UNIVERSITY OF PISA

School of Engineering

Master of Science in Computer Engineering

Distributed Systems and Middleware Technologies

PROJECT DOCUMENTATION

JENOMA: A JAVA LIBRARY FOR THE DISTRIBUTED EXECUTION OF GENETIC ALGORITHMS ON AN ERLANG MANAGED CLUSTER

WORKGROUP:

Diego Casu

Iacopo Pacini

ACADEMIC YEAR 2020/2021

Contents

1. Introduction

Genetic algorithms are search and optimisation methods that mimic the natural selection process of biological evolution to solve both constrained and unconstrained optimisation problems. Based on operators such as selection, crossover and mutation, they perform a systematic random search in the global space of the solutions in order to increase the likelihood of finding global optima.

A candidate solution in the genetic domain is represented by an individual, whose properties are codified in terms of a chromosome, namely a set of domain specific attributes called genes; each individual is associated to a fitness score expressing the goodness of the solution. A population of individuals is evolved across generations to improve its fitness and generate better solutions for the problem.

The fundamental aspect when designing a genetic algorithm is to provide a balance between the exploration and exploitation of the the search space. Exploration and exploitation correspond to global and local search, respectively: an unbalanced mix could lead to a premature convergence to an unsatisfying local optimum (exploitation) or to a very slow search process (exploration). This balance is achieved choosing accurately the operators that drive the evolutionary process, but also employing the right population size to increase the likelihood of finding good solutions and to keep a significant population variability. In case of problems with high dimensional solutions, a large population size could be needed, posing a problem in terms of required computational power.

The objective of this project is to design and implement JEnoma, a Java library for the distributed execution of genetic algorithms on a set of remote machines, exploiting Erlang to manage the cluster and the communication between nodes. JEnoma offers the possibility to distribute a large population over multiple machines, balancing the workload and operating in parallel on subsets of individuals, possibly exploiting multi-threading; moreover, it supports a wide variety of genetic strategies, leaving to the user the choice of either using the provided implementations of the most common operators or defining her custom ones.

The document is organised as follows: Section 2 discusses the design of the distributed solution, with particular attention to the management of the cluster and the execution flow of the genetic algorithm; Section 3 contains practical references about the structure and use of the Java library; Section 4 reports the testing results obtained solving the travelling salesman and knapsack problems.

2. Design

2.1 Cluster topology

JEnoma requires a cluster of $N + 1$ communicating machines, composed of one coordinator and N workers. At run-time, the library organises them into a star topology, as shown in Figure 1.

