UNIVERSITY OF NEBRASKA-LINCOLN

String Search using Genetic Algorithm

CSCE 896 Project 2

Dongpu Jin

9/30/2012

Introduction

Genetic Algorithm is a widely used search-based AI technique that is often applied to the field of software engineering. Genetic Algorithm mimics human population evolution process by integrating selection, crossover, and mutation into the algorithm. It iteratively evolves the population and stops until an optimal solution is found. The goal of this project is to apply genetic algorithm to search for a string "AIandHeuristicSE" from a population of all 16 character strings. Experiment results suggest that binary distance fitness function outperforms squared distance fitness function, swap two genes for crossover worsen the performance, and larger population size improves the search. Next section explains the implementation details. Experiment section shows experiment designs and results. The last section concludes the report.

Implementation

In genetic algorithm representation, each character is mapped to a gene and each string of 16 characters is mapped to a chromosome. One generation of population is represented by an iteration of the program. The program stops iterating until it finds the string. The stopping criteria imply that there is at least one chromosome reaches the optimal fitness, which is computed by a fitness function. In this project, two fitness functions are used. One is squared distance fitness function, where the distance of mismatch characters are squared first and then added together. The other one is binary distance fitness function, which simply adds the number of mismatching characters. The optimal fitness is zero for both fitness functions.

A genetic algorithm consists of two important components including crossover and mutation. Crossover lets better chromosomes exchange parts of their genes such that their offspring contain better genes. Mutation randomly alters genes with certain probability such that new information can be introduced into the population. These two steps allow a genetic algorithm to mimic human population evolution and eventually lead to an optimal solution. At the code level, crossover and mutation are implemented as separate functions. The crossover function first sorts the chromosomes based on fitness ranking. Top half of chromosomes are select to be the parents, the rest chromosomes are discarded. Then, offspring is produced by one-point crossover, where a random point is set in two chromosomes and the entire chunks of genes after that point are exchanged. The mutation function randomly picks a character with in a random chromosome and alters it to a different character.

Figure 1 shows the output after program execution for squared distance fitness (left) and binary distance fitness (right). The optimal solution is indicated by a star ('*') in the population. If there are multiple solutions exist in the generation, they all will be

marked. The output also lists statistics including the fitness function, mutation rate, number of iterations, and CPU time.

```
cse cse896/hw2> ./ga
Please select a fitness function:
1. Sqared Distance (default)
2. Binary Distance
1
Please enter mutation rate (0~1): 0.03

Current population:
AIandHeurVsticSE
AIandHeurjsticSE
AIandHNurjsticSE
AIandHeurjsticSE
lIandHeurjsticSE
AIandHeurjsticwE
AIandHeurjsticSE
AIandHeurjsticSE
AIandHeurjsticSE
AIandHeurjsticSE
AIandHeurjsticSE
AIandHxurjsticSE
AIandHeuristicSE*
AIandHeurjsticSE
AIandHeurjtMicSE

Squared distance fitness function (#1)
Mutation rate: 0.03
Number of iteration: 916
Total time: 60ms
cse cse896/hw2>
```

```
cse cse896/hw2> ./ga
Please select a fitness function:
1. Sqared Distance (default)
2. Binary Distance
2
Please enter mutation rate (0~1): 0.03

Current population:
AIandHeurwxticSE
AIandHeurbqticSE
AIandHeurwsticSE
AJandHeurbGticSE
AIandHeuqbsticSE
AIandHeurwsticSE
AIandHeurwstidSE
AIandHeurbsticSE
AIandHeurwsticSE
AIandHeurbsticSE
AIandHeurwsticSE
AIandHeurbsticSE
AIandHeurwsticSE
AIandHeuristicSE*
AIandHeurbeticSE
AIandHeurwsticSE

Binary distance fitness function (#2)
Mutation rate: 0.03
Number of iteration: 390
Total time: 0ms
cse cse896/hw2>
```

Figure 1 Output after program execution for squared distance fitness (left) and binary distance fitness (right).

Experiment

In order to fully understand the behavior of the genetic algorithm, it is necessary to run various experiment with different sets of parameters. The entire experiment mainly consists of three parts. The first part is to repeatedly run the algorithm using squared distance fitness function and binary distance function at the same mutation rate. The goal is to compare the efficiency and stability of convergence for both functions. In the second part of the experiment, crossover function and the population size are changed one at a time, such that we can understand the impact, in terms of number of generation and CPU time, for each parameter. Finally, the fitness landscape 3D visualizations show the population evolution through the generations. The visualizations provide an intuitive way for developers to understand the overall behavior of the entire population. The rest of this section provides more details about the experiments.

