Voltorb Flip Algorithm Analysis

Evin Jaff and Lane Bohrer

Washington University in St. Louis,
St. Louis, Missouri

Spring 2021

Contents

Acknowledgements

Evin Jaff and Lane Bohrer would like to extend their thanks to Professor Bill Siever for his support as an advisor for this independent study. Without his guidance, we would not have been able to accomplish as much as we did this semester.

Evin would like to thank the development team behind HeartGold and SoulSilver for creating one of the best Pokémon games of all time as well as for creating a minigame that tormented him so much that he decided to study it in college.

Lane would like to thank Zakary Littlefield for writing an open source Voltorb Solver in 2010 that allowed her to spend the semester adding to existing code instead of writing a solver from the ground-up. She would also like to thank her friends who laughed only the appropriate amount when she told them what she was working on.

1 Introduction

The game Voltorb Flip was originally written for the 2009 video games Pokémon HeartGold and SoulSilver, a remake of the famous Pokémon Gold and Silver for the GameBoy Color released in 2000. Voltorb Flip was developed as a replacement for a slot machine game in the original version because of changes in rules from regulatory bodies such as PEGI on the element of gambling in video games [1].

In response, Voltorb Flip was created. It can best be described as a fusion of game concepts from Minesweeper, Sudoku, and Nonogram. The fusion of these concepts results in a game that is probabilistic in nature, but requires skill to determine the best move and still has many scenarios in which boards are discreetly solvable. In this report, we will discuss the mechanics of Voltorb Flip and a solution algorithm, as well as report our statistical analysis and discuss alternative implementations and future areas to explore.

[1]Nintendo's statement on slot machines can be found here: https://www.gamesradar.com/european-pokemon-platinums-missing-game-corner-explained/

2 Game Mechanics

2.1 Pokémon HeartGold and SoulSilver Implementation

The original implementation of Voltorb Flip in Pokémon HeartGold and SoulSilver is relatively simple. It uses a 5x5 board with row and column headers that indicate a sum of scoring tiles on top and the number of Voltorbs (or sometimes referred to as bombs) on the bottom. An image of a traditional game is shown in Figure 1.

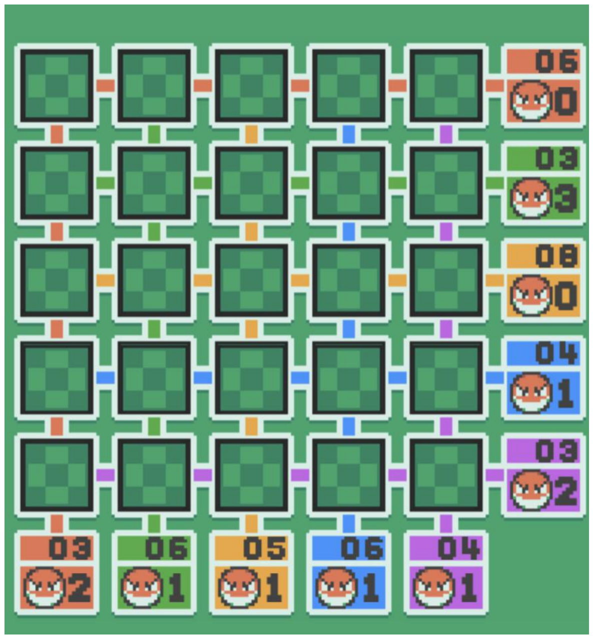

Figure 1: A picture of a Voltorb Flip game from Pokémon HeartGold and SoulSilver (2009)

2.2 Generalized Rules

To start, the player sees a five by five game board with tiles flipped over. Each of these 25 tiles holds a value of three, two, one, or a Voltorb. Off to the side, each row and column has a sum of the number of points and the number of Voltorbs, which function like Minesweeper's bombs, within that row or column. With this knowledge, the player selects tiles to flip over, gaining coins based on tile values. The round ends when either a Voltorb is flipped or there are no scoring tiles (twos or threes) remaining.

