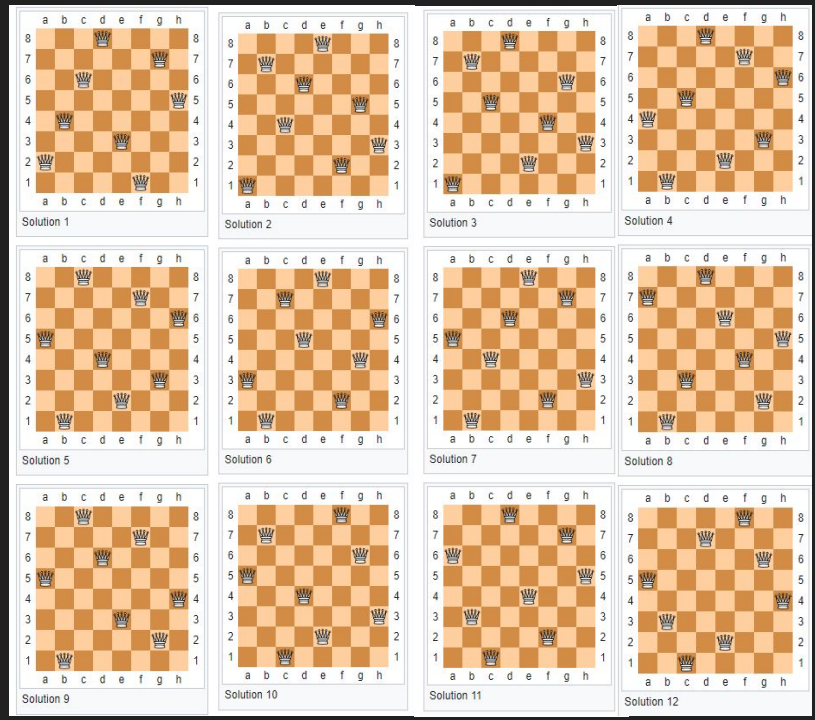
N-Queens Evolutionary Computation

By: JP Latreille

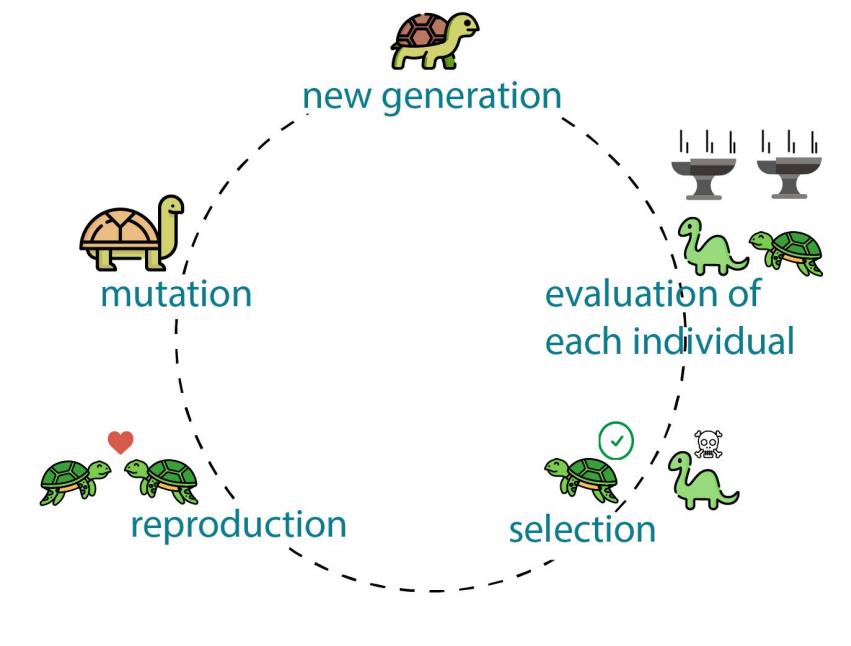
N-Queens Problem Statement

- The N-Queens problem is a toy computational problem that asks how many queens can fit on an N by N chessboard without sharing the same row, column or diagonal.
- For an 8x8 board there are 92 possible correct solutions.
- Only 12 solutions to an 8x8 are rotationally unique (obtained by rotating the board 90, 180 or 270 degrees).
- This problem makes for a good example of a problem that can be optimized through genetic algorithms in a very explainable way.



