2018

Forty Thieves AutoStart

AN ALGORITHM ANALYSIS

TOM HEYSEL – 15TJH2 - 200000838

1.0 Introduction

The purpose of this assignment is to write a program to simulate the AutoStart Function of a Forty Thieves Solitaire game. The program must deal a completely random deck each time, distribute the cards following the proper Forty-Thieves distribution (see Appendix A, Rules of Forty Thieves Solitaire), and move the most possible cards into their final positions. This repeats until a minimum of 15 cards have been placed in their sorted piles and tracks the movements of each card, producing a histogram of their distribution upon completion. See Figure 1 below, which defines the terminology used throughout this report.

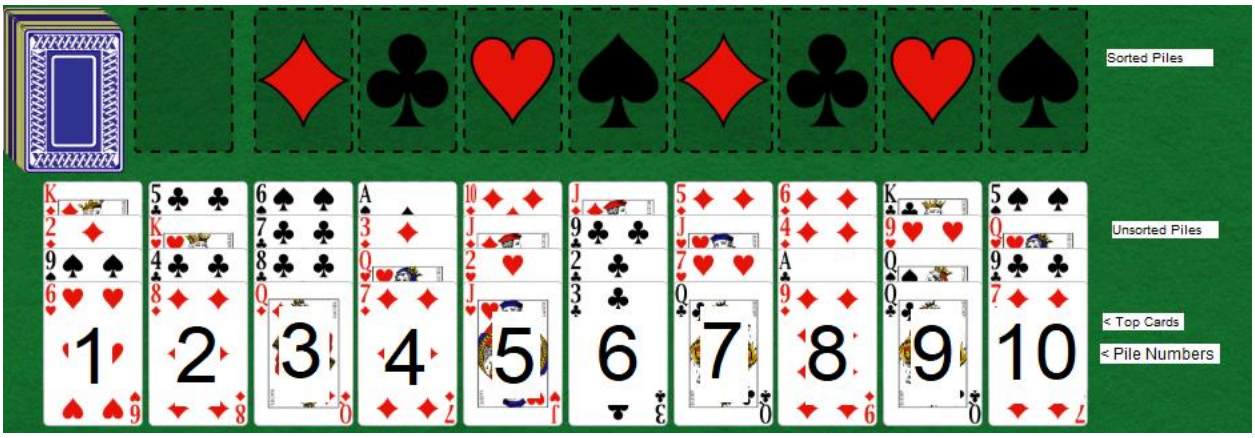

Figure 1 Table Definitions

2.0 Potential Algorithms and Decision Making

2.1 Greedy

The first potential algorithm I explored was greedy. The approach would traverse the unsorted piles, analyzing their top card on each pass, and if a card can be moved to a sorted pile, it is. This repeats until a pass over all the unsorted piles results in no cards being moved. The “analyzing” if a card can be moved would be as follows: first read its suit, then compare its value to the those of its corresponding suits’ sorted piles. If its value is one greater than what is currently there, or if it is an ace and the sorted pile is empty, it can be placed. If these conditions are not met, it cannot.

This algorithm is how most AutoStart functions operate, as it allows the cards to be moved around the board without any bias towards helping the player. However, in many instances, it fails to move the greatest number of cards possible. See Figure 1 below, which depicts a partial view of the table. Here a greedy algorithm would choose to move the leftmost 2 of Spades, instead of the one on the right, producing a sub-optimal solution.

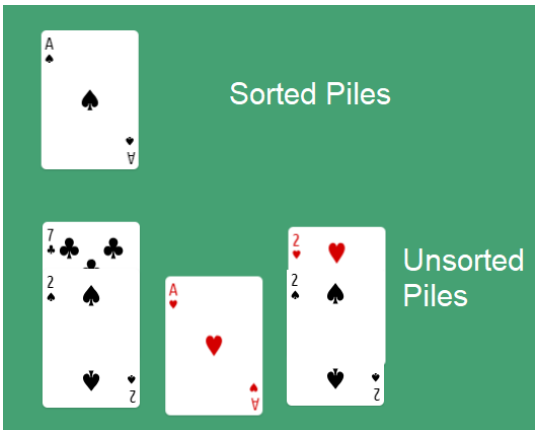

Figure 2 Greedy Algorithm Falls Short

In this example, the greedy algorithm clearly falls short to organize cards while moving as many as possible into their final position. As this algorithm cannot produce an optimized solution, it isn't really a greedy algorithm and is not a viable option.

