Alexander Liu, Aniruddha Murali, Filipp Zakharchenko
Professor Constantine Dovrolis
CS 3510
May 2nd, 2021

Algorithm A

The basic ideas for this algorithm include using the “obvious” deterministic clues to reveal as many (a) bombs and (b) safe squares as possible without actually opening them. The hypothesis is that this approach would lead to an optimized number of squares that we need to open before we can submit the bomb list.

One of the ideas for this algorithm was taken from “Algorithmic Approaches to Playing Minesweeper” [1], and specifically the Double Set Single Point approach described there. Their version of the algorithm maintains two sets, S and Q, where S is the “safe” squares and Q is a queue to determine the squares to return to later, when more information is available.

The algorithm A takes a similar approach, but modifies it to better match the optimization requirements. It maintains 4 sets:

- unopenedSquares (same as starter code) – maintains a set of unopened squares, represented with tuples of x and y coordinates.
- bombsFoundSoFar (same as starter code) - maintains a list of bombs. We add to it in 2 cases: when we uncover a bomb by the process of digging, or when we are certain a square includes a bomb (more on this later).
- questionableQueue – maintains a set of squares for which we cannot make any deduction, because there is not enough information available. We return to these squares as more information becomes available and try to not dig them if we don’t have to.
- probedQueue – maintains a set of squares that have been “probed” using the system of digs.

At every call to performAI, there are 2 possibilities: either bombsFoundSoFar contains all the bombs (exit the algorithm) or it does not, in which case we perform a step of the algorithm. Every step involves the following:

- We process information from the previously opened square: it may be a bomb or it may have a "0", in which case we can make a deduction about the square itself and the surrounding squares. Otherwise, we just add it to be processed later when we have more information about the surrounding squares.
- We process the "questionable" squares without opening to see if we have enough information to infer anything. There are 2 deterministic strategies: either we know all

[1] Becerra, David J. 2015. Algorithmic Approaches to Playing Minesweeper. Bachelor's thesis, Harvard College.

neighbors are mines, or we know all neighbors are clear. Otherwise, keep the square as questionable, we might know more later.

- If we've processed everything and no new information could be inferred, we end the turn by digging up a new square. For this, there is a probeIfNeeded subroutine. It only calls open_square_format if the size of probedQueue is 0. It chooses the square to open randomly among only the unopened squares.

Another subroutine that the algorithm depends on is unmarkedNeighbors. It's an O(1) method that checks how many of the neighbors for a particular square either (a) have not been opened, (b) are known to be bombs, either by digging or inference. It is used 3 times within one iteration of the algorithm.

The correctness follows from the usage of purely deterministic strategies. It is true that if a square contains a 9, it is a bomb, per the project specification. It is also true that if the square contains a 0, it is safe and all the neighbors are also safe, which we can determine without having to open them. By combining the above approaches, we keep the correctness of the random approach, while achieving the required optimization of minimizing the total number of digs.

In the worst case, performAI will be called n times, where n is the number of squares. This would happen in the rare case of the algorithm not being able to make any inferences. Each iteration of the algorithm calls probeIfNeeded (O(1)) twice and iterates through all elements in questionableQueue. Assuming that set operations are O(1), we can guarantee the bottleneck complexity of O(n) for one iteration of the algorithm. Thus, the algorithm's runtime is upper-bounded by O(n^2).

Alexander Liu, Aniruddha Murali, Filipp Zakharchenko
Professor Constantine Dovrolis
CS 3510
May 2nd, 2021

Algorithm B

The idea for this algorithm was that given an opened square on the grid with at least one unopened square surrounding it, we want to calculate the probability that a surrounding unopened square is a bomb.

The algorithm maintains three lists: self.queue, self.dontOpen, and self.bombs. self.queue maintains a current list of squares that we know are bombs but haven't opened yet. This is helpful because for each iteration of performAI(), we only submit one square, so in the next iteration, we first check to see if there is anything in the queue to open. self.dontOpen keeps track of squares that we know are safe and therefore should not be opened. We want to find all the bombs with the least possible number of digs, so if we know a square is not a bomb, we should not dig it. self.bombs keeps track of all the bombs found throughout the algorithm.

