Using Genetic Algorithms to solve complex optimization problems

New Mexico

Supercomputing Challenge

Final Report

April 4, 2012

Team 68

Los Alamos High School

Team Members:

Alexander Swart

Teacher:

Mr. Goodwin

Project Mentors:

Pieter Swart

Executive Summary

The goal of this project is the development of fast genetic algorithms that can solve both continuous and discrete optimization problems. There are usually a few ways to solve any problem. In a brute force solution, the program attempts to solve the problem using a randomly generated data set. This method is very easy to implement but usually very inefficient and can even fail completely. Genetic algorithms allow one to solve some problems more efficiently. The basic idea behind a genetic algorithm is based on natural selection. My genetic algorithms initially generate a random population. Then the initial random population is subjected to the specific task. Depending on how well they do at that task , they are included or discarded. Those that are included are incorporated into a new population. The program combines two or more parent elements to form a new population. The program does this until the new population is the same size as the original population. Finally a small percentage of mutation is added to the new population. The mutation process helps to avoid from becoming stuck in a 'divot' or a local minima other than the intended minimum. This process is repeated many thousands of times, preferably on multiple threads. I developed and optimized GAs for both continuous and discrete problems. In order to further increase the speed of my program, I wrote genetic algorithms for each piece of code, and used C-optimized Python. I used two different approaches for each problem: a random search algorithm, and a method using GAs. In each problem, the GA-based algorithms vastly outperformed any random search algorithm.

Background

Genetic Algorithms (GAs) provide a powerful approach to complex problems involving optimization or design. In the absence of a clear analytical approach, GAs work especially well with complicated solutions, numerous unknowns, many constraints, and large solution sets. A brief description of a GA is as follows. The optimization problem is modeled by an artificial world populated with organisms. The optimization is characterized by a fitness function. Each organism has an artificial genotype or data set that mimics real-life DNA. A process of natural selection eliminates organisms (solutions) that score poorly as measured by the fitness function. A process that acts like reproduction combines DNA, or mixes the genotype of two or more organisms. A mutation-based process adds small random changes to the best solutions. Mutations can be positive or negative; positive mutations benefit the organism and negative mutations harm the organism. Mutations are essential to the algorithm in that they assist in ensuring that organisms do not get stuck in a local dip in the energy landscape.

Problem Statement

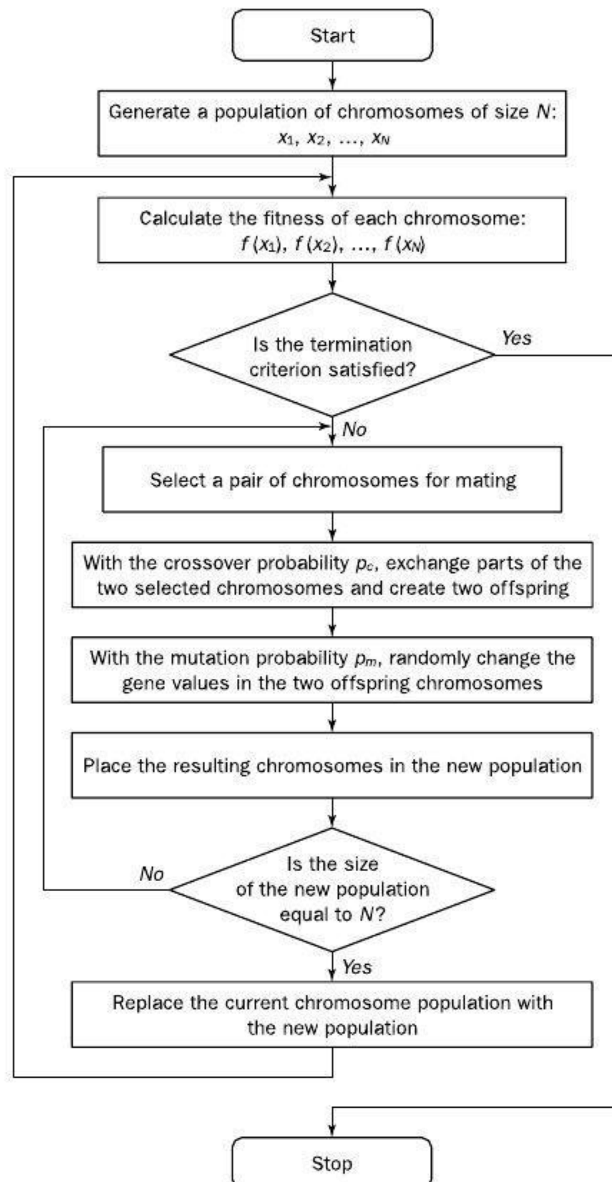
The original goal for this project was to simulate protein folding. However, I soon realized that this was on the scale of a two year project. Instead I choose a new subject; using genetic algorithms to solve complicated optimization problems. Genetic algorithms help solve many problems in all fields of science. However, while they can solve problems, they are even better

