Solitaire Adventures

Teresa Dong & Henry Dong

Abstract

The goal of this project is to build a playable solitaire simulation and statistically analyze the winrate of the implemented algorithms. Reverse column sequence is by far the most important factor in determining win rate %. Every single one of the top 20 permutations follows the same exact column sequence from right to left (tallest to shortest stacks). Given the importance of column sequence, it's no surprise that permuting flipped piles between Tableau columns appears to be the most important first move as well.

Background

Solitaire is famous among mathematicians for their inability to *precisely* calculate the odds of winning (Diaconis 1999) a game. The number of different solitaire games is $52! \approx 8x10^{67}$. Given this sheer number of possible games, we must rely on simulation to approximate the probability of winning.

In this project, we leverage Python to implement a playable solitaire simulation and evaluate a number of algorithms to win our game. The performance of our simulation is benchmarked against the performance of world-class solitaire players (our parents) to determine its real-world efficacy.

First, we'll cover the implementation details of our solitaire simulation setup. Next, we'll cover the algorithms devised to tackle our simulation. Finally, we'll go over the code and explain our learnings.

Simulation Setup

As stated earlier, the biggest challenge when analyzing solitaire is the sheer number of possible card arrangements/games. Not only is this a hindrance to approximating the probability of winning, it is also a major obstacle to writing an algorithm that can effectively play the game. In their report, Jupiter Scientific notes this inherent difficulty in writing a program to effectively play solitaire, and resorts to a "human" Monte Carlo method—having their staff play 100 games *with all cards facing up*—to estimate an upper bound probability of winning with "best play" of ~43% (Application of Human Monte Carlo 2013). Please note that Jupiter's variant of solitaire, with all cards facing up, is not a standard variant of solitaire, and can therefore serve as a best-case benchmark, rather than a realistic goal of what we will achieve in our simulation.

Because we want to build a more realistic solitaire simulation, we must first decide on which variant of solitaire we want to simulate. Bicycle, the playing card manufacturer, claims there are 11 commonly-played variants of card-based solitaire:

1) [Classic] Solitaire
2) Klondike (Most-Popular)
3) Spider

4) Accordion
5) Street and Alleys
6) Napoleon at St. Helena
7) Devil's Grip
8) Emperor
9) Wish
10) Nerts
11) Kings Corner

Rules

For ease of implementation, we chose to build our simulation using the first variant. The rules of Classic Solitaire, as described by Bicycle, are as follows:

Game Objective

- The ultimate objective of the game is to build the entire deck of cards, in sequence and in suit, onto each of the four foundations.
- *If* this is accomplished, the game is won.

Card Format

- Classic Solitaire uses one 52-card deck of cards.
- The cards rank from K(high) to A(low).

How to Play

Definitions

There are four "piles" in the game:

1. **Tableau**
    a. 7 piles that make up the main table
    b. Only Kings can occupy Tableau vacancies (empty/no card)
    c. Cards must be played in descending order
    d. Stacks of cards may be moved between Tableau columns
    e. Once all revealed cards in a Tableau column are removed, the bottom card of the unrevealed pile is flipped
2. **Foundation**
    a. 4 piles on which a whole suit must be built up
    b. Each of the 4 aces must serve as the bottom card of each respective foundation
    c. Foundation must be built up in ascending order
3. **Stock**
    a. The remaining cards not laid out in the Tableau are left in the stockpile to be drawn by the player
4. **Waste**
    a. Cards drawn from the stock that are not placed in the Tableau or Foundation are laid face up in waste pile

b. Once all Stock cards are drawn, the Waste pile is then flipped over to serve as the new Stock

Valid Actions (Moves)

In building our simulation, we identified and implemented the following possible moves:

1. Flip and move card from Stock to Waste
2. Move eligible [top] Waste card to Foundation
3. Move eligible card from waste to Tableau
4. Move eligible Tableau card to Foundation
5. Move stacks between Tableau columns
6. Move any King [from Waste or another Tableau column] to any empty Tableau column

These moves form the basis of our gameplay as well as our devised algorithms.