We start off with two cases:

- The number of bombs found equals the total number of bombs on the board, in which case we're done with the algorithm. (len(self.bombs) == self.numBombs)
- We still have bombs to find, so we perform another iteration of the algorithm.

If we have bombs left to find, then we iterate over the set of opened squares and call self.calculateProb() where we treat each square as the "center square". self.calculateProb() takes in the center square, the set of squares surrounding the center square, and the current board state. It uses the following formula to calculate the probability that that a surrounding unopened square is a bomb:

$$P(x) = \frac{center\ square\ number - current\ number\ of\ surrounding\ bombs}{number\ of\ surrounding\ unopened\ squares - number\ of\ surrounding\ squares\ in\ self.dontOpen}$$

The numerator tells us the number of surrounding bombs left to find while the denominator tells us the number of unrevealed squares that these bombs could be assigned to. In the denominator, we subtract the number of surrounding squares in self.dontOpen because we know those squares cannot be bombs.

For each time we call performAI(), we maintain 3 variables maxProb, maxSquare, and maxUnrevealed - these represent the probability, square location, and surrounding squares of the center square that results in the highest calculated probability. Once we calculate the probability given a center square, we have three cases for what to do with that information:

- If the probability was 1, that means we know that all the unopened squares surrounding the center square must be a bomb. We open one of the squares and if there are other

unopened squares, we add them to self.queue. The following calls of performAI() will open those squares.
- If the probability is 0, we know that those unopened squares aren't bombs, so we add them to self.dontOpen.
- If the probability is greater than maxProb, update maxProb, maxSquare, and maxUnrevealed.

If none of the opened squares with at least one unopened square surrounding it resulted in a probability of 1, then we look at the surrounding squares (from maxUnrevealed) of maxSquare and check if any of them are not in self.dontOpen. If we find a square that is not in self.dontOpen, we open that square. Otherwise, all squares from the maxUnrevealed list are squares we shouldn't open, so the last resort is to open a random square from the set of unopened squares that are not in self.dontOpen.

The correctness of the algorithm follows from the fact that we know that a square with a 9 is a bomb and that any time we add a square to self.dontOpen, we know that the square is not a bomb because if it were a bomb, it would not follow the rules of Minesweeper. The randomness in the algorithm doesn't affect the outcome of the list we return that contains all discovered bombs, so it doesn't affect the correctness of the algorithm.

