2003

A risk-averse strategy for blackjack using fractional dynamic programming

Ryan A. Dutsch
Louisiana State University and Agricultural and Mechanical College

A RISK-AVERSE STRATEGY FOR BLACKJACK USING FRACTIONAL DYNAMIC PROGRAMMING

A Thesis

Submitted to the Graduate Faculty of the
Louisiana State University and
Agricultural and Mechanical College
in partial fulfillment of the
requirements for the degree of
Master of Science

in

The Department of Mathematics

by
Ryan A. Dutsch
B.S., Southeastern Louisiana University, 2000
August 2003

Acknowledgments

I would like to express an extreme amount of gratitude to my advisor, Dr. George Cochran. His patience and support throughout the whole process of the project was greatly appreciated. I would also like to express thanks to my committee members Dr. William Adkins and Dr. James Geaghan for taking time out of their busy schedules. Also, I would like to give a special thanks to two good friends, Yoo-hwan Hwang and Dr. Dennis Merino. Without their help and guidance this project would not have been possible.

Table of Contents

Acknowledgments ..... **ii**

Abstract ..... **iv**

1. Introduction **1**

2. Criteria for Risk-Averse Strategies **3**
2.1 Strategy(a) 5
2.2 Strategy(b) 8

3. Dynamic Programming/Fractional Dynamic Programming **9**

4. Structure of Program **13**

5. Results **15**

Appendix: Source Code **17**

References **25**

Vita **26**

Abstract

We present how blackjack is related to a discrete-time control problem, rather than a zero-sum game. Using the compiler Visual C++, we write a program for a strategy for blackjack, but instead of maximizing the expected value, we use a risk-averse approach. We briefly describe how this risk-averse strategy is solved by using a special type of dynamic programming called fractional dynamic programming.

1. Introduction

The game of blackjack can be modelled as a discrete-time stochastic control problem, since the game can be broken down into a discrete number of time-steps, and at each state of the game, the player selects one of a number of controls to move to a state at the next time-step. In blackjack, these time-steps could be represented as the number of cards in a player's hand. For blackjack, these time-steps run from when the player starts at time 0 to time 9. Time 0 will represent when the player has two cards in his hand, and time 9 will represent when the player has 11 cards in his hand. Each hand of three or more cards has two controls attached to it, representing the two options of either drawing an additional card or standing on the hand. Similarly, a hand at time 0 has three or four controls attached to it, representing the playing options of hitting, standing, doubling down, or splitting pairs. Since the dealer must hit until the value of his cards is seventeen or greater, he can not choose to play freely. This means blackjack can not be represented as a zero-sum game, since the strategy of the player is against the fixed play of the dealer.

The term *risk-averse* could have a few different meanings. We will solve two possible *risk-averse* strategies. The strategies are: (a) to maximize the probability of never going broke starting with a fixed finite bankroll, and (b) to maximize the probability of not losing money after a fixed (large) number of games, with an infinite bankroll. It turns out that Strategy (a) maximizes $\frac{m_1}{m_2}$, where m_1 is the expected profit to the player and m_2 represents second moment of the player's profit. And for Strategy (b), we must maximizes $\frac{m_1}{\sigma}$, where m_1 is the expected profit and σ represents the standard deviation in the profit. Depending on the

given situation, both these strategies can be considered risk averse. We will find both strategies using dynamic programming. Dynamic programming is an iterative method that starts at the last possible time and works back to the first possible time. As we said previously, we will assume that these time steps are the number of cards in the player's hand. For each time-step there is a certain number of options, or controls, and attached to each control is a probability distribtion on the states available at the next time-step. So using dynamic programming you find the probability measure for each option at every time, then choose the best option given the featured strategy. The two strategies we find will be to maximize a fraction. In this case, fractional dynamic programming will be used. Fractional dynamic programming is a branch of dynamic programming, which is defined in the same reasoning, which maximizes a fraction by solving an iterative sequence of classical dynamic programming problems.

We will describe the rules of our single-deck blackjack game. Our game is similar to the standard blackjack game, except for a few minor deviations that we need to make known. The play of the dealer will be to hit his hand until reaching a point value of at least seventeen, soft or hard, or to bust, which means a point value above twenty-one. There will be a maximum of four resplits allowed, and the player can double-down after splitting pairs. Given these deviations it is easy to show that the greatest possible profit of a single game will be 8, since hypothetically we could double-down on four hands in a single game.

2. Criteria for Risk-Averse Strategies

Martingale theory will be needed to show that Strategy(a) is equivalent to maximizing $\frac{m_1}{m_2}$, where m_1 is the expected value and m_2 represents the second moment. The following definition is from [1].

Definition 2.1. *Martingale. The sequence of random variables and* $\sigma-fields$ $\{X_n, \mathcal{F}_n\}$ *is called a martingale if and only if we have for each* n*:*

(a) $\mathcal{F}_n \subset \mathcal{F}_{n+1}$ *and* X_n *is* $\mathcal{F}_n-$*measurable*

(b) $E(\,|X_n|) < \infty$

(c) $X_n = E(X_{n+1}|\mathcal{F}_n)$, *a.e.*

Using the definition of a martingale, we will show two examples of martingales, the first being elementary, and the latter being relevant to my project. The second example will be shown in Strategy(a). Suppose $X = \begin{cases} +1 \text{ with probability } p \\ -1 \text{ with probability } q \end{cases}; p > q$ and $q = 1 - p$. (Think of X as the profit from a single bet.) Let X_i be the profit from the i^{th} bet, so $(X_1, X_2, X_3, \ldots)$ are identically independently distributed random variables. Then the total profit for the first n bets is $S_n = \sum_{i=1}^{n} X_i$.

The following theory was shown to be by Dr. Cochran. Let $Y_n = \left(\frac{q}{p}\right)^{S_n}$. I will show that Y_n is a martingale.

