Algorithm A

Algorithm A is essentially a series of cascading optimizations that occur in every iteration where it must choose a square to open. It was approached initially as an optimization to the trivial algorithm of mining tiles randomly. The idea was to make use of known safe tiles and known bomb tiles in order to make smarter choices and find the bomb list with fewer opened tiles. We began this approach by creating a list to keep track of known bombs and a set to keep track of safe squares. Beginning the algorithm, every time the algorithm chooses a new tile, it begins with iterating through every board element and adds the element to a list of unopened tiles if the element has a value of -1 (indicating that a tile is unopened. Next, the algorithm checks to see if the length of the list of known bombs is equal to the number of bombs given in the board json file. If so, it submits the bomb list. Otherwise, it iterates through each element in the board again, where we run a series of optimizations. First, if the element that it is currently on is a zero tile, it adds every neighbor of that tile to the safe square set. Next, if the current element is not a known bomb (not in the bomb list), then check to see if it has a value of 9, indicating a bomb that the algorithm might have missed. If it is not a confirmed bomb, we run an optimization algorithm on it to find confirmed bombs. Essentially, we go through and find every neighbor that is either uncovered or is a bomb. If the number on the tile is equal to the number of neighbors that are either uncovered or bombs, every uncovered element is going to necessarily be a bomb; thus, we add those elements to the bomb list and we need not mine the tile to know that they are bombs. For every element that is not of value 0, we run one more optimization to find known safe squares. Similarly to the confirmed bombs optimization, we find all neighbors of the current element that are confirmed bombs. If the number on the tile is equal to the number of confirmed bombs, we add every uncovered tile that is not a confirmed bomb to the safe tiles list. Once we are done iterating through every element, we iterate through our list of confirmed bombs and remove every confirmed bomb that is in our previously made uncovered tiles list. If our set of confirmed safe tiles is not of length 0, we randomly choose a tile to open from that set, as that will give us the most information about the remaining unopened tiles. Otherwise, we check to see if the unopened tiles set is of length 0. If it is, we have reached our final answer and we may submit our list of bombs. Otherwise, we randomly choose a tile from the list of unopened tiles (this is where we must make our algorithmic leap of faith :D).

Having run hundreds of tiles to affirm that the algorithm produces the correct bomb list, we are convinced of its practical correctness; however, it also makes sense. Any time we are forced to make a move that we are not 100% sure is a safe square, we run the risk of mining a bomb. This is ok, as the algorithm handles any bombs mined. For this reason and the above explanation, the algorithm passes both our practical and theoretical verification.

In terms of runtime, let's define the area of the board (length times width) as n. We have two for loops, where each for loop iterates through every element in the board. In the first for loop, we simply add tiles of value -1 to a list; thus, the first for loop runs in O(n) time. In the second for loop, we run up to two optimizations on each element, and each optimization iterates through each neighbor node, thus running in constant time. The second for loop thus also runs in O(n) time. Finally, we must run the for loops for every tile we must select, which runs n times in the case that we are forced to make a random move literally every time; thus, the algorithm runs in

O(n^2) time in the worst case. Something very important to note is that our python implementation happens to run with a worse run-time due to a lack of drive to fiddle with storing elements in fixed size arrays. We utilize dictionaries, lists, and sets to simplify operations, but our algorithm may be done using purely arrays. In the case that we use purely arrays, the algorithm runs in O(n^2) time.

Below we have several plots depicting the performance of our algorithm. Important to note is that at least 5 trials were conducted for each individual data point in the graph. All plots were done with extensive testing and plotted in matplotlib.