2.3 Level and Coin System

After a round ends, the game will select a level for the next round based on the player's success in the prior round. If the player won the round (cleared the board of twos and threes without flipping a Voltorb), they will advance to the next level. If they lost, their new level is determined based on how many tiles were flipped over in the prior round, jumping down to a level that matches that number or the current level, whichever is lower. The highest possible level is 8. The game continues on until the player chooses to withdraw.

Coins function as the scoring system in Voltorb Flip. A global wallet tracks the player's coins throughout their gameplay session. Within a round, the player gains coins that will be added to the wallet at the end of the round. The first tile flipped is added to the score, and then the current score is multiplied by each subsequent flipped tile. When a Voltorb is flipped, it acts as a zero, bringing the score to zero and ending the round. A player can choose to walk away from the game whenever they want.

3 Voltorb Flip Solver

With these game rules and mechanics, we look forward to determining solutions to the board. Unlike similar problems such as Sudoku, it is not possible to return a completed final board that solves a given board due the conditioning of future moves from the results of previous moves. Instead, Voltorb Flip ”solutions” are similar to the solutions generated by a chess algorithm for example, in which the algorithm returns its decision of the best move.

3.1 Human Solution Methods

When a player begins their first game, they will quickly pick up on some key strategies and rules of the game. A row with a higher point total is a good bet for having more twos and threes, and a column with no Voltorbs will be very safe to flip in, for example. The human strategies get more complex than this, but follow logical rules like these. Some of them are listed here:[2]

1. **The Safe House Rule**: If the board has a row or column with zero Voltorbs, that row will have a zero percent chance of flipping over a Voltorb.

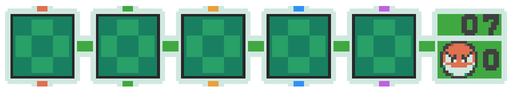

Figure 2: A row to which the Safe House Rule applies

2. **The n-Sum Rule**: If the number of Voltorbs and the sum of points add up to five, we can determine that the row has only ones and Voltorbs, and is therefore not worth flipping in.

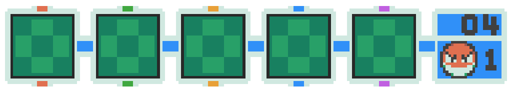

Figure 3: A row to which the n-Sum Rule applies

3. **Extended Subset n-Sum Rule**: This is an extension of the n-sum rule where the relevant row/column has some number of tiles that are known either through flipping the tile over or other deductions. Excluding them and taking out their effect on the sums, you can apply the same logic to the subset as a standard n-sum of that subset's length.

[2]photos from https://www.dragonflycave.com/johto/voltorb-flip

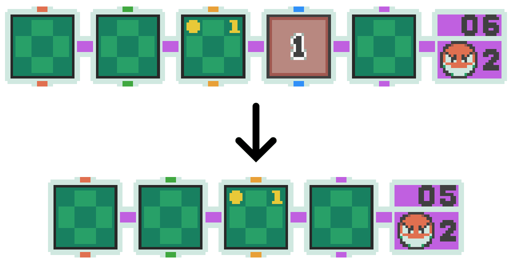

Figure 4: Applying the Extended Subset n-Sum Rule

4. **Extended n-Sum Rule (Only Voltorbs Left)**: If via the extended flipped tiles rules or the general n-sum rule, you reach a point in which the sum of numbered tiles (one, two, and three) is zero, you can guarantee that all the tiles in that set are Voltorbs. The Extended Subset n-Sum Rule can then be applied to the row or column that intersects the identified Voltorb to gleam further information.

3.2 Implementation of An Algorithm

While players are limited to those heuristics described above because they can't picture all possible combinations, a computer doesn't have this limitation. For this project, we adapted a solver [3] that works by generating possible boards, narrowing them down to see which ones fit the known constraints, and generating the probability that each spot will have a certain value (specifically Voltorbs and scoring tiles). The choice of which tile to flip was initially left up to the user, but was adapted to be one of a handful of choices:

1. **MinRisk**, where the tile with the lowest probability of being a Voltorb is chosen;
2. **ScoreFirst**, where the tile with the highest probability of being a scoring tile is chosen;
3. **CutoffN**, where the algorithm starts as ScoreFirst and changes to MinRisk after n iterations;
4. **BogoFlip**, where the tile to flip is chosen at random.

