A Heuristic Approach to the Game of Sylver Coinage

Gilad Moskowitz[1] and Vadim Ponomarenko[2]

[1] Mathematics Department
San Diego State University
San Diego, CA 92182
gilad.moskowitz@gmail.com
[2] Mathematics Department
San Diego State University
San Diego, CA 92182
vadim123@gmail.com

Abstract. Sylver Coinage is a zero-sum terminating game, making the search for an optimal strategy very enticing. Many of the challenges that existed with creating computer programs to play games like Chess and Go exist for Sylver Coinage as well. However, unlike Chess and Go, working towards finding an optimal strategy in the game of Sylver Coinage presents some new and interesting challenges. We attempt to make some headway on the problems associated with finding a strategy for Sylver Coinage using several heuristic algorithms employed by bots to play the game.

1 Introduction

John H. Conway discovered the game of Sylver Coinage, and popularized it in the 1982 book, *Winning Ways, for your mathematical plays* [1]. In this game, two players face off taking turns naming positive integers that can not be created as a sum combination of perviously named integers. The first player forced to name 1, loses. An example game is given in Table 1, below.

Since its inception, Sylver Coinage has been studied extensively in both published [3–6, 9] and unpublished [2, 7, 8] works. A lot is known about winning and losing positions, but not much is known about how to actually win from those positions. For instance, Hutchings's Theorem [1] tells us that any prime number greater than or equal to five is a winning first move. However, there is no known strategy for actually finding a winning move after playing such a prime number. Another problem that comes up in the study of Sylver Coinage is that of infinite positions.

Definition 1. *An* infinite position *in the game of Sylver Coinage is one in which there are still infinitely many remaining legal moves.*

The first move played by each player is played in an infinite position, but we can see that sometimes a game can remain in an infinite position for a long

Table 1. Example Sylver Coinage Game.

Player	**Move Played**	**Remaining Possible Moves**	**Explanation**
1	3	1, 2, 4, 5, 7, 8...	Any move that is not a multiple of 3
2	5	1, 2, 4, 7	8 = 3 + 5, 9 = 3 + 3 + 3, and 10 = 5 +5, so all moves above 7 can be made as 8 + 3k, 9 + 3k, or 10 + 3k for $k \geq 0$.
1	7	1, 2, 4	Remaining moves after playing 7.
2	4	1, 2	Remaining moves after playing 4.
1	2	1	Remaining moves after playing 2.
2	1	—	Player 2 is forced to play 1 and loses.

time. For instance, if Player 1 plays $2^{2^{100}}$, then Player 2 can play $2^{2^{100}-1}$, then Player 1 can play $2^{2^{100}-2}$ keeping the game infinite since no odd number has been played, and this can go on for a long time. There can also be infinitely many possible combinations that keep a game in an infinite position. However, a game cannot remain in an infinite position forever [1]. All Sylver Coinage games will eventually end.

Comparing Sylver Coinage to games like Chess and Go, we see three distinct problems with the analysis of the game. They are: the size of a position, the lack of an established corpus of human strategy, and the lack of a natural way to naively evaluate a position.

In a game of Sylver Coinage, we have the existence of infinite positions, and we already discussed how this adds difficulty to the problem. Also, many finite positions have a much larger move set than any position in a game of Chess or Go. This fact makes is so that computationally calculating a winning move in a given position is almost impossible, and even a look-ahead strategy would be too computationally intensive.

Little has been discovered that concerns the strategies used to win a game of Sylver Coinage. Some positions have been analysed completely and full winning strategies are known, similar to how there are rules to the endgame in Chess, but there are still many finite positions that have no known winning strategy. The only "opening" that is proven to be a winning strategy is a prime number greater than 5. However, there is no known strategy to winning after playing a prime number greater than 5 as an opening move. There is a complete lack of literature on attempted strategies and human evaluation of general positions. When attempting to write a bot to play a game like Chess or Go, there are many resources on strategy that can be used to improve the bot. For Sylver Coinage, no such resources exist.

