Solving Minesweeper Using Genetic Algorithm and Genetic Programming

JP Park jae.park@uga.edu
Joey Liou jsl31324@uga.edu
Ethan Bray ethan.bray@uga.edu
Ellison Pyon ep14244@uga.edu
Dominic Francesconi dominic.francesconi@uga.edu

December 15, 2021

Abstract

Minesweeper is a puzzle game that was bundled with early versions of windows intended to teach the users about how to interact with the interface. It has received a significant attention from the mathematicians and computer scientists, who developed a myriad of methods in an attempt to solve or approach the game. In this paper, we explored ways of utilizing different evolutionary algorithms, mainly genetic algorithm and genetic programming, to create a solver for the minesweeper game or problem related to it.

1 Introduction

1.1 Minesweeper

Minesweeper is a single-player puzzle game. The game is played on an rectangular board of squares, each of which contains a mine or a number. Initially, the contents of the squares are hidden from the user, and the goal of the player is to reveal all the squares that do not contain a mine by using the numbers as clues about the placement of the mines. The meaning of the numbers is simple: they indicate the number of mines in the adjacent squares. For instance, as shown in figure 1, the number 3 in the middle means that there are 3 mines placed among the 8 squares around it. Since 3 of them are already revealed to be safe, the 3 mines have to be in the 5 unrevealed squares.

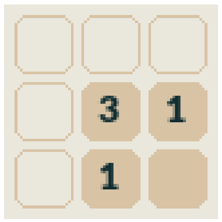


Figure 1: A partially revealed minesweeper board

The rules of the game is not difficult, which explains its popularity. Surprisingly, however, it has been proven by Richard Kaye in 2000 that the minesweeper is an NP-complete problem reducible to the famous Boolean Satisfiability problem (SAT) [1]. This specific version of the minesweeper problem is called the Minesweeper Consistency Problem (MCP). As opposed to the regular game of minesweeper, MCP problem is a problem of determining whether there exists a placement of the mines on a partially revealed board such that the numbers and the mines are consistent. In this paper, we will be focusing on a different problem called the Minesweeper Inference Problem (MIP). As we will see, this problem is more relevant to us than the MCP.

1.2 Minesweeper Inference Problem

The Minesweeper Inference Problem has two parts: a decision problem and a search problem [3]. The decision problem asks whether or not there exists a hidden square whose content can be inferred by the information that is available to the player at that moment. The search problem involves finding the squares whose content can be inferred. The MIP is not an NP-complete problem; however, it is shown to be among the class of co-NP-complete problems [3].

1.3 Minesweeper Solver

There have been many automated solvers created to help with inferring the contents of the hidden squares. Examples include the use of probabilistic inference method [4], search tree with heuristics similar to the Monte-Carlo search tree for chess [6], and cellular automation [5]. Recently, there have been a trend to consider the Minesweeper Inference Problem as a Constraint Satisfaction Problem (CSP). For example, Yimin Tang and his co-researchers used logic inference, CSP, and sampling to create a minesweeper solver [8]. They, however, do not go into details about how they approached the CSP aspect of the problem. From their description, it seems that they randomly generated solutions and sampled the best ones from them. In our paper, we attempt to utilize the power of genetic algorithm to deal with the CSP aspect of the minesweeper problem.

1.4 Genetic Algorithm

Genetic algorithm is an evolutionary algorithm that borrows the concept of natural selection in biology. It starts with a random solutions, but use fitness evaluation to select the best ones and create new solutions that inherit the "good" aspects of the old solutions. They are usually used for CSP problem since the solutions generated are usually complete, as opposed to partial solutions. We also wanted to exploit the general nature of the evolutionary algorithm by creating a minesweeper solver without a domain-specific knowledge like logic inference that involve the use of matrix.

1.5 Genetic Programing

With any form of artificial intelligence, decision making is based on a set of predefined principles. More advanced system of artificial intelligence may be able to establish a form of memory to learn and adapt to whatever challenges the system is defined to deal with. With our initial goals under our Genetic Programming portion of the project, we wanted to try to build a series of simple functions that when combined would be able to generally solve the minefield consistency problem for most minesweeper configurations. Due to the change of moving away from the restraints of P = NP and the consistency problem, we still wanted to follow a similar idea.

Overall, there have been a series of papers following a problem of solving minesweeper with genetic programming. In most cases though, they use a series of functions that don't really use proper memory associated with artificial intelligence. For the individuals in those papers, they are generally composed of simple functions attempting to brute force the right order of actions to solve a game minesweeper. Additionally, these works are limited within a small minefield of usually about a 5x5 set of cells, where some have 3 or less mines. In order to fulfill a similar role of solving a minesweeper game using genetic programming, we instead decided to create a system to differentiate between empty cells and cells that hold mines.

In a normal game of minefield, a person will uncover a covered field and deduce which cell is empty or holds a mine. With our own implementation, we rather want to differentiate based on a board which is partially uncovered. To achieve this goal, we have elected to use the Deap library only for generating tree individuals, crossover, and mutation. Overall, this has been very beneficial with generation, running, and testing different ideas in order to achieve our goal, but limitations with the library have led to concerns of bloat control and limitations of terminals.

