CSE598 Mini Project 6 - Distributed and Swarm AI

PSO vs Generic GA in Efficiency of Finding Global Minimum

- Aidan Hoppe

Introduction

Particle Swarm Optimization is a population based optimization algorithm that can be used to find the global optimum/minimum of an optimization function. It functions similarly to a simple generic genetic algorithm by the fact that it searches using a population of solutions at one time and uses information about the current best solution to guide other solutions. However over multimodal spaces, a simple genetic algorithm can have trouble reliably identifying the global minimum of the solution space quickly. Particle swarm optimization starts with a random population of possible solutions. These solutions are then evaluated according to the optimization function. The particles (solutions) remember their personal best position and fitness in that location and are also made aware of the global best position and fitness found by the swarm. The particles iteratively move towards their best position and the position found by the swarm with random proportions until one particle finds the global optimum.

Methods

For this project I created a particle swarm optimization algorithm from scratch and a simple genetic algorithm from scratch and then pitted them against each other in a series of tests to see the benefits of each. I used the well known test functions for optimizations problems: Matya's Function, Levi Function N.13, and Boothe's Function. I chose these three because they all have only one global optimum of 0 over the desired interval $-10 < x,y < 10$. This makes my job easy because I can change which function my programs optimize with just 1 word change. Matya's and Boothe's are not multi-modal and were primarily used in the creation of my algorithms because they can give indication on whether the algorithms are converging to the correct values easily. The hyper-parameters relevant to PSO are the population size, the inertial value, and the acceleration value. I set these values as 100, .5 and .5 respectively. These were chosen in an attempt to simply create an average, generic PSO algorithm that can be compared with a genetic algorithm of the same population size. The hyperparameters for the GA are as follows: population size of 100, and genetic diversity through crossover and mutation. Crossover occurs when two parents produce offspring, they swap x and y genes for their children. There is a probability .6 to mutate genes and mutations occur by shifting a gene +/- a random number between (0,.5). This way, mutations occur often, but incrementally so that the algorithm can still achieve some degree of accuracy.

In order to test the algorithms, I set a threshold value to determine whether the optimum has been reached or not. Since this depends on the user definition of successfully reaching the optimum, I tested with a couple values: .000001, .00001, .0001, and .001. This means that the algorithm can return a solution that evaluates to less than the threshold and it will be considered optimal. These threshold values are chosen because for the test functions used, all global optimum (minimum) are at 0. I then ran 500 iterations of particle swarm until the acceptance threshold was met and 50 iterations of the GA until the acceptance threshold was met. The average time taken to reach the threshold for each was recorded and logged in a table.

After recognizing the domination of PSO over the simple GA, I wanted to try and inject some of the power of PSO into the GA in the most basic way possible and see if we could significantly improve it's performance. From the results in Figure 1 below, it is clear that PSO exponentially dominates the simple GA when needing high precision. This is because PSO has tools to inch towards the expected optimum while the simple GA at best makes random guesses in a 1 unit range around the current best solutions. We want to narrow down that 1 unit range if we are confident that we are approaching the optimum in order to spend more of our guesses focused around probable more-optimal solutions. We do this by reducing the mutation severity with time, so that the mutations applied are within $\frac{-1}{\sqrt{iterations}} < mutationVal < \frac{1}{\sqrt{iterations}}$. I chose to add the square root because I was having trouble converging on the correct value quick enough and could get "stuck in space" by not being able to mutate far enough from my current position each iteration.

Results

Below is the table that records PSO's average time to convergence vs the simple GA's for the tested threshold values.

Acceptance Threshold	PSO time taken AVG (seconds)	GA time taken AVG (seconds)	PSO : GA
.001	.00627	.0244	1 : 3.89
.0001	.00781	.2609	1 : 33.41
.00001	.00949	1.264	1 : 133.19
.000001	.01162	3.6322	1 : 312.58

Figure 1

What we expect, and do see, is that the particle swarm algorithm increasingly outperforms this simple GA as the acceptance threshold goes down. This means that for fine-detailed accuracy in a timely manner, PSO is much more effective than a simple GA. This is expected though, because a simple GA does not have a mechanism for quickly narrowing down on a value significantly less than it's mutation range. For example, in the case where the genes mutate by a random float between -.5 and .5, if your best solution's positions are already very close to the answer, you will have trouble essentially guessing a better solution near yours because >99.9% of mutations are now moving you away from the optimum. This led to an idea about variable mutation rates to upgrade the simple GA and allow it to transfer from exploration to exploitation. The concept is just a GA where the mutation range (the severity of possible mutation) is reduced with more iterations. This way, the GA has a tool to overcome the obstacle that the simple GA has above where it cannot reliably guess answers VERY similar to the current best. The upgraded GA's performance and comparison to PSO are recorded below.

