Algorithm 1:

Description: The algorithm will cycle through the entire 2d array and check the value of each square. If the value of the square is -1, then it will add that square into an unopenedSquares list. If the value of the square is nine, then the square is a bomb and it will get added to a bombsFoundSoFar list. If the value of the square is zero, then that means there are no bombs adjacent to that tile. So, it will then add all the adjacent squares into a not bombs list. We know these tiles can never be a bomb since they are adjacent to a zero square. If the value of the square is one to eight, then we will look at all the adjacent squares and check for a couple of things. First, we will check if the number of unopened adjacent squares equals the value of the center square. If they do equal each other, then we know for sure that all the unopened squares are bombs and they will be added to the bombsFoundSoFar list. We will also take into account if the center square has an opened bomb adjacent to it. So if we have a center square of three and two unopened squares and an opened bomb, then we know that the two unopened squares are also bombs. Finally, if the number of bombs in the bombsFoundSoFar list is equal to the number of bombs on the board then we end the game. If all the bombs have not been found, then we will pick a random square that is not in the bombsFoundSoFar list or the not- bombs list.

Justification of correctness: We can prove this algorithm is correct by considering all the cases it covers. Firstly, if we select a square with the value zero, this means there are zero bombs in the eight adjacent squares. So it is safe to add these eight squares into a list and never open them. Next, we will consider all the cases when we open a square that is one to eight. This means that there is at least one bomb in the eight adjacent squares. To figure out which adjacent squares are bombs, we will cycle through all the adjacent squares and count the number of opened bombs and the number of unopened squares. If this number does not equal the number of the center square, then we cannot determine where any bombs are. If the number does equal the number of the center square, then all the unopened squares must equal bombs. To show this, we can imagine a square with the value three. If it has five opened adjacent squares and none of them are bombs, and there must be three bombs adjacent to the center square, then we know for sure that all the unopened squares are bombs. Now imagine there are six opened adjacent squares and one of them is a bomb. The counter will still be equal to the center square because the number of bombs plus the unopened squares will equal three. So, we know the unopened squares are bombs.

Runtime analysis: Every time we uncover a square, it requires us to loop through all n squares. In the worst case, when we have a cluster of bombs, it could uncover these bombs last since it selects squares randomly. If there are no zero squares, then it will not add any tiles to the notBombs list. If both of these cases are present, then it is possible for the algorithm to run through all n squares n times. This means the worst case runtime would be O(n^2) where n is the number of squares.

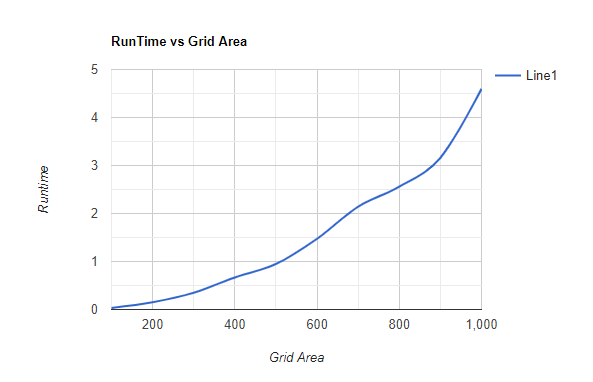
Algorithm 2:

Our second algorithm approaches this problem by always digging around the highest number showing on the board that hasn’t already been satisfied. So, starting with the safe square as the center tile, it would mine a random tile from its surrounding grid until the center tile is satisfied, meaning either the number of bombs in its surrounding grid is equal to its number or the number of bombs in its surrounding grid + the unopened squares in its surrounding grid is equal to its number. If we encounter another number while mining around our center tile, we do an immediate check on its surrounding to see if it is satisfied, and, if it is not, we add it to a dictionary of unsatisfied numbers to keep track of all the numbers showing on the board. If the (unsatisfied) number we encounter is higher than our current center tile, we abandon our current center (keeping it in the list of unsatisfied numbers), and continue mining around the higher number instead. When we encounter 8s or 0s, we mark all the surrounding tiles as bombs or not-bombs respectively. Since it’s not guaranteed that we’ll always have an unsatisfied number showing on the board, we may sometimes have to randomly choose an unopened tile to be our new center. If we find ourselves with an 8, 0, or a bomb as our center, we update our list of bombs or the surrounding tiles accordingly, and then randomly choose another unopened tile to be our center.

