Parabola Volume 55, Issue 1 (2019)

Random walks: an application for detecting bias in Spider Solitaire programs

Trevor Chi-Yuen Tao[1]

1 Introduction

Games such as Klondike, Freecell and Spider Solitaire are popular pastimes for players, available on personal computers or phones. However, a common complaint is these software programs can cheat if the player's win rate is too high. For instance, Microsoft Hearts (a game for four players) has been considered unfair by some players since they receive too many bad hands if the win rate exceeds a certain threshold. Such claims are often difficult to substantiate without supporting evidence, and it is possible that players are playing below their best or have "selective memories" (e.g., they are more apt to remember painful losses than easy wins). A particularly striking example of false accusation is Backgammon NJ for the Android phone. Many weak players accused the AI of cheating by loading the dice, and the programmer(s) were forced to provide evidence such as computer rollouts to refute such accusations.[2] Fortunately, basic knowledge of statistics and probability is enough to avoid unfounded accusations. It is now common knowledge that many Backgammon programs such as TDGammon [1] can compete with or even outperform the best human players.

2 Random Walks

2.1 Movement between states

Let us assume that a computer scientist is new to Chess and that their task is to determine which side has an advantage given a game state. One idea may be to pretend that two monkeys make random moves for both sides until the game ends. To be more specific, in any game state S we can compute N, the number of legal options available for the side to move. We can then stipulate that each legal move occurs with probability $1/N$. We can simulate many games of random moves starting from state S and tally the results. For instance, we might find that in 1000 games, White wins 400 games, draws 300 and loses 300. Since White had more wins than losses, we might guess that the game state S favours White. Assuming one point for a win and half a point for

[1] Trevor Tao has a PhD in applied mathematics. He is a research scientist currently working for the Australian Department of Defence. Trevor is a keen chess and scrabble player and his other hobbies include mathematics and music. Trevor is the brother of world-renowned mathematician Terence Tao.

[2] http://www.njsoftware.com/note.html

a draw, White will expect to win 0.55 points per game so we can say that S has an equity of 0.55 for the White player. The sequence of moves from S to a final position is known as a *random walk*. Obviously, this example is impractical for tournament games because human players do not make random moves. But the important point is that we are able to (in theory) associate an equity to any game state.

A simpler example of a random walk is the well-known phenomenon of Gambler's Ruin. Suppose that a gambler starts with \$100 and wins or loses a dollar with probability 0.5. The gambler continues to play until either broke or has doubled their capital. Any state of the random walk can be represented by the gambler's current capital. If we define the equity of a state to be the probability of the gambler reaching \$200, then it is easily shown that the initial equity is 0.5. However, if the probability of winning a dollar were reduced to 18/38 (e.g., betting on Red on a Roulette Wheel with double-zero), then the equity becomes almost zero! One can verify this with some algebra or computer simulation. It turns out that

$$E(I) = \begin{cases} \dfrac{i}{200} & \texttt{if } \; p = 0.5\,; \\ \dfrac{p^i - (1-p)^i}{p^{200} - (1-p)^{200}} & \texttt{otherwise}\,. \end{cases}$$

where E is the equity, p is the probability of winning an individual game and i is the gambler's capital.

2.2 Evaluating individual states

Incidentally, assigning a value to game states is nothing new. A familiar example is the minimax or alpha beta pruning algorithm [2] to determine the best move in two-player strategy games. Although such algorithms are outdated by modern standards, they are often used in teaching computer science courses at undergraduate level. Each state in a tree receives a "static" evaluation which does not depend on neighbouring game states. For instance, in chess we can evaluate features such as piece mobility, control of the centre and king safety. One problem is that choosing which features to include for evaluation is non-trivial. Also, it is difficult to quantify the features described above. In contrast, random walks avoid such problems since we only need to enumerate the set of legal moves in any game state. Of course, I do not claim that random walk algorithms can compete with the likes of Magnus Carlsen or Garry Kasparov! Nevertheless, random walk methods have found serious use in many applications. Examples include Brownian motion [3], sampling social networks [4], text classification [5] and animal movement [6].

3 Spider Solitaire

3.1 Win Rate for Games With/Without Undo

Spider Solitaire is a card game for one player. There are two decks of cards and the player aims to organise cards into eight complete suits and remove them from the tableau. Cards can be dealt from the stock or moved from one column to another according to certain rules. The rules will not be discussed in detail; it is assumed the reader already has some knowledge of rules and basic strategy since this information is easily found online. Although many researchers have analysed Spider Solitaire, it is tacitly assumed that undo is allowed, and the task is to find at least one sequence of moves that results in victory. It turns out that nearly every game is winnable, even at the 4-suit level (c.f. [7] and references therein). I am unaware of any research into the probability of winning a game without undo.

