Evolutionary Algorithms with Keen

Ignacio Slater Muñoz
reachme@ravenhill.cl

Contents

1. Installation

This section guides you through the process of setting up the Keen framework and the EvolutionPlotter in your Kotlin project using Gradle Kotlin DSL.

1.1. Gradle Kotlin DSL Setup

1.1.1. Step 1: Specify Versions if `gradle.properties`

First, define the versions of Keen and Compose in your `gradle.properties` file. Make sure to replace these with the latest versions available.

```
// gradle.propertis
// Keen framework version. Replace with the latest version.
keen.version=1.1.0
// Compose version for the EvolutionPlotter. Replace at your discretion.
compose.version=1.5.11
```

1.1.2. Step 2: Configure Plugin Management in `settings.gradle.kts`

This step is essential only if you plan to use the EvolutionPlotter. Here, you configure the plugin management for the Compose plugin.

```
// settings.gradle.kts
pluginManagement {
    repositories {
        // Standard Gradle plugin repository.
        gradlePluginPortal()
        // Repository for JetBrains Compose.
        maven("https://maven.pkg.jetbrains.space/public/p/compose/dev")
        // Google's Maven repository, sometimes needed for dependencies.
        google()
    }

    plugins {
        // Apply the Compose plugin with the specified version.
        id("org.jetbrains.compose") version extra["compose.version"] as String
    }
}
```

1.1.3. Step 3: Configure Project Plugins, Repositories, and Dependencies
In your build script, configure the necessary plugins, repositories, and dependencies.

```
// Retrieve the Keen version defined earlier.
val keenVersion: String by extra["keen.version"] as String

plugins {
    /* ... */
    // Include this only if using the EvolutionPlotter.
    id("org.jetbrains.compose")
}

repositories {
    // Maven Central repository for most dependencies.
    mavenCentral()
    /* ... */
}

dependencies {
    // Keen core library dependency.
    implementation("cl.ravenhill:keen-core:$keenVersion")
    // Compose dependency, required for the EvolutionPlotter.
    implementation(compose.desktop.currentOs)
    /* ... */
}
```

1.2. Additional Notes:
- Ensure that the versions specified in `gradle.properties` are compatible with your project setup.
- The `pluginManagement` block in `settings.gradle.kts` is crucial for resolving the Compose plugin, especially if you're using features like the EvolutionPlotter.
- Remember to sync your Gradle project after making changes to these files to apply the configurations.

2. One Max Problem
The One Max Problem (OMP) is a fundamental optimization challenge that is frequently utilized as a benchmark within the field of evolutionary algorithms and other heuristic search strategies. Evolutionary algorithms, inspired by natural selection, such as genetic algorithms, and heuristic methods, which seek practical solutions at the expense of completeness, often leverage OMP to test their efficacy in navigating complex solution spaces.

2.1. The Core Challenge
Consider the task described below:

> Given a binary string $x = (x_1, x_2, ..., x_n)$ of length n, where each x_i is either 0 or 1, identify a binary string that maximizes the sum of its bits (essentially, the count of ones).

For instance, in a binary string `1101`, the sum of its bits is 3, as there are three `1`s. The goal is to maximize this sum.

2.2. Defining Fitness
The fitness function, $\varphi(x)$, crucial for evaluating potential solutions, is defined as:

$$\varphi(x) = \sum_{i=1}^{n} x_i$$

This function tallies the `1`s in the binary string. Achieving a fitness score equal to n signifies an optimal solution, where every bit is `1`. The choice of this fitness function is intuitive, as it directly quantifies the objective of the OMP, making it an ideal measure for optimization.

2.3. The Significance of Unimodality

OMP is characterized as a unimodal problem, which means it contains a singular peak or optimal solution in its landscape - all ones in the binary string. This feature simplifies the search process since any improvement in fitness unequivocally moves a solution closer to the global optimum. However, it's this simplicity that also makes OMP an intriguing test case, contrasting with multimodal problems that contain numerous local optima, complicating the path to the global maximum.

2.4. Navigating the Solution Space

Given a string length n, the solution space, comprising 2^n potential strings, expands exponentially. This vastness renders exhaustive search impractical for large n. Efficient optimization algorithms, such as genetic algorithms, employ mechanisms like crossover and mutation - inspired by biological evolution - to explore this space creatively and efficiently, avoiding the computational cost of evaluating every possible solution.

2.5. Broader Implications

While OMP serves primarily as a theoretical benchmark, the strategies and insights derived from solving it are applicable to more complex, real-world problems. For instance, the principles of incremental improvement and exploration versus exploitation, critical in solving OMP, are equally relevant in optimizing network configurations, financial portfolios, and many other domains where optimal solutions are sought within immense search spaces.

