Ted Holmberg

Fall 2019: CSCI 6522: Programming Assignment #1 - Part B:

Genetic Algorithm Algorithm

Abstract:

In Part B, Genetic Algorithm is implemented to fold a protein structure in 2d space that maximizes the given fitness function which states that hydrophobic amino acids should cluster together into topological neighborhoods

Table of Contents:

I. Problem Statement

Part 2 [Marks 80]

#2. A protein sequence can be expressed as a simplified presentation, which is made of only 2 types of amino-acids, hydrophobic (H) and hydrophilic or polar (P). For example, the sequence of Figure 1(a) can be expressed as hphpphhphpphphhpphph ('1' is indicating the first residue). And for the possible folds generated from the sequence, it can also be placed on a 2D (square) HP-model as showed in Figure 1.

Protein structure prediction (PSP) using hydrophobic (H) and hydrophilic (P or Polar) or HP lattice model was introduced by Dill. It uses a simplified version of the amino acid sequence having only two types of monomers, namely 'H' and 'P', and the chain is placed as a self-avoiding-walk (SAW) on this lattice path. Search using this model looks for the valid conformation (i.e., SAW) which has the maximum number of topological neighboring (TN) (Figure 1) of H-H contacts, where the Hs are not already covalent bonded (or sequentially connected) within the amino acid chain or sequence.

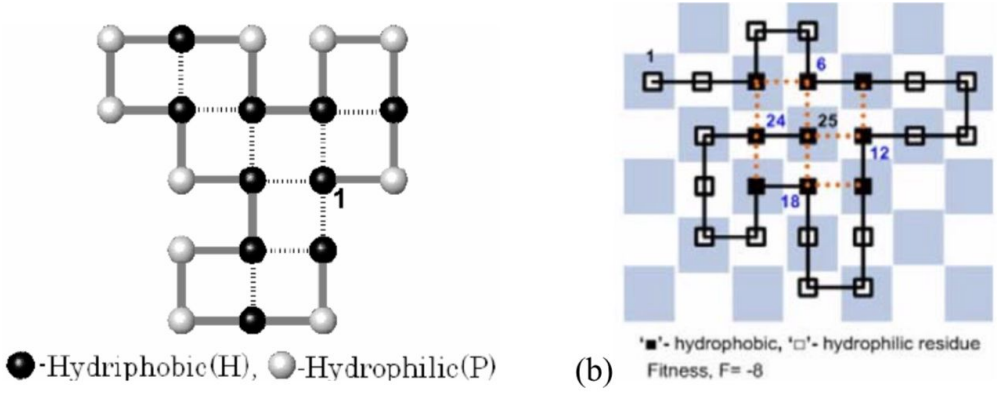


Figure-1: Conformation in 2D HP Model shown by solid line. Dotted line indicates TN. (a) Fitness = -(TN Count) = -9. (b) Fitness = -(TN Count) = -8.

For the given HP-model, we will follow a fitness function to find the best fold on a 2D HP model, which can be expressed by the following simplified energy matrix:

	H	P
H	-1	0
P	0	0

Assuming amino-acid sequence can be given as $\boldsymbol{S} = S_1, S_2, S_3, \ldots, S_m$. and a conformation (structure) $\boldsymbol{c}$ out of that is searched such that $\boldsymbol{c}^* \in C(S)$, energy $E_{min} = \min \{(E(c) \mid c \in C\}$. Here, $\boldsymbol{m}$ is the total number of amino-acid or residue in a sequence and $C(\boldsymbol{S})$ is the set of all valid (i.e., SAW) conformations of $\boldsymbol{S}$. If the number of TNs in a conformation $\boldsymbol{c}$ is k then the value of $E(c) = -k$ which is regarded as fitness function and expressed as $F=-k$.

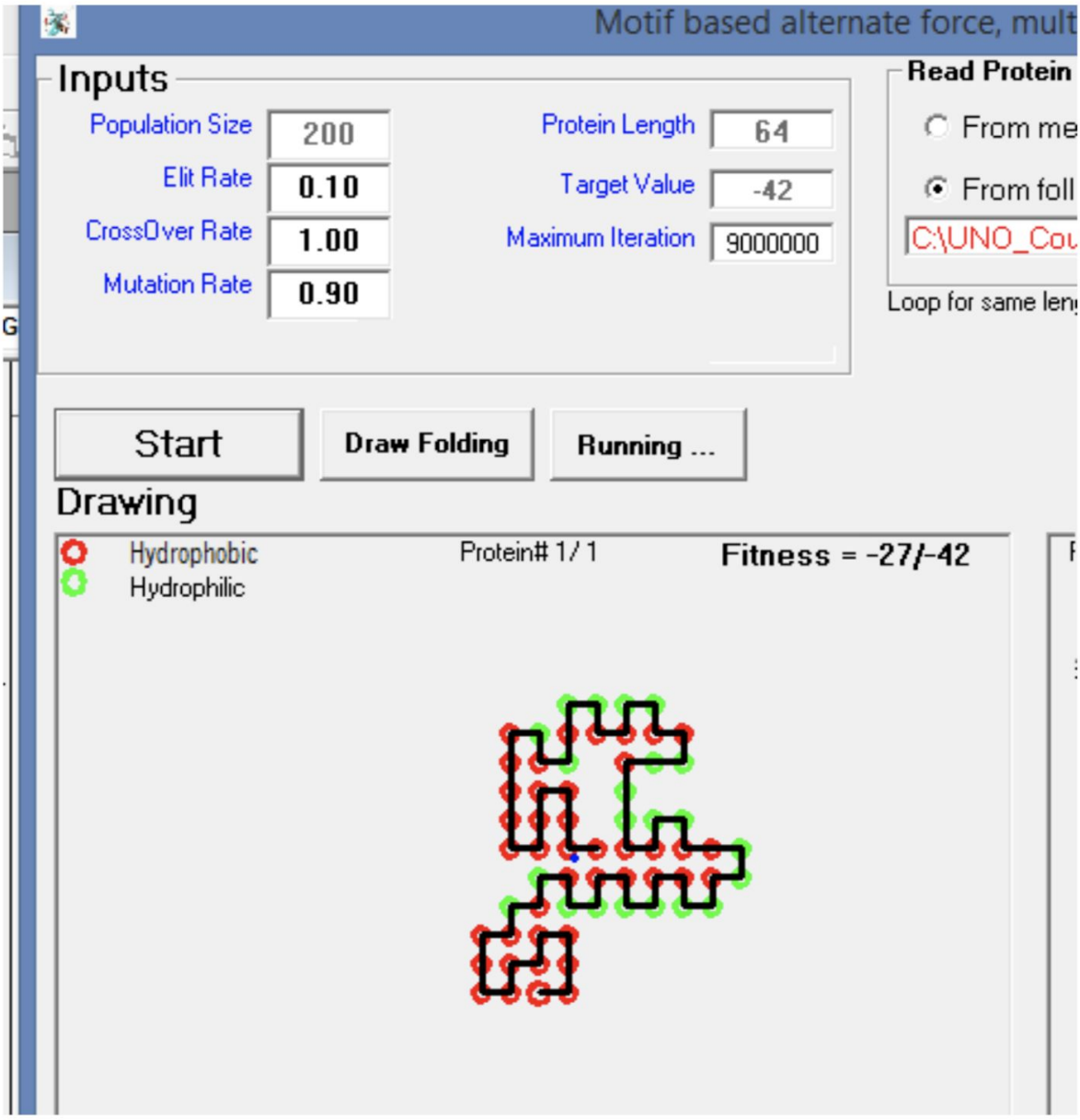
To Do:
You need to develop a Genetic Algorithm (GA) based structural search algorithms. The algorithm for given sequences will search for the best conformation or will go for minimum Energy conformation, and it will visually show or draw the best conformation /structure found in each generation along with the computed fitness value.