Acceptance Threshold	PSO time taken AVG (seconds)	Upgraded GA time taken AVG (seconds)	PSO : UGA
.001	.00627	.00712	1 : 1.1355
.0001	.00781	.0247	1 : 3.163
.00001	.00949	.0915	1 : 9.642
.000001	.01162	.2425	1 : 20.87

Figure 2

Clearly this upgrade has had a massive improvement in terms of bringing the simple GA up to the speed of particle swarm optimization, but it still is not enough to beat it. The upgraded GA is about 10x faster than the simple GA though which accounts for a significant amount of the difference between PSO and the GA.

Discussion

This does give a good insight into the success of PSO and what makes it just so effective versus a standard GA. We can see that a big bottleneck for the GA when competing against PSO is that the GA does not have a method for "honing in" on a solution more accurately than random guesses within its mutation range. PSO tackles this problem by moving towards a new solution by some value related to its current distance from the best known distance. This imposes an upper bound on the movement that the particle will take, encouraging the particle to not jump significantly farther than its distance to the global best + its personal best. The particles will begin to "swarm" and hone in on the optimal solution, making more and more guesses around it until the acceptance threshold is reached. We applied this same idea of the upper bound on movement to the simple genetic algorithm in an attempt to upgrade it and were greatly successful. By allowing the particles/individuals to make large jumps in the beginning and take smaller and smaller jumps as the program goes on, the algorithms follow a form of exploration then exploitation. By making large jumps in the beginning, the particles and the individuals are able to explore the solution space and find some good leads. Later on, the particles/individuals will jump less and less so that they can nestle into the minimum that they are surrounding.

Appendix

Particle Class:

```
class Particle:
    def __init__(self, pos, velocity, pbest):
        #pos = [x, y]
        self.pos = pos
        #velocity = [vx, vy]
        self.velocity = velocity
        #pbest = [eval, pos]
        self.pbest = pbest
```

Particle Swarm Main Function:

```
def particleSwarm(population, gbest):
    for par in population:
        #eval = booth(par)
        eval = levi13(par)
        #eval = matyas(par)
        if eval < par.pbest[0]:
            par.pbest = [eval, [par.pos[0], par.pos[1]]]
        if eval < gbest[0]:
            gbest = [eval, [par.pos[0], par.pos[1]]]
        #UPDATE VELOCITY
        vnewx = INERTIA * par.velocity[0] +
ACCELERATION*(random.random()*(par.pbest[1][0]-par.pos[0])+random.ra
ndom()*(gbest[1][0]-par.pos[0]))
        vnewy = INERTIA * par.velocity[1] +
ACCELERATION*(random.random()*(par.pbest[1][1]-par.pos[1])+random.ra
ndom()*(gbest[1][1]-par.pos[1]))
        par.velocity = [vnewx, vnewy]
        #UPDATE POSITION
        par.pos[0] = par.pos[0]+vnewx
        par.pos[1] = par.pos[1]+vnewy
    return population, gbest
```

Genetic Algorithm Parent Selection:

```
def getParents(population):
    parents = []
    for _ in range(PARENTS_SIZE):
        temp = random.choices(population, k=10)
        temp.sort()
        parents.append(temp[0])
        population.remove(temp[0])
    return parents
```

Upgraded Genetic Algorithm Offspring Generation + Crossover/Mutation:

```
def getOffspring(parents, iterations):
    offspring = []
    for _ in range(40):
        ps = random.choices(parents, k=2)
        #crossover
        off1 = [MAX, [ps[0][1][0], ps[1][1][1]]]
        off2 = [MAX, [ps[1][1][0], ps[0][1][1]]]
        #mutation
        if random.random()>.4:
            off1[1][0] +=
random.uniform((-1/math.sqrt(iterations)),(1/math.sqrt(iterations)))
            off1[1][1] +=
random.uniform((-1/math.sqrt(iterations)),(1/math.sqrt(iterations)))
        if random.random()>.4:
            off2[1][0] +=
random.uniform((-1/math.sqrt(iterations)),(1/math.sqrt(iterations)))
            off2[1][1] +=
random.uniform((-1/math.sqrt(iterations)),(1/math.sqrt(iterations)))
        offspring.append(off1)
        offspring.append(off2)
    return offspring
```

Genetic Algorithm:

```
def geneticAlgorithm(population, iterations):
    newpop = []
    best = MAX
    for p in population:
        p[0] = fitness(p[1])
        if p[0]<best:
            best = p[0]
    parents = getParents(population)
    for p in parents:
        newpop.append(p)
    offspring = getOffspring(parents, iterations)
    for o in offspring:
        newpop.append(o)
    return best, newpop
```