**Minesweeper Final AI Report**

**Team name Phantom Search**

**Member #1 (name/id) Timothy Lee 69451957**  **Member #2 (name/id or N/A) James Wilson 93265297**

**I. Minimal AI**

**I.A. Briefly describe your Minimal AI algorithm:**

For the minimal AI the logic engine was the only part needed despite the fact that most of the algorithm was implemented at the time. This original implementation, however, had run at times in excess of 3 hours due to a bug preventing the program from running on expert level boards.

The AI is divided between helper functions, logic engine, and the brute force algorithm. Helper functions perform a minimum of the logical computations, however there was a bounding error in the first implementation that swapped columns and rows which works fine for square boards but broke on the expert level rectangular boards. The logic engine searches the board for obvious safe moves and mines. Once the logic engine has exhausted the current board state, the border is passed into the brute force algorithm for computation.

**Constructor**

The key part of the constructor is the limit which scales based on the number of mines on the board. This affects the calculations for endgame scenarios: the greater the limit, the more interior tiles are incorporated into the final calculation.

**Logic Engine**

All percepts are tracked and updated on a board that the AI maintains as a representation of the environment. Any time the percept equals zero all neighboring tiles are selected for uncovering. This is the first check that runs every time the *getAction()* function is called. If there are no safe moves based on the zero percept then the *flagAMine(*) function takes over. This function iterates over the entire board and for every tile with a hint greater than zero that has exactly the same number of open tiles next to it as the hint, those tiles get marked for the flagging action. The flag and safe moves are stored as sets in order to avoid duplicate moves. At this point all safe areas should be opened and all obvious mines are flagged. From here the next task is to find any tiles that are safe based on the neighboring hint and flag count. The *goForth()* function deduces safe moves or calls the brute force solver.

**Brute Force Algorithm**

The brute force algorithm functions as a backtracking search method with two helper functions, one to perform border optimization and another to generate valid solutions. The *bruteSolver()* method takes the data and computes either a guaranteed mine, safe tile, or selects the safest potential tile.

The first action is to determine if the board is in an endgame state. This is done by comparing the difference between border tiles and all remaining tiles. If the difference exceeds the limiting factor then optimization is turned on, causing board-segregation function to evaluate border tiles. Without optimization all remaining tiles are evaluated at once. The *borderSegregate()* function evaluates the border for adjacency. All border tiles are passed and then categorized based on adjacency. This function can provide major speed optimizations if there exists a condition with separate borders. (This is a relatively rare border state).

The borders are evaluated one at a time with a copy of the current board state. Solutions are returned to the solver function with the *solutionGeneratorRE()* method that recursively backtracks through the entire border. Each cycle the hints on the board are tested for overflow conditions, either too many mines or too many empty tiles per the adjacent hint. If the board has not exceeded the constraints then the function calls itself after toggling on or off a mine on the border. Once the entire border has been cycled and the solution conditions are valid it is saved as possible solution. The issue with this approach is that even though the function is O(2N) the operational cost of each call is O(N2). This problem has been addressed in the final AI.

Each solution represents the border as a sequence of Boolean values. By iterating through the border it is possible to evaluate the state that each solution has for that position on the border. If all of the solutions have the same state then that border position is *guaranteed* to be either a mine or safe-move. Once a guaranteed position is found the move is stored. The function finishes after all guaranteed tiles have been inserted into the proper set. If there are no guaranteed moves then the most probable safe tile is determined. On the off chance that no solution is generated, the blind guess function finds any position yet to be uncovered or flagged and moves there.

**I.B. Describe your Minimal AI algorithm’s performance:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Board Size** | **Sample Size** | **Score** | **Worlds Complete** |
| **8x8** | 1000 | 862 | 862 |
| **16x16** | 1000 | 1698 | 849 |
| **16x30** | 1000 | 0 | 0 |
| **Total Summary** | **3000** | **2560** | **1711** |
| **Mean** | **1280** | |  |
| **Std Deviation** | **591.1** | |  |
| **99% Interval** | **(1252.2, 1307.8)** | |  |

**II. Final AI**

**II.A. Briefly describe your Final AI algorithm, focusing mainly on the changes since Draft AI:**

There were two major changes made to the minimal AI going into the final AI: fixing the border evaluation for expert boards and optimizing the solution generator.

Expert level boards were causing segmentation faults and range check errors due to inconsistent column and row checks within the helper functions. This was a relatively quick fix that exemplified just how slow the algorithm was running on the more complex boards.

Initially the O(N2) computation within the O(2N) was initially considered to be less of an issue and attempts were made to limit the amount of recursion based on border size or mine count. However these attempts only resulted in worse accuracy and not much gain in speed. The next goal was to reduce the computations of each cycle by only checking relevant tiles and not the entire board. A complex approach was initially taken in order to compute a parallel set of tiles adjacent to the border to be checked for satisfiability however it was later realized that the border could simply be checked for adjacent hint tiles to achieve the same effect. Once this was properly implemented there was approximately a 66% performance gain while maintaining the same level of accuracy.

**II.B. Describe your Final AI algorithm’s performance:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Broad Size** | **Sample Size** | **Score** | **Worlds Complete** |
| **8x8** | 1000 | 903 | 903 |
| **16x16** | 1000 | 1714 | 857 |
| **16x30** | 1000 | 1272 | 424 |
| **Total Summary** | **3000** | **3889** | **2184** |
| **Mean** | **1296.3** | |  |
| **Std Deviation** | **406.0** | |  |
| **99% Interval** | **(1277.2, 1315.4)** | |  |

**III. In about 1/4 page of text or less, provide suggestions for improving this project.**

The most prominent issue with this algorithm is time complexity. The algorithm produces good results but in order to achieve this level of accuracy it is not clear how to reduce the run time. Parallel processing could be one method by which the solutions might be generated faster either by checking the entire border in one pass or by computing segregated borders concurrently. Artificially segregating the border was considered as a means of guaranteeing a run time however, this would severely impact the accuracy while still not providing a fool proof means of ensuring a runtime.