These algorithms are compared in detail more in section 4.3.

3.3 Generalization of n-length board

One of the ways we explored the solving algorithm was to expand it to work on a generic size board. The original solver was built specifically for the game's standard five by five board, so the code was genericized to an n-length square board through refactoring nested for loops into a more flexible recursive pattern. The iterative and recursive solutions do similar work, validating each potential board row by row and column by column. The difference between the two methods is that the recursive solution is not bound by static loops, and is free to dynamically run the additional loops that become necessary to solve a larger board. The configuration of loops will vary based on the dimensions of the board that is being solved.

[3]Written by Zakary Littlefield, https://sourceforge.net/projects/voltorb/

3.4 Proof of Complexity for an Iterative Solution

First, we assume that no data is cached and we start with empty memory. Then, we generate all the boards that are possible with our row/column constraints. This is done in a n-tuple nested loop, where the loop runs from 0 to $\binom{n}{c_{n,r}}$ where $c_{n,r}$ is the number of Voltorbs in that row given by the row constraints. This will generate n sums in the form:

$$\sum_{i_n=0}^{\binom{n}{c_{n,r}}} \sum_{i_{n-1}=0}^{\binom{n}{c_{n-1,r}}} \sum_{i_{n-2}=0}^{\binom{n}{c_{n-2,r}}} ...f$$

where f is the work function inside the loop. The value $c_{n,r}$ is of the range $[0, n]$. Due to the nature of the nCr function, the sum will vary wildly as a result of this in accordance with the nCr function where the result is $\frac{n!}{([0,n])!(n-[0,n])!}$. The best-case value to yield a low number would be the ends of the function at either 0 or n, in which the function will return 1. We can determine the average-case value by taking the mean of the nCr function, which will be equal to

$$\frac{(\sum_{i=1}^{n} nCr\,(x, i))}{n} = \frac{2^n}{n}$$

Evaluating this sum yields an average of $2^n/n$, which we will assume as the upper bound for each of our sums. So then our resulting sum becomes

$$\sum_{i_n=0}^{\frac{2^n}{n}} \sum_{i_{n-1}=0}^{\frac{2^n}{n}} \sum_{i_{n-2}=0}^{\frac{2^n}{n}} ...1$$

Where there are $2n$ sums since we have to evaluate for vertical and horizontal constraints. We can then evaluate the sums for a final value. This will give us $2 * 2^n$ and see that our final complexity for the average case is $\Theta(2^n)$.

To find the best case, we do the same steps, except we choose to either set $c = 0$ or $c = n$. This yields an upper sum bound of:

$$2n * (\frac{n!}{(0!(n-0)!}) = n$$

We can then evaluate the summation with those upper bounds and obtain a best-case complexity of $\Theta(n^2)$. However, we can easily reason that this situation will never realistically occur in a game of Voltorb flip, since this complexity implies that every row an column would either be full of Voltorbs or have no Voltorbs present which would not be possible.

To find the worst case we use $\frac{n}{2}$ since it yields the maximum value of the nCr function:

$$2n\frac{n!}{(\frac{n}{2})!(n-\frac{n}{2})!} = 2n\frac{n!}{(\frac{n}{2})!^2}$$

Unfortunately, big O notation is tougher to calculate, but a limit test yields:

$$\lim_{x\to\infty} \frac{h(x)}{g(x)} = \lim_{x\to\infty} \frac{2n\frac{n!}{(\frac{n}{2})!^2}}{2^n} = \infty = \Omega(2^n)$$

$$\lim_{x\to\infty}\frac{h(x)}{g(x)} = \lim_{x\to\infty}\frac{2n\frac{n!}{(\frac{n}{2})!^2}}{n!} = 0 = O(n!)$$

Therefore, $n^{2*\frac{n!}{(\frac{n}{2})!^2}}$ is $\Omega(n^n)$
With this work, we can generate these conclusions:

Best-Case	Worst-Case	Average-Case
$\Theta(n^2)$	$\Theta(2^n)$	$O(2^n)/\Omega(2^n)$