1.5.1 Terminologies

- GP - Genetic Program
- Individual - Individual within the population of tree format comprised of terminal point offsets and primitive functions
- Known cell - Cells that are numbered 1-8
- Unknown cell - Cells that may be empty, empty near a numbered cell, or cells that possess a mine
- True Unknown cell - Cells that are -5, indicating that the system does not know what is there yet.
- Predicted Unknown cell - Cell that may be labeled as 0, -2, or -1, indicating that that cell is predicted to be either empty, empty near a numbered cell, or a cell that possesses a mine.
- Input coord - A coordinate setup to test the tree's offset calls against.
- Perfect minefield - The desired minefield we wish individuals to create by editing the dummy minefield
- Dummy minefield - A minefield of unknown and knowns that individuals attempt to edit into the perfect minefield

2 Literature Review

"Many programming problems require the design of an algorithm which has a 'yes' or 'no' output for each input" [1]. In Minesweeper is NP-complete by Richard Kaye, Kaye focused on the difference between two subsets of algorithms for the P=NP Problem. Firstly, Polynomial-time (P) Computable Algorithms, or Deterministic Algorithms, are a subset of algorithms that aim to solve problems with provided output for a given input. These problems are noted as polynomials because their run-time is consistent with whatever machine they are used on within an amount of time of order n, or n 2, or n 3, or n4,... with n being a certain number of symbols required for a given problem. The second subset of algorithms of interest is Nondeterministic Polynomial-time (NP) Computable Algorithms. NP algorithms aim to solve problems that are not uniquely defined; in other words, NP algorithms do not have consistent outputs based on consistent inputs which provides more random results. Kaye's idea behind using NP algorithms for minesweeper is that by guessing the positions of mines and then verifying if those guesses were correct, the program can learn to determine whether correct guesses were possible at all.

Allan Scott argues in his paper that while Minesweeper may not be NP-complete, it is co-NP-complete. In other words, he claims that while Minesweeper cannot be solved in polynomial time, a wrong answer can be proven wrong in polynomial time. Scott goes even further to say that the Minesweeper Consistency problem can be proven to be co-NP-hard, as the problem can be reduced to a problem in polynomial time. To do so, he chooses a different approach from Kaye by reducing the Minesweeper Consistency problem to the Minesweeper Inference problem. The Minesweeper Inference problem is defined as follows: "Does there exist at least one covered and non-flagged square whose content (either 'mine' or 'no mine') can be inferred from the available information?"[3]. He uses the Minesweeper Inference problem to prove that Minesweeper can be defined as co-NP-complete.

Chris Quartetti's paper, "Evolving a Program to Play the Game Minesweeper," describes several techniques to create programs to play Minesweeper through the use of genetic programming[7]. His model attempts to solve an 8x8 grid with 10 hidden mines. He describes: "a subset of the game will be used as a fitness measure due to CPU time and memory limitations. In general, the number of possible board configurations is given by the binomial theorem:

$$\binom{squares}{mines} = \frac{squares!}{mines!(squares - mines)!} \tag{1}$$

Hence the standard 8x8 grid with 10 mines has over 4 billion starting configurations." [7] As such, the program evaluates the fitness of each individual on every combination of a 6x6 board with 1 mine, which amounts to 36 cases. The terminal and function sets consist of an integer constant that represents 8 compass directions plus a null value, a function to move around the grid, functions to mark or uncover the square indicated by direction, a function to hook two functions together, a function to evaluate one or another argument depending on if a square is covered, a function to return the number of adjacent mines for the indicated square, and two automatically defined functions that were co-evolved with the main branch. For the parameters, populations of size 50,000 were used in runs with 151 generations, as populations start to plateau after this point. Crossover occurs with a rate of 90%, split into 10% for leaf nodes and 80% for interior nodes. No mutation was used. The resulting best-of-run individual was tested on all possible boards with one mine and on 100 random configurations with two mines. The individual scores 100% on all test cases used during evolution, and above 90% on boards from 5x5 to 11x11 with one mine. The individual was also more successful on boards with two mines when the mines are spaced farther apart from each other, likely due to looking like a single mine when viewed locally in the board. Quartetti compares the individual to a human following a strategy: it first navigates the board and uncovers squares, then becomes careful not to uncover mines. The techniques and results mentioned in this paper provided a useful frame of reference for our own genetic program.

3 Experiments and Development

3.1 Genetic Algorithm

For this part of the project, our objective was to create a program that can play the minesweeper with general, domain-independent knowledge and see how it performs. The program is divided into two main tasks that are repeated until the program wins or loses the minesweeper game. The first task is to use multiple parallel genetic algorithm instances to find a list of acceptable and optimal placements of the mines along the "frontier". The second task is to use a list of mine placements found by the genetic algorithm to calculate, for each square in the ”frontier”, the probability that the square contains a mine. These probabilities are then used to select the squares to reveal or flag.

3.1.1 Frontier

The program first begins by selecting a random square to start the game. To prevent the program from losing before it does anything, we programmed the game such that the first clicks are always safe.

Once the squares are opened, the program keeps track of the ”frontiers”, the edges of the hidden squares as shown in figure 2. The red squares indicate the squares to consider for the inference search problem since they are neighbors of numbers, which allows any inference. Every time a square is revealed, the revealed square is removed from the frontier, and squares from the new edge are added to the frontier for subsequent consideration. After the game has started, the frontier is fed as input to the genetic algorithm to begin the program.

