CS 3510 Final Project

Submission Date: Sunday, 2nd May 2021

Group members:

- Ali Kazmi,
  - Email: akazmi30@gatech.edu
- Adrian Nusalim
  - Email: anusalim3@gatech.edu
- Matthew Fishel
  - Email: mfishel7@gatech.edu

Content:

- "Safe Square Method" algorithm.
- "Bomb Blossom Method" algorithm.
- Experimental plots and comparison between (1) and (2).

Submitted Files (in addition to what was provided):

- ALGO1_safe_squares_init_query.py
- ALGO1_safe_squares_py
- ALGO2_bomb_blossom.py
- average.py
- Scaling_plot.ipynb
- Several bash scripts (see readme for full list and descriptions)

Github repository:

To structure this report, we will look at each of our two algorithms. All code (and commit history) can be found at https://github.gatech.edu/akazmi30/minesweeper (a private git repo).

Algorithm 1: Safe squares method

To run on a json input:
python ALGO1_safe_squares.py <boardfilename>

NOTE: Run ALGO1_safe_squares.py from the command line with <board_file_name> parameter. The file should be a json.

Description:

This algorithm focuses on solving the complement problem to finding all the bombs. It attempts to find all safe squares without digging on a bomb. The algorithm uses a priority queue and maintains a set of unexplored squares on the board as well as a set of found bombs. This algorithm attempts to play a faithful game of Minesweeper, wherein stepping on a bomb is to be avoided.

Correctness and runtime:

When a non-bomb square is dug i.e. explored, it is added to the priority queue with a priority value equal to its square value. Lower square values receive a higher priority, so a '0' is the highest priority and an '8' is the lowest. If a bomb is accidentally dug, it is not added to the queue as the square value '9' gives no information about the surrounding squares.
While the queue is not empty, the highest priority element is popped from the queue and evaluated by its square value compared to its number of found bomb neighbors. If the number of found bomb neighbors and square value is equal, all unexplored neighbors are safe. The unexplored neighbors are "dug" and added to the priority queue. Adding to the priority queue is O(log n) for each element added to the queue.
If the number of unexplored neighbors is *equal* to the square value minus the number of neighbor bombs, all unexplored neighbors are bombs and are added to the found bomb set. These squares are specifically not dug or added to the priority queue.
If a square has unexposed bomb neighbors *and* unexposed non-bomb neighbors, the algorithm must guess for its next square to dig. It uses the ratio of unexplored bomb neighbors to unexplored neighbors in comparison to the ratio of undiscovered bombs to unexplored squares on the board. If it is less likely to step on a bomb in the immediate neighbors of the square, the algorithm picks randomly among neighbors for its next square. Otherwise, the algorithm picks randomly from the board.
Evaluating the algorithm's next steps based on the top of the priority queue is O(1). Adding a square to the priority queue is O(log n). Elements are added and removed from the priority queue on the order of n times. Therefore the algorithm is **n*[O(1) + O(log n)]** or **O(n log n)**.

The algorithm resembles a greedy algorithm with some guessing guided by a heuristic. Board configuration plays a role in whether bombs are "dug", which this algorithm seeks to avoid. Our experiments showed that about 4 of 5 bombs could be located without digging their square. Finding the complement set typically requires exploring *most* of the board, as is normal in a game of Minesweeper. Our experimental results bear this out. However, the algorithm is not trivial. We show no indication that our algorithm needs to explore the entire board to locate bombs. The fact that most bombs are not "stepped on" further supports this conclusion.

Note on the Safe Squares approach and our code

In a normal Minesweeper, a "safe" square is provided to start the game. Our set of standard boards gives a starter square with a non-zero value. This algorithm does not have special handling for this wrinkle to the problem and could conceivably perform better with a 0-value start square. One option for an improvement to our algorithm and code would be to mine some portion of the board randomly. We experimented with this option. ALGO1_safe_squares_init_query.py is included with our files. We found that this approach primarily improved on our standard Safe Squares algorithm when all bombs were found in the random query. Otherwise, results seemed to be indistinguishable. Finding bombs by guessing is not in the spirit of solving the complement problem. We include the file as a curiosity and evidence of work. Our provided analysis is based solely on the script which does not randomly query the board: ALGO1_safe_squares.py

Algorithm 2: Bomb blossom

To run on a json input:
ALGO2_bomb_blossom.py <board_file_name>

NOTE: Run ALGO2_bomb_blossom.py from the command line with <board_file_name> parameter

(the file should be a json input)

It should look like this

“Python ALGO2_bomb_blossom.py
standard_boards/varied_density/20x_20y_20d_4.json”

Description:

Our second algorithm makes no effort to avoid bombs and focuses entirely on finding them quickly. For this algorithm, we again deploy a priority queue. However, in this case, a higher square value equates to a higher priority.