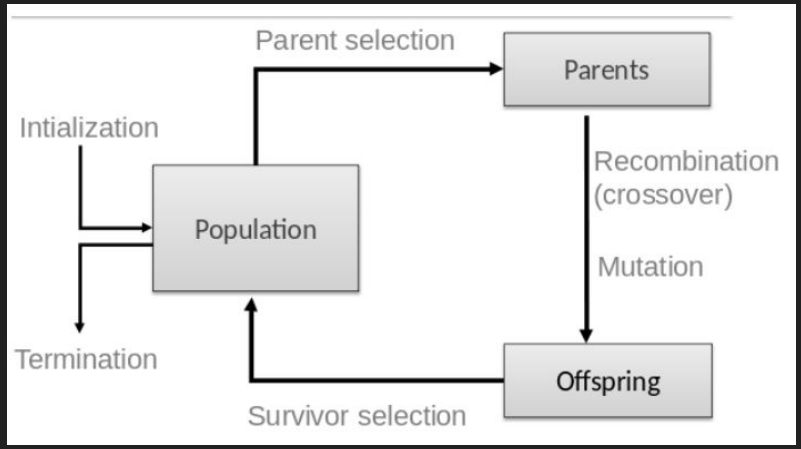
Advantages of Evolutionary Computation

- Explainability
  - Ability to view how we got to solution by looking through the generations that lead to solution
- Robust
  - Will converge over time if correctly implemented
- Uses probability selection to converge to a solution
- Good choice for large scale optimization problems with a wide search space.



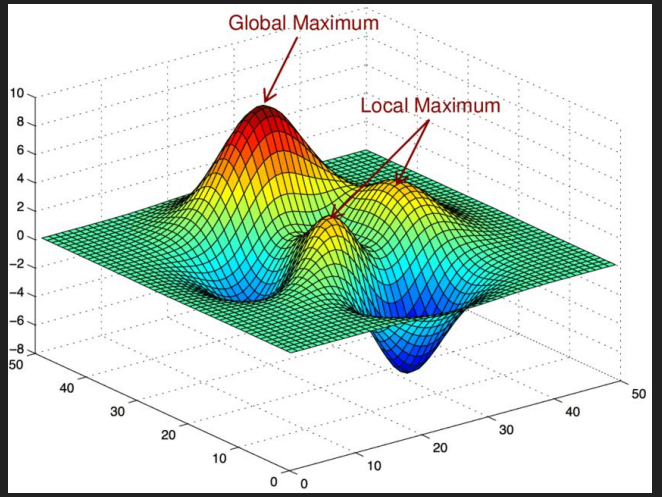
Evolutionary Computation Methods Overview

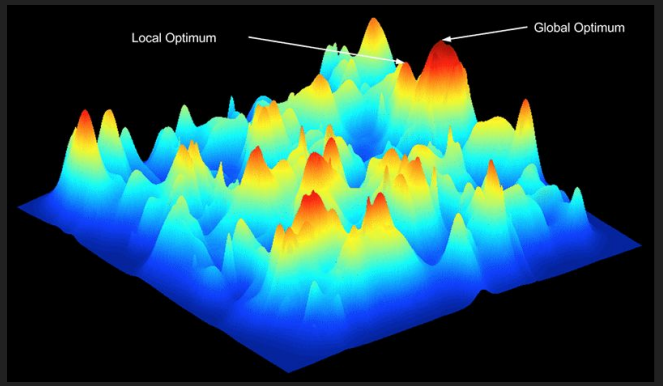
```
BEGIN
  INITIALISE population with random candidate solutions;
  EVALUATE each candidate;
  REPEAT UNTIL ( TERMINATION CONDITION is satisfied ) DO
    1 SELECT parents;
    2 RECOMBINE pairs of parents;
    3 MUTATE the resulting offspring;
    4 EVALUATE new candidates;
    5 SELECT individuals for the next generation;
  OD
END
```



