CS572

Evolutionary Computation

Project 1 Genetic Algorithm

Heyan Huang

February 20, 2014

Abstract

Steady-state genetic algorithm is applied to six famous difficult-to-solve problems. For this project, I chose the population size of 100, sample size of 10, applied Tournament selection method to select 2 elite parent. In each of my new generation, I kept a version of my elite parent, then 2-point crossover with crossover rate of 0.9, mutated the 2 individuals in each representation dimension with mutation rate of 0.5. And I replaced the worst 2 individuals in previous generation with the mutated 2 individuals. I repeated the process until find good solution. For all the six functions we are able to find the best solution within 2000 generations. Some small tests also indicates that there are strong relationship between generation counts and population size. Beside, the implementation for this project has very good object oriented design, which maximumly minimized the code size and made it easy for debug.

1 Program Design & Data Structures

- In this project, I have implemented Object-Oriented(OO) design as suggested by Dr. Soule.
- Since we have six functions, and each of them also have low threshold value and high threshold value, it will be difficult to handle without a good design. So for this project, they are all defined in the "fitness.h" interface.
- There are three classes, Function, Individual and Population.
- Function object packs the threshold values and fitness functions all together for any of the six functions; Functions are classified into two categories, Separable and inseparable. The essential things be able to packs functions are the two different categories of function pointers which will later be used for calculating the fitness for each individual.
- Individual object sets pointer to Function instance, and also generate initial values, builds solution representation;
- Population object sets pointer to an array of individuals and define the necessary genetic algorithm functions like 2-point crossover, builds new generation.
- the main.cpp program linked all these objects together and solves the problems.

Codes are attached for reference. The files include the following:

- fitness.h
- fitness.cpp
- function.h
- function.cpp
- individual.h
- individual.cpp
- population.h
- population.cpp
- main.cpp
- makefile

2 Algorithm Descriptions

2.1 Genetic Algorithm

In this Project, since I am trying to implement an OO design together with handling pointer, to try to avoid complexity and also considering my laptop CPU limitations, I chose to use the easier to implement Steady-state Algorithm at the cost and risk of slowing down and may not get the expected Results within limited time frame.

My genetic algorithm essential steps are listed as followed:

- Instance a Population object of Population size 100;
- Using Tournament selection method selected sample size of 10 individuals during each generation;
- Compare sample fitnesses and keep the best two elite individuals untouched into the new generation. Trying to speed up the evolution, I replaced the worst two individuals in each parent generation with the 2-point crossover and mutated version of previously found sample elite individuals. Crossover and mutations are done with specific crossover rate and mutation rate.
- I loop through these steps trying to approach the final steady state. Loop will eventually end when best individual gets the fitness less than 0.0001.