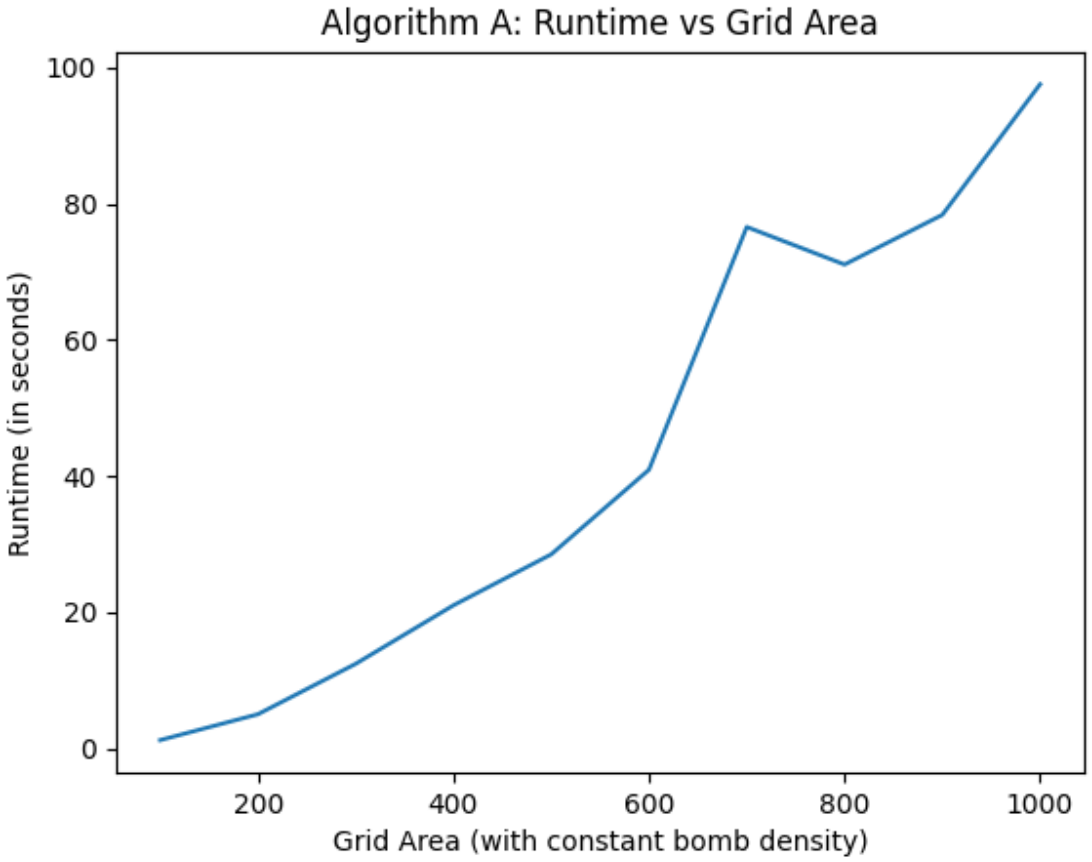


Figure 1: Here is our graph of Runtime vs Grid Area. Our Algorithm A takes much longer to execute as grid area increases, which is expected given its O(n^3) run-time.