There are two fundamental ways in which to modify the difficulty of winning a game of Spider. Firstly, the number of suits can be 1, 2 or 4 with more suits indicating higher difficulty, but this is always determined by the player. The second way is to rearrange the starting layout. As an extreme example, it is easy to construct an initial game state so that no face-down cards in the tableau can be exposed with any sequence of legal plays. In general, this is not controlled by the player, and they may suspect that the cards are stacked after a series of losses.
Most players find it difficult to win games at the 4-suit level. But Steve Brown gives some detailed strategies in [8] and reports a win rate of 48.7% over 306 games. Moreover, he achieves this without undoing any moves, restarting or rejecting games with poor initial state. He also points out his play is not perfect and an expert player could do even better, perhaps with a win rate in excess of 60%. By experimenting with Brown's strategies, I am indeed able to achieve a win rate in excess of 48.7% without undo, restart or rejecting games.

3.2 The ideal unbiased Spider Solitaire program

We know that Spider Solitaire is played with 2 decks of standard playing cards. If we ignore the equivalence of cards with the same rank and suit, the number of possible starting states is $104! = 104 \times 103 \times \cdots \times 1 \approx 1.03 \times 10^{166}$. It is natural to assume that each starting game state will occur with equal probability. Certainly, this is what one would expect if the game was played manually and assuming perfect shuffling. Let us say that a Spider Solitaire program that assigns equal probability to all starting states is *unbiased*.

During the course of the game, a player will expose cards, either by turning over face-down cards in the tableau or dealing from the stock. The probability distribution of possible initial game states is updated accordingly whenever a move exposes at least one more card. This allows one to theoretically compute the probability of winning any game state, given perfect play. For instance, suppose that in the endgame only

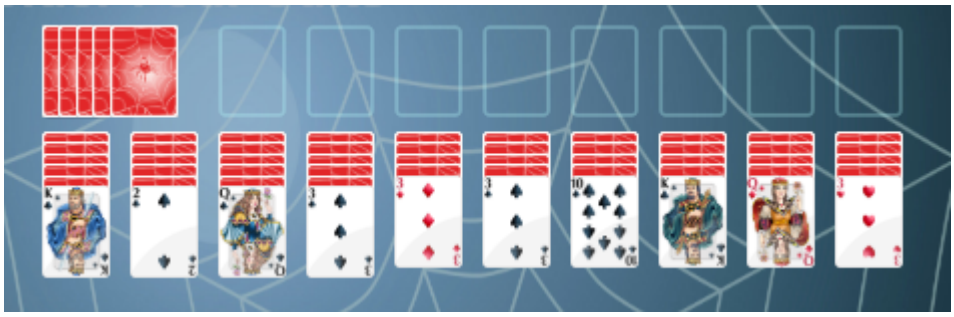

Figure 1: An example initial stage with 8 legal moves and 3 guaranteed turnovers.

five (5) cards are unseen, one of which is the Queen of Spades. Assume that an expert player determines that a win is guaranteed if the next exposed card is the Spade Queen. Then the chance of victory is at least 20%. If any card other than the Queen of Spades guarantees a loss, then the chance of victory is exactly 20%. Of course in practice, it is impossible to compute winning probabilities with many cards unseen, and human players must "guess" the optimal line of play.

In an earlier report [9], I investigated whether a Spider Solitaire program can be proved biased using the static evaluation methods described in Section 2. For instance, given an initial game state, we can compute the number of "guaranteed turnovers" (see Figure 1) even if the worst possible cards were exposed. However, I did not find enough evidence to prove that it was biased.

4 Experiment

4.1 Free Spider Solitaire Server

We wish to answer the following question: if a player wins too many games, does Spider Solitaire then arrange the cards to make it harder for the player to win?

I played 40 games of Spider on the Free Spider Solitaire (FSS) server[3] at the 4-suit level. For purposes of this report, I allowed myself to undo moves for two reasons: firstly, this ensured that I was able to win every game, thus giving the program more "incentive" to stack the cards unfavourably in later games and hence more chance of proving that the program is indeed biased. Secondly, I needed to record the identity of every face-down card so I could simulate random walks as described above and compute the equity. If the program was unbiased, I would expect every hand to have random equity. But if the program was biased, then I would expect games would become harder to win as I accumulated more victories.[4] As an (admittedly contrived)

[3] https://www.free-spider-solitaire.com/

[4] Note that with this methodology it is impossible for the player to change the equity of an individual game by making sub-optimal moves, intentionally or otherwise.

analogy, suppose you participate in a Chess tournament in which each game had a computer-generated random initial state instead of the usual starting configuration. After completing the tournament you review the games and simulate random moves as described above. You then find that the initial states always favoured a grandmaster whenever his opponent was not a grandmaster. Even though the random moves from both sides were far from optimal(!), you would probably suspect something was wrong with the program that generated the initial states. A similar method can be used to detect bias in Spider Solitaire.