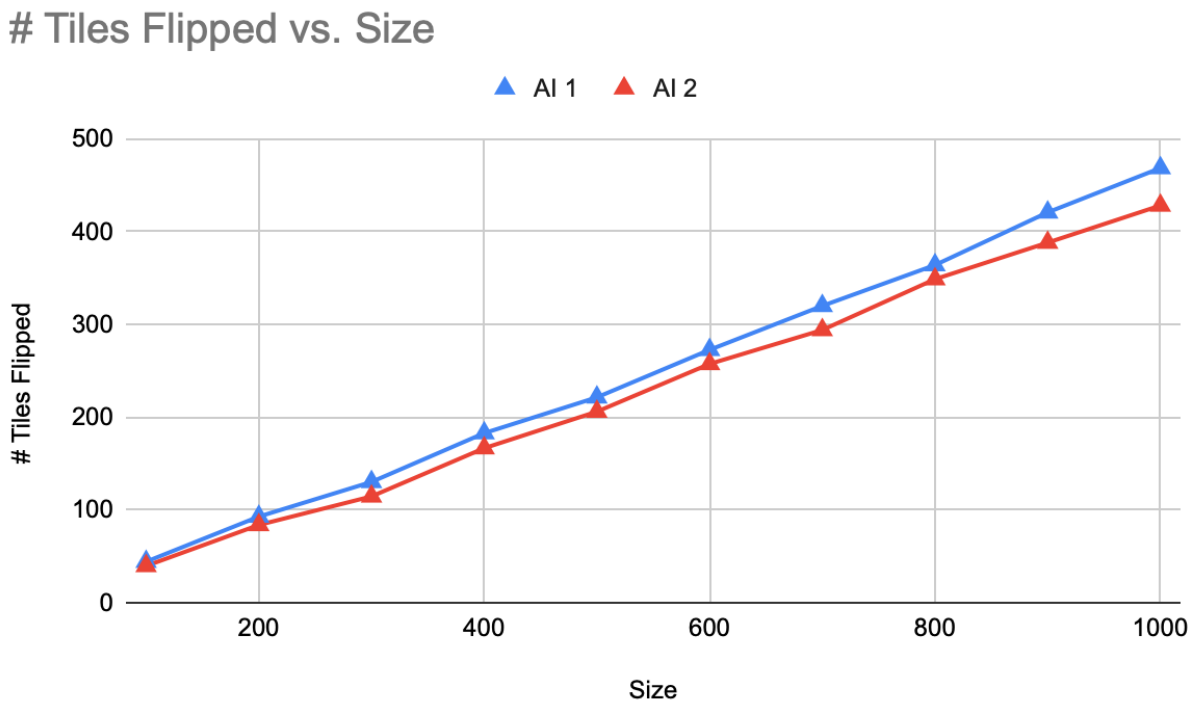
In the worst case, we have to iterate through all opened squares, calculate the probability for each square, and check if any squares from maxUnrevealed is not in self.dontOpen, only to open a random square from the set of unopened squares that are not in self.dontOpen. We first need to find all opened and unopened squares and add them to their respective lists - we iterate over each row and column, so this is O(n^2). Next, we iterate over all opened squares, calculate the probability for each one, and then do a constant number of operations. When calculating probability, we iterate through the set of surrounding squares and do a constant number of operations, so calculating probability is O(n). Therefore, iterating over all opened squares and performing the operations in the for loop is O(n^2). Then, we iterate through maxUnrevealed to see if any unopened squares surrounding maxSquare is in self.dontOpen - this is O(n). Finally, we need to pick a random unopened square. This involves picking a random number and then accessing that index in the list. Both of these are constant operations, so picking a random unopened square is O(1). So, each iteration of performAI() is O(n^2) + O(n^2) + O(n) + O(1) = O(n^2), and at worst, we call performAI() over n squares, so the algorithm's total runtime is O(n^2 * n) = O(n^3).

Alexander Liu, Aniruddha Murali, Filipp Zakharchenko
Professor Constantine Dovrolis
CS 3510
May 2nd, 2021

Comparison of the two algorithms

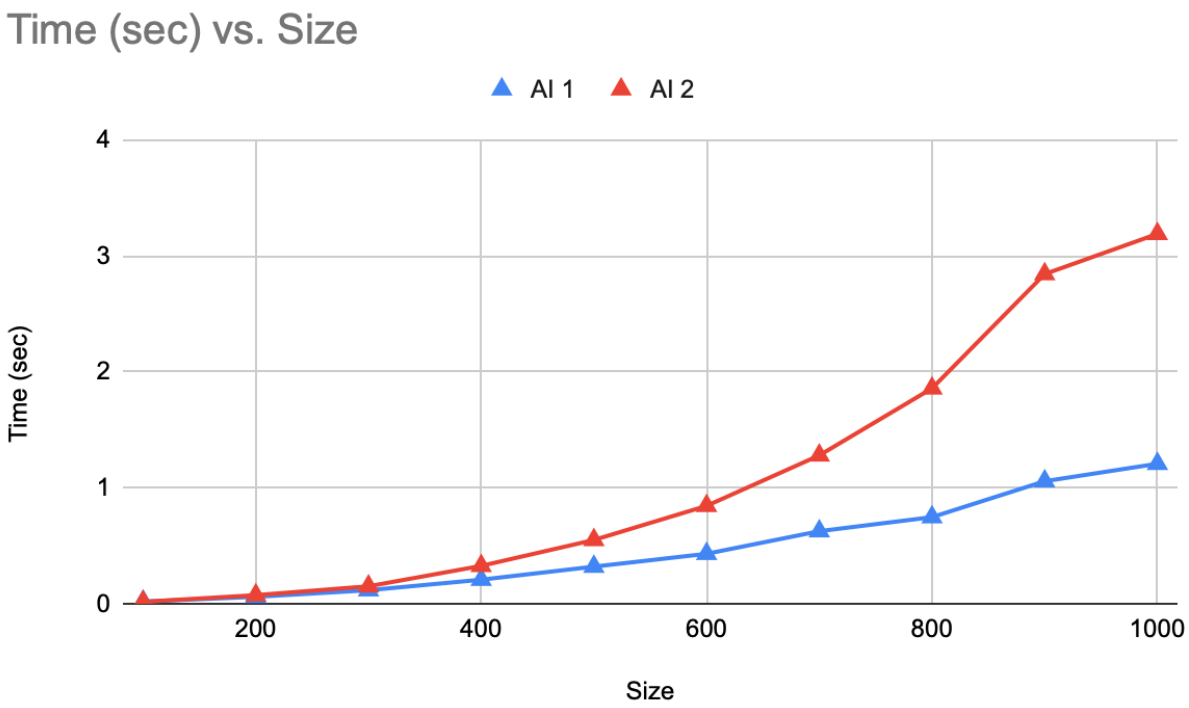
In the following graphs, we will compare the two algorithms described previously. The purpose of these graphs is to illustrate practical differences between the two algorithms.