Lemma 2.2. $E\left[\left(\frac{q}{p}\right)^{X}\right] = 1$

Proof. We compute the expected value of $\left(\frac{q}{p}\right)^{X}$ in a straightforward manner:

$$\begin{aligned} E\left[\left(\tfrac{q}{p}\right)^X\right] &= \left(\tfrac{q}{p}\right)^1 P(X=1) + \left(\tfrac{q}{p}\right)^{-1} P(X=-1) \\ &= \left(\tfrac{q}{p}\right) p + \left(\tfrac{p}{q}\right) q \\ &= q + p = 1 \end{aligned}$$

□

Now to show the proof that example 1 is a martingale, we will use the definition of a martingale and Lemma 2.2.

Claim 2.3. *(Y_n) is a martingale with respect to $\mathcal{F}_n = \sigma(X_1, X_2, \ldots, X_n)$.*

Proof. Recall from Lemma 2.2 that $E\left[\left(\tfrac{q}{p}\right)^X\right] = 1$. We can easily see that the expected value of S_n is bounded, since the profit for a single game is $|S_n| \leq n$. This means $E\left(\left(\tfrac{q}{p}\right)^{S_n}\right) < +\infty$. To show part (c) of the definition of a martingale:

$$\begin{aligned} E(Y_{n+1}|\mathcal{F}_n) &= E\left(\left(\tfrac{q}{p}\right)^{S_{n+1}}|\mathcal{F}_n\right) \\ &= E\left(\left(\tfrac{q}{p}\right)^{S_n+X_{n+1}}|\mathcal{F}_n\right) \\ &= E\left(\left(\tfrac{q}{p}\right)^{S_n}\left(\tfrac{q}{p}\right)^{X_{n+1}}|\mathcal{F}_n\right) \\ &= \left(\tfrac{q}{p}\right)^{S_n} E\left(\left(\tfrac{q}{p}\right)^{X_{n+1}}|\mathcal{F}_n\right) \\ &= \left(\tfrac{q}{p}\right)^{S_n} E\left(\left(\tfrac{q}{p}\right)^{X_{n+1}}\right) = Y_n. \end{aligned}$$

The fourth equality makes use of the fact that if X is $\mathcal{F}$-measurable then $E(XY|\mathcal{F}) = XE(Y|\mathcal{F})$, while the fifth equality makes use of the fact that if X is independent of $\mathcal{F}$, then $E(X|\mathcal{F}) = E(X)$. □

2.1 Strategy(a)

In this section, I'll show that minimizing the probability of eventually going broke is essentially equivalent to maximizing $\frac{E(X)}{E(X^2)}$, where X is the player's profit from a single game.

If X_i denotes the player's profit for the i^{th} game, then the total player's profit after n games is $S_n = \sum_{i=1}^{n} X_i$. Let B_0 be the initial bankroll or the amount of money at the starting point, and let M be an amount of money at which the player is willing to stop. (Note that given strategies may make the player's profit exceed $\$M$.)

Definition 2.4. *A map $T : \Omega \to \{0, 1, 2, \ldots; \infty\}$ is called a stopping time if,*

$$\{T \leq n\} = \{\omega : T(\omega) \leq n\} \in \mathcal{F}_n, \forall n \leq \infty.$$

Define the stopping time to be $T_M = min\{n \in N : S_n \leq -B_0 \text{ or } M \leq S_n \leq M + 8\}$, and the total player's profit at the stopping time, $S_{(T_M)}$, will either be $-B_0$ or $M \leq S_{(T_M)} \leq M + 8$.

Assume the $(X_i)'s$ are independently identically distributed random variables. The moment generating function of X is denoted $\hat{f}_X(t)$ which is equal to $E(e^{tX})$. If you look at t near 0, then the standard series expansion of $\hat{f}_X(t)$ is $\hat{f}_X(0) + \hat{f}_X'(0)t + \hat{f}_X''(0)\frac{t^2}{2} + \hat{f}_X'''(0)\frac{t^3}{3!} + \cdots$. And we see that $\hat{f}_X(0) = 1$, $\hat{f}_X'(0) = E(X) = m_1$, $\hat{f}_X''(0) = E(X^2) = m_2, \ldots$. Since the game has a positive expected value to the player, we know $\hat{f}_X'(0) > 0$, and $\hat{f}_X''(t) = E(X^2 e^{tX}) > 0$ for all t. This tells us the graph of $\hat{f}_X$ has a positive slope at $t = 0$ and is always concave up. Since the player could also lose money, $P(X < 0) > 0$, which means $\lim_{t \to -\infty} \hat{f}_X = +\infty$. Thus there exists a unique $c < 0$ such that $\hat{f}_X(c) = 1$ hence $E(e^{cX}) = 1$ for some $c < 0$. We can approximate this c by replacing $\hat{f}_X$ with a second degree Taylor

polynomial, $p_X = 1 + m_1 t + m_2 \frac{t^2}{2}$. If we set p_X equal to 1, and solve for t, we get $t\left(m_1 + \frac{m_2 t}{2}\right) = 0$, which leads to the solutions $t = 0$ or $t = -2\left(\frac{m_1}{m_2}\right)$. Given $c < 0$, c is approximately $-2\left(\frac{m_1}{m_2}\right)$.

In order to use Doob's Optional Stopping Theorem, we must convert (S_n) into a martingale. So let $Y_n = e^{cS_n}$, where c is the constant determined above.

Claim 2.5. *(Y_n) is a martingale with respect to $\mathcal{F}_n = \sigma(X_1, X_2, \ldots, X_n)$.*

Proof. Recall that $E(e^{cX}) = 1$. It is trivial to show that the expected value of $S_n = \sum_{i=1}^{n} X_i$ is bounded. We know that the profit for a single game is $|S_n| \leq 8n$. So $E(e^{cS_n}) < +\infty$. As for part (c) of the definition, we need to show that: $E(Y_{n+1}|\mathcal{F}_n$. Now,

$$\begin{aligned}
E(e^{cS_{n+1}}|\mathcal{F}_n) &= E\left(e^{c\sum_{i=1}^{n+1} X_i}|\mathcal{F}_n\right) \\
&= E\left(e^{c\sum_{i=1}^{n} X_i} e^{cX_{n+1}}|\mathcal{F}_n\right) \\
&= E\left(Y_n e^{cX_{n+1}}|\mathcal{F}_n\right) \\
&= Y_n E\left(e^{cX_{n+1}}|\mathcal{F}_n\right) \\
&= Y_n E\left(e^{cX_{n+1}}\right) \\
&= Y_n 1 = Y_n.
\end{aligned}$$

The fourth equality makes use of the fact that if X is F-measurable then $E\left(XY|\mathcal{F}\right) = XE\left(Y|\mathcal{F}\right)$, while the fifth equality makes use of the fact that if X is independent of $\mathcal{F}$, then $E\left(X|\mathcal{F}\right)$=$E(X)$. □

In order to use the Doob's Optional Stopping Theorem, (T_M) must meet the following conditions.

