CIS 579: Artificial Intelligence

Assignment #2
Genetic Algorithms

Nicholas Butzke

June 13, 2023

1 Introduction

This paper demonstrates the effectiveness of Genetic Algorithms. Inspired by genetic variation and the system of "survival of the fittest" in biology, this Genetic Algorithm will solve an optimization problem.

2 Problem Description

Implement a simple genetic algorithm with fitness-proportionate selection (roulette-wheel sampling), population size 50, single-point crossover rate $p_c = 0.7$, and bit-wise mutation rate $p_m = 0.001$. Try it on the following fitness function: $f(x) = \sum_{i=0}^{\infty} \mathbf{1}_{\{1\}}(x_i)$, where x is a genome of length 10. Perform 30 runs, and measure the average generation at which the string of all ones is discovered. Perform the same experiment with crossover turned off ($p_c = 0$).

3 Supporter Function Breakdown

3.1 randomGenome(length)

```
def randomGenome(length: int) -> str:
    genome = ""
    for i in range(length):
        genome += str(random.choice([0,1]))
    return genome
```

This function returns a random genome (bit string) of a given length.

3.2 makePopulation(size, length)

```
def makePopulation(size: int, length: int) -> list:
    population: list = []
    for i in range(size):
        population.append(randomGenome(length))
    return population
```

This function returns a new randomly created population of the specified size, represented as a list of genomes of the specified length.

3.3 fitness(genome)

```
def fitness(genome: str) -> int:
    return genome.count('1')
```

This function returns the fitness value of a genome.

3.4 evaluateFitness(population)

```
def evaluateFitness(population: list) -> float:
    fitList: list = np.empty([0,2])
    for i, ele in enumerate(population):
        fitList = np.append(fitList, [[i, fitness(ele)]], axis = 0)
    return float(np.average(fitList[:, 1], axis=0)), max(fitList, key=
                                            lambda x: x[1])
```

This function returns a pair of values: the average fitness of the population as a whole and the fitness of the best individual in the population with its index.

3.5 selectPair(population)

```
def selectPair(population: list) -> str:
    avgFit, NULL = evaluateFitness(population)
    weights = []
    for ele in population:
        weights.append(int(((fitness(ele)/avgFit)/len(population))*100))

    g1 = random.choices(population, weights)[0]
    g2 = random.choices(population, weights)[0]

    return g1, g2
```

This function selects and returns two genomes from the given population using fitness-proportionate selection.

3.6 crossover(genome1, genome2)

```
def crossover(genome1: str, genome2: str) -> str:
    crossoverPoint = random.randrange(0, len(genome1)-1)
    return genome1[:crossoverPoint] + genome2[crossoverPoint:]
```

This function returns two new genomes produced by crossing over the given genomes at a random crossover point.

3.7 mutate(genome, mutationRate)

```
def mutate(genome: str, mutationRate: float) -> str:
    newGenome = ""
    for c in genome:
        if random.randrange(0,99,1) <= mutationRate*100:
            newGenome += str(int(c) ^ 1)
        else:
            newGenome += c
    return newGenome
```

This function returns a new mutated version of the given genome.

4 Genetic Algorithm Implementation

```
genome: str = ""
count: int = 0
population = makePopulation(populationSize, 10)
avgFit, m = evaluateFitness(population)
genome = population[int(m[0])]
print("Generation    " + str(count) + ": average fitness " + "{:.2f}".
                                      format(avgFit) + ", bestfitness " + "
                                      {:.2f}".format(m[1]))
```

This chunk performs the creation of the initial population. All genomes are randomly generated based of the given parameters and no crossover or mutation will happen here. The generated population will be evaluated and the genome with the highest fitness will be selected.

```
while(genome != "1"*10 and count < 30):
    count += 1
    if random.randrange(0,99,1) <=int(crossoverRate*100):
        pair = selectPair(population)
        genome = crossover(pair[0], pair[1])
    genome = mutate(genome, mutationRate)
    population = makePopulation(len(population)-1,10)
    population.append(genome)
    avgFit, m = evaluateFitness(population)
    genome = population[int(m[0])]
    print("Generation    " + str(count) + ": averagefitness " + "{:.2f}".
                                          format(avgFit) + ", bestfitness "
                                           + "{:.2f}".format(m[1]))
```

This is the main loop of the genetic algorithm. The while conditions are the end conditions and the algorithm will stop if the goal is found or if the generation count reaches 30. Within this loop there will be 3 main functions. First, the program will crossover the genomes at

a frequency based off the crossoverRate. If crossing over genomes occurs, a pair of genomes will be selected, and this pair will crossover. The outcome of the crossover will become the selected genome. Second, the selected genome will be processed by the mutation function. This will make each character in the genome have a probability to flip based off the mutationRate. Finally, the program will generate a new population of genomes with space for the selected genome to carry over. If the population size is 50 then 49 genomes will be randomly generated with 1 genome carrying over. The population will then be evaluated and the genome with the highest fitness will be selected for the next loop.

5 Results

The algorithm was given two different values for p_c and subsequently ran 30 times for each p_c value. The average amount of generations the algorithm took to reach an end condition were then recorded for results.

p_c	**Avg.Generations**	**Runtime**	**Average time per Generation**
0.7	13.83	0.19000s	0.00046s
0	19.63	0.20200s	0.00034s

Setting p_c to a non-zero value increased the generational efficiency of the algorithm however this implementation of crossover was computationally expensive. While the algorithm is more likely to find a solution in less generations, it is also more likely to take longer per generation. This may be due to the additional fitness evaluation and selection of optimal genomes to crossover.

To test this the algorithm performed an additional 10,000 runs for each p_c value.

p_c	**Avg.Generations**	**Runtime**	**Average time per Generation**
0.7	15.09	61.46210s	0.00041s
0	14.24	46.29651s	0.00033s

This additional test provided similar values but with greater confidence in the accuracy of the results.

6 Conclusion

Theoretically, crossover genome production should more rapidly discover the optimal solution on generational efficiency. However, this did not appear in the results. While the algorithm frequently found the optimal solution the current implementation of the crossover function was too computationally expensive and still unoptimized for the 0.7 p_c value. Further optimization of the crossover function should improve efficiency on both generations produced and time per generation.

Regardless of crossover efficiency, this genetic algorithm showed itself to be more efficient than a brute force optimization of the problem. A brute force solution would converge on 2^{10} or 1024 genome attempts whereas the genetic algorithm converged to 734 genome attempts to find the optimal solution. Given a more complex problem the genetic algorithm would be predicted to more significantly generate an optimal solution when compared to a brute force algorithm.