Explanation of Search Space

- The complexity of a problem is defined by the set of all possible solutions or values in which the inputs can take in.
- A search space is likely to have local optimum and global optimum, which will provide good and great answers to the problem.
- Genetic algorithms will help us to efficiently climb through the search space to find a local maximum.
- The larger the board the longer it will take to converge and more likely it is to stay on a local minimum



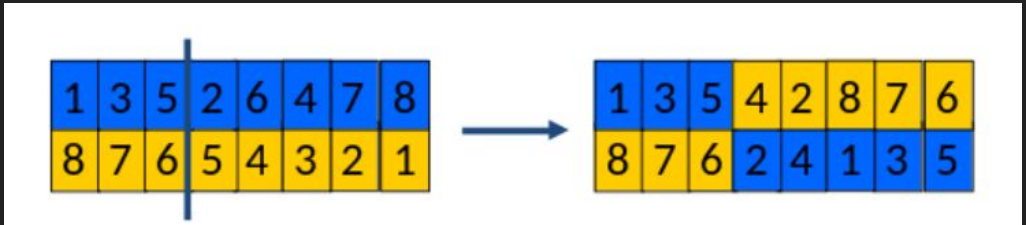


Representation

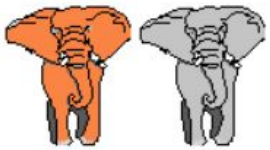
- Each individual in a population will consist of a genome and a fitness
- The genome will describe the solution to the problem and will be represented as an N sized array based on the board size of N by N.
- Each position in the array will consist of a integer representing which row the queen will fall and its column position will correspond to the index in the array.

Convergence Methods: Recombination

- Recombination create a new individual out of two highly rated individuals from the prior population
- This will help to maintain helpful chunks of information in solutions.
- This is done by choosing a random point in the strings and flipping the parts to create two new individuals, some error correction may need to be made to avoid repetition.



Parents

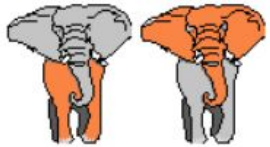


Mark middle of genome:

Parent 1: **1 1 1 | 1 1 1 1**

Parent 2: **0 0 0 | 0 0 0 0**

Offspring



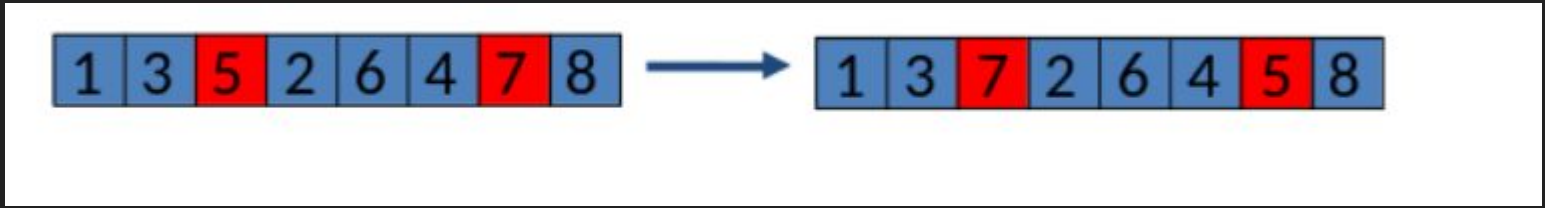
Swap around mark to generate new genome:

Child 1: **1 1 1 | 0 0 0 0**

Child 2: **0 0 0 | 1 1 1 1**

Mutation Method

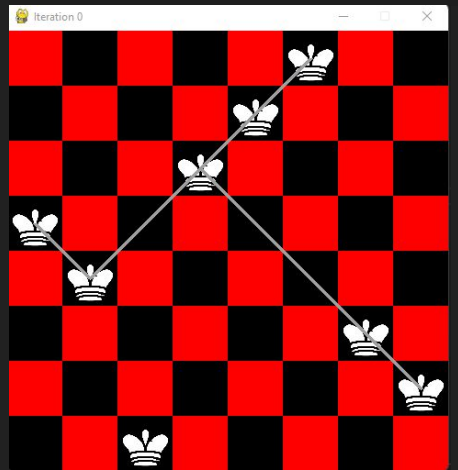
- Mutation will happen to a population at a set probability and will randomly change the genome of the individual.
- For each position in the genome the probability is tried for each position, if the probability is hit the value will randomly swap positions with another in the genome.
- This will take one individual and will produce one individual
- Done after recombination to population



Evaluation of Fitness

- The fitness is incremented for each queen in conflict with another
- The perfect fitness for each problem should converge to 0.
- I wrote an algorithm to check diagonals of the chess board to see if any of the pieces shared a left/right diagonal or row and then will penalized the fitness based on the sum of conflicts.

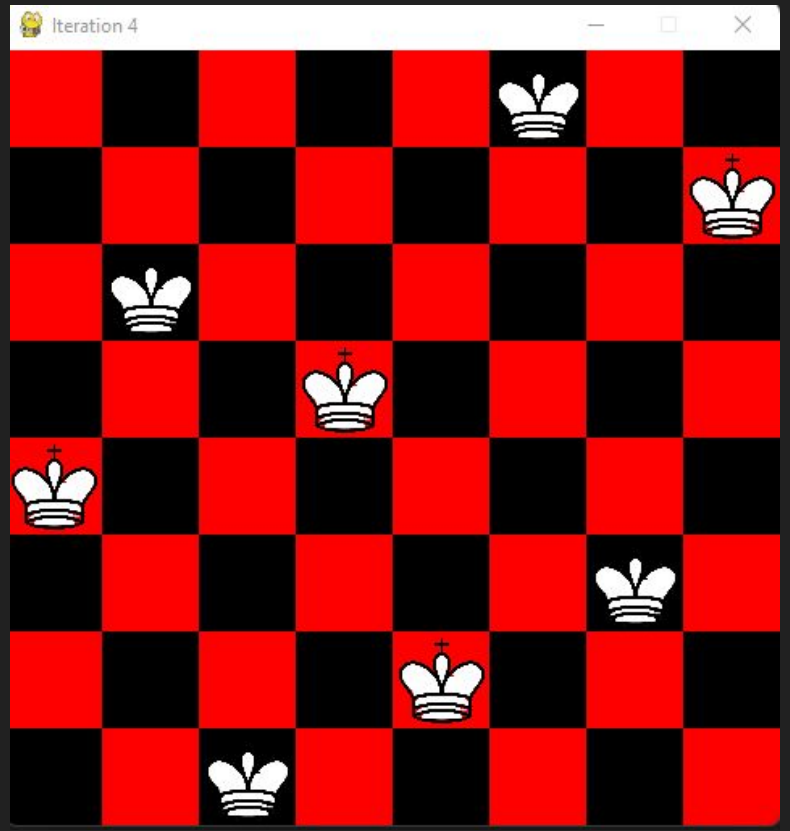
```python
def evaluate_individual(individual: Individual) -> None:
    """
    Purpose:        Computes and modifies the fitness for one individual
    Parameters:     Objective string, One Individual
    User Input:     no
    Prints:         no
    Returns:        None
    Modifies:       The individual (mutable object)
    Calls:          Basic python only
    """
    left_diagional = []
    right_diagional = []
    for col, row in enumerate(individual["genome"]):
        row = int(row)
        if row - col == 0:
            left_diagional.append(0)
        elif row - col < 0:
            left_diagional.append((row - col))
        elif (row - col) > 0:
            left_diagional.append(row - col)
        right_diagional.append(row + col)
    right_fit = sum([right_diagional.count(i) > 1 for i in right_diagional])
    left_fit = sum([left_diagional.count(i) > 1 for i in left_diagional])
    rpt_fit = sum([individual["genome"].count(i) > 1 for i in individual["genome"]])

    # for every confict found add a point to fitness
    individual["fitness"] = right_fit + left_fit + rpt_fit
```