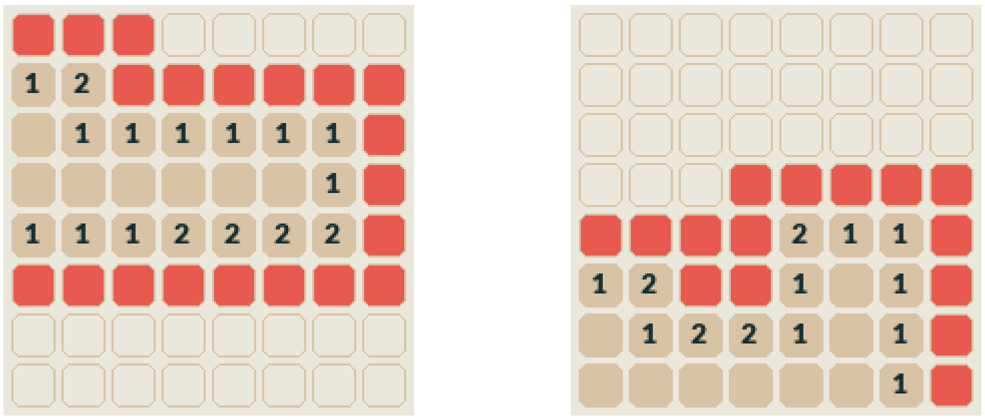


Figure 2: Partially revealed minesweeper board with frontiers shown as red squares

3.1.2 First part: Genetic algorithm

The genetic algorithm will have the individuals represented by a binary vector with the length equal to the number of squares in the frontier. Each bit corresponds with a specific square in the frontier and indicates whether the square has a mine (1) or not (0). The population size depends on the number of squares in the frontier. We opted to use twice the number of squares in the frontier with the minimum population size set to 10. The fitness of the individual is evaluated by the numbers that touches the frontier. Specifically, for each squares that have a mine, the numbers that touch the square are subtracted by 1. The sum of the absolute value of these resulting numbers is the fitness value for that individual. In essence, the fitness value is the measure of how much the numbers are violated by the placement of the mines given by the individual, so it will need to be minimized. The Python algorithm for evaluating the fitness is given in figure 3.

We decided to use a steady state scheme. The population was initialized with random binary vectors. Two parents for each round were selected by two binary tournaments. For crossover, we experimented with one-point crossover, k-point crossover, and uniform crossover. The crossover probability was 1, and we used a bit-flip mutation with each bit having $1/n$ probability of flipping.

The replacements were also selected by two binary tournaments, where the worse of the tournament was chosen.

```python
def evaluate(individual):
    calc = {}

    for square, hasMine in zip(frontier, individual):
        for neighbor in square.neighbors:
            if neighbor.revealed:
                if neighbor not in calc:
                    calc[neighbor] = 0

                if mine:
                    calc[neighbor] += 1

    error = 0
    for square, estimated in calc.items():
        if square.safe:
            error += abs(square.value - estimated)

    return error
```

Figure 3: Fitness evaluation function of the genetic algorithm

A genetic algorithm run terminated after a set number of iterations passed without any improvement to the best fitness. After termination, a specified number of best individuals were selected from the population and were placed in a pool of best individuals to be used for the second part. To allow diversity, we let several genetic algorithms to run in parallel with different initial population.

For the technical aspect, we first used Python along with pymoo module to create the genetic algorithm. However, we realized that the speed was underwhelming due to the nature of Python being an interpreted language. After the realization, we changed to C++ to see if we can see an improvement in the speed. Unfortunately, we were not able to find any useful modules for the genetic algorithm in C++ and created the algorithm from scratch. Most likely for this reason, we were not able to see any significant improvement in the performance.

3.1.3 Second part: Probability calculation

Once the pool had enough individuals, each genes were examined to see what the proportion of the corresponding square in the frontier had mines across all the individuals in the pool. For example, if there were 10 individuals with near-optimal fitness in the pool and 6 of these individuals had mine in the first gene, then the square that corresponds to the first gene is given the probability of 6/10. The minesweeper boards in figure 4 gives a demonstration of how the process works. As seen, each square in the frontier is given its own probability. The figure shows more redness with higher probability of the square containing a mine. There are 5 squares that have near 0 probability of having mines in them, so these squares are chosen to be revealed. This process along with the genetic algorithm repeat until the program reveals all squares not containing a mine or reveals a mine by accident.