2.6. Representation and evaluation

In the context of Keen, the representation of candidate solutions is fundamental. These solutions are modeled as collections of "genetic material," mirroring the concept of genes and chromosomes in biological systems. This genetic material, depending on its organization and granularity, can represent different dimensions of solutions:

Genetic material	**Mathematical construct**
Gene	Scalar
Chromosome	Vector
Genotype	Matrix

Table 1: Genetic material and their mathematical equivalent

Given the binary nature of the problem domain in Keen, solutions can be effectively represented as arrays of boolean values, encapsulated by the `BooleanGene` and `BooleanChromosome` classes.

Consider the following Kotlin code snippet for defining a genotype:

```kotlin
// Define the size of each chromosome, here set to 50 genes.
private const val CHROMOSOME_SIZE = 50
// Initial probability for each gene to be `true`, set at 15%.
private const val TRUE_RATE = 0.15

// Constructing a genotype with specified characteristics.
val gt = genotypeOf {
  chromosomeOf {
    booleans {
      size = CHROMOSOME_SIZE  // Number of genes in a chromosome.
      trueRate = TRUE_RATE    // Probability of a gene being `true`.
    }
  }
}
```

This code defines a genotype with chromosomes consisting of 50 genes each, where each gene has a 15% chance of being `true`. This setup is particularly useful in problems where the solution space can be binary encoded.

Evaluating the fitness of these genotypes is crucial for guiding the genetic algorithm towards optimal solutions. The fitness function, in this case, is designed to count the number of `TrueGenes` within a genotype:

```
// Function to evaluate the fitness of a genotype.
private fun count(genotype: Genotype<Boolean, BooleanGene>) =
  genotype.flatten()  // Convert the genotype to a list of genes.
          .count { it }  // Count the number of `TrueGenes`.
          .toDouble()  // Convert the count to a Double for compatibility.
```

In this function, the genotype is first flattened to transform its structured genetic material into a linear list of genes, making it easier to apply operations like counting. The number of `TrueGenes` reflects the fitness of the genotype, with higher counts indicating potentially more optimal solutions.

By meticulously crafting the representation of solutions and defining a meaningful fitness function, Keen leverages the principles of genetic algorithms to efficiently navigate the solution space of complex optimization problems.

2.7. Initialization

The initialization phase marks the commencement of a genetic algorithm's evolutionary journey. This crucial stage involves creating the initial population of individuals that will explore the solution space. The individuals in this population represent potential solutions to the problem at hand.

In a genetic algorithm, the population is akin to a diverse ecosystem of solutions. The size of this population and the method of its initialization are pivotal decisions. While randomness is a common approach, allowing for a broad exploration of the solution space, informed initialization can be employed when prior problem knowledge is available, potentially leading to a more directed search.

Once the population is initialized, each individual is evaluated to determine its fitness value. This initial assessment is essential as it provides a snapshot of the population's diversity and the quality of solutions, guiding the subsequent evolutionary steps.

In problems like the OMP, where specific insights into the optimal solution are not available, the initialization typically involves generating random binary strings for each individual in the population. For instance, with a population of 100 individuals and a chromosome length of 50, each individual would start with a randomly generated 50-bit string.

To manage this process, the Evolution Engine comes into play. This component of the genetic algorithm framework orchestrates the initialization, evaluation, and evolution of the population. Configuring the Evolution Engine involves specifying parameters such as population size and the structure of individuals, as demonstrated in the following Kotlin code snippet:

```
// Define the population size.
private const val POPULATION_SIZE = 100
```

```kotlin
// Configure and instantiate the Evolution Engine.
val engine = evolutionEngine(::count, genotypeOf {
    chromosomeOf {
        booleans {
            size = CHROMOSOME_SIZE  // Length of each individual's chromosome.
            trueRate = TRUE_RATE    // Initial probability of a gene being `true`.
        }
    }
}) {
    populationSize = POPULATION_SIZE  // Set the population size.
    // Additional configurations can be added here.
}
```

In this snippet, `evolutionEngine` is configured with a fitness evaluation function (`::count`), the structure of individuals (`genotypeOf` block), and the size of the initial population. This setup forms the foundation from which the genetic algorithm embarks on its search for optimal solutions, evolving the population over successive generations.

2.8. Selection

Following the initialization phase, the GA enters its main evolutionary cycle, with selection being a pivotal process. This step mimics natural selection by preferentially choosing individuals with higher fitness for reproduction, thus steering the population towards more optimal solutions.

In this context, the concept of elitism is introduced through a survival rate σ, which determines the fraction of the population that advances to the next generation unchanged. Specifically, the top $\lfloor \sigma N \rfloor$ individuals, based on their fitness, are preserved, while the rest are replaced by offspring generated through genetic operators. This blend of elitism and generation of new individuals helps balance exploration and exploitation in the search space.