When a square is dug, it is added to the priority queue with the priority noted. While the queue is not empty and the size of the found bomb set is less than the number of total bombs, the top of the queue--which has the greatest number of bomb neighbors of all explored squares--is popped, and the unexplored neighbors of this square are explored. If a neighbor is a bomb, it is added to the found bomb set. If a neighbor is not a bomb, it is added to the priority queue.

Correctness:

We determined that there is a rare circumstance where a guess is necessary to proceed. If the last element of the priority queue has *only* bomb neighbors, no further squares are available to add from this square. At this point, the algorithm must randomly select a square from the unexplored squares to add to the priority queue. In this event, the algorithm will resume preferring the next square to explore with the highest number of bomb neighbors. In this way, the algorithm is sure to proceed until all bombs are located.

Runtime analysis: Our 2nd algorithm also takes a greedy approach to finding bombs and is much less reliant on guessing than our 1st. However, no effort is made to avoid bombs. The runtime of our 2nd algorithm resembles the first. Popping a queue element and evaluating neighbors is O(1). Adding squares to the priority queue is O(log n). Each operation occurs on the order of n times. Therefore our 2nd algorithm is **n[O(log n) + O(1)]** or **O(n log n)**.

Running our code

To run our code for a single jason input, in the terminal type
python <algo name><files directory><result type>
<algo name> could be ALGO2_bomb_blossom.py or ALGO1_safe_squares.py
<result type> could be 0 or 1. When the result type is 0, it would return in the style of "(squares number) (dig count) (num_bombs) (delta)".

To run all the algorithm with varied density or varied size all at once using the bash file:

Running the bash file would require modification of the "FILES" on the bash code (use a path you have locally with the standard boards folder), in which could be run afterwards using
bash runSizeTrials <algo name>
Or
bash runDensityTrials <algo name>

We generated these graphs as follows:

1. Generate the output of a single game as text using the algorithms above
2. Writing bash scripts (runDensityTrials.bash and runSizeTrials.bash) that run the given algorithm (passed as stdin) many times (one for each board in the given standard boards for the respective trial, either varying by density or size). runSizeTrials *generates varied_size_output_algo1.txt* and *varied_size_output_algo2.txt*. Similarly, runDensityTrials generates *varied_density_output_algo1.txt* and *varied_density_output_algo2.txt*.
3. Afterwards, we created a python file named average.py. Inside that python file, we are reading through every line of the txt file from where we generated output (which include squares number, digcounts, bomb counts, and time) all the boards. This prints the output (basically a condensed version of the text file we passed in), so that our bash script can pipe it into our new averaged files later.
4. Afterwards, we create a bash script in which we would run these python files for the varied density and varied size, meaning that there are 4 commands since there are 2 algorithms. Note generateAveragedFiles.bash depends on us already having runSizeTrials.bash and runDensityTrials.bash.
5. Once we have the averaged txt file, it gives us information on num_squares, dig_count, bomb_count, and runtime (in milliseconds). We created an ipynb (jupyter notebook) to parse this text file with pandas, thus giving us a dataframe to work within python. After we have a dataframe, we do the plotting of the graphs with matplotlib.
    a. (minor note: if you attempt to run this, watch your paths--- the bash scripts have some of our file paths hardcoded. Also, the jupyter notebook expects the files to be in the right folder and already generated- if they

aren't, the graphs will not be generated and errors of the missing file will be thrown)