Theorem 2.6. *Doob's Optional-Stopping Theorem*

Let T be a stopping time. If the following conditions holds and X is a martingale, then $E(X_T) = E(X_0)$.

(a) *$T < \infty$ a.s.;*

(b) *$E|X_T| < \infty$;*

(c) $\underline{\lim}_{n\to\infty} \int |X_n| 1_{T>n} dP = 0.$

In our example, T_M satisfies conditions (a)-(c) because each game has a positive expected value and if $T_M(\omega) > n$ then $-B_0 < S_n(\omega) < M$, and so $\underline{\lim}_{n\to\infty} \int |Y_n| 1_{T_M>n} dP < (constant)(P(T_M > n))$. By Doob's Optional Stopping Theorem, $E(Y_{(T_M)}) = E(Y_1) = E\left(e^{cX_1}\right) = 1$. But $E(e^{cS_{(T_M)}}) = e^{c(-B_0)}P(S_{(T_M)} = -B_0) + e^{cM}P(S_{(T_M)} \geq M)$. So $P(S_{(T_M)} = -B_0)$ is the probability of going broke, which is the same as $P^{(M)}_{Broke}$. Also $P(S_{(T_M)} \geq M)$ is the probability of not going broke, which is the same as $1 - P^{(M)}_{Broke}$. If we solve for the probability of going broke, then $P^{(M)}_{Broke} = \frac{1-e^{cM}}{e^{c(-B_0)}-e^{cM}}$. When M is large, one may safely neglect the possibility that we can "overshoot" M, since the "overshoot" value is much smaller than M. Let $A_M = \{\omega \in \Omega : S_{T_M} = -B_0\}$ be the event that the player goes broke before reaching an upper stop limit of M. So $\bigcup_{M=1}^{\infty} A_M$ will be the event that the player eventually goes broke, since A_M is an increasing sequence of events. Then $P_{Broke} = P\left(\bigcup_{M=1}^{\infty} A_M\right) = \lim_{M\to\infty} P(A_M) = \lim_{M\to\infty}\left[\frac{1-e^{cM}}{e^{c(-B_0)}-e^{cM}}\right] = \frac{1}{e^{-cB_0}} = e^{cB_0} = e^{\left[\frac{-2B_0 m_1}{m_2}\right]} \equiv g\left(\frac{m_1}{m_2}\right)$, where g is a decreasing function. So to minimize P_{Broke} for a fixed bankroll, you must maximize $\frac{m_1}{m_2}$.

2.2 Strategy(b)

Now for Strategy(b), our goal is to maximize the profit after n fixed (large) bets assuming we are playing with an infinite bankroll. Again let $S_n = \sum_{i=1}^{n} X_i$ be the player's total profit after playing n games. By the Central Limit Theorem, $S_n \approx Normal(n\mu, n\sigma^2)$ where μ is the mean player's profit from one bet, and where σ^2 is the variance in profit after one bet. We want to maximize the probability when $S_n > 0$, which means we will have a positive profit after n bets. So using the central limit theorem, we calculate

$$\begin{aligned}
P(S_n > 0) &= P\left(\tfrac{S_n - n\mu}{\sqrt{n\sigma^2}} > \tfrac{0 - n\mu}{\sqrt{n\sigma^2}}\right) \\
&= P\left(Z > \tfrac{-n\mu}{\sqrt{n\sigma^2}}\right), \text{ where } Z \sim N(0,1) \\
&= P(Z > (-\sqrt{n})\left(\tfrac{\mu}{\sigma}\right)) \\
&= 1 - P(Z \leq (-\sqrt{n})\left(\tfrac{\mu}{\sigma}\right)) \\
&= 1 - \Phi\left[-\sqrt{n}\left(\tfrac{\mu}{\sigma}\right)\right], \text{ where } \Phi[x] = P(Z \leq x).
\end{aligned}$$

Obviously since n is fixed, we must maximize the fraction $\frac{\mu}{\sigma}$ to maximize the $P(S_n > 0)$. It is easy to show that the standard deviation $\sigma = \sqrt{m_2 - (m_1)^2}$, where m_1 is μ and m_2 is the second moment. So $\frac{\mu}{\sigma}$ is the same as $\frac{m_1}{\sqrt{m_2 - (m_1)^2}}$.

3. Dynamic Programming/Fractional Dynamic Programming

Dynamic programming is an approach to finding a best solution among several feasible alternatives. Dynamic programming is commonly used to find a minimum or maximum solution for applicable problems. Dynamic programming can briefly be described as a backwards iterative method in which you start at the last possible time-step and work backwards to the first time-step. Each time-step has a certain number of states that could happen. Within each state there are certain options or controls one can take to the next time-step. Each option has a probability measure, which will represent the probability for that particular option. So using dynamic programming you find the probability measure for each option at every time, then choose the best option given the featured strategy. The two strategies we found above need to maximize a fraction. In this case, fractional dynamic programming will be used. Fractional dynamic programming is branch of dynamic programming, which maximizes a fraction. The problems below will show the typical theory of fractional dynamic programming.