at optimizing the solutions to a problem. In a scenario in which you need to design the most aerodynamic car, for example, you would use a genetic algorithm. It would take the original design and develop a more efficient solution. Since genetic algorithms are used in such a variety of fields, it would be very worthwhile to improve on the general theory. During the tour at Sandia National Laboratory I learned that the power for their supercomputers cost them millions of dollars annually. Since time is money, if one could improve on the algorithms used on supercomputers to solve problems, these improvements would save millions, if not billions of dollars. In addition, just by increasing the efficiency of you solution algorithms, it would allow for an increase in computing power, decreasing the need for larger, more expensive computers and allow more discoveries in science and medicine.

In order to understand how to improve our computing situation, we must first gain knowledge on how to write efficient GAs. GAs take a population, run it through the required task (such as a function), return the result, and then generate a new population. In order for the new population to avoid becoming stuck in something like a local minima, it is important to incorporate a degree of 'mutation' into your results. A simple way of explaining how GAs work is that they attempt to find the lowest point in a landscape. For example you want to find the lowest point in a golf course. A GA will plop down in some random spot, roll straight down to the lowest point while taking a look around. If it sees a lower landscape, it will eventually 'jump' to the spot. This jump is facilitated by mutation.

However, GAs do have some disadvantages. For example, code complexity can be an issue, especially if you write you own like I did. Also, it is possible that you can have round-off errors,

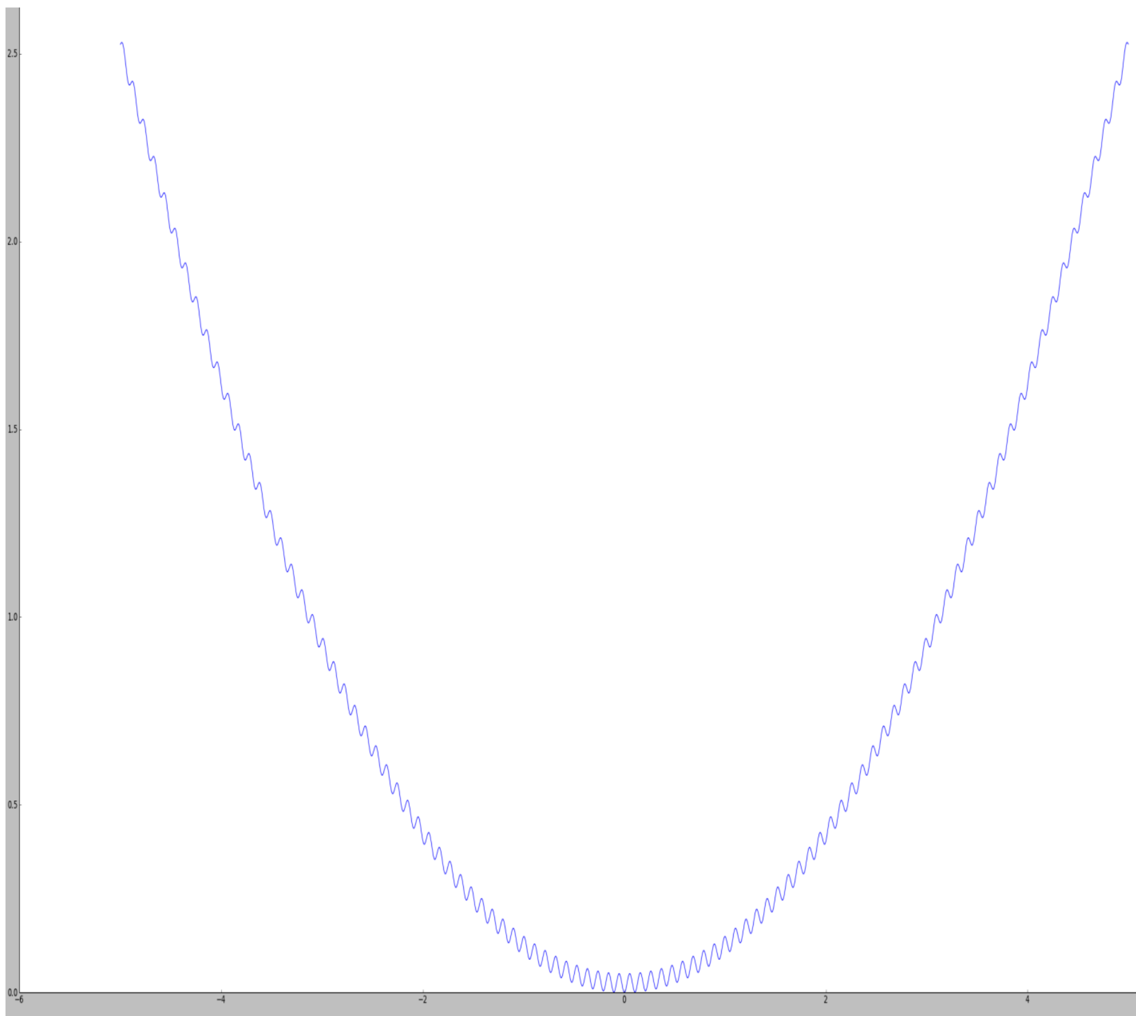
however, this is a chance with any program. The purpose of my project is to aid the development of more efficient GAs. I hope to be able to write my own GAs capable of solving intensive problems.