The pseudo-algorithm is listed followed:

```

Generate a population of size 100

Loop until best individual reaches threshold fitness value
    record the best individual index, representation \& fitness

    set sample size = 10
    set crossover rate = 0.8
    set mutation rate = 0.5

    loop do

        // pick elite 2 individuals from random sample
        randomly select a sample of 10, save Population indexes as value
        winner Fitness = Sample[0].fitness
        winner index = sample[0]
        second Fitness = winner Fitness
        second index = winner index
        Loop ( sample size - 1 ) times
            temp Fitness =  Sample[i].fitness;
            if temp Fitness better than winner Fitness
                second Fitness = winner Fitness
                second index = winner index
                winner Fitness = temp Fitness
                winner index = temp index
            if temp fitness worst than winner Fitness and
               temp fitness better than second Fitness
                second Fitness = temp Fitness
                second index = temp index
        return struct {winner index, second index} as twoIdx

        // Get worst 2 individual indexes from Population:
        worst Fitness = Population[0].fitness
        worst index = 0
        second Fitness = worst Fitness
        second index = worst index
        Loop population size -1 times
            temp Fitness = Population[i].fitness;
            if temp Fitness worst than worst Fitness
                second Fitness = worst Fitness
                second index = worst index
                worst fitness = temp Fitness
                worst index = temp index
            if temp fitness better than winner Fitness and
               temp fitness worse than second Fitness
                second Fitness = temp Fitness
                second index = temp index
        return struct {worst index, second index} as twoIdx

        // keep elite parent into new generation untouched
        copy parent representations into worst individuals positions
        winfst = sample winner first
        winsnd = sample winner second
        fst = population worst first
        snd = population worst second
        Loop number of dimension p times
            popu[fst].point[i] = popu[winfst].point[i]
            popu[snd].point[i] = popu[winsnd].point[i]
            popu[fst].fitness = popu[winfst].fitness
            popu[snd].fitness = popu[winsnd].fitness

        // 2-point crossover parent with crossover rate
        generate random float number
        if random value greater then crossover rate
            do nothing and end of this step
        else
            generate two indexes within range [1, p-2] and not equal, lowIdx, highIdx
            allocate temporary float memory space of size (highIdx-lowIdx)
            copy second parent middle fragment to temporary space
            copy first parent middle fragment into second parent middle positions
            copy temporary fragment into first parent middle positions

        // mutate the crossover parent with mutation rate
        calculate divider corresponding to generation counter
        extract Function specific low threshold value low
        extract Function specific high threshold value high
        allocate temporary float space for store mutated value
        Loop number of dimension p times
            generate random float number
            if random value less then mutation rate
                generate random value delta corresponding to Function range and divider
                apply increase or decrease to current dimension \&
                store the result in temporary float space
                check, repeat \& make sure mutated value is valid
            copy from temporary space to original dimension value position
        calculate new fitness value for mutated Individual
        repeat above loop one more time for the other crossover individual

        // generate new child Population
        by conduct above three main modules, it is currently new generation already

    End loop
```

2.2 Initial Random Solution Generation

Initial random solution is the solution we randomly generated to start and begin our evolutionary process. According to different function problem requirements, any random start point must have all the valid values for 30-dimension. So as far as the randomly generated points are within the valid value range, and as far as any value in the valid range has equal probability to get generated, which means our random generator can produce any valid value within range with equal probability and has no bias to weaken our tests, we are good to go.

The initial random solutions are coded in the "individual.h" interface, necessary codes are listed below:

```
typedef float (*funPtrSep)(float);       // separable
typedef float (*funPtrNSe)(float*, int); // inseparable

class Function {
 public:
    Function(float l, float h, funPtrSep ptr, bool flag);
    Function(float l, float h, funPtrNSe ptr, bool flag);

    float low;
    float high;
    float fitness;
    bool sepaFlag;

    funPtrSep sepPtr;
    funPtrNSe nsePtr;
};

class Individual {
 public:
    Individual(Function* funptr);

    // functions
    void generate();    // generate an individual
    float getFitness();// calculate the fitness when initialization

    float* point;       // pointer to dynamic array of dimension size p (constant)
    float fitness;      // store fitness value for the point
    float mutRate;
    Function* funPtr;
};

void Individual::generate() {
    int high = (int)((*funPtr).high);
    if (high < (*funPtr).high)
        high = high + 2;

    for (int i = 0; i < p; ++i) {
        if ( (rand()%100/100.0) >= 0.50 )
            point[i] = rand() % high - (rand() % 100000)/100000.0;  // pos
        else
            point[i] = -rand() % high + (rand() % 100000)/100000.0; // neg

        // check if value is within valid range for specific function
        while (point[i] < (*funPtr).low || point[i] > (*funPtr).high) {
            if ( (rand()%100/100.0) >= 0.50 )
                point[i] = rand() % high - (rand() % 100000)/100000.0;
            else
                point[i] = -rand() % high + (rand() % 100000)/100000.0;
            } // while

        }      // for
        fitness = getFitness(); // caculate the fitness for the point
}
```

2.3 Representation

Each solution is represented by a 30-dimension dynamically allocated float array storing the float values for the corresponding dimensions, together with a float value storing the fitness float results for that solution.

Solution representation is defined in "individual.h" interface. Corresponding codes are included as followed:

```
class Individual {
 public:
    Individual(Function* funptr);

    float* point;  // pointer to dynamic array of dimension size p (constant)
    float fitness; // store fitness value for the point
};
```

2.4 Fitness function

Fitness function is the particular objective function that is used to measure and summarise how good the current solution is to achieve the global optimum. For this project, we have six different fitness functions. Each of them is independent of any of the other.