3.5 Proof of Complexity for a Recursive Solution

Due to the structure of our recursive method, we can modify the previous summation of one set of for loops and set it as the work function $f(n)$ of the recursive function. We can do this, since the recursion is mainly focused on dynamically constructing existing loops and ignores a tree structure, so the original complexity of the constant work inside should stand excluding a few performance tweaks that would have already been implemented in the iterative solution. The recursive function $f(n)$, which represents the individual for loops, is defined as the components of a per-sum for loop, which will be expressed as:

$$f(n) = \sum_{i_n=A_{min}}^{A_{max}} \left(\sum_{j_n=0}^{\binom{n}{c_{n,r}}} 1\right)$$

A_{max} and A_{min} represent constants, specifically the range of what possible scoring tiles exist, with 0 being the minimum (a Voltorb) and 3 being the highest (a tile of 3) possible. This will result in a constant multiplier which will not directly affect the asymptotic complexity and will be excluded for that reason in this proof. However, this would be necessary in other scenarios to account for other setups in which the scoring tiles system might affect the complexity in this way. We setup the recurrence to follow the program structure such that:

$$T(n) = T(n-1) + f(n) \text{ where } T(0) = C$$

Master method will not be usable in this situation due to the style of recursion, however, we can directly solve this problem using our knowledge of recursion depth and similar incrementally recursive functions such as the Gamma function. Since the only value that will change between each function will be $\binom{n}{c_{m,r}}$ (specifically $c_{m,r}$), we can index them similar to the iterative proof in which $c_{m,r}$ represents the number of Voltorbs/scoring tiles as dictated by the conditions of the loop. Evaluating this for an n-dimensional board gives us a sum of n total f(n) terms that run from n to 0. With this form, we can make a simplifying assumption about the nCr function similar to the iterative solution that since $c_{m,r}$ is on the order of $\Theta(1)$, we can factor these f(m) terms into parentheses and obtain one unifying summation that will be just like the iterative solution. In addition to some other simplification from evaluating the inside sum, we obtain this for our value of $T(n)$.

$$T(n) = n\left(\sum_{j_n=0}^{\binom{n}{c_{n,r}}} 1\right)$$

One thing to note is that the multiplier n is still subject to a potential outside constant multiplier similar to the iterative proof depending on how many scoring tiles are run, but this will not affect the complexity with our assumption that tiles range on a limited set from $[0, 3]$. With this together, we can see that the underlying complexity will be the same when we evaluate the best, worst, and average cases for the nCr function. Allowing us to conclude this as our table of complexity for the recurrence

Best-Case	Worst-Case	Average-Case
$\Theta(n^2)$	$\Theta(2^n)$	$O(2^n)/\Omega(2^n)$

4 Simulations

4.1 Simulation Setup

For our simulations, we generated boards to test our solver. We wrote a board generator, which would create boards based on the difficulty (level) and board size (n). The process for these can be seen in the tables in Appendix A. We also wrote a MoveLogger to track what occurred during our simulations, including turn count, tiles flipped, and score. This is how we produced the following results.

4.2 BogoFlip

As a baseline, we first analyzed a large simulation of BogoFlip. On a simulation of 2 million boards, we analyzed the number of turns achieved on all boards (Figure 5) and on just the boards that were won (Figure 6). As expected, BogoFlip preforms poorly, losing most games after the first or second flip. As turns go on, the likelihood of staying alive falls off quickly, giving the histogram a strong right skew. In specifically the winning boards, which make up .5219% of boards played, the distribution is much closer to normal because the loses are taken out and what is shown is just the number of turns it takes to win.