2.2 Branch and Bound

Objective Function:

Maximize the total number of cards moved to their correct final position, or inversely, minimize the total number of cards left in their original position.

Bound Definitions:

Global Upper Bound (GUB):

The GUB is the total number of cards left in their original position for best possible solution that we know of. The initial GUB is 40, 10 piles of 4 cards. Unfortunately, due to the nature of the problem there is no opportunity to initially prune possible solutions to reduce this global upper bound, as all cards have the potential to be moved at the start of the game. As cards begin to move into their final position, this

variable will keep track of the maximum number of cards that can be moved from the current best solution.

Upper Bound (UB) on a Partial Solution:

For any given top card, the UB is the total number of cards remaining so far, assuming we don't take any more cards other than the card in question (if possible).

Ex 1: Pile 10's top card is the Ace of Clubs and there are 37 cards remaining. The UB for the Ace of Clubs is 36. This is derived from 37 – 1 (the Ace of Clubs can be moved).

Ex 2: Pile 6's top card is the 9 of Diamonds and there are 35 cards remaining. The UB for the 9 of Diamonds is 35. This is derived from 35 – 0 (assuming the 9 of Diamonds can't be moved).

Lower Bound (LB) on Partial Solution:

For any given top card, the LB is the total number of cards remaining so far, minus the number of cards that can be subsequently moved from the pile in question.

Ex 1: Pile 3 in top down order contains Ace of Spades, Ace of Hearts, Jack of Hearts, 4 of Diamonds with a total of 32 cards remaining. The LB for the Ace of Spades is 30. This is derived from: 32 – 1 (Ace of Hearts can be moved) – 1 (if the Ace of Spades is moved then so can the Ace of Hearts) – 0 (assuming the Jack of Hearts cannot be played).

Ex 2: Pile 9 in top down order contains King of Hearts, Ace of Clubs, Two of Clubs, Three of Clubs with a total of 35 cards remaining. The LB for the King of Hearts is 35. This is derived from: 35 – 0 (assuming the King of Hearts cannot be played).

Expansion of Tree and Pruning

Pruning

The nature of this being a game-algorithm provides a very straight forward method in deciding whether or not to prune a branch. If it is against rules for a card to be moved, don't explore the branch where the card is moved! Branches can also be pruned based on their LB. If a given branch's LB is greater than the GUB, then the branch can be pruned as it will never lead to an optimized solution.

Expansion

Depth-First Expansion assesses the validity of a potential branch using the techniques described in the section Pruning, above. If it is a branch that is worth being examined further, it is stored into a LIFO

queue for future expansion. Similarly, *Breadth-First Expansion* stores nodes in a FIFO queue and *Best-First Expansion* stores nodes in a minheap, sorted by their LB.

Where Branch and Bound Falls Short

Consider the following.