The six fitness functions are represented below:

$$f_{Sph}(x) = \sum_{i=1}^{p} x_i^2$$
$$f_{Ros}(x) = \sum_{i=1}^{p-1}[100 * (x_{i+1} - x_i^2)^2 + (x_i - 1)^2]$$
$$f_{Ras}(x) = 10 * p + \sum_{i=1}^{p}(x_i^2 - 10cos(2\pi x_i))$$
$$f_{Sch}(x) = 418.9829 * p + \sum_{i=1}^{p} x_i * sin(\sqrt{|x_i|})$$
$$f_{Ack}(x) = 20 + e - 20 * exp(-0.2\sqrt{\frac{1}{p}\sum_{i=1}^{p} x_i^2}) - exp(\frac{1}{p}\sum_{i=1}^{p} cos(2\pi x_i))$$
$$f_{Gri}(x) = 1 + \sum_{i=1}^{p} \frac{x_i^2}{4000} - \prod_{i=1}^{p} cos(\frac{x_i}{\sqrt{i}})$$

Since each of our point/solution has 30-dimension, we plug the point array values into these fitness functions specifically, that way we would be able to get the fitness for the solution.

By staring carefully at these six functions, we would be able to differentiate that three of these functions are separable, and the other three can not be. According to this attribute, I have design separate strategies for these two sets.

Separatable Functions

- For separable functions, are fitness function is originally defined at dimension layer, and then at the point layer, I simply sum the fitness of 30-dimension all together to get the point fitness because the function is separable.
- Dimension layer functions are defined in the "fitness.h" interface, and point layer functions are defined in the "individual.h" interface. Mythologically speaking, the point layer definition is simply loop and call the dimension layer functions through all the 30-dimension.

Take Schwefel function as the example, necessary codes are listed as followed:

```
const float schl = -512.0300;
const float schh = 511.9700;

float getSchFitness(float x);

float getSchFitness(float x) {   // dimension-layer fitness
    float y = 418.9829;
    if (x - 0.0 < 0.000000001)
        y = y + x*sin(sqrt(-x));
    else
        y += x*sin(sqrt(x));
    return y;
}

float Individual::getFitness() { // point-layer fitness
    float y = 0;

    if ((*funPtr).sepaFlag)      // if separatable functions
        for (int i = 0; i < p; ++i)
            y += funPtr->getFitness(point[i]);
    else                         // non separatable functions
        y = funPtr->getFitness(point, p);
    fitness = y;
    return y;
}
```

Nonseparatable Functions

- For inseparable functions, I have no other choice but define the fitness measure at the point layer.
- To implement that, I passed in the point representation pointer to float array and the integer length of the array as the fitness function argument.

- In order to maintain these a lot six functions, I would better be able to design some implementation that can help save codes, facilitate the execution steps uniformly.
- As an possible solution that I can come up with, I defined all these detail oriented threshold value constants, dimension layer fitness and point layer fitness all together in the "fitness.h" interface super layer of my Object-Oriented design (OOD).
- Since later on when I mutate individuals, I will still need to have access to the low and high thresholds for specific functions, I think it will be wise for me to pack these function-correspond constants and functions all together to be the "Function" objects.
- In order be able to call these detail layer functions, I typedeffed two different type of function pointers.
- And in my "individual.h" interface, I just need to call the corresponding functions using the right function pointers.

Take Rosenbrock function (inseparable) and Schwefel function (separable) as the comparison, necessary codes are listed as followed:

```
const float rosl = -2.048;
const float rosh = 2.048;
const float schl = -512.0300;
const float schh = 511.9700;

float getRosFitness(float* x, int p) ;
float getSchFitness(float x);

float getRosFitness(float* x, int p) { // point layer fitness
    float y = 0.0;
    for (int i = 0; i < p-1; ++i)
        y += 100*pow((x[i+1]-pow(x[i], 2)), 2) + pow((x[i]-1), 2);
    return y;
}

float getSchFitness(float x) {    // dimension-layer fitness
    float y = 418.9829;
    if (x - 0.0 < 0.000000001)
        y = y + x*sin(sqrt(-x));
    else
        y += x*sin(sqrt(x));
    return y;
}

typedef float (*funPtrSep)(float);       // separable
typedef float (*funPtrNSe)(float*, int); // inseparable

float Individual::getFitness() { // point-layer fitness
    float y = 0;

    if ((*funPtr).sepaFlag)       // if separable functions
        for (int i = 0; i < p; ++i)
            y += funPtr->getFitness(point[i]);
    else                          // if inseparable functions
```

```
        y = funPtr->getFitness(point, p);
    fitness = y;
    return y;
}
```