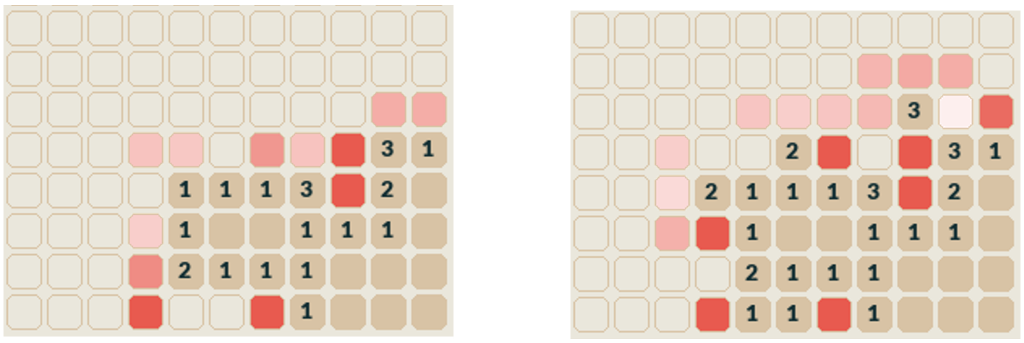


Figure 4: Before and after revealing safe squares

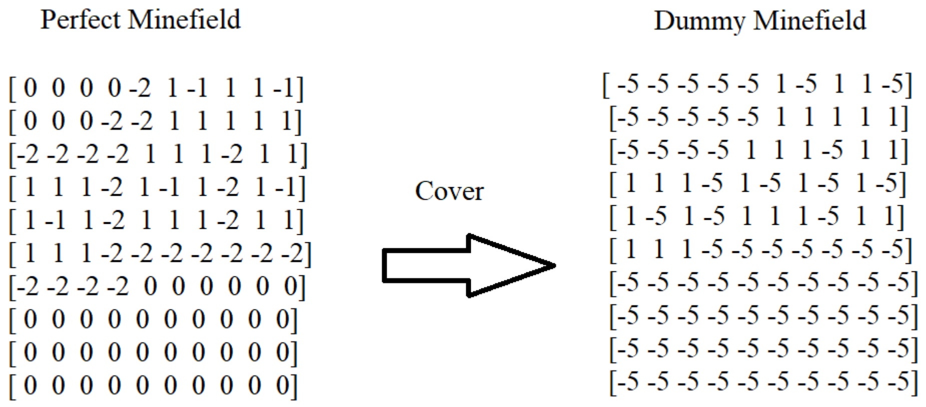
3.2 Genetic Programming

Minesweeper is fairly simple in terms of rules in determining how the game works, but developing a strategy to differentiate unknown cells is a different level of formulation. In some cases of a current board, there may be multiple points to account for that may hold a mine, as for example a mine being in cell x may be dependent on holding a mine if cell y is empty, but cell y may be dependent in being empty that cell z is holding a mine. With our own project, we are able to narrow down the options more as the numbered cells indicating if a mine is near are uncovered. Our individuals within the program are only able to evolve as well as we give them the tool-set to perform optimally. As we know the locations of all the positively numbered cells, our individuals within the GP are generally able to deduce a relative location of where they should be, but in some cases it is not always accurate.

3.2.1 Minefield Formulation

With a normal minefield game, each cell can generally be divided into two classes, known cells and unknown cells. In a normal game, known cells are those we know are empty, we know are numbered, or we know possess a mine. In a normal game, unknown cells are those that are 'covered' and we have not tried to uncover them yet. For our case with the Genetic Programming algorithm, we only wish to distinguish between certain types of unknown cells on a minefield that is mostly uncovered. More specifically, we want to build a formula that can distinguish between cells that are empty and do not possess a mine, cells that are empty but near numbered cells, and cells that possess a mine.

We generate a minefield normally, where we assign a series of mines then number the cells near those mines accordingly. At first, every cell is uncovered, including empty cells, numbered cells, and cells that possess a mine. Once the field is generated, we begin to assign values to the minefield that would arrange it to distinguish between the two types of empty cells and cells that possess a mine. Cells that possess a mine are labeled as -1, cells that are empty but near numbered cells are labeled as -2, and those cells that are empty and not near numbered cells are labeled as 0. We then create a copy of this field, named 'dummyfield', that takes any cell that is 0, -1, and -2 and changes those values to -5. We treat values that are -5 as 'true unknowns' while cells with values less than 1 as unknowns that are 'predicted knowns'. A cells that are marked as true unknowns are stored in a array of coords, where each of these is treated as an 'input coord.' This significance is later explained in the fitness section. This -5 change during formulation is done to cover the cells we want to test and treat every possible cell that can hold a mine as an unknown. Dummyfield is then edited when we perform our fitness functions, as the goal is for individuals to turn the dummyfield into the perfect minefield. Below is an example how we cover the perfect minefield for fitness testing.



Perfect Minefield

```
[ 0  0  0  0 -2  1 -1  1  1 -1]
[ 0  0  0 -2 -2  1  1  1  1  1]
[-2 -2 -2 -2  1  1  1 -2  1  1]
[ 1  1  1 -2  1 -1  1 -2  1 -1]
[ 1 -1  1 -2  1  1  1 -2  1  1]
[ 1  1  1 -2 -2 -2 -2 -2 -2 -2]
[-2 -2 -2 -2  0  0  0  0  0  0]
[ 0  0  0  0  0  0  0  0  0  0]
[ 0  0  0  0  0  0  0  0  0  0]
[ 0  0  0  0  0  0  0  0  0  0]
```

Cover

Dummy Minefield

```
[ -5 -5 -5 -5 -5  1 -5  1  1 -5]
[-5 -5 -5 -5 -5  1  1  1  1  1]
[-5 -5 -5 -5  1  1  1 -5  1  1]
[ 1  1  1 -5  1 -5  1 -5  1 -5]
[ 1 -5  1 -5  1  1  1 -5  1  1]
[ 1  1  1 -5 -5 -5 -5 -5 -5 -5]
[-5 -5 -5 -5 -5 -5 -5 -5 -5 -5]
[-5 -5 -5 -5 -5 -5 -5 -5 -5 -5]
[-5 -5 -5 -5 -5 -5 -5 -5 -5 -5]
[-5 -5 -5 -5 -5 -5 -5 -5 -5 -5]
```

Figure 5: Dummy Minefield Formulation

3.2.2 Individuals

Our individuals are represented by a series of functions and terminal points. The primitive functions are either those that push a terminal point up a tree or those that make an edit to a dummy minefield then push a terminal point up the tree. Our terminal points are tuples that are composed of x and y offsets. As of now, each individual is comprised of branches that, for the most part, are of uneven depths but can be of equal depths based on the randomness of the initial generation, mutation, or crossover. We define minimum depth and maximum depth before running. For the initial population of individuals, no two individuals can have the same fitness, as this is done to make the population more diverse. Specifics on functions can be see in the development section.