Algorithm Performance and Findings

In researching potential strategies to implement as algorithms in our simulation, it quickly became apparent that there is no simple, surefire way to always win in Solitaire. This is because, depending on the variant of Solitaire, only 80-90% of games are actually winnable (Searching Solitaire in Real Time 2007).

We performed a two-stage analytical approach to find the optimal algorithm:

- First, we evaluated 2 basic algorithms against our human benchmark to verify some hunches we had regarding optimal rule ordering.
- Next, we took a computational approach to quantitatively verify the optimal move and column orderings for our algorithm.

Basic Algorithms

In this first stage, we see that in terms of Win Rate %, our **Bottoms Up** algorithm vastly outperformed our **Recreational Player** algorithm and even our human benchmark. *See Table below.*

Takeaways

- First, this confirmed our suspicion that despite the typical player's eagerness to move cards into Foundation, it's actually more optimal to move cards to Foundation as a "move of last resort", when all other possible moves in a given turn have been exhausted.
- Second, this confirmed our suspicion that move order of execution matters in determining win rate. We further explore this in the next stage of our analysis.

Table of Basic Algorithms

Note: Reported results are based on simulations of 1000 iterations with a maximum cutoff of 300 moves.

Algorithm	Rules	Win Rate %	Avg Moves to Win
Recreational Player *Basic Algorithm*	**Recreational Player** algorithm represents a reasonable synthesis of best practices that would be followed by a human who regularly plays solitaire. This algorithm is by no means *the* optimal way to play solitaire—just a collection of "correct" ways to play. **Each turn:** 1. Flip first card in Stock pile [and add to Waste pile] 2. If possible, move any eligible Tableau cards to Foundation 3. If possible, move eligible [top] Waste card to Foundation 4. If possible, move any Kings to empty Tableau column 5. If possible, add eligible Waste card(s) to Tableau 6. Permute flipped (exposed) piles between Tableau columns to check if new [hidden] cards can be exposed a. Only move piles which will expose new card 7. Next turn	22.2% ± 2.58%	123.33
Bottoms Up *Reverse Algorithm*	**Bottoms Up** algorithm is identical to The Recreational Player with one key difference: *Steps 2 through 6 for The Recreational Player algorithm are executed in reverse order.* Given the fact that moves to Foundation are irreversible in our variant of solitaire, we suspected that moves to foundation early on in The Recreational Player led to higher loss rates.	30.1% ± 2.84%	131.4
The Novice *Human Benchmark*	**The Novice** represents a player with relatively little experience playing solitaire. Our parents' solitaire skills were put to the test. There is no logic. Only ego.	28.6% *Sample size too small for CI*	109.2

Finding Optimal Move Sequence through Permutations

Having verified that even reversing just order-of-execution could result in a huge improvement in algorithm performance, we leveraged permutation to find more optimal execution sequence for our algorithms.

In our simulation, there are two types of sequences through which we could permute:

- **Move sequence**: the order in which we perform possible moves:
  Please note that in this approach, the first move from Basic Algorithms, "Flip first card in Stock pile [and add to Waste pile]", remains fixed as the first move here
  1. If possible, move any eligible Tableau cards to Foundation
  2. If possible, move eligible [top] Waste card to Foundation
  3. If possible, move any Kings to empty Tableau column
  4. If possible, add eligible Waste card(s) to Tableau
  5. Permute flipped (exposed) piles between Tableau columns to check if new [hidden] cards can be expose
- **Column sequence**: the Tableau column order in which we apply each possible moves: Tableau columns 0 through 6 (Python starts on index 0)

Takeaways

Examining the table of top 20 permutations (out of 120 possible permutations) below, we observe a number of clear patterns:

- **Reverse column sequence is by far the most important factor in determining win rate**
  - Every single one of the top 20 permutations follows the same exact column sequence from right to left (tallest to shortest stacks)
- **Given the importance of column sequence, it's no surprise that permuting flipped piles between Tableau columns (Move 5) appears to be the most important first move**
  - That being said, Moves to Foundation may not be as detrimental early on in sequence as suggested by the first stage of our analysis

Top 20 Permutations

Note: Reported results are based on simulations of 500 iterations with a maximum cutoff of 300 moves. CI95 is nearly identical across the top 20 permutations at ±3%.