2.5 Selection

Since two of the most widely used selection methods are covered during class, Tournament selection method and the Roulette wheel selection method. Generally speaking, the Roulette wheel selection method is a weight or rank based selection, so applying this method could potentially easily result in over-selection or under-selection. In over-selection, the best individuals would be able to dominant the offspring quite often. While in situations when all the individuals are equally likely to be selected, it could easily ended up with under-selection.

On the other side, the Tournament selection is completely random. By selecting a certain size of sample from the population, we would be able to compare the fitness among the sample. And we select the best two as the parent of the new generation.

I don't know my population that well yet, I have finished implementations using both methods, but for safety related reasons, I applied the Tournament selection method in my project here. For the detail, for the population size of 100, I used the Tournament method to randomly select 10 individuals as my sample, and among the sample, by comparing the fitness values, I selected the best two individuals as the parent for my new generation.

The selection methods are defined in the "population.h" interface. The complete codes are attached in behind for reference.

2.6 Steady State Algorithm

The most widely used algorithms are generational algorithm and the steady-state algorithm. After the selection step, these two algorithms behave slightly differently.

The generational algorithm keeps the elite parent into the new generation untouched, and by applying crossover and mutation methods on these parent with randomness to produce the rest of the generation. The steady-state algorithm also keep the parent into new generation untouched, but the crossover and mutation on parent happened only once to replace some pair of individuals in the old generation, and keep the rest of the old generation untouched as well to form the new generation.

For implementation, I originally finished the steady-state algorithms with all the functions smoothly, then I tried generational algorithms. But since there are too often crossover, mutation resulted memory swapping, there is no way I can finish my project considering the calculating speed and CPU limitations. So I still ended up with steady-state algorithms to finish this project.

2.7 2-point Crossover

1-point crossover, 2-point crossover and uniform crossover are all covered during class. 1-point and uniform crossover are slightly easier, and I choose the 2-point one.

After having selected the best 2 individuals from the sample, I also have selected the two worst individuals from the population in order to facilitate the evolution process. When I implement the crossover, I firstly copied the parent individuals selected from the sample to the worst population individuals positions, and then I do the crossover on parent individuals on place with a crossover rate float values passed in into the object instance.

For crossover, here are all the details:

1. I randomly generated two integers in the range of [0, 29] inclusive (by "% size") as the index for the two crossover positions in the solution representation. When circumstances like one of the index is 0, or 29, or the two indexes are equal, I repeat my initial generation until indexes are within range [1, 28] inclusive, and the two are not equal. If necessary, I swap the two so that my lowIdx is smaller than highIdx for sure.

```
int lowIdx = rand() % size;
int highIdx = rand() % size;

while (lowIdx == 0 || highIdx == size-1
      || highIdx == 0 || lowIdx == size-1
      || lowIdx == highIdx) {
    lowIdx = rand() % size;
    highIdx = rand() % size;
 }
if (lowIdx > highIdx)
    swap(lowIdx, highIdx);
```

2. I allocated an temporary array of size (highIdx - lowIdx), and I stored the middle fragment of the second parent into this temporary space;

```
float temp[highIdx-lowIdx];
```

3. I copied the middle fragment of the first parent into the corresponding positions of the second parent, so the second parent has the middle fragment originally came from the first parent;
4. I copied the temporary space array into the middle fragment of the first parent so that the first parent has the middle fragment originally came from the second parent;

   Code for step 3 and 4 are listed as followed for quick reference:

```
int fst = winIdx.fst;
int snd = winIdx.snd;
int tmpidx = lowIdx;

// save and swap fragments
for (int i = 0; i < highIdx-lowIdx; ++i) {
     temp[i] = popu[snd].point[tmpidx];
     popu[snd].point[tmpidx] = popu[fst].point[tmpidx];
     popu[fst].point[tmpidx] = temp[i];

     tmpidx++;
 }
```

5. In order to control the crossover rate, I generated an random float value within range of [0, 1] exclusively. If my random value is greater than the crossover float rate, I give up and do nothing in this step; Otherwise, I conduct the above 4 steps.