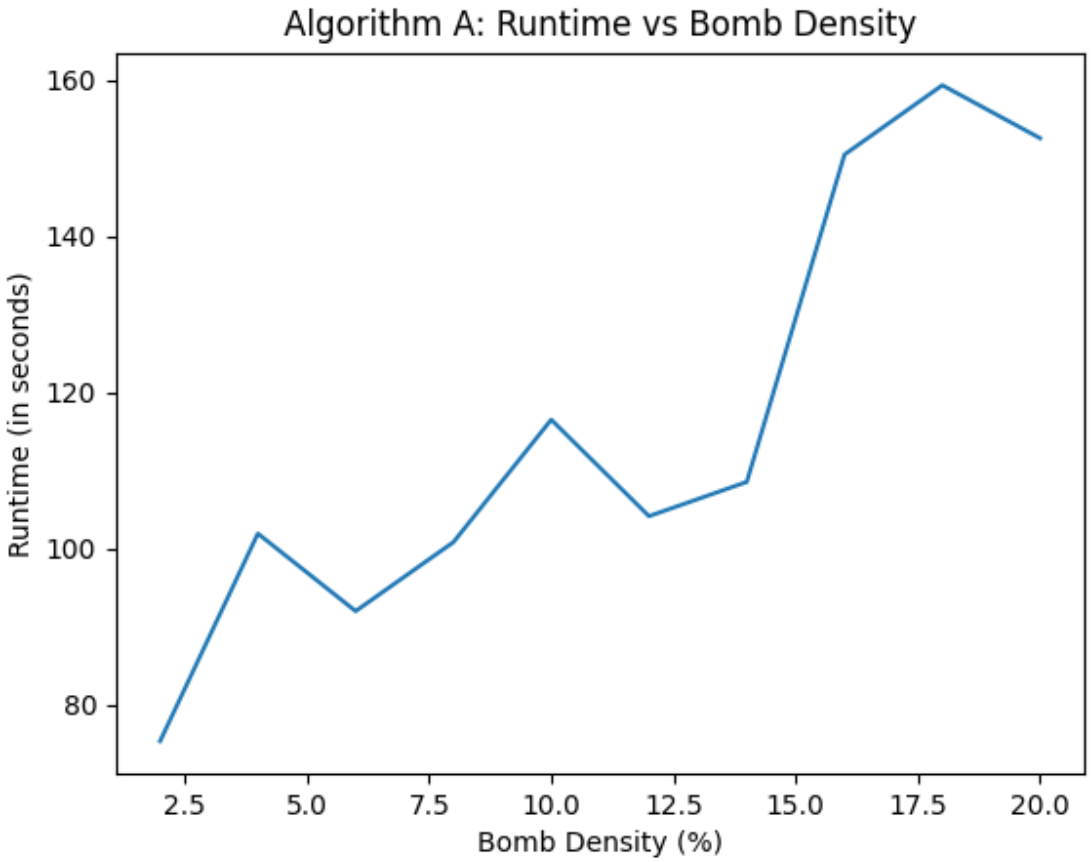


Figure 4: Here is our graph of Performance vs Bomb Density. Something important to note was that the grid area was 1000, which makes the runtime significantly larger but more asymptotically precise. There is clearly a positive correlation between bomb density and runtime, which was to be expected given the algorithm description.