Definition 2.8.1 (Selection operator): The selection process is formalized through a selection operator, denoted as Σ, which is defined for a population P comprising N individuals, each with a fitness value φ_i. The operator is described as:

$$\Sigma(P : \mathbb{P}, n : \mathbb{N}, ...) \to P'$$

where $\mathbb{P}$ is the set of all possible populations, $\mathbb{N}$ represents the set of natural numbers, P' is the selected subset of the population, and n is the number of selections to be made.

A commonly used method within this operator is the **roulette wheel selection**, where each individual's chance of being selected is proportional to its fitness. This can be mathematically expressed as:

$$\rho_{\Sigma(i)} = \varphi_i \sum_{[j=1]}^{N} \varphi_j$$

where $\rho_{\Sigma(i)}$ represents the selection probability of the i-th individual.

Configuring the selection mechanism within a GA is typically straightforward, as demonstrated in the following Kotlin snippet for the Keen library:

```kotlin
val engine = evolutionEngine(::count, genotypeOf {
    chromosomeOf {
        booleans {
            size = CHROMOSOME_SIZE
            trueRate = TRUE_RATE
        }
    }
}) {
    // For selecting parents for crossover.
    parentSelector = RouletteWheelSelector()
    // For selecting individuals to survive to the next generation.
    survivorSelector = TournamentSelector()
    /* Additional configurations */
}
```

This configuration illustrates the use of `RouletteWheelSelector` for parent selection, where probabilities are aligned with individuals' fitness, and `TournamentSelector` for survivor selection, which involves selecting the best among a randomly chosen subset of individuals. The flexibility to use different selectors for these phases allows for a tailored approach, potentially enhancing the GA's ability to converge on optimal solutions.

2.9. Variation

Variation is the cornerstone of GA, facilitating the creation of new individuals from existing ones to explore the solution space comprehensively. This process is essential to circumvent premature convergence to sub-optimal solutions, analogous to how genetic diversity in nature fosters adaptability and resilience in species.

The primary mechanisms of variation in GAs are crossover and mutation. **Crossover** resembles biological recombination, merging genetic information from two or more parents to produce offspring. **Mutation**, akin to spontaneous genetic mutations in nature, introduces random alterations to an individual's genetic makeup.

To formally define a variation operator, which is pivotal in generating new individuals within a population, consider the following:

Definition 2.9.1 (Variation operator): A variation operator is a mechanism that derives new individuals from existing ones in a population. Formally, it can be represented as a function:

$$\phi : \left(P : \mathbb{P}, \rho_\phi : \mathbb{R}, \ldots\right) \to \mathbb{P}$$

where:

- $\mathbb{P}$ represents the set of all possible populations.
- $\mathbb{R}$ denotes the set of real numbers, corresponding to the range of the probability parameter.
- P specifies the particular population subject to variation.
- ρ_ϕ is the probability of applying the variation operator to an individual within P.

The ellipsis (…) signifies additional parameters that may be included based on the specific implementation and characteristics of the variation operator.

Variation operators in genetic algorithms are typically variadic, capable of accepting a variable number of parent individuals to produce offspring. This adaptability enables a diverse array of genetic combinations within the population, encouraging a thorough exploration of potential solutions.

2.9.1. Crossover

In GA, a prominent variation operator is the *crossover*. This operator mirrors the genetic recombination seen in nature. It facilitates the exchange of genetic material between two individuals, spawning a new generation.

Definition 2.9.1.1 (Crossover operator): The crossover operator recombines genetic material from existing individuals to create new ones. Formally, it is represented as:

$$X(P : \mathbb{P}, \rho_X : \mathbb{R}, \ldots) \to \mathbb{P}$$

with the following parameters:

- $\mathbb{P}$ – the set of all possible populations,
- $\mathbb{R}$ – the set of real numbers,
- P – the population under variation,
- ρ_X – probability of applying the operator to an individual.

In our example, we will employ a emph{single-point crossover} operator. This operator picks a random index in the parent chromosome and swaps the genes from each section (before and after the cut point).

Take, for instance, two parent individuals, $I_1 = 1100$ and $I_2 = 0001$. Utilizing the *single-point crossover* operator, we could select the first half of the genes: 11 from I_1 and 00 from I_2. This exchange would produce the offspring $O_1 = 1101$ and $O_2 = 0000$

Implementing this in *Keen* is simple, since it provides a wide array of crossover operators.

```kotlin
val engine = evolutionEngine(::count, genotypeOf {
    chromosomeOf {
        booleans {
            size = CHROMOSOME_SIZE
            trueRate = TRUE_RATE
        }
    }
}) {
    alterers += SinglePointCrossover(chromosomeRate = 0.6))
    /* Other configurations */
}
```

It is important to note the use of the += operator instead of =, this is because, to allow a more flexible syntax, the alterers are initialized as a read-only empty mutable list.