2.8 Mutation Rate

While crossover provides possible potential recombination of better solutions, we would also need mutation to introduce diversity and variability into the population. The mutation rate is a float value within range of [0, 1] exclusively.

For the implementation, after my 2-point crossover, looping through out the 30-dimensions, for each dimension which means for each element of the solution representation, for each dimension or each element of solution representation, I do the following steps:

1. I generated an random float value within range of [0, 1] exclusively. If my random value is greater than the mutation float rate, I give up and do nothing for the specific dimension, or in other word, for the

specific element of the solution representation. Otherwise, I do the following;

2. I allocated new float temporary space for new mutated dimension value;
3. I generated an random float increased or decreased value delta by dividing the valid range of specific function by a scaling factor which is corresponded to the generation count. Necessary codes are listed below as quick reference:

```
lowThd = (*(popu[0].funPtr)).low;
highThd = (*(popu[0].funPtr)).high;

extern int genNum ;
if (cnt % 1000 == 0)
    genNum *= 10;

for (int i = 0; i < p; ++i) {
    if (rand()%1000/1000.0 < mutRate)
    {
        delta = (float)((highThd - lowThd) / genNum );
        // ...
```

   By scaling down the mutation amount when generation counts increases, I would be able to tune my mutation from originally ruff ones to the near finally refined mutations, which helped a lot with the Schwefel function.
4. I applied the increase or decrease to the original values by 50%-50% chance. My value before mutation is untouched, and I store my mutated value in my temporary space;
5. I checked the mutated dimension to see if the new dimension value is valid. if it does, I am fine; Otherwise, I repeat step 3 and 4 until I get valid dimension value for specific function. Note, I still have my originally dimension values since I have stored my mutated value in another temporary space to preserve originally dimension value.
6. When my mutation is valid, I copied my mutation from temporary space into the originally space so that dimension mutation is done.

I repeat the above steps for all 30-dimension for each of the two crossover individuals.

When both the two crossover individuals are done with these mutations, I am done with the mutation for one steady-state generation. And for the followed generations, for these steps, the mechanism are also similar just like this.

2.9 Produce New Generation

1. As having being stated from previous steps, after I selected 2 individuals as the parent using Tournament selection method, I have also selected the worst two individuals from the population. And I copied my parent into the positions of the worst population individuals so I have an extra copy of my parent in current generation.
2. Then, I 2-point crossover parent on place with crossover rate, and mutation each dimension for each individual of the parent with mutation rate.
3. And correspondingly, I update my two mutated individuals with updated fitness values, so they have refreshed and complete representation.

All my crossover and mutation in implementation are happened on place. So for each generation after the above steps are done,

- I have a untouched copy of my sample elite parent located in originally worst population individuals place;
- On my sample elite index places, I have 2-point crossover with crossover rate, and each individual each dimension mutated with mutation rate and within valid range for each of the six functions;
- All other individuals in original generation are kept untouched into the next generation.

I have done everything I need for the new generation, and it is the new generation already!

2.10 Detail Attention

- Check the range of the mutated dimensions. Each mutated dimension must be within the range of function definition.
- In class constructors, since I have to deal with pointers, constructor does only allocate memory space job. And corresponding, in the destructor, I will have to manually release system resource to delete [] the pointer to float array for Individual::point, Population::weight, Population::popu, Population::idxArray for Tournament selection sample indexes.