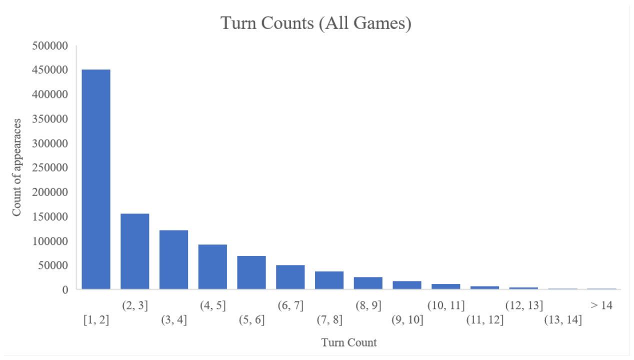

Figure 5: Turn Counts on All Boards

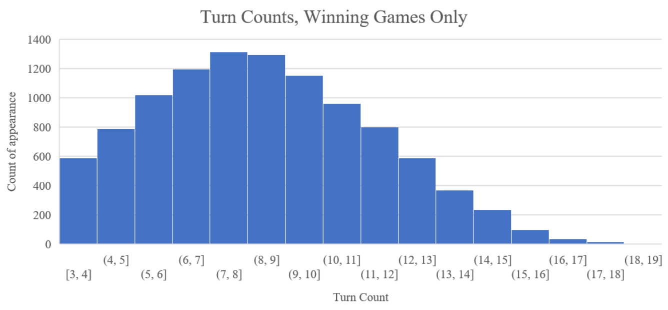

Figure 6: Turn Counts on Winning Boards

4.3 Comparison of Solvers that use calculated probability

Comparing our three solving algorithms shows their differences in success. For the purpose of this simulation, all four algorithms were fed the same set of 500 boards to account for variance in the board generation as a factor in differing success. The board were all level 2. When we originally wrote MinRisk and ScoreFirst, we thought that those two seemed quite different as a human player. However, once we see them played over many boards, it becomes clear that they end up acting quite similarly. In fact, MinRisk, ScoreFirst, and CutoffN (with any N value, N=8 is shown here as an example) all result in nearly identical win rates (Figure 7). BogoFlip, as expected performs at a poor rate, winning one of the 500 boards.

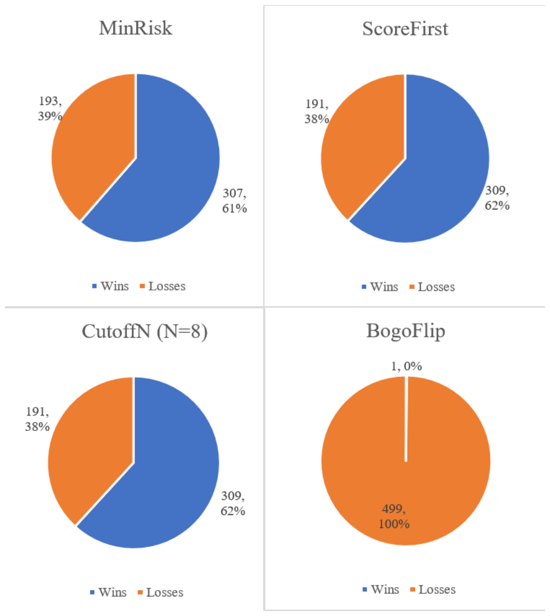

Figure 7: Win Percentage by Strategy

This similarity in win rate begs the question: are they different in any other way? As shown in Figure 8, the turn count to finish a game is much lower in ScoreFirst. This is because it seeks out scoring tiles more efficiently than MinRisk which finds them while flipping other low-risk tiles as well. For this reason, ScoreFirst appears to be the better strategy.

