sTh6hAs7h

Ac7cTd6d3c4c9hQc7h9d8hJc8c5sTc6hAdQs9s7s2cKc2s2d4s8d
6sAh5d8sJs3sQhKd3hQdJh5cKh4h3d5hTsJd7dKs9c2h4dTh6cAs
```

[2] http://code.google.com/p/math-intersect-programming/downloads/list

7 Concluding Remarks and Future Work

We are very interested in the existence or non-existence of optimal stackings for other games. We are also intersted in the properties of optimal stackings for games like Hold 'Em, where they seem to be so plentiful. Also for Hold 'Em, we have yet to compute whether there are optimal stackings for n players, but given the abundance of two-player stackings, we conjecture that they do exist. An interesting question might be: how fast does the complexity of finding optimal stackings grow as n grows?

With minor modifications to our algorithm, we could investiage other poker variants, such as Omaha or Lowball. Additionally, we could relax the notion of strict optimality to find decks which are α-optimal for a specific player, where $0 < \alpha < 1$ is the proportion of cuts which win for the desired player.

We leave all opportunities for future work (or recreation).