The fractional dynamic programming theory below comes straight from [2]. Consider the two optimization problems stated below:

Problem A $c = \max_{z \in Z} r(z) = \frac{v(z)}{w(z)}$, *where* v *and* w *are real-valued functions on some set* Z, *and* $w(z) > 0, \forall z \in Z$. *Let* Z^* *denote the set of optimal solutions to this problem. We assume that the set* Z^* *is not empty.*

The optimal solution for Problem A can be obtained by using a parametric approach, similar to the following:

Problem B $a(\lambda) = \max_{z \in Z} r_\lambda(z) = v(z) - \lambda w(z), \lambda \in R$. *Let* $Z^*(\lambda)$ *denote the set of optimal solutions to the Problem B. It is assumed that Problem B has at least one optimal solution for each* $\lambda \in R$.

The justification for seeking the solution of Problem A via Problem B is in the fact that a $\lambda \in \mathbf{R}$ exists such that every optimal solution to Problem A is also a optimal for Problem B.

Theorem 3.7. *$z \in Z^*$ if, and only if, $z \in Z^*(r(z))$.*

Proof. Part 1. $y \in Z^* \Rightarrow y \in Z^*(r(z))$.

Let y be any element of Z^*. Then, by definition $r(y) = c$, and

$$c = \frac{v(y)}{w(y)} \geq \frac{v(z)}{w(z)}, \forall z \in Z. \tag{3.1}$$

Since w is positive on Z, multiplying (3.1) by $w(z)$ yields

$$v(z) - cw(z) \leq 0, \forall z \in Z.$$

On the other hand, $r(y) = c$ entails that

$$v(y) - cw(y) = 0.$$

Therefore,

$$v(y) - r(y)w(y) \geq v(z) - r(y)w(z), \forall z \in Z \tag{3.2}$$

which in turn implies that $y \in Z^*(r(y))$. □

Proof. Part 2. $y \in Z^*(r(z)) \Rightarrow y \in Z^*$.

Let y be any element of Z such that $y \in Z^*(r(y))$. Then , by definition (3.2) holds. Also, since $r(y) = v(y)/w(y)$, the left-hand side of (3.2) is equal to zero. Thus, (3.2) entails that

$$0 \geq v(z) - r(y)w(z), \forall z \in Z. \tag{3.3}$$

Now, $w(z) > 0$, for all $z \in Z$, dividing (3.3) by $w(z)$ yields

$$r(y) \geq \frac{v(z)}{w(z)}, \forall z \in Z^*.$$

□

The following corollary is an immediate implication of Theorem 3.7.

Corollary 3.8. *The equation $a(\lambda) = 0$ has a unique solution $\lambda \in \mathbf{R}$; specifically, $a(\lambda) = 0$ if, and only if, $\lambda = c$.*

The common approach to solving the equation $a(\lambda) = 0$ would be Dinkelbach's algorithm.

Algorithm 3.9. *Dinkelbach's Algorithm*

Step 1. *Select some $z \in Z$ and set $k = 1, z^{(1)} = z$, and $\lambda^{(1)} = r(z)$.*

Step 2. *Solve Problem $B(\lambda^{(k)})$ and select some $z \in Z^*(\lambda^{(k)})$.*

Step 3. *If $a(\lambda^{(k)}) = 0$, set $z' = z$ and $\lambda' = r(z)$, and stop. Otherwise, set $z^{(k+1)} = z$ and $\lambda^{(k+1)} = r(z)$.*

Step 4. *Set $k = k + 1$ and go to Step 2.*

As you step through Dinkelbach's algorithm λ will converge to maximize $a(\lambda)$. Thus Dinkelbach's algorithm allows one to convert a problem of maximizing a

fraction into a parametrized sequence of problems in which the objective function has been linearized. If we let $c = \frac{m_1}{m_2}$, then $a(\lambda) = m_1 - \lambda m_2$. The problem of maximizing $m_1 - \lambda m_2$ can be solved by classical dynamic programming.

4. Structure of Program

We use the programming language C++ to find the risk averse strategy by converging λ with Dinkelbach's algorithm . We first initialize λ to zero. Then a recursive function deals every possible player's hand with a dealer's upcard. This process starts at the largest possible time-step of eleven cards. There can only be one possible hand at this time, which is four aces, four twos, and three threes. It is easy to see this hand has the value of twenty-one. The program then attaches a probability measure to each option for every possible hand in that particular time step. For the three to eleven card hands, the options would be hit or stand. Next, the program returns to a data structure for each option. The data structure includes the first moment, second moment, probability of winning, probability of losing, and the probability of pushing of that given option. Now, using the risk averse strategy(a), we must calculate the value of $a(\lambda) = m_1 - \lambda m_2$ for both hitting and standing. And, if using the risk averse strategy(b), we must calculate the value of $a(\lambda) = m_1 - \lambda\sigma$. The largest value of $a(\lambda)$ will be the strategy to pick in either of the two strategies. Then, we save an array containing the dealer's upcard and the player's hand in addition to the data structure of the better strategy according to our risk averse criterion into a file. Then, we iterate down to ten cards in the player's hand, and do the same process over. Again, we have two options, hit or stand. We calculate the value of standing, the same as previously. But for hitting, there is a file containing all the non-busting eleven card hand(s), with the corresponding data. So the program looks up the saved data for hitting the ten card hand. (Note: the saved files only contain non-busting hands). This process is dynamic programming. Now, we save the data, for the options that have the

larger $a(\lambda)$, for the ten card hands. Then, we iterate the time-steps down to the nine and so forth, eventually reaching the two-card hands. We then have every non-busting hand with three or more cards saved with their data into files. Now, for two card hands there will be four options instead of two. These four options are hitting, standing, splitting, and double-down. According to the rules of the game we allow a maximum of four resplits and we can double-down after resplits. So we have to take that into consideration. We then analyze the two card hands, then return an overall expected value for this strategy. Then, λ will be readjusted by using Dinkelbach's algorithm, so that we will get a better approximation of the given strategy, whether Strategy(a) or Strategy(b). This cycle continues until λ converges to give the maximum objective.