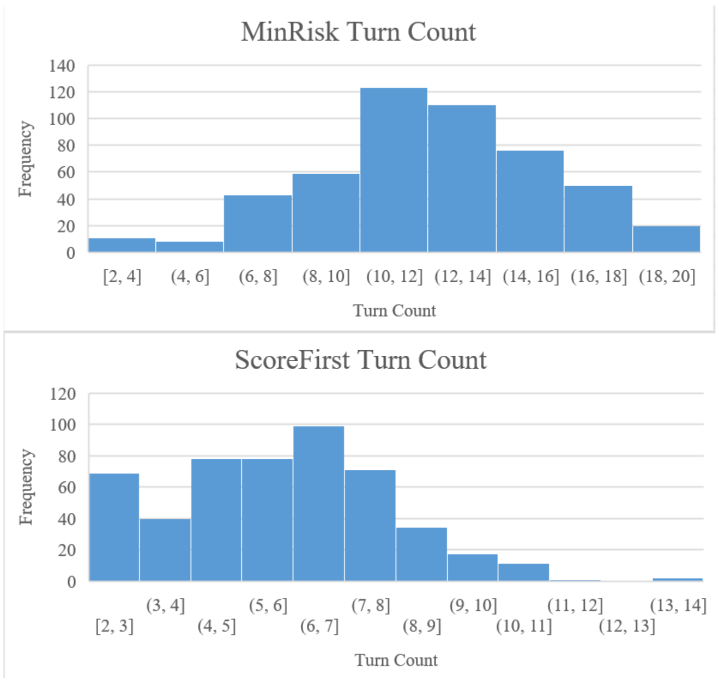

Figure 8: Turn Counts for MinRisk and ScoreFirst

4.4 The Effects of Board Size on Algorithms

Once the solver had been modified to handle generic board sizes, we ran a simulation to test our calculations for runtime. While the limitations of our computers kept us from running board sizes above 6x6, the data show a clear trend that falls within our expected range of exponential to factorial (Figure 9).

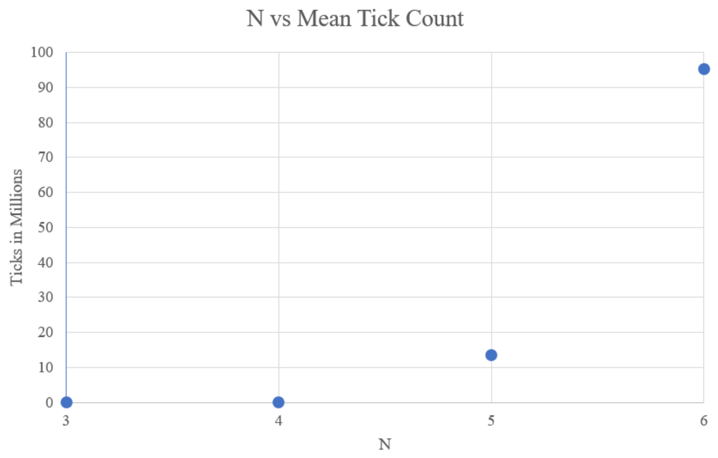

Figure 9: Board length N vs tick count

5 Future Work and Considerations

5.1 Reduction of Complexity using a partially correct algorithm

Due to the high complexity of probability calculation, one of the easiest ways to bring down the complexity would be to implement a version of the solver that takes advantage of the human solution strategies mentioned. This algorithm would not always guarantee an objectively correct method since the human solution method is inherently flawed, however the savings in complexity could outweigh the cost of partial correctness. Additionally, there are often multiple correct solutions on a given board, so there is a strong chance that the human solutions method would end up picking another best move or one that is extremely similar in its risk or reward.

The best way to achieve this would be to implement a collection of our human heuristics as a complete strategy. A list of the estimated complexity for each heuristic is detailed below:

- **The Safe House Rule**: This rule would have a complexity of $\Theta(2n)$ to scan the sums and Voltorb counts of each row.
- **The n-Sum Rule**: Similar to the Safe House rule, the n-Sum rule would have a complexity of $\Theta(2n)$ to scan the sums and Voltorb counts of each row.
- **Extended Subset n-Sum Rule**: This would have a varied best and worst case complexity based on how many subsets were generated. The implementation could change, but the expected complexity to implement this would most likely be on the order of $\Theta(n^3)$ to look for and analyze each subset.
- **Extended n-Sum Rule (only Voltorbs left)**: This extension of the n-sum rule could be run as a simple check to mark known tiles. This would most likely only run for $\Theta(n^2)$ if this heuristic was run for all the boards.

5.2 Parallelization of a Solver

We strongly believe that this solution algorithm could be parallelized. We envision that this would be done by dividing the workload when it during the phases of generating states and applying constraints for columns and rows. However, this implementation could arise the possibility of redundant states being processed, which could be handled after-the-fact when combining the results of the threads later in the solution. Additionally, we know that this would be possible since similar algorithms such as Sudoku solver have had implementation written for GPGPU computation such as Nvidia's CUDA.[4]

[4]Parallel Sudoku: https://github.com/vduan/parallel-sudoku-solver

A Further Information about Board Generation

When translating to an n-length board, the original allocations of Voltorbs it becomes necessary to to determine a scalable method to determine the total number of Voltorbs and scoring tiles to insert in a board. We used a percent method to handle scaling of the board to an n-length board by getting the percent composition of each board template and use that to determine the new board quantity. We can find the required number of a certain tile for a given board template by multiplying the board's length n from a given board.

