Evolutionary Computation Assignment

1. Introduction

In computer science, **evolutionary computation** is a family of population-based, trial and error problem solvers with a metaheuristic or stochastic optimization character. In evolutionary computation an initial set of candidate solutions is generated and iteratively updated. Each new generation is produced by stochastically removing less desired solutions, and introducing small random changes. In biological terminology, a population of solutions is subjected to some kind of selection and mutation. As a result, the population will gradually evolve to increase in fitness, in this case the chosen fitness function of the algorithm. Evolutionary computation techniques can produce highly optimized solutions in a wide range of problem settings, making them popular in computer science. Many variants and extensions exist, suited to more specific families of problems and data structures.

Genetic algorithms form a subset of evolutionary computation in that they generally only involve techniques implementing mechanisms inspired by Charles Darwin's theory of natural/biological evolution such as reproduction, mutation, recombination, natural selection and survival of the fittest. Candidate solutions to the optimization problem play the role of individuals in a population, and the cost function determines the environment within which the solutions "live". Evolution of the population then takes place after the repeated application of the above operators. In this process, there are two main forces that form the basis of genetic systems: recombination, mutation and crossover create the necessary diversity and thereby facilitate novelty, while selection acts as a force increasing quality. Many aspects of such a process are stochastic. Changed pieces of information due to recombination and mutation are randomly chosen. On the other hand, selection operators can be either deterministic, or stochastic so that individuals with a higher fitness have a higher chance to be selected than individuals with a lower fitness, but even the weak individuals can have a chance to become a parent or to survive.

Genetic algorithms can be used in most optimization problems, where we know how the answer could look but don't have the answer specifically so we start off with a relatively bad answer and slowly work our way to a relatively good answer. Genetic algorithms are sufficiently randomized in nature, but they perform much better than random local search (where we just try random solutions, keeping track of the best so far), as they exploit historical information as well.

2. Terminology

Before I move on I would like to briefly explain the meaning of some words I'll be using in the context of this paper:

Gene – a carrier of genetic information regarding a specific trait

Chromosome (Individual) – all genes that carries genetic information of an individual, possible solution of our optimization problem

Population – all individuals

Parents – individuals that are combined to create a new individual

Mating pool – all parents

Crossover – production of offspring by combining parents.

Mutation – a change in a gene sequence, a way to introduce variation in our population by adding random noise to genes

Fitness – a measure of adaptation of an individual, function that tells us how good each individual is

Generation – level of advancement towards population with better fitness

Stopping criterion – way to stop the algorithm, here a limit of allowed generations to reach

3. Main idea

3.1. Description

The main idea of Evolutionary algorithms is that we have a pool of possible solutions to the given problem. These solutions then undergo recombination and mutation (like in natural genetics), produce new children, and the process is repeated for various generations. Each individual (or candidate solution) is assigned a fitness value (based on its objective function value) and the fitter individuals are given a higher chance to mate and yield fitter

3.2. Steps

The main steps of evolutionary algorithm are:

1. Generate the initial population
2. Evaluate the fitness of each individual in population
3. Select parents for reproduction
4. Breed new individuals
5. Update population

6. If the stop condition is met return best solution, else go to step 2

3.3. Strategies

Each of these steps can come in different variations depending on the chosen strategy. I'm going to describe how each of this steps look like in my code down below together with its implementation using three different strategies:

- Simple Evolution Strategy
- Simple Genetic strategy
- Covariance-Matrix Adaptation Evolution Strategy (CMA-ES)

4. Preparation

Before we can move on we need to define some variables and import some functions from a numpy library to make math related operations easier. Although I use randomly generated numbers, I've set the generator's seed to a fixed value to be able to reproduce the same sequence of random numbers every time.

```
from numpy import empty, argmax, argmin, argsort, array, concatenate, sqrt, floor
from numpy.random import seed, uniform, normal
from numpy.linalg import norm
seed(0)
```

5. Generating initial population

The DNA of every individual is composed of genes and each of those genes comes in different versions. In our case, our individuals are going to be sets of bomb coordinates, where each coordinate is a gene. It means that based on the number of fitness function inputs, each individual in the population will have nBombs*2 genes, two genes for each two coordinates of each bomb. The main idea when we create the first population is that we must make the population as wide as possible so we wouldn't point the population towards a local minimum. In our case, we are just going to create a population composed of randomly placed bombs, that way our population can explore a wide variety of different arrangements of bombs.

```
nBombs = 3
populationSize = 100
population = uniform(low=0, high=100, size=(populationSize, nBombs, 2))
```