Problem Solution

My simulations were based in Python, and I used an environment called IPython plus the Enthought Python Distribution to help visualize my results. Although the Python implementation was slightly slower than other solutions, I found it to be sufficient when combined with the efficiency of the GA and methods with the Numpy library. My first problem was simple, find the minimum of a 2-D function($.05\sin(30x)^2 + .1x^2$.) This is a highly oscillatory function, as shown below. This oscillatory nature allows the GA to easily be caught in a local minima

Figure 1. Illustration of the function: $.05\sin(30x)^2 + .1x^2$



This task had an easy to distinguish solution(min = 0), but it was much harder to achieve computationally. My GA was just a basic algorithm, it worked by generating a random population of 10,000 data points between -5 and 5. Next it takes each of those x-coordinates and runs them through the fitness function. Then it has to determine which one did the best. Since this is just a function minimization, the program takes the lowest 10 percent and includes them in the new population. This top ten percent are the parents. Then is generates a new population using the average of two randomly chosen parents and repeats this until it has a population size of 10,000 (the size of the original population.) Next it adds a small degree of mutation, this help from the solution becoming stuck in a 'divot' or local minima. The mutation step multiplies adds a very small number to each element in the population. It obtains this number by multiplying a random number from 0 to .000001 by the smallest element of the population. Then it adds this number to every element in the population. This function was designed to test the mutation ability of my GA. After about 1000 iterations, I reached a number around 10^{-6} ,after this it barely improved at all. I tried to add in mutation as a percent of the smallest member of the population, and I still had the same problem. This led me to conclude that this was a rounding discrepancy.

Figure 2. Result after 10 iterations

```
D:\Super>python genealg1.py 10
1.25365145466e-09
```

Figure 3. Result after 10,000 iterations

```
D:\Super>python genealg1.py 10000
1.47240769722e-08
```

This demonstrates the rounding discrepancy that is part of my program. However this is not a serious problem, due to the fact that rounding discrepancies will occur in any computer program. In addition to this, these values are very close to zero. These illustrations also show how rapidly the program converges. It achieves the same accuracy in 10 iterations that it achieves in 10,000 iterations.

In addition to the above mentioned rapid converegeence, my program is also very fast when compared to other methods. For example my GA has a maximum execution time (for ten iterations) of about 2.49 seconds while the random search method has an eqiluvant convergant time of about 21.5 seconds. Note that the random method cnan sometimes, but rarely have a far smaller min execution time (0.0125 seconds) than my GA. (It achieved this only rarely due to the random number generator 'getting lucky.') In addition to this, the randomized method is also much less reliable, with a variance in execution time of about 1720 percent. One interesting thing I noticed was that the execution times for one and 10,000 iterations were about the same (~2.1 seconds.) This leads me to conclude that my program itself is fast, only the setup is slow.