b. After we have these text

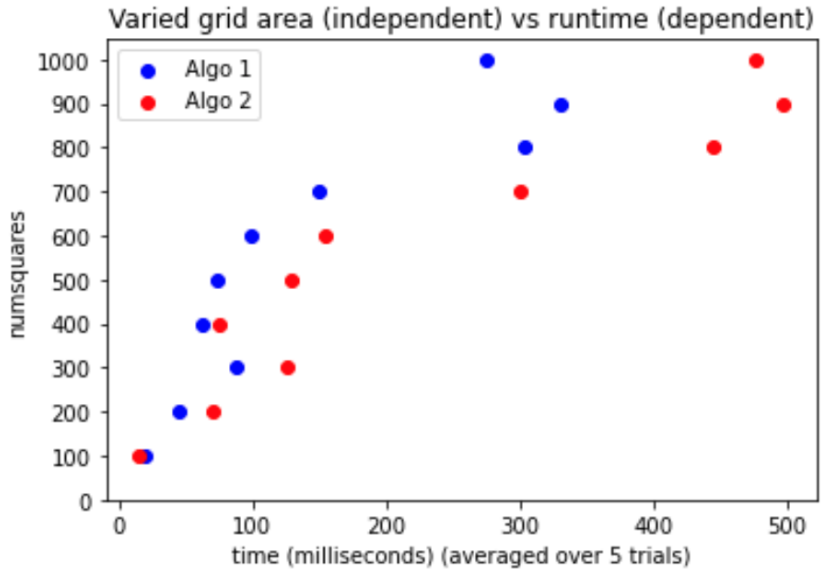
6. Once our data was in a pandas dataframe (can be tested using df.head()), we used matplotlib pyplot to get the graphs below!

Example workflow:

- Bash generateAveragedFiles.bash (the only command we need to enter)
  - (inside the bash)
  - [remove all the old files]
  - python average.py varied_density_output_algo1.txt >> averaged_density_output_algo1.txt
  - then it does that for varied density and varied size boards on algo 1 and algo 2

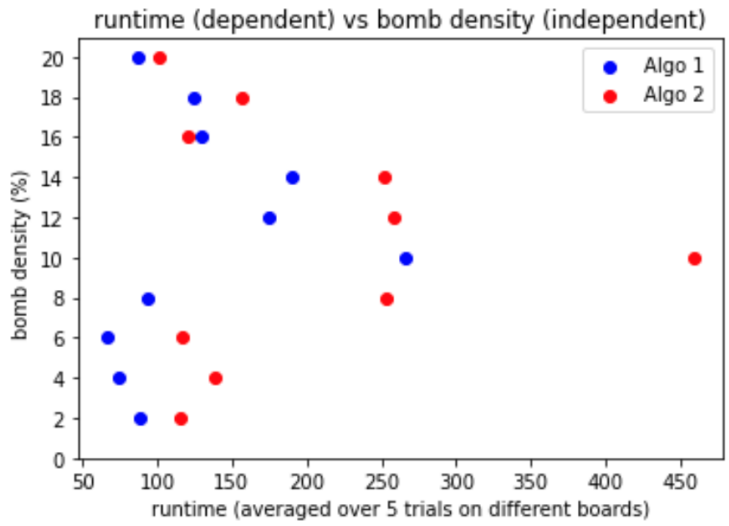
Experimental plots with two curves

Runtime vs grid area (from 100 to 1000 squares in steps of 100)



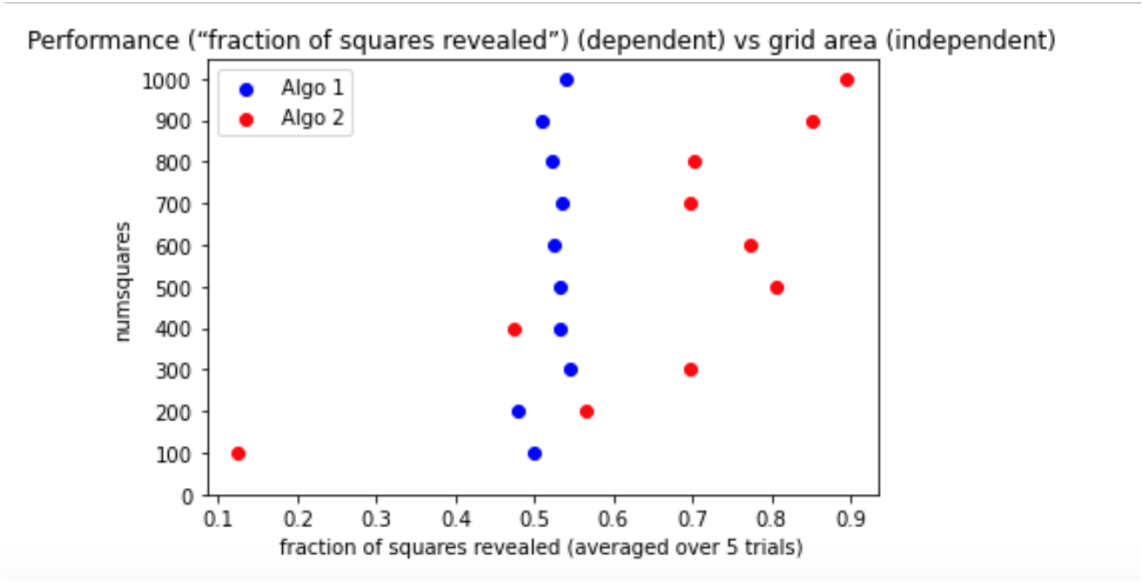
Our algorithm 1 and 2 seems to show the same behavior, in which more time would be needed to compute as the number of squares increases. This also verifies the algorithm runtime we mentioned above. As our algorithm 2 focuses on finding the bombs, we need more time in order to find that as we need to iterate (dig) through the boards. On the other hand, our algorithm 1 focuses on finding the safe squares. The same thing happens, as there are more iteration(dig) needed as the board size increases, it would also require more time as the grid area gets bigger. When comparing algorithm 1 and 2, it seems that algo 1 (safe squares algorithm) is faster than the bomb blossom algorithm in some cases where avoiding guessing makes it less optimal.