We are able to create the initial population randomly using the uniform function. According to the selected parameters, it will be of shape (1000, 3, 2) and it’s values will range between 0 and 100. This is 100 individuals in the population and each one has 6 genes, two for each two coordinates of each bomb. Even though it’s a seemingly small number of chromosomes, assuming the coordinates of the bombs can only be integers, it would still mean that for position table of size 100×100 and 3 bombs brute force algorithm would have to test $(tableLength \cdot tableWidth)^{nBombs} = (100 \cdot 100)^3 = 10^{12}$ scenarios.

6. **Evaluating the fitness of each individual in the population**

Main goal of our fitness function is to place some insect-bombs in such places so that we minimize number of all remaining, living wasps in given nests and it looks as follows:

$$F\left(X_1, Y_{1,\ldots}, X_N, Y_N\right) = \sum_{j=0}^{M} n_{Nj}$$

where:

$F\left(X_1, Y_{1,\ldots}, X_N, Y_N\right)$ – fitness function equal

the to the number of all remaining wasps after bomb N

X_i and Y_i – coordinates of bomb i

N – number of bombs

M – number of nests

n_{0j} – number of wasps in a nest j before any bomb

n_{ij} – number of wasps remaining in a nest j after bomb i calculated by equation:

$$n_{ij} = \left\lfloor \frac{n_{((i-1)\,j)} \cdot d_{ij}}{d_{max}} \right\rfloor$$ where:

d_{max} – greatest possible distance between two nests

d_{ij} – Euclidean distance between bomb i and nest j

We can implement such function as follows:

```
# greatest possible distance between two nests
dMax = sqrt(20000)
# all bomb coordinates of currently processed solution
for i, bombs in enumerate(population):
        # temporary numbers of wasps in each nest
        tmpWasps = wasps[:]
        # coordinates of currently processed bomb
        for coords in bombs:
                # distances between each nest and currently processed bomb
                distances = norm(positions - coords, axis=1)
                # information about what part of wasps in
                # each nest will remain after the bomb
                parts = distances / dMax
                # updating numbers of wasps in each nest
                tmpWasps = parts * tmpWasps
        # fitness of currently processed solution
        fitness[i] = sum(tmpWasps)
```

7. Selecting parents

Based on the fitness function we are going to select the best individuals within the current population as parents (in the Simple Evolution Strategy we choose one parent) for creating new individuals and the most common approaches are either simply selecting a set of the best individuals or using fitness proportionate selection (“roulette wheel selection”) where each individual has different probability of being selected according to it’s fitness. Simply selecting the nParents best of individuals can be implemented as follows:

```
# number of parents used for reproduction
nParents = 20
# indices that would sort an array
indices = argsort(fitness)
# indices of the nParents best solutions
parentIds = indices[:nParents]
# nParents best solutions indicated by parentIndices
parents = population[parentIds]
```

8. Breeding new individuals through crossover and mutation operations

8.1. Simple Evolution strategy

Since it’s one of the simplest evolution strategies we simply clone the best solution multiple times and add some kind of noise (for example samples from normal distribution) to each individual within offspring. This simple algorithm will generally only work for simple problems. Given its greedy nature, it throws away all but the best solution, and can be prone to be stuck at a local optimum for more complicated problems. It would be beneficial to sample the next generation from a probability distribution that represents a more diverse set of ideas, rather than just from the best solution from the current generation. This step can be implemented as follows:

```
# nOffspring clones
clones = repeat(parents[newaxis], nOffspring, 0)
# offspring created by adding noise to clones
offspring = clones + normal(size=(nOffspring, nBombs, 2))
```

8.2. Simple Genetic strategy

Keeping on the biologic analogy for the breeding in our genetic algorithm the goal of sexual reproduction is to mix the DNA of individuals, so let’s do the same thing here. The DNA of our individuals is defined by values of their genes. Thus in order to mix their DNA, we just have to mix their genes. There are lots of ways to do this but the simplest solution is for each bomb to take the mean of the parent bomb’s coordinates. The parents are selected in a way similar to a ring. Parents with indices 0 and 1 are selected at first to produce offspring, then parent 1 with parent 2 and so on. When we reach the last parent and still need to produce more offspring, we select the parent with the last index and go back to the parent with index 0, and so on so there can be more children than parents.