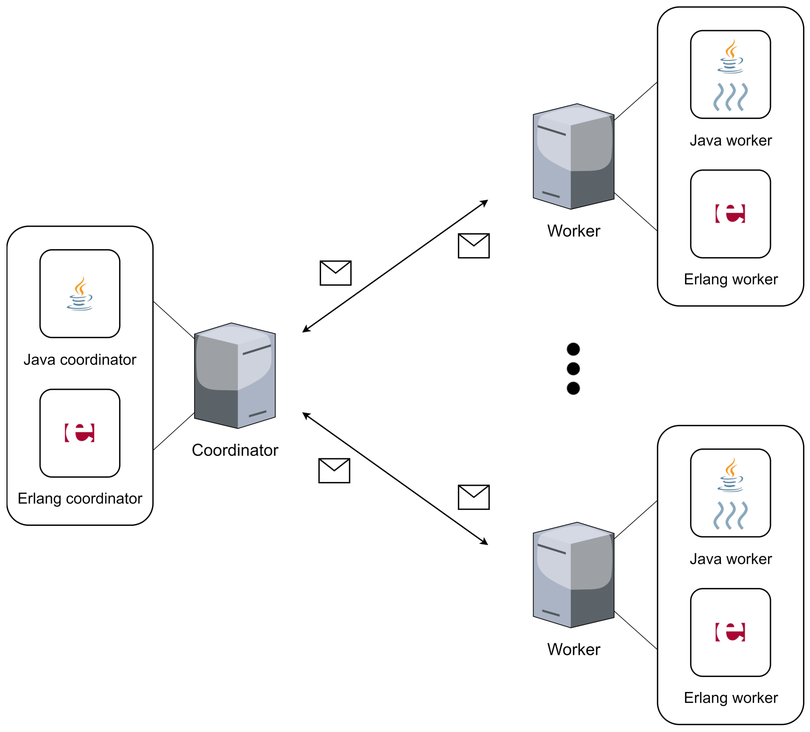


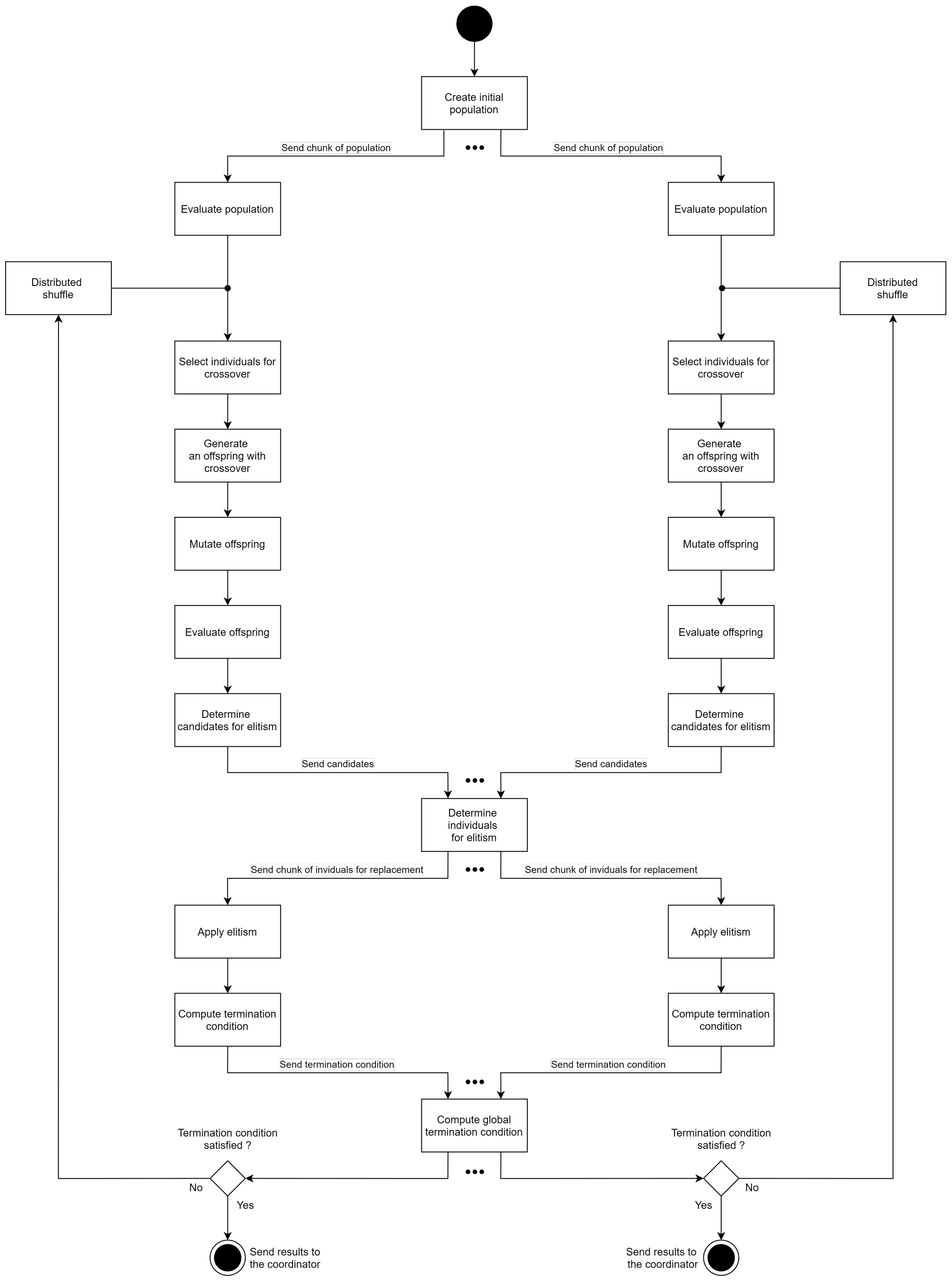
Figure 1: Cluster organisation in JEnoma.

The coordinator is responsible for receiving the user-defined genetic algorithm, along with the initial population, and for distributing the workload evenly among the workers, so that each one can operate on an independent portion of the population; moreover, it orchestrates the communication whenever collective tasks must be performed. A worker is responsible for executing the actual genetic algorithm on the assigned population, possibly exploiting multi-threading, and for contacting the coordinator to report about the progression of the computation.