Runtime vs bomb density (from 2% to 20% in steps of 2%)



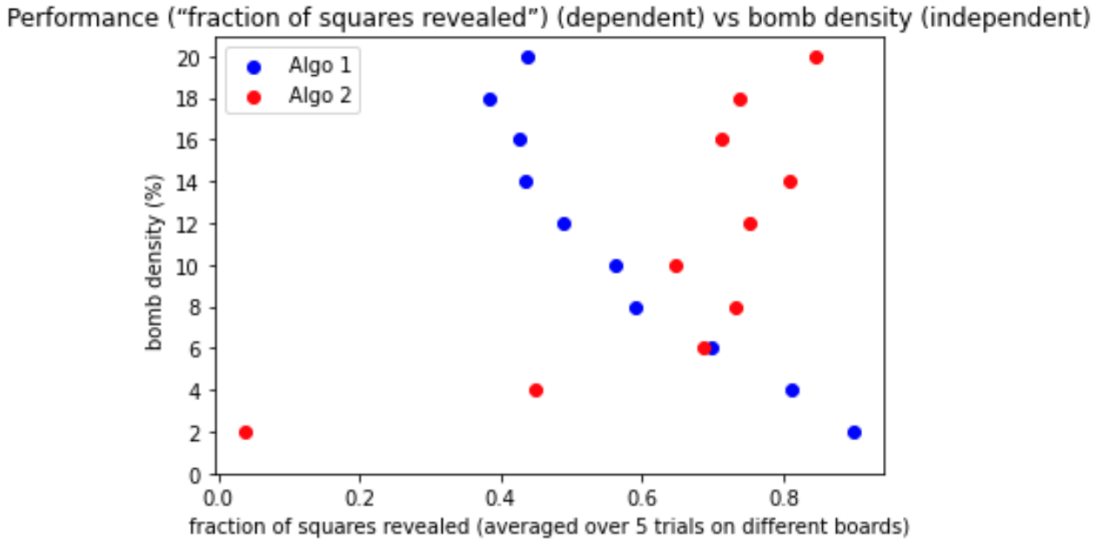
When comparing the runtime versus the bomb density, it looks like our algorithm 1 and 2 have similar behaviors. Algorithm 2 seems to perform slower than algorithm 1. It seems that the algorithm gets slower in the middle number of the bomb density. Somehow, the bomb blossom algorithm takes more time as the bomb density reaches the middle point. There is an outlier in the runtime for algo2 at bomb density 10, which we expect was due to outside factors like our computer's ram running low due to other programs (microsoft teams) and not the algorithm itself.

Performance (“fraction of squares revealed”) vs grid area



By definition, the fraction of squares revealed is the number of squares we dig/(the total number of squares). Generally, the performance of the fraction of squares that needs to be revealed gets bigger as the grid area gets bigger. However, the second algorithm seems to be somehow inconsistent because there is some random guessing involved. On the other hand, the first algorithm seems to be consistent because the reason behind this algorithm is to minimize guessing. In conclusion, the safe square algorithm is expected to be consistent while the bomb blossom algorithm will be more inconsistent as they have different nature on when they need to guess.

Performance (“fraction of squares revealed”) vs bomb density



The bomb density vs fraction for squares works differently on the algorithm 1 and 2. For algorithm 1, as there are more bombs contained on the grid, there number of squares that need to be revealed gets less. This is because we are trying to find the safe squares while minimizing the number of bombs that we actually need to dig. For algorithm 2, as there are more bombs contained on the grid, the number of squares that need to be revealed increases. This is because we are trying to iterate through all the bombs through the priority queue, and find the bomb as quickly as possible. As a result, as there are more bombs, it is our priority, and thus, more bombs means more squares to be revealed.