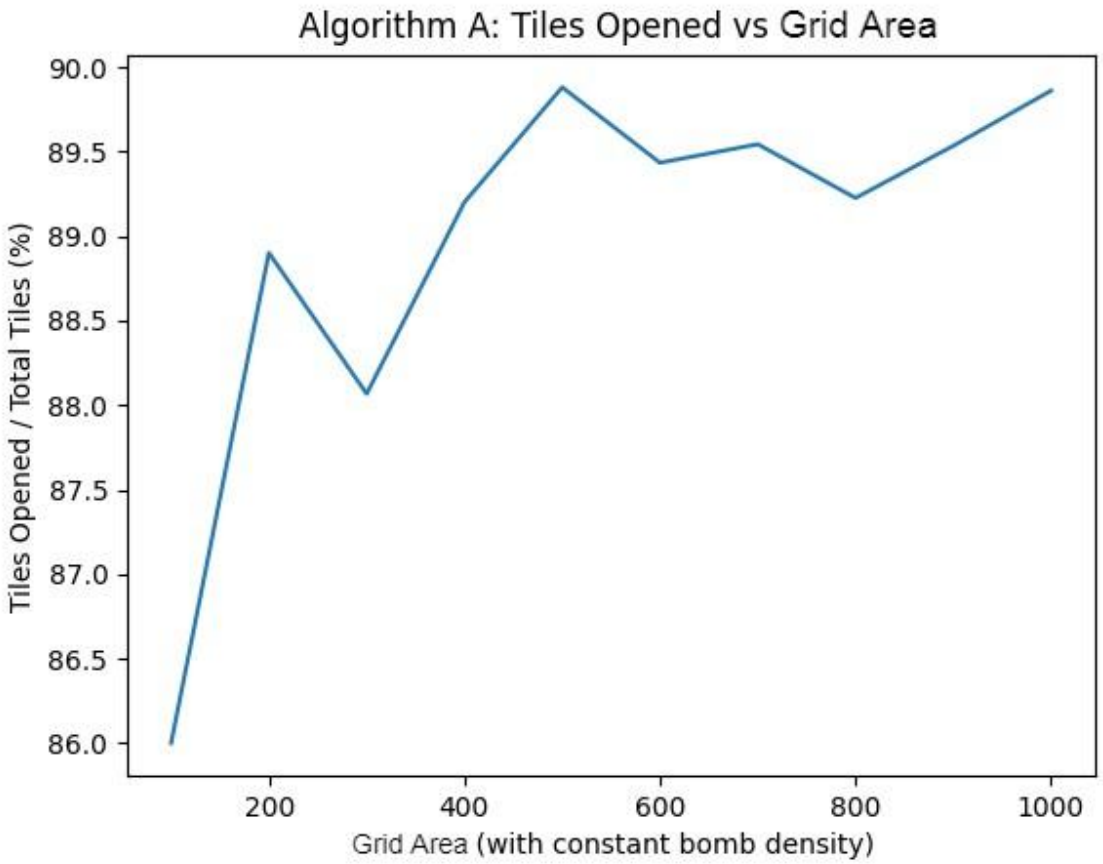


Figure 3: Here is our graph of Performance (“fraction of squares revealed”) vs Grid Area. Our Algorithm A performs significantly worse in terms of performance, as defined in the problem description, as grid area increases.

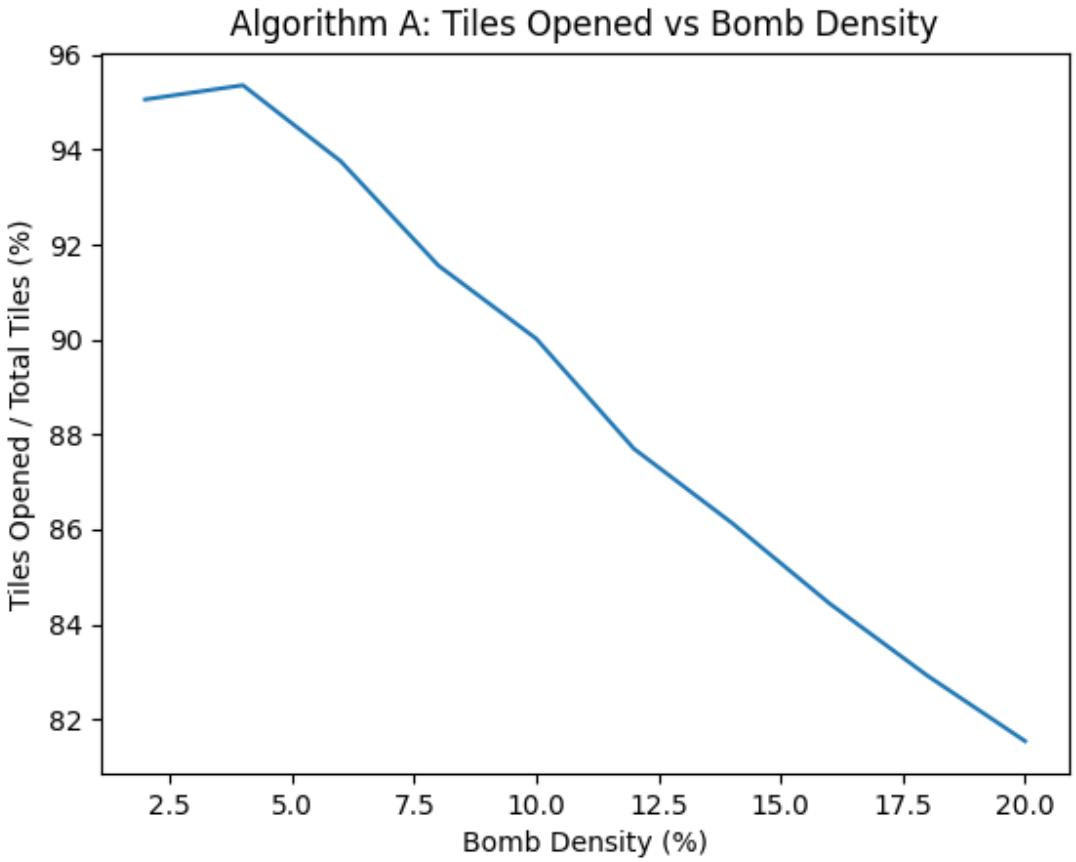


Figure 4: Here is our graph of Performance vs Bomb Density. Something important to note was that the grid area was 1000, which makes the ratio of tiles opened to bomb density worse but more asymptotically precise. There is an obvious negative correlation between performance and bomb density.