Lastly, a major problem of Sylver Coinage is the lack of a naive way to evaluate a position. In Chess, for instance, pieces have value. One may evaluate positions based on material gains of each player. For instance, in most situations, a position in which a player doesn't have a queen on the board is much worse than the same position with the player still in control of their queen. In Sylver Coinage, there is no inherent way to evaluate positions and to make material gains as it were. Therefore, making a bot proves even more difficult as there is no easy implementation of a strategy that prioritizes material gains, a strategy often used historically by primitive chess bots.

With modern computational power, we hope to find new insights on the game using various algorithms that play against each other. Hence, our goals include producing a bot that plays well, and making inroads on the second and third problems. Instead of human analysts writing books on what they believe is a good position, we will have objectively skilled bots that can give their opinions. Further, the algorithms of our good bots make progress on the third problem, by giving us a way to evaluate a position.

2 Bot Strategies for Playing Sylver Coinage

In pursuit of an optimal strategy for the game of Sylver Coinage, we began the development of several bots to play the game against each other. We will discuss the development of the heuristic strategy used by the current most successful bot, and show results from testing this bot against various prior versions.

All the bots discussed only begin to implement their strategy in a finite position. While the position is still infinite they rely on the same heuristics and some random choices to play their moves. For this reason, we will only focus on the strategy in a finite position. We begin by describing the first set of "naive" bots with minimal strategy. These bots were

- randomBot - Always picks a random legal move
- alwaysMin - Always picks the smallest legal move greater than 3 (since picking 1, 2, or 3 in any position will lead to a loss if the opponent plays correctly)
- alwaysMax - Always picks the largest legal move
- maximalOdd - The most complicated of the initial batch. This bot would pick the largest legal move that would return an *odd position.* If no move returns an odd position, it will just pick the largest move.

Definition 2. *We say the* parity *of a position is even when the number of remaining legal moves is even, and odd when the number of remaining legal moves is odd.*

The motivation for the maximalOdd bot came from the realization that if you can always return a position with odd parity to your opponent, you will eventually return a position where the only remaining legal moves are 1, 2, and 3 thereby winning. When we ran a round robin tournament with these four aforementioned bots, we got the results found in Table 2.
We see that the maximalOdd bot outperforms its competition. Comparing its success rate versus each individual bot over 1000 games, we found that against alwaysMax, the maximalOdd bot won all 1000 games; against alwaysMin, it won 876 games; and against randomBot, it won 760 games.

The performance of the maximalOdd bot led us to look for advancements that can be made on this strategy. We came up with the following definitions, leading to our more advanced maxThen1Weak bot.

Table 2. Round Robin Tournament with three rounds, where each match between two bots consists of 50 games. Each match gets scored as follows, 3 points for a win, 1 point for a tie, and 0 points for a lose.

Bot Name	**Match Score (27 maximum)**	**Total Wins**	**Win Percentage**
maximalOdd	27	397	88.22%
alwaysMax	18	218	48.44%
randomBot	3	167	37.11%
alwaysMin	6	118	26.22%

Definition 3. *We say that a position is* weak*, or* 0-weak*, if the parity of the position is odd.*

A position in which the only remaining move is 1 is a lost position. Similarly, a position with only 1, 2, and 3 as legal moves is lost. So we see that in many cases a position of odd parity can lead to a loss. This leads us to our next concept.

Definition 4. *We say that a position is* 1-weak *if for every remaining move, $j > 1$, there is a different unique remaining move, $k > 1$, such that playing k in response to j results in a weak position. Note that a 1-weak position is also a weak position as every move other than 1 has a pair that can be played to return a weak position.*

The idea here is that if Player 1 is in a 1-weak position and for every move there is a response that returns a weak position, then Player 1 will end up playing into a weak position again no matter their move. Therefore, they are unlikely to win if the opponent plays perfectly.

Working with this definition, we built a bot, maxThen1Weak, that checks all the remaining legal moves and sees if playing any of them will result in the opponent ending up in a 1-weak position. However, checking every move and then checking if a position is 1-weak is very costly in terms of computation.