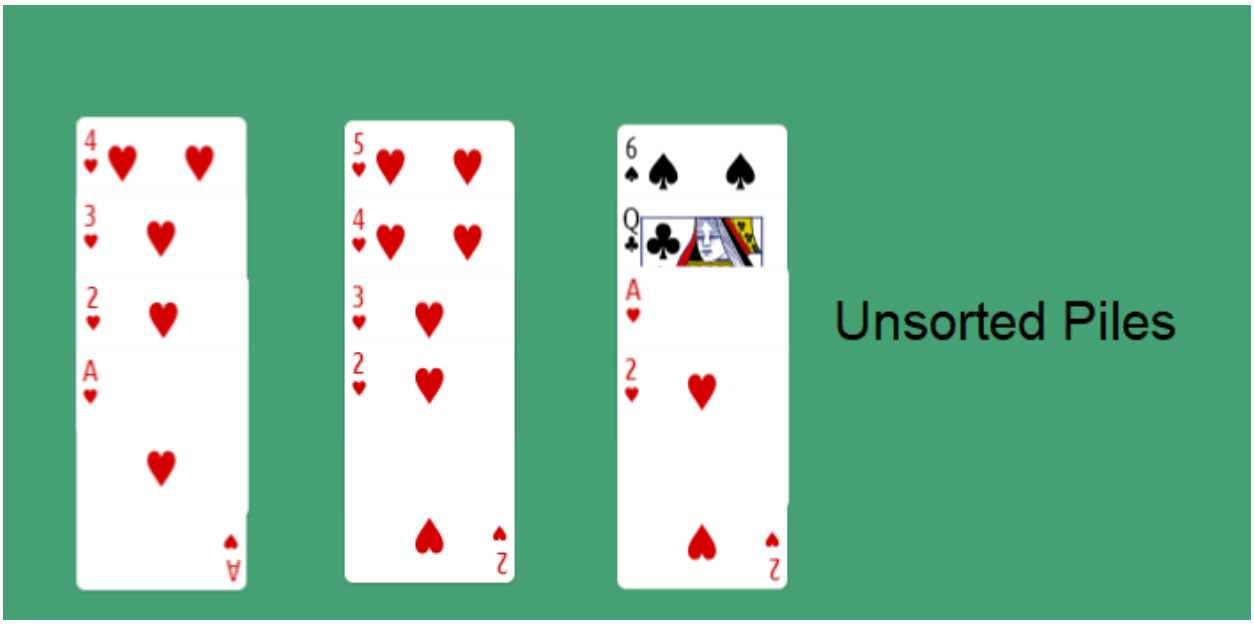

Figure 3 Arrangement disproving the Branch and Bound Algorithm

Using the branch and bound algorithm we can prove that the program will select a non-optimal solution. First, the program must move the Ace, it has no other option, this adjusts the GUB to 39. After this move it must evaluate the bounds of the three Two's it has available to it. All three will have a UB of 38, assuming no cards other than the Ace of Hearts has been moved to its final position. The leftmost has a LB of 36, the middle 35 and the right 38. Using best-first transversal, the program will dive into computing the optimal solutions of the middle and left piles due to their low LB values. Once it has fully explored their potential solutions, the GUB will be 35 (from moving the Ace and all middle pile cards) and the minheap-storing potential branches will have the rightmost Two as its root. Since this Two has a LB *greater than* the GUB it will be pruned.

But it is clear that the *optimal* solution moves first the Ace, then the right most Two, then all the cards of the middle pile, moving a total 7 cards, as opposed to the branch and bound moving only 5.

2.3 NP-Complete

Given the issues with both the greedy and branch and bound algorithms, I was unsure of how to approach finding the solution to this problem, when I started to wonder if there even *was* a solution. After some research, I found a 2015 paper by Chuzo Iwamoto and Yuta Matsui, published in The Institute of Electronics, Information and Communication Engineers Journal, proving that the generalized version of the Forty-Thieves game is NP-Complete [1]. The authors prove this by 1) showing that the problem is in NP, and, 2) by reducing the known NP-Complete 3SAT problem to the decision problem: can all the cards be moved to their final positions following the rules of Forty-Thieves? For the purposes of the AutoStart function, the decision problem would be changed from "all the cards" to "the initial 40 cards". As the authors proved NP-Completeness for the *generalized* Forty-Thieves Game, we can make the adjustment without loss of generality. Now that it is known that the problem is NP-Complete, we can explore potential Heuristics and Approximation Algorithms.

Approximation Algorithms

As we are just starting the chapter on Approximation Algorithms in class now, I am choosing to leave their potential solutions behind, due to my lack of understanding on how to generate *provable guarantees* regarding how significantly my solution deviates from the optimal one. I instead chose to explore Heuristics.

Heuristics: Final Decision

"A heuristic is a technique designed for solving a problem more quickly when classic methods are too slow, or for finding an approximate solution when classic methods fail to find any exact solution. This is achieved by trading optimality, completeness, accuracy, or precision for speed [2]." In the contexts of our decision problem, we are trading the optimality of a solution for speed. As we have shown above, neither the greedy nor the branch and bound "algorithms" produce an optimal solution, so they cannot truly be identified as algorithms, but instead offer potential heuristic solutions. I selected to use a variation of the original greedy algorithm to optimize the time and space complexity of the solution. The modification follows the original greedy structure that if it can be moved to it's final position it is, but changes slightly in the case where it is not possible to move a card to its final position. The additional logic checks if the card can be moved to an intermediate position (see Appendix A for definitions of final and intermediate positions). This choice was selected over a branch and bound heuristic as it requires no recursion. By removing recursion, we reduce the time and memory required to find a solution, as well as provide more straight-forward logic for those reading the code.