Each machine, independently of its role, executes in Java its computations, while it delegates to an Erlang component the management of the communication: the latter implements the distributed protocols and hides the complexity, composition and heterogeneity of the cluster to the Java component by making available the final results of collective tasks. Moreover, it monitors the status of the cluster: if a failure of at least one component is detected by a timeout expiration, all the machines are stopped and their resources deallocated.

2.2 A genetic algorithm in JEnoma

2.2.1 Overview



Create initial population
Send chunk of population
Send chunk of population
Evaluate population
Evaluate population
Distributed shuffle
Distributed shuffle
Select individuals for crossover
Select individuals for crossover
Generate an offspring with crossover
Generate an offspring with crossover
Mutate offspring
Mutate offspring
Evaluate offspring
Evaluate offspring
Determine candidates for elitism
Determine candidates for elitism
Send candidates
Send candidates
Determine individuals for elitism
Send chunk of inviduals for replacement
Send chunk of inviduals for replacement
Apply elitism
Apply elitism
Compute termination condition
Compute termination condition
Send termination condition
Send termination condition
Compute global termination condition
Termination condition satisfied ?
Termination condition satisfied ?
No
Yes
No
Yes
Send results to the coordinator
Send results to the coordinator

Figure 2: Distributed execution of a genetic algorithm in JEnoma. The operations placed at the sides are done by workers, while the central flow describes the coordinator.

JEnoma supports the execution of coarse-grained parallel genetic algorithms, i.e. algorithms that assume the presence of a population on each computing node, with migration of individuals at each iteration. The library supports a wide variety of genetic strategies, since it offers the possibility to define:

- custom-encoded chromosomes, exploiting generic objects as genes;
- custom genetic operators for the evaluation, selection, crossover and mutation stages;
- the presence and level of elitism during generations, enabling strictly or overlapping generational procedures;
- custom termination conditions, whose evaluation is done following a map-reduce approach.

The only enforced constraint is that, at each generation, the size of the population cannot grow or shrink: each offspring is as numerous as the previous generation, meaning that the final population will have the same size of the original one.

The execution flow is depicted in Figure 2: once the initial population has been created by the user, the coordinator splits it in balanced portions, each one assigned to a different worker. A worker evaluates its initial subset of the population, assigning to each individual a fitness value, and iteratively applies the genetic operators until the termination condition is satisfied; across the generations, the elitism — if enabled — and the shuffle are performed. At the end, the scattered chunks of the global population are sent to the coordinator and merged.

Whenever randomised operations are required, pseudorandom number generators based on the Mersenne Twister algorithm are used. To make the results of a computation reproducible, a base seed can be configured: even if the latter is a single seed, the library ensures that different machines — and different threads in the same machine — use different PRNGs instances with distinct seeds derived from the base seed. The derivation process starts with each machine generating in isolation a private seed by hashing the base seed and the worker name; then, the k-th thread of a node uses as seed the private one incremented by k.

In the following, the design of the stages is discussed, along with their eventual multi-thread support.

2.2.2 Evaluation, selection, crossover and mutation

The evaluation, selection, crossover and mutation stages are executed in isolation by each worker on their portion of population, exploiting the relative user-defined operator. With the exception of the selection stage, they exploit Java multi-threading, spawning a number of threads equal to the number of logical cores available in the machine. More in details:

- the evaluation and mutation stages are multi-threaded, with each thread operating on an independent subset of the population. Since multiple threads do not share individuals, no particular synchronisation pattern is adopted;
- the selection stage is single-threaded and involves the entire population of the worker at the same time, as a user would expect; this means that the population returned by the custom selector operator is effectively

the overall mating pool for the specific node. In any case, the user is free to define its multi-threaded selection strategy inside the operator, dealing with eventual synchronisation issues;

- the crossover stage exploits multi-threading, so that each thread randomly chooses two distinct individuals from the mating pool and attempts to cross them. Since the crossover should not modify the parents, but only produce new children, the concurrent access of the same individuals by different threads at the same time is considered safe and no particular synchronisation pattern is adopted.