5. Results

The results from the program were not too surprising. Comparing the risk-averse strategy to one that maximizes the expected value, there are only a few differences among the two card hands. The result table below shows the expected value, variance, and the criterions of each of the three strategies. Let Strat1 be the strategy to maximize the expected profit. Let Strat2 be our risk averse strategy(a); the strategy to maximize $\frac{m_1}{m_2}$, which is equivalent to maximizing the probability of not going broke. And let Strat3 be our risk averse strategy(b); the strategy to maximize $\frac{m_1}{\sigma}$, which is equivalent to maximizing the probability of being "ahead" after a fixed number of games. We will show two examples using these three strategies. For the first example, if the player has an initial bankroll of \$100, what would be the probability that the player will not go broke? It is easy to show this probability using the formula $1 - e^{\left[\frac{-2B_0 m_1}{m_2}\right]}$ found in the subsection Strategy(a). The probability for the first example is found in the table below beside Ex1. For the second example, we will show the probability of being ahead in the game after 1000 hands, given the player has an infinite bankroll. We will find the probability for the second example using $1 - \Phi\left[-\sqrt{n}\left(\frac{\mu}{\sigma}\right)\right]$, which we showed in the subsection Strategy(b). In order to find the probability for the second example, we must look at a table for the standard normal distribution. This probability is found in the table below beside Ex2.

As stated earlier, there were only a few number of differences in the three strategies. There is only four changes in play from the first strategy to the second, and one change from the first to third. The following table will describe the differences

TABLE 5.1. Results Table

	$Strat1$	$Strat2$	$Strat3$
m_1	0.003570	0.003556	0.003569
m_2	1.318818	1.309273	1.315575
$\frac{m_1}{m_2}$	0.002707	0.002716	0.002713
$\frac{m_1}{\sigma}$	0.003109	0.003108	0.003112
$Ex1$	41.807%	41.911%	41.876%
$Ex2$	53.9159%	53.9146%	53.9197%

among the three strategies. (Note: that PC denotes the player's cards, and DU denotes the dealer's upcard).

TABLE 5.2. Strategy Table

$(PC) - DU$	$Strat1$	$Strat2$	$Strat3$
$(A, 6) - 2$	$DBL - DOWN$	HIT	$DBL - DOWN$
$(A, 2) - 4$	$DBL - DOWN$	HIT	$DBL - DOWN$
$(A, 8) - 6$	$DBL - DOWN$	$STAND$	$STAND$
$(3, 3) - 8$	$SPLIT$	HIT	$SPLIT$

Remember the above Strategy Table only describes the strategies for two card hands. So it would be a little misleading to say there are only four discrepancies. There could exist more differences in one or more of the different time steps. If would be difficult to show all the strategies for every different state and time step. Although, hypothetically our program can do this task.