The individuals themselves wont return a fitness function to derive score from. Instead, the minefield they edit will be checked against the perfect minefield.

3.2.3 Fitness

The goal of our fitness system is to create a 1:1 copy of the original minefield by deducing if every unknown cell is a cell that is empty, empty near a positively numbered cell, or is a cell that possesses the mine. All cells that are positively numbered, indicating a mine is near it, are already uncovered. We do this by taking every coordinate, the 'input coord', that is -5 on minefield generation and running the individual's GP tree function against it. Each primitive function in the tree will take two terminal offsets and perform a check to see if the offseted cell from the input coord cell verifies the conditions of the function. If the check comes out as desired, it will perform another check and may edit a copy of the original 'dummy' minefield based on combination of the offset and input coord. The function will then decide which offset to return up to the next function in the individual's tree based on if either offset fulfilled the current primitive function's check.

After all the input coordinates have been passed into the tree function, we get the edited minefield and compare it against the original 'perfect' minefield. We then assign a score based on how many unknown cells are the same or similar between the edited minefield and the 'perfect' minefield, where we max fitness points for perfectly computed cells and partial points for cells that may be the wrong type of 'empty' cell. Example of fitness function can be seen below. Each run of the fitness function uses a covered dummyfield as specified in generation.

```
def fitness(self, indv):
    fit = 0
    tree = copy.copy(indv)
    self.funcObj.setMinefield(copy.copy(self.dummyminefield))
    mf = self.perfectMinefield

    for inputCoord in self.minefieldcords:
        self.funcObj.setCord(inputCoord)
        gp.compile(tree, self.orgPrimitiveset)

    mp = self.funcObj.getResult()
    for arc in self.minefieldcords:
        if mf[arc[0]][arc[1]] < 1:
            if mf[arc[0]][arc[1]] == -2 and mp[arc[0]][arc[1]] == -2:
                fit +=2
            if mf[arc[0]][arc[1]] == -2 and mp[arc[0]][arc[1]] == 0:
                fit += 1
            if mf[arc[0]][arc[1]] == 0 and mp[arc[0]][arc[1]] == 0:
                fit += 2
            if mf[arc[0]][arc[1]] == -0 and mp[arc[0]][arc[1]] == -2:
                fit += 1
            if mf[arc[0]][arc[1]] == -1 and mp[arc[0]][arc[1]] == -1:
                fit += 5
    return fit
```

Figure 6: Fitness evaluation function of the genetic programming

3.2.4 GP Algorithms and Operators

We have two different algorithms that are defined, steady state and a $(\mu + \lambda)$ generational system. Most tests are conducted using steady state as the generational system takes a considerable amount of time to finish. Both terminate either after a set number of replacements or generations in the case of the generational. In the case a best fit has been achieved, both will terminate.

Our primary replacement operator, replacementClosest, will replace the individuals in the current population whose fitness is closest to whichever child it is currently attempting to insert. In the case both have the same closest fitness relation in the current population, it prioritizes one over the other, then selects the next closest for the other child.

Our secondary operator, replacementWorst, will attempt to replace the worst individual in the population with the desired child if the child has better fitness. In the case that an individual already exists in the population with the same fitness as the child, that individual already in the population is replaced by the child.

As with the Deap library, we have mutation and crossover functions already predefined. We use a version of one point crossover and a version of uniform mutation that inserts a small tree on a desired node to be replaced. The generated individual at most has a maximum depth of 5 a minimum depth of 2. Once this small individual is generated, a node on the individual we want the actual mutation to occur to is replaced with the root and proceeding branches of the newly generated individual. Mutations occur with a 50% chance.

3.2.5 Terminal Points

For our trees, we utilized terminal point offsets in the form of tuples who, when fed into our primitive functions, will be offset from the input coords specified in the fitness function. The tuples are in the form of (i,j) where x is the number of rows in the minefield and y is the number of columns and -x<i<x and -y<j<y. This works quite well, as we are able to reach multiple different cells based on each current input coord and make a determination based on them. We chose to have our terminals be offsets from other cells as we came to the conclusion this would allow our system to go back and check cells that are predicted unknown and see if anything about those surrounding cells have changed. If our terminals were just the coordinates of all unknowns, some may not be checked based on tree generation or the trees may not be able to go back and check certain cells after certain changes. With these offsets we can guarantee most functions will attempt to edit each cell more then once after passing through all possible input coords as specified in our fitness function.

3.2.6 Primitive Functions

For any case of making a decision on what type of cell an unknown cell is, it is foremost important to decide based on the immediate surrounding, but cells further from the immediate surrounding may set up conditions that would otherwise prevent certain cells from being a certain prediction. As our terminals are offsets, the hope is that we will be able to do a majority of checks against other cells to slowly narrow what is predicted correctly and what isn't. Most of our trees' primitive functions are either in the category of pushing a terminal offset up the tree or making a decision based on the terminal offset against the input coord. With certain terminal offsets, the offsetted cells when combining with the input coord may be in 'important locations.'