- Submit program code and data such a way so that it can be run to check and verify the result visually. Describe,' How to run your code', in your *run_readme.txt* file. Please, avoid asking to install (programming) package to run your program, rather provide executable(s).

- Well commented programming code will score high.

Input: A sample input file (input.txt) is provided for the problem sequences along with their best fitness found. Your program will read one problem sequence at a time, go for searching the corresponding solution structure. Once the termination is meet, the program will select the next problem sequence and search for the corresponding structure, and so on. The program terminates when it is done with all the sequences in the input file.

Sample (GUI) Output for Part#2



II. Approach

Genetic Algorithm:
A simple genetic algorithm (with both MAX iterations and break conditions) will be used to evolve a solution to the protein folds.

Basic Algorithm:

1. Initialization (Pop1)
2. Compute Fitness (Pop1, for all chromosome C_i)
3. Sort(Pop1)
4. Examine: C1/Progress or Max_gen, Exit()?
5. Elite(5 to 10%); Pop2 ← Pop1
6. Crossover (~80%); Pop2(remaining) ← Pop1
7. Fillup Pop2 randomly (remaining, if any)
8. Mutation (Pop2(Non_Elite) ← Pop1); 5% to 50%
9. Pop1 ← Pop2; gen = gen+1
10. Go to Step #2

Algorithm Explanation for this Project:
Step 1: Initialize a population of random protein folds
Step 2: Calculate the fitness for each of the possible folds for comparison
Step 3: Sort the folds from highest fitness score to lowest.
Step 4: Check for exit based on if a fold exists meeting target value or if at max generation
Step 5: Otherwise, a new generation: carry over (%) elite proteins directly to the next population.
Step 6: From the remaining non-elite population, perform crossovers operations for 80%
Step 7: For the remaining amount, generate new random folds and add to population.
Step 8: Then perform mutations on a % of non-elite population only.
Step 9: Update the current population with the new population & increment generation counter
Step 10: Go to Step #2

Technology & Implementation:
JavaScript selected to implement the project for its portability. As such all testing and executing of this application is made trivially easy for any computing device with a browser. The remainder of this section will detail the approach toward implementation.

III. Protein Modeling

The first step toward implementing a genetic algorithm for this solution is to model the protein structure. A fundamental component to this problem is how the protein will fold in 2d space. Thus a 2d coordinate system must be maintained. This solution makes use of two user-defined javascript classes: Point and Protein.

class Point

```
class Point {
    constructor(x,y){
        this.x=x;
        this.y=y;
    }
    toString(){
        return `(${this.x},${this.y})`
    }
    equals(other){
        return this.x === other.x && this.y === other.y;
    }
    getNeighbor(direction){
        switch(direction){
            case 'left':  return new Point(this.x-1,this.y);
            case 'right': return new Point(this.x+1,this.y);
            case 'up':    return new Point(this.x,this.y+1);
            case 'down':  return new Point(this.x,this.y-1);
        }
    }
    isNeighbor(other){
        let result = false;
        if ( other.equals(this.getNeighbor('left'))) result = true;
        else if ( other.equals(this.getNeighbor('right'))) result = true;
        else if ( other.equals(this.getNeighbor('up'))) result = true;
        else if ( other.equals(this.getNeighbor('down'))) result = true;
        return result;
    }
}
```

The Point class is responsible for maintaining the 2d position for each amino amino acid in the protein sequence. The Point class has a constructor that takes in a parameter an x,y coordinate and stores it. A point only needs 4 methods: toString, equals, getNeighbor, isNeighbor.

toString(): displays the state of the point as text for debugging purposes.
equals(): allows for equality operations between two different points based on position.
getNeighbor(): returns a new Point object with coordinates in the given direction from this point
isNeighbor(): given a point, reports true or false if another point is adjacent to this one.

class Protein

```
class Protein{
    constructor(sequence){
        this.sequence = sequence;
        this.occupied = [];
        this.invalid = [];
        this.path = [];
        this.fold();
        this.calculate_fitness();
    }
    walk()               //detailed later
    fold()               //detailed later
    show()               //detailed later
    calculate_fitness()  //detailed later
    rotate()             //detailed later
    mutation()           //detailed later
    swap()               //detailed later
    getPath()            //detailed later
    updatePlot()         //detailed later
    copy()               //detailed later
    equals()             //detailed later
}
```

The Protein class contains many of the necessary methods that drive this genetic algorithm application. For that reason, its implementation will be broken down into multiple sections of this report.

This first examination will provide and overview of the methods contained within the class and details for the constructor for the class.

constructor: the protein constructor takes in a string sequence and stores it. Additionally to perform the 2d folding, the protein maintains a set of occupied points, and invalid points that lead to "invalid" folds. The protein also records the path from start to end using 'left', 'right', 'up', and 'down' directions. After initializing the state variables, it performs a folding and then calculates its fitness.

IV. Folding Operations

This section will detail the Folding operations that the Protein class must perform.

Operation 1: Random Folding of a Protein
In order to randomly generate a 2d protein fold, there are five methods that are implemented: isOccupied, isInvalid, shuffle, walk, and fold.

```
isOccupied = (point) => this.occupied.some( (e) => e.equals(point)   )
isInvalid = (point) => this.invalid.some( (e) => e.equals(point)   )
shuffle = (array) => array.sort(() => Math.random() - 0.5);
```

IsOccupied checks a point in the occupied array & reports true/false if that point already within
IsInvalid checks a point in the invalid array & reports true/false if that point already within
shuffle method mixes up the contents of an array.

```
walk(index, checked=null){
    let origin = this.occupied[index];
    let unchecked = this.shuffle( ['left', 'right', 'up', 'down'] );
    if (checked) unchecked.unshift(checked);
    while (unchecked.length > 0){
        let direction = unchecked.shift();
        let nextPoint = origin.getNeighbor(direction);
        if ( this.isOccupied(nextPoint) == false
             && this.isInvalid(nextPoint) == false)
        {
            this.path.push(direction);
            return nextPoint;
        }
    }
    return null;
 }
```

The walk method takes an index as a parameter for a current location in the sequence. From that index, the origin point for this next step will be grabbed from the occupied array. There are four possible directions: up, left, right, down. Those are shuffled into an order to try. While the collection of unchecked steps is not empty, then a direction is pulled out, and request the point to provide that neighboring point. If that point is not occupied and is not invalid then it is added into the path and returned back, otherwise no valid step can be taken.

```
fold(){
    this.occupied.push( new Point(0,0 );
    let index = 0;
    while (index < this.sequence.length-1){
        let next_step = this.walk(index);
        if (next_step === null){
            index--;
            let deadend = this.occupied.pop();
            this.invalid.push(deadend);
            this.path.pop();
        }
        else{
            index++;
            this.occupied.push(next_step);
        }
    }
}
```

The fold method is the main algorithm for protein folding. When a protein is initially folded, the origin point of (0,0) is pushed into the occupied array. Then the index for the current sequence number is initialized to 0. While the index is less than the number of aminos in the sequence, then a new 2d point must be found for the fold. The current index is passed to the walk method to find the next valid step that can be taken. If the next step is null, that means a deadend occurred, and backtracking must happen. To perform a backtrack, the index is decremented, and the previously occupied cell, is instead add to the invalid array. And the step before that is popped off to retake in another direction. If the next step was not null, then the index is incremented and its add to the occupied array.