Appendix: Source Code

```
/*******************************************************/
/*This program is a risk averse strategy for blackjack,*/
/*that is the property of Ryan Dutsch. No reproduction */
/*of this property can be made without permission.     */
/*Copyright 2003.                                      */
/*******************************************************/

#include <iostream.h>
#include <conio.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <search.h>
#include <math.h> // for fabs
#include <assert.h> // for assert

/******   Data Types        *********/
struct  data {
   double m1;
   double m2;
   double loseprob;
   double winprob;
   double pushprob;
}ZeroDat;

class filedata {
public:
   filedata(){};
   filedata(data*);
   int playerhand[11];
   data info;
};

#define  STD  0
#define  HIT  1
#define  DBL  2
#define  SPL  3

/******   Global variables  *********/
int Hand[11];
int DealerHand[12];
int Deck[11];
int CardsInDeck;
int rank;
int splitctr;
double dist[6];
int playertotal;
int TotalCards;
const int maxsplit=4;
const int MaxCards[11] =
   { 0,4,4,4,4,4,3,3,2,2,2};
const int MaxHand[11] =
   { 0,4,4,4,4,4,4,4,4,4,16};
const char beststrategy[4][10] = {"STAND", "HIT", "DBL-DOWN", "SPLIT"};
const char rankchar[12] = " A23456789T";
FILE * CardDataFile;
FILE * TxtOutFilePtr;
const char   TextFileName[20] = "BJ2CardStrategy.txt";
double lambda;
data AccumDat;
filedata* lastrounddata;
unsigned int filectr;
unsigned int LastRoundCtr;
int MostCards;

/******   Function Prototypes ******/
void Fitk(int , int );
void DoSomething( void );
int min( int, int);
int ValueofHand( int localhand[11] );
int dealertotalup(const int* );
int waitforkeypress( void );
void RemainDeck( void );
void dealcard(double, int);
data stand(void);
data hit(void);
int compare( const void *handone, const void *handtwo );
void analyzegame( void );
void playerhandloop ( void );
double criterion(data*);
data doubledown(void);
data split(void);
void ReadFinalOutput(void);
```

```
main()
{  double oldlambda=1;
   lambda= 0;

   TxtOutFilePtr = fopen(TextFileName, "wt");
   if(TxtOutFilePtr == NULL)
   {  printf("ERROR:  Couldn't open output text file.\n");
      return(1);
   }
   while(lambda != oldlambda)
   {  oldlambda = lambda;
      fprintf(TxtOutFilePtr, "Lambda = %8.6lf\n", lambda);
      for (int i=0; i<=10; i++) Deck[i]=MaxHand[i];
      AccumDat = ZeroDat;
      analyzegame();
      lambda = AccumDat.m1 / AccumDat.m2;
      fprintf(TxtOutFilePtr, "M1 = %8.6lf\t M2 = %8.6lf\t", AccumDat.m1, AccumDat.m2);
      fprintf(TxtOutFilePtr, "Standard Deviation = %8.6lf", sqrt(AccumDat.m2-(AccumDat.m1*AccumDat.m1)));
      fprintf(TxtOutFilePtr, "\n\n**********************************************\n\n");
   }
   ReadFinalOutput();
   return 0;
}

void Fitk(int cards, int pointer)
{  if (cards==0)
   {  for (int i=pointer; i<11; i++) Hand[i]=0;
      DoSomething();
      ValueofHand(Hand);
      //waitforkeypress();
   }else if( (pointer==10)&&(cards<= min(Deck[10], MaxCards[10]) ) )
   {  Hand[pointer] = cards;
      DoSomething();
      ValueofHand(Hand);
      //waitforkeypress();
   }else
   {  if (pointer < 10)
      {  int k = min(min(cards, Deck[pointer]), MaxCards[pointer]);
 for(int j=k; j>=0; j--)
         {  Hand[pointer]= j;
    Fitk(cards-j, pointer+1);
         }
      }
   }
}

void DoSomething()
{  double crithit, critstand, critmax;
   static count = 1;
   int value=ValueofHand(Hand);
   if(value<=21)
   {  cout<<count++<<".  ";
      for (int i=1; i<=10; i++) cout << Hand[i]<<" ";
      cout<<value<<" ";
      if (value>21) cout<<"BUST";
      cout<<endl;
      for(i=1; i<=10; i++)
         Deck[i] -= Hand[i];
      CardsInDeck=0;
      for(i=1; i<=10; i++) CardsInDeck += Deck[i];
      int numcardsinhand =  51-CardsInDeck;
      int strategy = STD;
      data standdat = stand();
      data hitdat;
      data* ptrbestdat;
      if( numcardsinhand <MostCards)
      {  hitdat = hit();
         crithit = criterion(&hitdat);
         critstand = criterion(&standdat);
         if (crithit>critstand) {
            ptrbestdat = (&hitdat);
            critmax = crithit;
            strategy = HIT;
         }
         else {
            ptrbestdat = (&standdat);
            critmax = critstand;
            strategy = STD;
         }
         if (numcardsinhand==2)
         {  data doubledowndat = doubledown();
            double critdoubledown = criterion(&doubledowndat);
            if (critdoubledown>critmax) {
               ptrbestdat=(&doubledowndat);
               critmax=critdoubledown;
               strategy = DBL;
            }
            for (int g=1; g<=10; g++)
            {  if (Hand[g]==2)
            {
```

```
            splitctr=1;
            rank=g;
            data splitdat=split();
            double critsplit = criterion(&splitdat);
            if (critsplit>critmax)
            {  ptrbestdat=(&splitdat);
               critmax=critsplit;
               strategy = SPL;
            }
         }
      }
      cout<<count++<<".  ";
      for (int i=1; i<=10; i++) cout << Hand[i]<<" ";
      cout<<value<<" ";
      if (value>21) cout<<"BUST ";

      cout<<beststrategy[ strategy ] <<endl;

      /*while(kbhit()) getch();
      char ch;
      while(!kbhit());
      ch = getch();   */

      /***  Adjust probs for dealer and player naturals  ***/
      double dnat = 0;
      if( Hand[0]==1 ) dnat = ((double)Deck[10])/CardsInDeck; //dealer's upcard = ace
      if( Hand[0]==10 ) dnat = ((double)Deck[1])/CardsInDeck; //dealer's upcard = ten

      data adjusteddata;
      adjusteddata.winprob = (ptrbestdat->winprob)*(1-dnat);
      adjusteddata.loseprob= (ptrbestdat->loseprob)*(1-dnat) + dnat;
      adjusteddata.pushprob = (ptrbestdat->pushprob)*(1-dnat);
      adjusteddata.m1 = (ptrbestdat->m1)*(1-dnat) - dnat;
      adjusteddata.m2 = (ptrbestdat->m2)*(1-dnat) + dnat;

      if( (Hand[1]==1)&&(Hand[10]==1) ) //player has a natural