- When replace worst Individual with mutated version of elite Individual, since they are pointed by pointer, I will have to conduct deep copy to ensure parent Population and child Population is on the same memory address, which also produces the trouble of frequent memory swap, and potentially increases cache miss rates.
- By OO design, I have included the fitness float variable to record the fitness. So in my Steady-state Algorithm implement, when I deep copy my elite Individual to the worst Individual memory place, I need to deep copy the pointer to 30-dimension point, and I would also need to copy the elite's fitness as well. similarly, when I mutate my elite a little bit within each generation, I would also need to recalculate and update the fitness value accordingly for the mutation happened.

2.11 Generation vs Population Size

I think that the population size and the number of generations are strongly related to each other. But to do a scientific research on this, we need to have some design and test on them. This part is filled out after having done all other parts of the project. So the design and necessary numbers are corresponded to the results came beforehand.

This small sub-project is designed as followed:

- Since steady-state is slightly slower than the generational algorithm, also based on my previous results, instead of using the suggested number of 5000, for my problem, I have used 20000 instead.
- By dividing the 20000 differently I got 8 combinations whose product is 20000, which are 20*1000, 25*800, 50*400, 100*200, 200*100, 400*50, 800*25 and 1000*20.
- Since the initial generation of the population has great randomness, to get a reasonable result, I will have to do repeats to overcome the shortcoming. So for each specific function and each combination, I will repeat 1000 times, and get the average population minimum fitness as the specific function specific combination minimum fitness, and the same method apply to the get the average fitness of the population as well.

- Among all the tests, all other factors are keep the same: sample size 10, crossover rate 0.9, and mutation rate 0.5.
- For each function, I will get the 8 combination results; I plot and get the feeling what's going on.
- I apply the same process for all other functions to remove bias.

For each function, the necessary program codes are listed as followed:

```
int sampleSize = 10;
float mutateRate = 0.5;
float crsRate = 0.9;
int* idxArray = new int[sampleSize];

int popuSize[8] = {20, 25, 50, 100, 200, 400, 800, 1000};
int genCount[8] = {1000, 800, 400, 200, 100, 50, 25, 20};
float minFitness[8];
float avgFitness[8];

int counter = 0;
twoIdx winIdx;
float min, avg;

for (int it = 0; it < 8; ++it) {
    min = 0.0;
    avg = 0.0;
    for (int i = 0; i < 1000; ++i) {
        Population* popu = new Population(popuSize[it], mutateRate, crsRate, indi);
        counter = 0;
        while (counter < genCount[it]) {
            popu->genRanIndi(sampleSize);
            idxArray = popu->idxArray;
            winIdx = popu->tourSelection(sampleSize);
            popu->newGen(winIdx, counter);
            ++counter;
        }
        min += popu->minFitness();
        avg += popu->avgFitness();
    }
    minFitness[it] = min / 1000.0;
    avgFitness[it] = avg / 1000.0;
 }

for (int i = 0; i < 8; ++i)
    printf("%5.5f\t␣%5.5f\n", minFitness[i], avgFitness[i]);
```

3 Results

3.1 General Results

By using a population size 1000, and Tournament selection sample size of 50, crossover rate 0.9, and mutation rate of 0.5, all these functions are performing pretty good. The Sphere function and Rastrigin function can converge within first 1000 generations using the steady-state method. And the other four functions can also converge to very small values within 2000 generations.