- This board has 7 pieces in conflict, two of the pieces are conflicted twice so the fitness will be 9.

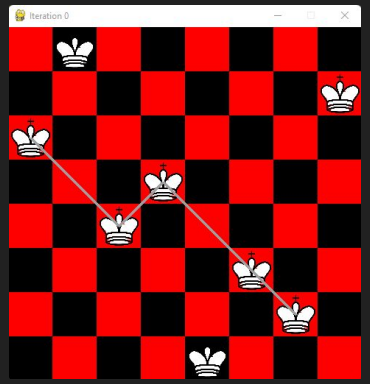
8x8 Solution

- The 8x8 solution was converged on very fast due to our efficient representation of the problem.
- With a population of 100 it could solve within 4 iterations.

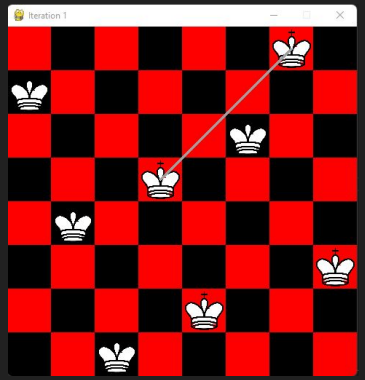


8X8 Longer Convergence

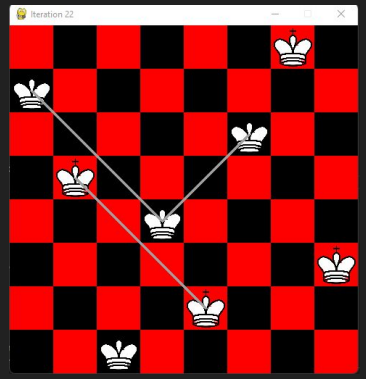
Generation 0



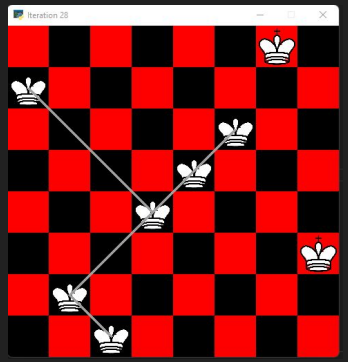
Generation 1



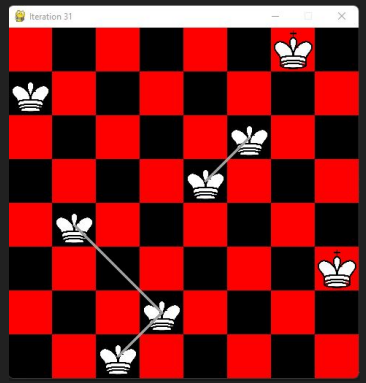
Generation 22



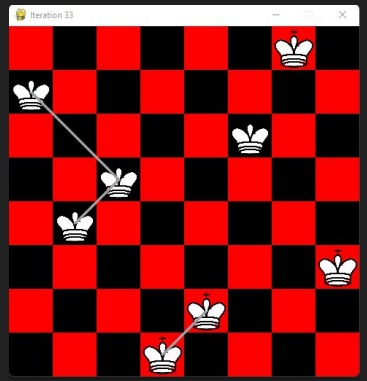