      {
         adjusteddata.winprob = 1-dnat;
         adjusteddata.pushprob = dnat;
         adjusteddata.loseprob = 0;

         adjusteddata.m1 = (1.5)* adjusteddata.winprob;
         adjusteddata.m2 = (1.5)*(1.5) * adjusteddata.winprob;
      }
      int lowrank =1;
      while(Hand[lowrank]==0) lowrank++;
      int hirank = 10;
      while(Hand[hirank]==0) hirank--;

      int cntlow = MaxHand[lowrank]-(Hand[0]==lowrank);
      int cnthi  = MaxHand[hirank]-(Hand[0]==hirank);

      double stateprob = (lowrank==hirank)?
         ((double) MaxHand[Hand[0]]*cntlow*(cntlow-1) )/ (52*51*50):
         ((double)  MaxHand[Hand[0]]*cntlow*cnthi )     / (52*51*25);

      AccumDat.m1 += adjusteddata.m1*stateprob;
      AccumDat.m2 += adjusteddata.m2*stateprob;

      AccumDat.winprob += adjusteddata.winprob*stateprob;
      AccumDat.loseprob += adjusteddata.loseprob*stateprob;
      AccumDat.pushprob += adjusteddata.pushprob*stateprob;

      fprintf( TxtOutFilePtr, "  %c,%c%12s%12.4lf%12.4lf\n",
      rankchar[lowrank], rankchar[hirank],
      beststrategy[ strategy ], adjusteddata.m1, adjusteddata.m2);
      }
   }
   else ptrbestdat=&standdat;
   filedata george(ptrbestdat);
   //if(value<=21)
   {  fwrite(&george, sizeof(filedata), 1, CardDataFile);
      filectr++;
   }
   if( (numcardsinhand==2)&&(Hand[1]==1)&&(Hand[10]==1) )
   // player has a natural
   {
   }
   for(i=1; i<=10; i++)
   Deck[i] += Hand[i];
   }
}

inline int min(int j, int k) { return( (j<k)? j: k); }

int ValueofHand( int localhand[11])
{  int value=0;

   for (int j=2;j<=10;j++)
   {  value=j*localhand[j]+value;
```

```
   }
   if ((localhand[1]!=0) && (value <= 11-localhand[1]))
   {  value=11+(1*(localhand[1]-1))+value;
      playertotal=value;
   } else
   {  value=value + localhand[1];
      playertotal=value;
   }
   return(value);
}

int  waitforkeypress( void )
{  while (_kbhit() ) _getch();
   while (!_kbhit() );
   return _getch();
}

int dealertotalup( const int*  hand)  //starts summing at index=1, not 0
// requires hand array to have 0's beyond end of hand
{  int dvalue=0;
   bool acepresent = false;
   for(int j=0;j<=11;j++)
   {  int card = hand[j];
      dvalue+= card;
      if(card==1) acepresent = true;
      if( !card ) break;
   }
   if(acepresent&&(dvalue<=11))
      dvalue += 10;
   return dvalue;
}

void RemainDeck(void)
{  CardsInDeck = 0;
   for(int i=1;i<=10;i++)
   {  Deck[i]=MaxHand[i]-DealerHand[i];
      CardsInDeck += Deck[i];
   }
}

void dealcard (double prob, int ptr)
{  int adjdecksize=CardsInDeck;
   int start=1;
   if ((ptr==1) && (DealerHand[0]==10))
   {  start=2;
      //adjdecksize=CardsInDeck-(MaxHand[1]-Hand[1]);
      adjdecksize=CardsInDeck-(Deck[1]);
   }
   int stop=10;
   if ((ptr==1) && (DealerHand[0]==1))
   {  stop=9;
      //adjdecksize=CardsInDeck-(MaxHand[10]-Hand[10]);
      adjdecksize=CardsInDeck- Deck[10];
   }

   double sumprob = 0;
   int sumcard = 0;

   for (int card=start; card<=stop; card++)
   if ( Deck[card] !=0)
   {  //  RemainDeck();
      double p= prob* ((double) Deck[card]/adjdecksize);
      sumprob += p;
      sumcard += Deck[card];

      Deck[card]--;
      CardsInDeck--;

      DealerHand[ptr]=card;
      int value=dealertotalup(DealerHand);
      if (value<17) dealcard(p, ptr+1);
      else if (value<22) dist[value-17] += p;
      else dist[5] += p;

      Deck[card]++;
      CardsInDeck++;

      for(int j=ptr; j<=11; j++) DealerHand[j] = 0;
   }
   if(sumcard != adjdecksize)
   cout<<"Bug in dealcard"<<endl;
   if( fabs(sumprob-prob) > 0.0000001)
   cout<<"Bug in dealcard"<<endl;
}

data stand(void)
{  data std;

   for (int k=0; k<=5; k++) dist[k]=0;
   dealcard(1,1);
```

```
   std.winprob=0;
   std.loseprob = 0;
   std.pushprob=0;

   if (playertotal>21)  std.loseprob=1;

   else if (playertotal>=17)
   {  int q = playertotal-17;

      std.pushprob=dist[q];
      for (int i=0; i<q; i++)
         std.winprob  +=dist[i];

      std.winprob += dist[5];

      for ( i=4; i>q; i--)
         std.loseprob  +=dist[i];
   }
   else
   {  std.winprob = dist[5];
      std.loseprob = 1-std.winprob;
   }
   std.m1 = std.winprob - std.loseprob;
   std.m2= std.winprob + std.loseprob;

#ifdef _DEBUG
   double sum = 0;
   sum = std.loseprob + std.winprob + std.pushprob;
   if( fabs( sum-1 ) > 0.0000001 )
   {  cout<<"Stand"<<Hand[0]<<Hand[1]<<Hand[2]<<Hand[3];
      cout<<endl;
   }
#endif
   return( std );
}

data hit(void)
{  data hitdat;
   hitdat.m1 = hitdat.m2 = hitdat.loseprob = hitdat.pushprob = hitdat.winprob = 0;
   filedata newhand;

   for(int i=0; i<11; i++)  newhand.playerhand[i] = Hand[i];
   //  deal one more card to current player hand
   for(int card = 1; card <= 10; card++)
   if(Deck[card]>0)
   {  newhand.playerhand[card]++;
      double prob = ((double)Deck[card]) / CardsInDeck;
      //  find hand in lastrounddata

      if(ValueofHand(newhand.playerhand) <=21)
      {  filedata * founddat = (filedata *) bsearch( &newhand,lastrounddata,
         LastRoundCtr, sizeof(filedata), compare);

         hitdat.winprob += founddat->info.winprob * prob;
         hitdat.loseprob += founddat->info.loseprob * prob;
         hitdat.pushprob += founddat->info.pushprob * prob;
         hitdat.m1 += founddat->info.m1 * prob;
         hitdat.m2 += founddat->info.m2 * prob;
      }
      else //hand is busted
      {  hitdat.loseprob += prob;
         hitdat.m1 -= prob;
         hitdat.m2 += prob;
      }
      // add the found data into hitdat, with probability weighting of card
      newhand.playerhand[card]--;
   }

#ifdef _DEBUG
   double sum = 0;
   sum = hitdat.loseprob + hitdat.winprob + hitdat.pushprob;
   if( fabs( sum-1 ) > 0.0000001 )
   {  cout<<"Hit"<<Hand[0]<<Hand[1]<<Hand[2]<<Hand[3];
      cout<<endl;
   }
#endif
   return(hitdat);
}

int  compare( const void *handone, const void *handtwo )
{  int *ptr1 = ((filedata *) handone)->playerhand;