One of the problems I encountered here is the choice of which one parent's bomb crossover with which second parent's bomb, as we should crossover bombs that are the closest to each other but the order of bombs within individual is random, so we can’t just crossover one parent's first bomb with the second parent's first bomb and so on, because doing so could produce bad offspring coming from good parents. To avoid that we can check for each of one parent’s bombs which of the second parent’s bombs is the closest and crossover with that bomb. Some kind of noise (for example samples from normal distribution) can also be injected into each new solution. This step can be implemented as follows:

```
# container for nOffspring solutions derived from-parents
offspring = empty((nOffspring, nBombs, 2))
for offspringId in range(nOffspring):
    # parents used to create new offspring solution
    mom = parents[offspringId % nParents]
    dad = parents[(offspringId + 1) % nParents]
    # temporary container for offspring solution
    tmpOffspring = empty((3, 2))
    # coordinates of currently processed mom bomb
    for i, coords in enumerate(mom):
        # distances between each dad bomb and currently processed mom bomb
        distances = norm(coords - dad, axis=1)
        # index of the dad bomb nearest to the currently processed mom bomb
        nearestId = argmin(distances)
        # dad bomb nearest to the currently processed mom bomb
        nearest = dad[nearestId]
        # child bomb derived from crossover between parent bombs
        childBomb = ( nearest + coords) / 2
        # child bomb mutated by adding sample from a normal distribution
        mutated = childBomb + normal(size=2)
        # filling temporary offspring container with bombs
        tmpOffspring[i] = mutated
    # filling offspring container with ready solutions
    offspring[offspringId] = tmpOffspring
```

8.3. Covariance-Matrix Adaptation Evolution strategy (CMA-ES)

Different approach to crossover similar to both Simple Evolution Strategy and Simple Genetic strategy could be combining all parents into a single, artificial individual and use it with the Simple Evolution Strategy. In this strategy we have the same problem with determining which bombs should we crossover with each other as we had in Simple Genetic strategy. To solve that problem we can just group all parents bombs into nBombs groups, crossover within such groups and then combine all groups representatives into single, artificial individuals. The main difference between this strategy and either Simple Evolution strategy or Simple Genetic strategy is that at the end we don't add simple samples from normal distribution. CMA-ES takes the results of each generation within each group, and adaptively increases or decreases the search space for the next generation by calculating the covariance matrix and adapting the mean and sigma parameters of a multivariate normal distribution to sample additional noise. This step can be implemented as follows:

```
# container for bomb groups
clusters = empty((nBombs, nOffspring, 2))
# nParents best solutions indicated by parentIndices
parents = population[parentIds]
# flattened population, coordinates of all bombs
flatPopulation = reshape(parents, (-1, 2))
# tool used to group bombs according to their coordinates
kMeans = KMeans(n_clusters=nBombs).fit(flatPopulation)
# centers of each group
centers = kMeans.cluster_centers_
# labels of bombs
labels = pairwise_distances_argmin(flatPopulation, centers)
for i in range(nBombs):
    # all bomb belonging to the group i
    cluster = flatPopulation[labels == i]
    # covariance matrix within cluster
    covMatrix = cov(cluster, rowvar=False)
    # new bomb coordinates sampled from a multivariate normal distribution
    clusters[i] = multivariate_normal(centers[i], covMatrix, nOffspring)
# container for nOffspring solutions from combining bomb groups
population = empty((nOffspring, nBombs, 2))
# each new individual takes one bomb from each group
for i, bombs in enumerate(zip(*clusters)):
    population[i] = bombs
```

8.4. Other

There are also some non-strategy-specific rules. For example we have to fix the number of parents and offspring per generation in order to keep a stable population, but the number of individuals in the first generation doesn't have to equal to the number of individuals in the next generations. Very important thing is that after being created each individual within offspring has its DNA mutated a little bit. Mutation serves an important function, as it helps to avoid getting stuck in a local minimum by introducing novel chromosomes that will allow us to explore other parts of the solution space.