Algorithm B

Algorithm B is a simplified version of the constraint satisfaction problem. In general, we create a 2d array of size grid area squared in order to keep track of all constraints. We iterate through the grid, tile by tile, and, for discovered squares, we utilize the number on the tile to create a constraint equation, where the sum of the unknown neighbor tiles equals the number on the known tile minus the amount of known neighbor bombs. We thus put a row in our constraint array (or matrix) for every single known tile each algorithmic iteration. After we fill in our constraint matrix, we augment it with our constraint sum array (which is simply an array where each row is the aforementioned sum of tiles in the corresponding constraint matrix row) and perform a row reduction on the augmented matrix in order to calculate known values of squares. For each row, we look at the last entry. If it is a zero, we know that every single variable that doesn't have a zero in the row must have a value equal to zero, indicating that it is a safe square. In this case, we add it to our safe squares list. If it is not a zero, we sum up every "1" constant in the row and see if it is equal to the last number in the row. If it is, we know that every variable in that row has a value of 1, meaning every tile in the row holds a known bomb. We add all of those tiles to the bomb list. Now that we have a list of known safe tiles and known bombs, we update a grid that parallels the boardState grid with our known values. From there, if we have one or more known safe squares to choose from, we randomly choose a safe square. If not, we randomly open an unopened square.

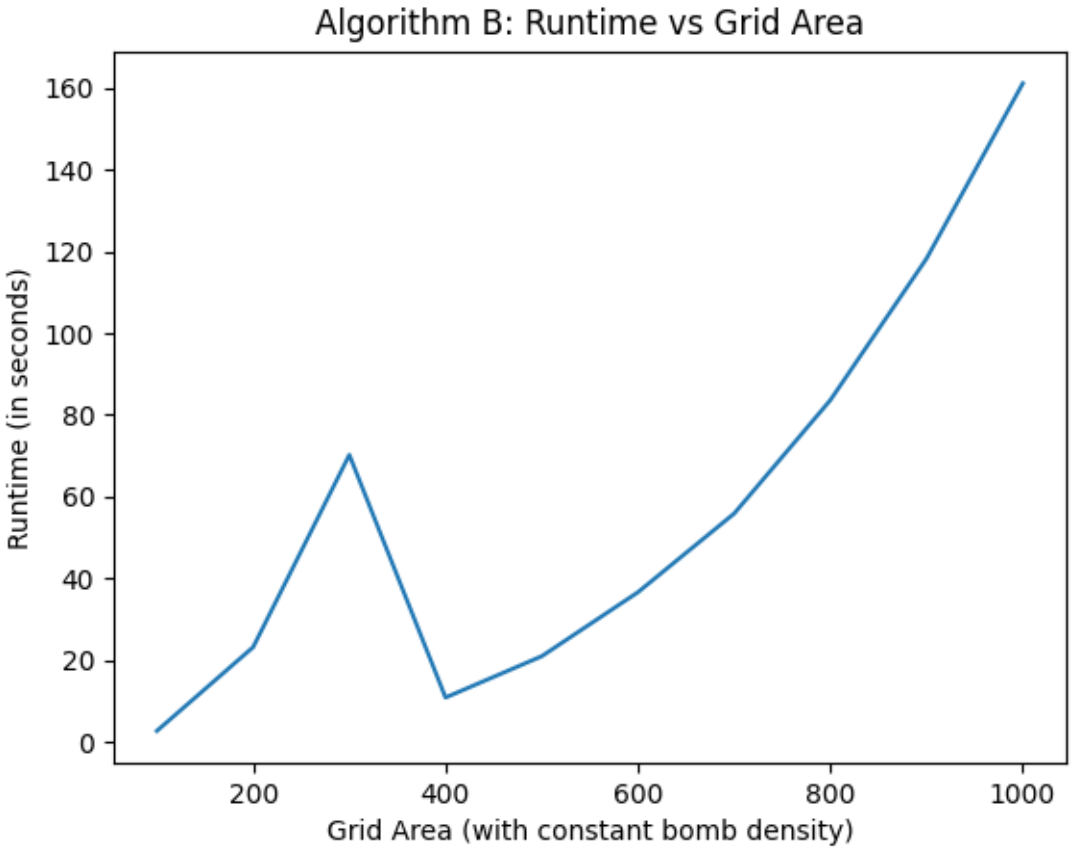
Something very important to note about this method is that it becomes far too inefficient on grid areas of greater than 300. We perform gaussian elimination recursively in order to row reduce our constraint matrix, and row reducing a matrix of area 10,000+ is highly inefficient; thus, we simply utilize the same constraint rules mentioned above for non-reduced matrices in order to find known safe tiles and known bombs. For this reason, the algorithm is a far smarter choice on matrices of area 300 or less. Algorithm A far outperforms this algorithm on matrices of grid area 400+. We believed that this approach was still important to implement, however, due to its novel and creative nature. Perhaps finding a way to either split up matrices or reduce the size of our constraint matrix would make this approach far more efficient.