Whereas the previous problem was a good example of a smooth but complicated function with multiple minima, the next problem involves searching for a discrete solution in a very large possible solution space. The N-Queens problem involves the placement of N queens on an N-by-N chess board. The objective is to place all the queens without any of the pieces able to take each other. This problem is very interesting in that it has a somewhat limited number of

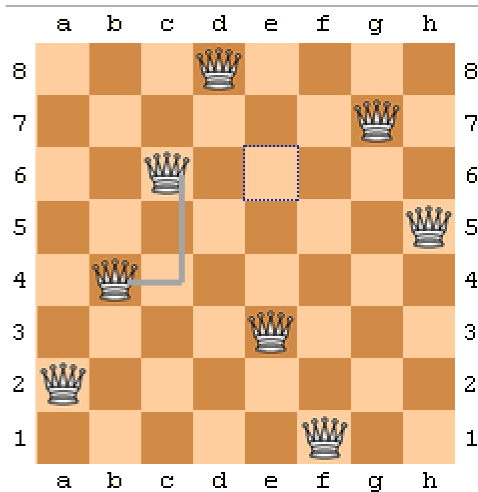
solutions but a located in a huge search space. For example, if this problem was solved with a brute force method it would require a search of $2.81474*10^{14}$ possible blind placement. However, using a slightly improved method in which each queen is placed on a separate row it would still require $1.6777 * 10^{7}$ blind solutions.

Figure 4. Chart showing number of solutions for n

n:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	..	24	25	26
unique:	1	0	0	1	2	1	6	12	46	92	341	1,787	9,233	45,752	..	28,439,272,956,934	275,986,683,743,434	2,789,712,466,510,289
distinct:	1	0	0	2	10	4	40	92	352	724	2,680	14,200	73,712	365,596	..	227,514,171,973,736	2,207,893,435,808,352	22,317,699,616,364,044

One interesting thing to note after looking at this chart is that the solution set for N = 6 is smaller than the solution set for N = 5. Another interesting fact is that there is no known formula for the number of solutions. A common solution trait is a distinctive stair step pattern, much like one a knight would make. I wrote a program that can solve the general N-queens problem. My program randomly places one queen on the board, recording it's position. Then it places another queen randomly on the board, except where the two queens paths conflict. When the path's of queens conflict, it means that one queen is able to take another queen. In order to check that no paths conflict, the program checks that there are no queens on the same row, column, or diagonal. The GA is used if this check returns false with a scoring of how badly it fails. If there are p queens on a board and if is declared invalid, then the GA returns to the last known valid step (p-1), and tries a slightly different position for the queen. At this time my GA can find some solutions for the N-queens. It is probably not possible to prove one can find all solutions with a GA.

Figure 5. One solution to the 8-queens variant, note the distinctive 'L'-shaped pattern



Conclusion

This year, I wrote and optimized several genetic algorithms that were capable of faster and more efficient calculation than random search methods. I was also able to program a simple N-queens problem and solve it for N= 8. The program generated populations, ran them through tasks, and then 'evolved' these populations. After these programs evolved for a sufficient time, they developed into solutions for the task required. I created two different methods for each task: A random search algorithm, and a method using GAs. Then I timed the maximum execution time for each method to achieve similar results. For the random search algorithm it varied greatly, but the genetic algorithm was somewhat constant, with a minimal random

component due to mutation. In each case, the GA based algorithms vastly outperformed any random based solution.

Significant Achievements

I endured many obstacles throughout the course of my project. I initially had problems with faulty Python installations and malware issues. I learned about the advantages and disadvantages of using genetic algorithm based method to solve problems. I also learned a lot about speed issues in some languages. I am currently learning writing GA methods using Cython to increase speed and efficiency. My major achievement was writing effective GA-based programs that could compute solutions quickly and efficiently, in both discrete and continuous cases. In each case, the GA-based algorithms vastly outperformed any random search algorithm.