3.0 Time Complexity

The Heuristic used has a complexity based only on its outputs. We will examine the following pseudo-code to prove this.

```
psuedoMain(){
   do{
      flag = 0;

      for each unsorted pile{
         if(checkIfCardCanBeMovedToFinalPosition() == true){
            moveCardToFinalPosition();
            flag = 1;
         }else if(checkIfCardCanBeMovedToInterPosition() == true &&
`                                cardHasntAlreadyBeenMovedToAnInterPosition){
            moveCardToInterPosition();
            flag = 1;
         }
      }
   }while(flag == 1);
}
```

3.1 Inner Loop

The inner for-each loop of the algorithm will scan through the ten unsorted piles, checking if its top card can be moved to its final or an intermediate position. The loop itself will take ten iterations, O(10), and the checking function that determines whether a card can be moved to its final position requires two O(1) comparisons (to compare with the two corresponding suit's sorted piles to see if the card can be moved), all of which reduces to O(1).

If the top card can be moved to its final position, it is a simple pop and push taking O(1) time. A flag is then set, indicating that the sorted piles have changed, and another outer loop is required.

If the card *cannot* be moved to its final position, the program calls a function to determine if it can be moved to any intermediate position, so as to open up more cards to the potential of being moved. Checking if a card can be moved to any intermediate position simply loops over the unsorted piles comparing the top cards and is therefore O(1) time. The program determines if a card has already been moved to an intermediate position by reading a flag in the card's struct definition, which also takes O(1) time. If there is an intermediate position that the card can be moved to, and its intermediate flag has not been set, then the card is moved. This requires another pop and push making the move O(1). It is

vital to have this intermediate position flag in the card struct definition to prevent the infinite loop shown in Figure 4, below.

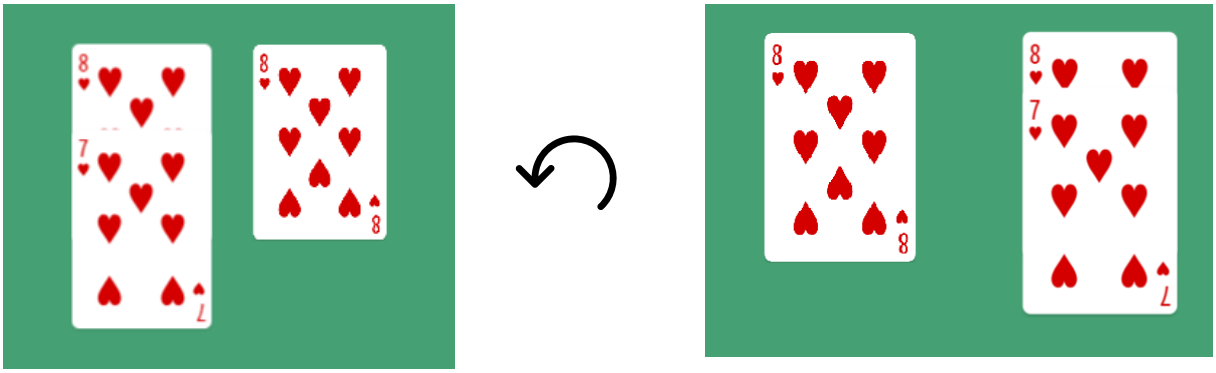

Figure 4 Infinite loop in intermediate positions

3.2 Outer Loop

The outer do-while loop runs until it the program has done a complete pass of all 10 unsorted piles where no cards are moved. Since each card can only be moved to an intermediate position at most once, an intermediate position move can cause a maximum of 39 loop iterations (one for every card excluding the very last on the table). Even in this worst case scenario, the big O is still reduced to a linear form, O(1). The final and dominating time complexity results from a card being moved to a sorted pile. In the worst case, each of these moves will require its own loop iteration, and therefore the final time complexity will be O(h) where h denotes the number of cards moved to their final positions.

3.3 Time Complexity Conclusion

In conclusion, the heuristic has been designed using time-efficient data structures such that the body of the function, which checks if a card can be moved to a final or intermediate position, has a combined upper bound of O(1) time. The inner loop, which computes the described main function for each of the 10 unsorted piles, has a joint complexity of O(1)*O(10) = O(1). Finally, the outer loop in its worst case will only be able to move one card per iteration to its final position. Let the number of cards in the sorted piles be h, and the heuristic has a total upper bound time complexity of O(1)*O(1)*O(h) = O(h) (functionality * inner loop * outer loop).