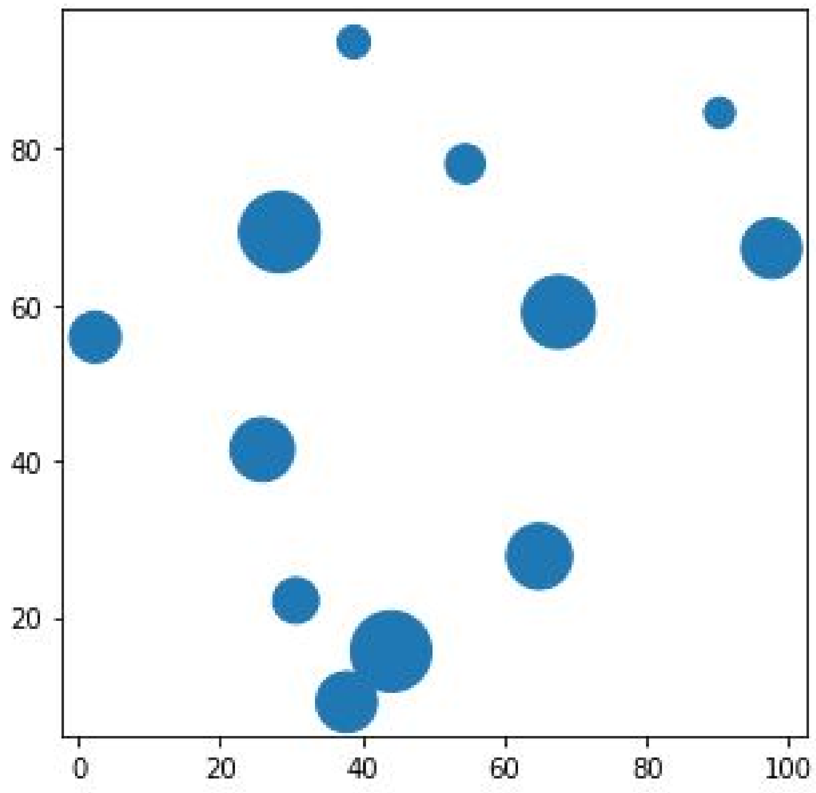
9. Updating population

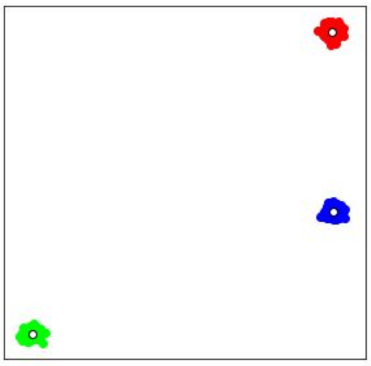
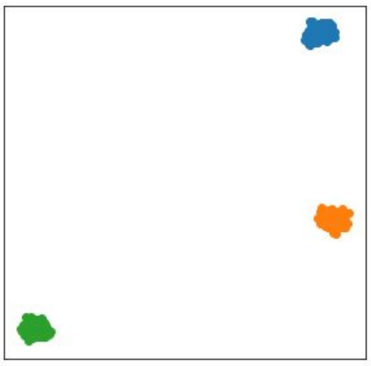
At this point, we successfully produced offspring from the selected parents and we are ready to create the new population of the next generation. Note that genetic algorithms are a random-based optimization technique. It tries to enhance the current solutions by applying some random changes to them. Because such changes are random, we are not sure that they will produce better solutions. For such reasons, it is preferred to keep the previous best solutions (parents) in the new population. In the worst case when all the new offspring are worse than such parents, we will continue using such parents. As a result, we guarantee that the new generation will at least preserve the previous good results and will not go worse. This design feature is called **elitism**. With elitism, the best performing individuals from the population will automatically carry over to the next generation, ensuring that the most successful individuals persist. The new population will have its first nParents solutions from the previous parents and the last nOffspring from the offspring as follows:

```
population = concatenate((parents, offspring))
```

10. Results

Below I'm presenting results of a few experiments using genetic algorithms. Each nest has random coordinates and random number of wasps inside (12 nests and 7691 wasps in total):



Simple Evolution strategy	Simple Genetic strategy	CMA-ES
Execution took: 0.296 s Number of wasps killed: 6541 	Execution took: 0.78 s Number of wasps killed: 6543 	Execution took: 1.026 s Number of wasps killed: 7448