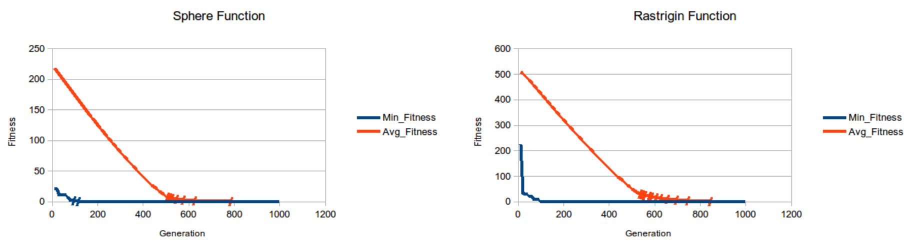

Figure 1: Sphere Function, Rastrigin Function

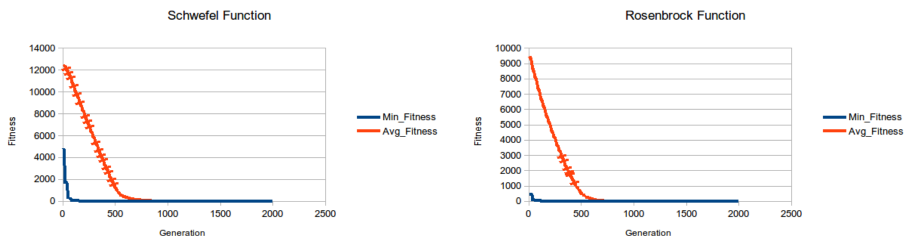

Figure 2: Schwefel Function, Rosenbrock Function

Compared with the Generational Algorithm, Steady-state Algorithm is very slow because during each generation change, only the worst two population individuals got crossover and mutated with certain rates at each step.

But due to the fact that each generation when I generate the new generation, I almost always (because the crossover rate I have used for this project is 0.9) crossover the sample elite parent and mutated them to replace the population worst two individuals. The worst ones were kicked out of the population pretty fast, so it helps to make average fitness converges fast.

The minimum fitness and average fitness plots are included below for each of the six functions.

From the above Figure 1 - Figure 3 plots, we can see that are very identical. In my implementation, all the codes for different functions are all identical except the very basic "fitness.h" interface layer definitions for the fitnesses of different functions and valid value range. So all the plots

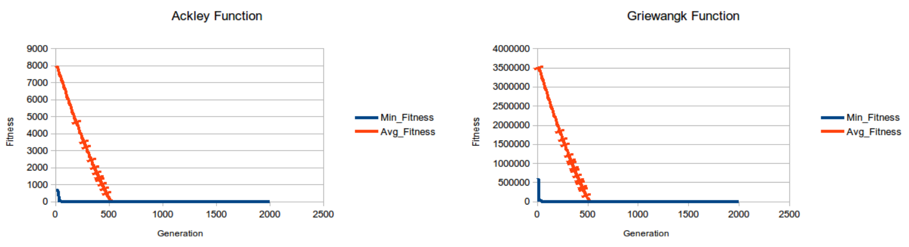

Figure 3: Ackley Function, Griewangk Function

above are the results that I have been expected.

3.2 Generation vs Population Size

Table 1: Sphere Function

Combination pSize*genCnt	Minimum Fitness	Average Fitness
20*1000	2.64207	2.77993
25*800	2.07673	2.18884
50*400	1.29104	1.34800
100*200	0.90835	1.89914
200*100	8.59669	77.07819
400*50	31.56960	187.18013
800*25	51.75002	211.62947
1000*20	56.83261	214.75609

Table 2: Rastrigin Function

Combination pSize*genCnt	Minimum Fitness	Average Fitness
20*1000	36.93418	36.99158
25*800	36.22066	36.26357
50*400	27.90036	27.92589
100*200	20.25126	25.83081
200*100	64.67305	272.61365
400*50	179.31686	468.8251
800*25	238.23174	503.35669
1000*20	253.00603	507.34882

From the above Table 1 - Table 6 and Figure 4 - Figure 6 results, we can clearly see that there are significant relationship between generations and the population size. As more clearly indicated by the Rastrigin function and the Schwefel function than others, the population size of 100 combined with generation count of 200 uniformly performs the better throughout all these six functions.

This result does not necessarily mean that every time we should choose these kind of numbers, but rather simply indicates the relationship exists. Since any other factors are all kept the same, in reality, when we need to

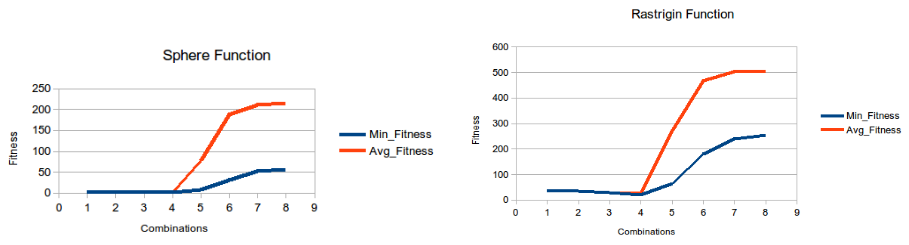

Figure 4: Sphere Function, Rastrigin Function

Table 3: Schwefel Function

Combination pSize*genCnt	Minimum Fitness	Average Fitness
20*1000	1192.64514	1210.27551
25*800	1114.95972	1131.40137
50*400	588.37323	596.22162
100*200	410.74921	714.9552
200*100	2878.34399	8155.80664
400*50	6023.64209	11729.87695
800*25	7186.91455	12343.68164
1000*20	7466.28564	12418.7041