In the first experiment, both squared distance fitness and binary distance fitness are run 10 times each with 0.03 mutation rates. Table 1 and Table 2 list the number of iterations each run takes along with corresponding CPU time.

For squared distance fitness GA, the maximum and minimum number of iterations are 1216 and 445, and the standard deviation is 282.8859. The values for binary distance fitness GA are 677, 216, and 144.3695, respectively. Interestingly, the values for binary distance are about the half of the values for squared distance. Put another way, the binary distance outperforms the squared distance by a factor of 2, in terms of number of iterations. However, when looking at the CPU time, the squared distance is much slower than the binary distance for more than two folds. The underlying reason is the the squared distance fitness function uses C++ *pow()* function to compute squared values, which increases the computation overhead, and thus leads to program slowdown. On the other hand, the standard deviation of binary fitness is about the half of squared distance as well. This implies that the binary fitness is more stable, in the sense of the variations of iterations for each run.

Squared Distance	1	2	3	4	5	6	7	8	9	10
Iterations	683	1199	908	596	445	1216	774	1001	464	594
Time(ms)	40	90	60	50	40	70	50	70	40	40

Table 1 Number of iterations and CPU time for Squared Distance fitness function for five runs. Mutation rate is 0.03. Initial population size is 32. Generation population size is 16.

Binary Distance	1	2	3	4	5	6	7	8	9	10
Iterations	339	415	248	470	216	677	455	527	407	595
Time(ms)	0	0	0	0	0	10	0	0	0	0

Table 2 Number of iterations and CPU time for Binary Distance fitness function for five runs. Mutation rate is 0.03. Initial population size is 32. Generation population size is 16.

Then mutation step is then commented out to see if a genetic algorithm is still functioning correctly without mutation. In this case, there is no random gene introduced after the crossover of two parent chromosomes during each generation. When running the program, the fitness starts to drop at the beginning. Then the fitness gets stuck at a certain value and never moves forward. As a result, the program never converges successfully. Therefore, both crossover and mutation are required in order to make genetic algorithm to work correctly. In addition, it would be interesting to comment out both the mutation and crossover steps to see the behavior. This essentially turns the genetic algorithm into a random algorithm, which generates random chromosomes at each generation until it finds the solution. However, when running the program, the fitness values become completely random at each generation. It does not show any trends of convergence. Therefore, it is fair to conclude that genetic algorithm outperforms random algorithm.

Next experiment shows the impact of crossover in terms of reliability of convergence. During each generation, the original one-point crossover function randomly finds a point from parent chromosomes and swaps the entire chuck of genes after that point. In this experiment, the crossover function is replaced by a customized version, which swaps only two random genes of two chromosome parents. Various program runs suggest that the maximum mutation rate in order for the program to successfully converge in a reasonable amount of time is 0.007 for squared distance fitness function and 0.019 for binary distance function. Table 3 and Table 4 list experiment results. As we can see, both number of iterations and CPU time increase dramatically. This is because swap two genes rather than entire chunk essentially reduce the diversity of the population. All of the chromosomes are very alike, and thus lead to slow evolution.

Squared Distance	1	2	3	4	5
Iterations	658693	190766	8394	344517	98872
Times(ms)	18980	5790	400	10400	3020

Table 3 Number of iterations and CPU time for Squared Distance fitness function for five runs. Mutation rate is 0.007. Initial population size is 32. Generation population size is 16.

Binary Distance	1	2	3	4	5
Iterations	1999576	858250	4228996	4054609	2407337
Times(ms)	16050	7330	32620	32450	19140

Table 4 Number of iterations and CPU time for Binary Distance fitness function for five runs. Mutation rate is 0.019. Initial population size is 32. Generation population size is 16. Different population sizes:

Squared Distance	1	2	3	4	5
Iterations	132	100	113	116	181
Times(ms)	280	230	200	240	300

Table 5 Number of iterations and CPU time for Squared Distance fitness function for five runs. Mutation rate is 0.03. Initial population size is 512. Generation population size is 256.

Binary Distance	1	2	3	4	5
Iterations	54	35	25	20	28
Times(ms)	30	20	20	10	10

Table 6 Number of iterations and CPU time for Binary Distance fitness function for five runs. Mutation rate is 0.03. Initial population size is 512. Generation population size is 256.

This experiment manipulates the size of population to explore the impact of population size on algorithm performance. The initial population size increases from 32 to 512, and the generation population size increases from 16 to 256. Table 5 and Table 6 show experiment results. As we can see, the number of iterations decreased dramatically. Intuitively, the more candidates to select, it becomes easier to find a solution. In contrast with number of iterations, the CPU time increases significantly. The program is slowed down due to the fact that GA needs to compute fitness for all the extra populations, and thus increases the computation overhead.