We have conducted extensive testing to ensure this algorithm's practical correctness. Its theoretical proof is basically provided above. We are essentially just creating accurate systems of equations, inserting them into a matrix, and row reducing the matrix to solve known equations. If we need to make a random move and we mine a bomb, the algorithm adjusts and continues. For this reason, our algorithm is both practically and theoretically correct.

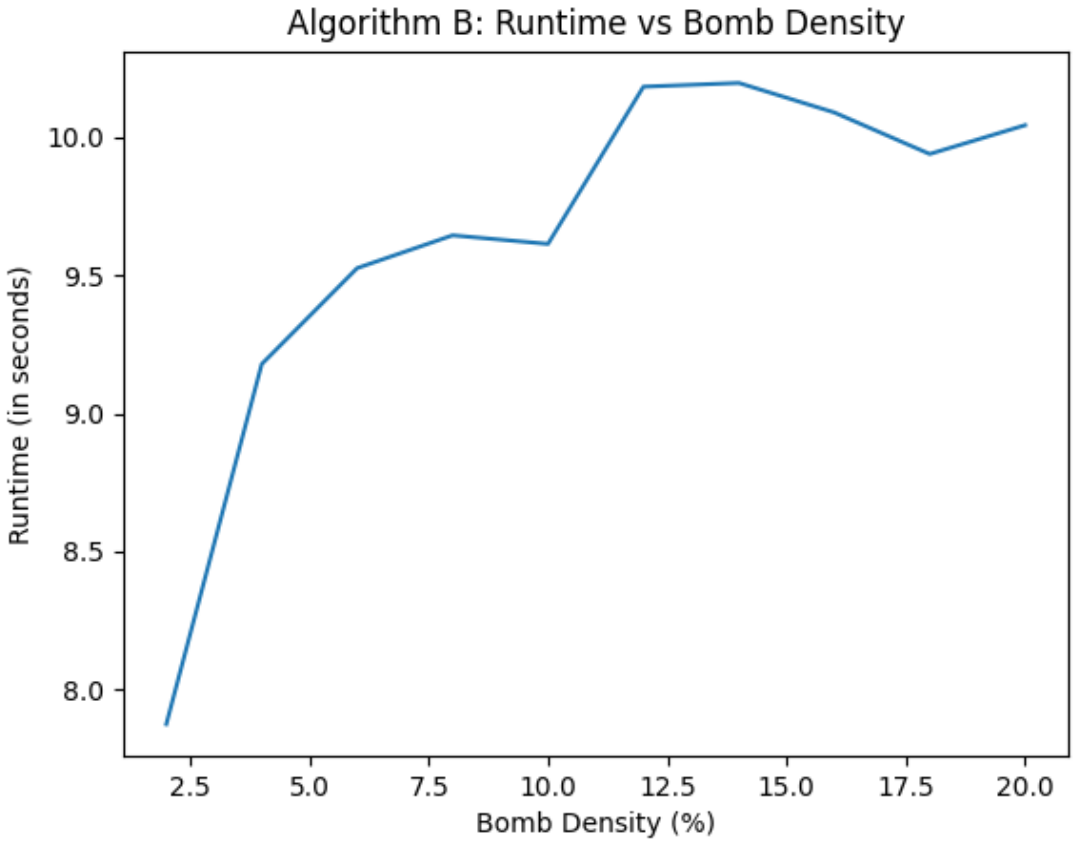
In terms of runtime, let's define the area of the board (length times width) as n. We have a constraint matrix of size n^2 that we iterate through and perform operations on for every algorithmic iteration, so we can focus on iterations and operations on this matrix, since our boardState array is only of size n. In our algorithm, we iterate through each square and choose it as our square to open in the worst case. For each square, we iterate through our board and add constraints to our constraint matrix. Once we add all of our constraints, we reduce our constraint matrix using gaussian elimination. Gaussian elimination, in the way we applied it, and in general, is simply O(m^3) for a mxm matrix. We have a constraint matrix of size n by n, where