4.2 Spider Monkey Algorithm

We first need to define a Spider Monkey algorithm that makes random moves. At any game state, we can enumerate a set of legal moves and specify that each occurs with equal probability. The only exception is dealing a fresh row of 10 cards which is clearly undesirable unless "no further progress is possible". However, determining if progress is possible is not trivial, so I simply imposed a move limit of 1000. More specifically, the algorithm is

```
for round = 1, 2, 3, 4, 5, 6 LOOP
    for iter = 1, 2, ..., 1000 LOOP
        Enumerate all legal moves (except dealing from the stock).
        Let N be the number of such moves.
        if N > 0, then choose a random legal move,
            assigning each move a probability of 1/N.
    END LOOP
    if round < 6, then deal 10 cards from the stock.
    if round = 6, then
        if all eight suits have been removed, then declare VICTORY;
        otherwise, concede DEFEAT.
END LOOP
```

We also assume that

(a) SpiderMonkey will never split an in-suit sequence if the destination column is empty. For instance, the 6-5 from an exposed 9-8-7-6-5 can be moved to another 7, but not to an empty column.

(b) SpiderMonkey will never shift all cards in a source column to an empty destination column (this is to speed up a winning endgame).

(c) SpiderMonkey can deal 10 cards from the stock even with one or more empty columns (this avoids a stalemate if the monkey removes too many suits prematurely).

With these specifications it is easily seen that any legal move can be uniquely identified by specifying a source and destination column. For instance, in Figure 1 the legal moves are `bd, be, bf, bj, ca, ch, ia, ih`, where columns are labelled `a-j` from left to right. Each move occurs with probability $1/8$.

To ensure that the monkey wins a fair percentage of games, I gave it a large handicap: I pretended that each game was played at the 1-suit level! Preliminary experimentation showed that a monkey making random moves can win more than half the time at the 1-suit level. This is the key observation that allows the random walk method to work.

4.3 Results

For each of the forty 4-suited hands played on FSS, I iterated the Spider Monkey algorithm 100 times at the 1-suit level so the estimated equity was a fraction of the form $N/100$ for some integer N. I defined an "inversion" as any pair of games such that the latter game had lower equity than the former [10]. Thus, no inversions would occur if the games were arranged in order of ascending equity, and similarly the maximum number would occur if the games were arranged in the reverse order. If two games had exactly the same equity, then I simply added "small random noise" as a tiebreaker.

In my experiment, I got 468 inversions for the equities shown in Figure 2. It is clear that a downward trend exists, with early games starting with high equities and later games gradually shifting towards lower values. Simulations reveals that if the program were unbiased, then the probability of 468 or more inversions occurring in 40 games is $0.036 < 0.05$, so we have reason to believe that the program is biased. In statistical language, we say that 468 inversions is significant at the $\alpha = 0.05$ level, and the null hypothesis (i.e., that the program is not biased) is rejected [11].

5 Discussion

Due to the random nature of the experiment, the results are not "replicable" in the usual sense but I would expect that a student with programming experience should be able to reproduce similar results by implementing the Spider Monkey algorithm as described above. A few example hands that I obtained with FSS with associated equities are shown in Figure 3. Each row of text corresponds to a column in FSS. For convenience, I have inserted spaces between the tableau and the stock. Note that only the ranks are shown since Spider Monkey is playing at the 1-suit level.

I have not discussed how or why a Spider program would be designed to arrange the cards to make it harder to win if the player has a win rate. It would certainly be legitimate for someone to ask what a software developer would gain by doing this. My best guess is that the designer of FSS wants to avoid giving a strong player hands that are too easy. Despite these good intentions, I nevertheless believe it is unethical to alter the difficulty level without explicitly warning the user. An example of a program