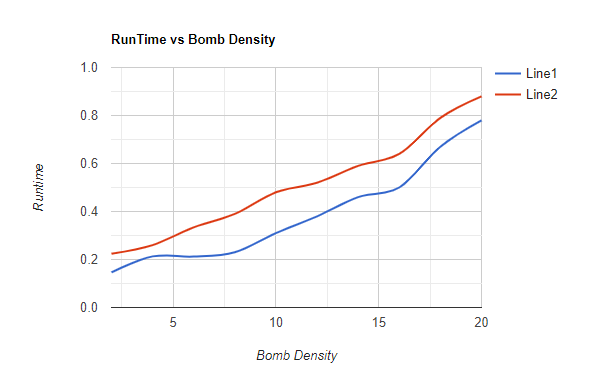
This algorithm will minimize the number of tiles we have to dig in a few ways. First, by digging around the highest tile showing on the board, we are maximizing our likelihood of finding bombs when we are forced to dig, which means we get to add them to our list (bringing us closer to finding all the bombs) and we’ll more quickly satisfy the numbers on the board when we encounter them. We are also minimizing our digs by not digging around 0s or 8s, since we know whether the surrounding tiles of these numbers are bombs. And, since we check whether a number is satisfied immediately when we encounter it, in some cases we may be able to make inferences with that information that help us satisfy our current center.

This algorithm first requires us to find all the unopened tiles in a grid, which can be done in O(n\*m) (where m is the number of rows and n is the number of columns). Then, we only have to dig each tile at most once. However, we have to find the surrounding grid of tiles that may be overlapping with grids we’ve already checked. If we say that, as an upper bound, we take each tile as a center at some point. The corners only have 3 surrounding tiles, the edges have 5, and the rest have 8. So we have 3\*4+5\*(2(n-2)+2(m-2))+8((n-2)(m-2)) which is on the order of O(n\*m). So, that’s the order of our worst case runtime.

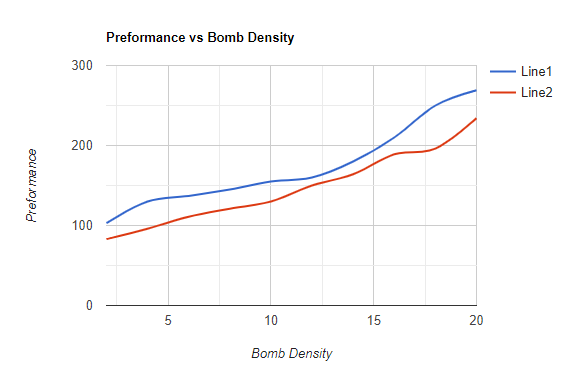
Graphs:



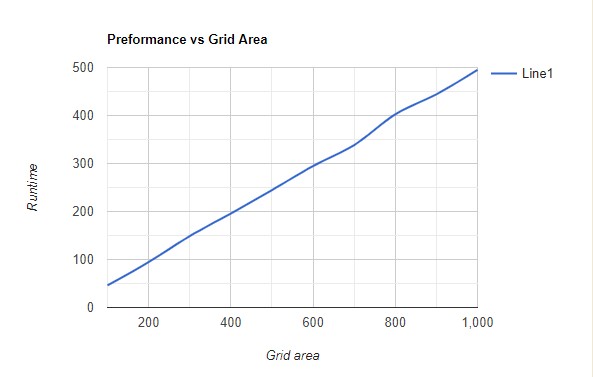
Data got from given json files. No data for algorithm 2 due to bug

’

Data gotten from given json files



Data gotten from given json files.



Data got from given json files. No data for algorithm 2 due to bug.