2.2.3 Elitism

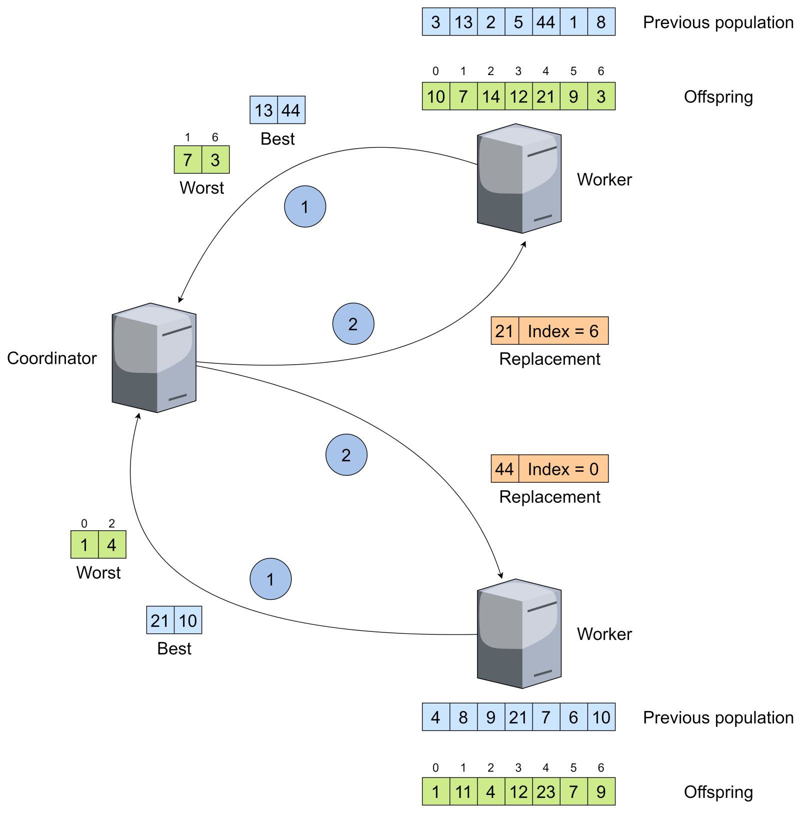


Figure 3: Elitism stage with $E = 2$ individuals involved.

The presence of elitism in an algorithm allows to implement overlapping generational procedures: a percentage of the fittest individuals of each generation is preserved by replacing the worst individuals of an offspring. In a distributed context, each node holds a portion of the overall population, thus an exchanging protocol is required to find the global best/worst individuals and to avoid their duplication during local replacements.

Called E the number of individuals involved in the elitism, each worker sorts its population by fitness and sends its E best and worst individuals to the coordinator; the worst ones are sent along with an index that identifies their placement in the local population. It should be noted that a worker cannot send less than E individuals, since it could happen that the global elite and/or the global worst set are held by a single node.

The coordinator receives $N \cdot E$ individuals, which are sorted again by fitness to extract the global best and worst ones. Then, the most unsatisfying worst individual is scheduled to be replaced by the best elite individual, the second most unsatisfying one by the second best elite one and so on. To let the destination worker replace correctly the individual, the replacement is paired with the index sent in the first phase.

To make the result of the elitism phase reproducible over multiple runs with the same seed, an individual is tagged with the sender ID, which is exploited when the coordinator sorts the individuals by fitness to deterministically order individuals with the same score. This ensures that different arrival orders caused by distinct interleavings of send operations originating at separate workers will always result in the same ordering of individuals at the coordinator.

2.2.4 Termination condition

In a genetic algorithm, the termination condition can be checked against:

- the number of generations elapsed. In this case, a maximum number of iterations is set;
- the population, generating performance indexes from the fitness of the individuals. For example, the algorithm could stop when a desired fitness is achieved or when the best fitness does not improve for a certain number of generations.

To avoid checking against the entire population at the coordinator and in a centralised way, the library offers the possibility to derive the termination condition with a map-reduce approach. In particular, the N workers act as mappers and the coordinator as the single reducer: given the number of elapsed generations and a population chunk, each worker computes a partial condition in isolation and sends it to the coordinator; the latter decides if the algorithm must continue or stop, according to the list of partial conditions. A condition can be any user-defined object, meaning that the only limitation of this approach is the one of having a termination check compatible with the map/reduce pattern.

2.2.5 Shuffle

The shuffle stage is fundamental to enable recombination between individuals of populations residing on different nodes. Differently from the other stages, the shuffle is completely distributed and does not require the participation of the coordinator: this choice was made to avoid costly centralised operations on the entire population at each iteration. The distribution of individuals is done in isolation by each node following a round-robin strategy: supposing the presence of N workers and a zero-based indexing, the k-th individual of a population is sent to the $|k|_N$-th worker. It should be noted that the sending worker is included itself in the list of possible destinations, meaning that a fraction of the population will not change node, and that this strategy preserves the population balancing on the machines.