4.0 Performance and Analysis

The movement of cards is tracked over the course of the AutoStart function to obtain a detailed view of what is happening at the base level. With a randomly shuffled deck each time, the AutoStart function runs in a do-while loop until the program successfully moves a minimum of 15 cards to their final positions. Below is a histogram depicting the number of cards moved to their final position, averaged over 50 trials.

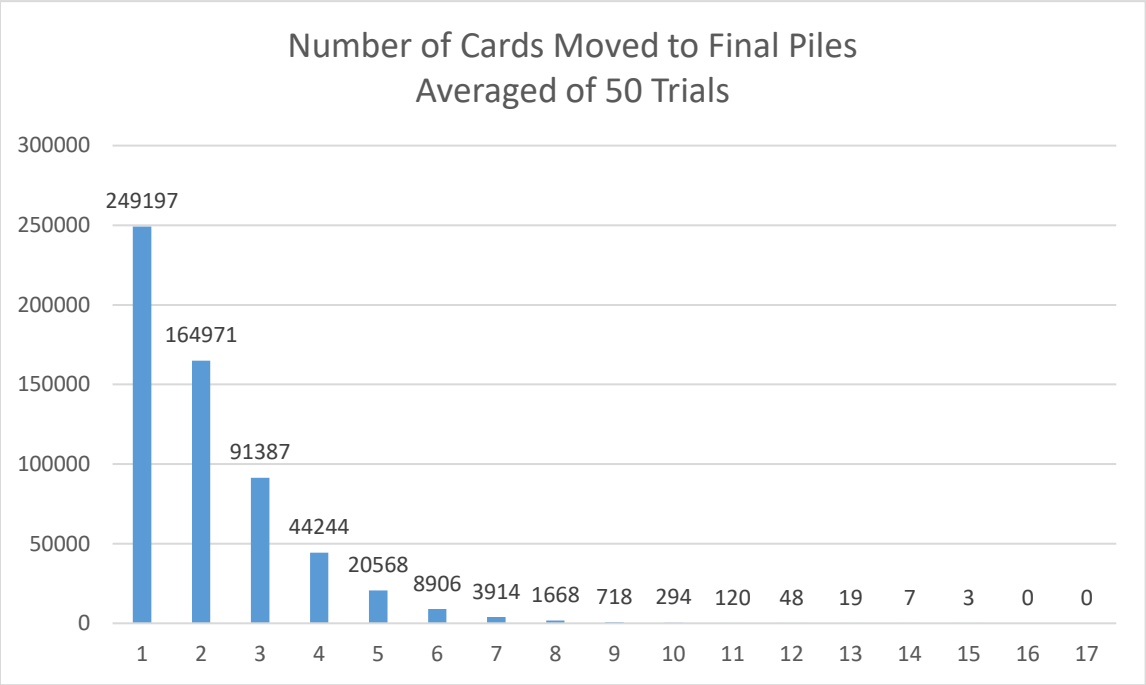

Figure 5 Histogram depicting # of cards moved per trial

It makes intuitive sense that, as the number of moves per iteration increases, the number of trials that produce that number of moves decreases. The more cards required to move, the less likely that arrangement will be dealt. A point of interest in the graph is the fact that 3 trials reach a 15-card distribution, when the program should stop after 15 or more moves to the final position have been achieved. It would seem as though there was an error in the algorithm, but we must remember that this is the *average* over 50 trials. In many cases, more than 15 cards were achieved, with the greatest number of cards moved to their final position being 22- more than half the original table!

To attempt to visualize the runtime, I tracked the number of re-deals, the total number of cards moved to their final position, and the time (in milliseconds) required to complete each AutoStart. Following the

nature of heuristics, we observe a wide range in possible times. See Figure 6 and 7 below for a distribution of the time and number of re-deals. In the best case it took only 124ms to generate a solution, and in the worst case 80229ms. On average, the heuristic takes 32909ms (roughly 33 seconds) .