Operation 2: Evaluate a Fold's Fitness

the calculate_fitness method is relevant towards folding operations as the fold's fitness evaluates how "good" a fold is based on the rules. This method makes use of the other methods already defined within the protein class

```
calculate_fitness(){
   let count = 0;
   //get all H indexes
    const hIndexes = this.sequence.flatMap((amino, i) => amino === 'h' ? i : []);
    // Next go through hIndexes and check for adjacencies and see if neighbor,
    // then make sure its not connected
       for (let i=0;i<hIndexes.length; i++){
           let this_h = hIndexes[i];
           //console.log(hydrophobe)
           for (let j=0; j<hIndexes.length; j++){
               let other_h = hIndexes[j];
               if ( Math.abs(other_h - this_h) > 1){
                   //check neighbors
                   let hydrophobe1 = this.occupied[this_h];
                   let hydrophobe2 = this.occupied[other_h];
                   if (hydrophobe1.isNeighbor(hydrophobe2) ){
                       count++;
                   }
               }
           }
       }
       //console.log(`fitness: ${-(count/2)}`);
       this.fitness = -(count/2);
   }
```

This method initializes a fitness count variable to 0. Then it uses the string sequence to find the indexes for 'h' (hydrophobic) amino acids. Then iterating over all of the 'h' amino acids and comparing them to all other 'h' amino acids, a quick check is made. If the two 'h' aminos have a distance greater than 1, which means they are not bound together and thus may contribute to the fitness score. So if those two hydrophobes are neighbors then the count can be incremented. After checking all 'h' aminos, the fitness score will be half the value, as each edge is counted twice, and the fitness is expressed as a negative value.

V. Genetic Algorithm Operations

Step 1: Mutations

```
rotate(direction, times=1){
        for (let i=0; i<times; i++){
            switch(direction){
                case 'left':  direction = 'up'; break
                case 'up':    direction = 'right'; break
                case 'right': direction = 'down'; break
                case 'down':  direction = 'left'; break
            }
        }
        return direction
    }
```

```
mutation(index){
        let sequence = this.sequence;
        this.invalid = [];              //destroy invalid cells
        this.occupied.splice(index+1); //destroy old occupied cells
        let turn = parseInt(Math.random() * 3)+1; //random number of 90
degree clockwise turns
        let oldPath = this.path.splice(index);
        let newPath = oldPath.map( e => e=this.rotate(e,turn) )
        while(newPath.length > 0){
            let newWay = newPath.shift();
            let next_step = this.walk(index,newWay);
            if (next_step === null){
                //console.log('back step')
                index--;
                let deadend = this.occupied.pop();
                this.invalid.push(deadend);
                newPath.unshift( newWay ); //undo this step
                newPath.unshift( this.path.pop() ); //undo last step
            }
            else{
                //console.log('forward step')
                index++;
                next_step.type = sequence[index];
                this.occupied.push(next_step);
            }
        }
        //console.log("mutated");
    }
```

Step 2: Crossovers

```
swap(index, newPath){
        let sequence = this.sequence;
        this.invalid = [];              //destroy invalid cells
        this.occupied.splice(index+1); //destroy old occupied cells
        let oldPath = this.path.splice(index);
        while(newPath.length > 0){
            let newWay = newPath.shift();
            let next_step = this.walk(index,newWay);
            if (next_step === null){
                //console.log('back step')
                index--;
                let deadend = this.occupied.pop();
                this.invalid.push(deadend);
                newPath.unshift( newWay ); //undo this step
                newPath.unshift( this.path.pop() ); //undo last step
            }
            else{
                //console.log('forward step')
                index++;
                next_step.type = sequence[index];
                this.occupied.push(next_step);
            }
        }
        //console.log("swapped");
    }
```

VI. Genetic Algorithm Considerations

Step 1: Generating a Population

```
const initialize = function(population_size, sequence){
    let arr = []
    for (let i=0; i<population_size; i++){
        arr.push( new Protein(sequence) );
    }
    return arr;
}
```

Step 2: Evaluating & Ranking

```
this_generation.map( protein => protein.calculate_fitness() )
this_generation.sort((p1, p2) => p1.fitness-p2.fitness);

if (fitness <= target_value) break;
```

Step 3: Elite Proteins

```
/get elite count for next gen
 let elite_count = parseInt(elite_rate * population_size)
 let elite_proteins = this_generation.slice(0, elite_count);
```

Step 4: Crossovers

```
//get nonelite count for next gen
 let nonelite_count = population_size - elite_count;

 //get crossover proteins for next gen
 let crossover_count = parseInt(crossover_rate * nonelite_count) / 2 * 2; //even;
       let crossover_indices = selectSamples(this_generation, crossover_count);
       let crossovers = crossover_indices.map(i => this_generation[i].copy() );
       for (let index=0; index<crossover.length; index+=2){
           let protein1 = crossovers[index];
           let protein2 = crossovers[index+1];
           let location = parseInt(Math.random() * protein_length)
           crossover(protein1, protein2, location);
       }
```

```
const crossover = function(protein1, protein2, index){
    let protein1_path = protein1.getPath(index);
    let protein2_path = protein2.getPath(index);
    protein1.swap(protein2_path, index);
    protein2.swap(protein1_path, index);
}
```

```
//roulette wheel selection
const randomChoice = function(probabilities) {
    let rnd = probabilities.reduce( (a, b) => a + b ) * Math.random();
    return probabilities.findIndex( a => (rnd -= a) < 0 );
}

const randomChoices = function(probabilities, count) {
    return Array.from(Array(count), randomChoice.bind(null, probabilities));
}

const selectSamples = function(population, count){
    let max = population.reduce( (sum,protein) => sum + protein.fitness, 0)
    let selection_probs = population.map( e => e.fitness/max)
    return randomChoices(selection_probs, count)
}
```

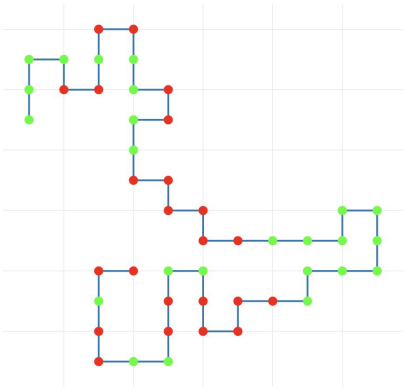
Step 5: New Random Folds

```
//get remaining proteins for next gen
let remaining_count = population_size - elite_count - crossover_count;
let remaining_proteins = initialize(remaining_count, sequence);
```

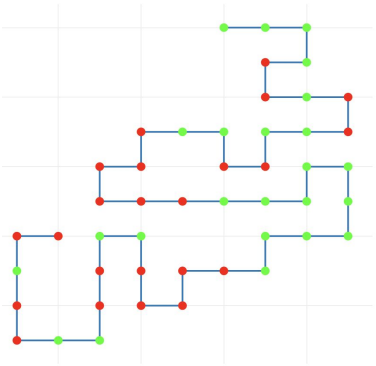
Step 6: Apply Mutations

```
//perform mutations on non-elite
let mutation_count = parseInt(mutation_rate * nonelite_count);
let nonelite_proteins = [ ...crossover_proteins, ...remaining_proteins];
let mututation_indices = new Set();
while (mututation_indices.size < mutation_count){
    let index = parseInt(Math.random() * nonelite_count);
     mututation_indices.add(index);
}
```

Before Mutations (index 27)



After Mutation (index 27)



Step 7: Next Generation

```
//set next generation
let next_generation = [ ...elite_proteins, ...nonelite_proteins]
this_generation = next_generation;
```

VII. Data Visualizations

Step 1: Plots

All visualization for this project is done using Plotly, an open source JS graphics library that is also available on Python and R. All text output is rendered using HTML.