Giving the bots any amount of time to calculate and play their moves could result in extremely long games. Suppose there are 10000 remaining legal moves. Testing all the moves to see if playing any of them would result in a 1-weak position is close in computational time to 10000^3 computations of position. Each computation of the remaining legal moves in a position is also costly. Thus, to prevent games from lasting too long, we implemented a time constraint on the bot to play their move. We chose to have each bot play with a 30 second chess clock. We wanted to prevent extremely long games while still giving bots time to make calculations on critical moves.

To make sure that our maxThen1Weak bot doesn't take too long to play a move, it only starts to implement its strategy when there are less than thirty remaining legal moves. Until then it will play the maximal legal move remaining. Even with this restriction on the implementation of the strategy, the bot still vastly outperforms the competition thus far. In a round robin tournament against

Table 3. Round Robin Tournament with three rounds, where each match between two bots consists of 50 games. Each match gets scored as follows, 3 points for a win, 1 point for a tie, and 0 points for a lose.

Bot Name	**Match Score (36 maximum)**	**Total Wins**	**Win Percentage**
maxThen1Weak	36	585	97.5%
alwaysMax	18	213	35.5%
alwaysMin	9	135	22.5%
maximalOdd	27	418	69.67%
randomBot	0	149	24.83%

all the previously mentioned bots, the maxThen1Weak bot performed as seen in Table 3.

We can see that the maxThen1Weak bot was much ahead of its competition, only losing 15 out of the 600 games played. In a head to head match of 1000 games against the maximalOdd bot, the maxThen1Weak bot was able to win 934 games.

However, as it turns out, there are many positions that are 1-weak, but not lost position. To address this we arrive at our next definition.

Definition 5. *We say that a position is* 2-weak *if for each remaining move, $j > 1$, there is a different unique remaining move, $k > 1$, such that playing k in response to j returns a 1-weak position.*

Remark 1. When checking if a position is 2-weak, we are seeing what the position will be after four moves have been played. In a given finite position, we can try to calculate every possible remaining position to find a complete strategy for winning, but this is very computationally expensive. With the strategy of 2-weak we only need to look a few moves ahead to give us a decent sense of the position.

So, using this new definition, we created the maxThen2Weak bot. This bot, similarly to the maxThen1Weak bot, only uses its strategy when there are less than thirty remaining legal moves. With this modification, we saw significant improvement. Adding maxThen2Weak to the round robin tournament we ran above and running the tournament again, we got the results found in Table 4.

We see that the maxThen2Weak bot does have a slight edge over the maxThen1Weak bot, but both are much stronger than the rest of the bots. In a head to head match of 1000 games against the maxThen1Weak bot, the maxThen2Weak bot was able to win 664 games.

Using the notion of a 2-weak position we devised an improvement for the maxThen2Weak bot. The pseudo-code for this bot, dubbed peekThen2Weak, is below.

```
def pretendMove(move, remainingMoves):
    return [i if(i in remainingMoves after move is played)]
```

Table 4. Round Robin Tournament with three rounds, where each match between two bots consists of 50 games. Each match gets scored as follows, 3 points for a win, 1 point for a tie, and 0 points for a lose.

Bot Name	**Match Score (45 maximum)**	**Total Wins**	**Win Percentage**
maxThen2Weak	45	674	89.87%
maxThen1Weak	36	650	86.67%
maximalOdd	27	423	56.4%
alwaysMax	15	216	28.8%
alwaysMin	6	113	15.07%
randomBot	6	174	23.2%

```
if(numberOfRemainingMoves < 30):
    for i in remainingMoves:
        nextPosition = pretendMove(i, remainingMoves)
        check if nextPosition is 2-weak
        if yes:
            play i
    if no i returns a 2-weak position:
        play max remainingMove

else if(numberOfRemainingMoves < 200):
    for i in remainingMoves:
        nextPosition = pretendMove(i, remainingMoves)
        if(numberOfMovesInNextPosition < 20):
            check if nextPosition is 2-weak
            if yes:
                play i
    if no i returns a 2-weak position:
        play max remainingMove

else:
    play max remainingMove
```

