Homework Two

Iacobescu Tudor, A6

26 nov 2019

1 Introduction

This document details an experiment in which a genetic optimization algorithm is evaluated against four multi-dimensional functions, and its performance compared with three optimization algorithms previously examined in Homework One.

A genetic optimization algorithm works similarly to the evolutionary process of natural selection. A result is reached through random mutation, coupled with a bias towards the propagation of more effective individuals. As such, an approximate solution to a problem may be reached through trial and error, and incremental improvements.

This paper will describe the method of the experimental procedure, show the resulting data, and compare the results with the ones previously obtained to compare a genetic algorithm to other common optimization algorithms.

2 Method

For the experiment, we will use a C++ implementation of a simple genetic minimization algorithm. This algorithm will be tested against the same four functions used in Homework 1, and its results evaluated and compared with those of Simulated Annealing and Iterated Hillclimber.

2.1 Algorithm

As before, the algorithm relies on representing real numbers from a given interval $[a, b]$ as a string of bits. These bits simulate integers from 0 to 2^n (where n is the number of bits) corresponding roughly to values within the search interval. The specifics and advantages of this are as presented within Homework One.

The actual genetic algorithm works by generating a random population of 150 "individuals", which are bitstrings representing an input for the function. The individuals receive mutations (random bit flips, with a low chance), some of them will get "crossed" (their bits will be randomly swapped), and then a new population is created, probabilistically based on how well each individual evaluates. This process is repeated 1500 times (or "generations"), and then the best individual from the last generation is picked as the result of the algorithm.

This is the general idea of the algorithm:

```
function geneticMinimize():
    population = generatePopulation(POP_SIZE)
    generation = 1

    while generation < GEN_LIMIT:
        population = mutate(population)
        population = crossover(population)
        population = select(population, func)
        generation++

    best = bestIndividual(population, func)
    return best
```

The way mutation works is by flipping random bits within each individual with a small chance. This chance decreases with the number of bits in the individual.

```
function mutate(population):
    for individual in population:
        for bit in individual:
            if random(0, 1) < MUT_CHANCE / individual.size()
                bit = !bit
    return population
```

Crossover shuffles the population, so that 20 of them are randomly picked, in pairs, to be crossed. The crossover is done with a randomly generated bitstring used as a bitmask - if a bit in the bitmask is 1, the corresponding bits of the two individuals get swapped.

```
function crossover(population):
    population = shuffle(population)
    // helper function, shuffles the vector
    for i = 0; i < 10; i++:
        { population[i], population[i+1] } =
```

```
            cross(population[i], population)
    return population

function cross(first, second):
    bitmask = generateBitset(first.size())
    for i = 0; i < bitmask.size(); i++:
        if bitmask[i]:
            swap(first[i], second[i])
    return { first, second }
```

Selection works by evaluating each individual, assigning them a "fitness" value, then building a sort of "wheel of fortune" mechanism where individuals with a higher fitness get a higher chance of being chosen. We then "spin the wheel" 150 times, creating a new population, of copies of individuals chosen by the wheel.

The fitness is defined as a difference between a treshhold (slightly higher than the worst individual, to give even bad individuals a chance to be chosen) and each individual's evaluation value. As such, better individuals get a higher fitness value.

```
function select(population, func):
    eval = []
    maxValue = func(population[0])
    for individual in population:
        ev = func(individual)
        if ev > maxValue:
            maxValue = ev
        eval.push(ev)

    fitness = []
    treshhold = maxValue + abs(maxValue) * 0.1
    for value in eval:
        fit = treshhold - value
        fitness.push(fit)

    wheel = []
    wheel[0] = fitness[0]
    for i = 1; i < POP_SIZE; i++:
        wheel[i] = wheel[i-1] + fitness[i]
    wheelEnd = wheel[POP_SIZE - 1]

    newPop = []
    for i = 0; i < POP_SIZE; i++:
        needle = random(0, 1) * wheelEnd
```

```
        which = 0
        while needle > wheel[which]:
            which++

        // which is the chosen individual, the wheel sector
        // that the "needle landed on"

        newPop.push(population[which])

    return newPop
```

2.2 Functions

The four functions used for this experiment are the Sphere function, the Dixon & Price function, Michalewicz's function, and Rastrigin's function. The following subsections include the function expression and graph for each of these. The graph is for the two-dimensional version of each function, in its given search interval, with the vertical axis being the value of the function.