Note: in this section, Algorithm A is renamed to AI 1, and Algorithm B is renamed to AI 2.



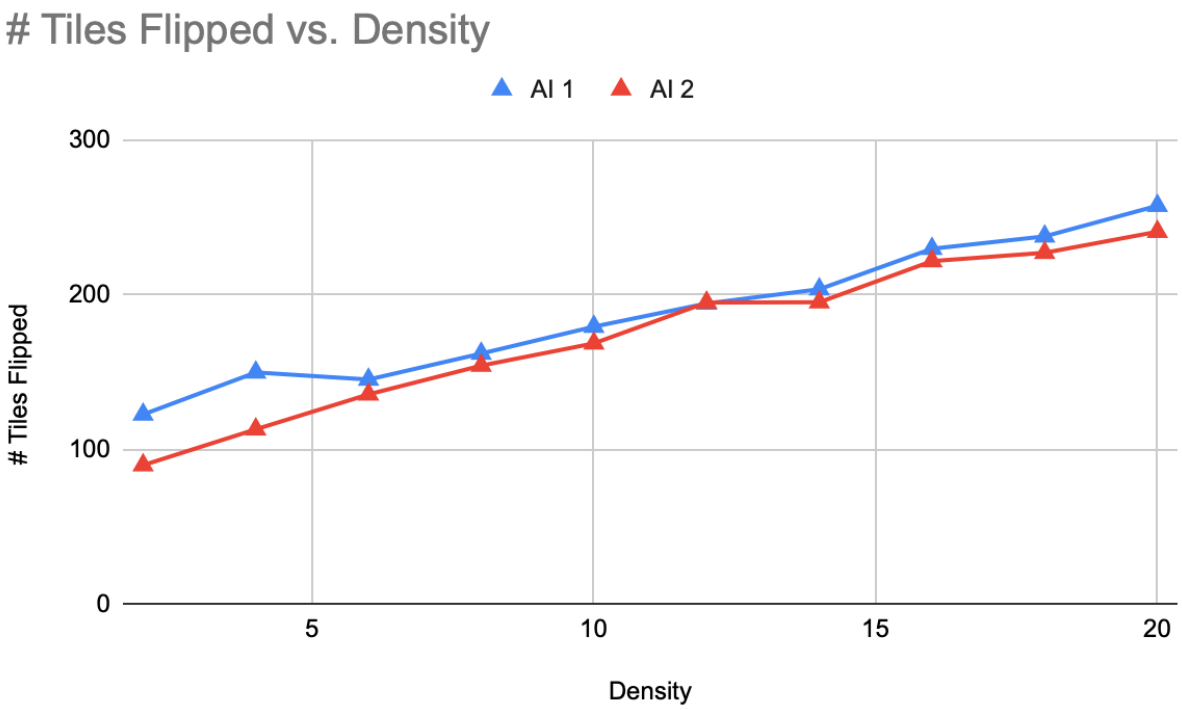
The first comparison is in the number of tiles that need to be flipped to solve the minesweeper with respect to size. For this visualization, all boards had 10 bombs. As we can see, AI 1 requires slightly more tiles to be flipped in order to solve the minesweeper board.