The logic of the bot is to see if playing a move will return a 2-weak position, thereby giving the opponent a bad position to play from. If there is a move that returns a small 2-weak position, play it. If not, play the largest remaining move, which will only get rid of one move, and hope that your opponent makes a mistake, letting you put them in a 2-weak position. One of the big goals for this bot is to make it play efficiently and keep the calculation times to a minimum. This means that we can't test for 2-weak positions after playing every move in a large position since the calculation time for that would be close to n^5 where n is the number of remaining moves. Through a lot of testing, we found that starting the checks for 2-weak positions when the number of remaining moves was less than 30 was optimal for maximizing the number of wins while staying within the time constraint. Although this is a relatively small position, there are still many positions that fall into this category. We have also seen that in larger positions opposing bots rarely have a better strategy and therefore often lose once the position becomes small.

Adding this bot to the round robin we have been running, we got the following results.

Table 5. Round Robin Tournament with three rounds, where each match between two bots consists of 50 games. Each match gets scored as follows, 3 points for a win, 1 point for a tie, and 0 points for a lose.

Bot Name	**Match Score (54 maximum)**	**Total Wins**	**Win Percentage**
peekThen2Weak	52	781	86.78%
maxThen2Weak	43	722	80.22%
maxThen1Weak	39	707	78.56%
maximalOdd	27	427	47.44%
alwaysMax	16	221	24.56%
alwaysMin	6	122	13.56%
randomBot	4	170	18.89%

3 Introducing Elo to keep track of bot success

The Elo rating system, named after Arpad Elo, is commonly used to represent the relative skill level of players in various games including board games like chess and Scrabble, and video games like League of Legends. The results of the round robin tournaments showed that certain bots are almost always able to beat other bots. This led to the idea of implementing an Elo ranking system for the bots. That way, bots could be tiered in some capacity and we can get a sense of how likely one bot is to beat another.

To generate an Elo ranking for each of the bots, we used a series of round robin tournament with Elo for each bot being calculated after it completed a match. Suppose bot A is playing against bot B. After the match the winning bot gets a score of 3 and the losing bot gets a score of 0. In the case of a tie, both bots get a score of 1. The formula for calculating each bots Elo is

$$E_A = \frac{1}{1 + 10^{\frac{R_B - R_A}{400}}}$$
$$E_B = \frac{1}{1 + 10^{\frac{R_A - R_B}{400}}}$$
$$R'_A = R_A + 20 * (BotAScore - 2 * E_A)$$
$$R'_B = R_B + 20 * (BotBscore - 2 * E_B)$$

where R_A and R_B are the bots' Elos going into the match and R'_A and R'_B are the bots' Elos following the match. The scoring was set in this manner so that if two bots of equal Elo played and the match resulted in a tie, neither bot's Elo would change. Also, with this system the bot's Elo would change fairly aggresively based on a win or loss. This was done in order to make sure that we can get fairly accurate Elos quickly. All the bots were given a base Elo of 800 except for alwaysMin with an Elo of 400, maxThen1Weak with an Elo of 1600, and a few other bots with various strategies. We then ran a round robin tournament with 14 bots including all the ones mentioned so far, which consisted of 4 rounds with each match having 25 games. We set the bots Elos in accordance with the results of this round robin and ran another round robin with the same bots. This time, three rounds with each match having 30 games. After this, we scaled all the bots Elos by setting the Elo of the worst bot, alwaysMin, to 400 and adjusting the rest of the Elos accordingly.

At this point, we had the following Elo ratings for each of the aforementioned bots.

Table 6. Bots and their respective Elo ratings after a series of round robin tournamets to initialize them.

Bot Name	**Elo**
peekThen2Weak	2605
maxThen2Weak	2514
maxThen1Weak	2279
maximalOdd	1313
alwaysMax	759
randomBot	429
alwaysMin	404

Once we had a fairly accurate Elo rating for each bot, we changed the formula for calculating Elo. Now, a winning bot would get a score of 1, a losing bot would get a score of 0, and in the case of a tie each bot would get a score of 0.5. The new Elo formula is

$$\begin{aligned}
E_A &= \frac{1}{1+10^{\frac{R_B-R_A}{400}}}\\
E_B &= \frac{1}{1+10^{\frac{R_A-R_B}{400}}}\\
R'_A &= R_A + 16 * (BotAScore - 2 * E_A)\\
R'_B &= R_B + 16 * (BotBScore - 2 * E_B).
\end{aligned}$$

This is a more conservative system so that the bots don't change score so drastically. We hope that as more bots and algorithms are deveoloped, we can use this Elo system to more accurately determine their success.