Move Sequence	Column Sequence	Win Rate %	Avg Moves to Win
51423	6543210	41.80%	129.277512
53421	6543210	39.80%	129.954774
51324	6543210	39.60%	127.025253
51234	6543210	39.40%	126.878173
54213	6543210	39.00%	130.384615
25134	6543210	38.80%	127.515464

35214	6543210	38.80%	126.798969
52413	6543210	38.80%	126.958763
35412	6543210	38.60%	130.440415
53142	6543210	38.40%	129.619792
13524	6543210	38.20%	123.848168
35142	6543210	38.20%	129.031414
54123	6543210	38.20%	130.706806
53124	6543210	38.00%	126.484211
23514	6543210	37.80%	126.846561
35421	6543210	37.40%	131.925134
53412	6543210	37.40%	130.042781
32514	6543210	37.00%	128.281081
54321	6543210	37.00%	130.697297
25143	6543210	36.80%	126.961957

Simulation Code

Our simulation code consists of three files with the following Classes:

Filename	Classes
solitaire.py	1. Strategy 2. Simulation
card_elements.py	1. Card 2. Deck
game_elements.py	1. Tableau 2. Foundation 3. StockWaste 4. Game

Our code can be run in two different ways:

1. Play Solitaire Manually in the Interactive Command Prompt, step by step through keyboard commands
2. To play Solitaire automatically, please uncomment the following block in solitaire.py.

To begin playing, please comment/uncomment the block below in the **__main__** function in solitaire.py

```
# ---Play Solitaire Manually in the Interactive Command Prompt ---------- #
simulation = Simulation('runs_manual.log',verbose=True)
simulation.runManual()
```

Then run the program by typing the following command in Terminal to start the program

```
python3 solitaire.py
```

Once that is typed, you should see the following in Terminal:

```
$ python3 solitaire.py
Valid Commands:
        mv - move card from Stock to Waste
        wf - move card from Waste to Foundation
        wt #T - move card from Waste to Tableau
        tf #T - move card from Tableau to Foundation
        tt #T1 #T2 - move card from one Tableau column to another
        h - help
        q - quit
        *NOTE: Hearts/diamonds are red. Spades/clubs are black.
-----------------------------------------------------------------
Waste    Stock                          Foundation
empty   24 card(s)             C       H       S       D

Tableau
        1       2       3       4       5       6       7

        Aclub   x       x       x       x       x       x
                8spade  x       x       x       x       x
                        Aspade  x       x       x       x
                                10club  x       x       x
                                        7heart  x       x
                                                Aheart  x
                                                        6club

-----------------------------------------------------------------
Enter a command (type 'h' for help):
```

Type any of the commands in Valid Commands to begin playing. When you are done and don't think you have any moves left type q to quit. Alternatively if you win the game, the game will automatically quit. Regardless at the end of the session you should see a summary in command line of your game performance.

```
Enter a command (type 'h' for help): q
Game exited.
Final Score: 0
Num Moves: 0
Game Duration: 111.84834718704224 seconds
```

These metrics are also outputted to the file you specified to the Simulation object. In this example it would be run_manual.log