n is the board area; thus, reducing this matrix is O(n^3). We then iterate through the constraint matrix and update our known square grid; thus, our runtime each iteration is dominated by our row reduction, which is O(n^3) done n times in the worst case; thus, our algorithm runs in O(n^4) time worst case.

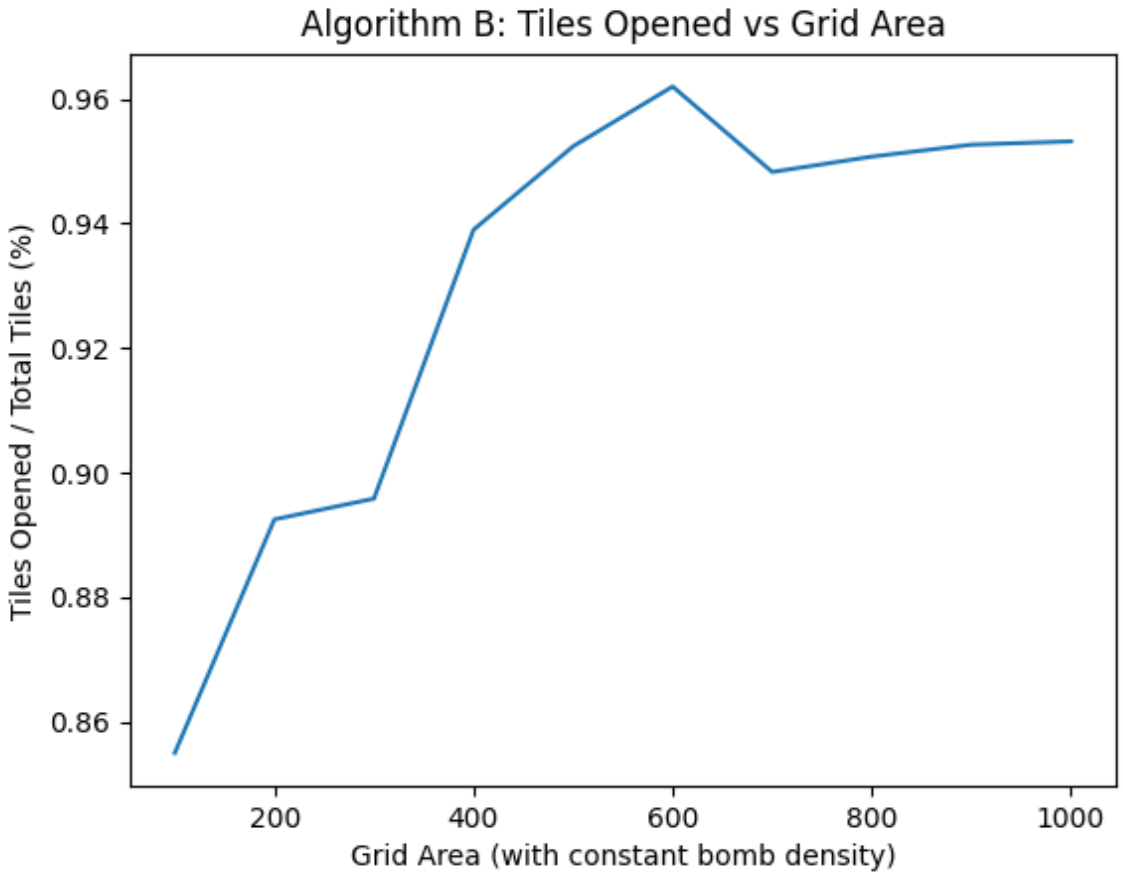
Below we have several plots depicting the performance of our algorithm. Important to note is that at least 5 trials were conducted for each individual data point in the graph. All plots were done with extensive testing and plotted in matplotlib.