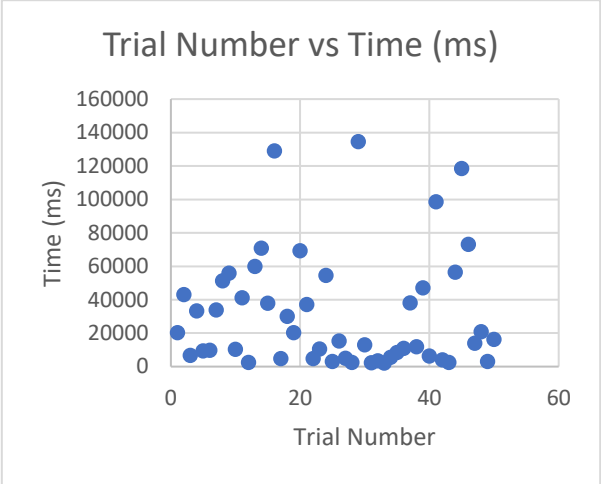

Figure 6 Trial Number vs Time

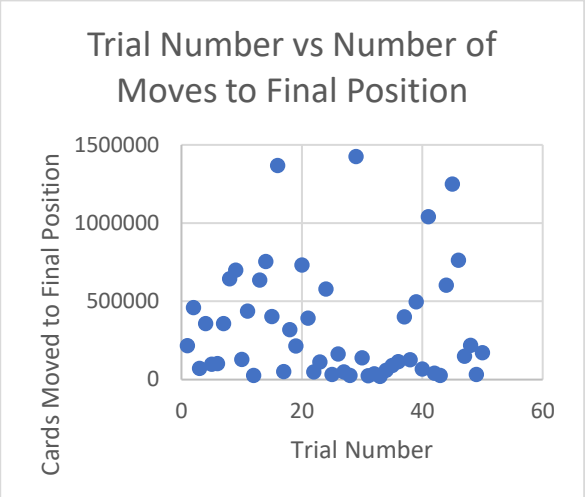

Figure 7 Trial Number vs Number of Moves to Final Position

Between the above graphs we can clearly see that the number of moves to a final position is tightly coupled to the runtime of the heuristic. By finding a ratio of the two variables for each trial we can see that the runtime is approximately 0.1 times smaller than that of the number of cards moved. This is shown in Figure 8.

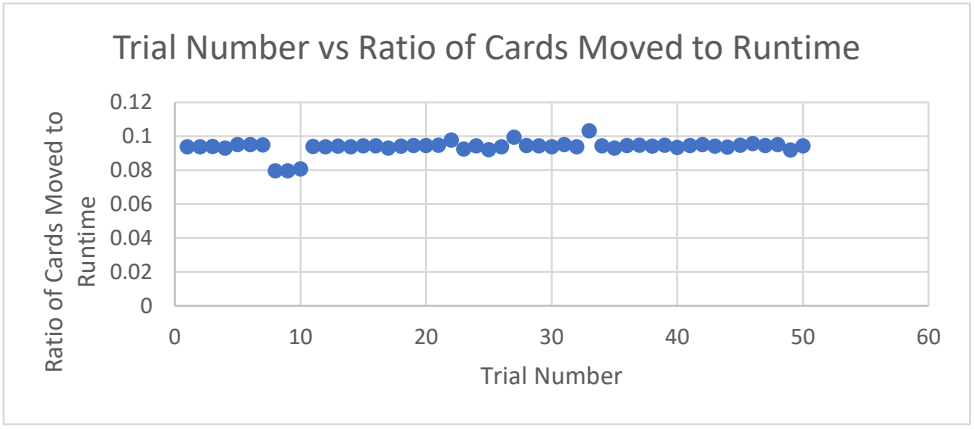

Figure 8 Ratio vs Trial Number

This constant ratio implies that a Cards Moved vs Runtime graph would display a linear graph with a slope of approximately 0.1. And that is exactly what is observed in Figure 9, below.