Function Name	***Function Description***
emptyCheck	Checks if either offset in relation to input coord is empty. If the offset is a neighbor of the input coord and empty, input coord is set to empty. Returns offset if either is empty, random if neither or both.
neighborCheck	Checks if either offset in relation to input coord is a direct neighbor. Returns offset if either are neighbor or random if neither or both.
outerNieghborCheck	Checks if either offset in relation to input coord is a outer neighbor. Returns offset if either are outer neighbor or random if neither or both.
numCheck	Checks if either offset in relation to input coord is a positively numbered cell. If it is a positively numbered cell and a direct neighbor of input coord, input coord is marked as -2. Returns offset if either are positively numbered or random if neither or both.
unknownCheck	Checks if either offset in relation to input coord is a true unknown. Returns offset if either are true unknown or random if neither or both.
fullCheck	Checks if either offset in relation to input coord is a positively numbered cell. If it is, checks if the neighbors to that cell have the number of unknowns corresponding to the cell's number. If the cell does, its unknown neighbors are marked as -1, indicating they hold a mine. Returns offset if either are positively numbered or random if neither or both.
filledCheck	Checks if either offset in relation to input coord is predicted to be a cell holding a mine. If it is, checks if every cell surrounding it is positively numbered. If it isn't, it is marked as -2, indicating it is an empty near a numbered cell. Returns offset if either are marked as holding a mine or random if neither or both.
emptyTouchCheck	Checks if either offset in relation to input coord is either type of empty cell. If it is, it determines if it is the correct type of empty cell and changes it based on its neighbors. Returns offset if either are marked as empty or random if neither or both.
outerCompare	Checks if either offset in relation to input coord is a outer neighbor. In the case it is an outer neighbor and the input coord cell isn't 0, the input coord cell is then set to -1, indicating it is predicted to hold a mine. Returns offset if either are outer neighbor or random if neither or both.
edgeCheck	Checks if either offset in relation to input coord is located on any edge of the minefield. Returns offset if either are on an edge or random if neither or both.

Table 1: Overview of all primitive functions possible within an individual.

3.2.7 Algorithm and Bloat

We developed the system with the previously mentioned $(\mu + \lambda)$ generational algorithm and steady state algorithm. Both were made to test if results were consistent between two different systems of avoiding local optimal convergence. It was found that though the generational algorithm was as consistent with results as the steady state, the generational did take far too much time to consistently test. For each individual having max depth of 15 on generation, our steady state could converge anywhere between 130 seconds to 300 seconds. For a our generation system where each parent made 6 children, it seemed to take upwards of 50 minutes for most runs with max depth greater or equal to 10. This could be due to creating too many children then taking the fitness of those children, but regardless it was decided to perform all future testing in the steady state algorithm.

Bloat is an issue with any form of genetic programming and is typically a phrase relating to the size of a tree, as a tree will become plump with functions and terminals that may not improve its score compared to smaller trees. Thankfully, we can try to limit by setting depth limits at least during initial creation, but crossover and mutation leads to other issues. For our one point crossover, a point for crossover on one tree is less likely to be a leaf based on if the tree has a high depth. Unfortunately, we do not have a way to limit the amount of leaf nodes selected, so bloat may occur in the case that if a point close to the root is selected on parent 1 then a leaf is selected on parent 2. Mutation is the same issue where we cant define what point we wish to select for mutation, but the uniform mutation function from the Deap library does allow us to limit how large the generated set of new nodes will be.

4 Results

4.1 Genetic Algorithm

We ran the program on randomly generated minesweeper boards at 3 different difficulty levels commonly seen in minesweeper games. The beginner board is an 9 by 9 board with 10 mines. The intermediate difficulty has 16 by 16 board with 40 mines. Lastly, the expert difficulty has 30 by 16 board with 99 mines. The performance of the program is shown on table 2. When the board got bigger (the number of mines stayed proportionally constant at around 20%), the average time increased and the win rate decreased significantly. The average time also includes the runs that lost the game, but some of these runs were able to last close to the end. The sample successful runs of beginner, intermediate, and expert difficulties are uploaded on these links: beginning, intermediate, and expert.

Configuration	Number of wins	Number of trials	Win rate	Average time (m)
Beginner 9 x 9, 10 mines	46	100	46%	2.41
Intermediate 16 x 16, 40 mines	7	25	28%	11.83
Expert 30 x 16, 99 mines	1	10	10%	27.01

Table 2: Genetic algorithm performance with uniform-crossover operator

To make the program faster, we tried to split the frontiers into smaller parts that are connected. Unfortunately, this led to the win rate being reduced drastically. On the expert difficulty, the program was not able to last more than 10 iterations. It was able to win 1 game from 15 games (6.6% win rate) on the intermediate difficulty, and 9 games from 50 games on the beginner board (18% win rate).

We also tried 3 different crossover algorithms as mentioned earlier: one-point crossover, k-point crossover (k was set to 3), and uniform-crossover. However, we were not able to see any difference in the performance, and we arbitrarily opted to use uniform-crossover.

4.2 Genetic Programming

While gathering results, we used two main settings of minefield size and two main settings of initialized tree depth. For minefield size, we tested sizes of 10x10 minefields as well as 16x30 minefields, as the 10x10 is a setting somewhere between easy and medium in a normal minesweeper game and 16x30 is usually reserved for the expert setting in minesweeper games. For these fields, 10% of the cells were guaranteed to hold a mine. For out tree depths, it was found that trees of max depth 15 were able to complete within a sufficient amount of time without too long of a run. Because of this, we alternated between two max depths on initialization of the population, one of 5 and one of 15, where the minimum for both was 3. To decide if our system was beneficial overall, we tested on 2 sorts of paradigms, one of which being the efficiency of our genetic program algorithm to produce good individuals within an acceptable time frame and the other being if the individuals created were applicable to other minefield settings. Though we have our $(\mu + \lambda)$ generational functioning, we believe that due to how long it takes to complete, it is not representative of generating a good individual as our steady state algorithm is. We have elected to present our results gathering data only from the steady state algorithm of our genetic programming structure. For generating individuals, we allowed a max iteration of 1500, where 1500 pairs of crossovers may be generated, allow for a maximum of 3000 possible insertions into the population.