Table 4: Rosenbrock Function

Combination pSize*genCnt	Minimum Fitness	Average Fitness
20*1000	59.4178	170.85864
25*800	54.88237	145.98579
50*400	46.41158	85.99285
100*200	53.25977	112.75067
200*100	175.77638	2932.50635
400*50	592.36963	7677.64697
800*25	1046.46228	9146.99121
1000*20	1187.62634	9352.22266

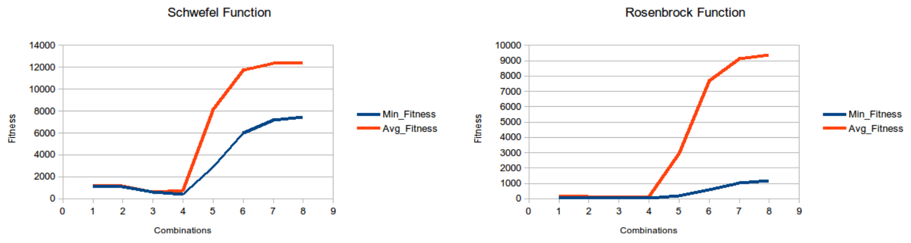

Figure 5: Schwefel Function, Rosenbrock Function

Table 5: Ackley Function

Combination pSize*genCnt	Minimum Fitness	Average Fitness
20*1000	48.39861	48.49332
25*800	44.48375	44.56775
50*400	33.19999	33.24855
100*200	25.11809	36.47882
200*100	211.21599	3018.77954
400*50	1353.1123	7036.66162
800*25	2259.34692	7847.84912
1000*20	2485.90039	7948.56494

Table 6: Griewangk Function

Combination pSize*genCnt	Minimum Fitness	Average Fitness
20*1000	92.59085	92.59692
25*800	92.11481	92.11967
50*400	91.14037	91.14434
100*200	143.45087	1361.99719
200*100	109380.66406	1371220
400*50	660936.5	3105995.25
800*25	1043758.25	3453765
1000*20	1132902.75	3496705.5

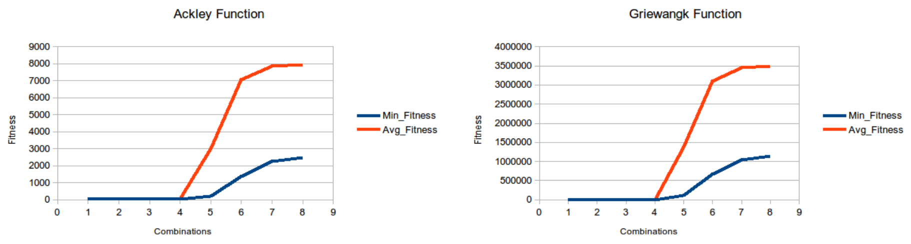

Figure 6: Ackley Function, Griewangk Function

choose reasonable as well as good and efficient numbers, we should conduct similar pretests to tune the factor values a little bit before we start.

4 Conclusions

- The steady-state genetic algorithm that I have used in this project performs pretty good to solve some problem that is difficult for regularly-based methods. Compared with project 1a and 1b, by using hill climbing, we could be possibly blocked in local optimum. On the other side, the genetic algorithm can always find the global optimum whenever it takes, it will find it. And actually by applying some scaling factor or tricks, it behave actually pretty good, not taking that long, and it's quite fast and efficient.
- But still, steady-state algorithm should be slightly slower compared with generational algorithm.
- There are strong relationship between generation counts and the population size. We don't want population size too small, which could easily result in bias, yet we don't want it to be too large not necessarily slow down the process. Before the initial projects, we should have similar pretests for project design.