To make the result of the shuffle phase reproducible over multiple runs with the same seed, an individual is tagged with the sender ID, which is exploited by each node to sort the population at the end of the shuffle. Given that each worker sends its individuals always in the same order (the sending process is sequential and the round robin strategy is fixed), different arrival orders caused by distinct interleavings of send operations originating at separate workers will always result in the same ordering of individuals at the receiver.

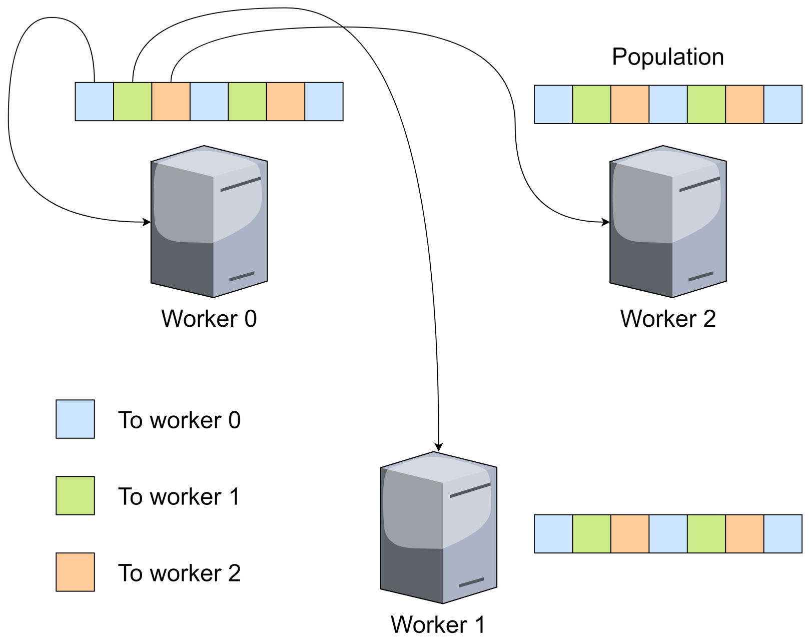


Figure 4: Distributed shuffle of individuals in the case of $N = 3$ workers.

3. User manual

3.1 Requirements

In order to execute a genetic algorithm in JEnoma, a cluster of $N + 1$ machines is needed, composed of one coordinator and N workers, with the constraint that a machine cannot be a coordinator and a worker at the same time. The coordinator is implicitly defined as the machine that starts the execution of the program: it is responsible for distributing the project files to the remote machines and starting the required processes, meaning that there is no need to manually pre-load files or start JEnoma processes on each machine of the cluster.

JEnoma supports clusters composed of machines with heterogeneous operating systems: it has been tested successfully on Windows 10 and Ubuntu 20.04. By default, JEnoma prepares two types of commands: the ones for Windows OSes and the ones for all the other OSes, supposed to be Linux distributions. In case of compatibility issues, custom modifications can be made to `ClusterUtils.java`, stored in the `it.unipi.jenoma.cluster` package.

Every machine, independently of its role, must have installed — with a version greater than or equal to the specified one — Java 16, Erlang 24, SSH and SCP; in addition, Maven 3.8.1 is required for the coordinator. The `PATH` environment variable must be updated so that the commands `java, jar, ssh, scp` and `mvn` can be issued directly from the shell. If the coordinator OS is a Linux distribution, the shell process must be invocable with the command `bash`.

3.2 Installation

3.2.1 Communication between machines

A sequence of steps are required to let the machines of the cluster communicate. More in details:

1. assign a symbolic name to each machine of the cluster. In the following, it is assumed that the workers are named *worker1,...,workerN* and the coordinator is named *coordinator*;

2. for each machine, add to `/etc/hosts` or to `C:\Windows\System32\drivers\etc\hosts` the entries mapping the symbolic names to the corresponding IP addresses. For example:

```
192.168.0.2   worker1
...
192.168.0.N+1 workerN
192.168.0.N+2 coordinator
```

3. create a user with the same name in each worker machine, for example named *jenoma*. The files transferred by the coordinator to a worker are saved in the home directory of this user, inside the `.jenoma` folder;

4. setup the coordinator so that it can access via SSH/SCP each worker, using SSH keys and authenticating as the previously created user. For each worker, a couple of private/public keys must be created, with the public key that must be stored inside the `authorized_keys` file of the worker itself. All the private keys must be stored in the coordinator inside a single folder of choice, where each key file must be named as the corresponding machine's symbolic name, without file extensions. For instance, the file holding the private key of *worker1* must be named exactly `worker1`;

5. test the SSH setup before running a program or an example, being sure that the workers are correctly added inside the `known_hosts` file of the coordinator.