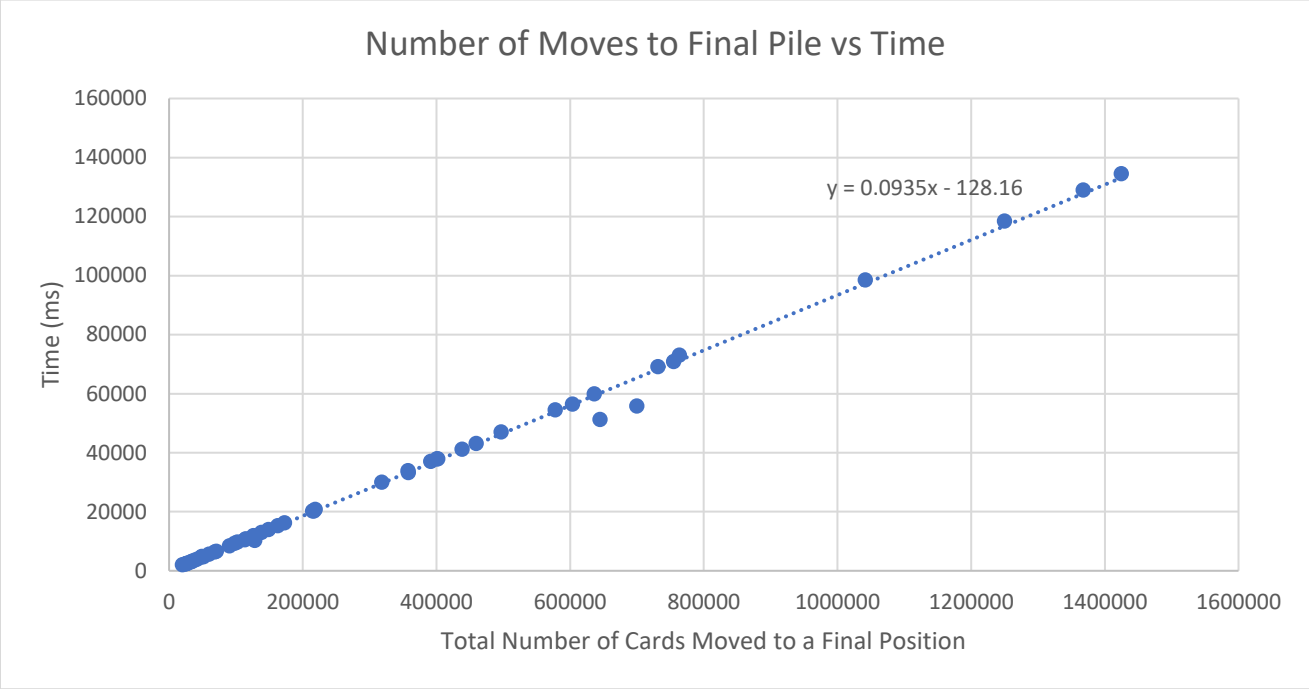

Figure 9 Graphical Proof of O(h) runtime

Analyzing the data in Figure 9, above, it can be concluded that, as predicted in 3.0 Time Complexity, the runtime is linearly proportional to the number of cards moved to their final position. A final detail strengthening the case to use a heuristic is the observation that the data on this graph is both linearly and is distributed unevenly towards lower runtimes.

Appendix A Rules of Forty Thieves Solitaire

Rules

Deck

Forty Thieves is played with two standard 52-card decks, for a total of 104 cards. Ace is low, King is high.

Goal

The goal of Forty Thieves is to build eight sorted piles up from Ace to King, keeping the suit.

Setup

Deal 40 cards to the unsorted piles in 10 columns of four cards each. The cards in each column should overlap so that all 40 cards are visible.

Moving Cards

Only one card at a time may be moved.

In the unsorted piles, only the top card of each column is available to be moved. Cards in the unsorted piles may be moved either to a sorted pile or to another column in the unsorted piles.

In the unsorted piles, a card can only be added to a column if it is one rank lower and the same suit as the card it is being played on. EXAMPLE: The 10 of Hearts can only be played on the Jack of Hearts.

A sorted pile must be started with an Ace. A card can only be added to a sorted pile if it is one rank higher and the same suit as the card it is being played on. EXAMPLE: The 4 of Spades can only be played on the 3 of Spades.

The top card of the stock can be drawn at any time and played to a foundation, played to a column in the tableau, or added face up to the discard pile.

The top card of the discard pile can be played to a foundation or to a column in the tableau at any time.

If there is an empty column in the unsorted piles, any card which can be legally moved may be played to that column.

Final Position: Position in which a card is in a correct final pile.

Intermediate Position: Position in which a card has been move from one unsorted pile to another. This move can only occur once per card.

Winning

A player wins Forty Thieves if all eight foundations are completely built, from Ace to King.

Appendix B References

[1] C. IWAMOTO and Y. MATSUI, "Computational Complexity of Generalized Forty Thieves", *IEICE Transactions on Information and Systems*, vol. 98, no. 2, pp. 429-432, 2015.

[2] “Heuristic (computer science),” Wikipedia, 19-Nov-2018. [Online]. Available: https://en.wikipedia.org/wiki/Heuristic_(computer_science). [Accessed: 14-Nov-2018].