4 Conclusion

One of the problems we mentioned earlier with the journey to finding an optimal strategy for the game of Sylver Coinage is the lack of a way to evaluate a position. Through our research, we've developed four ways of analyzing a position 0-weak, 1-weak, 2-weak, peek-then-2-weak. Building on this, we can discuss two more definitions.

Definition 6. *We say that a position is* n-weak *if for each remaining move, $j > 1$, there is a different unique remaining move, $k > 1$, such that playing k in response to j returns a* (n - 1)-weak *position. For example, in a 3-weak position, for every $j > 1$ there is a response $k > 1$ such that the resulting position after playing both moves is a 2-weak position.*

Definition 7. *We stay that a position is* ∞-weak *if for every move greater than 1 that the current player can play for the rest of the game, their opponent will always have a unique response greater than 1 that returns a weak position.*

We see that an ∞-weak position is a lost position as the opponent will always have a response to keep the parity odd, eventually returning the position with 1 being the only legal move. It is important to note that finding an ∞-weak position is not guaranteed, and that not all lost positions are necessarily ∞-weak. Also, it is very computationally difficult to determine if a position is ∞-weak as we have to calculate every possible remaining position. Even just calculating if a position is n-weak for $n > 2$ gets very computationally expensive very quickly. However, surely finding a move that puts our opponent in a 3-weak position is better than a move that puts our opponent in a 2-weak position. Therefore, logic says that with enough computational power, a bot that searches for a move that returns a 3-weak position, then, failing to find one, searches for a move that returns a 2-weak position, and so forth would be stronger than our current bots. This shows that still, there is much work to be done in the search for an optimal strategy for the game Sylver Coinage.

Our bots provide headway in addressing the second and third problems we mentioned in the introduction regarding the analysis of the game. With regards to the second problem, the various strategies implemented by the bots provide some material concerning strategies for winning a game of Sylver Coinage. With regards to the third problem, although we still have no way of naively evaluating a position, the hierarchical nature of our bot strategies suggests that our analysis of position has some value. That leads to the likelihood of finding ways to evaluate a position, even if that value comes from a bot's analysis.

We hope that with the development of the bots thus far and the growth of Sylver Coinage as a competitive bot played game, we can make great headway on the problems associated with finding a winning strategy for the game.

References

1. Berlekamp, E.R., Conway, J.H., Guy, R.K.: Winning ways, for your mathematical plays. Academic Press, London (1982)

2. Blok, T.: Sylver coinage positions with g=2 (2021), https://userpages.monmouth.com/c̃olonel/sylver/Sylver_Coinage_positions_with_g=2.pdf
3. Eaton, R., Herzinger, K., Pierce, I., Thompson, J.: Numerical semigroups and the game of sylver coinage. The American Mathematical Monthly **127:8** (2020)
4. Guy, R.K.: Twenty questions concerning conway's sylver coinage. The American Mathematical Monthly **83** (1976)
5. Michael, T.S.: How to Guard an Art Gallery and Other Discrete Mathematical Adventures. Johns Hopkins University Press (2009)
6. Nowakowski, R.J.: …, Welter's Game, Sylver Coinage, Dots-and-Boxes, …. In: Combinatorial games (Columbus, OH, 1990), Proc. Sympos. Appl. Math., vol. 43, pp. 155–182. Amer. Math. Soc., Providence, RI (1991). https://doi.org/10.1090/psapm/043/1095544, https://doi.org/10.1090/psapm/043/1095544
7. Sicherman, G.: New results in sylver coinage (1991), https://userpages.monmouth.com/ colonel/sylver/index.html
8. Sicherman, G.: Late news of sylver coinage (1996), https://userpages.monmouth.com/ colonel/sylver/index.html
9. Sicherman, G.: Theory and practice of sylver coinage. Integers **2** (2002)