3.2.2 Configuration file

Once the machines have been setup for communication, a JSON configuration file for the coordinator must be created. Such file must contain the following fields:

- `jarpath`, specifying the path of the fat JAR containing both the JEnoma library and the user code. More details about the build process and the output JAR can be found in the Execution section;
- `sshKeyFolder`, specifying the path of the folder containing the private SSH keys used to access the workers;
- `sshUser`, specifying the user created in the workers for the SSH authentication;
- `coordinator`, specifying the symbolic name of the coordinator machine;
- `workers`, containing a list of the symbolic names of the workers;
- `timeoutSetupCluster`, specifying the timeout for the cluster setup. The timeout is in milliseconds and should be tuned according to the time needed for the coordinator to transfer the files to all the remote machines, start their required processes and receive their heartbeats;
- `timeoutWorker`, specifying the timeout for a worker reply during the execution of the algorithm. The timeout is in milliseconds and should be tuned according to the time needed by a single worker to perform an iteration of the algorithm;
- `seed`, specifying an integer base seed used in the algorithm to instantiate the pseudorandom number generators. If the symbolic names of the machines and the base seed are the same, multiple executions of the algorithm produce the same results. More details about the seed derivation process can be found in the Overview section.

An example of configuration file is the following:

```json
{
  "jarPath": "./target/jenoma-1.0-jar-with-dependencies.jar",
  "sshKeyFolder": "your_path/your_SSH_key_folder",
  "sshUser": "jenoma",
  "coordinator": "coordinator",
  "workers": [
    "worker1",
    ...
    "workerN"
  ],
  "timeoutSetupCluster": 30000,
  "timeoutWorker": 100000,
  "seed": 1
}
```

3.3 Writing an algorithm

The following sections clarify how the code to solve a problem should be organised. User code must be placed inside a dedicated package in the project, so that it can be correctly bundled with the library files in the same output JAR: having a single archive to transfer to remote machines is fundamental to ensure the correct initialisation of the cluster.

3.3.1 Loading the configuration file

As explained in the Configuration file section, a JSON file specifying the information about the cluster and the algorithm must be provided. To retrieve its representation in Java, the `Configuration` class can be used, by simply passing the file path at construction time.

```java
public class HelloJEnoma {
    public static void main(String[] args) throws IOException {
        Configuration configuration = new Configuration("configuration.json");
        System.out.println(configuration);
    }
}
```

Any field of the configuration can be retrieved using the relative Getter method.

3.3.2 Chromosomes, individuals and populations

The definition of candidate solutions for the problem can be done exploiting the classes defined in the `it.unipi.jenoma.population` package. In particular, `Chromosome<T>` allows to define custom encoded chromosomes, where each gene is an instance of `T` and with the only constraint of `T` implementing `Serializable`. The `Individual` and `Population` classes are convenience wrappers: the first represents a candidate solution in terms of a chromosome and a fitness value, the latter a list of candidate solutions.

In theory, an `Individual` is not parameterised and can hold a generic implementation of `Chromosome`, leading to the possibility of heterogeneous populations composed of differently encoded chromosomes. If such a setting is required, it is up to the user to manage correctly the operations involving assorted candidate solutions and to define the appropriate operators.

A trivial initialisation of a population can be done as follows:

```java
public class HelloJEnoma {
    public static void main(String[] args) throws IOException {
        int populationSize = 1000;
        int chromosomeSize = 10;
        int geneUpperBound = 50; // Exclusive bound.

        Configuration configuration = new Configuration("configuration.json");
        PRNG prng = new PRNG(configuration.getSeed());
        Population initialPopulation = new Population(new ArrayList<>());

        for (int i = 0; i < populationSize; i++) {
            List<Integer> genes = new ArrayList<>();

            for (int j = 0; j < chromosomeSize; j++)
                genes.add(prng.nextInt(geneUpperBound));  // Next int in [0, geneUpperBound).

            Chromosome<Integer> chromosome = new Chromosome<>(genes);
            initialPopulation.addIndividual(new Individual(chromosome));
        }

        System.out.println(initialPopulation);
    }
}
```