Alexander Liu, Aniruddha Murali, Filipp Zakharchenko
Professor Constantine Dovrolis
CS 3510
May 2nd, 2021



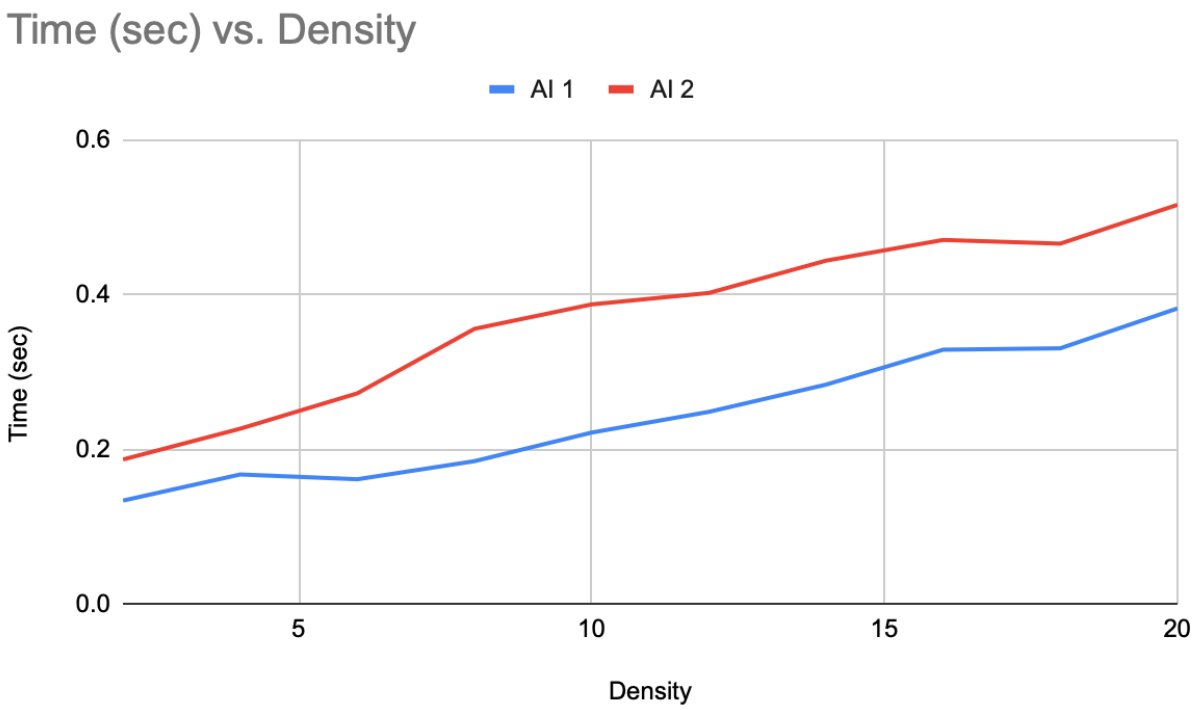
The second graph plots the total time needed to complete minesweeper with respect to board size. For this visualization, all boards had 10 bombs. Notice how the gap between AI 1 and AI 2 increases with board size. This matches well with our algorithm runtime analysis, as we solved that AI 1 was O(n^2) and AI 2 was O(n^3).

Alexander Liu, Aniruddha Murali, Filipp Zakharchenko
Professor Constantine Dovrolis
CS 3510
May 2nd, 2021



The next graph explores the relationship between the number of tiles needed to be revealed with respect to the density of bombs, i.e. the percentage of tiles that are bombs. For this demonstration, all boards had dimensions 20 tiles x 20 tiles to keep board size constant. The graph shows that AI 1 requires slightly more tiles than AI 2 does to solve minesweeper puzzles with higher bomb density.

Alexander Liu, Aniruddha Murali, Filipp Zakharchenko
Professor Constantine Dovrolis
CS 3510
May 2nd, 2021



The last graph plots time needed for minesweepers with increasing bomb density. For this demonstration, all boards had dimensions 20 tiles x 20 tiles to keep board size constant. As shown here, AI 1 was faster compared to AI 2 when solving boards of varying bomb density.

We can conclude from these visualizations that although AI 2 needed to dig fewer tiles than AI 1 on all tested dimensions, AI 1 was faster than AI 2 with respect to the same dimensions. This can be attributed to multiple reasons, including algorithm differences and implementation quality.