Step 2: Web Workers

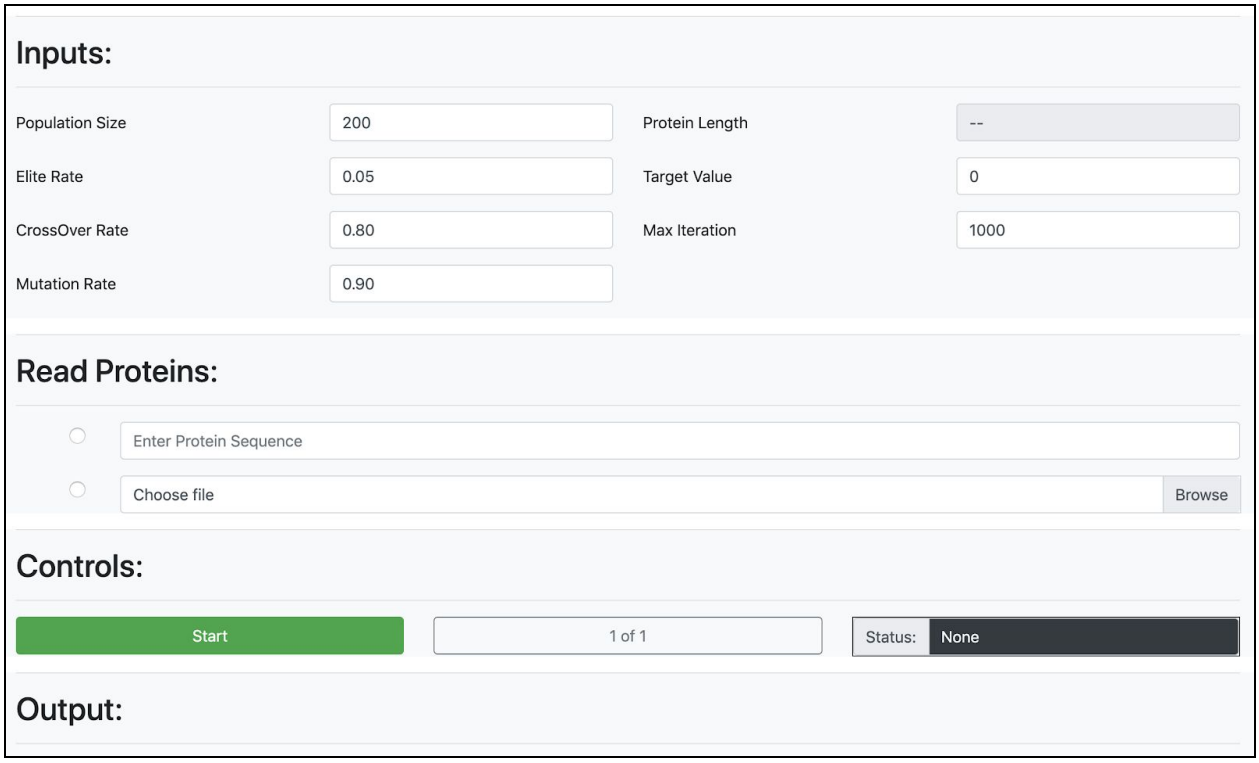
In order to execute high computational algorithms in the browser, it is mandatory to deploy those tasks to web workers, background processes that run a different thread than the main browser thread. Otherwise, the browser will lag and start to timeout. Messages may be alerted to the user than the page has become unresponsive and they may unknowingly may cancel the process. This project implements web workers to prevent this and to provide a better user experience.

VIII. Application Instructions

How to execute the Application:
It is very easy to run this genetic algorithm application. Go to the following url:

https://scalemailted.github.io/MachineLearning2-HW1B/

The following webpage will open:

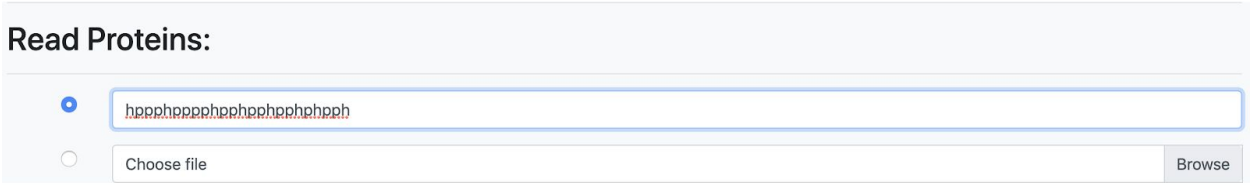


INPUTS:
You may customize the controls for: population size, elite rate, crossover rate, mutation rate, max iteration, and target value.

READ PROTEINS:

Approach 1:

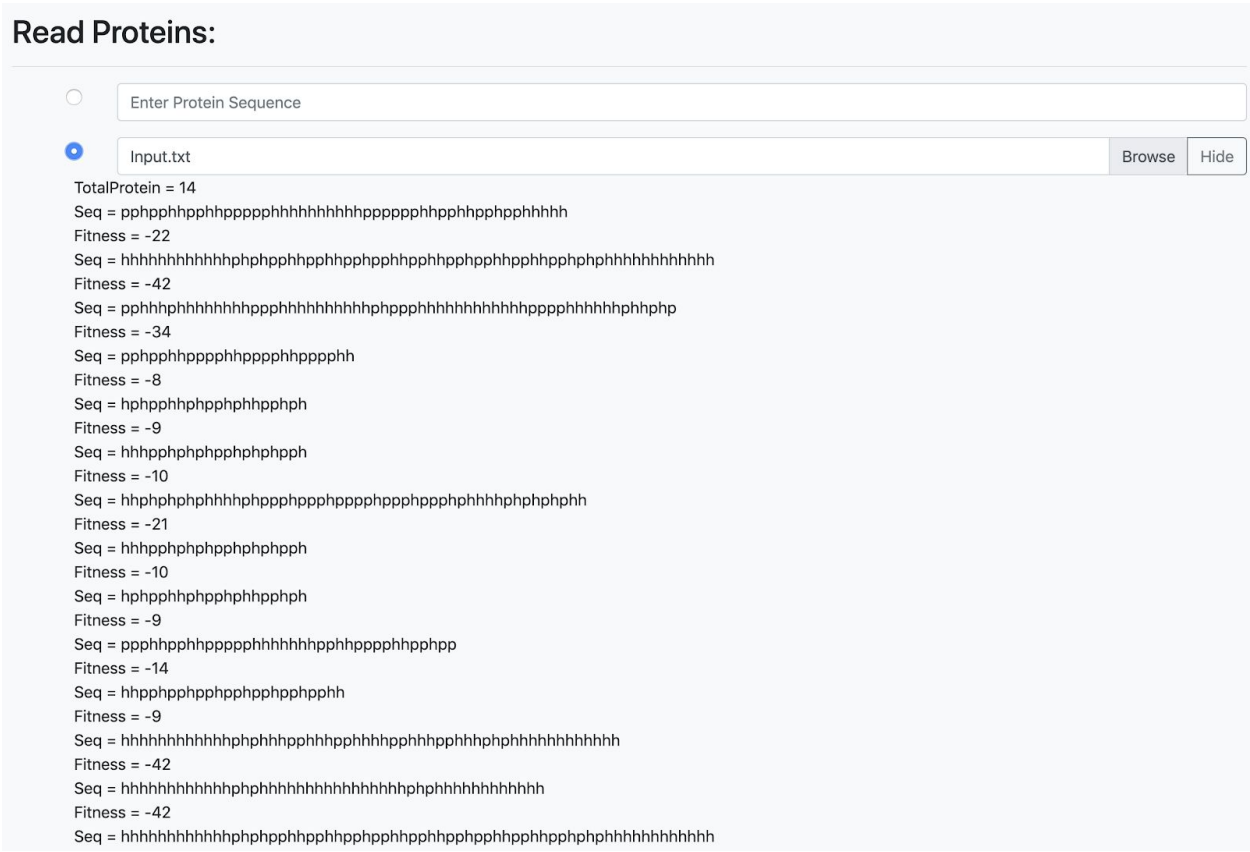
Proteins may either be manually typed into the text field ,& select the radio button, see below:



*Don't forget to set the target value if you manually type a protein

Approach 2:

Load the protein from a file. Note, the file must be formatted as per the following format:



To load a file, click the field and it will load a file explorer for you to select the text file. It will show the contents of the file under the field. You may hide/show the files contents by clicking on the corresponding button. Note: loading proteins from file will also set their targets.

The program will attempt to run the folds on each protein in the text file.

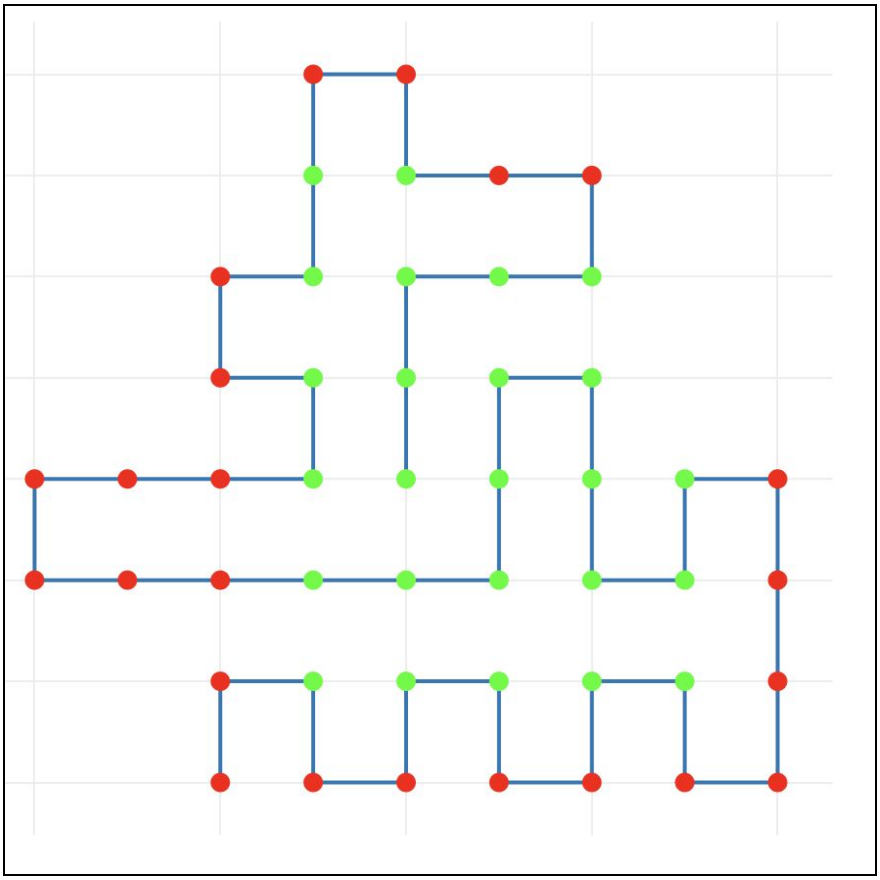
CONTROLS

Once a protein sequence is loaded or manually typed with a target value, you can run the app by pressing the green button. At any time you can stop the app, but hitting the stop button. You can alternate between proteins by stopping and starting the program to skip around. When a protein is loaded from a file and executed, its loaded into the text field above. If you wish to rerun that protein, select the top field and you may toggle to run that protein multiple times.

IX. Results

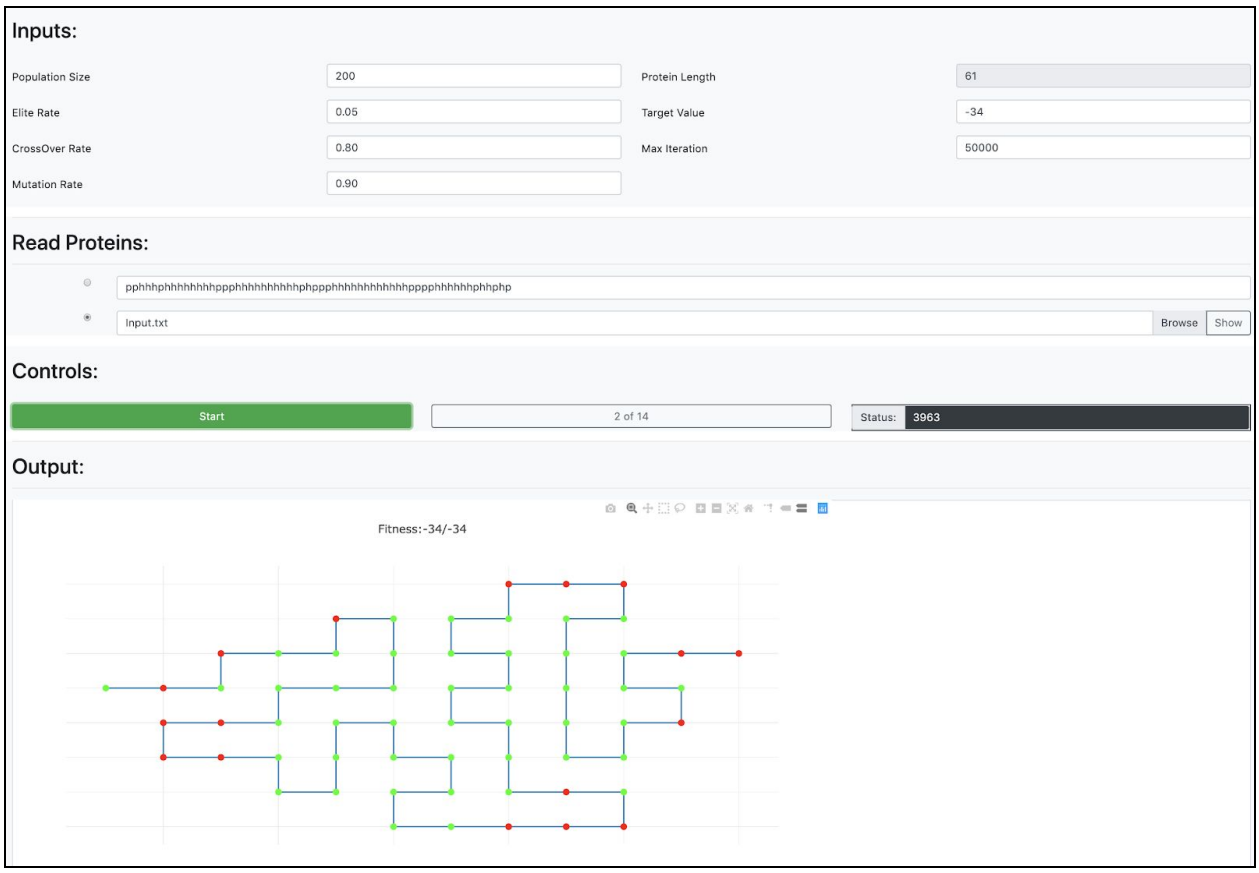
The following screenshots are from some sample runs of the GA application collected over a 1 hour period.