3.3.3 Genetic operators

Custom genetic operators can be defined implementing the interfaces defined in the `it.unipi.jenoma.operator` package, hence `Evaluation`, `Selection`, `Crossover`, `Mutation` and `TerminationCondition<T>`. A custom operator must be designed according to the computational flow presented in the Overview section; moreover, the following considerations must be taken into account:

- a `Selection` operator and the `map`/`end` methods of `TerminationCondition` are the only operators executed in a single thread. Custom multi-threaded implementations are possible, with thread-safety that must be ensured by the user;
- an operator must not modify the passed individual/population, except for the case of mutation stages;
- the object representing an operator is always the same across the generations, meaning that some state can be stored and updated, for instance to enable adaptive genetic algorithms. The manipulation of such fields must be thread-safe, if the operator is used in a multi-threaded stage. It should be noted that different workers have distinct instances of the same operator.

Possible implementations of common operators are available in the `it.unipi.jenoma.operator.common` package.

For what concerns elitism, there is no interface to implement: the user can only instantiate an `Elitism` object, passing the number of involved individuals. If the latter is 0, the elitism stage is disabled, reducing the time spent in communication during the iterations.

3.3.4 Starting the computation

Executing the algorithm on the cluster is simple and straightforward: given a population and the operators, a `Coordinator` must be instantiated and started. A `Coordinator` object accepts `GeneticAlgorithm` instances, which wrap together individuals and operators. The `start()` method launches the initialisation of the cluster and the computation: it is a blocking method that returns the final population only when the algorithm ends. However, a `stop()` method to force the computation termination and the deallocation of the resources is available, in case the user wants to call `start()` in a separate thread and still have the possibility to interrupt the coordinator.

```java
public class HelloJEnoma {
    public static void main(String[] args) throws IOException {
        ...

        Evaluation evaluation = (individual, logger) -> {...};
        Crossover crossover = new UniformCrossover(0.5);
        Mutation mutation = (individual, prng, logger) -> {...};
        Selection selection = new RouletteWheelSelection(10);
        Elitism elitism = new Elitism(5);
        TerminationCondition<Boolean> terminationCondition = new NGenerationsElapsed(50);

        Coordinator coordinator = new Coordinator(new GeneticAlgorithm(configuration,
                                                                initialPopulation,
                                                                evaluation,
                                                                selection,
                                                                crossover,
                                                                mutation,
                                                                terminationCondition,
                                                                elitism));
        Population finalPopulation = coordinator.start();
        System.out.println(finalPopulation);
    }
}
```

3.4 Execution

The project can be built running either `build.sh` or `build.bat`, depending of the current OS. The output of the build process is saved in the `target` folder and consists mainly in two JAR files: `jenoma-1.0` and `jenoma-1.0-jar-with-dependencies`. The latter is a fat JAR packaging the code and all the dependencies of JEnoma; its path is the one to be included in the configuration file.

The starting point of execution is the coordinator machine: to run a user-defined algorithm or an available example, the command `java -cp jarPath yourpackage.yourclass` can be issued. For instance, supposing to leave the JAR file into the `target` folder, to have a configuration file named `configuration.json` and to run the command from the main directory of the project, the available examples can be started with the commands:

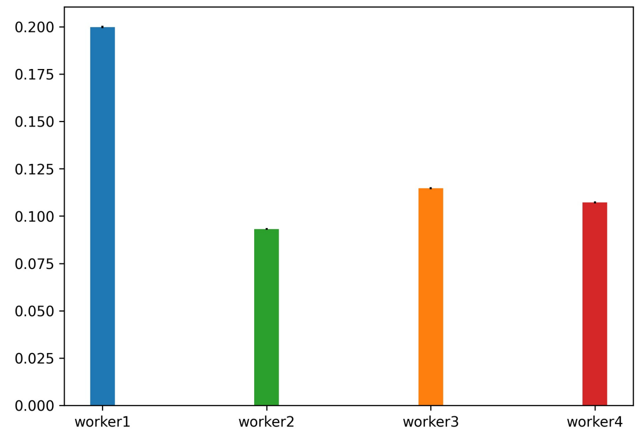
```
> java -cp target/jenoma-1.0-jar-with-dependencies.jar
    it.unipi.jenoma.example.TSP configuration.json
> java -cp target/jenoma-1.0-jar-with-dependencies.jar
    it.unipi.jenoma.example.KnapsackProblem configuration.json
```