Visualization of the population fitness landscape allows developers to view the overall population evolution intuitively. In the following experiment, 3D visualizations of individual chromosome fitness with respect to generations are plotted using Matlab. The X axis represents individual chromosome, the Y axis represents generations, and the Z axis represents fitness for individual chromosome. With population of 256 chromosomes, binary distance converges with less than 50 iterations and squared distance converges with less than 140 iterations. Comparing to around 600 iterations for both algorithms with population of 16, they are quite significant improvements. As we can see, the program converges faster with larger population, in terms of number of iterations, which also matches the conclusion we drew from the previous experiment.

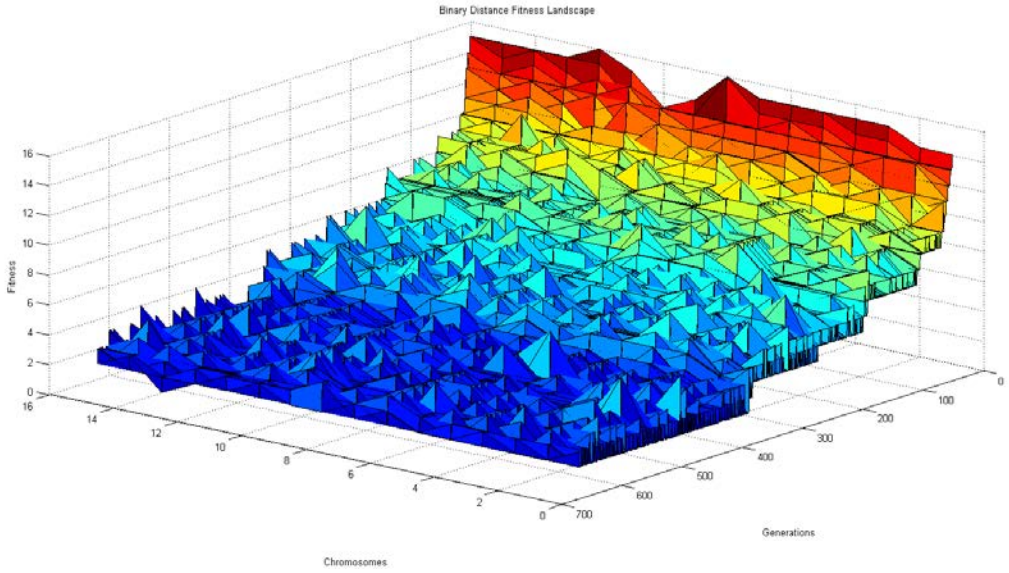

Figure 2 Binary Distance Fitness Landscape. Mutation rate 0.03. Population 16.

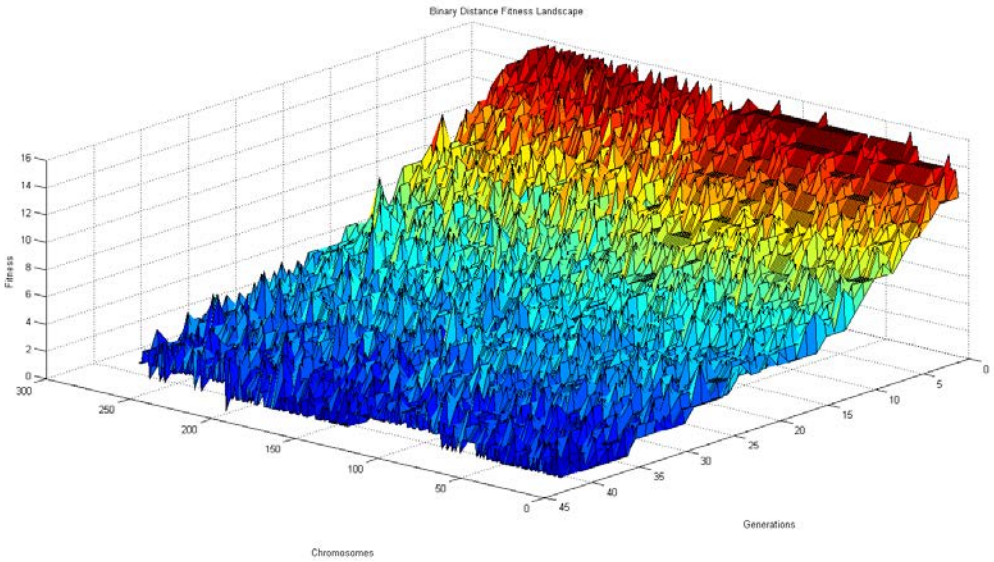

Figure 3 Binary Distance Fitness Landscape. Mutation rate 0.03. Population 256.