3. To play Solitaire automatically, please uncomment the following block in solitaire.py.

To run the program automatically, you will need to specify some parameters.

```
# ------- Run Automatic Solitaire Simulations based on Parameters ----------- #
## Specify Rule Order
rule_order=list(range(1,6))
## Specify Column Order
col_order=list(range(7))
## Specify Number of Simulation Iterations
num_runs = 1000
## Specify Number of Max Turns per Round (To prevent infinite looping)
max_turns = 300

# Create a Simulation Object with Run Parameters
simulation =
Simulation('logs/runs_auto_basic_{num_run}_{max_turns}_{rule_order}_{col_order}.
log',num_runs=num_runs,max_turns=max_turns)

# Create a Strategy Object that specifies Rule Order and Column Order
strategy =
Strategy(rule_order=rule_order,col_order=col_order,verbose=simulation.verbose)

# Run the Simulation
simulation.runAuto(strategy)
```

To run the program, again go to terminal and run the following command:

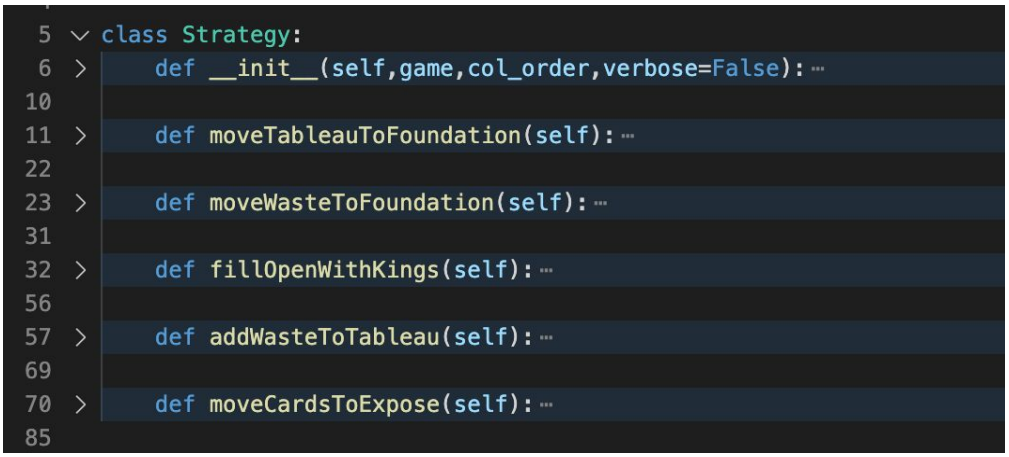
```
python3 solitaire.py
```

Once it’s done running, the run metrics will be outputted into the .log file specified above.

Classes

solitaire.py

- The **Strategy** class is used to store and order possible actions based on the rule_order and column order specified to the constructor of the class
    - **orderedRuleDict()** is used to order the rules based on the class rule_order
    - There are also multiple possible functions for the possible actions. One example is **fillOpenWithKings()** which is used to fill blank Tableau spaces with available Kings based on the column order for the class
        - The order in which these possible actions are applied is specified by the **orderedRuleDict()** function above
- The **Simulation** class is used to run simulations. It takes in an output file to output the results. For automatic runs, it also takes in number of runs to determine how many times to run the simulation and max_turns to determine when to end the game
    - **simulateRulePerm**(strategy) is used to enact game actions in the order specified by rule_order and col_order strategy object provided
    - **runManual()** allows us to test our program with human players through an interactive command line prompt where they can input command line commands such as ‘wf’ to move a card from Waste to Foundation or ‘tf2’ to move a card from the 2nd Tableau to Foundation
    - **runAuto()** is the function that allows us to run multiple iterations of a game automatically as specified by the num_runs parameter. In each iteration a new Game is created and the specified strategy is applied. When the game is done running, the outputted to the class output log file



```
86  class Simulation:
87      def __init__(self,output_log,alg='manual',num_runs=100,max_turns=100,verbose=False):
104
105     def simulateBasic(self):
133
134     def basicAuto(self):
150
151     def runManual(self):
178
179     def outputToLog(self):
185
186     def runAutoBasic(self):
192
```

