Project 1B

Tao Zhang
CS 472

February 20, 2014

1 Descrition of Genetic Algorithm (GA)

Generally, my Genetic Algorithm used Steady-State type that:

- generate a population
- have enough loops
- for each loop
  1. tournament select two parents
  2. copy the gene to 2 children
  3. mutate 2 children
  4. replace two loser (tournament selection) by two children
- finally check the fitness of each individual and find the best

My code for the GA so far has these functions:

- int main(): just a main function, which will present the final result.

```
/*
 * #include <...>
 */

// #define bunch of constant values

int main()
{
  GeneticAlgorithm();

  // print solution
  // print Best fitness

  return 0;
}
```

- double GeneticAlgorithm(): The major algorithm part. It will finally return the best fitness, also pass the reference of the best individual (solution) so far.

```
        /* Steady-State */
double GeneticAlgorithm(double resultA[X_I])
{
  // Generate a population
  for 0 to Population step 1
    // Generate each individual
    for 0 to 30 step 1
      // Take random values in range for each vector
    // Calculate each individuals fitness, put into an array
```

```
  while(count < population*100) // as Dr. Soul suggested
  {
    // select father by Tournament
    // select mother by Tournament

        /* crossover is not need right now,
         * but I just make it as copy the same gene
         */
    // copy father to born child1
    // copy mother to born child2

    // Mutate both children

    // select two losers by Tournament
    // Replace two losers by two children
  }

  /* Finally, check the fitness array */
  bestEval <- 9999(inital);
  for 0 to population step 1
    if fitness[i] < bestEval
      bestEval <- fitness[i];
      best = i  // record the position;

  // copy best solution to resultA

  return bestEval;
}
```

- double evaluate(): just the function to calculate the fitness.
- int selection(): so this a tournament selection function which select a winner/loser (depend on the index) from a sample of N random individuals (N is also a random number). Finally the function returns a int which is the index of that individual among population array.

```
        /* Tournament selection */
int selection(int good_poor, double fitness[POP])
/* good_poor is the index to determe we are select a winner or loser*/
{
  // get a random winner and calc his fitness
  winner <- rand()%POP;
  winner_fitness <- fitness[winner];

  // generate a random N
  N <- rand()%POP ;
  // loop to get the best
  for 0 to N step 1
    temp <- rand%POP
    if(good_poor == GOOD && fitness[temp] < winner_fitness){
    // this is for winner
      winner_fitness <- fitness[temp];
      winner <- temp;
    }else if(good_poor == POOR && fitness[temp] > winner_fitness){
    // this is for loser
```

```
      winner_fitness <- fitness[temp];
      winner <- temp;
    }

    return winner; // the index among population
}
```

- void mutate(): Mutate the children by "creep". For each vecter in child, we have 50percent chance to mutate the value by +/- a small value (I did 0.01).

```
void mutate(double child[X_I])
{
  for 0 to 30 step by 1
    if rand()%10 < 5  // 50% chance
      // 50% + and 50% -
      child[i] += (rand()%10 < 5 ? -1 : 1)*INTERVAL;
}
```

2 Results

GA Sphere	1	2	3	4
Population	100	1000	1000	1000
Mutation Interval	0.01	0.01	0.01	0.001
Running Times(*POP)	100	100	500	100
First Best Fitness	473.867100	509.951500	548.528900	537.843500
Final Best Fitness	0.001300	0.000600	0.000500	0.000009
real time	0m0.062s	0m3.544s	0m17.745s	0m3.548s

I tested 4 situations:

- 100 population with running times as 100 times the amount of population
- 1000 population with running times as 100 times the amount of population
- 1000 population with running times as 500 times the amount of population
- and 1000 population with running times as 100 times the amount of population, but with Mutation interval of 0.001

My codes work very well I think.

3 Concluion

The results are good enough to go. As the population and running times go higher, the more accurate answer we will get. Also, I can narrow my mutation inverval, which can make the final fitness much more accurate.