Generation 29



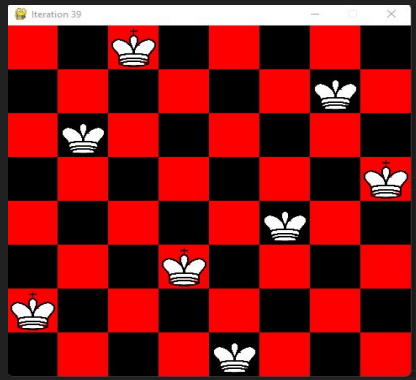
Generation 31



Generation 33

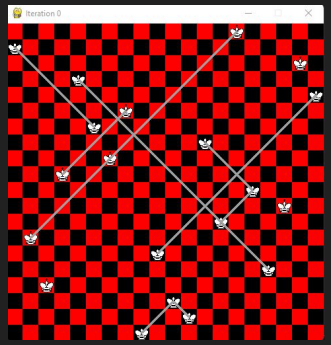


Generation 39

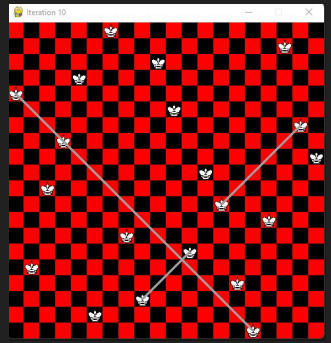


20x20 Solution Convergence

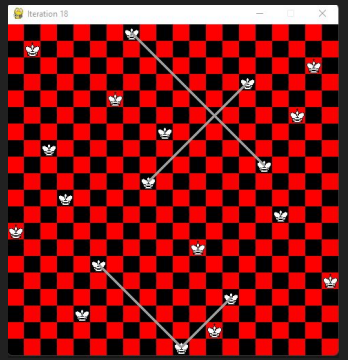
Generation 0



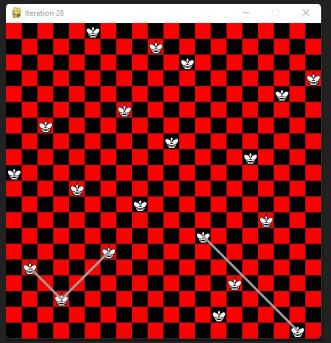
Generation 10



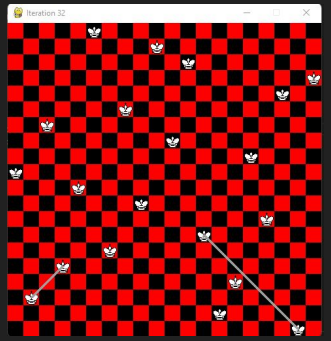
Generation 18



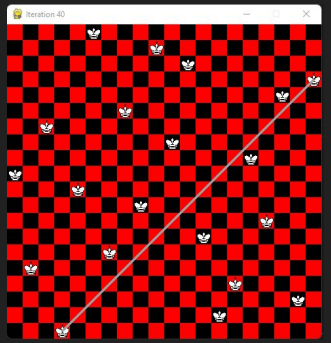
Generation 28



Generation 32

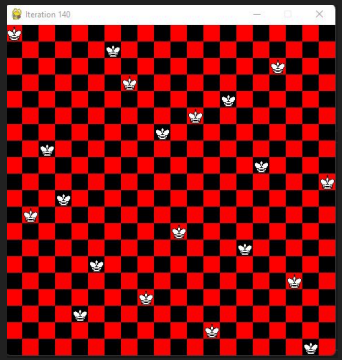


Generation 40



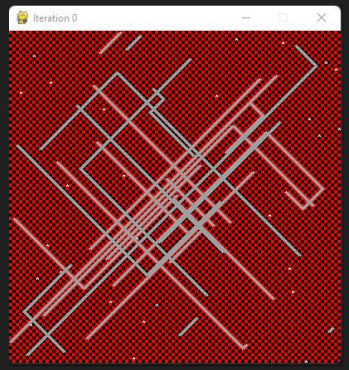
Stuck at a local maximum for 100 iterations…