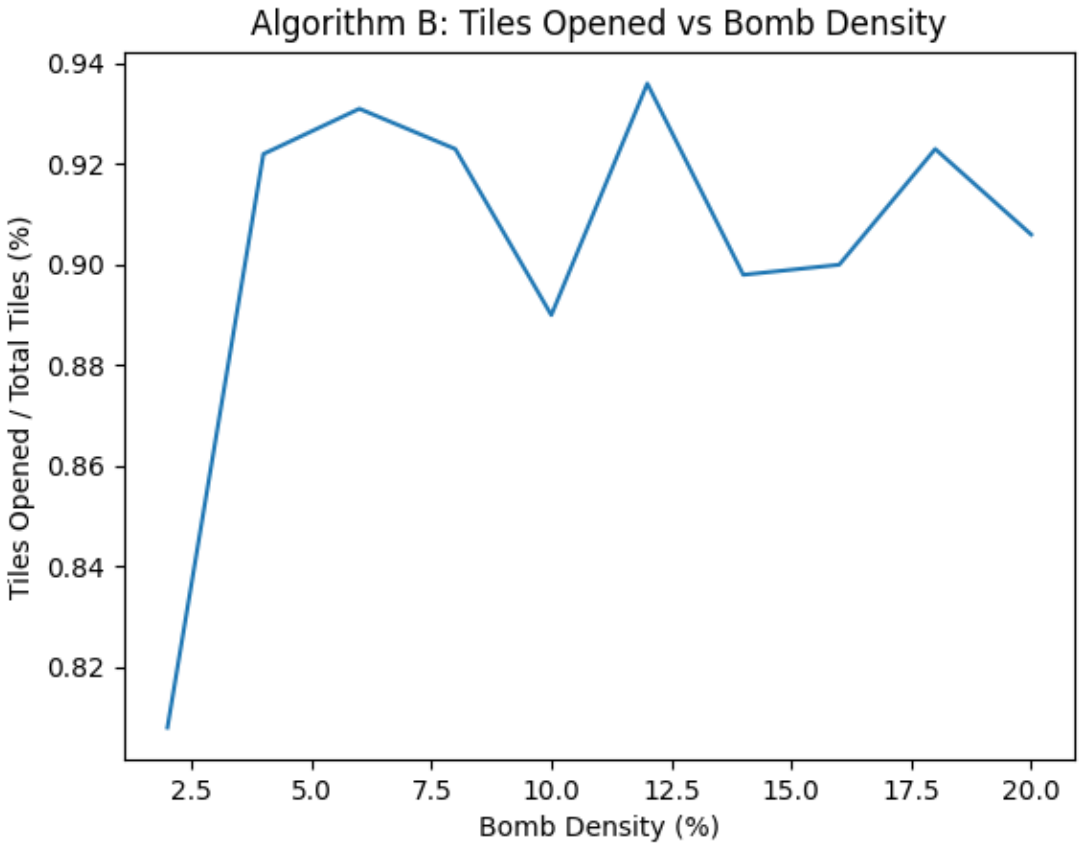
Note: It is unsurprising that there is a strong positive (exponential) correlation with runtime and grid area; however, what was interesting is how stark the decrease in runtime was once the algorithm halted its row-reducing (at about grid area 300). This visualizes just how time intensive row reduction is within the algorithm.



Note: There is a loose positive correlation between runtime and bomb density. This makes sense, given the algorithm does not make excellent use of found/known bombs as inputs. This plot had a constant grid area of 300. This contrasts our grid area of 1000 in the algorithm A plots, but, as stated previously, the algorithm becomes highly inefficient for grid areas past 300.



Note: Here we can see there is a strong (yet slight volatile) positive correlation between grid area and tiles opened. We see a strong uptick in tiles opened past grid areas of 300. This makes sense, given this is when we stop row reducing our constraint matrix.



Note: This algorithm performs extremely well on grids of abysmally low bomb density in terms of the tiles opened metric, and past that, there is an extremely loose positive correlation between the tiles opened matrix and bomb density. This plot had a constant grid area of 300. This contrasts our grid area of 1000 in the algorithm A plots, but, as stated previously, the algorithm becomes highly inefficient for grid areas past 300.