card_elements.py

- **Card** and **Deck** classes were used to initialize our starting deck and distribute cards

```
3   class Card():
4       card_to_name = {1:"A", 2:"2", 3:"3", 4:"4", 5:"5", 6:"6", 7:"7",
6
7       def __init__(self, value, suit):
13
14      def isBelow(self, card):
16
17      def isOppositeSuit(self, card):
22
23      def canAttach(self, card):
28
29      def flip(self):
31
32      def __str__(self):
34
```

```
36  class Deck():
37      unshuffled_deck = [Card(card, suit) for card in range(1, 14) for suit in ["club", "diam", "heart", "spade"]]
38
39      def __init__(self, num_decks=1,verbose=False):
42
43      def flip_card(self):
45
46      def deal_cards(self, num_cards):
48
49      def __str__(self):
```

game_elements.py

- **Tableau** class manages the 7 starting piles of cards and their interactions with the Stock/Waste and the Foundation.

```
 8  class Tableau():
 9      # Class that keeps track of the seven piles of cards on the Tableau
10
11      def __init__(self, card_list, verbose=False):…
15
16      def flip_card(self, col):…
20
21      def pile_length(self):…
24
25      def addCards(self, cards, column):…
37
38      def tableau_to_tableau(self, c1, c2):…
50
51      def tableau_to_foundation(self, foundation, column):…
64
65      def waste_to_tableau(self, waste_pile, column):…
74
```

- **Foundation** class handles the placement of cards into the Foundation piles.

```
118 class Foundation():
119
120     def __init__(self,verbose=False):…
123
124     def addCard(self, card):…
143
144     def getTopCard(self, suit):…
152
153     def gameWon(self):…
```

- **StockWaste** class moves cards in between the Stock pile and Waste pile, and provides getters/setters for Stock and Waste piles.

```
 75 class StockWaste():
 76     """ A StockWaste object keeps track of the Stock and Waste piles """
 77
 78     def __init__(self, cards,verbose=False):…
 82
 83     def stock_to_waste(self):…
 98
 99     def pop_waste_card(self):…
103
104     def getWaste(self):…
110
111     def getStock(self):…
117
```

- **Game** class initializes one single game by creating the Tableau, Foundation, StockWaste Piles.
    - **gameWon()** In addition, it includes helper functions such as gameWon() to discover when a game is won (aka all 4 foundation classes are filled from 1 to 13 for each suit).
    - **getFinalMetrics()** It also keeps track of metrics includes a scoring function getFinalMetrics() for number of moves, game duration, score using Microsoft Windows Solitaire Scoring:

Move	Points
Waste to Tableau	5
Waste to Foundation	10
Tableau to Foundation	10
Turn over Tableau card	5

    - **printTable()** prints the current state of the game and converting inputted commands to python instructions. These are hidden using the verbose keyword for automated runs of the game
    - **printValidCommands()** helper functions for manual runs of the game such as outputting instructions

```
163 class Game():
164     def __init__(self,verbose=False):…
176
177     def getFinalMetrics(self):…
182
183     def gameWon(self):…
185
186     def printValidCommands(self):…
197
198     def printTable(self, tableau=None, foundation=None, stock_waste=None):…
221
222     def takeTurn(self, command):…
320
```

References

1. Diaconis, Persi. "Mathematics of Solitaire". Mathematics Department and Graduate School Colloquium Archive 1998-1999. Retrieved 22 December 2011.
2. "The Application of Human Monte Carlo to the Chances of Winning Klondike Solitaire". Jupiter Scientific. 2013.
3. "Klondike" (p.195) in Hoyle's Rules of Games (3rd edition) by Philip D. Morehead (ed.), 2001. ISBN 0-451-20484-0

4. "Searching Solitaire in Real Time" (PDF). ICGA Journal. September 2007. Retrieved 31 July 2020.