Generation 140

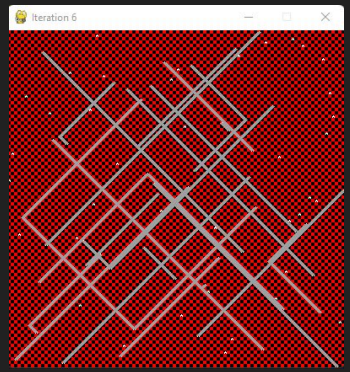


100x100 solution

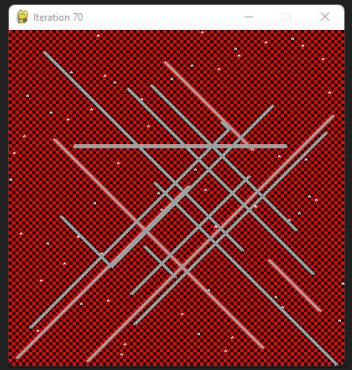
Generation 0



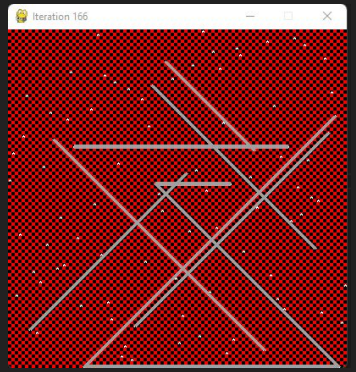
Generation 6



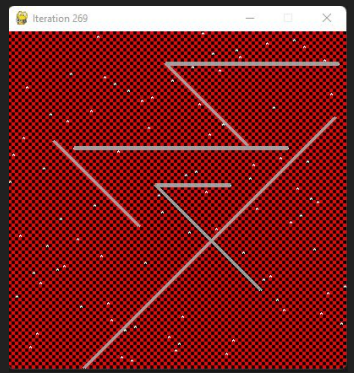
Generation 70



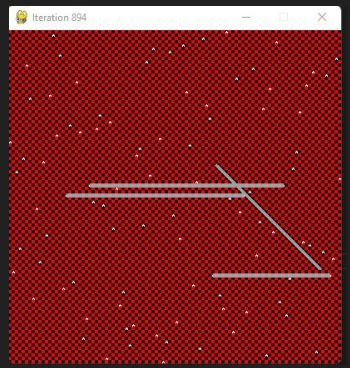
Generation 166



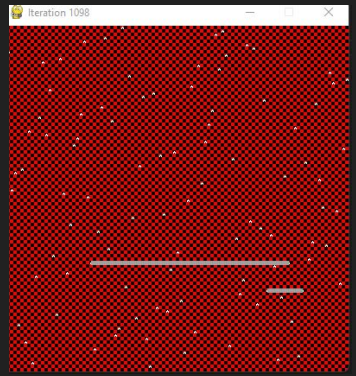
Generation 269



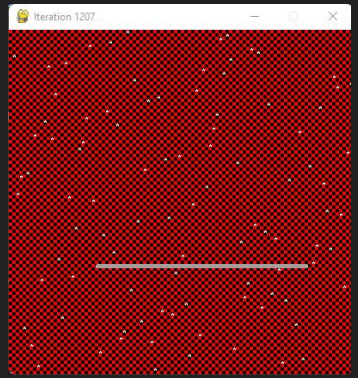
Generation 894



Generation 1098



Generation 1207



A fitness of 4 was the lowest we could converge to for N=100

Questions?