Sequence 0 of 14

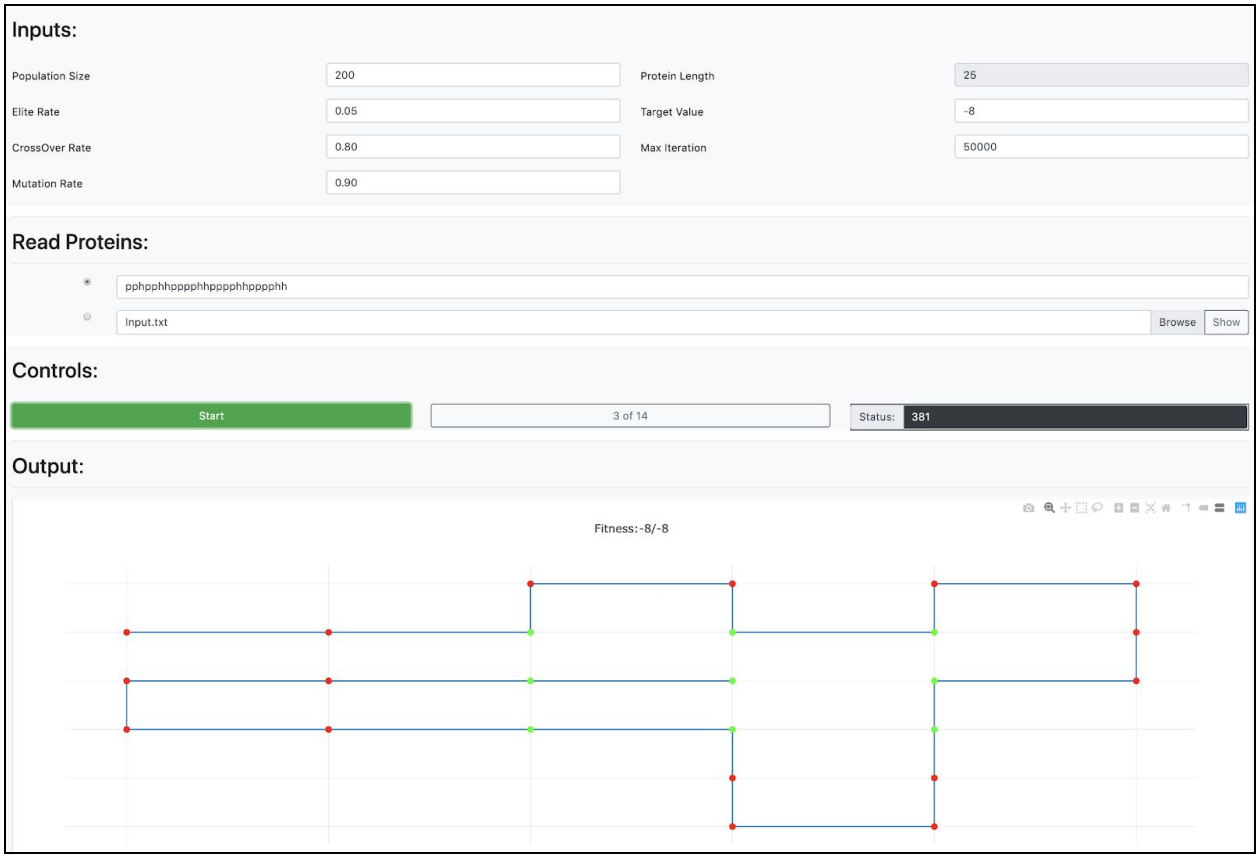


Fitness: -22/-22

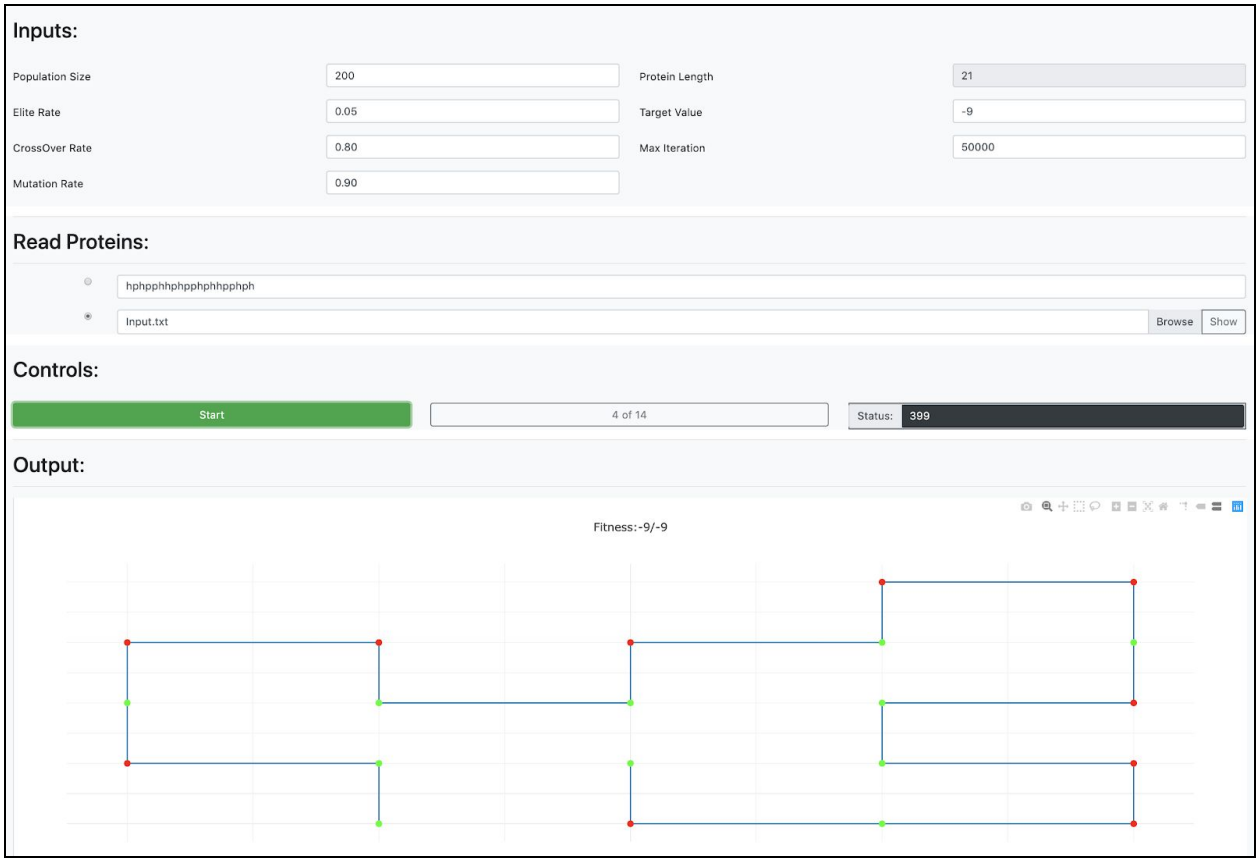
Sequence 2 of 14