   int *ptr2 = ((filedata *) handtwo)->playerhand;

   for (int i=1;i<=10;i++) {
      if (ptr1[i]<ptr2[i]) return (-1);
      else if (ptr1[i]>ptr2[i]) return (1);
   }
   return 0;
```

```
}

void analyzegame(void)
{  int dealercard;

   for (int i=1; i<=10; i++)
   {  dealercard=i;
      Hand[0]=DealerHand[0]=dealercard;
      Deck[i]--;

      fprintf( TxtOutFilePtr, "\nDealer Upcard = %c\n\n", rankchar[dealercard]);
      playerhandloop();

      Deck[i]++;
   }
}

double criterion( data* dat)
{  double crit=dat->m1 -(lambda*(dat->m2));
   return crit;
}

data doubledown(void)
{  data doubledowndat;
   doubledowndat.m1 = doubledowndat.m2 = doubledowndat.loseprob =
      doubledowndat.pushprob = doubledowndat.winprob =0;

   for(int card = 1; card <= 10; card++)
   if(Deck[card]>0)
   {  Hand[card]++;
      double prob = ((double)Deck[card]) / CardsInDeck;
      Deck[card]--;
      CardsInDeck--;

      if(ValueofHand(Hand) <=21)
      {  data founddat1 = stand();

         doubledowndat.winprob += founddat1.winprob * prob;
         doubledowndat.loseprob += founddat1.loseprob * prob;
         doubledowndat.pushprob += founddat1.pushprob * prob;
         doubledowndat.m1 += founddat1.m1 * prob;
         doubledowndat.m2 += founddat1.m2 * prob;
      }
      else //hand is busted
      {  doubledowndat.loseprob += prob;
         doubledowndat.m1 -= prob;
         doubledowndat.m2 += prob;
      }
      Hand[card]--;
      Deck[card]++;
      CardsInDeck++;
   }
   doubledowndat.m1 *= 2;
   doubledowndat.m2 *= 4;

#ifdef _DEBUG
   double sum = 0;
   sum = doubledowndat.loseprob + doubledowndat.winprob + doubledowndat.pushprob;
   if( fabs( sum-1 ) > 0.0000001 )
   {  cout<<"Dbl-Down"<<Hand[0]<<Hand[1]<<Hand[2]<<Hand[3];
      cout<<endl;
   }
#endif
   return doubledowndat;
}

data split (void)
{  data *bestdat;
   data splitdat;
   splitdat.m1=splitdat.m2=splitdat.loseprob=splitdat.pushprob=splitdat.winprob=0;
   Hand[rank]--;

   //filedata temphand(&splitdat);
   for(int card = 1; card <= 10; card++)
   {
      if(Deck[card]>0)
      {
         Hand[card]++;
         double prob = ((double)Deck[card]) / CardsInDeck;
         Deck[card]--;
         CardsInDeck--;
         //temphand.playerhand[card]++;
         assert(ValueofHand(Hand) <=21);

         data spltstddat=stand();
         data splthitdat=hit();
         data spltdbledat=doubledown();
         double critspltstd=criterion(&spltstddat);
         double critsplthit=criterion(&splthitdat);
         double critspltdble=criterion(&spltdbledat);
```

```
         double maxcrit = critspltstd;
         bestdat = &spltstddat;
         if(critsplthit>maxcrit)
         {  maxcrit = critsplthit;
            bestdat = &splthitdat;
         }
         if(critspltdble>maxcrit)
         {  maxcrit = critspltdble;
            bestdat = &spltdbledat;
         }

         if ((card==rank) && (splitctr<maxsplit)) {
            splitctr++;
            data newsplitdat=split();
            //calculate critnewslitdat
            double critsplit=criterion(&newsplitdat);

            if(critsplit>maxcrit)
            {  maxcrit = critsplit;
               bestdat = &newsplitdat;
            }
         }
         splitdat.m1 += bestdat->m1 * prob;
         splitdat.m2 += bestdat->m2 * prob;
         splitdat.winprob += bestdat->winprob * prob;
         splitdat.loseprob += bestdat->loseprob * prob;
         splitdat.pushprob += bestdat->pushprob * prob;

         Hand[card]--;
         Deck[card]++;
         CardsInDeck++;
      }
   }
   splitdat.m2 =(2*splitdat.m2)+(2*(splitdat.m1*splitdat.m1));
   splitdat.m1=(2*splitdat.m1);
   Hand[rank] ++;
   return (splitdat);
}

void playerhandloop ( void )
{  if ((Hand[0]==1) || (Hand[0]==2)) MostCards=10;
   else MostCards=11;
   for (int i=MostCards; i>=2; i--)
   {
      CardDataFile = fopen("test.dat", "wb");
      if(CardDataFile==NULL)
      {
         cout<<"Couldn't open test file!";
         exit(1);
      }

      filectr=0;

      Fitk(i, 1);
      fclose(CardDataFile);
      LastRoundCtr = filectr;

      if (i<=10) delete [] lastrounddata;

      lastrounddata = new filedata[filectr];
      // CardDataFile.open("test", ios::in|ios::out|ios::binary);
      CardDataFile = fopen("test.dat", "rb");
      if(CardDataFile==NULL)
      {
         cout<<"Couldn't open test file!";
         exit(1);
      }

      fread(lastrounddata, sizeof(filedata), filectr, CardDataFile);
      fclose(CardDataFile);

      qsort (lastrounddata, filectr, sizeof(filedata), compare);
   }
}

filedata::filedata(data* dat)
{  for (int i=0;i<=10;i++) playerhand[i]= Hand[i];
   info.m1=dat->m1;
   info.m2=dat->m2;
   info.loseprob=dat->loseprob;
   info.pushprob=dat->pushprob;
   info.winprob=dat->winprob;
}

void ReadFinalOutput(void)
{  fclose( TxtOutFilePtr );

   char command[40] = "NotePad ";
   strcat(command, TextFileName);
```

```
   system( command );
}
```

References

[1] Kai Lai Chung. *A Course in Probability Theory*. Academic Press, San Diego, CA, 2001.

[2] Moshe Sniedovich. *Dynamic Programming*. Marcel Dekker, Inc., New York, NY, 1992.

[3] David Williams. *Probability with Martingales*. Cambridge University Press, Cambridge, Great Britain, 1991.

[4] Peter A. Griffin. *The Theory of Blackjack. Huntington Press*, Las Vegas, NV, 1988.

[5] Stanford Wong. *Professional Blackjack*. Pi Yee Press, La Jolla, CA, 1994.

[6] Lance Humble and Carl Cooper. *The World's Greatest Blackjack Book*. Doubleday Dell Publishing Group, Inc., New York, NY, 1987.

Vita

Ryan Dutsch was born on December 4, 1977, in Baton Rouge, Louisiana. He finished his undergraduate studies at Southeastern Louisiana University in the summer of 2000. In August 2000, he came to Louisiana State University to pursue graduate studies in Mathematics. He is currently a candidate for a degree of Master of Science in Mathematics. He is working with George Cochran on a discrete time control problem.