For example, if we wanted to generate a 50-length board of difficulty level 5, we can select a random level 5 configuration, for example the first one, with 7 two tiles, and do $.28 * 50^2$ to know to insert 700 twos, $.04 * 50^2$ to know to insert 100 threes, and 800 Voltorbs.

In our implementation, we opted to use a more inefficient, but still reasonable method to assign spaces. We opted to generate a two random numbers that represented a board index, and then attempt to place a scoring tile there. If a scoring tile was already there, it would then keep trying to generate a new position until it eventually found one. I would then repeat this for all the scoring tiles, as well as for the Voltorbs. Then, the rest of the board was filled in with 1's. It would've been overall more efficient to keep an array of indexes to where all the non-scoring tiles were, and then eliminate them from the array. However, in practice, the practical running time difference between generating a 5-length and a 100-length board is negligible.

The table containing the percentage weights as well as the original amounts of tiles for a 5x5 game is detailed on the next page.[5]

[5]Table obtained from Bulbapedia https://bulbapedia.bulbagarden.net/wiki/Voltorb_Flip

Level	Number of 2's	Number of 3's	Number of Voltorbs	Number of Coins	Percentage of 2's	Percentage of 3's	Percentage of Voltorbs
Level 1	3	1	6	24	12.0%	4.0%	24.0%
	0	3	6	27	0.0%	12.0%	24.0%
	5	0	6	32	20.0%	0.0%	24.0%
	2	2	6	36	8.0%	8.0%	24.0%
	4	1	6	48	16.0%	4.0%	24.0%
Level 2	1	3	7	54	4.0%	12.0%	28.0%
	6	0	7	64	24.0%	0.0%	28.0%
	3	2	7	72	12.0%	8.0%	28.0%
	0	4	7	81	0.0%	16.0%	28.0%
	5	1	7	96	20.0%	4.0%	28.0%
Level 3	2	3	8	108	8.0%	12.0%	32.0%
	7	0	8	128	28.0%	0.0%	32.0%
	4	2	8	144	16.0%	8.0%	32.0%
	1	4	8	162	4.0%	16.0%	32.0%
	6	1	8	192	24.0%	4.0%	32.0%
Level 4	3	3	8	216	12.0%	12.0%	32.0%
	0	5	8	243	0.0%	20.0%	32.0%
	8	0	10	256	32.0%	0.0%	40.0%
	5	2	10	288	20.0%	8.0%	40.0%
	2	4	10	324	8.0%	16.0%	40.0%
Level 5	7	1	10	384	28.0%	4.0%	40.0%
	4	3	10	432	16.0%	12.0%	40.0%
	1	5	10	486	4.0%	20.0%	40.0%
	9	0	10	512	36.0%	0.0%	40.0%
	6	2	10	576	24.0%	8.0%	40.0%
Level 6	3	4	10	648	12.0%	16.0%	40.0%
	0	6	10	729	0.0%	24.0%	40.0%
	8	1	10	768	32.0%	4.0%	40.0%
	5	3	10	864	20.0%	12.0%	40.0%
	2	5	10	972	8.0%	20.0%	40.0%
Level 7	7	2	10	1152	28.0%	8.0%	40.0%
	4	4	10	1296	16.0%	16.0%	40.0%
	1	6	13	1458	4.0%	24.0%	52.0%
	9	1	13	1536	36.0%	4.0%	52.0%
	6	3	10	1728	24.0%	12.0%	40.0%
Level 8	0	7	10	2187	0.0%	28.0%	40.0%
	8	2	10	2304	32.0%	8.0%	40.0%
	5	4	10	2592	20.0%	16.0%	40.0%
	2	6	10	2916	8.0%	24.0%	40.0%
	7	3	10	3456	28.0%	12.0%	40.0%