Acknowledgments

Throughout the course of my project, many people contributed towards its completion. I would like to thank Lee Goodwin for his assistance in organizing this event at my school. I would also like to thank John Donahue for his feedback on my interim report. In addition to this, Pieter Swart provided me with a significant amount of information on how genetic algorithms work. Finally, I would like to thank those who evaluated my project at the Northern New Mexico Community College.

Works Cited

"Cython: C-Extensions for Python." Cython: C-Extensions for Python. Web. 02 Apr. 2012. <http://www.cython.org/>.

"Eight Queens Puzzle." Wikipedia. Wikimedia Foundation, 04 Feb. 2012. Web. 02 Apr. 2012. <http://en.wikipedia.org/wiki/Eight_queens_puzzle>.

"Enthought, Inc. :: Scientific Computing Solutions." Enthought, Inc. Web. 02 Apr. 2012. <http://www.enthought.com/>.

"NVIDIA Developer Zone." NVIDIA Developer Zone. NVIDIA Corporation. Web. 02 Apr. 2012. <http://developer.nvidia.com/>.

"Scientific Computing Tools For Python." Scientific Computing Tools For Python — Numpy. Web. 02 Apr. 2012. <http://numpy.scipy.org/>.

Code:

Genetic Algorithm for a complicated function:

```
1. '''
2. Alexander Swart
3. 4-4-2012
4. Supercomputing 2012
5. '''
6. import numpy as np
7. import matplotlib.pyplot as plt
8. from sys import argv
9.
10.       npop = 10000 # Size of population
11.       top = .1    # The best of the pop as a fraction
12.
13.       # noise specification
14.       mu = 0.0001
15.
16.       #Noise Amplitude
17.       noise = 0.000001
18.
19.
20.
21.
22.
23.
24.       def randpop(npop ,pmin,pmax):
25.           # populates the initial graph with random points
26.           a = np.random.random(npop)
27.           pop = pmin + (a*(pmax-pmin))
28.           return pop
29.
30.       def natsec(pop,top,fitfunc):
31.           # calculates subpopulation using the fitness function
32.           # pop = origional population
33.           # top =  The best of the pop as a fraction
34.           g = fitfunc(pop)
35.           indicies = g.argsort()
36.           t = int(len(g)*top) #Length of the new sub population
37.           subpop = pop[indicies[:t]]
38.           return subpop
39.
```

```
40.     def fitfunc(x):
41.         # The fitness function
42.         # Parabola with fluctuation,optimal range: 10 points
43.         return (.05*np.sin(30*x)**2)+ .1*x**2
44.
45.     def spawn(subpop,npop):
46.         # keep subpop
47.         # rest of new_population = avge of 2 randomly chosen subpop
              members
48.         subpop_len = len(subpop)
49.         k = 0
50.         new_population = np.zeros(npop)
51.         for k in range(subpop_len):
52.             new_population[k] = subpop[k]
53.         k = subpop_len
54.         while k<npop:
55.             n = np.random.randint(0,subpop_len - 1)
56.             random1 = subpop[n]
57.             j = np.random.randint(0,subpop_len - 1)
58.             random2 = subpop[j]
59.             avg = (random1+random2)/2.0
60.             new_population[k] = avg
61.             k = k+1
62.         return new_population
63.
64.     def mutate(pop):
65.         # Slightly mutates each value in the new population
66.         small_pop =  np.min(pop)
67.         pop += (small_pop)*(noise*np.random.normal(0,mu,len(pop)))
68.         return pop
69.
70.     def init(num_iteration):
71.         k = 0
72.         if k <= num_iteration:
73.             pop = randpop(npop,-5,5)
74.             while k<100:
75.                 subpop = natsec(pop,top,moo)
76.                 newpop = spawn(subpop,npop)
77.                 newpop = mutate(newpop)
78.                 k = k+1
79.                 if k>300: noise = 0.00000001
80.         return min(moo(newpop))
81.
82.     if __name__ == '__main__':
83.         k = 0
```

```
84.         if k <= argv:
85.             pop = randpop(npop,-5,5)
86.             while k<100:
87.                 subpop = natsec(pop,top,moo)
88.                 newpop = spawn(subpop,npop)
89.                 newpop = mutate(newpop)
90.                 k = k+1
91.                 if k>300: noise = 0.00000001
92.         print np.min(moo(newpop))
93.
94.
```