4.2.1 Steady State Efficiency

Overall we have had favorable results with our steady state algorithm. As previously stated, our generational system takes far too long to get any worthwhile results so all information will come from our steady state variant. During minefields of small sizes, it was found that our population would often converge on a best fit before the algorithm reached past 50 iterations, where those minefields smaller than 5x5 would often create an individual of best fit during population initialization. This means that overall at difficulty settings lower then easy and unbeknownst to a normal minesweeper, our genetic programming algorithm will produce individuals that can differentiate all possible empty, empty near known cells, and cells that possess mines.
For those minefield settings harder then 5x5 fields, our best fit will vary. We generally have proportional scores higher than .90 of the possible score for each field, but the cases of reaching a max score tend to take at least 1500 iterations as previously defined in the super section. Cells that hold mines are generally predicted correctly with some exceptions, but issues occur during differentiation of empty and empty near numbered cells. This issue may relate to the strength of our functions, as though we have some setup to deal with this differentiation, not all functions tend to be reached with the right offset in the tree. In some runs, we also have coverage issues, as a good but less then optimal individual will at least attempt to try and assign a prediction to all cells that are unknown. In these runs with a coverage issues, not all of the current dummyfield will have a prediction/edit as made by the best fit individual. This may be because we do not have penalties for failing to edit each cell, but overall the fitness scores are quite good as indicated by table 3.
Time is also a statistic we looked at for each run. We had 8 possible runs with a combination of max tree depth, minefield size, and replacement operator. Though our worst replacement operator seemed to create individuals with the similar proportional score to our closest replacement operator, it appears to take the longest when it comes to the expert minefield setting of 16x30. The easier setting of 10x10 minefields don't have too much of a time difference, but its clearer that only due to the time constraint of our worst operator, the closest replacement operator does perform better.

Depth,Size,Replacement	***Avg. Proportional Score***	***Average Time(seconds)***
15 max depth, 16x30, closest	0.95936	170.150889
15 max depth, 10x10, closest	0.99813	24.4551857
5 max depth, 16x30, closest	0.95575	168.009946
5 max depth, 10x10, closest	0.99288	30.1335972
15 max depth, 16x30, worst	0.97651	461.7006968
15 max depth, 10x10, worst	0.99316	82.042813
5 max depth, 16x30, worst	0.97748	759.431931
5 max depth, 10x10, worst	0.99244	44.1767717

Table 3: Steady state results with closest and worst replacement operators averaged after 15 runs. For average score, the score is proportional to the max possible score, where 1.0 is max score.

4.2.2 Generalized Formulas

With a game such as minesweeper, it would be best to try and develop a general function to try and differentiate minefields of similar size. In the case there is any sort of generalized function, the tree functions would at least be limited to cells of the same structure, as the inclusion of terminal offsets require that they be within the previously stated range. Early on, we believed we could try to develop a generalized set of primitive functions and terminals that could solve minesweeper games set within the same row and column domain. This proved to be much harder then possible and most likely unrealistic. To validate this theory, we ran the genetic programing algorithm with a min tree depth of 3 and max three depth of 15 on population initialization. After running for 1500 iterations, we validated the best individual against 30 different minefields to test its fitness. Overall, the fitness on the best individual originally managed to reach around .985 proportional to the best possible score on its own minefield it was tested and built against. Against the 30 different fields the range was variable. For one best fit individual, the maximum scored tested against the 30 validation fields was .93333, but another validation field also resulted in the minimum score of 0.78974. Other instances of testing different best generated individuals reached around .89 max proportional score and minimum of .66 proportional accuracy.

With these levels of difference between both, it could be argued that our system could generate individuals at least are able to differentiate more than half of the empty cells, empty cells near numbers, and cells possess mines. Overall, there is little consistency with this system as being able to develop generalized formulas, so it should be better used for one-off instances. More in depth information can be seen in table 4 below in our validation attempts.

Depth,Size,Replacement	***Proportional Score on Original Field***	***Avg. Proportional Score on Validation***
15 max depth, 16x30, closest	0.97222	0.91716
15 max depth, 10x10, closest	0.97656	0.82003
5 max depth, 16x30, closest	0.94413	0.87648
5 max depth, 10x10, closest	1.0000	0.75490
15 max depth, 16x30, worst	0.99057	0.89053
15 max depth, 10x10, worst	1.00000	0.79131
5 max depth, 16x30, worst	0.97546	0.90084
5 max depth, 10x10, worst	0.99230	0.85296

Table 4: Results of running best fit individuals with given settings from against 30 validation minefields.

5 Conclusions

5.1 Genetic Algorithm

Our minesweeper solver using the genetic algorithm did not perform as well as other solvers. The solver using logic inference, CSP, and sampling by Tang, Yimin, performed better at significantly less time [8]. We believe that the reason for our slow up was that the number of squares in the frontier rapidly grew with larger boards and it became more and more difficult to find the optimal placement of the mines along that frontier as the game proceeded. Furthermore, with increased sized of the individuals, it became more difficult for GA to find the global optimum where the fitness was equal to 0, meaning no violation of the numbers. This led to the probabilities being less than optimal and the program choosing wrong squares to reveal.