2.2.1 Sphere function

$$f(x_1 \cdots x_n) = \sum_{i=1}^{n} x_i^2, x_i \in [-5.12, 5.12]$$

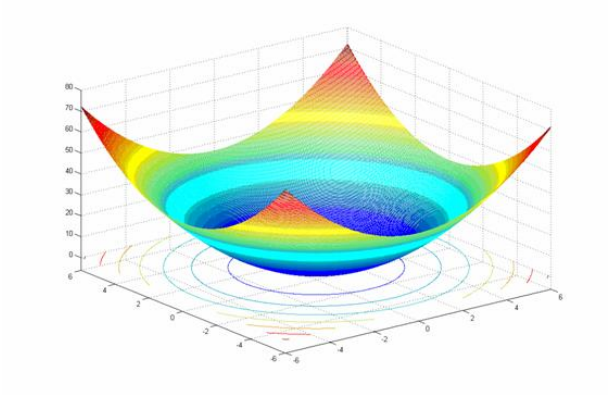

Figure 1: Sphere function graph

2.2.2 Dixon & Price function

$$f(x_1 \cdots x_n) = (x_1 - 1)^2) + \sum_{i=2}^{n} i(2x_i^2 - x_{i-1})^2, x_i \in [-10, 10]$$

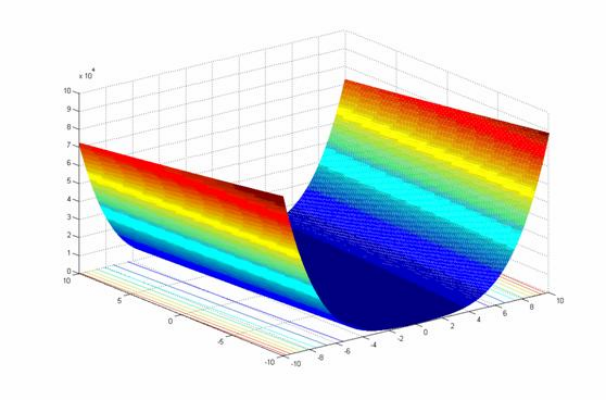

Figure 2: Dixon & Price function graph

2.2.3 Michalewicz function

$$f(x_1 \cdots x_n) = -\sum_{i=1}^{n} sin(x_i) \left[\frac{ix_i^2}{\pi}\right]^{2m}, m = 10, x \in [0, \pi]$$

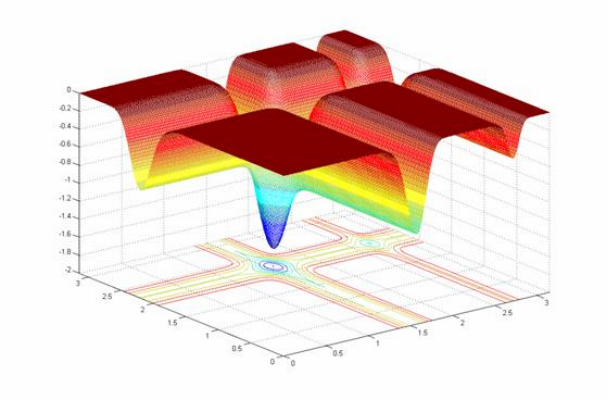

Figure 3: Michalewicz function graph

2.2.4 Rastrigin function

$$f(x_1 \cdots x_n) = 10n + \sum_{i=1}^{n} (x_i^2 - 10cos(2\pi x_i)),$$

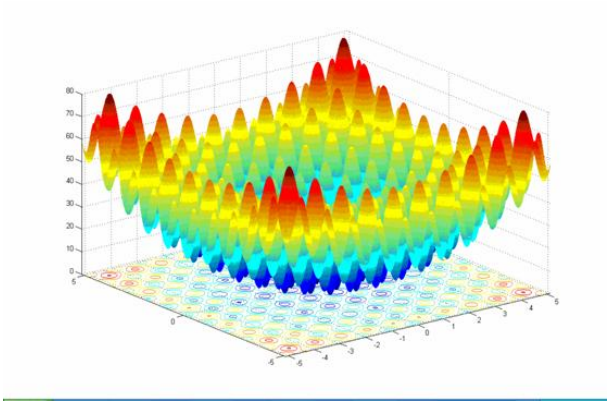

Figure 4: Rastrigin function graph

3 Experiment

For each of the four functions (for 2, 5 and 30 dimensions) the algorithm was run 30 times in parallel, each run limited to a time just above 600000ms (10 minutes). The results for each of the runs were transcribed in output files, which were then processed with a simple R script into a large LaTeX table, which was then manually split, edited and spliced with the tables from last time. The results are below.