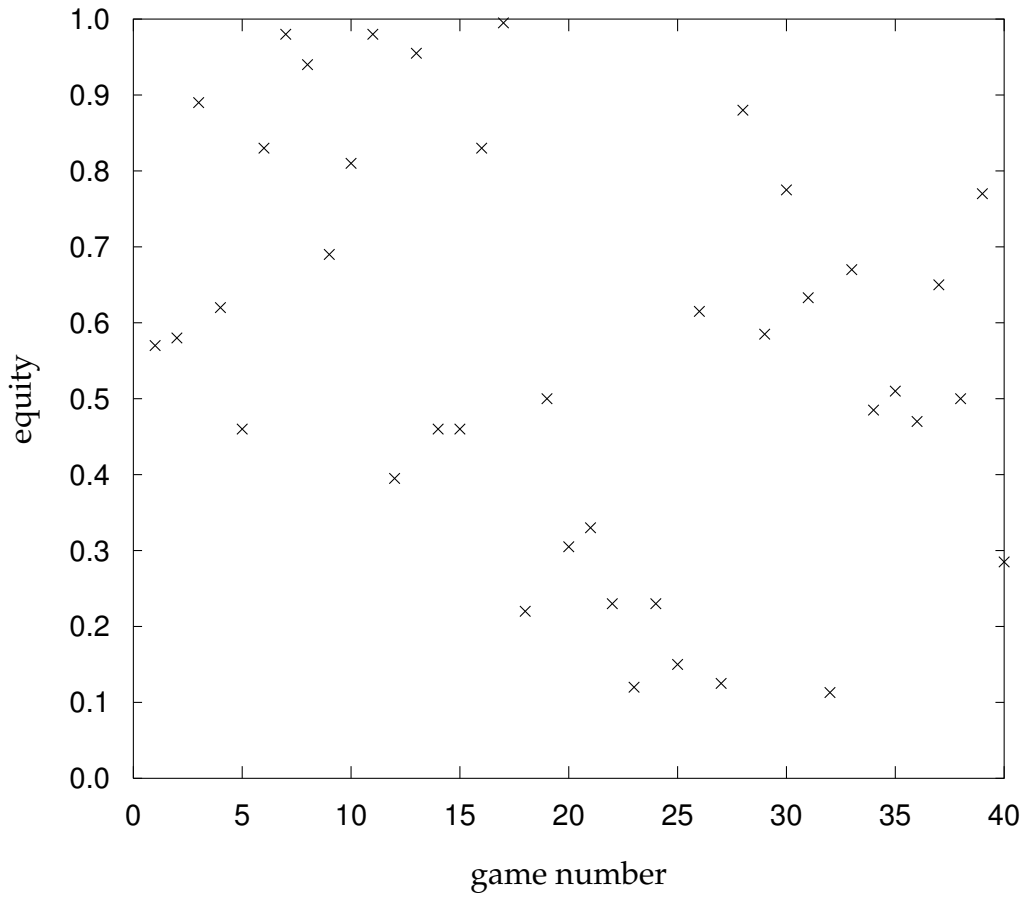

Figure 2: Equities for 40 games of Spider played on FSS.

```
Game 3, equity = 0.88  Game 15, equity = 0.45  Game 32, equity = 0.11
 4jq82k  qq86a          4k3k62  9277k           23428j  24750
 j5k240  j90k4          3q028k  04j6a           65669j  99506
 2aa307  62kqq          7j43a5  59j6j           j60ajq  87qkq
 260934  6j942          600ak3  338qq           24q6q3  69098
 j7ak3   a8q94          9j37k   8kj87           82753   3525a
 k6q54   a6590          25869   94527           8k990   jq8k3
 7788k   776k9          aq539   7a502           8k073   k3aa4
 j4569   335j3          47495   k5q0q           24k6k   k487q
 8q755   89a2a          q80a8   a6q64           2j4j5   j7a90
 38j02   70530          906a2   j824j           34q5a   770aa
```

Figure 3: Example hands with equities.

with warning is the Microsoft Windows 10 version of Spider Solitaire. The user has the option of selecting various difficulty levels, confident in the knowledge that deals cannot be unbiased.

References

[1] G. Tesauro, Temporal difference learning and TD-gammon, *Commun. ACM* **38** (3) (1995), 58–68.

[2] J. Pearl, The solution for the branching factor of the alpha-beta pruning algorithm and its optimality, *Commun. ACM* **25** (8) (1982), 559–564.

[3] D.T. Gillespie, The mathematics of Brownian motion and Johnson noise, *Am. J. Phys.* **64** (3) (1996), 225–240.

[4] J. Lu and D. Li, Sampling online social networks by random walks, in *Proceedings of the First ACM International Workshop on Hot Topics on Interdisciplinary Social Networks Research*, HotSocial '12, ACM, pp. 33–40, New York, NY, 2012.

[5] Y. Xu, X. Yi, and C. Zhang, A random walks method for text classification, in *Proceedings of the SIAM International Conference on Data Mining*, pp. 340–347, 2006.

[6] P.F.C. Tilles, S.V. Petrovskii, and P.L. Natti, A random walk description of individual animal movement accounting for periods of rest, *Open Science* **3**(11) (2016), 16 pp.

[7] M. Weisser, *How Many Games of Spider Solitaire are Winnable? Explorations Into The Mathematics Underlying Spider Solitaire*, Graduate Liberal Studies Works (MALS/MPhil), 2012.

[8] S.N. Brown, *Spider Solitaire Winning Strategies*, Private Publisher, US, 2016.

[9] T. Tao, Spider solitaire, *Plus* (2018), https://plus.maths.org/content/spider-solitaire.

[10] B. Margolius, Permutations with inversions, *J. Integer Seq.* **4** (2001), Article 01.2.4.

[11] P. Sedgwick, Understanding statistical hypothesis testing, *BMJ* **348** (2014), g3557.