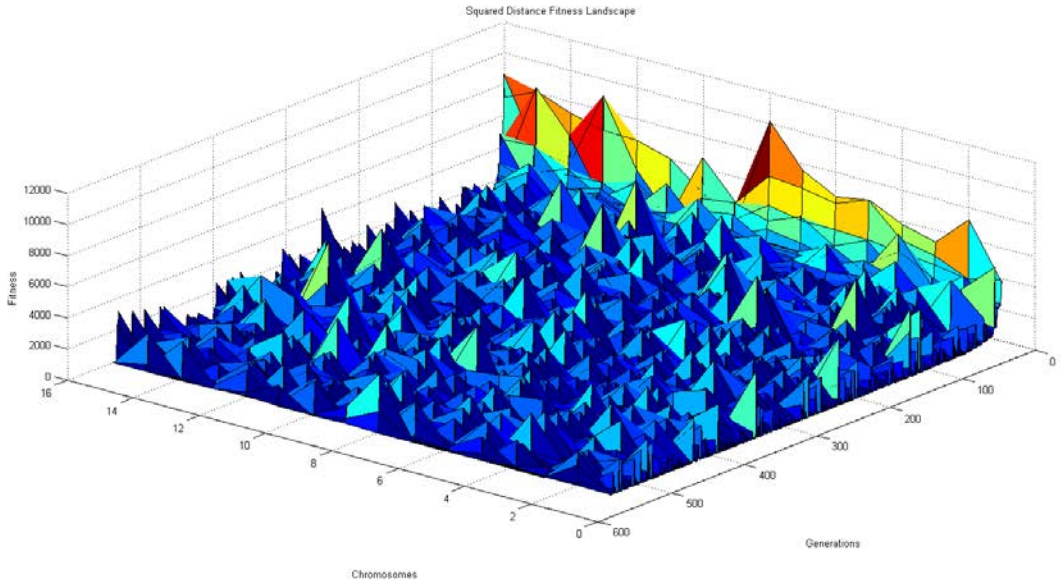

Figure 4 Squared Distance Fitness Landscape. Mutation rate 0.03. Population 16.

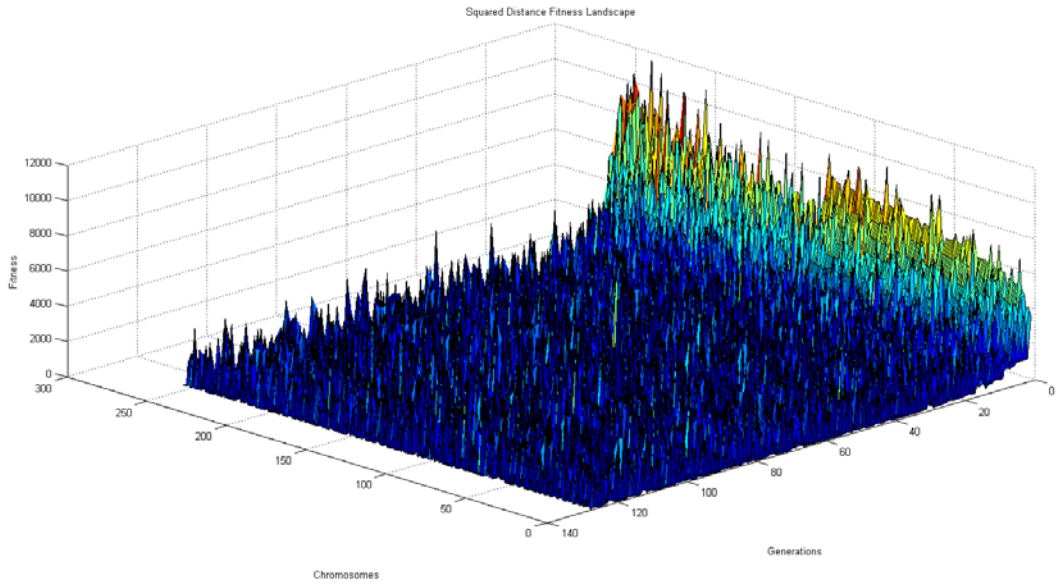

Figure 5 Squared Distance Fitness Landscape. Mutation rate 0.03. Population 256.

Conclusion

In this project, a genetic algorithm is developed to search for a string. Two fitness functions including squared distance and binary distance are compared against each other. Comparison results show that binary distance outperforms squared distance by a factor of two. In addition, crossover function and population size are manipulated. The results suggest that swap two genes for crossover worsen the performance while larger

population improves the search. Finally, fitness landscape is visualized using 3D plots. The 3D visualization provides developers with an overall view of the population evolution.