4 Results

The following table shows a summary of the result data, on each function and number of dimensions respectively. It presents the mean, maximum, minimum and standard deviation of the results and times.

More interestingly, the following is a table comparing the four tested algorithms.

5 Result analysis

As before, Dixon & Price proves challenging to the algorithm because of its search domain size, and Rastrigin's complex graph with many local minima causes problems for higher dimensions as well.

function	d	rMean	rMax	rMin	rSDev	tMean	tMax	tMin	tSDev
dixon&price	2	0.74	4.26	0.01	0.98	49621.27	50289	48391	602.65
dixon&price	5	13.10	98.05	1.42	18.31	89026.77	90552	86896	1078.34
dixon&price	30	4944.14	10687	966.67	2773.47	443351.67	444711	440748	1117.59
michalewicz	2	-1.78	-1.48	-1.80	0.06	41696.53	42560	39150	1037.61
michalewicz	5	-4.39	-3.88	-4.64	0.22	82597.03	83960	80162	1179.32
michalewicz	30	-23.64	-20.96	-26.61	1.18	402037.57	404535	396398	2340.09
rastrigin	2	1.61	5.30	0.00	1.20	48127.53	49429	46334	1121.48
rastrigin	5	8.10	14.44	2.14	3.29	82636	85186	79665	2033.70
rastrigin	30	81.56	106.80	54.84	13.75	389563.27	397649	374288	7044.99
sphere	2	0.01	0.04	0.00	0.01	49067.73	49836	47400	740.50
sphere	5	0.04	0.13	0.01	0.03	85068.53	86145	83524	664.93
sphere	30	1.82	2.98	0.74	0.54	395912.27	401374	384129	4946.80

Table 1: Genetic minimization result (r) and run time (t) statistics. Times are in ms.

func	d	IHC/FA		IHC/SA		Sim. Annealing		Genetic		min(f)
		rMean	tMean	rMean	tMean	rMean	tMean	rMean	tMean	
	2	0.03	1928.13	0.01	6672.93	0.99	5248.50	0.74	49621.27	0
d&p	5	185.17	9573.10	4.49	40694.50	114510	6172.77	13.10	89026.77	0
	30	1039584	48142.93	673153	1017064	3592796	26734.00	4944.14	443351.67	0
	2	-1.80	1655.00	-1.80	5771.40	-0.34	600000	-1.78	41696.53	-1.80
mich	5	-3.07	7348.27	-3.73	34603.47	-0.60	600000	-4.39	82597.03	-4.68
	30	-8.33	28802.57	-8.86	887536.93	-3.58	600000	-23.64	402037.57	?*
	2	0.53	1888.13	0.36	6406.43	2.04	50906.13	1.61	48127.53	0
rast	5	18.96	8878.93	8.37	38223.17	59.77	207850.47	8.10	82636.00	0
	30	368.88	27435.30	349.83	947533.87	563.78	558424.10	81.56	389563.27	0
	2	0.00	1845.03	0.00	6381.27	0.92	270970.37	0.01	49067.73	0
sph	5	1.56	8813.83	0.00	39890.83	43.26	582070.63	0.04	85068.53	0
	30	133.49	27035.33	111.89	938650.63	269.46	600000	1.82	395912.27	0

Table 2: Comparison table. Function names shortened and certain large values truncated to save width.
?*: Michalewicz function minimum for 30 dimensions unknown.
IHC/FA: Iterated Hillclimber, First Ascent
IHC/SA: Iterated Hillclimber, Steepest Ascent

While the second table is forced to omit a few pieces of important data (such as the standard deviations of the times and results), it does still yield some interesting information. The genetic algorithm, while slower and less accurate than the others for simple problems (i.e. 2 dimensions), at higher dimensions it produces more accurate data than all other algorithms, and in the case of IHC/SA and Simulated Annealing, better times as well.

6 Conclusion

While genetic algorithms aren't necessarily the best solution for every problem, their adaptability makes them ideal for many complex problems. As previously thought, the abnormalities that plagued the previous experiments did not appear in this one. Genetic algorithms are more adaptable, and do not suffer as much from the particular properties of each function.

One unique feature that this series of experiments didn't cover is their extension to problems where the efficiency of a solution isn't always this simple to determine, or changes over time. Hopefully, future experiments will show these properties as well.

7 References

- Functions and graphs: `http://www-optima.amp.i.kyoto-u.ac.jp/member/student/hedar/Hedar_files/TestGO_files/Page364.htm`