In terms of the different crossovers, the absence of a noticeable differences in the performance could be attributed to the fact that the performance of these crossover operators may depend on the shape of the frontier, not just the order. Although the frontier was saved as a one-dimensional list of squares, the shape of the frontier on the board is much different than just a line. Therefore, distributional bias and positional bias might not have played a big of a role or the effect fluctuated based on the shape of the frontier.

We were quite surprised at the performance, however. This program did not have domain-specific knowledge apart from the fitness evaluation, which was just an application of the rules of the game. The program did not use logic inference, matrix, or common pattern dictionary (which is what people usually use). It essentially was a guess and check, where the guesses evolved with each check. Although the program was slow, the program was able to show an acceptable rate of success. This shows the advantage of evolutionary algorithms, mainly its ability to be generalized to many domains of problems.

We believe that the use of genetic algorithm is on a promising path. Instead of the fitness function we used, we can utilize a fitness function that borrows from a problem specific knowledge. One of the example is to consider the inference search as logic equations to solve with variable (similar to SAT problem) and use matrix calculation. Also, smart repair operators could be used that incorporates well-known minesweeper strategies and patterns.

5.2 Genetic Programming

For our genetic programming systems, we have found favorable results, though not consistently perfect predictions. As stated before in the genetic programming portion of our results section, there still appears to be issues relating to differentiating between empty cells. Part of this could be a error on our part in developing primitive functions, as this system only works as well as the cases in minesweeper we are able to identify. The primitive functions are only as strong as we can make them, where in a game in minesweeper there can be a variable amount of actions people can make to deduce what cell is what on a minefield. An additional issue of our primitive functions is that we allow them to freely edit any and all cells on the dummyfield that a current individual is editing. This means that in the case of an individual attempting to edit the minefield, it may offset and make an accurate prediction on one cell based off of an offset from its current input coord. On running the individual's tree from another input coord, that may change the accurate prediction based on another function and offset in the tree, which may make the accurate prediction and false prediction, overall decreasing accuracy. A fix to this may be to setup a second form of memory to establish cells that the system can 100% verify is correct after editing.

Another issue is bloat, which is covered in development, but there are still issues of bloat over time. With initial depth settings of 5 or 15, crossover or mutation may create individuals of much larger depth. Retrospectively, there are ways to track the depth of an individual as with the Deap library we are using. Simple checks could've been in place to prioritize individuals with smaller max depth, but the thought was limiting depth to a certain level would result in lower fitness scores. As of now, we have not checked stats in regards to fitness score and max depth. We believe that with the larger minefield settings, namely the 16x30 fields we used in testing, bloat occurs more frequently only due to the fact there is more terminal point variety in the larger fields.

Finally, we do believe that though this system of differentiating between the defined unknown cells does perform well, the application is limited. In this system now, it only works in the case that all numbered cells are uncovered in a normal minesweeper game, which is very unlikely. This system could be optimized to try and make predictions on minefields when trying to expand edges, our GA attempts to do, but otherwise the computation time may make this non optimal. The application of our genetic programming system may be better suited to smaller minefields, as our better suited steady state algorithm seems to take substantial amount of time for larger fields.

6 Contributions

- JP Park - Introduced the idea of minesweeper solver using generic algorithm and genetic programming. Coded a significant portion of the genetic algorithm. Made the outline of the progress report presentation. Reviewed and suggested literature that were relevant to our topic.
- Joey Liou - Discovered and analyzed research related to the minesweeper consistency problem.
- Ethan Bray - Performed significant development on genetic programming portion of project, formulation of tree and discussion of research topics.

- Ellison Pyon - Formulated genetic programming primitives. Researched literature relevant to minesweeper. Reviewed professional work on Minesweeper Consistency Problem to reference in literature review.
- Dominic Francesconi - Researched and reviewed literature relevant to the topics of genetic algorithms and programming in the context of minesweeper. Assisted in the formulation of genetic programming primitives.

References

[1] Kaye, Richard. "Minesweeper is NP-complete." The Mathematical Intelligencer 22.2 (2000): 9-15.

[2] Zhang, Shiwei, et al. "Optimization of neural network based on genetic algorithm and BP." Proceedings of 2014 International Conference on Cloud Computing and Internet of Things. IEEE, 2014.

[3] Scott, A., Stege, U. & van Rooij, I. Minesweeper May Not Be NP-Complete but Is Hard Nonetheless. Math Intelligencer 33, 5–17 (2011). https://doi.org/10.1007/s00283-011-9256-x

[4] Vomlel J., Tichavský P. (2014) An Approximate Tensor-Based Inference Method Applied to the Game of Minesweeper. In: van der Gaag L.C., Feelders A.J. (eds) Probabilistic Graphical Models. PGM 2014. Lecture Notes in Computer Science, vol 8754. Springer, Cham. https://doi.org/10.1007/978-3-319-11433-0_35

[5] Adamatzky, Andrew. (1997). How cellular automaton plays Minesweeper. Applied Mathematics and Computation - AMC. 85. 127-137. 10.1016/S0096-3003(96)00117-8.

[6] Buffet O., Lee CS., Lin WT., Teytuad O. (2013) Optimistic Heuristics for MineSweeper. In: Chang RS., Jain L., Peng SL. (eds) Advances in Intelligent Systems and Applications - Volume 1. Smart Innovation, Systems and Technologies, vol 20. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-35452-6_22

[7] Quartetti, Chris. (2000). Evolving a Program to Play the Game Minesweeper. Genetic Algorithms and Genetic Programming at Stanford. 6.

[8] Tang, Yimin., Jiang, TianHu., Yanpeng. (2018). A Minesweeper Solver Using Logic Inference, CSP and Sampling.