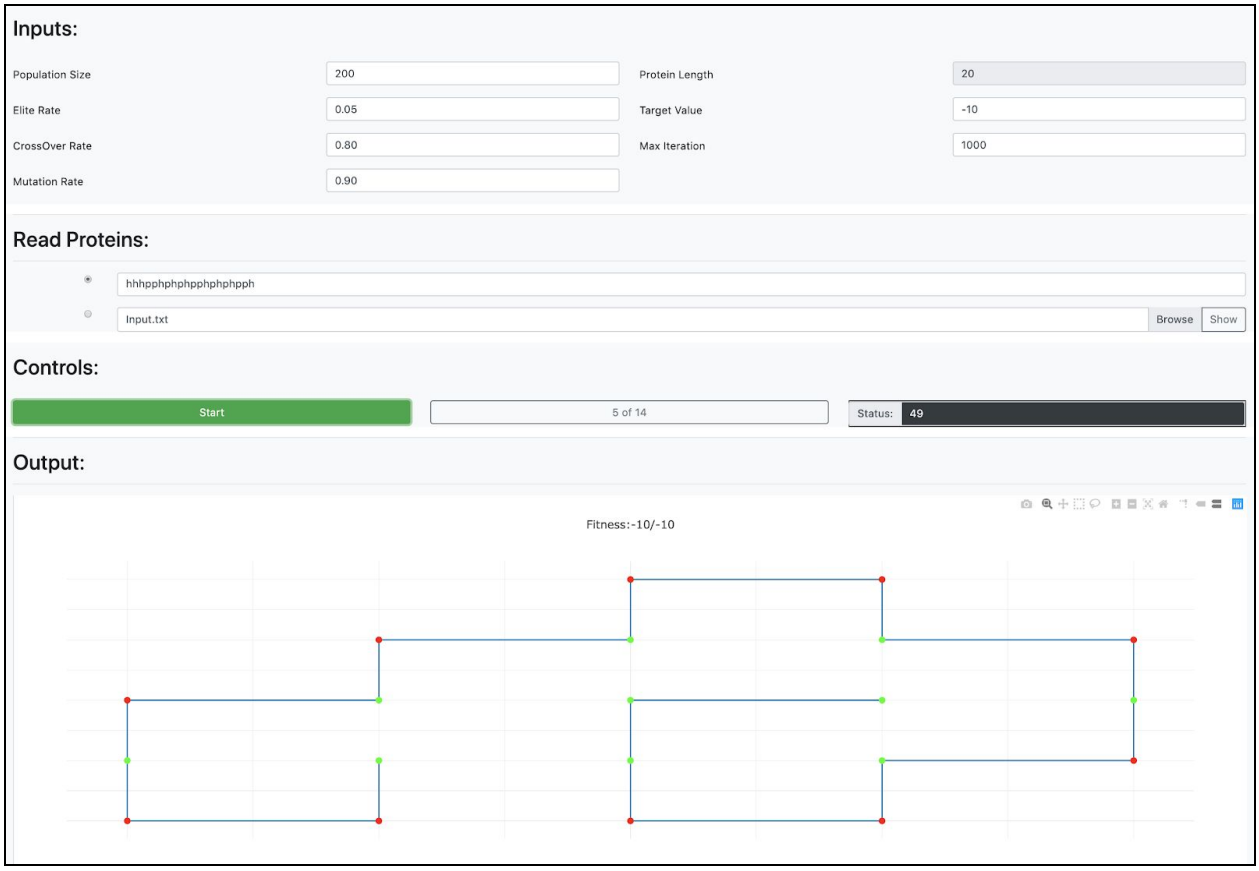
Sequence 3 of 14:



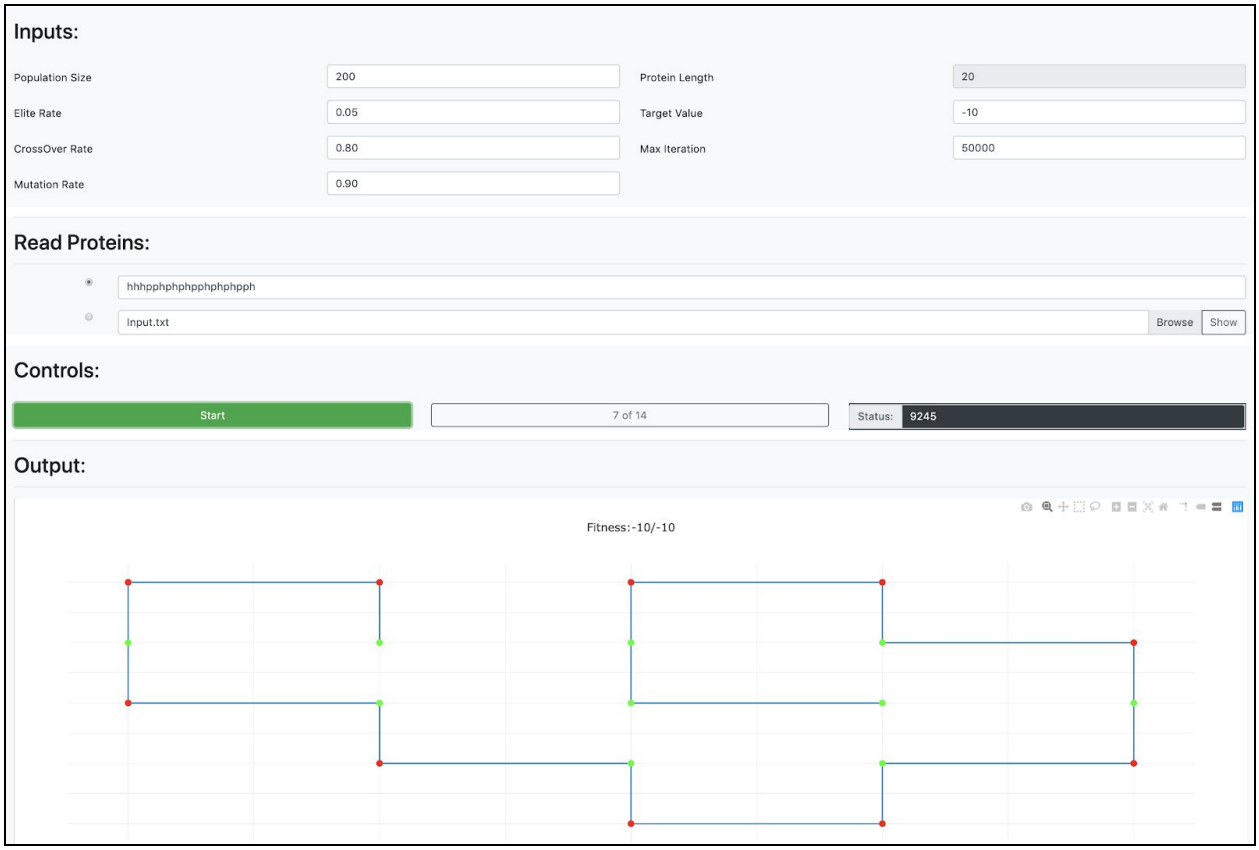
Sequence 4 of 14:



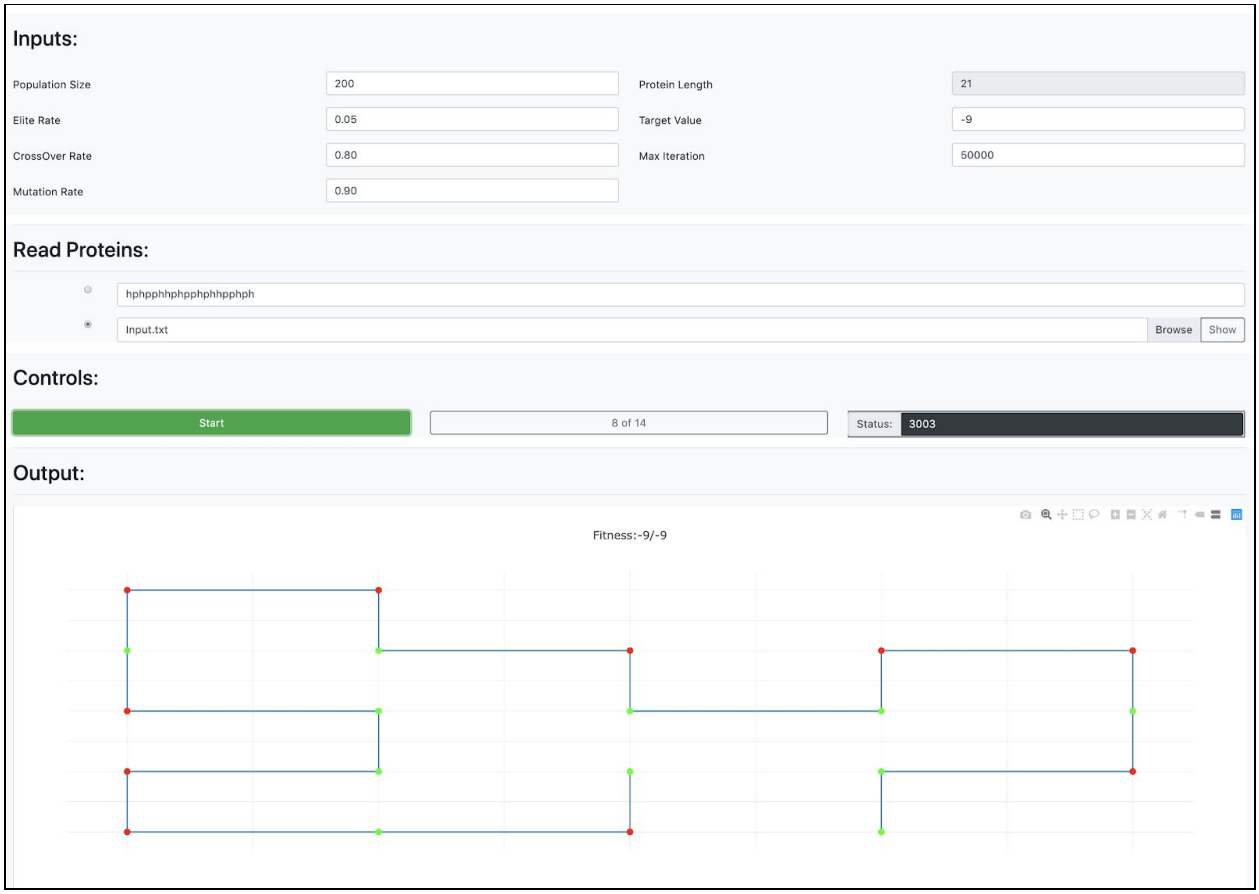
Sequence 5 of 14:



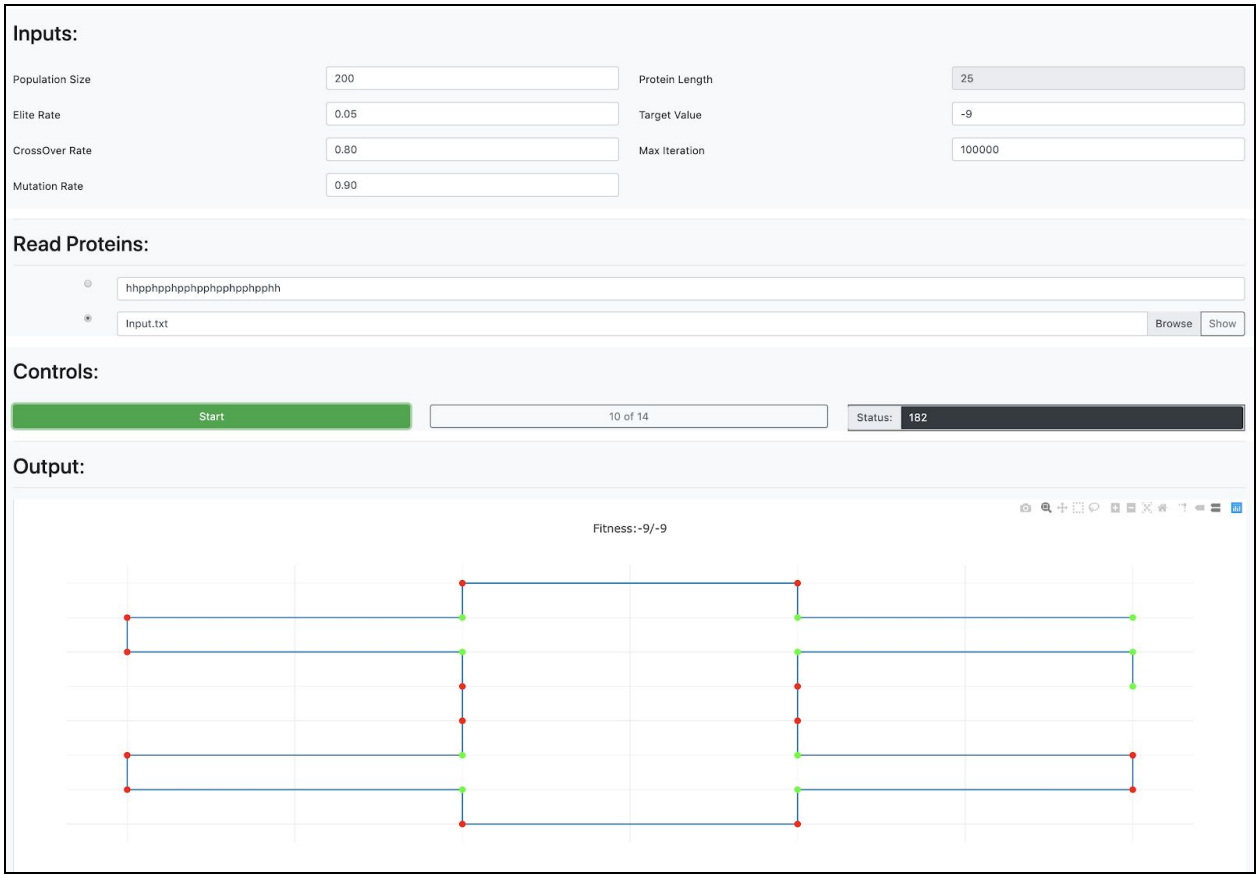
Sequence 7 of 14:



Sequence 8 of 14:



Sequence 10 of 14:



X. Conclusions

For source code:
https://github.com/scalemailted/MachineLearning2-HW1B