4. Testing

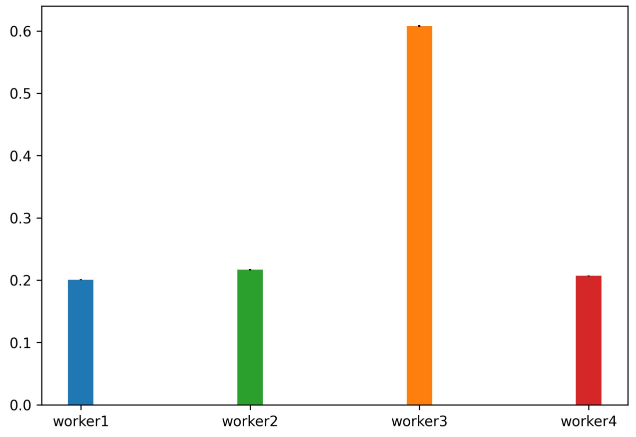
The library has been tested executing the travelling salesman and knapsack examples on a cluster of five virtual machines, running Ubuntu 20.04 and hosted on Microsoft Azure. More in details, the cluster was composed of:

- four workers, divided into two B4ms Standard instances (4 vCPUs, 16GB of RAM), one DS2 Standard v2 instance (2 vCPUs, 8 GB of RAM) and one DS3 Standard v2 instance (4 vCPUs, 16 GB of RAM);
- one coordinator, using a DS2 Standard v2 instance (2 vCPUs, 8 GB of RAM).

Both the problems were tested with a population of 500 000 individuals and for 100 generations.



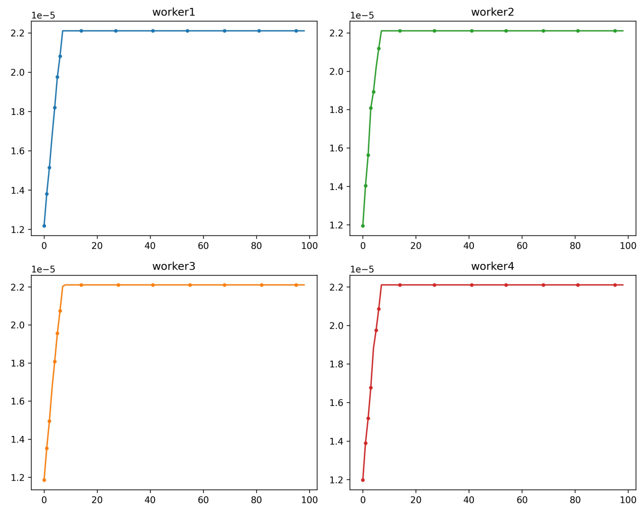
(a) Travelling salesman problem.



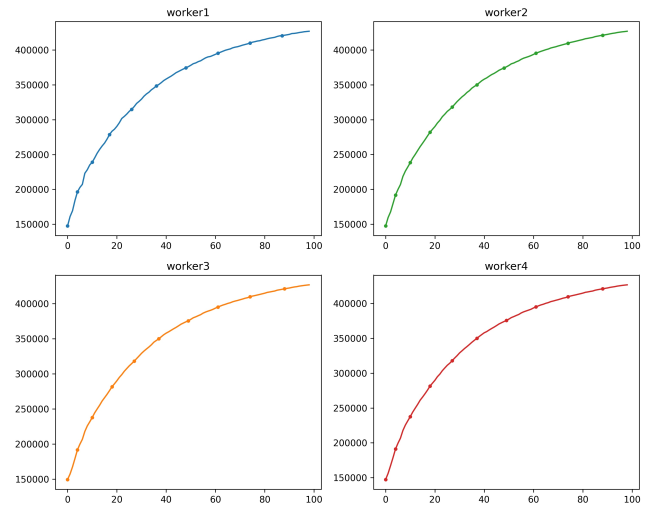
(b) Knapsack problem.

Figure 5: Average communication time - computation time ratio.

As the generations elapse, JEnoma produces better candidate solutions in terms of fitness, meaning that the implementation of the genetic execution flow is correct. Unfortunately, running such algorithms in a distributed environment seems to suffer of a non negligible overhead due to communication: the statistics show that, on average, the machines spent almost 80% of the time in communication. A more in depth analysis should be performed to understand if this overhead becomes acceptable as the population size grows, and to understand if the ratio improves exploiting a local cluster of machines, instead that a remote one whose physical organisation and placement is not manageable.



(a) Travelling salesman problem.



(b) Knapsack problem.

Figure 6: Fitness of the best individual across generations.